**LITERATURE**

To order Intel Literature or obtain literature pricing information in the U.S. and Canada call or write Intel Literature Sales. In Europe and other international locations, please contact your local sales office or distributor.

**INTEL LITERATURE SALES**
P.O. BOX 58130
SANTA CLARA, CA 95052-8130

In the U.S. and Canada
call toll free
(800) 548-4725

**CURRENT HANDBOOKS**

Product line handbooks contain data sheets, application notes, article reprints and other design information.

<table>
<thead>
<tr>
<th>TITLE</th>
<th>LITERATURE ORDER NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLETE SET OF HANDBOOKS&lt;br&gt;(Available in U.S. and Canada only)</td>
<td>231003</td>
</tr>
<tr>
<td>AUTOMOTIVE PRODUCTS HANDBOOK&lt;br&gt;(Not included in handbook set)</td>
<td>231792</td>
</tr>
<tr>
<td>COMPONENTS QUALITY / RELIABILITY HANDBOOK</td>
<td>210997</td>
</tr>
<tr>
<td>EMBEDDED CONTROL APPLICATIONS HANDBOOK</td>
<td>270648</td>
</tr>
<tr>
<td>8-BIT EMBEDDED CONTROLLER HANDBOOK</td>
<td>270645</td>
</tr>
<tr>
<td>16-BIT EMBEDDED CONTROLLER HANDBOOK</td>
<td>270646</td>
</tr>
<tr>
<td>32-BIT EMBEDDED CONTROLLER HANDBOOK</td>
<td>270647</td>
</tr>
<tr>
<td>MEMORY COMPONENTS HANDBOOK</td>
<td>210830</td>
</tr>
<tr>
<td>MICROCOMMUNICATIONS HANDBOOK</td>
<td>231658</td>
</tr>
<tr>
<td>MICROCOMPUTER PROGRAMMABLE LOGIC HANDBOOK</td>
<td>296083</td>
</tr>
<tr>
<td>MICROPROCESSOR AND PERIPHERAL HANDBOOK&lt;br&gt;(2 volume set)</td>
<td>230843</td>
</tr>
<tr>
<td>MILITARY PRODUCTS HANDBOOK&lt;br&gt;(2 volume set. Not included in handbook set)</td>
<td>210461</td>
</tr>
<tr>
<td>OEM BOARDS AND SYSTEMS HANDBOOK</td>
<td>280407</td>
</tr>
<tr>
<td>PRODUCT GUIDE&lt;br&gt;(Overview of Intel's complete product lines)</td>
<td>210846</td>
</tr>
<tr>
<td>SYSTEMS QUALITY / RELIABILITY HANDBOOK</td>
<td>231762</td>
</tr>
<tr>
<td>INTEL PACKAGING OUTLINES AND DIMENSIONS&lt;br&gt;(Packaging types, number of leads, etc.)</td>
<td>231369</td>
</tr>
<tr>
<td>LITERATURE PRICE LIST (U.S. and Canada)&lt;br&gt;(Comprehensive list of current Intel Literature)</td>
<td>210620</td>
</tr>
<tr>
<td>INTERNATIONAL LITERATURE GUIDE</td>
<td>E00029</td>
</tr>
</tbody>
</table>

**About Our Cover:**

Being the leader in 32-bit embedded controller architecture means providing the most versatile, the most reliable high performance family of products. With our support of such applications as robotics and laser printers, we plan on remaining an innovative leader in the embedded control market.
Intel the Microcomputer Company:

When Intel invented the microprocessor in 1971, it created the era of microcomputers. Whether used as microcontrollers in automobiles or microwave ovens, or as personal computers or supercomputers, Intel's microcomputers have always offered leading-edge technology. In the second half of the 1980s, Intel architectures have held at least a 75% market share of microprocessors at 16 bits and above. Intel continues to strive for the highest standards in memory, microcomputer components, modules, and systems to give its customers the best possible competitive advantages.

32-BIT EMBEDDED CONTROLLER HANDBOOK

1989
Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel Products:

Above, BITBUS, COMMputer, CREDIT, Data Pipeline, ETOX, FASTPATH, Genius, i, i, ICE, iCEL, iCS, iDBP, iDIS, i2ICE, iLBX, i_m, iMDDX, iMMX, Inboard, Insite, Intel, int3al, Intel376, Intel386, Intel486, int3alBOS, Intel Certified, Intelevision, int3aligent Identifier, int3aligent Programming, Intellec, Intellink, iOSP, iPDS, iPSC, iRMK, iRMX, iSBC, iSBX, iSDM, iSX M, KEPROM, Library Manager, MAPNET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, ONCE, OpenNET, OTP, PC BUBBLE, Plug-A-Bubble, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, UPI, and VLSICEL, and the combination of ICE, iCS, iRMX, iSBC, iSBX, iSX M, MCS, or UPI and a numerical suffix, 4-SITE, 376, 386, 486.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

*MULTIBUS is a patented Intel bus.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 58130
Santa Clara, CA 95052-8130

© INTEL CORPORATION 1988
CUSTOMER SUPPORT

INTEL'S COMPLETE SUPPORT SOLUTION WORLDWIDE

Customer Support is Intel's complete support service that provides Intel customers with hardware support, software support, customer training, consulting services and network management services. For detailed information contact your local sales offices.

After a customer purchases any system hardware or software product, service and support become major factors in determining whether that product will continue to meet a customer's expectations. Such support requires an international support organization and a breadth of programs to meet a variety of customer needs. As you might expect, Intel's customer support is quite extensive. It can start with assistance during your development effort to network management. 100 Intel sales and service offices are located worldwide — in the U.S., Canada, Europe and the Far East. So wherever you’re using Intel technology, our professional staff is within close reach.

HARDWARE SUPPORT SERVICES

Intel's hardware maintenance service, starting with complete on-site installation will boost your productivity from the start and keep you running at maximum efficiency. Support for system or board level products can be tailored to match your needs, from complete on-site repair and maintenance support economical carry-in or mail-in factory service.

Intel can provide support service for not only Intel systems and emulators, but also support for equipment in your development lab or provide service on your product to your end-user/customer.

SOFTWARE SUPPORT SERVICES

Software products are supported by our Technical Information Phone Service (TIPS) that has a special toll free number to provide you with direct, ready information on known, documented problems and deficiencies, as well as work-arounds, patches and other solutions.

Intel's software support consists of two levels of contracts. Standard support includes TIPS (Technical Information Phone Service), updates and subscription service (product-specific troubleshooting guides and; COMMENTS Magazine). Basic support consists of updates and the subscription service. Contracts are sold in environments which represent product groupings (e.g., iRMX® environment).

CONSULTING SERVICES

Intel provides field system engineering consulting services for any phase of your development or application effort. You can use our system engineers in a variety of ways ranging from assistance in using a new product, developing an application, personalizing training and customizing an Intel product to providing technical and management consulting. Systems Engineers are well versed in technical areas such as microcommunications, real-time applications, embedded microcontrollers, and network services. You know your application needs; we know our products. Working together we can help you get a successful product to market in the least possible time.

CUSTOMER TRAINING

Intel offers a wide range of instructional programs covering various aspects of system design and implementation. In just three to ten days a limited number of individuals learn more in a single workshop than in weeks of self-study. For optimum convenience, workshops are scheduled regularly at Training Centers worldwide or we can take our workshops to you for on-site instruction. Covering a wide variety of topics, Intel's major course categories include: architecture and assembly language, programming and operating systems, BITBUS™ and LAN applications.

NETWORK MANAGEMENT SERVICES

Today's networking products are powerful and extremely flexible. The return they can provide on your investment via increased productivity and reduced costs can be very substantial.

Intel offers complete network support, from definition of your network's physical and functional design, to implementation, installation and maintenance. Whether installing your first network or adding to an existing one, Intel's Networking Specialists can optimize network performance for you.
Table of Contents

Chapter 1
   Introduction to the 80960KB ................................................................. 1-1

Chapter 2
   80960KB Hardware Reference .............................................................. 2-1

Chapter 3
   80960KB Programmer's Reference ....................................................... 3-1

Chapter 4
   DATA SHEETS
      80960KB Embedded 32-Bit Microprocessor ........................................ 4-1
      376™ High Performance 32-Bit Embedded Processor .......................... 4-35
      82370 Integrated System Peripheral ................................................. 4-126

Chapter 5
   DEVELOPMENT SUPPORT TOOLS
      80960 Development Tools Fact Sheet .............................................. 5-1
      ADA-960 Compiler Fact Sheet ......................................................... 5-4
      ICE-960 Fact Sheet ......................................................................... 5-8
Any of the following products may appear in this publication. If so, it must be noted that such products have counterparts manufactured by Intel Puerto Rico, Inc., Intel Puerto Rico II, Inc., and/or Intel Singapore, Ltd. The product codes/part numbers of these counterpart products are listed below next to the corresponding Intel Corporation product codes/part numbers.

<table>
<thead>
<tr>
<th>Intel Corporation Product Codes/Part Numbers</th>
<th>Intel Puerto Rico, Inc. Product Codes/Part Numbers</th>
<th>Intel Singapore, Ltd. Product Codes/Part Numbers</th>
<th>Intel Corporation Product Codes/Part Numbers</th>
<th>Intel Puerto Rico, Inc. Product Codes/Part Numbers</th>
<th>Intel Singapore, Ltd. Product Codes/Part Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>376SKIT</td>
<td>p376SKIT</td>
<td></td>
<td>KM2</td>
<td>pKM2</td>
<td></td>
</tr>
<tr>
<td>903</td>
<td>p903</td>
<td></td>
<td>KM4</td>
<td>pKM4</td>
<td></td>
</tr>
<tr>
<td>904</td>
<td>p904</td>
<td></td>
<td>KM5</td>
<td>pKM5</td>
<td></td>
</tr>
<tr>
<td>913</td>
<td>p913</td>
<td></td>
<td>KM6</td>
<td>pKM6</td>
<td></td>
</tr>
<tr>
<td>914</td>
<td>p914</td>
<td></td>
<td>KT60</td>
<td>pKT60</td>
<td></td>
</tr>
<tr>
<td>923</td>
<td>p923</td>
<td></td>
<td>KW140</td>
<td>pKW140</td>
<td></td>
</tr>
<tr>
<td>924</td>
<td>p924</td>
<td></td>
<td>KW40</td>
<td>pKW40</td>
<td></td>
</tr>
<tr>
<td>952</td>
<td>p952</td>
<td></td>
<td>KW80</td>
<td>pKW80</td>
<td></td>
</tr>
<tr>
<td>953</td>
<td>p953</td>
<td></td>
<td>M1</td>
<td>pM1</td>
<td></td>
</tr>
<tr>
<td>954</td>
<td>p954</td>
<td></td>
<td>M2</td>
<td>pM2</td>
<td></td>
</tr>
<tr>
<td>ADAICE</td>
<td>pADAICE</td>
<td></td>
<td>M4</td>
<td>pM4</td>
<td></td>
</tr>
<tr>
<td>B386M1</td>
<td>pB386M1</td>
<td></td>
<td>M8</td>
<td>pM8</td>
<td></td>
</tr>
<tr>
<td>B386M2</td>
<td>pB386M2</td>
<td></td>
<td>MDX3015</td>
<td>pMDX3015</td>
<td></td>
</tr>
<tr>
<td>B386M4</td>
<td>pB386M4</td>
<td></td>
<td>MDX3015</td>
<td>pMDX3015</td>
<td></td>
</tr>
<tr>
<td>B386M8</td>
<td>pB386M8</td>
<td></td>
<td>MDX3016</td>
<td>pMDX3016</td>
<td></td>
</tr>
<tr>
<td>C044KIT</td>
<td>pC044KIT</td>
<td></td>
<td>MDX3016</td>
<td>pMDX3016</td>
<td></td>
</tr>
<tr>
<td>C252KIT</td>
<td>pC252KIT</td>
<td></td>
<td>MDX3016</td>
<td>pMDX3016</td>
<td></td>
</tr>
<tr>
<td>C28</td>
<td>pC28</td>
<td></td>
<td>MDX457</td>
<td>pMDX457</td>
<td></td>
</tr>
<tr>
<td>C32</td>
<td>pC32</td>
<td></td>
<td>MDX457</td>
<td>pMDX457</td>
<td></td>
</tr>
<tr>
<td>C452KIT</td>
<td>pC452KIT</td>
<td></td>
<td>MDX457</td>
<td>pMDX457</td>
<td></td>
</tr>
<tr>
<td>D66ASM</td>
<td>pD66ASM</td>
<td></td>
<td>MDX458</td>
<td>pMDX458</td>
<td></td>
</tr>
<tr>
<td>D66C86</td>
<td>pD66C86</td>
<td></td>
<td>MDX458</td>
<td>pMDX458</td>
<td></td>
</tr>
<tr>
<td>D66EDI</td>
<td>pD66EDI</td>
<td></td>
<td>MSA96</td>
<td>pMSA96</td>
<td></td>
</tr>
<tr>
<td>DCM9111</td>
<td>pDCM9111</td>
<td></td>
<td>NLAN</td>
<td>pNLAN</td>
<td></td>
</tr>
<tr>
<td>DOSNET</td>
<td>pDOSNET</td>
<td></td>
<td>PCLINK</td>
<td>pPCLINK</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>pF1</td>
<td></td>
<td>PCX344A</td>
<td>pPCX344A</td>
<td></td>
</tr>
<tr>
<td>GUPILOGICIID</td>
<td>pGUPILOGICIID</td>
<td></td>
<td>R286ASM</td>
<td>pR286ASM</td>
<td></td>
</tr>
<tr>
<td>H4</td>
<td>pH4</td>
<td></td>
<td>R286EDI</td>
<td>pR286EDI</td>
<td></td>
</tr>
<tr>
<td>I044</td>
<td>pI044</td>
<td></td>
<td>R286PLM</td>
<td>pR286PLM</td>
<td></td>
</tr>
<tr>
<td>I252KIT</td>
<td>pI252KIT</td>
<td></td>
<td>R286SSC</td>
<td>pR286SSC</td>
<td></td>
</tr>
<tr>
<td>I452KIT</td>
<td>pI452KIT</td>
<td></td>
<td>R86FOR</td>
<td>pR86FOR</td>
<td></td>
</tr>
<tr>
<td>I86ASM</td>
<td>pI86ASM</td>
<td></td>
<td>RC64410</td>
<td>pRC64410</td>
<td></td>
</tr>
<tr>
<td>ICE386</td>
<td>pICE386</td>
<td></td>
<td>RCX920</td>
<td>pRCX920</td>
<td></td>
</tr>
<tr>
<td>II010</td>
<td>pII010</td>
<td></td>
<td>RMX286</td>
<td>pRMX286</td>
<td></td>
</tr>
<tr>
<td>II086</td>
<td>pII086</td>
<td></td>
<td>RMXNET</td>
<td>pRMXNET</td>
<td></td>
</tr>
<tr>
<td>II086</td>
<td>pII086</td>
<td></td>
<td>SA01</td>
<td>pSA01</td>
<td></td>
</tr>
<tr>
<td>III11</td>
<td>pIII11</td>
<td></td>
<td>SA06</td>
<td>pSA06</td>
<td></td>
</tr>
<tr>
<td>III11</td>
<td>pIII11</td>
<td></td>
<td>SBC010</td>
<td>pSBC010</td>
<td></td>
</tr>
<tr>
<td>III18</td>
<td>pIII18</td>
<td></td>
<td>SBC012</td>
<td>pSBC012</td>
<td></td>
</tr>
<tr>
<td>III18</td>
<td>pIII18</td>
<td></td>
<td>SBC020</td>
<td>pSBC020</td>
<td></td>
</tr>
<tr>
<td>III19</td>
<td>pIII19</td>
<td></td>
<td>SBC028</td>
<td>pSBC028</td>
<td></td>
</tr>
<tr>
<td>III21</td>
<td>pIII21</td>
<td></td>
<td>SBC040</td>
<td>pSBC040</td>
<td></td>
</tr>
<tr>
<td>III26</td>
<td>pIII26</td>
<td></td>
<td>SBC056</td>
<td>pSBC056</td>
<td></td>
</tr>
<tr>
<td>III26</td>
<td>pIII26</td>
<td></td>
<td>SBC108</td>
<td>pSBC108</td>
<td></td>
</tr>
<tr>
<td>III515</td>
<td>pIII515</td>
<td></td>
<td>SBC116</td>
<td>pSBC116</td>
<td></td>
</tr>
<tr>
<td>III520</td>
<td>pIII520</td>
<td></td>
<td>SBC18603</td>
<td>pSBC18603</td>
<td></td>
</tr>
<tr>
<td>III520</td>
<td>pIII520</td>
<td></td>
<td>SBC186640</td>
<td>pSBC186640</td>
<td></td>
</tr>
<tr>
<td>III531</td>
<td>pIII531</td>
<td></td>
<td>SBC18651</td>
<td>pSBC18651</td>
<td></td>
</tr>
<tr>
<td>III531</td>
<td>pIII531</td>
<td></td>
<td>SBC186530</td>
<td>pSBC186530</td>
<td></td>
</tr>
<tr>
<td>III532</td>
<td>pIII532</td>
<td></td>
<td>SBC18678</td>
<td>pSBC18678</td>
<td></td>
</tr>
<tr>
<td>III533</td>
<td>pIII533</td>
<td></td>
<td>SBC18848</td>
<td>pSBC18848</td>
<td></td>
</tr>
<tr>
<td>III621</td>
<td>pIII621</td>
<td></td>
<td>SBC18856</td>
<td>pSBC18856</td>
<td></td>
</tr>
<tr>
<td>III707</td>
<td>pIII707</td>
<td></td>
<td>SBC208</td>
<td>pSBC208</td>
<td></td>
</tr>
<tr>
<td>III707</td>
<td>pIII707</td>
<td></td>
<td>SBC214</td>
<td>pSBC214</td>
<td></td>
</tr>
<tr>
<td>III815</td>
<td>pIII815</td>
<td></td>
<td>SBC215</td>
<td>pSBC215</td>
<td></td>
</tr>
<tr>
<td>INAS961</td>
<td>pINAS961</td>
<td></td>
<td>SBC220</td>
<td>pSBC220</td>
<td></td>
</tr>
<tr>
<td>IPAT86</td>
<td>pIPAT86</td>
<td></td>
<td>SBC221</td>
<td>pSBC221</td>
<td></td>
</tr>
<tr>
<td>KAS</td>
<td>pKAS</td>
<td></td>
<td>SBC28610</td>
<td>pSBC28610</td>
<td></td>
</tr>
<tr>
<td>KC</td>
<td>pKC</td>
<td></td>
<td>SBC28610</td>
<td>pSBC28610</td>
<td></td>
</tr>
<tr>
<td>KH</td>
<td>pKH</td>
<td></td>
<td>SBC28612</td>
<td>pSBC28612</td>
<td></td>
</tr>
<tr>
<td>KM1</td>
<td>pKM1</td>
<td></td>
<td>SBC28614</td>
<td>pSBC28614</td>
<td></td>
</tr>
<tr>
<td>Intel Corporation Product Codes/Part Numbers</td>
<td>Intel Puerto Rico, Inc. Product Codes/Part Numbers</td>
<td>Intel Singapore, Ltd. Product Codes/Part Numbers</td>
<td>Intel Corporation Product Codes/Part Numbers</td>
<td>Intel Puerto Rico, Inc. Product Codes/Part Numbers</td>
<td>Intel Singapore, Ltd. Product Codes/Part Numbers</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------</td>
<td>---------------------------------------------</td>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>SBC28616 pSBC28616</td>
<td></td>
<td></td>
<td>SBCMEM310 pSBCMEM310</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC300 pSBC300</td>
<td></td>
<td></td>
<td>SBCM312 bSBCM312</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC301 pSBC301</td>
<td></td>
<td></td>
<td>SBCM320 pSBCM320</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC302 pSBC302</td>
<td></td>
<td></td>
<td>SBCM340 pSBCM340</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC304 pSBC304</td>
<td></td>
<td></td>
<td>SBE96 pSBE96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC307 pSBC307</td>
<td></td>
<td></td>
<td>SBX217 pSBX217</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC314 pSBC314</td>
<td></td>
<td></td>
<td>SBX218 pSBX218</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC314 pSBC314</td>
<td></td>
<td></td>
<td>SBX270 pSBX270</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC322 pSBC322</td>
<td></td>
<td></td>
<td>SBC311 pSBC311</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC324 pSBC324</td>
<td></td>
<td></td>
<td>SBC328 pSBC328</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC337 pSBC337</td>
<td></td>
<td></td>
<td>SBX331 pSBX331</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC341 pSBC341</td>
<td></td>
<td></td>
<td>SBX344 pSBX344</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC386 pSBC386 sSBC386</td>
<td></td>
<td></td>
<td>SBX350 pSBX350</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC386116 pSBC386116</td>
<td></td>
<td></td>
<td>SBX351 pSBX351</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC386210 pSBC386210</td>
<td></td>
<td></td>
<td>SBX354 pSBX354</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC386212 pSBC386212</td>
<td></td>
<td></td>
<td>SBX488 pSBX488</td>
<td></td>
<td>sSBX586</td>
</tr>
<tr>
<td>SBC38622 pSBC38622</td>
<td></td>
<td></td>
<td>SBC3010 pSCOM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC38624 pSBC38624</td>
<td></td>
<td></td>
<td>SBC3015 pSDK51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC38628 pSBC38628</td>
<td></td>
<td></td>
<td>SBC3020 pSDK85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC38631 pSBC38631</td>
<td></td>
<td></td>
<td>SBC3025 pSDK86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC38632 pSBC38632</td>
<td></td>
<td></td>
<td>SBC3030 pSDK31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC38634 pSBC38634</td>
<td></td>
<td></td>
<td>SBC3035 pSDK45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC38638 pSBC38638</td>
<td></td>
<td></td>
<td>SBC3040 pSDK50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC428 pSBC428 sSBC428</td>
<td></td>
<td></td>
<td>SBC3050 pSDK51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC464 pSBC464</td>
<td></td>
<td></td>
<td>SBC3055 pSDK85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC517 pSBC517</td>
<td></td>
<td></td>
<td>SBC3060 pSDK86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC519 pSBC519</td>
<td></td>
<td></td>
<td>SBC3065 pSDK31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC534 pSBC534</td>
<td></td>
<td></td>
<td>SBC3070 pSDK45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC548 pSBC548</td>
<td></td>
<td></td>
<td>SBC3080 pSDK50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC550 pTSBC550</td>
<td></td>
<td></td>
<td>SBC3090 pSDK85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC550 pSBC550</td>
<td></td>
<td></td>
<td>SBC3100 pSDK86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC552 pSBC552</td>
<td></td>
<td></td>
<td>SBC3110 pSDK31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC556 pSBC556</td>
<td></td>
<td></td>
<td>SBC3120 pSDK45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC569 pSBC569</td>
<td></td>
<td></td>
<td>SBC3130 pSDK50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC589 pSBC589</td>
<td></td>
<td></td>
<td>SBC3140 pSDK85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC604 pSBC604</td>
<td></td>
<td></td>
<td>SBC3150 pSDK86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC608 pSBC608</td>
<td></td>
<td></td>
<td>SBC3160 pSDK31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC614 pSBC614</td>
<td></td>
<td></td>
<td>T60 pT60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC618 pSBC618</td>
<td></td>
<td></td>
<td>TA096 pTAE96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC655 pSBC655</td>
<td></td>
<td></td>
<td>TA252 pTAE52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC6611 pSBC6611</td>
<td></td>
<td></td>
<td>TA452 pTAE42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC8010 pSBC8010</td>
<td></td>
<td></td>
<td>W140 pW140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC80204 pSBC80204</td>
<td></td>
<td></td>
<td>W280 pW280</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC8024 pSBC8024</td>
<td></td>
<td></td>
<td>W40 pW40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC8030 pSBC8030</td>
<td></td>
<td></td>
<td>W80 pW80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC80605 pSBC80605</td>
<td></td>
<td></td>
<td>XNX286DOC pXNX286DOC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC80612 pSBC80612</td>
<td></td>
<td></td>
<td>XNX286DOC0CB pXNX286DOC0CB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC80614 pSBC80614</td>
<td></td>
<td></td>
<td>XNXBASE pXNXBASE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC80630 pSBC80630</td>
<td></td>
<td></td>
<td>XNXIDB pXNXIDB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC80635 pSBC80635</td>
<td></td>
<td></td>
<td>XNXIDESK pXNXIDESK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC806C38 pSBC806C38</td>
<td></td>
<td></td>
<td>XNXIPLAN pXNXIPLAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC8825 pSBC8825</td>
<td></td>
<td></td>
<td>XNXIWORD pXNXIWORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC8840 pSBC8840</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC8845 pSBC8845</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC905 pSBC905</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBCLNK001 pSBCLNK001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Introduction to the 80960KB
INTRODUCTION

This handbook provides detailed programming information and hardware system design information for the Intel 80960KB processor (which is part of the 80960K series of embedded-processor products) as well as information on other 32-bit microprocessors, peripherals and development support tools.

Hardware designers can use this information as a guideline for developing microprocessor systems. Applications programmers, compiler designers, and designers of operating-system kernels will also find needed information on the software architecture, instruction set, and programming of the 80960KB processor.

All of the processors in the 80960K series of products are based on the Intel 80960 architecture. Most of the information in this handbook also applies to the 80960KA processor. The only difference between the 80960KB and 80960KA processors is that the 80960KA does not provide on-chip support for floating-point operations or operations on decimal numbers.

Wherever appropriate, design examples are included. These designs are based upon functional 80960KB boards and systems, and are simplified for ease of understanding. These simplified designs have not been tested except for examples that include part numbers.

The Programmer’s Reference provides programmers and system designers with detailed information about the processor’s programming environment and kernel (or executive) support facilities. It also provides detailed reference information on the 80960 architecture, beyond that found in the architecture overview.

OVERVIEW OF THE PROGRAMMER’S REFERENCE

The following is a brief overview of the contents of each section of the Programmer’s Reference portion of the manual:

Section 7 — Execution Environment. Describes the environment in which instructions are executed. The topics discussed include the address space, registers, instruction pointer, and arithmetic calls.

Section 8 — Procedure Calls. Describes the various mechanisms available for making procedure calls. The topics discussed include the local call/return mechanism, procedure stack, branch-and-link procedure calls, procedure table calls, and supervisor call mechanism.

Section 9 — Data Types and Addressing Modes. Describes non-floating-point data types and how bits and bytes are addressed. The addressing modes provided for addressing data in memory are also described in this section.

Section 10 — Instruction Set Summary. Overview of all non-floating-point instructions in the 80960KB instruction set, arranged by functional groups. The assembly language instruction format is also described.
Section 11 — Processor Management and Initialization. Describes the processor management facilities. Included is a discussion of the system data structures required to operate the processor, the software requirements for processor management, and the requirements for physical memory. Processor Initialization concludes the section.

Section 12 — Interrupts. Description of the interrupt mechanism, interrupt priority, interrupt table, interrupt handling procedures, and the software requirements for handling interrupts.

Section 13 — Fault Handling. Describes the processor’s fault-handling mechanism, including the fault-table structure, fault handling procedures, and the software requirements for handling faults. Each fault is detailed in a reference section at the end of the section.

Section 14 — Debugging. Describes the debugging and monitoring support facilities, including the trace control register.

Section 15 — Instruction Set Reference. Alphabetical listing of the complete 80960KB instruction set, with detailed descriptions of each instruction, assembly-language syntax, examples, and algorithms.

Section 16 — Floating Point Operation. Description of the floating-point processing facilities of the processor. This section includes an overview of floating-point numbers as well as a description of the 80960KB floating-point data types and their relationship to the IEEE floating-point standard. Floating-point instructions, exceptions, and faults are also described.

Section 17 — Interagent Communication. Describes the interprocessor communication (IAC) mechanism, which allows several processors to communicate with one another over the bus. The topics discussed include the IAC mechanism and software requirements for using internal IACs. Each IAC is described in detail in a reference section at the end of the section.

Appendix A — Instruction and Data Structure Quick Reference. Provides two lists of the 80960KB instructions - one alphabetical by assembly-language mnemonic and one by machine language opcode.

Appendix B — Machine-Level Instruction Formats.

Appendix C — Instruction Timing. Describes the 80960KB processor’s instruction pipeline and its effect on instruction timing. Includes each instruction’s clock cycle requirement.

Appendix D — Initialization Code. A listing of the code to initialize the 80960KB processor.

Appendix E — Considerations for Portable Software. Discusses the 80960KB architecture aspects that should be considered if code written for the 80960KB processor is intended to be ported later to other implementations of the 80960 architecture.
NOTATION AND TERMINOLOGY CONVENTIONS

The following paragraphs describe the notation style conventions used in the architectural overview and programmer's reference chapters, as well as terminology that has special meaning as used in this handbook.

Integer numbers are presented in decimal format unless otherwise indicated by the subscript “H” for hexadecimal or “B” for binary.

An active low signal is represented by a solid line over the signal name.

Reserved and Preserved

Certain fields in the processor's system data structures are described as being either reserved fields or preserved fields.

A reserved field is one that other implementations of the 80960 architecture can use. To help ensure that a current software design will be compatible with future processors based on the 80960 architecture, the bits in the reserved fields should be set to 0 when the structure is initially created. Thereafter, software should not access these fields.

Some fields in system data structures are shown as being required to be set to either 1 or 0. These fields should be treated as if they were reserved fields. They should be set to the specified value when the data structure is created, and should not be accessed by software after that.

A preserved field is one that the processor does not use. Software may use preserved fields for any function.

Set and Clear

The terms set and clear are used in this manual to refer to the value of a bit field in a system data structure. If a bit is set, its value is 1; if the bit is clear, its value is 0. Likewise, setting a bit means giving it a value of 1 and clearing a bit means giving it a value of 0.

OVERVIEW OF THE 80960KB ARCHITECTURE

The 80960KB processor introduces the 80960 architecture - a new 32-bit architecture from Intel. This architecture has been designed to meet the needs of embedded applications such as machine control, robotics, process control, avionics, and instrumentation.

The 80960 architecture can best be characterized as a high-performance computing engine. It features high-speed instruction execution and ease of programming. It is also easily extensible, allowing processors and controllers based on this architecture to be conveniently customized to meet the needs of specific processing and control applications.
The following are some of the important attributes of the 80960 architecture:

- Full 32-bit registers
- High-speed, pipelined instruction execution
- A convenient program execution environment with 32 general-purpose registers and a versatile set of special-function registers
- A highly optimized procedure call mechanism that features on-chip caching of local variables and parameters
- Extensive facilities for handling interrupts and faults
- Extensive tracing facilities to support efficient program debugging and monitoring
- Register scoreboarding and write buffering to permit efficient operation when used with lower performance memory subsystems.

OVERVIEW OF THE SINGLE PROCESSOR SYSTEM ARCHITECTURE

The central processing module, memory module, and I/O module form the natural boundaries for the hardware system architecture. The modules are connected together by the high bandwidth 32-bit multiplexed L-bus, which can transfer data at a maximum sustained rate of 53M bytes per second for an 80960 processor operating at 20 MHz.

Figure 1 shows a simplified block diagram of one possible system configuration. The heart of this system is the 80960B processor, which fetches instructions, executes code, manipulates stored information, and interacts with I/O devices. The high bandwidth L-bus connects the 80960KB processor to memory and I/O modules. The 80960KB processor stores system data, instructions, and programs in the memory module. By accessing various peripheral devices in the I/O module, the 80960KB processor supports communication to terminals, modems, printers, disks, and other I/O devices.

80960KB Processor and the L-Bus

The 80960KB processor performs bus operations using multiplexed address and data signals, and provides all the necessary control signals. For example, standard control signals, such as Address Latch Enable (ALE), Address/Data Status (ADS), Write/Read Command (W/R), Data Transmit/Receive (DT/R), and Data Enable (DEN), are provided by the 80960KB processor. The 80960 processor also generates byte enable signals that specify which bytes on the 32-bit data lines are valid for the transfer.

The L-bus supports burst transactions, which access up to four data words at a maximum rate of one word per clock cycle. The 80960KB processor uses the two low-order address lines to indicate how many words are to be transferred. The 80960KB processor performs burst transactions to load the on-chip 512-byte instruction cache to minimize memory accesses for instruction fetches. Burst transactions can also be used for data access.
To transfer control of the bus to an external bus master, the 80960KB provides two arbitration signals: hold request (HOLD) and hold acknowledge (HLDA). After receiving HOLD, the processor grants control of the bus to an external master by asserting HLDA.

The 80960KB processor provides a flexible interrupt structure by using an on-chip interrupt controller, an external interrupt controller, or both. The type of interrupt structure is specified by an internal interrupt vector register. For a system with multiple processors, another method is available, called inter-agent communication (IAC) where a processor can interrupt another processor by sending an IAC message.

**Memory Module**

A memory module can consist of a memory controller, Erasable Programmable Read Only Memory (EPROM), and static or dynamic Random Access Memory (RAM). The memory controller first
conditions the L-bus signals for memory operation. It demultiplexes the address and data lines, generates the chip select signals from the address, detects the start of the cycle for burst mode operation, and latches the byte enable signals.

The memory controller generates the control signals for EPROM, SRAM, and DRAM. Specifically, it provides the control signals, multiplexed row/column address, and refresh control for dynamic RAMs. The controller can be designed to accommodate the burst transaction of the 80960KB processor by using the static column mode or nibble mode features of the dynamic RAM. In addition to supplying the operational signals, the controller generates the READY signal to indicate that data can be transferred to or from the 80960KB processor.

The 80960KB processor directly addresses up to 4G bytes of physical memory. The processor does not allow burst accesses to cross a 16-byte boundary, to ease the design of the controller. Each address specifies a four-byte data word within the block. Individual data bytes can be accessed by using the four byte-enable signals from the 80960KB processor. Chapter 5 provides design guidelines for the memory controller.

I/O Module

The I/O module consists of the I/O components and the interface circuit. I/O components can be used to allow the 80960KB processor to use most of its clock cycles for computational and system management activities. Time consuming tasks can be off-loaded to specialized slave-type components, such as the 8259A Programmable Interrupt Controller or the 82530 Serial Communication Controller. Some tasks may require a master-type component, such as the 82586 Local Area Network Control.

The interface circuit performs several functions. It demultiplexes the address and data lines, generates the chip select signals from the address, produces the I/O read or I/O write command from the processor’s W/R signal, latches the byte enable signals, and generates the READY signals. Since some of these functions are identical to those of the memory controller, the same logic can be used for both interfaces. For master-type peripherals that operate on a 16-bit data bus, the interface circuit translates the 32-bit data bus to a 16-bit data bus.

The 80960KB processor uses memory-mapped addresses to access I/O devices. This allows the CPU to use many of the same instructions to exchange information for both memory and peripheral devices. Thus, the powerful memory-type instructions can be used to perform 8-, 16-, and 32-bit data transfers.

HIGH PERFORMANCE PROGRAM EXECUTION

Much of the design of the 80960 architecture has been aimed at maximizing the processor’s computational and data processing speed through the use of increased parallelism. The following paragraphs describe several of the mechanisms and techniques used to accomplish this goal.
Load and Store Model

One of the more important features of the 80960 architecture is its performance of most operations on operands in registers, rather than in memory. For example, all arithmetic, logic, comparison, branching and bit operations are performed with registers and literals.

This feature provides two benefits. First, it increases program execution speed by minimizing the number of memory accesses necessary to execute a program. Second, it reduces the memory latency encountered when using slower, lower-cost memory parts.

To support this concept, the architecture provides a generous supply of general-purpose registers. For each procedure, 32 registers are available, 28 of which are available for general use. These registers are divided into two types: global and local. Both types of registers can be used for general storage of operands. The only difference is that global registers retain their contents across procedure boundaries, whereas the processor allocates a new set of local registers each time a new procedure is called.

The architecture also provides a set of fast, versatile load and store instructions. These instructions allow burst transfers of 1, 2, 4, 8, 12, or 16 bytes of information between memory and the registers.

On-Chip Caching of Code and Data

To further reduce memory accesses, the architecture offers two mechanisms for caching code and data on chip: an instruction cache and multiple sets of local registers. The instruction cache allows prefetching of blocks of instruction from memory. This helps ensure that the instruction execution pipeline is supplied with a steady stream of instructions. It also reduces the number of memory accesses required when performing iterative operations such as loops. The architecture allows the size of the instruction cache to vary. For the 80960KB processor, it is 512 bytes.

To optimize the architecture's procedure call mechanism, the processor provides multiple sets of local registers. This allows the processor to perform procedure calls without having to write the local registers out to the stack in memory. The number of register sets depends on the processor implementation. The 80960KB processor provides four sets of local registers.

Overlapped Instruction Execution

The 80960 architecture also enhances program execution speed by overlapping the execution of some instructions. In the 80960K series of processors, this is accomplished through register scoreboard.

Register scoreboard permits instruction execution to continue while data is being fetched from memory. When a load instruction is executed, the processor sets one or more scoreboard bits to indicate the target registers to be loaded. After the target registers are loaded, the scoreboard bits are cleared. While the target registers are being loaded, the processor is allowed to execute other instructions that do not use these registers.
The processor uses the scoreboard bits to ensure that the target registers are not used until the loads complete. (Scoreboard bits are checked transparently from software.) This technique allows code to be executed such that some instructions can be executed in zero clock cycles (that is, executed for free).

**Single-Clock Instructions**

The 80960 architecture is designed to let a processor execute commonly used instructions, such as moves, adds, subtracts, logical operations, and branches, in a minimum number of clock cycles (preferably one cycle). The architecture supports this concept in several ways. For example, the load and store model described earlier eliminates the clock cycles required to perform memory-to-memory operations, by concentrating on register-to-register operations.

In addition, all of the instructions in the 80960 architecture are 32 bits long and aligned on 32-bit boundaries. This lets instructions be decoded in one clock cycle, and eliminates the need for an instruction-alignment stage in the pipeline.

The 80960KB processor takes full advantage of these features of the architecture, resulting in more than 50 instructions that can be executed in a single clock cycle.

**Efficient Interrupt Model**

The 80960 architecture provides an efficient mechanism for servicing interrupts from external sources. To handle interrupts, the processor maintains an interrupt table of 248 interrupt vectors, 240 of which are available for general use. When an interrupt is signaled, the processor uses a pointer to the interrupt table to perform an implicit call to an interrupt handler procedure. In performing this call, the processor automatically saves the state of the processor prior to receiving the interrupt, performs the interrupt routine, then restores the state of the processor. A separate interrupt stack is also provided to segregate interrupt handling from application programs.

The interrupt handling facilities also allow interrupts to be evaluated by priority. The processor is then able to store interrupt vectors that are lower in priority than the current processor task in a pending interrupt section of the interrupt table. The processor checks and services the pending interrupts at defined times.

**SIMPLIFIED PROGRAMMING ENVIRONMENT**

Because of its streamlined execution environment, processors based on the 80960 architecture are particularly easy to program. The following paragraphs describe some of the architecture features that simplify programming.

**Highly Efficient Procedure Call Mechanism**

The procedure call mechanism makes procedure calls and parameter passing between procedures simple and compact. Each time a call instruction is issued, the processor automatically automatically...
saves the current set of local registers and allocates a new set for the called procedure. Likewise, on a return from a procedure, the current set of local registers is deallocated and the local registers for the procedure being returned to are restored. This means a program never has to explicitly save and restore those local variables that are stored in local registers.

**Versatile Instruction Set and Addressing**

The selection of instructions and addressing modes also simplifies programming. A full set of load, store, move, arithmetic, comparison, and branch instructions are provided, with operations on both integer and ordinal data types. Operations on bits and bit strings are simplified by a complete set of Boolean and bit-field instructions.

The addressing modes are efficient and straightforward, while at the same time providing the necessary indexing and scaling modes required to address complex arrays and record structures. The large 4-gigabyte address space provides ample room to store programs and data. The availability of 32 addressing lines allows some address lines to be memory-mapped to control hardware functions.

**Extensive Fault Handling Capability**

To aid in program development, the 80960 architecture defines a wide range of faults that the processor detects, including arithmetic faults, invalid operations, invalid operands, and machine faults. When a fault is detected, the processor makes an implicit call to a fault handler routine, in a way similar to the interrupt mechanism described previously. The information collected for each fault allows program developers to quickly correct faulting code, and allows automatic recovery from some faults.

**Debugging and Monitoring**

To support debugging systems, the 80960 architecture provides a mechanism for monitoring processor activity by means of trace events. When the processor detects a trace event, it signals a trace fault and calls a fault handler. Intel provides several tools that use this feature, including an in-circuit emulator (ICE) device.

**SUPPORT FOR ARCHITECTURAL EXTENSIONS**

The 80960 architecture provides several features that enable processors based on this architecture to be easily customized to meet the needs of specific embedded applications, such as signal processing, array processing, or graphics processing.

The most important of these features is the set of 32 special function registers. These registers provide a convenient interface to circuitry in the processor or pins that can be connected to external hardware. They can be used to control timers, to perform operations on special data types, or to perform I/O functions. The special function registers are similar to the global registers. They can be addressed by all of the register access instructions.
EXTENSIONS INCLUDED IN THE 80960K SERIES PROCESSORS

The 80960K series of processors provides a complete implementation of the 80960 architecture, plus several extensions to that architecture. These extensions fall into two categories: floating-point processing and interagent communication.

On-Chip Floating Point

The 80960KB processor provides a complete implementation of the IEEE standard for binary floating-point arithmetic (IEEE 754-185). This implementation includes a full set of floating point operations, including add, subtract, multiply, divide, trigonometric functions, and logarithmic functions. These operations are performed on single precision (32-bit), double precision (64-bit), and extended precision (80-bit) real numbers.

One of the benefits of this implementation is that the floating-point handling facilities are integrated into the normal instruction execution environment. Single and double precision floating-point values are stored in the same registers as non-floating point values. Four 80-bit floating-point registers are provided to hold extended-precision values.

Interagent Communication

All of the processors in the 80960K series provide an interagent communication (IAC) mechanism, allowing agents connected to the processor’s bus to communicate with one another. This mechanism operates similarly to the interrupt mechanism, except that IAC messages are passed through dedicated sections of memory. The sort of tasks handled with IAC messages are processor reinitialization, stopping the processor, purging the instruction cache, and forcing the processor to check pending interrupts.
80960KB Architectural Overview
1.0 INTRODUCTION

The 80960KB is the first 32-bit microprocessor designed especially for embedded applications. At an operating frequency of 20 MHz, this high performance processor can sustain an instruction execution rate of seven and one-half million instructions per second (MIPS), and burst rates of 20 MIPS. The 80960KB processor enhances embedded system performance by integrating special features to eliminate the need for additional peripheral devices and the associated software overhead. For instance, the 80960KB processor offers an on-chip floating-point processing unit, an improved interrupt handling capability, and support for debugging and tracing. This chapter describes the architectural attributes and enhancements of the 80960KB processor for embedded computing.

1.1 ARCHITECTURAL ATTRIBUTES FOR EMBEDDED COMPUTING

For over a decade, Intel has designed a large variety of 8- and 16-bit microcontrollers to fit the needs of embedded applications. Based on this experience, several architectural attributes shared by both microcontrollers and microprocessors can be implemented that benefit embedded applications and enhance microprocessor performance. Because the 80960KB processor incorporates these attributes (listed below) in its architecture, embedded applications are easy to design, perform well, and get to market fast.

- Simple load/store design
- Large general-purpose register sets
- Boolean and bit-field instructions
- Small number of operations and addressing modes
- Simplified instruction format
- Minimum cycle operation

1.1.1 Load/Store Design

In the 80960 family architecture, operations are register-to-register, with only LOAD and STORE instructions accessing memory. This attribute simplifies the instruction set and shortens cycle time. The 80960KB processor uses LOAD and STORE instructions to access memory. It further minimizes accesses to memory by providing a 512-byte, direct-mapped instruction cache. When a memory access is required, the processor can perform a burst transaction that accesses up to four data words with one word transferred every clock cycle.

1.1.2 Large General-Purpose Register Sets

Because the instructions operate on operands within registers, the 80960 family uses many registers. The 80960KB processor features large, versatile register sets. For maximum flexibility, each processor provides 32 32-bit registers and four 80-bit floating-point registers.
There are two types of general-purpose registers: local and global. The processor automatically accesses the 16 local registers when a procedure call is performed. Multiple sets of local registers are stored on-chip to further increase the efficiency of this register set, as shown in Figure 1. The register cache holds up to four local register frames, which means that up three procedure calls can be made without having to access the procedure stack resident in memory.

The 20 global registers retain their contents across procedure boundaries. The global registers consist of sixteen 32-bit registers \((G_0 \text{ through } G_{15})\) and four 80-bit registers \((FP_0 \text{ through } FP_3)\) as shown in Figure 2. While all registers can be used for floating-point operations, the 80-bit registers are used for accumulation of extended precision results.

### 1.1.3 Small Number of Addressing Modes

The 80960 family uses relatively few addressing modes to facilitate a fast, simple interpretation by the control engine. The 80960KB processor provides simple, fast addressing modes, as well as a few complex addressing modes to allow optimizations for code density.

### 1.1.4 Simplified Instruction Format

A simplified instruction format eases the hardwired decoding of instructions, which again speeds control paths. The 80960KB processor’s instruction formats are simple and word aligned; all instructions are one word long except for one class that uses the subsequent word as a 32-bit displacement. To further enhance performance, the instructions do not cross word boundaries. This feature eliminates a pipeline stage (that would have to align instructions) and decreases instruction execution time.
1.1.5 Overlapped Execution

To optimize performance, the 80960KB processor overlaps instruction execution by means of write buffering and register scoreboarding. Write buffering allows a write instruction to proceed as soon as it is placed in the buffer. It does not have to wait for the actual write operation to occur on the L-bus.

Similarly, register scoreboarding is a design technique that allows the 80960KB to continue execution of instructions when it encounters a LOAD instruction. When the LOAD instruction begins, the 80960KB sets a scoreboard bit on the target register. After the target register is loaded with data, the processor resets the bit. While the data is being retrieved, additional instructions that do not reference the target register can be executed.

The 80960KB ensures that these additional instructions do not reference the target register by checking the scoreboard transparently (no software required). Thus, the scoreboard feature reduces the effect of slow memory speed and provides a useful tool for optimizing procedures.
1.1.6 Minimum cycle operation

The 80960KB processor executes most of the core instructions in a single clock cycle. For these instructions, the 80960KB processor uses hardwired logic rather than microcode to execute the instruction.

The 80960KB also supports a number of important multicycle instructions, such as 32-bit multiply and divide instructions. These auxiliary functions require more than one clock cycle because it is more efficient to use microcode than hardwired logic. On the other hand, the integration of these functions on-chip eliminates much software overhead and the negative effects on code density that would be otherwise required. Thus, the additional functionality of the 80960KB enhances overall system performance while keeping code size small.

1.2 ADDITIONAL 80960KB ARCHITECTURAL ENHANCEMENTS

The 80960KB incorporates two useful features: an on-chip floating-point processing and debugging functions. The floating-point unit can be used for applications that require precision such as machine-control operations. The debugging function significantly decreases development time.

1.2.1 Floating-point Operation

The on-chip floating-point unit of each processor improves the performance of floating-point calculations by eliminating bus overhead used to transfer operands to a coprocessor. The processor provides hardware support for both mandatory and recommended portions of IEEE standard 754 for floating-point arithmetic, exponential, logarithmic, and other transcendental functions. By integrating the floating-point unit on-chip, the 80960KB processor reduces the overall chip count for a system, decreases power consumption, and increases overall performance and reliability.

1.2.2 Debug Capabilities

The processor provides extensive system debug capabilities, an important feature for embedded computing where the ability to instrument an application may be limited. The 80960KB processor allows breakpoint instructions that stop program execution on various events, such as procedure calls, or certain instructions. Another debug facility traces the activity of the processor while it is executing a program. Tracing is done by recording the addresses of instructions that cause trace events to occur. For example, a trace event can occur on the execution of a specific instruction, branch, or procedure call. To ensure that the 80960KB is operating properly, the processor performs a self-test when it is reset. If the self-test is successful, the 80960KB begins operation, otherwise it enters the stopped state.

1.3 STANDARD BUS INTERFACE

The advanced features of the 80960KB processor are implemented using a performance optimized bus interface. The processor uses a high bandwidth local bus (L-bus) that consists of standard signal
groups: a 32-bit multiplexed address/data path and control signals for data transactions. Because of
the large amount of caching, the L-bus supports burst transactions that transfer up to four successive
data words. Transactions on the L-bus can use 8-, 16-, and 32-bit data types and address up to 4G bytes
of physical memory. Bus arbitration can be accomplished by simply using the hold request/hold
acknowledge protocol.

1.4 INTER-AGENT COMMUNICATION/COPROCESSOR CAPABILITIES

The 80960KB processor offers a flexible way to manage interrupts. It accepts interrupts in one of
three ways: by communicating with an external interrupt controller using the standard Interrupt/
Interrupt Acknowledge signals, by activating the on-chip interrupt controller, or by accepting an
inter-agent communication (IAC) message. This allows the 80960KB to act as a coprocessor on a
shared bus with another CPU.

1.5 SUMMARY

The 80960KB processor optimizes embedded system performance by using a new 32-bit architec-
ture. The 80960 family architecture includes a load/store design, large general purpose (register sets,
fast addressing modes, a simplified instruction format, and minimized instruction execution cycles.

To further enhance system performance, the 80960KB processor provides floating-point operation,
interrupt controller capabilities, and debug functions. By intergrating these functions on-chip, the
80960KB reduces the power requirements and overall chip count for a system.

As a result of the 80960 architecture, the 80960KB processor provides unprecedented performance.
For a speed selection of 20 MHz, it can sustain an instruction execution rate of over seven and one-
half MIPS and burst rates of 20 MIPS, speeds comparable to that of super minicomputers. The high
instruction execution rates are made possible through a innovative design that incorporates an on-
chip instruction cache with burst-transfer capability.

2.0 80960KB SYSTEM ARCHITECTURE

This section illustrates the flexibility and power of the 80960KB system architecture using the
advanced 32-bit 80960KB processor. The section examines system configurations from general
perspective to explain the design concepts. Subsequent sections describe the the system design.

2.1 OVERVIEW OF A SINGLE PROCESSOR SYSTEM ARCHITECTURE

The central processing module, memory module, and I/O module form the natural boundaries for the
hardware system architecture. The modules are connected together by the high bandwidth 32-bit
multiplexed L-bus, which can transfer data at a maximum sustained rate of 53M bytes per second for
an 80960KB processor operating at 20 MHz.
Figure 3 shows a simplified block diagram of a possible system configuration. The heart of this system is the 80960KB processor, which fetches program instructions, executes code, manipulates stored information, and interacts with I/O devices. The high bandwidth L-bus connects the 80960KB processor to memory and I/O modules. The 80960KB processor stores system data and instructions and programs in the memory module. By accessing various peripheral devices in the I/O module, the 80960KB processor supports terminals, modems, printers, disks, and other I/O devices.

2.1.1 80960KB Processor and the L-Bus

The 80960KB processor performs bus operations using multiplexed address and data signals and provides all the necessary control signals. For example, standard control signals, such as Address Latch Enable (ALE), Address/Data Status (ADS), Write/Read command (W/R) Data Transmit/Receive (DT/R) and Data Enable (DEN) are provided by the 80960KB processor. The 80960KB processor also generates byte enable signals that specify which bytes on the 32-bit data lines are valid for the transfer.
The L-bus supports burst transactions, which access up to four data words at a maximum rate of one word per clock cycle. The 80960KB processor uses the two low-order address lines to indicate how many words are to be transferred. The 80960KB processor performs burst transactions to load the on-chip 512-byte instruction cache to minimize memory accesses for instruction fetches. Burst transactions can also be used for data accesses.

To transfer control of the bus to an external bus master, the 80960KB processor provides two arbitration signals: hold request (HOLD) and hold acknowledge (HLDA). After receiving HOLD, the processor grants control of the bus to an external bus master by asserting HLDA.

The 80960KB processor provides a flexible interrupt structure by using an on-chip interrupt controller, an external interrupt controller, or both. The type of interrupt structure is specified by an internal interrupt vector register. For a system with multiple processors, another method is available, called inter-agent communication (IAC) where a processor can interrupt another processor by sending an IAC message.

Complete details of the L-bus and bus operations are discussed in Section 3.

### 2.1.2 Memory Module

A memory module can consist of the memory controller, Erasable Programmable Read Only Memory (EPROM), and static or dynamic Random Access Memory (RAM). The memory controller first conditions the L-bus signals for memory operation. It demultiplexes the address and data lines, generates the chip select signals from the address, detects the start of the cycle for burst mode operation, and latches the byte enable signals.

The memory controller generates the control signals for EPROM, SRAM, and DRAM. In particular, it provides the control signals, multiplexed row/column address, and refresh control for dynamic RAMs. The controller can be designed to accommodate the burst transaction of the 80960KB processor by using the static column mode or nibble mode features of the dynamic RAM. In addition to supplying the operation signals, the controller generates the READY signal to indicate that data can be transferred to or from the 80960KB processor.

The 80960KB processor directly addresses up to 4G bytes of physical memory. The processor does not allow burst accesses to cross a 16-byte boundary to ease the design of the controller. Each address specifies a four-byte data word within the block. Individual data bytes can be accessed by using the four byte enable signals from the 80960KB processor.

Section 4 provides design guidelines for the memory controller.

### 2.1.3 I/O Module

The I/O module consists of the I/O components and the interface circuit. I/O components can be used to allow the 80960KB processor to use most of its clock cycles for computational and system management activities. Time consuming tasks can be off-loaded to specialized slave-type components, such as the 8259A Programmable Interrupt Controller, or the 82530 Communication
Controller. Some tasks may require a) master-type component, such as the 82586 Local Area Network Control.

The interface circuit performs several functions. It demultiplexes the address and data lines, generates the chip select signals from the address, produces the I/O read or I/O write command from the processor’s W/R signal, latches the byte enable signals, and generates the READY signal. Because these functions are the same as some of the functions of the memory controller, the same logic can be used for both interfaces. For master-type peripherals that operate on a 16-bit data bus, the interface circuit translates the 32-bit data bus to a 16-bit data bus.

The 80960KB processor uses memory-mapped addresses to access I/O devices. This allows the CPU to use many of the same instructions to exchange information for both memory and peripheral devices. Thus, the powerful memory-type instructions can be used to perform 8-, 16-, and 32-bit data transfers.

Section 5 describes design guidelines for the I/O interface by examining representative design examples.

2.2 SUMMARY

The basic hardware system configuration is modular and flexible. The processor, memory, and I/O modules form the natural boundaries in the basic hardware system architecture. The high-bandwidth L-bus that supports burst transfers is used for the data path between the 80960KB processor and other modules.

3.0 THE 80960KB PROCESSOR AND THE LOCAL BUS

The 32-bit multiplexed local bus (L-bus) connects the 80960KB processor to memory and I/O and forms the backbone of any 80960KB processor based system. This high bandwidth bus provides burst-transfer capability allowing up to four successive 32-bit data word transfers at a maximum rate of one word every clock cycle. In addition to the L-bus signals, the 80960KB processor uses other signals to communicate to other bus masters. This section, which describes these signals and the associated operations, follows the outline shown below:

- L-bus states and their relationship to each other
- L-bus signal groups, which consist of address/data and control
- L-bus read, write, and burst transactions
- L-bus timing analyses and timing circuit generation
- Related L-bus operations such as arbitration, interrupt, and reset operations
3.1 OVERVIEW OF THE 80960KB L-BUS

The L-bus forms the data communication path between the various components in a basic 80960KB hardware system. The 80960KB processor utilizes the L-bus to fetch instructions, to manipulate information from both memory and I/O devices, and to respond to interrupts. To perform these functions at a high data rate, the 80960KB processor provides a burst mode, which transfers up to four data words at a maximum rate of one 32-bit word per clock cycle. The 80960KB L-bus has the following features:

- 32-bit multiplexed address/data path
- High data bandwidth relative to the speed selection of the 80960KB processor
- Four byte enables and a four-word burst capability that allow transfers from 1 to 16 bytes in length
- Support for TTL latches and buffers.

3.2 BASIC L-BUS STATES

The L-bus has five basic bus states: idle (T_i), address (T_a), data (T_d), recovery (T_r), and wait (T_w). During system operation, the 80960KB processor continuously enters and exits different bus states as shown in Figure 4. This state diagram assumes that only one bus master resides on the L-bus.

The processor occupies the T_i state when no address/data transfers are in progress. When a new request is received, the 80960KB processor enters the T_a state to transmit the address.

Following a T_a state, the 80960KB processor enters a T_d state to transmit or receive data on the address/data lines provided that the data is (indicated by the assertion of READY at the input of the processor). If the data is not ready, the processor enters a T_w state and remains in this state until data is ready.

T_w states may be repeated as many times as necessary to allow sufficient time for the memory or I/O device to respond.

After a data word is transferred, the 80960KB processor exits the T_d or T_w state for a single word transfer or enters the T_d state again to transfer another data word for a burst transaction. If the next data word is not ready during the next clock cycle for a burst transaction, the processor enters the T_w state again.

When the 80960KB processor completes the data transfer of all the data words (one or up to four), it enters the recovery (T_r) state to allow sufficient time for devices (such as memories) on the bus to recover. The processor returns to the T_i state if no new request is pending, or enters the T_i state if a new request is pending.
3.3 L-BUS SIGNAL GROUPS

The L-bus states are used to define some of the L-bus signals. As shown in Figure 5, the signals on the L-bus consist of two basic groups: address/data, and control.
3.3.1 Address/Data

The address/data signal group consists of 32 bidirectional lines. These signals are multiplexed and serve a dual purpose depending upon the bus state.

$LAD_{31} - LAD_2$  

Local Address/Data$_{31}$ through Local Address/Data$_{2}$ represent the address signals on the L-bus during the Ta state. $LAD_2$ is the least significant bit, and $LAD_{31}$ is the most significant address bit. $LAD_{31}$ through $LAD_2$ contain a physical word address.

$LAD_1$ and $LAD_0$ specify the number of data words to transfer for a burst transaction. The address/data signals float to a high impedance state when not activated.

SIZE ($LAD_1 - LAD_0$)  
The SIZE signal indicates whether one, two, three, or four words are transferred during the current transaction. During a T_a state, $LAD_1$ and $LAD_0$ represent the word size signals. The encoding is shown in Table 1.

<table>
<thead>
<tr>
<th>Word Selection</th>
<th>$LAD_1$</th>
<th>$LAD_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Word</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>2 Words</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>3 Words</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>4 Words</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 1. SIZE Signal Decoding
LAD

Local Address/Data through Local Address/Data represent the data signal on the L-bus during the T
and T
states. LAD
is the least significant and LAD
is the most significant address bit. The address/data signals float to a high impedance state when not activated.

3.3.2 Control

The control signal group consists of 12 signals that permit the transfer of data. These signals can be used to control data buffers, address latches, and other standard interface logic.

**ALE**

The Address Latch Enable is an active low signal that can be used to latch the address from the 80960KB processor. ALE is asserted during the T_a state and deasserted before the beginning of the T_d state. ALE floats to a high impedance level when the processor is not operating on the bus (i.e., it is in the idle state), or is at the end of any bus access.

**ADS**

Address/Data Status is an active low signal that is driven by the 80960KB processor to indicate an address state. ADS is asserted during every T_a state and deasserted after the following T_d and T_w states. For a burst transaction, ADS is asserted again every T_d (and T_w) state where READY was asserted in the prior cycle. The signal is an open drain output.

**DT/R**

Data Transmit/Receive indicates the direction of data flow to or from the L-bus. For a read operation or an interrupt acknowledgement, DT/R is low during the T_a, T_w, and T_d states to indicate that data flows into the 80960KB processor. For a write operation, DT/R is high during the T_a, T_w, and T_d states to indicate that data flows from the 80960KB processor. DT/R never changes states when DEN is asserted. The DT/R line is an open drain output of the 80960KB processor.

**DEN**

Data Enable is an active-low signal that can be used to enable data transceivers. DEN is asserted during all T_d and T_w states. The DEN line is an open drain output of the 80960KB processor.

**W/R**

The Write/Read signal instructs memory or I/O device to write or read data on the L-bus. The 80960KB processor asserts W/R during a T_a state. The signal remains valid during subsequent T_d and T_w states. W/R is an open drain output of the 80960KB processor.

**BE3-BE0**

The Byte Enable output signals of the 80960KB processor specify which bytes (up to four) on the 32-bit data bus are transferred during the transaction. Table 2 shows the decoding scheme.

The byte enable signals are valid from the 80960KB processor before data is transferred, as shown in Figure 6 (assumes no wait states). The byte enable signals that are valid for the first data word are specified during the T_a state.
For a four-word burst transaction, the byte enable signals that are valid for the second word are asserted during the first data state \( (T_{d0}) \) for the third word during the second data state \( (T_{d1}) \) and for the fourth word during the third data state \( (T_{d2}) \). The byte enable signals are undefined during the last data state \( (T_{d3}) \) of the last word transferred.

**Table 2. Byte Enable Signal Decoding**

<table>
<thead>
<tr>
<th>Byte Enable Signal</th>
<th>Address Line Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \overline{BE}_0 )</td>
<td>( LAD_7-LAD_0 )</td>
</tr>
<tr>
<td>( \overline{BE}_1 )</td>
<td>( LAD_{15}-LAD_8 )</td>
</tr>
<tr>
<td>( \overline{BE}_2 )</td>
<td>( LAD_{23}-LAD_{16} )</td>
</tr>
<tr>
<td>( \overline{BE}_3 )</td>
<td>( LAD_{31}-LAD_{24} )</td>
</tr>
</tbody>
</table>

**Figure 6. Byte Enable Timing Diagram**

Although not shown in the diagram, the byte enable signals of each word are latched internally by the 80960KB processor and remain valid during every data or wait state until READY is applied. After READY is applied, the byte enable signals change during the next \( T_d \) state or become undefined for the last data transfer.

The 80960KB processor asserts only adjacent byte enables. For example, the 80960KB processor does not perform a bus operation with only \( \overline{BE}_0 \) and \( \overline{BE}_2 \) active.

The Byte Enable lines are open drain outputs.
**READY**

READY signal indicates that the data on the L-bus can be sampled (read) or removed (write) by the 80960KB processor. If READY is not asserted following T_a state or in between T_d states, a T_w state is generated. The READY is an active-low input signal to the 80960KB processor.

**LOCK**

Bus LOCK prevents other bus masters from gaining control of the L-bus during a bus operation. It is activated by certain 80960KB processor operations and instructions.

The 80960KB processor uses the bus LOCK signal when it performs a RMW memory operation. When the processor performs a RMW-Read operation, it asserts the LOCK signal during the T_a state and holds LOCK asserted. If the signal was already asserted, the processor waits until this signal is deasserted before performing the RMW-Read operation. The processor deasserts the LOCK signal during the T_a state when it performs a RMW-Write operation.

The 80960KB processor asserts the LOCK signal during the interrupt acknowledge sequence. LOCK is an input and an open drain output.

**CACHE**

The Cacheable signal specifies whether the data is cacheable. If the 80960KB processor asserts CACHE during the T_a state, then the data is cacheable. The CACHE signal is undefined during the T_d and T_w states. The CACHE signal floats to a high impedance state when the L-bus is not acquired.

Table 3 summarizes the L-bus signals.
<table>
<thead>
<tr>
<th>Signal Group</th>
<th>Signal Symbol</th>
<th>Signal Function</th>
<th>Active State</th>
<th>Direction</th>
<th>Type of Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Address/</td>
<td>Address</td>
<td>32-bit address</td>
<td>$T_a$</td>
<td>O</td>
<td>3-state</td>
</tr>
<tr>
<td>Data</td>
<td>(LAD$<em>{21}$-LAD$</em>{0}$)</td>
<td>32-bit data</td>
<td>$T_d$, $T_w$</td>
<td>I/O</td>
<td>3-state</td>
</tr>
<tr>
<td>Size</td>
<td>(LAD$<em>{1}$-LAD$</em>{0}$)</td>
<td>Specifies number of words to transfer</td>
<td>$T_a$</td>
<td>O</td>
<td>3-state</td>
</tr>
<tr>
<td>Control</td>
<td>ALE</td>
<td>Enables address latch</td>
<td>$T_a$</td>
<td>O</td>
<td>3-state</td>
</tr>
<tr>
<td></td>
<td>ADS</td>
<td>Identifies an address state</td>
<td>$T_a$, $T_d$, $T_w$</td>
<td>O</td>
<td>Open drain</td>
</tr>
<tr>
<td></td>
<td>DT/R</td>
<td>Controls direction of data flow</td>
<td>$T_a$, $T_d$, $T_w$</td>
<td>O</td>
<td>Open drain</td>
</tr>
<tr>
<td></td>
<td>DEN</td>
<td>Enables data transceiver/latch</td>
<td>$T_d$, $T_w$</td>
<td>O</td>
<td>Open drain</td>
</tr>
<tr>
<td></td>
<td>W/R</td>
<td>Read/write command</td>
<td>$T_a$, $T_d$, $T_w$</td>
<td>O</td>
<td>Open drain</td>
</tr>
<tr>
<td></td>
<td>BE$<em>{3}$-BE$</em>{0}$</td>
<td>Specifies which data bytes to transfer</td>
<td>$T_a$, $T_d$, $T_w$</td>
<td>O</td>
<td>Open drain</td>
</tr>
<tr>
<td></td>
<td>READY</td>
<td>Indicates data is ready to transfer</td>
<td>$T_d$, $T_w$</td>
<td>I</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>LOCK</td>
<td>Locks bus</td>
<td>Any</td>
<td>I/O</td>
<td>Open drain</td>
</tr>
<tr>
<td></td>
<td>Cache</td>
<td>Indicates cacheable transaction</td>
<td>$T_a$</td>
<td>O</td>
<td>3-state</td>
</tr>
</tbody>
</table>

Note: 1 except first $T_d$, $T_w$
2 except last $T_d$, $T_w$

Additional pins are used by the 80960KB processor to control the execution of instructions and to interface to other bus masters. These pins include the arbitration, interrupt, error, and reset signals. Each of these signal groups are explained in separate sections.
3.4 L-BUS TRANSACTIONS

The 80960KB processor uses the L-bus signals to perform transactions, which are simply L-bus operations where data is transferred to (or from) the CPU from (or to) another component. During a transaction, the 80960KB processor can transfer up to four words of data for a single address to enhance system throughput. This is especially useful when loading cache memory.

3.4.1 Clock Signal

The 80960KB hardware system typically uses two clock signals, CLK2 and CLK, to synchronize the transitions between L-bus states. CLK2 is the clock input to the 80960KB and is double the specified processor frequency. CLK is the clock input signal to the peripheral devices, and it is the operating frequency of the 80960KB processor. Figure 7 shows the relationship between the system CLK2 and CLK.

![Clock Relationships](image)

3.4.2 Basic Read

The basic transaction reads or writes one data word. Figure 8 shows a typical timing diagram for a basic read transaction (for exact timings, see the 80960KB processor data sheet). A read transaction may be preceded and succeeded by any type of bus transaction. The following sequence of events explains the flow of the timing diagram. For simplicity, no wait states are shown.

1. The 80960KB processor generates several signals during the \( T_a \) state.
   - It transmits the address on the address/data lines. \( \text{LAD}_1 \) and \( \text{LAD}_0 \) specify a single word transaction.
   - It asserts \( \text{ALE} \). An \( \text{ALE} \) signal can be used to latch the address.
   - It asserts \( \text{ADS} \).
   - It asserts \( \overline{\text{BE}}_3-\overline{\text{BE}}_0 \) to specify which bytes are used when reading the data word.
   - It brings \( \text{W/R} \) low to denote a read operation.
   - It brings \( \text{DT/R} \) signal low. \( \text{DT/R} \) can be used for the direction input to data transceivers.
2. During the $T_d$ state, several actions occur.
   - The 80960KB processor reads the data on the address/data lines.
   - The 80960KB processor asserts $DEN$. $DEN$ can be used to enable data transceivers.
     READY is asserted by external timing logic and data is transmitted from the storage devices. If READY is not asserted, the data transfer is delayed generating a $T_w$ state. The $T_w$ state is repeated, until READY is asserted.

3. The $T_r$ state follows the data state. This allows the system components adequate time (one processor clock cycle) to remove their outputs from the bus before the 80960KB processor generates the next address on the address/data lines. During the $T_r$ state $W/R$, $DT/R$, and $DEN$ become inactive.
3.4.3 Basic Write

Figure 9 shows a typical timing diagram for a basic write transaction with one wait state. Like the read transaction, a write operation may be preceded and succeeded by any type of bus transaction. The following sequence of events explains the flow of the timing diagram.

1. Similar to the read transaction, the 80960KB processor generates several signals during the $T_a$ state.
   - It transmits the address on the address/data lines. $LAD_1$ and $LAD_0$ specify a single word transaction.
   - It asserts $ALE$. An $ALE$ signal can be used to latch the address.
   - It asserts $ADS$.
   - It asserts $BE_3$-$BE_0$ to specify which bytes are used when writing the data word.
   - It brings $W/R$ high to denote a write operation.
   - It brings $DT/R$ signal high. $DT/R$ can be used for the direction input to data transceivers.
2. During the T_d state, several actions occur.
   - The 80960KB places the data on the address/data lines.
   - The 80960KB processor asserts DEN. DEN can be used to enable data transceivers.
   - READY is not asserted by external timing logic. Consequently, data is held on the LAD lines.

3. During the T_w state READY is asserted and the data is written to the storage device. Note that the W/R, DT/R and DEN remain constant until the bus state after READY is asserted.

4. The T_r state follows the wait state. During the T_r state W/R, DT/R, and DEN become inactive.

3.4.4. Burst

The 80960KB processor supports burst transactions that read or write up to four words at a maximum rate of one word every processor clock cycle. Burst transactions are always contained within a 16-byte boundary. If a transaction crosses a 16-byte boundary, the 80960KB processor automatically splits the transaction into two accesses.

The byte enable signals are valid for each word to allow partial-word write operations for a burst write transaction. The CACHE output signal during a T_a state applies to all words of a burst transaction.

A burst read or write transaction is similar to a basic read or write operation. It differs primarily in the number of data words transferred: the basic transaction always transfers one data word, the burst transaction transfers up to four data words. For a burst transaction, the byte enable signals are applied during the T_a state, and subsequently during every T_d or T_w state before the data word is transferred. Figure 10 shows the timing for a three-word burst read transaction without wait states. Figure 11 shows the timing for a two-word burst write transaction with a wait state occurring during the transfer of the first word. Note that the byte enable signals remain constant until the data state after READY is asserted.

3.5 TIMING GENERATION

In an 80960KB processor-based system, timing signals must be generated for the clock and reset inputs. To generate these signals, discrete logic should be utilized to minimize skew and maintain the rise and fall times as short as possible. This section describes a typical circuit that synthesizes the clock signal. The RESET timing generation is discussed in the "RESET and Initialization" section.

3.5.1 80960KB Processor Clock Requirements

In order to design a clock generator, the clock input specifications to the 80960KB processor are examined first. The clock (CLK2) waveform is shown in Figure 12. The clock pulse is specified by five parameters listed below:
The clock fall time \( (t_f) \)
- The clock low time \( (t_l) \)
- The clock rise time \( (t_r) \)
- The clock period \( (t_{cy}) \)

![Diagram of clock waveforms and transaction timing](image)

**Figure 10. 80960MC Processor Burst Read Transaction**

The time required to go from 90% of the difference between the high and low voltage levels to (10% of the difference (or from low to high)) is defined as the clock fall (rise) time. The clock low time specifies the time required for the clock to remain within 10% of the low voltage level. Similarly, the clock high time specifies the required time for the clock pulse to remain within 10% of the high voltage level. The clock period is the sum of \( t_f + t_l + t_r + t_h \).

The clock generator must have fast enough rise and fall times to comply with the requirements for high and low time and the overall clock period. For example, consider a clock pulse with a 50% duty cycle at 40 MHz. The clock period is specified at minimum of 25 ns, low time at minimum of 8 ns, and high time at minimum of 8 ns. This implies that the sum of the rise and fall time must not be greater than 9 ns. Thus, the clock generator should be designed to have rise and fall times not greater than 4.5 ns each.
Besides specifying a maximum clock rate, the 80960KB processor requires a minimum CLK2 rate of 8 MHz to maintain the state of the internal dynamic cells. Due to this minimum frequency requirement, the 80960KB processor cannot be single-stepped by disabling the clock.
3.5.2 Clock Generation

Figure 13 shows an example of a clock generator that produces two clock pulses, one double the frequency of the other with the skew between the pulses in the range of 1 to 3 ns. This particular circuit produces a 40-MHz clock at 50% duty cycle with rise and fall times of less than 4 ns. The circuit design consists of four devices: an oscillator, a pulse shaping network, a synchronous up/down counter, and a NAND gate driver. The output of the 80-MHz hybrid clock oscillator connects to the pulse shaping network (two NAND gates in series) which in turn feeds into the clock input of the up/down counter. This counter produces a 40-MHz CLK2 output signal and a 20-MHz CLK output signal. Because the outputs of the counter are synchronous, the skew between CLK2 and CLK is typically less than 2 ns. To provide adequate signal margin and maintain fast rise and fall times, the two clock signals are conditioned by the NAND gate driver. The timing waveforms of the clock circuit are shown in Figure 14.

**Figure 13. Clock Generation Circuit**

**Figure 14. Clock Timing Waveforms**
If the opposite phase CLK is preferred, U/D pin can be connected to \( V_{CC} \).

The hybrid clock oscillator typically requires 5 ms to stabilize after power is applied. The 80960KB processor cannot begin to execute instructions until after the clock and VCC have reached their DC and AC specifications. The RESET signal can be used to control the start of the CPU execution when power is applied. This is discussed in the “RESET and Initialization” section.

### 3.6 Arbitration

When multiple bus masters exist, an arbitration protocol is used to exchange control of the bus. The protocol assumes that there are two bus masters: one that controls the bus by default, and the other that requests control of the bus when it performs an operation, such as a DMA controller. More than two bus masters may exist on the L-bus, but this requires external arbitration logic. There should be no more than two 80960KB processors, however, on an L-bus.

Assuming that there are only two bus masters, this section examines the bus arbitration, bus states, and timing diagrams for different combinations of bus masters, as shown in Table 4.

#### Table 4. Combination of Bus Masters

<table>
<thead>
<tr>
<th>Bus Master Combination</th>
<th>Bus Master that Requests Control of the Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE 1</td>
<td>80960KB PROCESSOR</td>
</tr>
<tr>
<td></td>
<td>I/O DEVICE</td>
</tr>
<tr>
<td>CASE 2</td>
<td>80960KB PROCESSOR</td>
</tr>
<tr>
<td></td>
<td>80960KB PROCESSOR</td>
</tr>
<tr>
<td>CASE 3</td>
<td>I/O DEVICE</td>
</tr>
<tr>
<td></td>
<td>80960KB PROCESSOR</td>
</tr>
</tbody>
</table>

#### 3.6.1 Single 80960KB Processor on the L-Bus

For the first case, the 80960KB processor controls the L-bus, and a master I/O peripheral, such as a DMA controller, requests control of the bus for operations. The 80960KB processor and the I/O peripheral exchange control of the bus with two signals: the hold request (HOLD) and hold acknowledge (HLDA) signals.

HOLD is an input signal of the 80960KB processor, which indicates that the master I/O peripheral is requesting control of the L-bus. When HOLD is asserted, the 80960KB processor surrenders control of the bus after it completes the current bus transaction. The processor acknowledges transfer of control of the L-bus to the other bus master by asserting the HLDA.

#### 3.6.2 State Diagram

Figure 15 shows the state diagram for a L-bus with an I/O peripheral bus master. This state diagram consists of the hold state \( T_h \) addition to the five basic states described in the “Basic L-Bus State”
section. The 80960KB processor enters the \( T_h \) state when it surrenders the control of the bus. It can enter the \( T_h \) state from the \( T_i \) or \( T_r \) state. When the 80960KB processor regains control of the L-bus, it enters the \( T_h \) state if a new request is pending or a \( T_i \) state if no new request is pending.

Figure 3-15. L-Bus States with Arbitration
3.6.3 Arbitration Timing

Figure 16 shows the arbitration timing diagram. The “T” state represents the last cycle of a transaction in which the READY signal was asserted or a T₁ state. The 80960KB processor receives a request to relinquish control of the bus when HOLD is asserted. After the 80960KB processor completes the current transaction, it responds to this request by floating the three-state output signals and deasserting the open drain output signals. The HLDA output signal, however, remains active and is asserted as the 80960KB processor enters a T₃ state. During the T₃ state, the CPU ignores all input signals except HOLD and RESET. When the HOLD input signal is deasserted, the 80960KB processor exits the T₃ state and deasserts HLDA.
3.6.4 Two 80960KB Processors on the L-Bus

For the next case, two 80960KB processors reside on the L-bus. During initialization, one is designated as the Primary Bus Master (PBM), the other as the Secondary Bus Master (SBM).

The exchange protocol that is used guarantees that neither device is kept off the bus indefinitely. The 80960KB processors use two pins for bus arbitration: the HOLD input pin, and the HLDA output pins. These input and output pins for the SBM are interpreted differently, however.

When the SBM is initialized, the pin normally used for HOLD input signal is interpreted as the hold acknowledge request (HLDAR) input signal. The assertion of 45 HLDAR indicates that the PBM relinquished control of the L-bus. Similarly, the HLDA output signal of the SBM is interpreted as the hold request (HOLDR) output signal. The SBM asserts HOLDR to request acquisition of the L-bus. Thus, bus arbitration between two 80960KB processors can be accomplished by connecting HOLD of the PBM to HOLDR of the SBM, and HLDA of the PBM to the HLDAR of the SBM, as shown in Figure 17.

When using the connection shown in Figure 17, a delay must be inserted between the input and output signals because the minimum clock-to-output delay is less than the maximum hold time of the input signals. The delay time must be greater than 5 ns, but less than the clock period minus the setup time minus the maximum clock-to-output delay (5ns ≤ Delay ≤ T_{Period} - T_{Setup} - T_{Clock-To-Output}).

3.6.5 Bus states for Two 80960KB Processors

The state diagram for the SBM is shown in Figure 18. Because there are two 80960KB processors, the LOCK signal is included in the state diagram. The SBM requests control of the L-bus by asserting HOLDR and subsequently enters the hold request (T_{hr}) state provided that the bus is not locked (locked means that LOCK is asserted by the PBM and the SBM has a RMW operation pending). The SBM remains in the T_{hr} state until it acquires control of the L-bus by receiving HLDAR. The SBM returns to the T_{i} state by deasserting HOLDR provided that the following two conditions exist:

- A RMW operation is pending
- The PBM asserted LOCK while the SBM was in the T_{hr} state.

The SBM gains control of the bus when HLDAR is asserted provided that the bus is not locked. After gaining control of the L-bus, the SBM performs the operations, and enters a T_{w} state if necessary. At the end of a transaction, the SBM goes to the T_{state} and deasserts HOLDR for at least one processor clock cycle to allow another peripheral bus master to gain access if needed. If another request is pending, the SBM enters the T_{hr} state and asserts HOLDR provided the bus is not locked. The PBM never forces the SBM off the bus.

3.6.6 Arbitration Timing for Two 80960KB Processors on the L-Bus

Figure 19 shows the timing diagram for acquiring and relinquishing the L-bus by an SBM. The SBM enters into the Hold Request (T_{hr}) state and asserts the HOLDR signal. It remains in the T_{hr} state until HLDAR is asserted, which indicates that the SBM can utilize the L-bus during the next state. When
the bus is no longer required, HOLDR is deasserted during the state following the last READY signal. Except for HOLDR, the output signals of the SBM go into a high impedance state or are deasserted for the case of open-drain outputs.

Figure 18. L-Bus States For Secondary Bus Master
3.6.7 Bus Exchange Example Between Two 80960KB Processors

Figure 20 shows an example of bus arbitration between a PBM and an SBM using the arbitration signals. Each bus master performs a one-word read and a two-word write transaction to demonstrate the fastest possible bus exchanges.

While the PBM is performing a read transaction, the SBM requests control of the L-bus by asserting HOLDR and entering the T_{hr} state. It remains in this state until the PBM grants the request by asserting HLDA after the read transaction is completed. After granting the request, the PBM enters the T_{th} state and remains in this state until its HOLD signal is deasserted. When the SBM completes the read transaction, it deasserts HOLDR and gives control back to the PBM.

The PBM now performs a two word write transaction after deasserting the HLDA. The SBM requests control of the bus again by asserting the HOLDR signal and enters the T_{hr} state. When the PBM completes the two-word write transaction, it grants the request by asserting HLDA and enters the T_{th} state. The SBM receives the signal on the HLDAR input and performs a two-word write transaction. When the SBM completes the transaction, the control of the L-bus is transferred to the PBM, and both the PBM and the SBM enter the T_{th} state.
3.6.8 A Peripheral Device As the Default Bus Master

Another case exists where a peripheral device controls the L-bus, and the 80960KB processor requests control of the bus to perform operations. This alternative is not advisable because it hinders system performance. The exchange protocol is identical to the one described in the previous section. The 80960KB processor is an SBM and uses two pins for bus arbitration: the HOLDR input pin and the HLDAR output pin. The state diagram is similar to the one shown in Figure 18. The lock conditions are not used for this case, however.

The peripheral device grants control of the L-bus bus asserting HLDAR when the SBM requests use of the L-bus. The peripheral device can obtain control of the L-bus again by deasserting HLDAR. If this occurs, the 80960KB processor surrenders control of the bus after it completes the current transaction, as shown in Figure 21. At that time, the 80960KB processor deasserts the HOLDR signal and places the other output signals into a high impedance state or a deasserted open drain level. The 80960KB processor may request access to the L-bus by asserting HOLDR again.

3.7 INTER-AGENT COMMUNICATION (IAC)

The IAC mechanism gives 80960KB processors the capability to send and receive messages to one another and to other bus agents. The IAC mechanism is essentially a non-maskable interrupt with pre-defined service routines. These routines are implemented in the 80960KB processor and are used to
perform control functions such as purging the instruction cache, setting breakpoint registers, or stopping and starting the processor. By using IAC messages, external agents can remotely control the 80960KB. This allows easy integration of the 80960KB into system environments.

IAC messages can also be used to generate interrupts that behave exactly the same as hardwired interrupts. Since the interrupt vector is encoded in the IAC message, any of the possible interrupt service routines can be invoked.

![Figure 21. Forced Relinquishment Timing Diagram for an SMB](image)

### 3.7.1 Overview of IAC Operation

Figure 22 shows a typical example of an IAC operation. In this case, an external processor gains control of the 80960KB by using an IAC operation. The external processor performs two functions: it writes the message in a buffer, called the message buffer; and it asserts the IAC pin of the 80960KB processor. Upon receipt of the IAC signal, the 80960KB processor stops executing its current process and performs a four-word read of the message buffer. After completing the read operation, the 80960KB processor automatically performs a one-word write operation to a pre-defined address to acknowledge the receipt of the message. The 80960KB processor then proceeds to perform the required action.

### 3.7.2 IAC Messages

The IAC messages are specifically defined and behave much like machine instructions. The 80960KB processor reserves the upper 16M bytes (FF000000H to FFFFFFFFH) of the 4M-byte address range for IAC message operations.
There are two types of IAC messages: internal and external. Internal IAC messages allow a program to send a command to its own processor. An internal IAC message is sent by writing to address FF000010H. Internal IAC messages cause no L-bus activity.

External IAC messages can be used to send a command to another processor on the L-bus or to a remote processor. A processor sends an external IAC message by writing to a buffer area and causing the IAC pin of the receiving 80960KB to be asserted.

When the IAC pin is asserted, the recipient processor reads the reserved address to fetch the data from its IAC message buffer. After reading the IAC message buffer, the recipient does a write operation to another reserved address to acknowledge receipt of the IAC message. The IAC pin is deasserted as a result of this write operation, and the processor is ready to receive another IAC.

3.7.3 Hardware Requirements for External IAC Messages

To use the external IAC feature of the 80960KB, the following items are needed: a four-word message buffer RAM mapped to a reserved address to store the message, logic to assert the IAC pin of the 80960KB, and decoding logic to deassert the IAC pin on command from the 80960KB.

3.7.4 Message Buffers

Each 80960KB processor that receives an IAC message must have four 32-bit words of message buffer. This buffer can use special hardware or a reserved area in RAM. For proper operation of the
buffer, two requirements must be met: the receiving 80960KB must be able to read this buffer at FF000010₉ if the receiving 80960KB's Local Processor Number (LPN) is equal to zero (see the "RESET and Initialization" section for details of the LPN), or at FF000030₉ if the LPN is equal to one; and the sending processor must be able to write this buffer.

### 3.7.5 IAC Pin Logic

When the IAC message buffer receives a message, logic asserts the IAC pin and keeps it asserted. After the 80960KB processor reads the IAC message, it performs a one-word write to address FF000000 if its LPN is zero, or FF000020 if its LPN is one. This reserved address serves two functions: it causes external logic to deassert the IAC pin, and it maps to a register that contains the current processor priority. If the low order three bits of the data word have a value of 100₅ (see Figure 23), the external logic should deassert the IAC pin on completion of the write operation.

![Figure 23. Data Settings](image)

### 3.8 EXTERNAL PRIORITY REGISTER

The 80960KB contains an internal register that keeps track of the current priority (a value between 0 and 31) at which it is executing. This priority is used to decide whether or not to service interrupts — higher priority interrupts are serviced, others are posted for later servicing. In some system designs it may be desirable to have this priority visible outside of the processor. To allow this, the 80960KB provides support for an external priority register. Whenever the priority of the 80960KB changes, the contents of this register are automatically updated.

This feature may be enabled in two steps. If the Write External Priority bit is set in the PRCB (see the 80960KB CPU Programmer's Reference Manual), then the external priority register is updated as a result of a MODPC instruction or whenever an interrupt occurs. If external IAC messages are enabled, then external priority is also updated whenever a result of an IAC is to change processor priority.

### 3.8.1 Hardware Requirements

The 80960KB expects to write its priority into a 5-bit register mapped to address FF000000 if its LPN is zero, or FF000020 if its LPN is one. To set the priority, the processor performs a one-word write operation in the form shown in Figure 23. The priority is contained in bit₂₀-bit₁₆, and bit₃ is
asserted to indicate that the priority is changed. It is necessary to use bit, as a qualifier to distinguish priority write operations from IAC message acknowledgments, which use the same reserved address.

### 3.8.2 External Priority and IAC Messages

The external priority register can be used to filter IAC messages. Since the processor always services the IAC pin (i.e., it is non-maskable), a low priority IAC message can interrupt a high IAC priority task. To prevent this, a system can associate a priority with each IAC message. This priority can then be compared to the priority stored in the external priority register and used to decide whether or not to accept the IAC message. One way to associate a priority with an IAC message is to encode the message priority into the IAC message destination address as shown in Figure 24. The range of reserved addresses shown in Figure 24 have been set aside for this purpose.

![Figure 24. Physical Address Interpretation for IAC Messages](image)

### 3.9 INTERRUPTS

The 80960KB processor responds to external events occurring at arbitrary times by means of an interrupt signal. Various sources, which include hardware components and special software instructions, generate an interrupt signal that can suspend execution of the 80960KB processor’s current instruction stream. The hardware-generated interrupts are discussed in this section. For complete information on software-generated interrupts, see the Programmer’s Reference Chapter of this handbook.

The 80960KB is unusual in that the interrupt controller automatically does the processor housekeeping tasks that are normally left for the programmer to deal with in the interrupt handling routine. The local registers are pushed onto the stack, state is saved, arithmetic controls are saved, priority of the processor is changed to the interrupt priority, and stack pointers are managed. All this is done automatically before entering the user written interrupt routine. The bottom line of this is that the programmer can simply worry about the function of the interrupt handling routine and not processor housekeeping, thus greatly simplifying the programming and debugging effort.

The 80960KB processor provides a flexible interrupt structure. The 80960KB processor can be interrupted using any of three methods below:
• Receipt of a signal on any or all of the four direct interrupt input signals (INT₀, INT₁, INT₂, and INT₃)
• Receipt of a signal on the interrupt request (INTR) line to obtain an external interrupt vector
• Receipt of an IAC message from a processor program or external source.

The choice of the method is determined by the setting in the on-chip Interrupt Control register. Interrupt signals can occur during any bus state regardless of which method is implemented.

This section provides details on the multiplexed interrupt pins, the three interrupt methods, the Interrupt Control register, synchronization, and interrupt latency.

### 3.9.1 Interrupt Signals

The interrupt signals are multiplexed on four pins of the 80960KB processor: INT₀/IAC, INT₁, INT₂/INTR, and INT₃/INTA. The on-chip Interrupt Control register determines how these pins are used (see "Interrupt Control Register" section).

**INT₀/IAC**

This pin multiplexes the **Interrupt₀** and **Inter-Agent Communication** request input signals. The 80960KB processor interprets this input signal as either INT₀ or IAC. The IAC signal indicates a request for interrupt service when it is asserted. The IAC signal denotes that a message is waiting when it is asserted.

**INT₁**

The **Interrupt₁** input signal indicates a request for interrupt service when it is asserted.

**INT₂/INTR**

This pin multiplexes the **Interrupt₂** and **Interrupt Request** input signals. The 80960KB processor interprets this input signal as either INT₂ or INTR. The INT₂ signal indicates a request for interrupt service when it is asserted. The INTR signal indicates an interrupt request from an external interrupt controller. The 80960KB processor responds with an interrupt-acknowledge sequence. To ensure an interrupt, the INTR signal must remain asserted until the first cycle of the interrupt-acknowledge transaction.

**INT₃/INTA**

This pin multiplexes the **Interrupt₃** input signal and **Interrupt Acknowledge** output signal. The 80960KB processor uses this pin as the INT₃ input signal or as the INTA output signal. The Interrupt Control register setting selects either the combination of INTR/INTA or INT₂/INT₃. The INT₃ input signal indicates a request for interrupt service when it is asserted. INTA acknowledges the interrupt request from an external interrupt controller. The INTA signal is latched by the 80960KB processor and remains valid during the T₄ state. This signal is open drain output.
3.9.2 Interrupt Control Register

The 80960KB processor uses a 32-bit, on-chip Interrupt Control register to define the function of the multiplexed interrupt pins. This 32-bit Interrupt Control register allocates eight bits for each of the four direct interrupt signals (INT₀, INT₁, INT₂, and INT₃). The eight bits contain the vector number for each interrupt signal, as shown in Figure 25. The vector number is automatically read when one of the interrupt signals (INT₀, INT₁, INT₂, and INT₃) is activated. For example, when an interrupt is signaled on INT₀, the 80960KB processor uses bit₁₇-bit₀ of the Interrupt Control register as the vector number.

The 80960KB processor uses the data field corresponding to INT₀ to determine identification of the INT₀/IAC input pin; a value of 0₀_H signifies the IAC function. If the data field corresponding to INT₂ has a value of 0₀_H, the 80960KB processor interprets the INT₂/INTR pin as the INTR input signal, and the INT₄/IINT₄ pin as the IINT₄ output signal. In other words, this setting specifies that the 80960KB processor should use these two pins for communication with an external interrupt controller. If the functions of INTR and IINT₄ are selected, the direct interrupt pins (INT₀ and INT₁) can still be used.

The on-chip Interrupt Control register may be read and written by the Synchronous Load (synld) and Synchronous Move (synmov) instructions at the address FF000004_H (see the 80960KB Programmer’s Reference Manual). The value of the data fields in the Interrupt Control register is FF000000_H after initialization. This setting specifies that the four interrupt pins function as IINT₄, INTR, INT₁ and IAC.

3.9.3 Using the Four Direct Interrupt Pins

The 80960KB processor can be interrupted by asserting any or all of the four interrupt input signals (INT₀, INT₁, INT₂, INT₃). If the signals are simultaneously asserted, the 80960KB assumes that INT₀ has the highest priority, followed by INT₁, INT₂, and INT₃. Software should follow this convention when programming the Interrupt Control register. When the interrupt input signals are asserted, the 80960KB processor utilizes a vector number specified by the Interrupt Control register as an index to an entry in the interrupt table located in memory. For complete software information on this topic, see the Programmer’s Reference Chapter of this handbook.
3.9.4 Using an External Interrupt Controller

The 80960KB processor can communicate with an external interrupt controller by performing an interrupt acknowledge sequence using the INTR and INTA signals. Figure 26 shows an example of the timing of an interrupt acknowledge sequence using the 8259A Programmable Interrupt Controller.

INTR is asserted by the 8259A and remains asserted until the 80960KB processor activates the INTA signal for the first time. When the 80960KB processor receives an interrupt request, the CPU completes the current transaction (or comes to some interruptible point), and asserts INTA. INTA remains valid through the $T_a$, $T_d$, and $T_e$ states. The first assertion of INTA triggers the 8259A to resolve priority among its interrupt requests.

To compensate for the timing of the 8259A, the 80960KB processor automatically inserts five $T_i$ states before asserting the INTA again to read the interrupt vector. Figure 26 shows READY asserted without a wait state during the first Interrupt Acknowledgement cycle and with one wait state during the second Interrupt Acknowledgement cycle. In practice, the 8259A would require about four wait
states in both cycles. The address during the \( T_a \) state for both interrupt acknowledge cycles is \( \text{FFFFF} \text{FFC} \). For more details, see the “8259A Programmable Interrupt Controller” portion in Section 5 of this chapter.

The 80960KB processor services the interrupt according to its priority. If the interrupt has higher priority than the current activity, the 80960KB processor services it immediately. Otherwise, after reading the interrupt vector, the 80960KB processor posts the interrupt vector in the interrupt table. Typically, the 80960KB processor responds within 4 usec for an interrupt with higher priority than the current process (assuming CLK2 at 40 MHz). If the interrupt has lower priority than the current activity, the interrupt is serviced when its priority is higher than the priority of the subsequent activity of the 80960KB processor.

3.9.5 Using IAC Requests for Interrupts

The 80960KB processor can also be interrupted by an IAC message. The 80960KB processor can send IAC messages to itself by using one of the Synchronous Move instructions. Because this message does not utilize the L-bus when sent to the same processor, no special hardware is required. More details are provided in the Programmer’s Reference Chapter of this manual.

3.9.6 Synchronization

The \( \text{INT} \text{/IAC} \), \( \text{INT} \text{, INT} \text{/INTR} \) and \( \text{INT} \text{_3} \) input signals can be either synchronous or asynchronous to the system clock (CLK2). Synchronous interrupt signals must be set up 3 ns prior to the rising edge of CLK2 and held for 10 ns after the rising edge of CLK2. To properly preset the interrupt signals for synchronous operation, \( \text{INT} \text{/IAC} \), \( \text{INT} \text{, INT} \text{/INTR} \) and \( \text{INT} \text{_3} \) must be deasserted for at least one processor clock cycle and asserted for at least one processor clock cycle. These signals may be deasserted and asserted individually.

If the interrupt signals are asynchronous to CLK2, the 80960KB processor internally synchronizes them. For the CPU to recognize the asynchronous interrupt input signals, they must be preset by deasserting them for at least two processor clock cycles, and then asserting them for at least two processor clock cycles.

3.9.7 Interrupt Flows

These signals may be deasserted and asserted individually. The 80960 interrupt controller intelligently manages interrupts. Once an interrupt is signalled, the 80960KB interrupt mechanism transfers control to a microcode interrupt routine. This 80960KB routine automatically allocates a new set of local registers onto the stack, posts pending interrupts, checks priorities, and suspends or aborts long instructions before executing the user’s interrupt handler. Once the interrupt handler has completed, the return instruction “knows” it is a return from interrupt and the 80960KB return routine restores the local registers, arithmetic, and process control registers, checks for pending interrupts, and returns to the next instruction of the interrupted code.
There are two main stages the 80960KB goes through before it executes the interrupt handler: hardware recognizes the interrupt and then a microcode interrupt routine executes. First the interrupt pin is pulled. Hardware stores this in a four-bit register. One bit is assigned to each pin. This register is used to capture subsequent interrupts once one interrupt has been recognized. Interrupts are recognized at instruction boundaries or interruptable points in long instructions (floating point). They are then immediately disabled. However, it is important to note that disabling interrupts does not disable the four-bit register. Interrupts are saved in this register until microcode reaches a point it can check the register again. When the register is read it is subsequently cleared. The highest priority bit in the four-bit vector is cleared, which indicates that the interrupt vector associated with it will be used. Then this vector is written back to the register by an ORing function with the register thus maintaining any new interrupts that may have been signalled.

Next the 80960KB recognizes that an interrupt occurred by the fact that an interrupt event has been stored in the four-bit register. At this point the interrupt microcode routine is called by a hardware mechanism in the interrupt controller. The interrupt routine executes the action described by the interrupt flow in Flow Chart 1. After the interrupt routine has completed, it “calls” the interrupt handler and commences executing instructions. The interrupt handler is user supplied. All the housekeeping needed to get into and out of the interrupt handler is completed by the 80960KB microcode interrupt routine before the interrupt handler is “called”. No processor housekeeping activities need to be done by the user’s interrupt handler.

The 80960KB has only one “return” instruction for all types of returns. There are three bits in the “previous frame pointer” (local register0) called the return status bit. See section 4 of the Programmer’s Reference Chapter of this manual. These bits have encoded in them the type of call and, therefore, the type of return that is to occur. The 80960KB manages this completely.

The flow diagrams show an interrupt flow, pending interrupt flow and interrupt return flow. Each of these are implemented as microcode routines in the hardware of the 80960KB.

3.9.8 Pending Interrupts

Pending interrupts are checked in certain situations. If a pending interrupt exists then the “pending interrupt” flow is executed. The four situations that pending interrupts are checked are as follows:

- Return form interrupt
- OR-
- MODPC instruction (if process priority is lowered)
- OR-
- Test pending interrupt IAC is executed
PENDING INTERRUPTS

HARDWARE REFERENCE

HARDWARE RECOGNITION
STOP INSTRUCTION FLOW
READ INTERRUPT VECTOR NUMBER

LESS THAN OR EQUAL TO

PRIORITY?

GREATER THAN

PROCESSOR IN INTERRUPTED STATE?

Y

STACK POINTER <- CURRENT STACK POINTER

READ INTERRUPT TABLE ENTRY
STATE <- INTERRUPTED
MODE <- SUPERVISOR
PROCESS PRIORITY <- INTERRUPT PRIORITY
CLEAR TRACE ENABLES

PROCESSOR STATE IDLE?

Y (MC ONLY)

SET RETURN STATUS BITS TO 'IDLE INTERRUPT'

N (KA AND KB)

SET RETURN STATUS BITS TO INTERRUPTED

SUSPENDED INSTRUCTION?

Y

SET RESUME BIT IN PCW
PUSH RESUMPTION RECORD TO THE STACK

N

PUSHONTO STACK:
INTERRUPT RECORD
PROCESS CONTROL WORD
ARITHMETIC CONTROLS

FP <- ROUNDUP(SP) + 64
CALL INTERRUPT HANDLER
NEW SP <- FP + 64

START FIRST INSTRUCTION
OF INTERRUPT HANDLER

Flowchart 1

2-39
GET HIGHEST PENDING PRIORITY INTERRUPT

IF ZERO

LESS

GREATER

READ PENDING INTERRUPT BYTE

CHECK PRIORITY

Y

CLEAR PENDING PRIORITY

N

CLEAR PENDING INTERRUPTS CLEAR PENDING PRIORITY BIT

CONTINUE INTERRUPT FLOW AT PENDING INTERRUPT

Flowchart 2
3.9.9 Interrupt Latency

The 80960KB interrupt controller manages the interrupt mechanism automatically and therefore there are many cases it deals with. Depending on the situation, latency may vary. The 80960KB’s interrupt latencies are comprised of a base latency and special case latencies added to it. These special cases consist of such things as using an 8259A interrupt controller, the local register cache being full, or an interrupt occurring while the processor is already in the interrupted state.

The base interrupt latency is 85 cycles as shown in Table 5. Table 6 describes the breakdown of the base interrupt latency. Notice that only takes 6 cycles for the 80960KB to respond to the interrupt. Four cycles for hardware recognition of the interrupt and a minimum of one cycle to respond if the interrupt occurs on an instruction boundary. The table indicates two cycles and assumes the interrupt is signalled at the beginning of a RISC instruction. This value will differ depending on the instruction being interrupted and the point at which the interrupt is signalled in the instruction. Table 7 gives values for integer execution, floating point, and transcendental floating point instruction interrupt boundaries.
Table 5. Interrupt Latencies

<table>
<thead>
<tr>
<th>Type of Latency</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Interrupt Latency</td>
<td>85</td>
</tr>
<tr>
<td>Return</td>
<td>78</td>
</tr>
<tr>
<td>Interrupt immediately followed by another interrupt. Second interrupt posted to interrupt table.</td>
<td>157</td>
</tr>
<tr>
<td>Return with a Pending Interrupt Posted</td>
<td>157</td>
</tr>
<tr>
<td>Pending Interrupt</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6. Constituent Parts of the Base Interrupt Latency.
(The total base interrupt latency is 85 cycles or 4.25 us.)

<table>
<thead>
<tr>
<th>Constituent Latencies</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Recognition</td>
<td>4</td>
</tr>
<tr>
<td>Stop Current Instruction Flow Assuming a Risc Instruction</td>
<td>2</td>
</tr>
<tr>
<td>Determine Next IP and Save</td>
<td>8</td>
</tr>
<tr>
<td>Read Interrupt Vector Number</td>
<td>18</td>
</tr>
<tr>
<td>Check Interrupt Priority</td>
<td>8</td>
</tr>
<tr>
<td>Read Interrupt Table Vector</td>
<td>14</td>
</tr>
<tr>
<td>Check if Processor Already Interrupted</td>
<td>6</td>
</tr>
<tr>
<td>Save Process Control and Write Interrupt Record</td>
<td>10</td>
</tr>
<tr>
<td>Compute Interrupt Record Address of New Local Register Set</td>
<td>10</td>
</tr>
<tr>
<td>Allocate New Local Register Set</td>
<td>3</td>
</tr>
<tr>
<td>Fetch New Instruction and Start Decoding</td>
<td>2</td>
</tr>
</tbody>
</table>

Other situations that add to the latency are interrupts signalled at the start of a multicycle instruction or multiple interrupts signalled at the same time. The first may cause a resumption record to be stored on the stack. This records all the necessary information the 80960KB needs to resume executing the interrupted instruction. Not all interruptable instructions cause a resumption record to be created. If an instruction has been executing for over approximately 520 cycles then a resumption record will be created. Less than that and the instruction is simply restarted upon return from the interrupt. This was an engineering trade-off between the overhead to save state after less than 520 cycles and restarting the instruction. Restarting the instruction requires fewer cycles for most cases.
Table 7. Special Case Latencies that are Added to the Base Latency

<table>
<thead>
<tr>
<th>Special Case Latencies</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>8259A Interrupt Expansion (4ws)</td>
<td>18</td>
</tr>
<tr>
<td>Frame Cache Full</td>
<td>24</td>
</tr>
<tr>
<td>Current Process in “Interrupt”</td>
<td>14</td>
</tr>
<tr>
<td>Risc Instructions (Worst Case)</td>
<td>3–4</td>
</tr>
<tr>
<td>Integer Execution</td>
<td>10–40</td>
</tr>
<tr>
<td>Floating Point</td>
<td>12–96</td>
</tr>
<tr>
<td>Transcendental Floating Point</td>
<td>90</td>
</tr>
<tr>
<td>Instruction Cache Miss (2 Wait State)</td>
<td>5</td>
</tr>
</tbody>
</table>

Multiple interrupts signalled at various times are handled on a first come first serve basis. Interrupts occurring at the same time are handled on a priority scheme with INT3<INT2<INT1<INT0. The first interrupt is handled as soon as the 80960KB reaches an interruptable state (e.g. end of instruction) and subsequent interrupts are read from the interrupt control register and posted in the interrupt table as soon as the microcode routine reinables interrupts. While interrupts are not enabled the event (another interrupt) is stored in the four-bit register described earlier. Posting a pending interrupt to the interrupt table adds about 60 cycles to the interrupt latency. This consists of comparing the priorities of the processor and interrupt, writing a “one” to the appropriate bits in the pending priorities field, if it is less than or equal to the current priority, and writing a “one” to the appropriate bits in the pending interrupt field in the interrupt table. The positions in the fields are pointed to by the index vector from the interrupt control register or an 8259A vector.

The minimum interrupt latency is 85 clocks or 4.25 usec at 20MHz. This latency assumes the instruction handler is in the cache. If there is an instruction cache miss, five clocks for caching the instructions must be added to the base latency (assuming a two wait state memory system). In most cases the instruction will be cached already. A program’s typical latency would add about 3 more clocks for non-RISC instructions. If there is a local register cache miss then 24 cycles or 1.2 usec should be added. The worst case interrupt latency would be 181 cycles or 9.05 usec. This assumes the interrupt is signalled at the beginning of an ediv instruction (40 cycles), there is a local register cache miss (24 cycles), the current process is in the “interrupt” state (14 cycles), and an 8259A with 4 wait states is being used (18 cycles).

It is important to note that during the microcode routine all of the stack manipulations, saving state, checking priorities, and allocating new registers is done automatically. When the 80960KB enters the user interrupt handler this routine does not have to do any housework, it can start immediately with useful code. The benefit is that this work is done by the processor in microcode and can be done quickly and efficiently. Also note that the 80960KB responds to an interrupt in as little as 6 clocks. This is from the point of interrupt pin assertion to the point that the instruction flow is stopped and the microcode routine starts the housekeeping tasks. Normally processors do not include any of the
housekeeping activities in the interrupt latency so care should be taken in comparing latencies.

Table 7 gives the latencies based on special cases that may occur. These values must be added to the base latency from Table 5.

For more details on interrupts see section 8 of the Programmer’s Reference chapter of this handbook.

3.10 RESET AND INITIALIZATION

The system RESET signal provides an orderly way to start or restart the 80960KB processor. When the 80960KB processor detects the low-to-high transition of RESET, it terminates all external activities and places the output pins in the high impedance state or deasserted condition. When the RESET signal falls low again, the 80960KB processor begins the initialization process and later starts fetching instructions from a specific address.

3.10.1 RESET Timing Requirements

To properly reset the 80960KB processor to a known state, the low-to-high transition of RESET must be asserted relative to any rising edge of CLK2 and remain asserted for at least 41 CLK2 cycles, as shown in Figure 27. RESET must be deasserted after the rising edge of CLK2, but prior to the next rising edge of CLK2. This establishes the next rising edge of CLK2 as edge A.

![Figure 27. RESET Timing Diagram](270647-38)

3.10.2 RESET Timing Generation

The RESET input signal to the 80960KB processor can easily be generated by implementing a synchronization circuit comprised of a two D-type flip-flops, as shown in Figure 28.

The user RESET signal is synchronized with the CLK signal by applying CLK to the clock input of both flip-flops. To protect against a metastable RESET signal, the output of the first flip-flop, SYNC, is applied to the input of the second flip-flop. The output of the second flip-flop results in a processor RESET signal. The timing diagram for these signals is shown in Figure 29. CLK or CLK2 can be used instead of CLK in Figure 29. Using CLK provides an edge A corresponding to the rising edge of CLK.
This circuit assumes an asynchronous user RESET signal. If the user RESET signal is already synchronous with the CLK signal, the same circuitry can be implemented as shown in Figure 30. In this case, however, the output from the first flip-flop is used to generate the processor RESET signal rather than being routed to the input of the second flip-flop.

3.10.3 Initialization

The initialization sequence of events is shown in Figure 31. When RESET is deasserted after a minimum of 41 CLK2 cycles, several actions take place: two input pins are sampled, the FAILURE output signal (see next 2 section for the pin description) is asserted, and the self-test is performed.
Figure 31. Initialization Flow Chart

READ INITIALIZATION PARAMETERS

DEASSERT RESET

SAMPLE LPN AND START-UP INPUT PINS

ASSERT FAILURE

PERFORM SELF-TEST

PERFORM SELF-TEST

ERROR?

YES → ENTER STOPPED STATE

NO

DEASSERT FAILURE

PERFORM SYSTEM CHECK

INITIALIZATION PROCESSOR

NO → ENTER STOPPED STATE

YES

READ 8 WORDS FROM PHYSICAL ADDRESS 0

PERFORM CHECKSUM ON THE 8 WORDS

CHECKSUM CORRECT?

NO → ASSERT FAILURE

YES → ENTER STOPPED STATE

PREPARE FOR OPERATION

SET UP FOR THE FIRST INSTRUCTION

CLEAR ANY LATCHED INTERRUPT SIGNALS

EXECUTE FIRST INSTRUCTION
When RESET is deasserted, the 80960KB processor samples the signals residing on the INTO/IAC and the BADAC pins (see the next section for the pin description of BADAC). At this time, these pins are interpreted as the Local Processor Number (LPN) and Startup (STARTUP) signals, respectively. The LPN input signal defines whether the 80960KB processor is a PBM (high voltage input level) or a SBM (low voltage input level). The STARTUP input pin indicates whether the 80960KB processor performs initialization (high voltage level) or not (low voltage level). The STARTUP signal is used to allow one or more processors to perform the active initialization. The input voltage levels for the LPN and STARTUP must be setup 3 ns before the rising CLK2 edge prior to edge A and held 10 ns beyond edge C, as shown in Figure 32.

Besides sampling the two input pins, the 80960KB processor asserts the FAILURE output signal a few cycles after RESET is deasserted. The FAILURE signal remains asserted while the CPU performs the self-test. If a failure is detected during the self-test, FAILURE remains asserted and the CPU enters the stopped state where the processor does nothing. If the self-test completes successfully, the CPU deasserts the FAILURE signal.

An 80960KB processor that is designated as the initialization processor proceeds by doing a checksum test of eight words fetched from memory at physical address 0000 0000H to ensure that the memory and L-bus are operating properly. If the initial checksum is incorrect, then the FAILURE signal is asserted (and remains asserted) and the 80960KB processor enters the stopped state. After a successful checksum test, the 80960KB processor uses some of the words as addresses to initialize data structures. Complete details are provided in the Programmer’s Reference chapter.

Just prior to executing the first instruction, the 80960KB processor clears any latched interrupt signals.
3.11 ERROR SIGNALS

The 80960KB processor provides an input signal (BADAC) for notification of an error in the system, and provides an output signal (FAILURE) for notification of an error within the processor.

**BADAC** When asserted, the **Bad Access** input signal indicates that an unrecoverable error occurred during the current data transfer. If, however, BADAC was asserted after a Synchronous Move or Synchronous Load instruction, the error is recoverable. The 80960KB processor samples the BADAC input signal during the cycle following the one when the last READY is asserted.

**FAILURE** The **FAILURE** signal indicates that an error occurred during initialization. The 80960KB processor always asserts FAILURE after the activation of the RESET signal. If a failure is detected during a self-test, FAILURE remains asserted. Otherwise, the processor deasserts FAILURE after a successful self-test is performed. If the initial memory checksum is incorrect, the initialization processor asserts FAILURE a second time, and keeps it asserted. FAILURE is an open drain output signal.

3.12 SUMMARY

The L-bus is a high speed 32-bit multiplexed bus with burst-transfer capability and is designed to operate with the high performance 80960KB processor. The L-bus consists of two signal groups: address/data, and control. These signal groups are utilized by the 80960KB processor to perform read, write, and burst transactions.

The arbitration, interrupt, and reset operations are related to the L-bus transactions. The arbitration operation transfers control of the L-bus to another bus master. Three methods are available to handle interrupts: by invoking the on-chip interrupt controller, by employing an external interrupt controller using the INTR/INTA signals, by using an IAC message. The reset function sets the 80960KB processor to a known internal state after it successfully completes the self-test. These operations offer power and flexibility to hardware system design using the 80960KB processor.

4.0 MEMORY INTERFACE

The high-speed bus interface has many features that enhance high-performance designs. In particular, the burst-transfer feature allows up to four successive 32-bit data word transfers at a maximum rate of one word every processor clock cycle. This section outlines approaches for memory designs that use these features, describes memory design considerations, analyzes the timing, and lists a number of useful examples. The concepts illustrated by these examples apply to a wide variety of memory system implementations.
4.1 BASIC MEMORY INTERFACE

Figure 33 shows the major logic blocks of the memory interface circuit. The data transceivers buffer the data to compensate for any slow devices that may be connected to the 80960KB processor. The address latches demultiplex the address/data signals from the 80960KB processor and latch the address. The address decoder selects the appropriate memory device from the latched address. To accommodate a memory burst transaction, the burst logic decrements the word count, increments the local address lines 3 and 2 (LAD₃ and LAD₂), and generates a CYCLE-IN-PROGRESS signal. The timing control generates a READY signal and other specific signals required by a particular memory device. The byte enable latch stores the byte enable signals.

Although not part of the basic memory interface, the DRAM controller, SRAM interface, DRAM, SRAM, and EPROM are included in Figure 33 for completeness. In a hardware system the DRAM, SRAM, and EPROM are typically located in separate subsystems.

Although the memory interface circuit can be designed using programmable logic, gate arrays, or other custom logic, the examples use standard components wherever possible to illustrate the design concepts.

4.1.1 Data Transceivers

Standard 8-bit transceivers can be used to provide isolation and additional drive capability for the L-bus. Transceivers can be used to prevent bus contention that can occur if some memories are slow to remove data from the L-bus after a read operation. For example, if a write operation follows a read operation, the 80960KB processor may drive the L-bus before a slow device has removed its output data, potentially causing a current spike on the power and ground lines. Transceivers, however, can be omitted if the data float time of the device is short enough and the load does not exceed the 80960KB device specifications.

The data transceivers can be controlled by two signals from the 80960KB processor: data transmit/receive (DT/R) and data enable (DEN). DT/R indicates the direction of data flow and DEN enables the transceivers.

4.1.2 Address Latch/Demultiplexer

Conventional transparent latches can be used to demultiplex the address/data lines of the 80960KB processor and to hold the address constant during the memory operation. The latch is controlled by the ALE signal from the 80960KB processor. ALE passes through an inverter, so that when ALE goes low, the address flows through the latch. The low-to-high transition of ALE can be used to latch the address. The output enable of the latch can be tied to ground. The lower four address lines (LAD₃-LAD₀) are latched by the burst logic.
4.1.3 Address Decoder

The 80960KB processor accesses both memory and I/O devices by supplying a 32-bit address and a read/write command. The address decoder determines which particular memory or I/O device is selected by decoding the address lines. The following discussion focuses on memory selection, and the “Address Decoder” portion of Section 5 discusses I/O device selection using memory-mapped I/O techniques.

The memory address can be divided into regions where one region can apply to EPROM or ROM, another to RAM, and another to the I/O registers. In a 80960KB-based system the ROM address space is likely to start at address 0000 0000H because the CPU begins execution at this address. The RAM or I/O regions can start at any other address in the 4G-byte address range except for addresses FF000000H through FFFFFFFFH, which the 80960KB processor reserves for inter-agent communication.

Because of the large address range of the 80960KB processor, the address can be divided into word address bits and chip select bits. Typically the higher-order address bits are decoded to generate the selection signal for ROM, RAM, or I/O devices.

The address decoder can be located either before or after the address latches. Usually, it is placed after the latches, so that the chip-select signal does not need to be latched. Figure 33 shows the block diagram of the address decoder placed behind the address latches.

4.1.4 Burst Logic

To enhance system performance, the 80960KB processor performs burst transactions that transfer up to four data words at a maximum rate of one word every clock cycle. A DRAM controller can be designed that takes advantage of the burst-transfer capability by using the static column mode or nibble mode features of the DRAM (see the “DRAM Controller” in this section. This DRAM controller requires a signal, called CYCLE-IN-PROGRESS, to identify the start and end of a memory cycle. The burst logic generates the CYCLE-IN-PROGRESS signal.

Figure 34 shows the flow chart for the burst logic. If ADS is low and DEN is high, then the burst logic latches LAD₃ through LAD₀, and asserts the CYCLE-IN-PROGRESS signal. The burst logic checks the SIZE signals (LAD₁ and LAD₀). If the value of the SIZE signals equal zero, then the burst logic runs one memory cycle, and terminates the CYCLE-IN-PROGRESS signal. If the value of the SIZE signals do not equal zero, the burst logic runs one memory cycle, increments the lower two latched address’s (A₂ and A₃), and decrements the SIZE value. When this is finished, the burst logic checks the value of the SIZE signals again.

The burst logic can be used with EPROM, SRAM, DRAM memories. However, it cannot be used in the DRAM static column or nibble modes, because they do not support burst transactions. Because the 80960KB processor ensures that a burst transaction cannot exceed four words or cross a 16-byte boundary, incrementing LAD₃ and LAD₂ after a single data word transfer makes the burst transfer transparent to the memory devices.
4.1.5 Timing Control Logic

The timing control logic accommodates memory devices that cannot transfer information at the maximum bus rate by inserting wait states until the data becomes available. The timing control logic consists of a counter and timing logic, as shown in Figure 35. The counter produces a 4-bit binary count. The count begins when the CYCLE-IN-PROGRESS signal is asserted. The timing logic asserts READY at the appropriate time based upon the count, the EPROM-CS, and the SRAM-CS signals. For a burst transfer, READY resets the counter to properly time a READY signal for the next data transfer. When CYCLE-IN-PROGRESS is deasserted, the clock counting is terminated.

Because the timing of DRAM is more complicated, the DRAM controller generates a DRAM-RDY signal to the timing control logic. In addition, the clock count, the W/R command, and SRAM-CS signal can also be used to generate SRAM-WE and SRAM-OE Signals.
4.1.6 Byte Enable Latch

The byte enable latch holds the byte enable signals constant until the DRAM controller or SRAM interface uses the signals. As mentioned in the “L-Bus Signal Groups” section in Section 3, the byte enable signals specify which bytes (up to four) on the 32-bit data bus are transferred during the data cycle. Each individual byte enable signal selects eight data lines as shown in Table 5.

<table>
<thead>
<tr>
<th>Byte Enable Signal</th>
<th>Address Line Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE&lt;sub&gt;0&lt;/sub&gt;</td>
<td>LAD&lt;sub&gt;7&lt;/sub&gt;-LAD&lt;sub&gt;0&lt;/sub&gt;</td>
</tr>
<tr>
<td>BE&lt;sub&gt;1&lt;/sub&gt;</td>
<td>LAD&lt;sub&gt;15&lt;/sub&gt;-LAD&lt;sub&gt;8&lt;/sub&gt;</td>
</tr>
<tr>
<td>BE&lt;sub&gt;2&lt;/sub&gt;</td>
<td>LAD&lt;sub&gt;23&lt;/sub&gt;-LAD&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>BE&lt;sub&gt;3&lt;/sub&gt;</td>
<td>LAD&lt;sub&gt;31&lt;/sub&gt;-LAD&lt;sub&gt;24&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

The byte enable signals are valid from the 80960KB processor before data is transferred. These signals are asserted during the address cycle for the first data word transfer; they are asserted again during the first data cycle for the second word transfer; the second data cycle for the third word transfer; and the third data cycle for the fourth word transfer. For each word, the byte enable signals remain valid throughout every data or wait cycle until READY is asserted. After READY is asserted, the byte enable signals change during the next processor clock cycle.

The ALE signal can be used to latch the first byte enable signals. READY can be used to latch the other byte enable signals for each word.
4.2 SRAM INTERFACE

The basic memory interface can be used in conjunction with the SRAM interface to read and write to SRAM. This section describes the SRAM interface and examines the timing.

4.2.1 SRAM Interface Logic

The SRAM interface logic uses the latched byte enable signals, the SRAM-OE₀, and the SRAM-WE signals to generate four output enable signals (SRAM-OE₃ through SRAM-OE₀) and four write enable signals (SRAM-WE₃ through SRAM-WE₀), as shown in Figure 36. These signals allow the 80960KB processor to write to the data byte that is specified by the byte enable signals. SRAMs with separate OE and CS signals require only one OE signal per bank since the 80960KB ignores unrequested bytes in read operations.

4.2.2 SRAM Timing Considerations

This section analyzes the critical timing paths of the SRAM control signals. From the critical path, the timing equations can be derived to determine the memory access time for no wait state operation.

When evaluating critical timing paths, the timing calculations should use worst-case data sheet parametric specifications, rather than typical specifications. By using worst-case timing values, reliable operation is assured over all variations in temperature, voltage, and individual device characteristics. These timing values are determined by assuming the maximum propagation delay to latch an address, select a memory device, and pass through data buffers and transceivers.

Figure 37 shows the critical timing path for a one-word SRAM read operation. The diagram consists of three time periods: the address setup period (Tₐₚₐₜₓₑ₅ₑₜₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑₑ┉
SRAMs with separate OE and CS require only one OE signal per bank.

Figure 36. Logic Diagram for SRAM Interface

Figure 37. Critical Timing Path for SRAM Read Operation
For a no wait state operation, the data transfer word must be completed in two system clock (CLK) cycles. The minimum time period for a no wait state operation \( T_{\text{mem-no-wait}} \) can be determined by using equation 1.

\[
T_{\text{mem-no-wait}} = 2\text{CLK} - T_{\text{addrset}} - T_{\text{dataset}} \quad (1)
\]

where:
- \( T_{\text{mem-no-wait}} \) = Memory access time for no wait state operation
- \( 2\text{CLK} \) = Two system clock (CLK) cycles
- \( T_{\text{addrset}} \) = Maximum delay to valid address
- \( T_{\text{dataset}} \) = Maximum throughput delay of address latch
- \( + \) Maximum delay to generate chip select
- \( + \) Maximum delay to generate SRAM-OEn
- \( + \) Maximum delay through data transceiver
- \( + \) Maximum data setup time of CPU

A similar analysis can be done for burst transactions. Equation 1 can be used to determine the access time for no wait state operation of the first word. For subsequent words, equation 2 can be used. In this equation, the address setup time is replaced by delay in the burst logic to change the address \( T_{\text{burst}} \). In this case, the data transfer of each subsequent word must be completed in one system clock (CLK) cycle (no address state). The minimum access time for a no wait state operation \( T_{\text{mem-no-wait}} \) can be determined by using the lesser value of equation 1 or equation 2.

\[
T_{\text{mem-no-wait}} = \text{CLK} - T_{\text{burst}} - T_{\text{dataset}} \quad (2)
\]

where:
- \( T_{\text{mem-no-wait}} \) = Memory access time for no wait state operation
- \( \text{CLK} \) = One system clock (CLK) cycles
- \( T_{\text{burst}} \) = Maximum delay to change the address
- \( T_{\text{dataset}} \) = Maximum delay through data transceiver
- \( + \) Maximum data setup time of CPU

The memory access time can be extended by delaying the READY signal and adding wait states.

The timing analysis described for a SRAM read operation can be used for EPROM timings. If EPROMs are only used to store initialization programs, they are seldom accessed compared to memory devices used to store program data or instructions. Consequently, the addition of wait states during the read cycle does not affect overall system performance.

Figure 38 shows the critical timing path for an SRAM write operation. The diagram consists of two time periods: the address setup period \( T_{\text{addrset}} \) and the memory response period \( T_{\text{mem}} \).

During the \( T_{\text{addrset}} \) period, the 80960KB processor outputs a valid address that is latched on the low-to-high transition of ALE. The address decoder generates the SRAM-CS signal from the latched address.
During the $T_{\text{mem}}$ period the SRAM responds to the commands and writes the data. The access time of the memory determines the duration of the $T_{\text{mem}}$ period. $T_{\text{mem}}$ can be varied in increments of clock cycles by delaying the READY signal.

Two timing paths should be considered during the $T_{\text{mem}}$ period: the path where data is supplied to the memory, and the path that monitors the memory write cycle time. The first path takes into account the time for the 80960KB processor to generate valid data, the throughput delay of a data transceiver, and the data setup time requirement of the memory. The second path is the memory write cycle specification. The longer of the two paths is the critical timing path.

By examining the timing path required to operate the SRAM, equation 2 can be derived which determines SRAM write cycle time for no wait state operation. The memory cycle time is determined by the lesser value of equation 1 or equation 2.

\[
T_{\text{mem-no-wait}} = 2\text{CLK} - T_{\text{addrset}} \quad (3)
\]

where: $T_{\text{mem-no-wait}}$

- Maximum delay to valid data
- Maximum throughput delay of data transceiver
- Maximum data setup time of memory

$2\text{CLK} = \text{Two system clock (CLK) cycles}$

$T_{\text{addrset}}$

- Maximum delay to valid address
- Maximum throughput delay of address latch
- Maximum delay to generate chip select

The memory access time can be extended by delaying the READY signal and generating wait states.
4.3 DRAM CONTROLLER

This section provides design guidelines for a DRAM controller. DRAMs offer static column mode and CAS before RAS refresh features. This section shows guidelines on how to use these features with the burst capability of the 80960KB processor to significantly enhance system throughput.

The DRAM controller multiplexes the address into a row and column address, performs the refresh operation, arbitrates between a refresh request and memory request, and generates the necessary control signals for the DRAM. To implement these functions, the memory controller uses an address multiplexer, arbiter, refresh interval timer, and DRAM timing and control as shown in Figure 39.

A standard DRAM controller can be used, but it typically degrades system performance.
4.3.1 Address Multiplexer

The address multiplexer divides the DRAM address into a row and column address. The proper selection of a row or column address is accomplished by the row/column select signal (ROW/COL) from the DRAM timing and control circuit.

4.3.2 Refresh Interval Timer

The refresh interval timer periodically generates a refresh request (REF-REQ) by counting enough bus cycles to equal the refresh interval period. Since a refresh request is processed after a completed operation, the refresh period must take into account the time required to perform a bus operation, as well as the DRAM refresh specification. For example, a 1M-bit DRAM that requires 512 refresh cycles within 8 ms needs a refresh cycle every 15.6 us. To meet the DRAM specification, the refresh interval timer must generate a refresh request in less than 15.6 us to compensate for any required time to complete the operation with wait states.

After the REF-REQ signal is generated, the arbiter sends a refresh acknowledge signal REF-ACK back to the interval timer to assure that refresh occurred before generating another REF-REQ.

4.3.3 Arbiter

DRAM controller uses an arbiter to decide whether a memory cycle or refresh cycle is performed. In a synchronous design, arbitration is easily performed because memory and refresh cycle requests never occur at or near the same time.

The arbiter monitors memory cycle requests and refresh requests. The arbiter detects a DRAM memory request by decoding two signals: DRAM-CS and CYCLE-IN-PROGRESS. The REF-REQ signal indicates that a refresh cycle must be performed. The arbiter arbitrates between a memory cycle or refresh cycle and generates a Memory/Refresh (MEM/REF) signal. The DRAM timing and control block uses the MEM/REF signal to start the generation of the control signals.

When a refresh cycle is performed, the arbiter sends a REF-ACK signal to the refresh timer, which uses this signal to begin another count.

4.3.4 DRAM Timing and Control

The DRAM timing and control circuit is the final logic block and core of the DRAM controller. The functions of this circuit include the following:

- Generating the DRAM control signals (RAS, CAS, and WE) with the proper timing relationships during system operation
- Generating the DRAM-RDY signal
- Performing the refresh function by asserting CAS before RAS
- Performing several warm-up cycles required by the DRAM when power is first applied.
The DRAM timing and control logic can be designed to take advantage of the burst-transfer capability of the 80960KB processor by implementing static column mode or nibble mode. With nibble mode, a multiplexed address is applied to the DRAM, and up to four bits of data are quickly transferred by successively toggling the CAS pulse. The DRAM timing and control logic can be designed to provide the successive CAS pulses by using the CYCLE-IN-PROGRESS and DRAM-RDY signals.

Static column mode can also be used to take advantage of the burst capability of the DRAM. Static column mode allows fast access to the bits located in the selected row of the DRAM simply by changing the column address after the first access.

Figure 40 shows a flow chart for the DRAM timing and control logic using static column mode. The DRAM timing and control circuit receives a refresh request or a memory request on the MEM/REF and CYCLE-IN-PROGRESS input signals. For a memory request, the DRAM timing and control determines whether a read or a write operation is desired from the W/R signal from the 80960KB processor.

For a read operation, the DRAM timing and control logic performs similar functions on the first word: it asserts WE; it brings ROW/COL high to select a row address; it asserts RAS; it brings ROW/COL low to select the column address; it asserts CAS3 through CAS0 (derived from the four latched byte enable signals); and it generates a DRAM-RDY signal. The DRAM-RDY signal causes the burst logic to increment the address and informs the 80960KB processor that the data word was written.

After completing these functions the DRAM timing and control logic samples the CYCLE-IN-PROGRESS to determine whether to transfer another data word. If so, the DRAM timing and control logic maintains the ROW/COL signal low to select the new column address, deasserts and asserts CAS3 through CAS0 to observe the CAS precharge specification of the DRAM, and generates another DRAM-RDY. The DRAM timing and control logic repeats the procedure until all the data words are transferred. Then the DRAM timing and control logic deasserts RAS0.

For a write operation, the DRAM timing and control logic performs similar functions on the first word: it asserts WE; it brings ROW/COL high to select a row address; it asserts RAS0 (derived from the four latched byte enable signals); and it generates a DRAM-RDY signal. The DRAM-RDY signal causes the burst logic to increment the address and informs the 80960KB processor by asserting READY that the data word was written.

After completing these functions the DRAM timing and control logic samples the CYCLE-IN-PROGRESS to determine whether the 80960KB wants to transfer another data word. If so, the DRAM timing and control logic maintains the ROW/COL signal low to select the new column address, deasserts and asserts CAS3 through CAS0 to observe the CAS precharge specification of the DRAM, and generates another DRAM-RDY. The DRAM timing and control logic repeats the procedure until all the data words are transferred. Then the DRAM timing and control logic deasserts RAS0.

Although only one RAS signal is required, four CAS signals (CAS3-CAS0) are generated to enable each byte of the L-bus. These CAS signals are generated by the byte enable decoder and correspond
to the byte enable signals of the 80960KB processor. For example, $\overline{\text{CAS}_0}$, which is mapped directly from $\text{BE}_0$, selects the least-significant data byte ($\text{LAD}_7$-$\text{LAD}_0$).

Figure 40. Flow Chart for DRAM Timing and Control Logic
A single \text{WE} control signal and four \text{CAS} signals ensure that only those DRAM bytes selected for a write cycle are enabled. All other data bytes maintain their outputs in the high-impedance state. A common design error is to use a single \text{CAS} control signal and four \text{WE} control signals, using the \text{WE} signals to write the DRAM bytes selectively in write cycles that use fewer than 32 bits. Although the selected bytes are written correctly, the unselected bytes are enabled for a read cycle. These bytes output their data to the unselected bytes of the data bus while the data transceivers output data to every bit of the data bus. When the two devices simultaneously output data to the same bus, bus contention occurs.

The refresh function can be performed by asserting the \text{CAS} signal before asserting \text{RAS}. The \text{CAS} before \text{RAS} refresh feature eliminates the need for an external refresh address counter. When the \text{CAS} pulse is activated prior to the assertion of the \text{RAS} pulse, the DRAM automatically performs a refresh cycle on one row by employing an on-chip address counter. Upon completion of the refresh cycle, the address counter is automatically incremented. The MEM/REF signal from the arbiter can be used by the DRAM timing and control logic block to initiate a \text{CAS} before \text{RAS} refresh cycle.

Besides generating the \text{RAS}, \text{CAS}, and \text{WE} signals, the DRAM timing and control logic generates a number of warm-up cycles for the DRAM after reset by issuing several refresh requests.

4.3.5 Timing Considerations for the DRAM Controller

Figure 41 shows a typical example of a timing diagram for a two-word read transaction that uses static column mode; similarly, Figure 42 is a typical example for a two-word write transaction. The example assumes a memory access time that requires two wait states (\(T_w\)) for the initial data word and one wait for the second data word.

The critical timing areas for both read and write transactions are noted by circled numbers in the diagrams, which are enumerated below.

1. The delay for the CPU to generate a valid address.
2. The delay for the DRAM timing and control logic to generate the CYCLE-IN-PROGRESS signal.
3. The delay to generate the DRAM row address. This time includes the address latch throughput delay, the multiplexer throughput delay, and the address driver delay.
4. The delay to generate \text{RAS}, which includes the delay to generate the \text{DRAM-CS} signal.
5. The row address hold time after the high-to-low transition of \text{RAS}.
6. The time required to generate the multiplexer control signal (\text{ROW/COL}) after the row address hold time is satisfied.
7. The time required to switch from a row to column address plus any driver delays.
8. The delay to generate and drive the \text{CAS} signals.
9. For a read transaction, the throughput delay of the data transceivers. For a write transaction, the delay by the CPU to generate valid data.
HARDWARE REFERENCE

10. For a read transaction, the data setup time of the CPU. For a write transaction, the throughput delay of the data transceivers.

11. The time required to increment and drive the column address.

12. For a write transaction only, the delay time to bring CAS high (terminate the CAS pulse for the first data byte), to precharge the CAS pulse (required by the DRAM), and to assert CAS again.

13. The RAS precharge time, which must be satisfied before another memory cycle can begin.

Figure 41. Timing Diagram for Two-word DRAM Read Transaction
4.3.6 DRAM Interleaving

Because the DRAM consists of dynamic nodes, a row precharge time is required to recharge the nodes after every memory cycle. This time must be included in the timing evaluation, as noted by the example. To avoid the precharge time delay of the DRAM, the memory array can be arranged so that each subsequent memory access is most likely to be directed to a different bank. In this configuration, wait time between accesses is not required because while one bank of DRAMs performs the current access, another bank precharges and is ready to perform the next access immediately.

If DRAMs are interleaved (i.e., arranged in multiple banks so that adjacent addresses are in different banks), the DRAM precharge time can be masked for most accesses. With two banks of DRAMs, one for even 32-bit addresses and one for odd 32-bit addresses, all sequential 32-bit accesses can be completed without waiting for the DRAM to precharge.

Even when random accesses are made, two DRAM banks allow 50 percent of back-to-back accesses to be made without waiting for the DRAMs to precharge. The precharge time is also masked when
the 80960KB processor has no bus accesses to be performed. During these idle bus cycles, the most recently accessed DRAM bank can precharge so that the next memory access to either bank can begin immediately.

4.0 SUMMARY

The memory interface circuit allows the 80960KB processor to communicate with the memory devices. The basic memory interface logic can be divided into six blocks: the data transceivers, the address latches, the address decoder, the burst logic, the DRAM timing and control logic, and the byte enable latch. The DRAM controller and SRAM interface complete the memory interface circuit. The DRAM controller can be designed to take advantage of the 80960KB processor's burst capability to enhance system performance.

5.0 I/O INTERFACE

The 80960KB processor supports 8-bit, 16-bit, and 32-bit I/O devices by mapping them into its 4 G-byte memory address space. This section describes the design considerations for the interface between the 80960KB processor and I/O components. Several examples illustrate the design concepts.

5.1 INTERFACING TO 8-BIT AND 16-BIT PERIPHERALS

The 80960KB processor accesses I/O devices by using a memory-mapped address. Consequently, memory-type instructions can be used to perform input/output operations. For example, the 80960KB processor's LOAD and STORE instructions can directly support 8-bit and 16-bit data moves to or from I/O peripherals. The instructions include those listed below.

- Load Ordinal Byte (reads a byte)
- Load Ordinal Short (reads 16-bit data)
- Store Ordinal Byte (writes a byte)
- Store Ordinal Short (writes 16-bit data)

These instructions perform the transfer on the data bits specified by the two low-order lines of the effective address. See the 80960KB CPU Programmer's Reference Manual for complete details.

5.2 GENERAL SYSTEM INTERFACE

In a typical 80960KB processor system design, a number of slave I/O devices can be controlled through a general system interface. Other I/O devices, particularly those capable of controlling the L-bus, can use the general system interface, but may require additional logic to isolate the bus. This section describes the general system interface and assumes that the 80960KB processor does not perform burst transactions to the I/O devices.
Figure 43 shows the major logic blocks of the general system interface. Standard 8-bit data transceivers add drive capability, provide bus isolation, and prevent bus conflicts that may occur with slow I/O components. The address latch demultiplexes the address/data lines and holds the address stable throughout the L-bus transaction. The address decoder generates the I/O chip-select signals from the latched address lines. The timing control block provides the READY signal to the 80960KB processor and the I/O read and I/O write command.

This basic interface circuit is quite similar to the one used in the basic memory interface described in section 4. For most systems the same data transceivers, address decoders, and address latches can be used to access both memory and I/O devices. The timing control logic can be implemented to accommodate both memory and I/O devices.

5.2.1 Data Transceivers

Standard 8-bit transceivers can be used to provide isolation and additional drive capability for the L-bus. Transceivers prevent bus contention that can occur if some devices are slow to remove data from the data bus after a read cycle. For example, if an I/O write cycle follows an I/O read cycle, the 80960KB processor may drive the L-bus before a slow device has removed its outputs from the bus, potentially causing a current spike. Transceivers, however, can be omitted if the data float time of the device is short enough and the load does not exceed the 80960KB device specifications.

The data transceiver can be controlled by two signals from the 80960KB processor: Data Transmit/Receive (DT/R) and Data Enable (DEN). DT/R indicates the direction of data flow and DEN enables the transceivers.

5.2.2 Address Latch/Demultiplexer

Standard transparent latches can be used to demultiplex the address/data lines of the 80960KB processor. The latch is controlled by the ALE signal from the 80960KB processor. The ALE signal passes through an inverter, such that when ALE goes low, the address flows through the latch. The low-to-high transition of ALE can be used to latch the address.

If only slave-type peripherals are used in a system, the output enable of the latches can always remain active by connecting it to ground. For systems with DMA devices, the output enable can be used to permit the DMA device to drive a common address bus.

5.2.3 Address Decoder

The address decoder determines which particular I/O device is selected by decoding the address. The I/O address can be any address in the 4 Gbyte address range except for the upper 16 Mbytes (addresses FF000000H through FFFFFFFF0H, which the 80960KB processor reserves for inter-agent communication and internal I/O. Typically, a small range of address bits are reserved for accessing I/O devices by defining certain higher-order address bits as an I/O access.
Figure 43. Simplified I/O Interface
As an example, consider a 32-bit address: A31 through A15 could indicate an I/O access when A31 is set to zero, and A30-A15 are set to one; A14 through A5 could then be used to specify a particular I/O device; and A4 through A0 can be used to access up to 8 registers of the I/O component. A1 and A0 are not used by the I/O device. This particular scheme selects up to 1,024 devices, while using only 32K bytes of the available 4 Gbytes of address space.

The address decoder can be located either before or after the address latches. Usually, it is placed after the latches, so that the chip-select signal does not need an additional latch.

### 5.2.4 Timing Control Logic

The timing control logic accommodates I/O devices that cannot transfer information at the maximum bus rate by inserting Wait States until the data becomes available. The timing control logic consists of a counter and timing logic, as shown in Figure 44. The counter produces a 4-bit binary count. The count is started at the beginning of the operation (determined by ADS and DEN) and is stopped by the READY signal. The timing logic asserts the READY signal, the I/O write command (I/O-WR), and the I/O read command (I/O-RD) based upon the clock count, the I/O chip select signal (I/O-CS), and the W/R command.

![Figure 44. Timing Control Block Diagram](image)

For many peripherals, the timing logic can be programmed to assert READY at the appropriate count for the selected device. Specific I/O chip select signals can be used to indicate how many clock cycles to wait before asserting READY.

For some I/O peripherals, particularly bus masters, READY cannot be determined by counting clock cycles. For these I/O devices, READY can be supplied by the device and passed on to 80960KB processor.

The timing control block can assert the I/O-RD or I/O-WR signal for I/O devices based upon the clock count. The timing for these signals can be selected for the slowest device to simplify the logic circuit or can be customized for each individual peripheral device to maximize performance.
5.2 I/O INTERFACE DESIGN EXAMPLES

The general system interface shown in Figure 43 can be used to connect the 80960KB processor to many slave peripherals. The following list includes some common peripherals compatible with this interface:

- 8259A Programmable Interrupt Controller
- 8253, 8254 Programmable Interval Timer
- 8272 Floppy Disk Controller
- 82062, 82064 Fixed Disk Controller
- 82510, Asynchronous Serial Controller
- 8274, 82530 Multi-Protocol Serial Controller
- 8255 Programmable Peripheral Interface
- 8041, 8042 Universal Peripheral Interface

This section provides guidelines and design considerations for interfacing the 80960KB processor to different types of I/O configurations. Specifically, four design examples are examined. The 8259A design example shows how to interface the 80960KB processor to a slave-type peripheral device. The 82586 design example shows how a 16-bit bus master reads and writes to the 80960KB processor’s system memory. The 82786 design example shows how the 80960KB processor can read or write to graphics memory using a 16-bit data bus.

5.3.1 8259A Programmable Interrupt Controller

The 8259A Programmable Interrupt Controller is designed for use in interrupt-driven microcomputer systems, where it manages up to eight independent interrupts. The 8259A handles interrupt priority resolution and returns an 8-bit vector to the 80960KB processor during an interrupt-acknowledge cycle. Intel Application Note AP-59 contains detailed information on configurations of the 8259A.

5.3.2 Interface

Figure 45 shows the connection of the 80960KB processor to a single 8259A Interrupt Controller. This circuit consists of the general system interface plus a bidirectional buffer. The example assumes that several interrupt requests occur at the same time so that priority resolution is required.

The data lines from the 8259A are not directly aligned to the 80960KB processor because of the difference in priority resolution between the devices. Although both devices use an 8-bit interrupt vector, the 80960KB processor implicitly defines the priority by dividing the interrupt vector by eight. The 8259A defines the priority in the lower three bits of the interrupt vector. Furthermore, the highest priority vector of the 80960KB processor has a value of 31 in the upper five bits of the interrupt vector. Whereas, the highest priority interrupt of the 8259A has a value of 0 in the lower three bits of the interrupt vector.
To resolve the priority difference, the interrupt vector from the 8259A can be inverted and rotated left by three bits as shown by the data alignment between the 80960KB processor and 8259A in Figure 45. Rotating the data bits in this manner provides two advantages: the interrupt table for the 8259A can be located by contiguous addresses, and the upper two most significant bits of the interrupt vector remain free to group interrupt vectors if additional 8259As are needed.

Care must be exercised, however, when programming the registers of the 8259A. For example, assume that the second initialization command word (ICW2 register) of the 8259A requires a data byte value of \(00011111\) \(_B\). To transfer the correct information, the 80960KB processor needs to write a data word with the value of \(00000111\) \(_B\) because this word is rotated left three places and inverted.

### 5.3.3 Operation

The 8259A starts the interrupt cycle by generating an interrupt request (INT) to the 80960KB processor, which receives the signal at the INTR input pin. This assumes the Interrupt Control register of the 80960KB processor is set to accommodate an external interrupt controller.

When the 80960KB processor comes to a breakpoint in its execution, it asserts the \(\text{INT}A\) signal twice. The first \(\text{INT}A\) signal acknowledges the interrupt request and causes the 8259A to prioritize the interrupt requests it received up to this point. The \(\text{INT}A\), together with the 8259A-CS, are applied to the timing control logic to generate a \(\text{READY}\) signal.

The 80960KB processor automatically asserts the second \(\text{INT}A\) signal five clock cycles after the assertion of \(\text{READY}\). After the second assertion of \(\text{INT}A\), the 80960KB processor reads the interrupt vector from the 8259A.

The bidirectional buffer inverts and passes the 8-bit vector to the 80960KB processor with the appropriate lines rearranged. The output enable signal for the data buffer is controlled by \(\text{INT}A\) for this operation. After the data transfer is completed, the timing control circuit generates a second \(\text{READY}\) signal to terminate the interrupt acknowledge cycle.

The same circuitry can be used to read or write to the 8259A registers. In this case, the 80960KB processor selects the 8259A through a memory-mapped address. Local address line 2 (A2) selects one of two internal registers of the 8259A. The I/O read or I/O write command is generated by the timing control circuit. The data passes through the bidirectional data buffer to or from the selected register of the 8259A.

The direction of data flow through the buffer is controlled by three logic gates shown in Figure 45. For an I/O write operation, the I/O Write command and 8259A-CS signal control the output enable signal of the bidirectional buffer. Similarly, for a read operation, the I/O Read command and the 8259A-CS signal control the output enable signal of the buffer. After the data is transferred, the timing control circuit asserts \(\text{READY}\).
5.3.4 82530 Serial Communication Controller

The 82530 Serial Communication Controller is a dual-channel, multi-protocol controller with on-chip baud rate generators, digital phase locked loops, various data encoding/decoding, and extensive diagnostic capabilities. The 82530 is designed to interface with high-speed serial communications lines using a variety of communication protocols, including asynchronous, synchronous, and HDLC/SDLC protocols. The 82530 contains two independent full-duplex channels.

The general system interface circuit previously described can be used to connect the 80960KB processor to the 82530, as shown in Figure 46. The 82530 can send an interrupt request to the 80960KB processor as shown or it can send the interrupt request to an interrupt controller, which in turn sends it to the 80960KB processor. The 80960KB processor responds to the interrupt request and issues an address. After the address is latched, the address lines are decoded to generate a chip-select (82530-CS) signal to activate the 82530.

The lower two address lines, A2 and A3, are used for channel selection and command/data selection. A2 is connected to the Channel-A/Channel-B(A/B) select input pin. This selects the channel that performs the serial read or write operations. A3 is connected to the Data/Command (D/C) select input pin. This signal defines the type of information transferred to or from the 82530 on the data lines (D7 through D0). A high level means data is transferred; a low level indicates a command.

The timing control circuit generates an I/O read or I/O write command based on the W/R command from the 80960KB processor. When the data transfer is completed, the timing control circuit sends a READY signal to terminate the transaction.

The baud rate clocks can be programmed in several ways, including use of an external crystal.

5.3.5 82586 Local Area Network Coprocessor Example

The 82586 is an intelligent, high-performance communications controller designed to perform most tasks required for controlling access to a local area network (LAN), such as Ethernet or Starlan. In many applications, the 82586 is the communication manager for a station connected to a LAN controller. Such a station usually includes a host CPU, shared memory, a Serial Interface Unit, a transceiver, and LAN controller link, as shown in Figure 47. The 82586 performs all functions associated with data transfer between the shared memory and the LAN link, including:

- Framing
- Link management
- Address filtering
- Error detection
- Data encoding
- Network management
- Direct memory access
HARDWARE REFERENCE

- Buffer chaining
- High-level (user) command interpretation

The 82586 has two interfaces: a 16-bit bus interface and a network interface to the Serial Interface Unit. The bus interface is described here. For detailed information on using the 82586, refer to the Local Area Networking Component User’s Manual.

Figure 46. Block Diagram for 82530 Interface
5.3.6 Interface

There are several ways to design an interface between the 82586 and the 80960KB processor. The chosen design example shows how to interface the 82586 using a shared bus. In this example, the 82586 operates in minimum mode at one-half the processor clock frequency.

The primary function of the interface circuit is to allow the 82586 to read and write 16-bit data using the 32-bit L-bus. This is accomplished by adding the high-order address lines and translating the 16-bit data lines to the 32-bit data lines by using byte enable signals.

Figure 48 shows the 82586 interface circuit, which includes the DRAM controller (see the “DRAM Controller” section in Chapter 4. This interface uses the general system interface circuit plus other logic units that specifically pertain to the 82586: the LAN data transceivers, the byte enable converter, and the LAN address latches. These logic blocks are highlighted by the shaded boxes.
The LAN data transceivers connect 16 data lines from the 82586 to both the upper and lower 16 bits of the L-bus. The data transfer is controlled by converting A, A1, and the BHE to four byte enable signals as shown in Figure 49. A1 selects between the upper and lower 16-bit data lines; A selects the lower data byte for either the upper or lower 16-bit data lines; and the byte high enable signal (BHE) selects the upper data byte for either the upper or lower 16-bit data lines. Data flows through the buffers when the appropriate byte enable signal is asserted. The direction of the data flow is controlled by the DT/R signal of the 82586.

The LAN address latches are used to demultiplex AD15 through AD. The address lines and BHE are latched by the ALE signal from the 82586. The upper address lines (A31 through A16) are generated by hardware programmable DIP switches.

The 82586 begins operation when the Channel Attention (CA) input signal is asserted. This signal is generated by gating the write command and 82586 chip select signal.
5.3.7 Operation

The interaction between the 82586 and the 80960KB processor is described below and is summarized in Figure 50.

- The 80960KB processor invokes the 82586 by supplying a memory-mapped address and a write command. The memory-mapped address results in an 82586-CS signal, which is gated with a write command to produce the CA signal.

- The 82586 responds by generating a hold request and waits for HLDA.

- The 80960KB processor asserts HLDA, which enables the outputs of the LAN address latches and disables the outputs of the address latches next to the 80960KB processor. The HLDA signal also gives control of the L-bus to the 82586.

- After the 82586 takes control of the bus, it generates a 16-bit address (AD₁₅ through AD), an ALE signal, and a BHE signal. The upper address lines are provided by the programmable DIP switches to produce an address on the L-bus.

- A₁ and A (from the 82586), and BHE are decoded to generate four byte enable signals (BE₃ through BE). DEN enables the output of the byte enable converter.

- DT/R from the 82586 controls the direction of the data flow through the buffers.

- The read or write signal from the 82586 is applied to the DRAM controller.

- The 82586 accesses DRAM by using the DRAM controller.

- The DRAM-RDY is asserted by the DRAM controller. This action enables the output of the LAN data transceiver and terminates the 82586 memory cycle. The timing control logic passes the DRAM-RDY signal as the READY signal to the 82586.

- The 82586 deasserts HOLD and the 80960KB processor deasserts HLDA. The 80960KB processor regains control of bus.
5.3.8 82786 Graphics Coprocessor Example

The 82786 is a high performance graphics coprocessor that provides high quality text and advanced display control. It provides full support for graphics primitives at up to 25 million pixels per second and bit-mapped text up to 25 thousand characters per second. This graphics processor supports advanced features such as hardware windows, zooming, panning, and scrolling. Intel Application Note AP-259 and Application Note AP-270 contain detailed information on 82786.

When using the 82786, it may be necessary for the 80960KB processor to write to graphics memory. The interface design example illustrates how the 80960KB processor can transfer a 32-bit data word to the 16-bit data bus of the 82786.
5.3.9 Interface

There are several ways to design an interface between the 82786 and the 80960KB processor. In this example, the 80960KB processor reads or writes to graphics memory by accessing the 82786 through the interface logic circuit. This example assumes that the 82786 operates in the slave mode, and that the 80960KB processor does not perform burst transfers. The 80960KB processor only performs burst transfers for instructions that specify accesses for more than one word or for instruction fetches.

The interface circuit translates a 32-bit data bus to a 16-bit data bus by dividing the data lines into the upper and lower 16 bits and sequencing the data transmission. When the 80960KB processor writes to graphics memory, the bidirectional transceivers sequence the lower and the upper data bits of the L-bus to the 16-bit data bus of the 82786.

The process is reversed when the 80960KB processor reads from graphics memory. The bidirectional transceivers form a 32-bit data word by latching the first 16-bit data word on the lower data lines, routing the next 16 bits to the upper data lines, and then passing the 32-bit data word on the L-bus.

Figure 51 shows the details of the graphics controller interface circuit. This interface uses the general system interface circuit plus the following logic units: the bidirectional transceivers, the data buffer control, the data bus controller, and the address translator. These logic blocks are highlighted by the shaded boxes.

The bidirectional transceivers pass data to (from) a 32-bit data bus from (to) a 16-bit data bus. Data is sequenced through the transceivers by the control signals generated by the data buffer controller.

The data buffer control logic generates the signals that operate and sequence the bidirectional transceivers. The direction signal for data flow through the transceivers is derived from the W/R signal of the 80960KB processor. The data buffer control logic generates four output enable signals: $G_{AB}^L$ enables the outputs on the B side for the lower 16 bits; $G_{BA}^L$ enables the outputs on the A side for the lower 16 bits; $G_{AB}^H$ enables the outputs on the B side for the higher 16 bits; and $G_{BA}^H$ enables the outputs on the A side for the higher 16 bits. These output enable signals are derived from the byte enable signals and are asserted when the slave enable signal (SEN) is activated by the 82786.

The select lines for the bidirectional transceivers allow data to flow from either the latched data or the input pins. These lines, which are not shown, can be hardwired.

The data bus controller provides the read (RD) and write (WR) commands, memory or I/O signal (M/IO), and a READY$_o$ signal. This circuit generates two read or write commands for every 32-bit data transfer to or from the 80960KB processor (one for each 16-bit data transfer). The data bus controller starts counting clock cycles when the 82786-CS and CYCLE-IN-PROGRESS signals are asserted. At the proper time (based upon clock counts), it asserts the read/write command. The data bus controller produces READY$_o$ after receiving the SEN signal from the 82786. READY$_o$ resets the count, and another read/write command is generated.
The address translator performs four functions: it converts the four byte enable signals to \( A, A_1, \) and \( \text{BHE}; \) it increments \( A \) after receiving \( \text{READY}_0 \) for the first 16-bit transfer; it generates the clock signal (\( \text{CBA}_1 \)) that latches the first 16-bit data word in the bidirectional transceivers when the 80960KB processor performs a read operation; and it generates the \( \text{READY} \) signal for the CPU.

Not shown is the cycle detector circuit that generates the \( \text{CYCLE-IN-PROGRESS} \) signal. This signal can be generated by using the circuit similar to the one shown in Figure 44. The start of the cycle can be detected by gating the \( \text{ADS} \) and \( \text{DEN} \) signals. The end of the cycle can be indicated by \( \text{READY} \).

### 5.3.10 Operation

The interaction between the 82786 and the 80960KB processor is summarized in Figure 52. The operation is divided into two 16-bit data movements for both a read and write operation.
FIRST 16-BIT DATA TRANSFER

INTERFACE CIRCUIT GENERATES CS, RD, (OR WR), BHE, AND M/IO

82786 RECOGNIZES CS BY ASSERTING SEN

READ OPERATION
INTERFACE CIRCUIT CONTROLS DATA FLOW:
- GENERATES READY0
- LATCHES DATA ON LOWER 16-BITS WITH CLOCK SIGNAL (CBA1)
- INCREMENTS ADDRESS A1

WRITE OPERATION
INTERFACE CIRCUIT CONTROLS DATA FLOW:
- GENERATES READY0
- ENABLES OUTPUT FOR DATA ON LOWER 16 BITS
- INCREMENTS ADDRESS A1

82786 DEASSERTS SEN

SECOND 16-BIT DATA TRANSFER

DATA BUS CONTROLLER GENERATES RD (OR WR), BHE, AND M/IO

82786 RECOGNIZES CS BY ASSERTING SEN

READ OPERATION
INTERFACE CIRCUIT CONTROLS DATA FLOW:
- GENERATES READY0
- ENABLES OUTPUT FOR LOWER AND HIGHER 16 BITS

WRITE OPERATION
INTERFACE CIRCUIT CONTROLS DATA FLOW:
- GENERATES READY0
- ENABLES OUTPUT FOR DATA ON HIGHER 16 BITS

INTERFACE CIRCUIT GENERATES READY FOR 80950 AND 82786 AND DEASSERTS SEN

Figure 52. Operational Flow Diagram for 82786 Interface Circuit
The 80960KB processor generates a memory-mapped address and data for the desired graphics memory location. It accesses the 82786 by triggering the interface circuit to generate the chip select signal and several operational signals: the read (RD) or write (WR) command, BHE, and the memory or I/O (M/I/O) signal. The 82786 begins the memory operation after it completes the current graphics processing activity. The 82786 acknowledges that it is performing a memory operation by asserting SEN.

After the 82786 asserts SEN, it begins a 16-bit memory read or write operation by translating the address inputs (A₂¹ through A) to a multiplexed DRAM address, and generating the DRAM control signals. Note that Aₐ and A are derived from the byte enable signals.

For a read operation, the data bus controller uses SEN to generate the READY signal. The assertion of READY causes the address translator to increment Aₐ and to generate CBAₐ, which latches the lower 16 data bits on the B inputs of the bidirectional transceivers to the A side.

Similarly, for a write operation, the data bus controller uses SEN to generate the READY signal. The assertion of READY causes the address translator to increment Aₐ. The data buffer control uses SEN and the byte enable signals to produce GABₐ, which enable the outputs for the lower 16 data bits of the bidirectional transceivers.

The 82786 then deasserts SEN and the transfer of the first 16 data bits is complete. To transfer the second 16 data bits, the interface circuit requests another memory operation by generating RD (or WR), BHE, and M/I/O (CS is already asserted). After it completes the current graphics processing activity, the 82786 begins the memory operation and asserts SEN.

For a read operation, the data bus controller uses SEN to generate the READY signal. The data buffer control uses SEN to assert GBAₐ and GBAₐ, which enable the outputs for the higher and lower 16 data bits.

For a write operation, the data bus controller uses SEN to generate the READY signal. The data buffer control uses SEN and the byte enable signals to produce GABₐ, which enable the outputs for the higher 16 data bits of the bidirectional transceivers.

The address translator generates READY for the 80960KB processor from the second READY to terminate the data transfer to the graphics memory.

5.4 SUMMARY

The 80960KB processor supports 8-bit, 16-bit, and 32-bit I/O interfaces. A general system interface circuit can be designed that connects to many slave-type peripherals. This interface can be expanded to accommodate a bus master peripheral or a 32-bit to 16-bit data bus translator. These interfaces were illustrated by four design examples.
1.0 A NEW 32-BIT ARCHITECTURE FROM INTEL

The 80960KB processor marks the introduction of the 80960 architecture—a 32-bit architecture from Intel. This architecture has been designed specifically to meet the needs of embedded applications such as machine control, robotics, processor control, avionics, and instrumentation. It represents a renewed commitment from Intel to provide reliable, high-performance processors and controllers for the embedded processor marketplace.

The 80960 architecture can best be characterized as a high-performance computing engine. It features high-speed instruction execution and ease of programming. It is also easily extensible, allowing processors and controllers based on this architecture to be conveniently customized to meet the needs of specific processing and control applications.

Some of the important attributes of the 80960 architecture include:

- full 32-bit registers
- high-speed, pipelined instruction environment
- a convenient program execution environment with 32 general-purpose registers and a versatile set of special-function registers
- a highly optimized procedure call mechanism that features on-chip caching of local variables and parameters
- extensive facilities for handling interrupts and faults
- extensive tracing facilities to support efficient program debugging and monitoring
- register scoreboarding and write buffering to permit efficient operation with lower performance memory subsystems

The following sections describe those features of the 80960 architecture that are provided to streamline code execution and simplify programming. Also described are those features that allow extensions to be added to the architecture.

1.1 HIGH PERFORMANCE PROGRAM EXECUTION

Much of the design of the 80960 architecture has been aimed at maximizing the processor's computational and data processing speed through increased parallelism. The following paragraphs describe several of the mechanisms and techniques used to accomplish this goal, including:

- an efficient load and store memory-access model
- caching of code and procedural data
- overlapped execution of instructions
- many one or two clock instructions
1.1.1 Load and Store Model

One of the most important features of the 80960 architecture is that most of its operations are performed on operands in registers, rather than in memory. For example, all the arithmetic, logic, comparison, branching and bit operations are performed with registers and literals.

This feature provides two benefits. First, it increases program execution speed by minimizing the number of memory accesses required to execute a program. Second, it reduces memory latency encountered when using slower, lower-cost memory parts.

To support this concept, the architecture provides a generous supply of general-purpose registers. For each procedure, 32 registers are available (28 of which are available for general use). These registers are divided into two types: global and local. Both these types of registers can be used for general storage of operands. The only difference is that global registers retain their contents across procedure boundaries, whereas the processor allocates a new set of local registers each time a new procedure is called.

The architecture also provides a set of fast, versatile load and store instructions. These instructions allow burst transfers of 1, 2, 4, 8, 12 or 16 bytes of information between memory and the registers.

1.1.2 On-Chip Caching of Code and Data

To further reduce memory accesses, the architecture offers two mechanisms for caching code and data on chip: an instruction cache and multiple sets of local registers. The instruction cache allows prefetching of blocks of instruction from memory, which helps insure that the instruction execution pipeline is supplied with a steady stream of instructions. It also reduces the number of memory accesses required when performing iterative operations such as loops. (The size of the instruction cache can vary. With the 80960KB processor, it is 512 bytes.)

To optimize the architecture's procedure call mechanism, the processor provides multiple sets of local registers. This allows the processor to perform most procedure calls without having to write the local registers out to the stack in memory.

(The number of local-register sets provided depends on the processor implementation. The 80960KB processor provides four sets of local registers.)

1.1.3 Overlapped Instruction Execution

Another technique that the 80960 architecture employs to enhance program execution speed is overlapping the execution of some instructions. This is accomplish through two mechanisms: register scoreboarding and branch prediction.

Register scoreboarding permits instruction execution to continue while data is being fetched from memory. When a load instruction is executed, the processor sets one or more scoreboard bits to indicate the target registers to be loaded. After the target registers are loaded, the scoreboard bits are
cleared. While the target registers are being loaded, the processor is allowed to execute other instructions that do not use these registers. The processor uses the scoreboard bits to insure that target registers are not used until the loads are complete. (The checking of scoreboard bit is carried out transparently from software.) The net result of using this technique is that code can often be optimized in such a way as to allow some instructions to be executed in zero clock cycles (that is, executed for free).

Conditional branch instructions commonly cause bottlenecks in the instruction execution pipeline, since the instruction decoder cannot decode instructions past the branch instruction until it knows the direction the branch is going to take. The 80960 architecture solves this problem with a technique called branch prediction. Branch prediction allows a programmer or compiler to select conditional branch instructions that indicate to the processor the direction a branch is likely to go. The decoder can then continue decoding instructions beyond the branch, even though the branch condition has not yet been tested. This technique eliminates waits between the decoder and execution unit, while branch conditions are being evaluated.

Note

The branch prediction mechanism is not implemented in the 80960KB series of processors.

1.1.4 Single-Clock Instructions

It is the intent of the 80960 architecture that a processor be able to execute commonly used instructions such as moves, adds, subtracts, logical operations, and branches in a minimum number of clock cycles (preferably one clock cycle). The architecture supports this concept in several ways. For example, the load and store model described earlier in this section (with its concentration on register-to-register operations) eliminates the clock cycles required to perform memory-to-memory operations.

Also, all the instructions in the 80960 architecture are 32-bits long and aligned on 32-bit boundaries. This feature allows instructions to be decoded in one clock cycle. It also eliminates the need for an instruction-alignment stage in the pipeline.

The design of the 80960KB processor takes full advantage of these features of the architecture, resulting in over 50 instructions that can be executed in a single clock-cycle.

1.1.5 Efficient Interrupt Model

The 80960 architecture provides an efficient mechanism for servicing interrupts from external sources. To handle interrupts, the processor maintains an interrupt table of 248 interrupt vectors (240 of which are available for general use). When an interrupt is signalled, the processor uses a pointer from the interrupt table to perform an implicit call to an interrupt handler procedure. In performing this call, the processor automatically saves the state of the processor prior to receiving the interrupt; performs the interrupt routine; and then restores the state of the processor. A separate interrupt stack is also provided to segregate interrupt handling from application programs.
The interrupt handling facilities also feature a method of evaluating interrupts by priority. The processor is then able to store interrupt vectors that are lower in priority than the task that the processor is currently working on in a pending interrupt section of the interrupt table. At certain defined times, the processor checks the pending interrupts and services them.

1.1 SIMPLIFIED PROGRAMMING ENVIRONMENT

Partly as a side benefit of its streamlined execution environment and partly by design, processors based on the 80960 architecture are particularly easy to program. For example, the large number of general purpose registers allows relatively complex algorithms to be executed with a minimum number of memory accesses. The following paragraphs describe some of the other features for the architecture that simplify programming.

1.2.1 Highly Efficient Procedure Call Mechanism

The procedure call mechanism makes procedure calls and parameter passing between procedures simple and compact. Each time a call instruction is issued, the processor automatically saves the current set of local registers and allocates a new set of local registers for the called procedure. Likewise, on a return from a procedure, the current set of local registers is deallocated and the local registers for the procedure being returned to are restored. On a procedure call, the program thus never has to explicitly save and restore those local variables and parameters that are stored in local registers.

1.2.2 Versatile Instruction Set and Addressing

The selection of instructions and addressing modes also simplifies programming. The architecture offers a full set of load, store, move, arithmetic, comparison and branch instructions, with operations on both integer and ordinal data types. It also provides a complete set of Boolean and bit-field instructions, to simplify operations on bits and bit strings.

The addressing modes are efficient and straightforward, while at the same time providing the necessary indexing and scaling modes required to address complex arrays and record structures.

The large 4-gigabyte address space provides ample room to store programs and data. The availability of 32 addressing lines allows some address lines to be memory-mapped to control hardware functions.

1.2.3 Extensive Fault Handling Capability

To aid in program development, the 80960 architecture defines a wide selection of faults that the processor detects, including arithmetic faults, invalid operands, invalid operations, and machine faults. When a fault is detected, the processor makes an implicit call to a fault handler routine, using a mechanism similar to that described above for interrupts. The information collected for each fault allows program developers to quickly correct faulting code. It also allows automatic fault recovery from some faults.
1.2.4 Debugging and Monitoring

To support debugging systems, the 80960 architecture provides a mechanism for monitoring processor activity by means of trace events. The processor can be configured to detect as many as seven different trace events, including the instruction execution, branch events, calls, supervisor calls, returns, prereturns, and breakpoints. When the processor detects a trace event, it signals a trace fault and calls a fault handler. Intel provides several tools that use this feature, including an in-circuit emulator (ICE) device.

1.3 SUPPORT FOR ARCHITECTURAL EXTENSIONS

The 80960 architecture described earlier in this chapter provides a high-performance computing engine for use as the computational and data processing core of embedded processors or controllers. The architecture also provides several features that enable processors based on this architecture to be easily customized to meet the needs of specific embedded applications, such as signal processing, array processing, or graphics processing.

The most important of these features is a set of 32 special function registers. These registers provide a convenient interface to circuitry in the processor or to pins that can be connected to external hardware. They can be used to control timers, to perform operations on special data types, or to perform I/O functions.

The special function registers are similar to the global registers. They can be addressed by all the register-access instructions.

1.5 EXTENSIONS INCLUDED IN THE 80960K SERIES PROCESSORS

The 80960K series of processor offer a complete implementation of the 80960 architecture, plus several extensions to the architecture. These extensions fall into two categories: floating-point processing and interagent communication.

1.5.1 On-Chip Floating Point

The 80960KB processor provides a complete implementation of the IEEE standard of binary floating-point arithmetic (IEEE 754-185). This implementation includes a full set of floating-point operations, including add, subtract, multiply, divide, trigonometric functions, and logarithmic functions. These operations are performed on single precision (32-bit), double precision (64-bit), and extended precision (80-bit) real numbers.

One of the benefits of this implementation is that the floating-point handling facilities are completely integrated into the normal instruction execution environment. Single- and double-precision floating-point values are stored in the same registers as non-floating point values. The four, 80-bit floating-point registers are provided to hold extended-precision values.
1.5.2 Interagent Communication

All of the processors in the 8096K series provide an interagent communication (IAC) mechanism, which allows agents connected to the processor’s bus to communicate with one another. This mechanism operates similarly to the interrupt mechanism, except that IAC messages are passed through dedicated sections of memory. The sort of tasks handled with IAC messages are processor reinitialization, stopping the processor, purging the instruction cache, and forcing the processor to check pending interrupts.

1.6 LOOK FOR MORE IN THE FUTURE

As has been shown in the preceding discussion, the 80960 architecture offers lots of possibilities and lots of room to grow. The first implementation of this architecture (the 80960KB processor) provides average instruction processing rates of 7.5 million instructions per second (7.5 MIPS) at 20 MHz clock rate and 10 MIPS at a 25 MHz clock rate.\(^1\) This performance places the 80960 KB at the top of the performance range for advanced, VLSI processor architectures.

However, the 80960KB is only the beginning. With improvements in VLSI technology, future implementation of this architecture will offer even greater performance. They will also offer a variety of useful extensions to solve specific control and monitoring needs in the field of embedded applications.

2.0 EXECUTION ENVIRONMENT

This section describes how the 80960KB processor stores and executes instructions and how it stores and manipulates data. The parts of the execution environment that are discussed include the address space, the register model, the instruction pointer, and the arithmetic controls. The execution environment’s procedure stack and procedure-call mechanism are described in section 3.

2.1 OVERVIEW OF THE EXECUTION ENVIRONMENT

When the 80960KB processor is initialized, it sets up an execution environment. It then begins executing instructions from a program, using this execution environment to store and manipulate data.

Figure 1 shows the part of the execution environment that the processor sets up to execute a procedure within a program. This environment consists of \(2^{32}\)-byte address space, a set of global and floating-point registers, a set of local registers, a set of arithmetic-control bits, the instruction pointer, a set of process-control bits, and a set of trace-controls bits. All of these items, except the address space, reside on the 80960KB chip.

\(^1\)MIP is equivalent to the performance of a Digital Equipment Corp. VAX 11/780.
Note

The floating-point registers shown in Figure 1 are not defined in the 80960 architecture. They are extensions to the architecture that have been added to the 80960KB processor to support floating-point operations on the extended-real (floating point) data type. (The 80960KA processor does not provide floating-point registers.)

The 32 special-function registers (shown in Figure 1 in a dashed box) are defined in the 80960 architecture. These registers are not implemented in the 80960KB and 80960KA processors.

When the instruction stream includes a procedure call, a procedure stack and some additional elements are added to this execution environment. These procedure-call related elements are shown and discussed in Section 3.

Figure 3-1. Execution Environment
2.2 ADDRESS SPACE

From the point of view of the processor, the address space is flat (unsegmented) and byte addressable, with addresses running contiguously from 0 to $2^{32} - 1$. Programs and the kernel can allocate space for data, instructions, and the stack anywhere within this space, with the following exceptions:

- Instructions must be aligned on word boundaries.
- Some of the addresses in the upper 16M bytes of the address space (addresses $\text{FF}000000_{16}$ through $\text{FFFF}FFFF_{16}$) are reserved for specific functions. In general, programs and the kernel should not use this section of the address space.

The memory requirements to support this address space are given in Section 6 in the section titled “Memory Requirements”.

2.3 REGISTER MODEL

The processor provides three types of data registers: global, floating-point, and local. The 16 global registers constitute a set of general-purpose registers, the contents of which are preserved across procedure boundaries. The 4 floating-point registers are provided to support extended floating-point arithmetic. Their contents are also preserved across procedure boundaries. The 16 local registers are provided to hold parameters specific to a procedure (i.e. local variables). For each procedure that is called, the processor allocates a separate set of 16 local registers.

For any one procedure within a program, 36 registers are thus available (as shown in Figure 2): the global registers, the 4 floating-point registers, and the 16 local registers. All of these registers are maintained on the processor chip.

2.3.1 Global Registers

The 16 global registers (g0 through g15) are 32-bit registers. Each register can thus hold a word (32 bits) of data. Registers g0 through g14 are general-purpose registers; g15 is reserved for the current frame pointer (FP). The FP contains the address of the first byte in the current (topmost) stack frame. (The FP and the procedure stack are discussed in detail in Section 3).

The general-purpose global registers (g0 through g14) can hold any of the data types that the processor recognizes (i.e. ordinals, integers, reals).

2.3.2 Floating-Point Registers

The four floating point registers (fp0 through fp3) are 80-bit registers. These registers can be accessed only as operands of floating-point instructions. All numbers stored in these registers are stored in extended-real format. (This format is described in section 11). The processor automatically converts floating-point values from real or long-real format into extended-real format when a floating-point register is used as a destination for an instruction.
Note
The floating-point registers are defined in the 80960 architecture as an option for processors such as the 80960KB that support floating-point operations. These registers may be omitted from implementations of the architecture that do not support floating-point operations.

Figure 2. Registers Available to a Single Procedure
2.3.3 Local Registers

The 16 local registers (r0 through r15) are 32-bit registers, like the global registers. The purpose of the local registers is to provide a separate set of registers, aside from the global and floating-point registers, for each active procedure. Each time a procedure is called, the processor automatically sets up a new set of local registers for that procedure and saves the local registers for the calling procedure. The program does not have to explicitly save and restore these registers.

Local registers r3 through r15 are general-purpose registers. Register r0 through r2 are reserved for special functions, as follows: register r0 contains the previous frame pointer (PFP); r1 contains the stack pointer (SP); and r2 contains the return instruction pointer (RIP). (The PFP, SP, and RIP are discussed in detail in Section 3). The processor accesses the local registers at the same speed as it does the global registers.

2.3.4 Register Alignment

Several of the processor’s instructions operate on multiple-word operands. For example, the load-long instruction (ldl) loads two words from memory into two consecutive registers. Here, the register number for the least significant word is specified in the instruction and the most significant word is automatically loaded into the next higher numbered register.

In cases where an instruction specifies a register number and multiple, consecutive registers are implied, the register number must be even if two registers are accessed (e.g. g0, g2) and an integral multiple of four if three or four registers are access (e.g. g0, g4). If a register reference for a source value is not properly aligned, the value is undefined. If a register reference for a destination value is not properly aligned, the registers that the processor writes to are undefined.

2.3.5 Register Scoreboarding

The 80960KB provides a mechanism called register scoreboarding that in certain situations permits instructions to be executed concurrently. This mechanism works as follows. While an instruction is being executed, the processor sets a scoreboard bit to indicate that a particular register or group of registers is being used in an operation. If the instructions that follow do not use registers in that group, the processor in some instances is able to execute those instructions before execution of the prior instruction is complete. In effect, the register scoreboarding mechanism allows some instructions to be executed for free (zero clock cycles).

A common application of this feature is to execute one or more fast instructions (instructions that take one to three clock cycles) concurrently with load instructions. A load instruction typically takes 3 to 9 clock cycles (depending on the design of system memory). Register scoreboarding allows other instructions to be executed concurrently with the load instruction, provided that the other instructions do not affect the registers being loaded. For example, the following group of instructions load a group of local registers while performing some other operations on data in global registers.
ld xyz, r6 # r6 ← data from address xyz
addi g4, g6, g7 # g7 ← g4 + g6
addi g9, g10, g11 # g11 ← g9 + g10
ld abc, r8 # r6 ← data from address abc
and g0, 0xffff, g1 # g1 ← g0 AND 0xffff
addi r6, r8, r7 # r7 ← r6 + r8

Here, the two addi instructions following the first load and the and instruction following the second load are performed for free.

The other situation where scoreboardng can be useful for procedure optimization is when floating-point instructions are being executed. Floating-point operations are handled by a separate execution unit in the processor. So, non-floating point instructions can often be executed concurrently with floating-point instructions, providing that they do not use the same registers and do not use the arithmetic-logic unit (ALU). (A detailed description of the register-scoreboarding mechanism is given in Appendix C.)

2.4 INSTRUCTION POINTER

The instruction pointer (IP) is the address (in the address space) of the instruction currently being executed. This address is 32 bits; however, since instructions are required to be aligned on word boundaries in memory, the 2 least significant bits of the IP are always zero.

Instructions in the processor are one or two words long. The IP gives the address of the lowest order byte of the first word of the instruction.

The IP is stored in the processor and cannot be read directly. However, the IP-with-displacement addressing mode allows the IP to be used as an offset into the address space. This addressing mode can also be used with the lda (load address) instruction to read the current value of the IP.

When a break occurs in the instruction stream (due to an interrupt or a procedure call), the IP of the next instruction to be executed (i.e. the RIP) is stored in local register r2, which is then stored on the stack. Refer to Section 3 for further discussion of this operation.

2.5 ARITHMETIC CONTROLS

The processor’s arithmetic controls are made up of a set of 32 bits, which are cached on the processor chip in the arithmetic-controls register. Figure 3 shows the arrangement of the arithmetic controls bit. The arithmetic controls bits include condition code bits; floating-point control and status bits; integer control and status bits; and a bit that controls faulting on imprecise faults.
The processor sets or clears these bits to show the results of certain operations. For example, the processor modifies the condition code bits after each comparison operation to show the result of the comparison. Other arithmetic control bits, such as the floating-point fault masks, are set by the currently running program to tell the processor how to respond to certain fault conditions.

Note

The arithmetic status flags and the floating-point flags and masks are not defined in the 80960 architecture. They are an extension of the architecture, which is provided in the 80960KB processor to support floating-point operations. For implementations of the architecture that do not support floating-point operations, these flags and masks are reserved bits.

2.5.1 Initializing and Modifying the Arithmetic Controls

The state of the processor's arithmetic controls is undefined at processor initialization or on a processor reinitialize (initiated with a reinitialize processor IAC). Part of the initialization code should thus be to set the arithmetic controls to a specific state.

The arithmetic controls can be examined and modified using the modify AC (modac) instruction. This instruction uses a mask to allow specific bits to be checked and changes.

The processor automatically saves and restores the arithmetic controls when it services an interrupt or handles a fault. Here, the processor saves the current state of the arithmetic controls in an interrupt record or fault record, then restores the arithmetic controls upon returning from the interrupt or fault handler, respectively.

The modac instruction can be used to explicitly save and restore the contents of the arithmetic controls.
2.5.2 Functions of the Arithmetic Controls Bit

The functions of the various arithmetic controls bits are as follows:

Note
In the following discussion, some of the arithmetic control bits are referred to as "sticky flags". A sticky flag is one that the processor never implicitly clears. Once the processor sets a sticky flag to indicate that a particular condition has occurred, the flag remains set until the program explicitly clears it.

2.5.3 Condition Code Flags

The processor sets the condition code flags (bits 0-2) to indicate the results of certain instructions (usually compared instructions). Other instructions, such as conditional-branch instructions, examine these flags and perform functions according to their state. Once the processor has set these flags, it leaves them unchanged until it executes another instruction that uses these flags to store results.

These flags are used to show either true or false conditions or inequalities (greater-than, equal, or less-than conditions). To show true or false conditions, the flags are set as shown in Table 1.

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>true</td>
</tr>
<tr>
<td>000</td>
<td>false</td>
</tr>
</tbody>
</table>

The condition code flags are set as shown in Table 2 to show inequalities.

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>unordered</td>
</tr>
<tr>
<td>001</td>
<td>greater than</td>
</tr>
<tr>
<td>010</td>
<td>equal</td>
</tr>
<tr>
<td>100</td>
<td>less than</td>
</tr>
</tbody>
</table>
The terms ordered and unordered are used when comparing floating-point numbers. If, when comparing two floating-point values, one of the value is a NaN (not a number), the relationship is said to be “unordered”. Reference to the portion of Section 11 entitled “Comparison and Classification” for further information about the ordered and unordered conditions.

### 2.5.4 Arithmetic Status Flags

The processor uses the arithmetic status fields (bits 3-6) in conjunction with the classify instructions (classr and classr1) to show the class of a floating-point number. When executing these instructions, the processor sets the arithmetic status bits as shown in Table 3, according to the class of the value being classified.

The “s” bit is set to the sign of the value being classified.

<table>
<thead>
<tr>
<th>Arithmetic Status</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>s000</td>
<td>zero</td>
</tr>
<tr>
<td>s001</td>
<td>denormalized number</td>
</tr>
<tr>
<td>s010</td>
<td>normal finite number</td>
</tr>
<tr>
<td>s011</td>
<td>infinity</td>
</tr>
<tr>
<td>s100</td>
<td>quiet NaN</td>
</tr>
<tr>
<td>s101</td>
<td>signaling NaN</td>
</tr>
<tr>
<td>s110</td>
<td>reserved operand</td>
</tr>
</tbody>
</table>

### 2.5.5 Integer Overflow Mask

The integer overflow mask (bit 12) and the integer overflow flag (bit 8) are used in conjunction with the arithmetic integer-overflow fault. The mask bit masks the integer-overflow fault. When the fault is masked, the processor sets the integer overflow flag whenever an integer or decimal overflow occurs, to indicate that the fault condition has occurred even though the fault has been masked. If the fault is not masked, the fault is allowed to occur and the flag is not set. The integer overflow flag is a sticky flag. (Refer to the discussion of the arithmetic integer-overflow fault in Section 8 for more information about the integer overflow mask and flag.)
2.5.6 No Imprecise Faults Flag

The no imprecise faults flag (bit 15) determines whether or not imprecise faults are allowed to be raised. If set, faults are required to be precise; if clear, certain faults can be imprecise. (Refer to the portion of Section 8 titled “Precise and Imprecise Faults” for more information about this flag.)

2.5.7 Floating-Point Flags and Masks

The floating-point flags (bits 16 through 20) and masks (bits 24 through 28) perform the same functions as the integer overflow flag and mask, except they are used for operations on real (floating-point) numbers. When a mask bit is set, its associated floating-point fault is masked. If a mask bit is set, the processor sets the flag for the associate fault whenever the fault condition occurs. All the floating-point flag bits are sticky bits. Refer to the portion of Section 11 titled “Exceptions and Fault Handling” for a detailed discussion of the floating-point faults and their associated flag and mask bits in the arithmetic controls.

2.5.8 Floating-Point Normalizing Mode Flag

The floating-point normalizing mode flag (bit 29) determines where or not floating-point instructions are allowed to operate on denormalized numbers. If set, floating-point instructions are allowed to operate on denormalized numbers; if clear, the processor generates a floating reserved-operand fault when it detects denormalized numbers that are used as operands for floating-point instructions. (Refer to “Normalizing Mode” in section 11 for more information on the use of this flag.)

2.5.9 Floating-Point Rounding Control

The floating-point rounding control fields (bits 31-30) indicates which rounding mode is in effect for floating point computations. These bits are set as shown in Table 4, depending on the rounding mode to be selected.

<table>
<thead>
<tr>
<th>Rounding Control</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>round to nearest (even)</td>
</tr>
<tr>
<td>01</td>
<td>Round down (toward negative infinity)</td>
</tr>
<tr>
<td>10</td>
<td>Round up (toward positive infinity)</td>
</tr>
<tr>
<td>11</td>
<td>Truncate (round toward zero)</td>
</tr>
</tbody>
</table>

(Refer to “Rounding Control” in Section 11 for more information on the use of the floating-point rounding control bits.)

All the unused bits in the AC register are reserved and must be set to 0.

3-15
2.6 PROCESS AND TRACE CONTROLS

The processor's process controls and trace controls are also cached on the processor chip. The processor controls are a set of 32 bits that control or show the current execution state of the processor. The process controls are described in detail in Section 6.

The trace controls are a set of 32 bits that control the tracing facilities of the processor. The trace controls are described in Section 10.

2.7 INSTRUCTION CACHING

The processor provides a 512-byte cache for instructions. When the processor fetches an instruction or group of instructions from memory, they are stored in this cache before being fed into the instruction-execution pipeline. The processor manages this cache transparently from the program being run.

This instruction cache is a read-only cache, meaning that once bytes from the instruction stream are written into the instruction cache, they cannot be changed. Because of this, the processor does not support self-modified programs in a transparent fashion. The only way to change the instruction stream once it has been written into the instruction cache is to purge the instruction cache. The IAC message “purge instruction cache” is provided for this purpose, as described in Section 12.

Note
The purge instruction cache IAC is not defined in the 80960 architecture. It is an implementation-dependent feature of the 80960KB processor.

3.0 PROCEDURE CALLS

This section describes the 80960KB processor's procedure call and stack mechanism. It also describes the supervisor call mechanism, which provides a means of calling privileged procedure such as kernel services.

3.1 TYPES OF PROCEDURE CALLS

The processor supports three types of procedure calls:

- Local call
- System call
- Branch and link

A local call uses the processor's call/return mechanism, in which a new set of local registers and a new frame on the stack are allocated for the called procedure. A system call is similar to a local call, however, it provides access to procedures through a system procedure table. The most important use of a system call is to call privileged procedures called supervisor procedures. A system call to a
supervisor procedure is called a supervisor call. A branch and link is merely a branch to a new instruction with the return IP stored in a global register.

In this section, the call/return mechanism is introduced first and is followed by a discussion of how this mechanism is used to make local calls and system calls.

**Note**
The processor’s interrupt- and fault-handling mechanisms are implicit procedure calls. These implicit calls are described in detail in Sections 7 and 8, respectively.

### 3.2 CALL/RETURN MECHANISM

The processor’s call/return mechanism has been designed to simplify procedure calls and to provide a flexible method for storing and handling variables that are local to a procedure.

Two structures support this mechanism: the local registers (on the processor chip) and the procedure stack (in memory). Figure 4 shows the relationship of the local registers to the procedure stack.

For each procedure, the processor automatically allocates a set of local registers and a frame on the procedure stack. Since the local registers are on-chip, they provide fast-access storage for local variables. If additional space for local variables is required, it can be allocated in stack frame.

When a procedure call is made, the processor automatically saves the contents of the local registers and the stack frame for the calling procedure and sets up a new set of local registers and a new stack frame for the called procedure.
This procedure call mechanism provides two benefits. First, it provides a structure for storing a virtually unlimited number of local variables for each procedure: the on-chip local registers provide quick access to often-used variables and the stack provides space for additional variables.

Second, a program does not have to explicitly save and restore the variables stored in the local registers and stack frames. The processor does this implicitly on procedure calls and on returns.

A detailed description of the call/return mechanism is given in the following paragraphs.

### 3.2.1 Local Registers and the Procedure Stack

For each procedure, the processor allocates a set of 16 local registers. Three of these registers (r1, r2 and r3) are reserved for linkage information to the procedures together. The remaining 13 local registers are available for general storage of variables.

The processor maintains a procedure stack in memory for use when performing local calls. This stack can be located anywhere in the address space and grows from low addresses to high addresses.

The stack consists of contiguous frames, one frame for each active procedure. As shown in Figure 5, each stack frame provides a save area for the local registers and an optional area for additional variables.

To increase the speed of procedure calls, the 80960KB processor provides four sets of local registers. Thus, when a procedure call is made, the contents of the current set of local registers often do not have to be stored in the procedure stack. Instead, a new set of local registers is assigned to the called procedure. When procedure calls are made greater than four deep, the processor automatically stores the contents of the oldest set of local registers on the stack to free up a set of local registers for the most recently called procedure.

Refer to the subsection “Mapping the Local Registers to the Procedure Stack” for further discussion of the relationship between the local register sets and the procedure stack.

### 3.2.2 Procedure Linking Information

Global register g15 (FP) and local registers r0 (PFP), r1 (SP) and r2 (RIP) contain information to link procedures together and to link the local registers to the procedure stack. The following paragraphs describe this linkage information.

### 3.2.3 Frame Pointer

The FP is the address of the first byte of the current (topmost) stack frame. On procedure calls, the FP for the new frame is stroed in global register g15; on returns, the FP for the previous frame is restored in g15.
The 80960KB processor aligns each new stack frame on a 64-byte boundary. Since the resulting FP always points to a 64-byte boundary, the processor ignores the 6 low-order bits of the FP and interprets them to be zero.
3.2.4 Stack Pointer

The procedure stack grows upward (i.e. toward higher addresses). The SP points to the next available byte of the stack frame, which can also be thought of as the last byte of the stack frame plus one. To determine the initial SP value, the processor adds 64 to the FP.

If additional space is needed on the stack for local variables, the SP may be incremented in one-byte increments. For example, the following instruction adds six words of additional space to the stack:

```
add sp, 24, sp  # sp ← sp + 24
```

With the Intel 80960KB Assembler, the keyword “sp” stands for register r1.

3.2.5 Padding Area

When the processor creates a new frame on a procedure call, it will, if necessary, add a padding area to the stack so that the new frame starts on a 64 byte boundary. To create the padding area, the processor rounds off the SP for the current stack frame (the value in r1) to the next highest 64 byte boundary. This value becomes the FP for the new stack frame.

3.2.6 Previous Frame Pointer

The PFP is the address of the first byte of the previous stack frame. Since the 80960KB ignores the 6 low-order bits of the FP, only the 26 most-significant bits of the PFP are stored here. The 4 least-significant bits of r0 are then used to store return status information.

3.2.7 Return Status and Prereturn-Trace Information

Bits 0 through 2 of local register r0 contain return status information for the calling procedure and bit 3 contains the prereturn-trace flag. When a procedure call is made (either explicit or implicit), the processor records the call type in the return status field. The processor then uses this information to select the proper return mechanism when returning to the calling procedure.

Table 5 shows the encoding of the return status field according to the different types of calls that the processor supports. Of the five types of calls allowed, the fault call (described in Section 8) and the interrupt and stopped-interrupt calls (described in Section 7) are implicit calls that the processor initiates. The local call (described in this section) is an explicit call that a program initiates using the call or callx instruction. The supervisor call (described at the end of this section in the portion “User-Supervisor Protection Model”) is an explicit call that a program makes using the calls instruction.
Table 5. Encoding of Return Status Field

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Call Type</th>
<th>Return Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Local call or supervisor call made from the supervisor mode</td>
<td>Local return</td>
</tr>
<tr>
<td>001</td>
<td>Fault call</td>
<td>Fault return</td>
</tr>
<tr>
<td>010</td>
<td>Supervisor call from user mode, trace was disabled before call</td>
<td>Supervisor return, with the trace enable flag in the process controls set to 0 and the execution mode flag set to 0</td>
</tr>
<tr>
<td>011</td>
<td>Supervisor call from user mode, trace was enabled before call</td>
<td>Supervisor return, with the trace enable flag in the process controls set to 1 and the execution mode flag set to 0</td>
</tr>
<tr>
<td>100</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Stopped-interrupt call</td>
<td>Stopped-interrupt return</td>
</tr>
<tr>
<td>111</td>
<td>Interrupt call</td>
<td>Interrupt return</td>
</tr>
</tbody>
</table>

The third column of Table 5 shows the type of a return action that the processor takes depending on the state of the return status field.

The processor records two versions of the supervisor call: one for when the trace-enable flag in the process controls is set prior to a supervisor call and one for when the flag is clear prior to the call. The trace controls are described in detail in Section 9.

The prereturn-trace flag is used in conjunction with the call-trace and prereturn-trace modes. If the call-trace mode is enabled when a call is made, the processor sets the prereturn-trace flag; otherwise it clears the flag. Then, if this flag is set and the prereturn-trace mode is enabled, a prereturn trace event is generated on a return before any actions associated with the return operation are performed. Refer to Section 9 for a detailed discussion of the interaction of the call-trace and prereturn-trace modes and the prereturn-trace flag.

3.2.8 Return Instruction Pointer

The RIP is the address of the instruction that the processor is to execute after returning from a procedure call. This instruction is the instruction that follows the procedure call instruction.

Since the processor uses the same procedure call mechanism to make implicit procedure calls to service faults and interrupts, programs should not use register r2 for purposes other than to hold the RIP.
3.2.9 Mapping the Local Registers to the Procedure Stack

The availability of multiple register sets cached on the processor chip and the saving and restoring of these register sets in stack frames should be transparent to most programs. However, the following additional information about how the local registers and procedures stack are mapped to one another can help avoid problems.

Since the local-register sets reside on the processor chip, the processor will often not have to access the stack frame in the procedure stack, even though space has been allocated on the stack for the current frame. The processor only accesses the current frame in the procedure stack in the following instances:

1. to read or write variables other than those held in the local registers, or
2. to read local registers that were stored in the procedure stack due to the nesting of procedures calls more than four deep.

This method of mapping the local registers to the register-save areas in the procedure stack has several implications. First, storing information in a local register does not guarantee that it will be stored in its associated word in the current stack frame. Likewise, storing information in the first 16 words of a stack frame does not guarantee that the local registers associated with the stack frame are modified.

Second, if you try to read the contents of the current set of local registers through a memory access to the first 16 words of the current stack frame, you may not get the expected result. This is also true if you try to read the contents of a previously stored set of local registers through a memory address to its associated stack frame.

The processor automatically stores the contents of a local register set into the register-save area of its associated stack frame only if the nesting of procedure calls (local or supervisor) is deeper than the number of local register sets.

Occasionally, it is necessary to have the contents of all local registers sets match the contents of the register-save areas in their associated stack frames. For example, when debugging software it may be necessary to trace the call history back through the nested procedures. This can not be done unless the cached local-register frames are flushed (i.e., written out to the procedure stack).

The processor provides the `flushreg` (flush local registers) instruction to allow voluntary flushing of the local registers. This instruction causes the contents of all the local-register sets, except the current set, to be written to their associated stack frames in memory.

Third, if you need to modify the previous FP in register r0, you should precede this operation with the `flushreg` instruction, or else the behavior of the `ret` (return) instruction is not predictable.

Fourth, local registers should not be used for passing parameters between procedures. (Parameters passing is discussed in the following subsection.)
Fifth, when a set of local registers is assigned to a new procedure, the processor may not clear or initialize these registers. The initial contents of these registers are therefore unpredictable. Also, the processor does not initialize the local register-save area in the newly created stack frame for the procedure, so its contents are equally unpredictable.

### 3.3. LOCAL CALL

A local call is made using either of two local call instructions: call and callx. These instructions initiate a procedure call using the call/return mechanism described earlier in this section.

The call instruction specifies the address of the called procedures as the IP plus a signed, 24-bit displacement (i.e., $-2^{23}$ to $2^{23} - 4$).

The callx instruction allows any of the addressing modes to be used to specify the procedure address. The IP with displacement addressing mode allows full 32-bit IP relative addressing.

The ret instruction initiates a procedure switch back to the last procedure that issued a call.

#### 3.3.1 Local Call Operation

During a local call, the processor performs the following operations:

1. Stores the RIP in current local-register r2.
2. Allocates a new set of local registers for the called procedure.
3. Allocates a new frame on the procedure stack.
4. Changes the instruction pointer to point to the first instruction in the called procedure.
5. Stores the PFP in new local-register r0.
6. Stores the FP for the new frame in global register g15.
7. Allocates a save area for the new local registers in the new stack frame.
8. Stores the SP in new local-register r1.

#### 3.3.2 Local Return Operation

On a return, the processor performs these operations:

1. Sets the FP in global register g15 to the value of the PFP in current local-register r0.
2. Deallocates the current local registers for the procedure that initiated the return and switches to the local registers assigned to the procedure being returned to.
3. Deallocates the stack frame for the procedure that initiated the return.
4. Sets the IP to the value of the RIP in new local-register r2.

3-23
The algorithms that the \texttt{call}, \texttt{callx}, and \texttt{ret} instructions use are described in greater detail in Section 10.

### 3.4. PARAMETER PASSING

The processor supports two mechanisms for passing parameters between procedures: global registers and argument list.

#### 3.4.1 Passing Parameters in Global Registers

The global registers provide the fastest method of passing parameters. Here, the calling procedure copies the parameters to be passed into global registers. The called procedure then copies the parameters (if necessary) out of the global registers after the call.

On a return, the called procedure can copy result parameters into global registers prior to the return, with the calling procedure copying them out of the global registers after the return.

#### 3.4.2 Passing Parameters in an Argument List

When more parameters need to be passed than will fit in the global registers, they can be placed in an argument list. This argument list can be stored anywhere in memory providing that the procedure being called has a pointer to the list. Commonly, a pointer to the argument list is placed in a global register.

Parameters can also be returned to the calling procedure through an argument list. Here again, a pointer to the argument is generally returned to the calling procedure through a global register.

The argument list method of passing parameters should be thought of as an escape mechanism and used only when there are not enough global registers available for passing parameters.

#### 3.4.3 Passing Parameters Through the Stack

A convenient place to store an argument list is in the stack frame for the calling procedure. Storing the argument list in the stack provides the benefit of having the list automatically deallocated upon returning from the procedure that set up the list. Space for the argument list is created by incrementing the \texttt{SP}, as described earlier in this chapter in the section titled "Stack Pointer".

Parameters can also be returned to the calling procedure through an argument list in the stack. However, care should be taken when doing this. The return argument list must not be placed in the frame for the called procedure, since this frame is deallocated on the return. Also, if the return list is to be placed in the frame of the calling procedure, the calling procedure must allocate space for this list prior to making the call.
3.5 SYSTEM CALL

A system call is made using the call system instruction calls. This call is similar to a local call except that the processor gets the IP for the called procedure from a data structure called the system procedure table. (System calls are sometimes referred to in this chapter as “system procedure-table calls”.)

Figure 6 illustrates the use of the system procedure table in a system call. The calls instruction requires a procedure-number operand. This procedure number provides an index into the system procedure table, which contains IPs for specific procedures.

The system call mechanism supports two types of procedure calls: local calls and supervisor calls. A local call is the same as that made with the call and calIx instructions, except that the processor gets the IP of the called procedure from the system procedure table. The supervisor call differs from the local call in two ways: (1) it causes the processor to switch to another stack (called the supervisor stack), and (2) it causes the processor to switch to a different execution mode.

The system call mechanism offers two benefits. First, it supports portability for application software. System calls are commonly used to call kernel services. By calling these services with a procedure number rather than a specific IP, applications software does not have to be changed each time the implementation of the kernel services is modified.
Second, the ability to switch to a different execution mode and stack allows kernel procedures and data to be insulated from applications code. This benefit is described in more detail later in “User-Supervisor-Protection-Model” later in this chapter.

### 3.6 SYSTEM PROCEDURE TABLE

The system procedure table is a general structure, which the processor uses in two ways. The first way is as a place for storing IPs for kernel procedures, which can then be accessed through the system call mechanism. The processor gets a pointer to the system procedure table from the initial memory image (IMI) as described in Section 6, “System Data-Structure Pointers”.

The second way a system procedure table is used is as a place for storing IPs for fault handler procedures. Here, the processor gets a pointer to the system procedure table from entries in the fault table, as described in Section 8, “Fault-Table Entries”.

The structure of the system procedure table is shown in Figure 7. The following paragraphs describe the fields in this table.

#### 3.6.1 Procedure Entries

The procedure entries specify the target IPs for the procedures that can be accessed through the system procedure table. Each entry is made up of an address (or IP) field and a type field. The address field gives the address of the first instruction of the target procedure. Since all instructions are word aligned, only the 30 most-significant bits of the address are given. The processor automatically provides zeros for the least-significant bits.

The procedure entry type field indicates the type of call to execute: local or supervisor. The encodings of this field are shown in Table 6.

<table>
<thead>
<tr>
<th>Entry Type Field</th>
<th>Procedure Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>local procedure</td>
</tr>
<tr>
<td>01</td>
<td>reserved</td>
</tr>
<tr>
<td>10</td>
<td>supervisor procedure</td>
</tr>
<tr>
<td>11</td>
<td>reserved</td>
</tr>
</tbody>
</table>
3.6.2 Supervisor Stack Pointer

When a supervisor call is made, the processor switches to a new stack called the supervisor stack. The processor gets a pointer to this stack from the supervisor-stack-pointer entry (bytes 12-15, bits 2-31) in the system procedure table. Since stack frames are word aligned, only the 30 most-significant bits of the supervisor stack pointer are given.

3.6.3 Trace Control Flag

The trace-control flag (byte 12, bit 0) specifies the new value of the trace-enable flag when a supervisor call causes a switch from user mode to supervisor mode. The use of this bit is described in Section 9.
3.6.4 System Call to a Local Procedure

When a calls instruction references a procedure entry designed as a local type (002), the processor executes a local call to the procedure selected from the system procedure table. Neither a mode switch nor a stack switch occurs.

The ret instruction permits returns from either a local procedure or a supervisor procedure. The return status field in local register r0 determines the type of return action that the processor is to take. If the return status field is set to 0002, a local return is executed. In a local return, no stack or mode switching is carried out.

3.7 USER-SUPERVISOR PROTECTION MODEL

The processor provides a mode and stack switching mechanism called the user-supervisor protection model. This protection model allows a system to be designed in which kernel code and data reside in the same address space as user code and data, but access to the kernel procedures (called supervisor procedures) is only allowed through a tightly controlled interface. This interface is provided by the system procedure table.

The user-supervisor protection model also allows kernel procedures to be executed using a different stack (the supervisor stack) than is used to execute applications program procedures. The ability to switch stacks helps maintain the integrity of the kernel. For example, it would allow system debugging software or a system monitor to be accessed, even if an applications program crashes.

3.7.1 User and Supervisor Modes

When using the user-supervisor protection model, the processor can be in either of two execution modes: user or supervisor. The difference between the two modes in that when in the supervisor mode, the processor

- switches to the supervisor stack, and
- may execute a set of supervisor only instructions.

Note

In the 80960KB implementation of the 80960 architecture, the only supervisor-only instruction is the modify process control instruction (modpc).

3.7.2 Supervisor Calls

Mode switching between the user and supervisor execution modes is accomplished through a supervisor call. A supervisor call is a call executed with the calls instruction that references a supervisor procedure in the system procedure table (i.e. a procedure with an entry type 102).

When the processor is in the user mode and it executes a calls instruction, the processor performs the following actions:
• It switches to supervisor mode
• It switches to the supervisor stack
• It sets the return status field in register R0 of the calling procedure to 01X₂, indicating that a mode and stack switch has occurred.

The processor remains in the supervisor mode until a return is performed from the procedure that caused the original mode switch. While in the supervisor mode, either the local call instructions (call and callx) or the calls instruction can be used to call supervisor procedures.

(The call and callx instructions call local (or user) procedures in user mode and supervisor procedures in supervisor mode. There is no stack or processor state switching associated with these instructions.)

When a ret instruction is executed and the return status field is set to 01X₂, the processor performs a supervisor return. Here, the processor switches from the supervisor stack to the local stack, and the execution mode iswitch from supervisor to user.

3.7.3 Supervisor Stack

When using the user-supervisor mechanism, the processor maintains separate stacks in the address space, one for procedures executed in the user mode (local procedures) and another for procedures executed in the supervisor mode (supervisor procedures). When in the user mode, the local procedure stack described at the beginning of this section is used. When a supervisor call is made, the processor switches to the supervisor stack. It continues to use the supervisor stack until a return is made to the user mode.

The structure of the supervisor stack is identical to that of the local procedure stack (shown in Figure 5). The processor obtains the SP for the supervisor stack from the system procedure table. When a supervisor call is executed while in the user mode (causing a switch to the supervisor stack), the processor aligns this SP to the next 64 byte boundary to form the new FP for the supervisor stack. When a local call or supervisor call is made while in the supervisor mode, the processor aligns the SP in the current frame of the supervisor stack to the next 64 byte boundary to form the FP pointer. This operation allows supervisor procedures to be called from supervisor procedures.

3.7.4 Hints on Using the User-Supervisor Protection Model

The user-supervisor has three basic uses in an embedded system application:
1. to allow the modpc instruction to be used,
2. to allow kernel code to use a separate stack from the applications code, and
3. to allow an external memory management unit (MMU) to provide protection for kernel code and data.

If an application does not require any of the above features, it can be designed to not use the user-supervisor protection model. Here, all procedure calls are to local procedures. If the system table is used, all the entries must be the local type (i.e. entry type 00₂).
If access to the \texttt{modpc} instruction is required, but the other two features are not, it is suggested that the system be designed to always run in supervisor mode. At initialization, the processor automatically places itself in supervisor mode, prior to executing the first instruction. The processor then remains in supervisor mode indefinitely, as long as no action is taken to change the execution mode to user mode (i.e. using the \texttt{modpc} instruction to change the execution mode bit of the process controls to 0). With this technique, all of the procedure calling instructions (\texttt{call}, \texttt{callx}, and \texttt{calls}) can be used. The processor only uses one stack, which is considered the supervisor stack. It gets the supervisor stack pointer from local register \( r2 \). (Prior to making the first procedure call, the supervisor stack pointer must be loaded into \( r2 \)).

The processor does not support the last use of the user-supervisor protection model directly. In other words, the processor does not provide a pin or other device that indicates to external hardware when a mode switch has occurred. Several techniques are available to perform this operation, which are beyond the scope of this discussion.

\section*{3.8 BRANCH AND LINK}

The \texttt{bal} (branch and link) and \texttt{balx} (branch and link extended) instructions provide an alternate method of making procedure calls. These instructions save the address of the next instruction (RIP) in a specified location, then branch to a target instruction or set of instructions. The state of the local registers and stack remains unchanged. (For the \texttt{bal} instruction, the RIP is automatically stored in global register \( g14 \); for the \texttt{balx} instruction, the location of the RIP is specified with one of the instruction operands.)

A return is accomplished with a \texttt{bx} (branch extended) instruction, where the address of the target instruction is the one saved with the branch and link instruction.

Branch and link procedure calls are recommended for calls to procedures that (1) do not call other procedures (i.e. for procedure calls that do not result in nesting of procedures) and (2) do not need many local variables (i.e. allocation of a new set of local registers does not provide any benefit). Here, local registers as well as global registers can be used for parameter passing.

\section*{4.0 DATA TYPES AND ADDRESSING MODES}

This section describes the data types that the 80960KB processor recognizes and the addressing modes that are available for accessing memory locations.

\subsection*{4.1 DATA TYPES}

The processor defines and operates on the following data types:

\begin{itemize}
  \item Integer (8, 16, 32 and 64 bits)
  \item Ordinal (8, 16, 32 and 64 bits)
  \item Real (32, 64 and 80 bits)
  \item Decimal (ASCII digits)
\end{itemize}
• Bit Field
• Triple-Word (96 bit)
• Quad-Word (128 bit)

Note
The real and decimal data types are not defined in the 80960 architecture. They are supported in the
80960KB processor, but not in the 80960KA processor.

The integer, ordinal, real, and decimal data types can be thought of as numeric data types because
some operations on these data types produce numeric results (e.g. add, subtract).

The remaining data types (bit field, triple word, and quad word) represent groupings of bits or bytes
that the processor can operate on as a whole, regardless of the nature of the data contained in the
group. These data types facilitate the moving of blocks of bits or bytes.

4.1.1 Integers

Integers are signed whole numbers, which are stored and operated on in two’s complement format.
The processor recognizes four sizes of integers: 8 bit (byte integers), 16 bit (short integers), 32 bit
(integers) and 64 bit (long integers). Figure 9 shows the formats for the four integer sizes and the
ranges of values allowed for each size.

Figure 9. Integer Format and Range
4.1.2 Ordinals

Ordinals are a general-purpose data type. The processor recognizes four sizes of ordinals: 8 bit (byte ordinals), 16 bit (short ordinals), 32 bit (ordinals), and 64 bit (long ordinals). Figure 10 shows the formats for the four ordinal sizes and the ranges of numeric values allowed for each size.

The processor uses ordinals for both numeric and non-numeric operations. For numeric operations, ordinals are treated as unsigned whole numbers. The processor provides several arithmetic instructions that operate on ordinals. For non-numeric operations, ordinals contain bit fields, byte strings, and Boolean values.

When ordinals are used to represent Boolean values, 12 represents a TRUE and a 02 represents a FALSE.

4.1.3 Reals

Reals are floating-point numbers. The processor recognizes three sizes of reals: 32 bit (reals), 64 bit (long reals) and 80 bit (extended reals). The real-number format conforms to ANSI/IEEE Std. 754-1985, the IEEE Standard For Binary Floating-Point Arithmetic. Real numbers are discussed in greater detail in Section 11.
4.1.4 Decimals

The processor provides three instructions that perform operations on decimal values when the values are presented in ASCII format. Figure 10 shows the ASCII format. Figure 11 shows the ASCII format for decimal digits. Each decimal digit is contained in the least-significant byte of an ordinal (32 bits). The decimal digit must be of the form `0011dddd`, where `dddd` is a binary-coded decimal value from 0 to 9. For decimal operations, bits 8 through 31 of the ordinal containing the decimal digit are ignored.

![Figure 11. Decimal Format](image)

4.1.5 Bits and Bit Fields

The processor provides several instructions that perform operations on individual bits or fields of bits within an ordinal (32 bit) operand. Figure 12 shows these data types.

![Figure 12. Bits and Bit Fields](image)

An individual bit is specified for a bit operation by giving its number in the ordinal in which it resides. The least-significant bit of a 32-bit ordinal is bit 0; the most-significant bit is bit 31.

A bit field is a contiguous sequence of bits of from 0 to 32 bits in length within a 32-bit ordinal. A bit field is defined by giving its length in bits and the bit number of its lowest-numbered bit.

A bit field cannot span a register boundary.

4.1.6 Triple and Quad Words

Triple and quad words refer to consecutive bytes in memory or in registers: a triple word is 12 bytes and a quad word is 16 bytes. These data types facilitate the moving of blocks of bytes. The triple-word data type is useful for moving extended-real numbers (80 bits).
The quad-word instructions (ldq, stq, and movq) offer the most efficient way to move large blocks of data.

### 4.2 BYTE, WORD, AND BIT ADDRESSING

The processor provides instructions for moving blocks of data values of various lengths from memory to registers (load) and from registers to memory (store). The allowable sizes for blocks are bytes, half-words (2 bytes), words (4 bytes), double words, triple words, and quad words. For example, the stl (store long) instruction stores an 8-byte (double word) block of data in memory.

When a block of data is stored in memory, the least-significant byte of the block is stored at a base memory address and the more significant bytes are stored at successively higher addresses.

When loading a byte, half-word, or word from memory to a register, the least-significant bit of the block is always loaded in bit 0 of the register. When loading double words, triple words, and quad words, the least-significant word is stored in the base register. The more significant words are then stored at successively higher numbered registers. Double words, triple words, and quad words must also be aligned in registers to natural boundaries as described in the section "Register Alignment".

Bits can only be addressed in data that resides in a register. Bit 0 in a register is the least-significant bit and bit 31 is the most-significant bit.

### 4.3 ADDRESSING MODES

The processor offers 11 modes for addressing operands. These modes are grouped as follows:

- Literal
- Register
- Absolute
- Register Indirect
- Register Indirect with Index
- Index with Displacement
- IP with Displacement

Most of the instructions use only the first two modes (literal and register). The remaining modes are used for memory related instructions.

Table 8 shows all the addressing modes, a brief description of the elements of the address in each mode, and the assembly-code syntax for each mode.
Table 8. Addressing Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Assembler Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>Register</td>
<td>register</td>
<td>reg</td>
</tr>
<tr>
<td>Absolute offset</td>
<td>offset</td>
<td>exp</td>
</tr>
<tr>
<td>Register Indirect</td>
<td>abase</td>
<td>(reg)</td>
</tr>
<tr>
<td>Register Indirect with offset</td>
<td>abase + offset</td>
<td>exp (reg)</td>
</tr>
<tr>
<td>Register Indirect with index</td>
<td>abase + (index*scale)</td>
<td>(reg) [reg*scale]</td>
</tr>
<tr>
<td>Register Indirect with index and displacement</td>
<td>abase + (index*scale) + displacement</td>
<td>exp (reg) [reg*scale]</td>
</tr>
<tr>
<td>Index with displacement</td>
<td>(index*scale) + displacement</td>
<td>exp [reg*scale]</td>
</tr>
<tr>
<td>IP with displacement</td>
<td>IP + displacement + 8</td>
<td>exp (IP)</td>
</tr>
</tbody>
</table>

4.3.1 Literals

The processor recognizes two types of literals: ordinal literal and floating-point literal. An ordinal literal can range from 0 to 31 (5 bits). When an ordinal literal is used as an operand, the processor expands it to 32 bits by adding leading zeros. If the instruction defines an operand larger than 32 bits, the processor zero-extends the value to the operand size. If an ordinal literal is used in an instruction that requires integer operands, the processor treats the literal as a positive integer value.

The processor also recognizes two floating-point literals (+0.0 and +1.0). These floating-point literals can only be used with floating-point instructions. As with the ordinal literals, the processor converts the floating-point literals to the operand size specified by the instruction.

A few of the floating-point instructions use both floating-point and non-floating-point operands (e.g. the convert integer-to-real instructions). Ordinal literals can be used in these instructions for non-floating-point operands.

Note

Floating-point literals are not defined in the 80960 architecture.
4.3.2 Register

A register is referenced as an operand by giving the register number (e.g. g0, r5, fp3). Both floating-point and non-floating-point instructions can reference global and local registers in this way. However, floating-point registers can only be referenced in conjunction with a floating-point instruction.

4.3.3 Absolute

Absolute addressing is used to reference a memory location directly as an offset from address 0 of the address space, ranging from $-2^{31}$ to $2^{31} - 1$. Typically, an assembler will allow absolute addresses to be specified through arithmetic expressions (e.g. $x + 44$), symbolic labels, and absolute values.

At the machine-level, two absolute-addressing modes are provided, depending on the instruction format (i.e. MEMA or MEMB). For the MEMA format, the offset is an ordinal number ranging from 0 to 2048; for the MEMB format, the offset is an integer (called a displacement) ranging from $-2^{31}$ to $2^{31} - 1$. After evaluating an absolute address, the assembler will convert the address into an offset and select the appropriate machine-level instruction type and addressing mode. (The machine-level addressing modes and instruction formats are described in Appendix B).

4.3.4 Register Indirect

The register indirect addressing modes allow an address to be specified with an ordinal value (32 bits) in a register or with an offset or a displacement added to a value in a register. Here, the value in the register is referred to as the address base (abase).

Again, an assembler will allow the offset and displacement to be specified with an expression or symbolic label, then evaluate the address to determine whether an offset or a displacement is appropriate.

4.3.5 Register Indirect with Index

The register indirect with index addressing modes allow a scaled index to be added to the value in a register. The index is specified by means of a value placed in a register. This index value is then multiplied by the scale factor. The allowable scale factors are 1, 2, 4, 8, and 16.

A displacement may also be added to the abase value and scaled index.

4.3.6 Index with Displacement

A scaled index can also be used with a displacement alone. Again, the index is contained in a register and is multiplied by a scaling constant before the displacement is added to it.
4.3.7 IP with Displacement

The IP with displacement addressing mode is often used with load and store instructions to make them IP relative.

Note that with this mode the displacement plus a constant of 8 is added to the IP of the instruction.

5.0 INSTRUCTION SET SUMMARY

This section provides an overview of the instruction set for the 80960KB processor. Included is a discussion of the instruction format and a summary of the instruction groups and the instructions in each group.

Section 10 gives detailed descriptions of each of the instructions. The instructions are listed in this section in alphabetical order. Included for each instruction are the assembly-language format, the action taken when the instruction is executed, and examples of how the instruction might be used.

Appendix C provides a detailed description of the factors that affect instruction timing. It also gives the number of clock cycles required for each instruction.

5.1. INSTRUCTION FORMATS

Instructions are described in two formats: assembly language and machine level.

5.1.1 Assembly-Language Format

The instructions are referred to by their assembly-language mnemonics. For example, the add ordinal instruction is referred to as the addo instruction.

An assembly-language statement consists of an instruction mnemonic, followed by from 0 to 3 operands, separated by commas. The following example shows the assembly-language statement for the addo instruction:

```
addo g5, g9, g7
```

Here, the ordinal operands in global registers g5 and g9 are added together and the result is stored in g7.

A detailed description of the nomenclature used to describe assembly-language instructions is given in Section 10.
5.1.2 Machine Formats

At the machine level of the processor, all instructions are word aligned. Most of the instructions are one word long, although some addressing modes make use of a two-word format.

There are four instruction formats: register (REG), compare and branch (COBR), control (CTRL), and memory (MEM). Each instruction uses one of these formats, which is determined by the opcode field of the instruction.

The machine-level formats for the instructions are described in detail in Appendix B.

5.2 INSTRUCTION GROUPS

The 80960KB processor implements all the instructions in the 80960 instruction set, which includes all of the data movement, arithmetic, logical, and program control instructions commonly found in computer architectures. The processor also includes a set of floating-point instructions and several instructions to handle architectural extensions found in the processor.

The 80960 instruction set is made up of the following group of instructions:

- Data Movement
- Arithmetic (Ordinal and Integer)
- Logical
- Bit and Bit Field
- Comparison
- Branch
- Call/Return
- Fault
- Debug
- Processor Management

The instruction-set extensions found in the 80960KB processor include the following groups of instructions:

- Integer to Real Conversion
- Floating Point
- Synchronous Move and Load
- Decimal.

Table 9 and 10 give a summary of the 80960 instructions and the 80960KB instruction-set extensions, respectively. The actual number of instructions is greater than those shown in this list, because for
some operations, several different instructions are provided to handle different operand size, data types, or branch conditions.

Table 9. Summary of the 80960 Instruction Set

<table>
<thead>
<tr>
<th>Data Movement</th>
<th>Arithmetic</th>
<th>Logical</th>
<th>Bit and Bit Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Add</td>
<td>And</td>
<td>Set Bit</td>
</tr>
<tr>
<td>Store</td>
<td>Subtract</td>
<td>Not And</td>
<td>Clear Bit</td>
</tr>
<tr>
<td>Move</td>
<td>Multiply</td>
<td>And Not</td>
<td>Not Bit</td>
</tr>
<tr>
<td>Load Address</td>
<td>Divide</td>
<td>Or</td>
<td>Check Bit</td>
</tr>
<tr>
<td></td>
<td>Remainder</td>
<td>Exclusive Or</td>
<td>Alter Bit</td>
</tr>
<tr>
<td></td>
<td>Modulo</td>
<td>Not Or</td>
<td>Scan For Bit</td>
</tr>
<tr>
<td></td>
<td>Shift</td>
<td>Or Not</td>
<td>Scan Over Bit</td>
</tr>
<tr>
<td></td>
<td>Extended</td>
<td>Nor</td>
<td>Extract</td>
</tr>
<tr>
<td></td>
<td>Multiply</td>
<td>Exclusive Nor</td>
<td>Modify</td>
</tr>
<tr>
<td></td>
<td>Extended</td>
<td>Not</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Divide</td>
<td>Nand</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rotate</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Branch</th>
<th>Call/Return</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>Unconditional</td>
<td>Call</td>
<td>Conditional Fault</td>
</tr>
<tr>
<td>Conditional</td>
<td>Branch</td>
<td>Call Extended</td>
<td>Synchronize Faults</td>
</tr>
<tr>
<td>Compare</td>
<td>Conditional Branch</td>
<td>Call System</td>
<td></td>
</tr>
<tr>
<td>Compare and</td>
<td>Compare and</td>
<td>Return</td>
<td></td>
</tr>
<tr>
<td>Increment</td>
<td>Branch</td>
<td>Branch and Link</td>
<td></td>
</tr>
<tr>
<td>Compare and</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decrement</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Debug</th>
<th>Processor</th>
<th>Miscellaneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modify Trace Controls</td>
<td>Modify Arithmetic Controls</td>
<td>Atomic Add</td>
</tr>
<tr>
<td>Mark</td>
<td>Modify Process Controls</td>
<td>Atomic Modify</td>
</tr>
<tr>
<td>Force Mark</td>
<td>Flush Local Registers</td>
<td>Scan Byte For</td>
</tr>
<tr>
<td></td>
<td>Test Condition Code</td>
<td>Equal</td>
</tr>
</tbody>
</table>
Table 10. Summary of the 80960KB Instruction-Set Extensions

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Floating Point</th>
<th>Synchronous</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert Real to Integer</td>
<td>Move Real</td>
<td>Synchronous Load</td>
<td>Move</td>
</tr>
<tr>
<td>Convert Integer to Real</td>
<td>Add</td>
<td>Synchronous Move</td>
<td>Add With Carry</td>
</tr>
<tr>
<td></td>
<td>Subtract</td>
<td></td>
<td>Subtract With Carry</td>
</tr>
<tr>
<td></td>
<td>Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Remainder</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scale</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Round</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Square Root</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sine</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cosine</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tangent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arctangent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Log</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Log Binary</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Log Natural</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exponent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Classify</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Copy Real Extended</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Compare</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following sections give a brief overview of the instructions in each of these groups. The floating-point instructions are described in Section 11.

5.3 DATA MOVEMENT

The data movement instructions include those instructions that move data from memory to the global and local registers; that move data from the global and local registers to memory; and that move data among these registers.

5.3.1 Load

The load instructions (listed below) copy bytes or words from memory to a selected register or group of registers:

- `ld` load
- `ldob` load byte ordinal
- `ldos` load short ordinal
ldib load byte integer
ldis load short integer
ldl load long
ldt load triple
ldq load quad

For the ld, ldob, ldos, ldib, and ldis instructions, a memory address and a register are specified in
the instruction and the value at the memory address is copied into the register. Zero and sign extending
is performed automatically for byte and short (half-word) operands.

The ld, ldl, ldt, and ldq instructions copy 4, 8, 12, and 16 bytes from memory into successive
registers.

Note
When using the load, store, and move instructions that move 8, 12, or 16 bytes at a time, the rules for
register alignment must be followed. Refer to the section 2, “Register Alignment” for a discussion of these
rules.

5.3.2 Store

For each load instruction there is a corresponding store instruction (list below), which copies bytes
or words from a selected register or group of registers to memory:

st store
stob store byte ordinal
stos store short ordinal
stib store byte integer
stis store short integer
stl store long
stt store triple
stq store quad

For the st, stob, stos, stib, and stis instructions, a register and memory address are specified in
the instruction and the value in the register is copied into memory. For the byte and short instructions,
the value in the register is automatically reformatted for the shorter memory location. For the stib
and stis instructions, this reformating can lead to overflow if the register value is too large to be
represented in the shorter memory location.

The st, stl, stt, and stq instructions copy 4, 8, 12 and 16 bytes from successive registers into memory.
5.3.3 Move

The move instructions, listed below, copy data from a register or group of registers to another register or group of registers.

- `mov`  move word
- `movl` move long word
- `movt` move triple word
- `movq` move quad word

These move instructions can only be used to move data among the global and local registers. A set of move-real instructions (movr, movrl, and movre) are provided for moving real number values between the global and local registers and the floating-point registers. The move-real instructions are described in Section 11.

5.3.4 Load Address

The `lda` instruction computes an effective address in the address space from an operand presented in one of the addressing modes. A common use of this instruction is to load a constant into a register.

5.4 ARITHMETIC

Table 11 lists all the arithmetic operations for which the 80960KB processor provides instructions and the data types that the instructions operate on. An “X” in this table indicates that the 80960 architecture provides an instruction for the specified operation and data types; an “E” indicates that an 80960KB instruction-set extension provides an instruction for the specified operation and data types. An “E*” indicates that the specified operation can be performed on the specified data type using 80960KB extended instructions, but that a unique instruction for this operation is not provided. For example, a specific instruction is not provided to add two extended-real values. However, this operation can be carried out with either the add real (addr) or the add long real (addrl) instruction.

With two exceptions, all the processor’s arithmetic operations are carried out on operands in registers. The processor does not provide instructions that perform arithmetic operations on operands in memory.

The two instructions that are exceptions are the `atadd` (atomic ad) and `atmod` (atomic modify) instructions, which are discussed later in this section.

A summary of the arithmetic instructions for real (floating-point) data types is provided in Section 11. The following sections describe the arithmetic instructions for ordinal and integer data types.
Table 11. Arithmetic Operations

<table>
<thead>
<tr>
<th>Arithmetic Operations</th>
<th>Integer</th>
<th>Ordinal</th>
<th>Real</th>
<th>Long Real</th>
<th>Extended Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>X</td>
<td>X</td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Subtract</td>
<td>X</td>
<td>X</td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Multiply</td>
<td>X</td>
<td>X</td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Divide</td>
<td>X</td>
<td>X</td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Remainder</td>
<td>X</td>
<td>X</td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Modulo</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Left</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Right</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Right Dividing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scale</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Round</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Square Root</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Sine</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Cosine</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Tangent</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Arctangent</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Exponent</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Log</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Log Binary</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Log Epsilon</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Classify</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>E*</td>
</tr>
<tr>
<td>Copy Sign</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>Copy Reversed Sign</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>
5.4.1 Add, Subtract, Multiply, and Divide

The following instructions perform add, subtract, multiply, or divide operations on integers and ordinals:

- addi: add integer
- addo: add ordinal
- subi: subtract integer
- subo: subtract ordinal
- muli: multiply integer
- mulo: multiply ordinal
- divi: divide integer
- divo: divide ordinal

These instructions perform operations on one-word operands in registers and store the results in a register.

5.4.2 Extended Arithmetic

The following four instructions are provided to support extended arithmetic operations to be performed (i.e. arithmetic operations on operands greater than one word in length):

- addc: add ordinal with carry
- subc: subtract ordinal with carry
- emul: extended multiply
- ediv: extended divide

The addc and subc instructions add or subtract two words (contained in registers) plus a condition code bit (used as a carry bit). If the result has a carry, the carry bit in the condition code is set. Also, a second condition code bit is set if the operation would have resulted in an integer overflow condition. (The three-bit condition code is contained in the arithmetic controls as described in Section 2.)

These instructions treat the operands as ordinals, however, the indication of overflow in the condition code facilitates a software implementation of extended-integer arithmetic.

The emul instruction multiplies two ordinals (each contained in a register), producing long ordinal result (stored in two registers). The ediv instruction divides a long ordinal by an ordinal, producing an ordinal quotient and an ordinal remainder.
5.4.3 Remainder and Modulo

The following instructions divide one operand by another and retain the remainder of the operation:

- `remi` remainder integer
- `remo` remainder ordinal
- `modi` modulo integer

The different between the remainder and modulo instruction lies in the sign of the result. For the `remi` and `remo` instructions, the result has the same sign as the dividend; for the `modi` instruction, the result has the same sign as the divisor.

5.4.4 Shift and Rotate

The processor provides the following five shift instructions:

- `shlo` shift left ordinal
- `shro` shift right ordinal
- `shli` shift left integer
- `shri` shift right integer
- `shrdi` shift right dividing integer

These instructions shift the operand a specified number of bits to the left or to the right. The `shlo`, `shli`, `shro`, and `shrdi` instructions are equivalent to multiplying (shift left) or dividing (shift right) by the power of 2. Bits shifted beyond the register boundary are discarded.

The `shri` instruction performs a conventional arithmetic shift right. However, when this instruction is used to divide an integer operand by the power of 2, it produces an incorrect quotient for negative operands. (The `shrdi` instruction produces the correct quotient when this divide operation is used on negative operands.)

The `rotate` instruction rotates the bits of the operand to the left (toward higher significance) by a specified number of bits. Bits shifted beyond the left boundary of the register (bit 31) appear at the right boundary (bit 0).

5.5 LOGICAL

The following instructions perform bitwise Boolean operations on the specified operands:

- `and` A and B
- `notand` (not A) and B
- `andnot` A and (not B)
- `xor` not (A=B)
or  
A or B
nor  
(not A) and (not B)
xnor  
A = B
not  
not A
notor  
(not A) or B
ornot  
A or (not B)
nand  
(not A) or (not B)

5.6 COMPARISON

The processor provides several types of instructions that are used to compare two operands. The following sections describe the compare instructions for ordinal and integer data types. The compare instructions for real data types are discussed in Section 11.

5.6.1 Compare and Conditional Compare

The compare instructions listed below, compare two operands then set the condition-code bits in the arithmetic controls according to the results.

- cmpi  
  compare integer
- cmpo  
  compare ordinal
- concmpi  
  conditional compare integer
- concmpo  
  conditional compare ordinal

The condition-code bits are set to indicate whether one operand is less than, equal to, or greater than the other operand. (Refer to Section 2, “Functions of the Arithmetic Controls Bits” for a discussion of meanings of the condition-code bits for conditional operations.)

The cmpi and cmpo instructions simply compare the two operands and set the condition-code bits accordingly.

The concmpi and concmpo instructions first check the status of bit 2 of the condition code. If it is not set, the operands are compared as with the cmpi and cmpo instructions. If bit 2 is set, no comparison is performed and the condition-code bits are not changed.

The conditional compare instructions are provided specifically to optimize two-sided range comparisons to check if A is between B and C (i.e., B ≤ A ≤ C). Here, a compare instruction (cmpi or cmpo) is used to check one side of the range (e.g., A ≥ B) and a conditional compare instruction (concmpi or concmpo) is used to check the other side (e.g., A ≤ C) according to the result of the first comparison.
5.6.2 Compare and Increment or Decrement

The following instructions compare two operands, set the condition-code bits according to the results, then increment or decrement one of the operands:

- `cmpinci` compare and increment integer
- `cmpinco` compare and increment ordinal
- `cmpdeci` compare and decrement integer
- `cmpdeco` compare and decrement ordinal

These instructions are intended for use at the end of iterative loops.

5.7 BRANCH

The branch instructions allow the direction of program flow to be changed by explicitly modifying the IP. The processor provides three types of branch instructions:

- unconditional branch
- conditional branch
- compare and branch

Most of the branch instructions specify the target IP by specifying a signed displacement to be added to the current IP. Other branch instructions specify the memory address of the target IP using one of the processor’s addressing modes. This latter group of instructions are called extended-addressing instructions (e.g., branch extended, branch and link extended).

5.7.1 Unconditional Branch

The following four instructions are used for unconditional branching:

- `b` Branch
- `bx` Branch Extended
- `bal` Branch and Link
- `balx` Branch and Link Extended

The `b` and `bx` instructions cause program execution to jump to the specified target IP. As described in Section 10, these two instructions perform the same function; however, they use different machine-level instruction formats.

The `bal` and `balx` instructions store the address of the next instruction in a specified register; then jump to the specified target IP. (For the `bal` instruction, the RIP is automatically stored in register G14; for the `balx` instruction the location of the RIP is specified with an instruction operand.) As described
in Section 3, the branch and link instructions provide a method of performing procedure calls that does not use the processor's call/return mechanism. Here, the saved instruction address is used as a return IP.

The bx and balx instructions can be made IP-relative by using the IP with displacement addressing mode.

5.7.2 Conditional Branch

With the conditional branch (branch if) instructions, the processor checks the condition-code bits in the arithmetic controls. If these bits match the value specified with the instruction, the processor jumps to the target IP. These instructions use the displacement plus IP method of specifying the target IP:

- be  branch if equal
- bn  branch if not equal
- bl  branch if less
- ble branch if less or equal
- bg  branch if greater
- bge branch if greater or equal
- bo  branch if ordered
- bno branch if unordered

(Refer to Section 2, “Functions of the Arithmetic Controls Bits” for a discussion of meanings of the condition-code bits for conditional operations.)

The bo and bno instructions refer to comparisons of real numbers. Ordered and unordered real numbers are described in Section 11.

5.7.3 Compare and Branch

The compare and branch instructions compare two operands, then branch according to the results. There are three subtypes of instructions in this group: compare integer, compare ordinal and check bit:

- cmpibe compare integer and branch if equal
- cmpibne compare integer and branch if not equal
- cmpibl compare integer and branch if less
- cmpible compare integer and branch if less or equal
- cmpibg compare integer and branch if greater
- cmpibge compare integer and branch if greater or equal
- cmpibo compare integer and branch if ordered
cmpibno  compare integer and branch if unordered
cmpobe  compare ordinal and branch if equal
cmpobne  compare ordinal and branch if not equal
cmpobl  compare ordinal and branch if less
cmpoble  compare ordinal and branch if less or equal
cmpobg  compare ordinal and branch if greater
cmpobge  compare ordinal and branch if greater or equal
bbs  check bit and branch if set
bbc  check bit and branch if clear

With the compare-ordinal-and-branch and compare-integer-and-branch instructions, two operands are compared and the condition-code bits are set, as with the compare instructions described earlier in this section. A conditional branch is then executed as with the conditional branch (branch if) instruction.

With the check-bit-and-branch instructions, one operand specifies a bit to be checked in the other operand. The condition-code bits are set according to the state of the specified bit (i.e. 0102 if the bit is set and 0002 if the bit is clear). A conditional branch is then executed according to the setting of the condition-code bits.

5.8 BIT AND BIT FIELD

The bit instructions perform operations on a specific bit in an ordinal operand or on a bit field.

5.8.1 Bit Operations

The following instructions operate on a specified bit:

  setbit  set bit
  clrbits  clear bit
  notbit  not bit
  chkbits  check bit
  alterbit  alter bit
  scanbit  scan for bit
  spanbit  span over bit

The setbit, clrbits, and notbit instructions set, clear, or complement (toggle) a specified bit in an ordinal.

The chkbit instruction causes the condition-code bits to be set according to the state of a specified bit in a register. The condition code is set to 0102 if the bit is set and 0002 otherwise.
The **alterbit** instruction alters the state of a specified bit in an ordinal according to the condition code. If the condition code is $010_2$, the bit is set; if the condition code is $000_2$, the bit is cleared.

The **scanbit** and **spanbit** instructions find the most significant set bit and clear bit, respectively, in an ordinal.

### 5.8.2 Bit Field Operations

There are two bit field instructions **extract** and **modify**. The **extract** instruction converts a specified bit field, taken from an ordinal value, into an ordinal value. In essence, this instruction shifts a bit field in a register to the right and fills in the bits to the left of the bit field with zeros.

The **modify** instruction copies bits from one register, under control of a mask, into another register. Only the unmasked bits in the destination register are modified.

### 5.9 BYTE OPERATIONS

The **scanbyte** instruction performs a byte-by-byte comparison of two ordinals to determine if any two corresponding bytes are equal. The condition code is set according to the results of the comparison.

### 5.10 CONVERSION

Data can be converted from one length to another by means of the load and store instructions. For example, the **ldis** instruction loads a short integer from memory to a register and automatically converts the integer from a half word to a full word.

The 80960KB extended instruction set provides instructions to perform conversions between integer and real data types. These instructions are described in Section 11.

### 5.11 CALL AND RETURN

The processor offers an on-chip call/return mechanism for making procedure calls to local procedures and kernel procedures. This call/return mechanism is described in detail in Section 3. The following four instructions are provided to support this mechanism.

- **call**
- **callx**
- **calls**
- **ret**

The **call** and **callx** instructions call local procedures. The **call** instruction specifies the target procedure (the first instruction of the procedure) by adding a signed displacement to the IP. The **callx**
instruction uses extended addressing, as described for the **bx** and **balx** instructions, to specify the target procedure. For both of these instructions, a new set of local registers and a new stack frame are allocated for the called procedure.

The **calls** instruction operates similarly to the **call** and **callx** instructions, except that it gets its target procedure address from the system procedure table. An index number included as an operand in the instruction provides an entry point into the procedure table.

Depending on the type of entry being pointed to in the procedure table, the **calls** instructions can cause a supervisor call to be executed. A supervisor call causes the processor to switch to the supervisor stack and to switch to supervisor mode. The supervisor call is described in detail in Section 3.

The **ret** instruction performs a return from a called procedure to the calling procedure (the procedure that made the call). This instruction obtains its target **IP** (return **IP**) from linkage information that was saved for the calling procedure. The **ret** instruction is used to return from local and supervisor calls and from implicit calls to interrupt and fault handlers.

### 5.12 Atomic Instructions

The atomic instructions perform read-modify-write operations on operands in memory. They insure that an operation on a specified memory location is completed before another agent with access to memory is allowed to access that memory location. These instructions are particularly useful in systems in which several agents have access to system memory.

There are two atomic instructions: atomic add (**atadd**) and atomic modify (**atmod**). The **atadd** instruction causes an operand to be added to the value in the specified memory location. The **atmod** causes bits in the specified memory location to be modified under control of a mask.

### 5.13 Conditional Faults

Generally, the processor generates faults automatically as the result of certain operations. Fault handling routines are then invoked to handle the various types of faults without explicit intervention by the currently running process. (Faults are discussed in detail in Section 8).

The following conditional fault instructions permit a fault to be generated explicitly according to the state of the condition-code bits:

- **faulte** fault if equal
- **faultne** fault if not equal
- **faultl** fault if less
- **faultle** fault if less or equal
- **faultg** fault if greater
5.14 DEBUG

The processor supports debugging and monitoring of program activity through the use of trace events. The following instructions support these debugging and monitoring tools:

- **modtc**: modify trace controls
- **mark**: mark
- **fmark**: force mark

The trace functions are controlled through the processor's trace controls bits. Some of these bits allow various types of tracing to be enabled or disabled. Other bits act as flags to indicate when an enabled trace event has been detected. (Trace controls are described in detail in Section 9.)

The **modtc** instruction permits the trace controls bits to be modified.

The **mark** instruction causes a breakpoint trace event to be generated if the breakpoint trace mode is enabled. The **fmark** instruction generates a breakpoint trace independent of the state of the breakpoint trace mode flag. The latter two instructions allow a breakpoint to be placed anywhere in a program.

5.15 PROCESSOR MANAGEMENT

The processor provides several instructions for use in controlling processor-related functions.

The **modpc** instruction provides a method of reading and modifying the contents of the process controls.

In certain instances, it is necessary to insure that the contents of the local-register save area of the stack frames are the same as the local registers. The flush local registers instruction (**flushreg**) automatically stores the contents of all the local register sets, except the current set, in the register save area of their associated stack frames.

The arithmetic controls cannot be addressed with the load, move, and store instructions or the bit instructions. Instead, special instructions are provided for this purpose.

The modify arithmetic controls instructions (**modac**) permits bits in the arithmetic controls register to be modified under the control of a mask.
The following test instructions allow the state of the condition-code bits to be tested:

- `teste` test if equal
- `testne` test if not equal
- `testl` test if less
- `testle` test if less or equal
- `testg` test if greater
- `testge` test if greater or equal
- `testo` test if ordered
- `testno` test if unordered

These instructions cause a TRUE (010₂) to be stored in a destination register if the condition code matches the condition specified with the instruction. Otherwise, a FALSE (000₂) is stored in the register.

### 5.16 80960KB NON-FLOATING-POINT INSTRUCTION-SET EXTENSIONS

The following non-floating-point instructions are extensions to the 80960 architecture instruction set. The synchronous load and move instructions are provided in both the 80960KB and 80960KA processor; the decimal instructions are provided only in the 80960KB processor.

#### 5.16.1 Synchronous Load and Move

The processor's store instructions are executed asynchronously with the memory controller. Once the processor sends data out its bus for storage in main memory, it continues with the next instruction in the instruction stream, assuming that its bus control logic will carry out the operation.

The 80960KB processor provides four special instructions for performing memory operations that perform store and move operations synchronously with memory.

The synchronous load instructions (synld) loads a word from a register into memory. When this instruction is performed, the processor waits until a condition code bit is set in the arithmetic controls, indicating that the operation has been completed, before it begins executing the next instruction.

The synchronous move instructions (synmov, synmovl, and synmovq) perform synchronous moves of data from one location in memory to another.

These instructions are used primarily for sending IAC messages, as described in Section 12.

#### 5.16.2 Decimal

The following three instructions are provided for use in decimal-arithmetic algorithms:
The instructions operate on 32-bit decimal operands that contain an 8-bit, ASCII-coded decimal in the least-significant byte of the word (as shown in Figure 11).

The `dmovt` instruction moves a decimal operand from one register to another and tests the least significant byte of the operand to determine if it is a decimal digit (0 to 9). It sets the condition code according to the results of the test: 010₂ if the operand contains a decimal digit and 000₂ otherwise.

The `dadde` and `dsuce` instructions operate similarly to the `adde` and `subc` instructions. They add or subtract two decimal digits plus bit 1 of the condition code (used as a carry-in bit). If the operation produces a decimal carry, the condition code is set accordingly. The subtraction operation is carried out in 10's complement arithmetic.

These instructions can be used iteratively to add or subtract decimal values of any length.

With the 80960KB processor, the most efficient method of multiplying or dividing decimal numbers is to convert them into extended-real numbers and use the `mulr` and `divr` instructions. Decimal values of up to 18 decimal digits can be handled with this technique.

### 6.0 PROCESSOR MANAGEMENT AND INITIALIZATION

This section describes the facilities for initializing and managing the operation of the 80960KB processor. Included is a description of the processor-management facilities and the steps required to initialize the processor. Appendix D gives a listing of the necessary 80960KB code to initialize the processor.

### 6.1 OVERVIEW OF PROCESSOR MANAGEMENT FACILITIES

This section and sections 7, 8, 9, and 12 describe the 80960KB’s processor-management facilities. These facilities are primarily software-related, although some hardware considerations are also discussed.

For the purpose of discussion in these sections, it assumed that the processor is going to execute a program made up of a system kernal (or executive) and applications code. This program may be located in ROM or RAM.

Such a program has the following facilities available to it to initialize, communicate with, and control the processor:

- Instruction List
- System Data Structures
Interrupts
IACs
Faults

These facilities allow system hardware and the kernel to initialize the processor and initiate instruction execution. They also provide software or external agents with methods of interrupting the processor to service external I/O devices.

The following paragraphs give an overview of these processor-management facilities.

### 6.1.1 Instruction List

At the most rudimentary level, the processor is controlled through a stream of instructions that the processor fetches from memory and executes one at a time. Once the processor is initialized, it begins executing instructions and continues until it is stopped.

### 6.1.2 System Data Structures

The processor defines several system data structures that reside in memory. These data structures (shown in Figure 13) offer a means of configuring the processor to operate in a specific way.

![System Defined Data Structures](image-url)

Figure 13. System Defined Data Structures
The system data structures can be located anywhere in the processor’s address space. The processor gets pointers to most of these data structures from the initial memory image (IMI). The IMI is described later in this section in “Initial Memory Image”.

The interrupt table provides pointers to interrupt-handling procedures. The interrupt vector numbers act as indices into this table. For the purpose of handling interrupts, a separate interrupt stack is maintained in the address space. The interrupt mechanism is described in Section 7.

The fault table provides pointers to fault-handling procedures. When the processor detects a fault, it generates a fault vector number internally that provides an index into the fault table. The fault mechanism is described in Section 8.

The system procedure table contains pointers to the kernel procedures, which are accessed using the system call (calls) mechanism. The system table structure is described in Section 3, “System Procedure Table”.

The processor uses two stacks for procedures calls: the local procedure stack and the (optional) supervisor stack. These stacks are described in Section 3.

The processor also contains a register, called the process controls register, that it uses to store information about the current state of the processor and the program it is executing. The process controls are described later in this section under “Process Controls”.

### 6.1.3 Interrupts

The processor defines two methods of asynchronously requesting services from the processor: interrupts and IAC messages. Interrupts are the more common of the two.

An interrupt is a break in the control flow of a program so that the processor can handle a more urgent chore. Interrupt requests are generally sent to the processor from an external source, often to request I/O services. When the processor receives an interrupt request, it temporarily stops work on its current task and begins work on an interrupt-handling procedure. Upon completion of the interrupt-handling procedure, the processor generally returns to the task that was interrupted and continues work where it left off.

Interrupts also have a priority, which the processor uses to determine whether to service the interrupt immediately or to postpone service until a later time.

### 6.1.4 IACs

The 80960KB processor provides an alternate method of communicating with other agents on the system bus are able to communicate with the processor through messages that are exchanged in a reserved section of memory.
Like interrupts, IACs are used to request that the processor stop work on its current task and begin work on another task. However, where an interrupt generally causes a temporary break in the execution of a program, an IAC often causes a permanent change in the control flow of the processor.

The IAC mechanism is described in Section 12.

6.1.5 Faults

While executing instructions, the processor is able to recognize certain conditions that could cause it to return an inappropriate result or that could cause it to go down a wrong and possibly disastrous path. One example of such a condition is a divisor operand of zero in a divide operation. Another example is an instruction with an invalid opcode. These conditions are called faults.

The processor handles faults almost the same way that it handles interrupts. When the processor detects a fault, it automatically stops its current processing activity and begins work on a fault-handling procedure.

6.2 PROCESS CONTROLS

The process-controls word (shown in Figure 14) contains miscellaneous pieces of information to control processor activity and show the current state of the processor. The various functions of this field are described in the following paragraphs.

![Figure 14. Process-Controls Word](image)

The execution mode flag determines whether the processor is operating in the user mode (clear) or supervisor mode (set). The processor automatically sets this bit on a supervisor call and clears it on a return from supervisor mode.

The priority field determines the priority (from 0 to 31) of the processor. When the processor is in the executing state, it sets its priority according to this value.
The state flag determines the state of the processor. The encoding of this bit is shown in Table 12.

**Table 12. Encoding of Processor State Field**

<table>
<thead>
<tr>
<th>State Field</th>
<th>Processor State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Executing</td>
</tr>
<tr>
<td>1</td>
<td>Interrupted</td>
</tr>
</tbody>
</table>

This bit tells software whether the processor
- is currently executing a program (0) or
- has been interrupted so it can service an interrupt (1).

The trace-enable and trace-fault-pending flags control tracing. The trace-enable field determines whether trace faults are to be generated (set) or not-generated (clear). The trace-fault-pending field is a flag that the processor uses to determine if a trace event has been detected (set) or not (clear). The use of these fields is discussed in detail in Section 9.

The resume flag signals the processor that an instruction has been suspended. The processor sets this flag whenever it suspends an instruction to handle an interrupt or fault. On a return from the interrupt or fault handler, the processor checks this flag and performs an instruction resumption action if the flag is set.

All of the bits in the process controls are set to zero as part of the initialized procedure. Bits 2 through 8, 11, 13, 15, and 21 through 31 are reserved. These bits should not be altered following initialization.

### 6.2.1 Changing the Process Controls

The kernel can change the process controls using any of the following three methods:

- Modify-process-controls instruction (modpc)
- Alter the saved process controls prior to a return from an interrupt handler
- Alter the saved process controls prior to a return from a fault handler

The modpc instruction reads and modifies the process controls cached in the processor.

In the latter two methods, the kernel changes the process controls in the interrupt or fault record that is saved on the stack. On the return from the interrupt or fault handler, the modified process controls are copied into the processor’s internal process controls.

**Note**

Changing the saved process controls by means of a fault handler can only be used if the fault handler was invoked by means of an implicit supervisor call.
When the process controls are changed as described above, the processor acts on the changes as soon as it receives the new information, except for the following situation.

If the modpc instruction is used to change the trace-enable flag, the processor does not guarantee to act on the change until after up to four more instructions have been executed.

6.3 PRIORITIES

The processor defines a priority mechanism for determining the order in which programs, interrupts, and IACs are worked on. Priorities range from 0 to 31, with 31 being the highest priority. Each interrupt vector is assigned a priority. Also, when the processor is executing a program, it sets its priority according to the priority field of the process controls.

Interrupt priorities serve two functions. First, they determine if the processor will service an interrupt immediately or delay servicing it with respect to its current priority. Second, they determine which interrupt of several interrupts is serviced first.

When the processor receives an IAC, it always services it immediately (i.e., treats the IAC as if it has a priority of 31). A mechanism is provided that allows priorities to be assigned to IACs. When using this mechanism, external hardware is required to intercept all IACs sent to the processor and to check their priority. This hardware then determines whether to send the IAC to the processor for servicing or delay it according to the current priority of the processor.

6.4 PROCESSOR STATES

The processor has four different operating states: executing, interrupt, stopped, and stopped-interrupted. The processor is placed in one of two states (executing or stopped) at initialization. After that, the processor and software control the processor’s state.

The processor can switch between the executing and interrupted states or between the stopped and stopped-interrupted states. However, the processor never switches from the executing state to the stopped state, unless it detects a series of fault conditions that it cannot handle.

Software can change the state of the processor in either of two ways: (1) issue a reinitialize IAC or (2) issue a freeze IAC. The reinitialize IAC forces the processor to reread the pointers from the IMI and begin executing instructions from a new IP. The freeze IAC forces the processor into the stopped state.

6.4.1 Executing and Interrupted State

In the executing state, the processor is executing the program.

If the processor is interrupted while in the executing state, it saves the current state of the program, switches to the interrupt state, and services the interrupt. Upon returning from the interrupt handler, the processor resumes work on the program.
6.4.2 Stopped and Stopped-Interrupted States

In the stopped state the processor ceases all activity. The only tasks it can perform while in this state are to service an interrupt or an IAC. While servicing an interrupt, the processor switches to the stopped-interrupt state. It then switches back to the stopped state upon completion of the interrupt routine. Likewise, while servicing an IAC, the processor switches to the stopped-interrupted state. If the IAC handling action does not result in a change in the processor's state, the processor switches back to the stopped state when it finishes the IAC handling action.

The only way to get the processor out of the stopped state (other than to service an interrupt) is to reinitialize the processor, either with a hardware reset or by sending it an external reinitialize IAC.

6.5 INSTRUCTION SUSPENSION

When the processor is interrupted while it is in the midst of executing an instruction, it does one of three things before it services the interrupt:

1. It completes the instruction.
2. It terminates the instruction and sets the processor state so that it is as if execution of that instruction had not yet begun.
3. It suspends the instruction and saves the necessary resumption information so that execution of the instruction can be continued when the processor begins work on the program again. This course of action is generally reserved for instructions that have a long execution time and that alter the internal and external processor state as they execute.

Which of these steps the processor takes depends on the instruction being executed. However, whichever step it takes is transparent to the software. The processor automatically saves the necessary state information so that work on the program can be resumed with no loss of information.

Refer to the section 7, “Interrupt Handling Action”, for more information on how resumption information is saved when an interrupt is services.

6.6 MEMORY REQUIREMENTS

The processor provides a $2^{32}$-byte address space. This address space can be mapped to read-write memory, read-only memory, and memory-mapped I/O. (The processor does not provide a dedicated, addressable I/O space.)

The address space is linear (or flat): there are no subdivisions of the address space such as segments. For the purpose of memory management, an external memory management unit (MMU) may subdivide memory into pages or restrict access to certain areas of memory to protect kernel code and data. But from the point of view of the processor, the address space is linear.
All of the address space is available for general use except the upper 16M bytes (FF000000<sub>16</sub> to FFFFFFFF<sub>16</sub>), which are reserved for special functions. (These functions are described in Section 12).

An address in memory is a 32-bit value in the range 0 to FFFFFFFF<sub>16</sub>. It can be used to reference a single byte, 2 bytes, 4 bytes, 8 bytes, 12 bytes or 16 bytes of memory depending on the instruction being used. (Refer to the descriptions of the load and store instructions in Section 10 for information on multiple-byte addressing.)

6.6.1 Memory Restrictions

The processor requires that the memory to which the address space is mapped has the following capabilities.

- It must be byte addressable.
- It must support burst transfers (i.e., transfers of blocks of contiguous bytes up to 16 bytes in length).
- It must guarantee indivisible access (read or write) for memory addresses that fall within 16-byte boundaries.
- It must guarantee atomic access for memory addresses that fall within 16-byte boundaries.

The latter two capabilities are required to allow multiple processor to share a common memory conveniently.

An indivisible access guarantees that a processor reading or writing a set of memory locations will complete the operation before another processor can read or write the same location. The processor requires indivisible access within an aligned, 16-byte block of memory.

An atomic access is read-modify-write operation. Here external logic must guarantee that once a processor begins a read-modify-write operation on a set of memory locations, it is allowed to complete the operation before another processor is allowed to access the same location.

As described above, the processor requires that when one processor is performing an atomic operation within an aligned, 16-byte block, other processors are delayed from performing another atomic operation within that block until the first operation has been completed.

The 80960KB processor provides two features to aid in implementing the memory requirements described above: SIZE lines and a LOCK line on the local bus.

The SIZE lines indicate the length of a memory access in bytes. These lines can be used to specify 1-, 2-, 4-, 8-, 12-, or 16-byte lengths. When making the multiple-byte access, the processor thus sends the memory controller a base address, on the address lines, and a length on the SIZE lines.
The LOCK line is used to synchronize atomic operations. When a processor performs an atomic operation, it first examines the LOCK line. If it is asserted, the processor waits until the line is not asserted (i.e., spins on the LOCK line). If the line is not asserted, the processor asserts the LOCK line when it is performing an atomic read and deasserts the line when it performs the companion atomic write.

The LOCK line mechanism allows only one atomic operation to be carried out in memory at one time.

6.7 SOFTWARE REQUIREMENTS FOR PROCESSOR MANAGEMENT

The processor-management facilities described earlier in this section allow the processor to be configured and operated in several ways. This section lists the data structures that the kernel must supply to operate the processor.

To use the processor, the kernel must provide the following items:

- IMI
- Other System Data Structures
- Address Space
- Stacks
- Code

The IMI comprises the minimum data structures that the processor needs to initialize the system.

As part of the initialization procedure, a more complete set of system data structures are established in memory. These data structures include an interrupt table and a fault table. If the system call mechanism is going to be used, a system procedure table is required.

Two stacks are also required: an interrupt stack and a local (or user) procedure stack. The initial stack pointer for the interrupt stack is given in the IMI. The initial stack pointer (SP) for the local-procedure stack is given in local register r1; the initialization code is required to establish the SP value in this register.

If the supervisor call mechanism is to be used, a supervisor stack must also be provided. The initial stack pointer for this stack is given in the system-procedure table. The supervisor stack can be placed anywhere in the address space.

Note

"Hints on Using the User-Supervisor Protection Model" in section 3 describes an application of the user-supervisor protection model, in which the processor is always in supervisor mode. When using this application, the local stack and the supervisor stack are the same. The processor gets the initial stack pointer for this stack from register r1.
Finally, three levels of code are required: initialization code, kernel code, and applications code. The initialization code is part of the IMI. (Appendix D gives an initialization code example.) The starting IP for the initialization code is also provided in the IMI.

6.8 PROCESSOR INITIALIZATION

This section describes how to initialize the 80960KB processor. It defines the mechanism that the processor uses to establish its initial state and begin instruction execution. It also describes some general guidelines for writing code to complete the initialization of the processor for specific applications.

Note
The 80960 architecture does not define an initial memory image or an initialization procedure. The following initialization requirements are specific to the 80960KB processor.

6.8.1 Initial Memory Image

The IMI performs three functions for the processor: (1) it provides check-sum words that the processor uses in its self-test routine at start-up, (2) it provides pointers to the system data structures, and (3) it provides scratch space that the processor uses to perform certain internal functions. Figure 15 shows the structure of the IMI.

The IMI is made up of four parts: the check-sum word, the system address table (SAT), and the processor control block (PRCB), and the initialization code. In an embedded application, all of the parts of this image will generally be held in ROM, except the scratch space of the PRCB. For this reason, the PRCB should be copied from ROM to RAM after system initialization. (The reinitialize IAC, described in Section 12, is used to give the processor the PRCB pointer for the relocated PRCB.)

6.8.2 Check-Sum Words

The check-sum words must be in memory locations 00000000 to 0000001F. The first of these words is a pointer to the base of the SAT. The second word is a pointer to the base of the PRCB. The fourth word is the instruction pointer to the first instruction of the initialization code.

The remaining words (word 3 and words 5 through 8) are check words, which must be chosen such that the one’s complement of the sum of the eight words plus FFFFFFFF equals 0.

6.8.3 System Address Table

The SAT is 158 bytes in size and can be located anywhere in the address space. It has four required entries. The word beginning at byte 136 must contain a pointer to the base (first byte) of the SAT. This pointer is identical to the pointer given in the first word of the check-sum words. The word beginning at byte 152 must contain a pointer to the base of the system procedure table. The words beginning at byte 140 and 156 must contain 00FC00FB and 304400FB respectively.

All of the other words in the SAT are preserved and can be used by software.
Figure 15. Initial Memory Image
6.8.4 Processor Control Block

The PRCB is 174 bytes long and can also be located anywhere in the address space. It has seven required entries and one reserved space.

Bits 0 through 30 of the word beginning at byte 4 must be zero.

The write-external-priority flag (bit 31 of the word beginning at byte 4) instructs the processor to write the priority of the processor to the IAC message control field whenever an interrupt (not caused by an IAC) or the execution of the modpc instruction occurs. When this bit is set, the write-external-priority mechanism is enabled; when the bit is clear, the mechanism is disabled. The use of this flag is described in Section 12.

The interrupt table pointer points to the first byte of the interrupt table. The interrupt stack pointer points to the top (first available byte) of the interrupt stack.

The words beginning at bytes 32 and 36 must each contain 0000027F16.

The fault table pointer points to the first byte of the fault table.

The word beginning at byte 44 must contain all zeros.

The processor uses the scratch space in the IMI for internal functions. This field should be set to all zeros at initialization or reinitialization of the processor and not accessed by software thereafter.

The remaining fields in the PRCB (bytes 8 through 19, bytes 28 through 31, and bytes 48 through 79) are reserved. They should be set to all zeros at initialization or restart and not accessed by software thereafter.

6.8.5 Initialization Code

The initial instruction list that the processor begins executing following its self test can be located anywhere in the address space.

6.8.6 Changing the Initial Memory Image

At initialization or on a reinitialize processor IAC, the processor reads the pointers from the IMI in memory and caches them.

In general, to change any of the IMI fields that have been cached on the processor chip, the kernel must first modify the IMI in memory, then reinitialize the processor using the reinitialize processor IAC. The processor then rereads the IMI and reloads the cached fields in its internal cache.
6.8.7 Building a Memory Image

The IMI shown in Figure 15 contains the minimum data structures required for the processor to initialize itself and begin executing code. To build a useful system, however, additional data structures are required, such as an interrupt table, a fault table, a system procedure table, a set of kernel procedures, a set of stacks, and a heap. Some of these data structures can be located in ROM along with the IMI; however, others must be in RAM because they must be writable.

Table 13 lists the various system data structures and shows which can be in ROM and which must be in RAM. The following paragraphs give the system limitations if a data structure is included in ROM.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>May Be in ROM</th>
<th>May Be in ROM with Limitations</th>
<th>Must Be in RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMI</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRCB</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SAT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Interrupt table</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Fault table</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Kernel Procedures</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Stacks and heap</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

All of the PRCB except the scratch space area must be in ROM. The scratch space must be in RAM.

The interrupt table must be in RAM for the processor to operate properly, because it contains the interrupt pending fields, which the processor must be able to write to.

The fault table can be in ROM, providing it will never be necessary to relocate the fault handler routines.

The kernel procedures can be in either ROM or RAM or both, depending on the design of the kernel.

6.8.8 Typical Initialization Scenario

Initialization of the 80960KB processor typically is handled in two stages. In the first stage of initialization the processor performs a self test and reads pointers from the IMI. During the second stage, the processor executes initialization code designed to build the remainder of the memory image so that execution of applications code can begin.
6.8.9 First Stage of Initialization

The following procedure shows the steps that system hardware and the processor go through in the first stage of initialization. The algorithm in Figure 16 gives the details of this procedure.

```
assert FAILURE pin;
perform self test;
if self test fails
    then enter stopped state;
else
    deassert FAILURE pin;
    enter predefined state;
    if STARTUP pin = 0
        then enter stopped state;
    else
        x ← memory(0); read 8 words beginning at address 0
        AC.cc ← 0000;
        temp ← FFFFFFF16 add_with_carry x(0);
        temp ← temp add_with_carry x(1);
        temp ← temp add_with_carry x(2);
        temp ← temp add_with_carry x(3);
        temp ← temp add_with_carry x(4);
        temp ← temp add_with_carry x(5);
        temp ← temp add_with_carry x(6);
        temp ← temp add_with_carry x(7);
        if temp ≠ 0
            then
                assert FAILURE pin;
                enter stopped state;
            else
                prcb_address ← memory(4);
                IP ← memory (12)
                fetch IMI;
                processor.priority ← 31;
                processor.state ← interrupted;
                FP ← IMI.interrupt_stack_pointer;
                clear any latched external interrupt/IAC signals;
                begin execution;
            endif;
        endif;
    endif;
endif;
```

Figure 16. Algorithm for First Stage of Initialization Procedure
1. Hardware asserts the RESET pin on the processor.

2. The processor samples LPN to get its locals processor (1 or 0). (LPN and STARTUP are signals that come from multiplexed information received on several processor pins.)

3. The processor asserts the FAILURE pin and performs a self test. If the processor passes the self test, it deasserts the FAILURE pin.

4. The processor samples STARTUP to determine whether it is the initializing processor (1) or not (0). If the processor is the initializing processor, it continues with the initialization procedure; if it is not, it goes into the stopped state. (In multiprocessing systems, all processors except the initializing processor are put in the stopped state.)

5. The processor reads the 8 check-sum words and checks that the check sum is 0.

6. Using the contents of the check-sum words, the processor determines the location of the SAT, the PRCB, and the first instruction to be executed.

7. The processor sets its process priority to 31 (highest possible) and its state to interrupted.

8. The processor clears any latched external interrupt or IAC signals. This means that the processor will not service any interrupts or IACs prior to beginning instruction execution.

9. The processor begins execution of the initialization instruction list.

After self test, the processor establishes its own state. For the initializing processor this state is interrupted; for any other processors in the system this state is stopped. Also at initialization, the trace controls are set to zero; the process controls are set to zero (except for the execution mode, which is set to supervisor, and the priority, which is set to 31); and the breakpoint registers are disabled.

Since the processor places itself in the interrupted state during the first stage of initialization, the initialization code is essentially a special interrupt-handler procedure.

**6.8.10 Second Stage of Initialization**

The processor activity during the second stage of initialization, which occurs once the processor begins instruction execution, is up to software. In general, this stage of initialization is used to copy to create additional data structure in memory, such as the interrupt table, the system-procedure table, and the fault table (if not in the initial memory image), and the kernel procedures.

Once these jobs are completed, the processor can then begin executing applications code.

Appendix D gives an example of the 80960KB code that might be used to carry out this second stage of initialization.

A common initialization technique is to create a new PRCB and interrupt table in RAM along with the other system data structures that are placed in memory in the second stage of initialization. The processor is then reinitialized to point to the PRCB and interrupt table. (The code in Appendix D uses this technique.)
The processor is reinitialized using the reinitialize IAC. This reinitialize IAC message includes new pointers to the SAT and PRCB. The processor reads the new PRCB, then begins instruction execution according to the control information contained in the PRCB.

7.0 INTERRUPTS

This section describes the 80960KB processor’s interrupt handling facilities. It also describes how interrupts are signaled.

7.1 OVERVIEW OF THE INTERRUPT FACILITIES

An interrupt is a temporary break in the control stream of a program so that the processor can handle another chore. Interrupts are generally requested from an external source. The interrupt request either contains a vector number or else points to a vector that tells the processor what chore to do while in the interrupted state. When the processor has finished servicing the interrupt, it generally returns to the program that it was working on when the interrupt occurred and resumes execution where it left off.

The processor provides a mechanism for servicing interrupts, which uses an implicit procedure call to a selected interrupt handling procedure, called an interrupt handler.

When an interrupt occurs, the current state of the program is saved. If the interrupt occurs during an instruction that requires many machine cycles, the instruction state is also saved and execution of the instruction is suspended.

The processor then creates a new frame on the interrupt stack and executes an implicit call to the interrupt handler selected with the interrupt vector.

Upon returning from the interrupt handler, the processor switches back to the program that was running when the interrupt occurred, restores it to the state it was in when the interrupt occurred, and resumes work on it.

Another feature of this interrupt handling mechanism is that it allows interrupts to be prioritized. If an interrupt is signaled that has the same or a lower priority than the processor’s current priority, the processor will save the interrupt vector and service the interrupt at a later time. Interrupts that are waiting to be serviced are called pending interrupts.

7.2 SOFTWARE REQUIREMENTS FOR INTERRUPT HANDLING

To use the processor’s interrupt handling facilities, software must provide the following items in memory:

- Interrupt Table
- Interrupt Handler Routines
- Interrupt Stack
These items are generally established in memory as part of the initialization procedure. Once these items are present in memory and pointers to them have been entered in the appropriate system data structures, the processor then handles interrupt automatically and independent from software.

The requirements for these items are given in the following sections.

### 7.3 VECTORS AND PRIORITY

Each interrupt vector is 8 bits in length, which allows up to 256 unique vectors to be defined. In practice, vectors 0 through 7 cannot be used, and vectors 244 through 251 are reserved and should not be used by software.

Each vector has a predefined priority, which is defined by the following expression:

\[
\text{priority} = \frac{\text{vector}}{8}
\]

Thus, at each priority level, there are 8 possible vectors (vectors 8 through 15 have a priority of 1, vectors 16 through 23 a priority of 2, and so on to vectors 246 through 255, which have a priority of 31).

The processor uses the priority of an interrupt to determine whether or not to service the interrupt immediately or to delay service. If the interrupt priority is greater than the processor’s current priority, the processor services the interrupt immediately; if the interrupt priority is equal to or less than the processor’s current priority, the processor saves the interrupt vector as a pending interrupt so that it can be serviced at a later time.

A priority-31 interrupt is always serviced immediately.

Note that the lowest program priority allowed is 0. If the current program has a 0 priority, a priority-0 interrupt will never be accepted. This is why vectors 0 through 7 cannot be used. In fact, there are no entries provided for these vectors in the interrupt table.

### 7.4 INTERRUPT TABLE

The interrupt table contains instruction pointers (addresses in the address space) to interrupt handlers. It must be aligned on a word boundary. The processor determines the location of the interrupt table by means of a pointer in the IMI.

As shown in Figure 17, the interrupt table contains one entry (i.e., one pointer) for each allowable vector. The structure of an interrupt-table entry is given at the bottom of Figure 17. Each interrupt procedure must begin on a word boundary, so the two least-significant bits of the entry are set to 0.

The first 36 bytes of the interrupt table are used to record pending interrupts. This section of the table is divided into two fields: pending priorities (byte-offset 0 through 3) and pending interrupts (byte-offset 4 through 35).
The pending priorities field contains a 32-bit string in which each bit represents an interrupt priority. The bit number in the string represents the priority number. When the processor posts a pending interrupt in the interrupt table, the bit corresponding to the interrupt's priority is set. For example, if an interrupt with a priority of 10 is posted in the interrupt table, bit 10 is set.

The pending interrupt field contains a 256-bit string in which each bit represents an interrupt vector. For example, byte-offset 4 is reserved, byte-offset 5 is for vectors 8 through 15, byte-offset 6 is for vectors 16 through 23, and so on. When a pending interrupt is logged, its corresponding bit in the pending interrupt field is set.
This encoding of the pending priority and pending interrupt fields permits the processor to first check if there are any pending interrupts with a priority greater than the current program and then to determine the vector number of the interrupt with the highest priority. Software should set these fields to 0 at initialization and not access these fields after that.

Note
Refer to the section, “Handling Pending Interrupts”, later in this section for a description of the processor’s pending interrupt mechanism.

7.5 INTERRUPT HANDLER PROCEDURES

An interrupt handler is a procedure that performs a specific action that has been associated with a particular interrupt vector. For example, a typical job for an interrupt handler is to read a character from a keyboard.

The interrupt handler procedure can be located anywhere in the address space. Each procedure must begin on a word boundary.

The processor execution mode is always switched to supervisor while an interrupt is being handled.

When an interrupt-handler procedure is called, the states of the processor controls and arithmetic controls for the interrupted program are saved. However, the interrupt handler shares the other resources of the interrupted program, in particular the global registers and the address space. This sharing of resources imposes one important restriction on the interrupt handler procedures.

The interrupt handler procedures must preserve and restore the state of any of the resources that it uses. For example, the processor allocates a set of local registers to the interrupt handler, just as it does on an explicit procedure call. If the interrupt handler needs to use the global or floating-point registers, however, it should save their contents before using them and restore them before returning from the interrupt handler.

7.6 INTERRUPT STACK

The interrupt stack can be located anywhere in the address space. The processor determines the location of the interrupt stack by means of a pointer in the IMI.

The interrupt stack has the same structure as the local procedure stack described in Section 3, “Procedure Stack”.

7.7 INTERRUPT HANDLING ACTIONS

When the processor receives an interrupt, it handles it automatically. The processor takes care of saving the processor state, calling the interrupt-handler routine, and restoring the processor state once the interrupt has been serviced. Software support is not required.
The following section describes the actions the processor takes while handling interrupts. It is not necessary to read this section to use the interrupt mechanism or write an interrupt handler routine. This discussion is provided for those readers who wish to know the details of the interrupt handling mechanism.

### 7.7.1 Receiving an Interrupt

Whenever the processor receives an interrupt signal, it performs the following action:

1. It temporarily stops work on its current task, whether it is working on a program or another interrupt procedure.
2. It reads the interrupt vector.
3. It compares the priority of the vector with the processor's current priority.
4. If the interrupt priority is higher than that of the processor, the processor services the interrupt immediately as described in the next sections.
5. If the interrupt priority is equal to or less than that of the processor, the processor sets the appropriate priority bit and vector bit in pending interrupt record and continues work on its current task.

### 7.7.2 Servicing an Interrupt

The method that the processor uses to service an interrupt depends on the state the processor is in when it receives the interrupt. The following sections describe the interrupt handling actions for various states of the processor. In all of these cases, it is assumed that the interrupt priority is higher than that of the processor and will thus be serviced immediately after the processor receives it. The handling of lower priority interrupts is described later in “Pending Interrupts”.

### 7.7.3 Executing State Interrupt

When the processor receives an interrupt while it is in the executing state (i.e. executing a program), it performs the following actions to service the interrupt; this procedure is the same regardless of whether the processor is in the user or the supervisor mode when the interrupt occurs:

1. The processor saves the current state of process controls and arithmetic controls in an interrupt record on the stack that the processor is currently using. This stack can be the local-procedure stack or the supervisor stack. (The interrupt record is described in the following section.)
2. If the execution of an instruction was suspended, the processor includes a resumption record for the instruction in the current stack and sets the resume flag in the saved process controls. (Refer to section 7, “Instruction Suspension”, for a discussion of the criteria for suspending instructions.
3. The processor switches to the interrupted state.
4. The processor sets the state flag in the process controls to interrupted, its execution mode to supervisor, and its priority to the priority of the interrupt. Setting the processor priority to that
of the interrupt insures that lower priority interrupts can not interrupt the servicing of the current interrupt.

5. Also in its internal process controls, the processor clears the trace-fault-pending and trace-enable flags. Clearing these flags allows the interrupt to be handled without trace faults being raised.

6. The processor allocates a new frame on the interrupt stack and switches to the interrupt stack.

7. The processor sets the frame return status field (associated with the PFP) to 1112.

8. The processor performs an implicit call-extended operation (similar to that performed for the callx instruction). The address for the procedure that is called is that which is specified in the interrupt table for the specified interrupt vector.

Once the processor has completed the interrupt procedure, it performs the following action on the return:

1. The processor deallocates the stack frame from the interrupt stack and switches to the local or supervisor stack (whichever one it was using when it was interrupted).

2. The processor copies the arithmetic controls field from the interrupt record into its arithmetic controls register.

3. The processor copies the process controls field from the interrupt record into its internal process controls.

4. If the resume flag of the process controls is set, the processor copies the resumption record from the interrupt record to the resumption record field of the PRCB.

5. The processor checks the interrupt table for pending interrupts that are higher than the priority of the program being returned to. If a higher-priority pending interrupt is found, it is handled as if the interrupt occurred at this point.

6. Assuming that there are not pending interrupts to be serviced, the processor switches to the executing state and resumes work on the program.

**7.7.4 Interrupt State Interrupt**

If the processor receives an interrupt while it is servicing an interrupt, and the new interrupt has a higher priority than the interrupt currently being serviced, the current interrupt-handler routine is interrupted. Here the processor performs the same action to save the state of the interrupted interrupt-handler routine as is described at the beginning of this section. Here, the interrupt record is saved on the top of the interrupt stack, prior to the new frame that is created for use in servicing the new interrupt.

**7.7.5 Interrupt Record**

The processor saves the state of an interrupted program (or interrupt-handler) routine in an interrupt record. Figure 18 shows the structure of this interrupt record.
LOCAL, SUPERVISOR, OR INTERRUPT STACK

STACK GROWTH

31

REGISTER SAVE AREA FOR CURRENT FRAME

ADDITIONAL VARIABLES AND PADDING AREA (OPTIONAL)

SP

INTERRUPT STACK

STACK GROWTH

31

PADDING AREA

RESUMPTION RECORD FOR SUSPENDED INSTRUCTION (OPTIONAL)

SAVED PROCESS CONTROLS

SAVED ARITHMETIC CONTROLS

VECTOR NUMBER

NEW FRAME

NSP*

interrupt record

NFP-16

NFP-12

NFP-8

NFP

*IF THE INTERRUPT IS SERVICED WHILE THE PROCESSOR IS WORKING ON ANOTHER INTERRUPT PROCEDURE, THE NEW STACK POINTER (NSP) WILL BE THE SAME AS THE SP.

RESERVED

Figure 18. Storing of an Interrupt Record on the Stack
The resumption record within the interrupt record is used to save the state of a suspended instruction. If no instruction is suspended, the resumption record is not created.

### 7.7.6 Stopped State Interrupt

The processor can also be interrupted while in the stopped state. The processor handles such interrupts in essentially the same way that it handles interrupts that occur while the processor is in the executing state, with the following exception. When the processor allocates the new frame on the interrupt stack, it sets the frame return field to $1102$. This causes the processor to revert to the stopped state when the processor returns from the interrupt-handler procedure.

### 7.7.7 Stopped-Interrupted State Interrupt

If the processor receives an interrupt while it is in the stopped-interrupted state, it handles the interrupt just as it would if it occurred in the interrupted state.

### 7.7.8 Pending Interrupts

As was described earlier in this section, the processor provides a mechanism for evaluating interrupts according to their priority. If the interrupt priority is equal to or lower than the processor’s current priority, the processor does not service the interrupt immediately. Instead, it posts the interrupt in the pending interrupt section of the interrupt table. The processor checks the interrupt table at specific times and services those interrupts that have a higher priority than its current priority. This pending interrupt mechanism provides two benefits.

1. The ability to delay the servicing of low priority interrupts (by posting them in the pending interrupt section of the interrupt table) allows the processor to concentrate its processing activity on higher priority tasks.
2. In a system that uses two or more 80960KB processors, both processors can share the same interrupt table. This interrupt-table sharing allows the processors to share the interrupt handling load.

The following paragraphs describe how the processor handles pending interrupts.

---

**Note**

The 80960 architecture defines the section of the interrupt table for storing pending interrupts and a mechanism for checking the interrupt table for pending interrupts. The method used for posting interrupts to the interrupt table and circumstances under which the processor check the interrupt table for pending interrupts is not defined.

In the following description of the pending interrupt mechanism, the information given in the sections titled “Posting Pending Interrupts” and “Checking for Pending Interrupts” is specific to the 80960KB processor. The information given in the section titled “Handling Pending Interrupts” is defined in the 80960 architecture and should be common in all processors that implement this part of the architecture.

---

3-76
7.7.9 Posting Pending Interrupts

An interrupt can be posted in the pending-interrupt record of the interrupt table in either of the following two ways:

1. The processor receives an interrupt with a priority equal to or lower than that of the program the processor is currently working on. The processor then automatically posts the interrupt in the pending-interrupt record.

2. The kernel can set the desired pending-interrupt and pending-priority bits in the interrupt table.

Using the first method, the processor performs an atomic read/write operation that locks the interrupt table until the posting operation has been completed. Locking the interrupt table prevents other agents on the bus from accessing the interrupt table during this time.

The second method of posting an interrupt is risky, because it does not use this locking technique. (The processor’s atomic instructions are not able to perform a locking operation that spans several instructions.) This method will work only if the kernel can insure the following:

- that no external I/O agent will attempt to post a pending interrupt simultaneously with the processor, and
- that an interrupt cannot occur after one bit (e.g. the pending priority bit) of the pending-interrupt record is set but before the other bit (the pending interrupt vector) is set.

7.7.10 Checking for Pending Interrupts

The processor automatically checks the interrupt table for pending interrupts at the following times:

- After returning from an interrupt-handler procedure
- While executing a modify-process-controls instruction (modpc), if the instruction causes the program’s priority to be lowered.
- After receiving a test pending interrupts IAC message.

7.7.11 Handling Pending Interrupts

The processor uses the same type of atomic read/write operation to check the interrupt table for pending interrupts as it does for posting pending interrupts. Again, this technique prevents other agents on the bus from accessing the interrupt table until the pending-interrupt check has been completed.

When the processor finds a pending interrupt, it handles it as if it had just received the interrupt. The handling mechanism is the same as is described earlier in this chapter for interrupts that are serviced as soon as they are received.
If the processor finds two pending interrupts at the same priority, it services the interrupt with the highest vector number first.

### 7.8 SIGNALING INTERRUPTS

**Note**

The 80960 architecture does not define a mechanism for signaling interrupts to the processor. The methods of signaling interrupts described in the following section are specific to the 80960KB processor.

The 80960KB processor can be interrupted in any of the following five ways:

- Signal on its interrupt pins
- Signal on its interrupt pins from an external interrupt controller
- An IAC message from external source
- An IAC message from a program in the processor
- A pending interrupt (described earlier in this chapter)

#### 7.8.1 Interrupts From Interrupt Pins

The processor has four interrupt pins, called INTO, INT1, INT2, and INT3. These pins can be configured in either of the following three ways:

- as four interrupt-signal inputs;
- as two interrupt inputs and two pins for handshaking with an interrupt controller such as the Intel 8259A Programmable Interrupt Controller; or
- as one IAC input and three interrupt inputs.

A 32-bit, interrupt-control register in the processor determines how these pins are used. Each interrupt pin is associated with one 8-bit field in the register, as shown in Figure 19.

![Figure 19. Interrupt-Control Register](image)

If the interrupt pins are to be used as four inputs, a different interrupt vector is stored in each of the four fields in the interrupt-control register. Then, when an interrupt is signaled on one of the pins, the processor reads the vector from the pin’s associated field in the register. For example, if an interrupt is signaled on pin INTO, the processor reads the vector from bits 0 through 7.
The processor assumes that the interrupt vectors in the interrupt register are arranged in descending order from the INTO field to the INT3 field (e.g., the priority of INTO ≥ INT1 ≥ INT2 ≥ INT3). To insure that interrupts are handled in the proper order, software should follow this convention.

If the INTO vector field is set to 0, the function of the INTO pin is changed to IAC, and it is used to signal the processor that an external IAC message has been sent to it. In fact, the INTO pin must be configured in this manner for the processor to service external IAC messages.

If the INT2 vector field is set to 0, the functions of the INT2 and INT3 pins are changed to INTR and INTA, respectively. Here, the INTR pin is used to receive signals from an interrupt controller and the INTA pin is used to send acknowledge signals back to the controller. When the processor receives a signal on the INTR pin, it reads an interrupt vector from the least significant 8 bits of its bus, then sends an acknowledge signal to the controller through INTA. When the INT2 and INT3 pins are configured in this manner, the processor ignores the INT3 vector field.

Note
Refer to the 80960KB Hardware Designer’s Reference Manual for more information on the use of INT2 and INT3 pins with an interrupt controller.

The interrupt-control register is memory mapped to addresses FF000004 through FF000007. Only the processor can read or write this register using the synchronous load (synld) and synchronous move (synmov) instructions. External agents on the bus cannot access this register.

The value in the interrupt-control register after the processor is initialized is FF000000.

7.8.2 IAC Interrupts

The processor can also receive an interrupt request by means of the IAC mechanism. (The IAC mechanism is described in detail in Chapter 13.) The interrupt IAC message can be sent to the processor either from an external bus agent, such as an I/O processor or another 80960KB processor, or internally as part of the currently running program. The interrupt vector is contained in the interrupt IAC message.

As with any other IAC message, the processor receives notice of an external interrupt-IAC message through the INTO pin, which has been configured as an IAC pin, as described in the previous section. The processor then reads the IAC message to get the interrupt vector.

A program running on the processor can signal an interrupt through an internal interrupt-IAC message. An internal IAC is sent to the processor by means of a synchronous move instruction. When the processor executes a synchronous move to its IAC message space, it signals an IAC message internally. The processor then reads the IAC message as it would for an external IAC.

8.0 FAULT HANDLING

This section describes the fault handling facilities of the 80960KB processor. The subjects covered include the fault-handling data structures, the software support required for fault handling, and the
fault handling mechanism. A reference section that contains detailed information on each fault type is provided at the end of the section.

8.1 OVERVIEW OF THE FAULT-HANDLING FACILITIES

The processor is able to detect various conditions in code or in its internal state (called “fault conditions”) that could cause the processor to deliver incorrect to inappropriate results or that could cause it to head down an undesirable control path. For example, the processor recognizes divide-by-zero and overflow conditions on integer calculations. It also detects inappropriate operand values, uncompleted memory accesses, or references to incomplete or non-existent system-data structures.

The processor can detect a fault while it is executing a program, an interrupt handler, or a fault handler. (In this section, when a program is referred to, it generally also means any interrupt handler or fault handler that may have been invoked while the processor was working on the program.)

When the processor detects a fault, it handles the fault immediately and independently of the program or handler it is currently working on, using a mechanism similar to that used to service interrupts.

A fault is generally handled with a fault-handling procedure (called a fault handler), which the processor invokes through an implicit procedure call. Prior to making the call, the processor saves the state of the current program and in some cases the state of an incomplete instruction. It also saves information about the faults, which the fault handler can use to correct or recover from the condition that caused the fault.

If the fault handler is able to recover from the fault, the processor can then restore the program to its state prior to the fault and resume work on the program. If the fault handler is not able to recover from the fault, it can take any of several actions to gracefully shut down the processor.

8.2 FAULT TYPES

All of the faults that the processor detects are predefined. These faults are divided into types and subtypes, each of which is given a number. The processor uses the type number to select a fault handler. The fault handler then uses the subtype number to select a specific fault-handling procedure.

Table 14 lists the faults that the processor detects, arranged by type and subtype. For convenience, individual faults are referred to in this chapter by their fault-subtype name. Thus a machine bad-access fault is referred to as simply a bad-access fault, or an arithmetic integer overflow fault is referred to as an integer overflow fault.

The fifth column of Table 14 shows each fault as it appears in the fault record (the word at offset 40 of the fault record is shown later in this section).
### Table 14. Fault Types and Subtypes

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Fault Subtype</th>
<th>Fault Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>No.</td>
<td>Name</td>
<td>No./Bit Position</td>
</tr>
<tr>
<td>1</td>
<td>Trace</td>
<td>Bit 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 7</td>
</tr>
<tr>
<td>2</td>
<td>Operation</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Arithmetic</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Floating Point</td>
<td>Bit 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 5</td>
</tr>
<tr>
<td>5</td>
<td>Constraint</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Protection</td>
<td>Bit 1</td>
</tr>
<tr>
<td>8</td>
<td>Machine</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>Structural</td>
<td>3</td>
</tr>
</tbody>
</table>

**Note**

The 80960 architecture defines a basic set of fault types and subtypes. Processors that provide extensions to the architecture may recognize additional fault conditions. The encoding of fault types and subtypes allows any of these extensions to be included in the fault table along with the basic faults. Space in the fault table will be reserved in such a way that processors that recognize the same fault types and subtypes will encode them in the same way.

For example, the floating-point faults (fault type 4) are an extension provided in the 80960KB processor (but not in the 80960KA processor). Any other processors based on the 80960 architecture that also recognize floating-point faults will also encode them as fault type 4.
8.3 FAULT-HANDLING METHOD

The processor handles all faults through an implicit procedure call to a fault handler. When a fault occurs while the processor is executing a program, the processor creates a fault record on its current stack. This record includes information on the state of the program and data on the fault. If the fault occurred while the processor was in the midst of executing an instruction, a resumption record for the instruction may also be saved on the stack.

Following the creation of the fault and resumption records, the processor selects a fault handler from a system-data structure called the fault table. It then invokes the fault handler (by means of an implicit call) and begins executing the handler procedure. As is described later in this section, the fault handler call can be a local call (call-extended operation), a local system-procedure-table call (local system-call operation), or a supervisor call.

This same procedure call method is used to handle faults that occur while the processor is servicing an interrupt or that occur while the processor is working on a fault handler.

8.3.1 Multiple Fault Conditions

It is possible for multiple fault conditions to occur simultaneously. For certain fault types, such as trace faults or protection faults, bit positions in the fault-subtype field are used to indicate the occurrence of multiple faults of the same type. As a general rule, however, the processor does not indicate situations where multiple faults occur. Instead, it records one of the faults and does not report on the faults that were not recorded.

If a fault occurs while the processor is executing a fault handling routine, the operating of the processor is not predictable.

8.3.2 Faults and Interrupts

If an interrupt occurs during an instruction that will fault, that has just faulted, or that has faulted while the processor is in the midst of selecting the fault handler, the processor will handle the fault in either of the following ways:

- It includes the fault information as part of its interrupt record and services the interrupt immediately. After it has serviced the interrupt, it handles the fault.
- It completes the selection of the fault handler, then services the interrupt just prior to executing the first instruction of the fault handler.

8.4 SOFTWARE REQUIREMENTS FOR HANDLING FAULTS

To use the processor's fault-handling facilities, the following system-data structures and procedures must be present in memory:
Software should generally load these items in memory as part of the initialization procedure. Once they are present in memory and pointers to them have been included in the IMI, the processor then handles faults automatically and independently from software.

Requirements for the fault table and fault-handler procedures are given in the following sections.

8.5 FAULT TABLE

The fault table provides the processor with a pathway to the fault handlers when the processor is using the implicit procedure-cal method of handling faults. As shown in Figure 20, there is one entry in the fault table for each fault type. When a fault occurs, the processor uses the fault type to select an entry in the fault table. From this entry, the processor then obtains a pointer to the fault handler for the type of fault that occurred.

The fault handler reads the fault subtype or subtypes from the fault record to determine the appropriate fault recovery action.

8.5.1 Location of the Fault Table in Memory

The fault table can be located anywhere in the address space. The processor obtains a pointer to the fault table from the IMI.

8.5.2 Fault-Table Entries

Each entry in the fault table is two words long. As shown in Figure 20, there are two types of fault-table entries allowed: local-procedure entry and system-procedure-table entry. The entry-type field determines the entry type.

A local-procedure entry (entry type $00_2$) provides an instruction pointer (address in the address space) for the fault handler procedure. Using this entry, the processor invokes the specified fault handler by means of an implicit call-extended operation (similar to that performed for the callx instruction). The second word of a local-procedure entry is reserved. It should be set to zero when the fault table is created and not accessed after that.

A system-procedure-table entry (entry type $10_2$) provides a procedure number in the system procedure table. Using this entry, the processor invokes the specified fault handler by means of an implicit call-system operation (similar to that performed for the calls instruction).
Fault-handling procedures in the system procedure table can be local procedures or supervisor procedures. A fault handler can thus be invoked through the fault table in any of three ways: implicit local-procedure call, implicit local procedure-table call, or implicit supervisor call.

**Figure 20. Fault Table and Fault-Table Entries**
8.6 FAULT-HANDLER PROCEDURES

The fault-handler procedures can be located anywhere in the address space. Each procedure must begin on a word boundary. The processor can execute the procedure in the user mode or the supervisor mode, depending on the type of fault table entry.

Note
To resume work on a program at the point where a fault occurred (following the recover action of the fault handler), the fault handler must be executed in the supervisor mode. The reason for this requirement is described in "Program and Instruction Resumption Following a Fault" in this section.

Many of the faults that occur can be recovered from easily. When recovery from the fault is possible, the processor's fault-handling mechanism allows the processor to automatically resume work on the program or interrupt that it was working on when the fault occurred. The resumption action is initiated with a `ret` instruction in the fault-handler procedure.

If recovery from the fault is not possible or not desirable, the fault handler can take one of the following actions, depending on the nature and severity of the fault condition (or conditions, in the case of multiple faults):

- Return to a point in the program or interrupt code other than the point of the fault
- Save the current state of the processor and call a debug monitor
- Save the current state of the processor and place the processor in the stopped state (using freeze IAC)
- Explicitly write the processor state, fault record, and instruction resumption record into memory and place the processor in the stopped state
- Place the processor in the stopped state without explicitly saving the processor state or the fault information.

When working with the processor at the development level, a common action of the fault handler is to save the fault and processor state information and make a call to a debugging device such as a debugging monitor. This device can then be used to analyze the fault.

8.6.1 Program and Instruction Resumption Following a Fault

The processor allows work on a program to be resumed at the point where the fault occurred following a return from a fault handler. If an instruction was suspended to handle the fault execution of the instruction can also be resumed on the return.

This resumption mechanism is similar to that provided by returing from an interrupt handler. It is only useful, however, for faults from which recovery is possible, such as the trace faults.
To use this mechanism, the fault handler must be invoked using an implicit supervisor procedure-table call. This method is required because to resume work on the program and a suspended instruction at the point where the fault occurred, the saved process controls in the fault record must be copied back into the processor on the return from the fault handler. The processor only performs this action if the processor is in the supervisor mode on the return.

If the fault handler is invoked with an implicit local-procedure call or an implicit local-procedure-table call, the return IP determines where in the program the processor resumes work, following a return from a fault handler. Here, the return is handled in a similar manner to a return from an explicit call with a `call` or `callx` instruction.

The return IP (referred to later in this section as the saved IP) is saved in the RIP register (r2) of the stack frame that was in use when the fault occurred. This IP may be the instruction the processor faulted on or the next instruction that the processor would have executed if the fault had not occurred. In either case, the resumption record is not used, so the processor might continue work on the program without completing the instruction that the fault occurred on.

A fault handler should thus be invoked with an implicit local-procedure or local-procedure-table call only if it is not required or desirable to resume the program at the point of the fault. The section, “Return Without Resumption”, discusses returning to a point in the program code other than the point of the fault.

### 8.7 FAULT CONTROLS

Certain fault types and subtypes have masks or flags associated with them that determine whether or not a fault is signalled when a fault condition occurs. Table 15 lists these flags and masks, the system data structures in which they are located, and the fault subtype they affect.

<table>
<thead>
<tr>
<th>Flag or Mask Name</th>
<th>Location</th>
<th>Fault Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Overflow Mask</td>
<td>Arithmetic Controls</td>
<td>Integer Overflow</td>
</tr>
<tr>
<td>Floating Overflow Mask</td>
<td>Arithmetic Controls</td>
<td>Floating Overflow</td>
</tr>
<tr>
<td>Floating Underflow Mask</td>
<td>Arithmetic Controls</td>
<td>Floating Underflow</td>
</tr>
<tr>
<td>Floating Invalid Operation Mask</td>
<td>Arithmetic Controls</td>
<td>Floating Invalid Operation</td>
</tr>
<tr>
<td>Floating Zero-Divide Mask</td>
<td>Arithmetic Controls</td>
<td>Floating Zero-Divide</td>
</tr>
<tr>
<td>Floating-point Inexact Mask</td>
<td>Arithmetic Controls</td>
<td>Floating Inexact</td>
</tr>
<tr>
<td>No Imprecise Faults Flag</td>
<td>Arithmetic Controls</td>
<td>All Imprecise Faults</td>
</tr>
<tr>
<td>Trace-Enable Flag</td>
<td>Process Controls</td>
<td>All Trace Faults</td>
</tr>
<tr>
<td>Trace-Mode Flags</td>
<td>Trace Controls</td>
<td>All Trace Faults</td>
</tr>
</tbody>
</table>
The integer and float-point mask bits inhibit faults from being raised for specific fault conditions (i.e., integer overflow and floating-point overflow, underflow, zero divide, invalid operation, and inexact). The use of these masks is discussed in the fault-reference section at the end of this section. Also, the floating-point fault masks are described in Chapter 11 in “Exceptions and Fault Handling”.

The no-imprecise-faults (NIF) flag controls the synchronizing of faults for a category of faults called imprecise faults. This flag should be set to 1. The function of this flag is described later in the section “Precise and Imprecise Faults”.

The trace-mode flags (in the trace controls) and trace-enable flag (in the processor controls) support trace faults. The trace-mode flags enable trace modes; the trace-enable flag enables the generation of trace faults. The use of these flags is described in the fault reference section on trace faults. Further discussion of these flags is provided in Section 9, “Trace-Enable and Trace-Fault-Pending Flags”.

8.8 SIGNALING A FAULT

The processor generates faults implicitly when fault conditions occur and explicitly at the request of software. Most faults are generated implicitly. The fault control bits described in the previous section allow the implicit generation of some faults to be either enabled (as with the trace faults) or masked (as with the floating-point faults).

8.8.1 Fault-If Instructions

The fault-if instructions (faulte, faultne, faultl, faultle, faultg, faultge, faulto, and faultno) allow a fault to be generated explicitly anywhere within an application program, kernel procedure, interrupt handler, or fault handler. When one of these instructions is executed, the processor checks the condition code bits in the arithmetic controls, then signals a constraint-range fault if the condition specified with the instruction is met.

8.9 FAULT RECORD

When a fault occurs, the processor records information about the fault in a fault record. The fault handler and processor use this information to recover from or correct the fault condition and resume execution of the process. Figure 21 shows the structure of the fault record. The use of the fields in this record are described in the following paragraphs.

The type number (byte ordinal) of a fault is stored in the fault-type field; the subtype number or bit positions (byte ordinal) is stored in the fault-subtype field.

The fault-flags field provides a set of general-purpose flags that the processor uses to indicate additional information about a particular fault subtype. Most of the faults do not use these flags, in which case the flags have no defined values.

The address-of-the-faulting-instruction field contains the IP of the instruction that caused the fault or that was being executed when the fault occurred.
The states of the process controls and arithmetic controls at the time that a fault is generated are stored in their respective fields in the fault record. This information is used to resume work on the program after the fault has been handled.

Finally, a three-word fault data field is provided for the fault. The information that is stored in these fields depends on the type of fault that occurs. Any part of a fault-data field that is not used for a particular fault has no defined value. The information that is stored in these fields for each fault type is given in the fault reference section at the end of this section.

8.9.1 Saved Instruction Pointer

The saved IP (the RIP that is saved in r2 of the stack frame in use when the fault occurred) is also part of the fault information that the processor saves when a fault occurs. This IP generally points to the next instruction that the processor would have executed if the fault had not occurred, although it may point to the faulting instruction. It is this instruction that the processor begins working on when the return from the fault handler is initiated.

8.9.2 Resumption Record

If the processor suspends an instruction as the result of a fault, it creates a 48-byte resumption record. The criteria that the processor uses to determine whether or not to suspend an instruction and the structure of the resumption record are the same as are used when an interrupt occurs.
8.9.3 Location of the Fault and Resumption Records

The fault and resumption records are stored in the stack that the processor is using when the fault occurs. This stack can be the local stack, the supervisor stack, or the interrupt stack.

8.10 FAULT HANDLING ACTION

Once a fault has occurred, the processor saves the program state, calls the fault handler, and restores the program state (if this is possible) once the fault recovery action has been completed. No software other than the fault-handler procedures is required to support this activity.

Three different types of implicit procedure calls can be used to invoke the fault handler according to the information in the selected fault-table entry: local call, local call through the system procedure table, and supervisor call (also through the system procedure table).

8.10.1 Implicit, Local Call/Return

When the selected fault-handler entry in the fault table is an entry type 002 (local procedure) the processor performs the following action:

1. The processor stores a fault record as shown in Figure 21 on the top of the stack that the processor is currently using. The stack can be the local stack, the supervisor stack, or the interrupt stack.
2. If the fault caused an instruction to be suspended, the processor includes an instruction resumption record on the current stack and sets the resume flag in the save process controls.
3. The processor creates a new frame on the current stack, with the frame-return status field set to 0012.
4. Using the procedure address from the selected fault-table entry, the processor performs an implicit call-extended operation to the fault handler.

If the fault handler is not able to perform a recovery action, it performs one of the actions described under “Possible Fault-Handler Actions”.

If the handler action results in a recovery from the fault, a ret instruction in the fault handler allows processor control to return to the program that was being worked on when the fault occurred. On the return, the processor performs the following action:

1. The processor deallocates the stack frame created for the fault handler.
2. The processor copies the arithmetic controls field from the fault record into the arithmetic controls register in the processor.
3. The processor then resumes work on the program it was working on when the fault occurred at the instruction in the return IP register.
8.11.2 Implicit, Local Procedure-Table Call

When the fault-handler entry selects an entry in the system procedure table (entry type 10) and the system-procedure-table entry is for local procedure, the processor performs the same action as is described in the previous section for a local procedure call/return. The only difference is that the processor gets the address of the fault handler from the system procedure table rather than from the fault table.

8.11.3 Implicit, Supervisor Call/Return

When the fault-handler entry selects an entry in the system procedure table (entry type 10,2) and the system-procedure-table entry is for a supervisor procedure, the processor performs the same action as is described in the previous section for a local procedure call and return, with the exceptions described in the following paragraphs.

On a supervisor fault-handler call, the processor performs the following additional actions:

1. If the processor is in user mode when the fault occurs, the fault record and resumption record are stored in the local stack. The processor then takes the stack pointer from the procedure table and switches to the supervisor stack. The execution mode is then set to supervisor.
2. If the processor is already in supervisor mode when the fault occurs, the fault record is stored in the current stack (which is the supervisor stack). The processor then creates a new frame on the current stack and begins work on the fault-handler procedure selected from the procedure table.
3. In both of the above cases, the processor copies the state of the trace-control flag (byte 2, bit 1) of the procedure table into the trace-enable flag field of the process controls.

On a return from the fault handler, the processor performs the following additional actions:

1. If the processor is in supervisor mode prior to the return from the fault handler (which it should be), it copies the saved process controls into its internal process controls.
2. If the resume flag of the process controls is set, the processor reads the resumption record from the stack.
3. The processor then resumes work on the program at the point it was working on when the fault occurred.

The restoration of the process controls causes any changes in the process controls through the action of the fault handler to be lost. In particular, if the ret instruction from the fault handler caused the trace-fault-pending flag in the process controls to be set, this setting would be lost on the return.
8.11.4 Program State After a Fault

As has been described earlier in this section, faults can occur prior to the execution of the faulting instruction (i.e., the instruction that causes the fault), during the instruction, or after the instruction. When the fault occurs before the faulting instruction is executed, the instruction can theoretically be executed on the return from the fault handler. So, the fault is not accompanied by a change in the control flow of the program.

When a fault occurs during or after the instruction that caused a fault, the fault may be accompanied by a change in the program's control flow such that the faulting instruction cannot be reexecuted. For example, when an integer-overflow fault occurs, the overflow value is stored in the destination. If the destination register was the same as one of the source registers, the source value is lost, making it impossible to reexecute the faulting instruction.

In general, changes in the program's control flow never accompany the following fault types or subtypes:

- All Operation Subtypes
- Arithmetic Zero-Divide
- All Floating-Point Subtypes Except Floating Inexact
- All Constraint Subtypes
- Prereturn Trace

Changes in the program's control flow always accompany the following fault types and subtypes:

- All Trace Subtypes Except Prereturn Trace
- Integer Overflow
- Floating Inexact

Changes in the program's control flow may or may not accompany the following fault types and subtypes:

- Structural
- Bad Access

The effect that specific fault types have on a program is given in the fault reference section at the end of this section under the heading "Program State Changes."

8.11.5 Return Without Resumption

There may be situations where the fault handler needs to return to a point in the program other than where the fault occurred. This can be done by altering the return IP in the previous frame. However,
if resumption information was collected with the fault (resulting in the resume flag being set in the saved process controls), such a return can cause unpredictable results.

To predictably perform a return from a fault handler to an alternate point in the program, the fault handler should clear the following information in the process-controls field of the fault record before the return: the resume and trace-fault-pending flags; the internal state field.

**Note**

A return of this type can only be performed if the processor is in supervisor mode prior to the return.

### 8.12 PRECISE AND IMPRECISE FAULTS

As described in the Section 2, “Register Scoreboarding,” the 80960KB processor is, in some instances, able to execute instructions concurrently. When two instructions are being executed concurrently, it is possible for them to generate faults simultaneously. When this occurs, one of the faults may not be signaled or may be signaled out of order, making it impossible to recover from that fault.

The processor provides two mechanisms to allow the circumstances under which faults are signaled to be controlled. These mechanisms are the no imprecise faults flag (NIF flag) in the arithmetic controls and the synchronize faults instruction (syncf). The following paragraphs describe how these mechanisms can be used.

Faults are grouped into the following categories: precise, imprecise, and asynchronous.

Precise faults are those that are intended to be recoverable by software. For any instruction that can generate a precise fault, the processor will (1) not execute the instruction if an unfinished prior instruction will fault and (2) not execute subsequent out-of-order instructions that will fault. The following faults are always precise:

- trace
- protection

Imprecise faults are those that in some instances are allowed to occur and not be signaled or be signaled out of order. These faults include the following:

- operation
- arithmetic
- floating point
- constraint
- type
Asynchronous faults are those whose occurrence has no direct relationship to the instruction pointer. This category includes the machine fault.

The NIF flag controls whether or not imprecise faults are allowed. When this flag is set, all faults must be precise. In this mode, the ability to execute instructions concurrently is essentially disabled. All faults that occur are signaled.

When the NIF flag is clear, faults in the imprecise category can in some instances occur and not be signaled. In this mode, the following conditions hold true:

1. When an imprecise fault occurs, the saved IP is undefined (but the address of the faulting instruction in the fault record is valid)
2. If instructions are executed concurrently when an imprecise fault occurs, the results produced by these instructions are undefined.
3. If instructions are executed out-of-order and multiple imprecise faults occur, only one of the faults is generated. The one that is selected is not predictable.

The syncf instruction forces the processor to complete execution of all instructions that occur prior to the syncf instruction and to generate all faults, before it begins work on instructions that occur after the syncf instruction. This instruction has two uses. One use is to force faults to be precise when the NIF is clear. The other use is to insure that all instructions are complete and all faults signaled in one block of code before execution of another block of code (for example, on Ada block boundaries when the blocks have different exception handlers).

The intent of these fault-generating modes is that compiled code should execute with the NIF clear, using the syncf instruction where necessary to ensure that faults occur in order. In this mode, imprecise faults are considered as catastrophic errors from which recovery is not needed.

If recovery from one or more of the imprecise faults is required (for example, a program that needs to handle unmasked floating-point exceptions and recover from them) and the fault handler cannot be closely coupled with the application to perform recovery even if the faults are imprecise, the NIF should be set. Executing with the NIF set will likely lead to slower execution times.

8.13 FAULT REFERENCE

This section describes each of the fault types and subtypes and gives detailed information about what is stored in the various fields of the fault record. The section is organized alphabetically by fault type.

8.13.1 Fault Reference Notation

The following paragraphs describe the information that is provided for each fault type.
8.13.2 Fault Type and Subtype

The fault-type section gives the number entered in the fault-type field of the fault record for the given fault type. The fault-subtype section lists the fault subtypes and their associated number or bit position in the fault-subtype field of the fault record.

8.13.3 Function

The function section gives a general description of the purpose of the fault type, then describes the purpose of each of the fault subtypes in detail. It also describes how the processor handles each fault subtype.

8.13.4 Fault Record

The fault record section describes how the flags, fault-data, and address-of-faulting-instruction fields of the fault record are used for the fault type and subtypes.

8.13.5 Saved IP

The saved IP section describes what value is saved in the RIP register (r2) of the stack frame the processor was using when the fault occurred.

8.13.6 Program State Changes

The program state changes section describes the effects that the fault subtypes have on the control flow of a program.
Arithmetic Faults

Fault Type: \(3_{16}\)

Fault Subtype:  

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Integer Overflow</td>
</tr>
<tr>
<td>2</td>
<td>Arithmetic Zero-Divide</td>
</tr>
<tr>
<td>3-F</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Function: Indicates that there is a problem with an operand or the result of an arithmetic instruction. This fault type applies only to ordinal and integer instruction, not floating-point instructions.

The integer-overflow fault occurs when the result of an integer instruction overflows the destination and the integer-overflow mask in the arithmetic-controls register is cleared. Here, the \(n\) least significant bits of the result are stored in the destination, where \(n\) is the destination size.

The arithmetic zero-divide fault occurs when the divisor operand of an ordinal or integer divide instruction is zero.

Fault Record:  

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Not used.</td>
<td>Not used.</td>
<td>IP for the instruction on which the processor faulted.</td>
</tr>
</tbody>
</table>

Saved IP: IP for the instruction that would have been executed next, if the fault had not occurred.

Prog. State Changes: A change in the program’s control flow accompanies the integer-overflow fault, because the result is stored in the destination before the fault is signaled. The faulting instruction can thus not be reexecuted.

A change in the program’s control flow does not accompany the arithmetic zero-divide fault, because the fault occurs before the execution of the faulting instruction.
Constraint Faults

Fault Type: \( 5_{16} \)

Fault Subtype:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Constraint Range</td>
</tr>
<tr>
<td>2-F</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Function:

Indicates that the processor is either in or not in the required state for the instruction to be executed.

The constraint-range fault occurs when a fault-if instruction is executed and the condition code in the arithmetic controls matches the condition required by the instruction.

Fault Record:

- Flags: Not used.
- Fault Data: Not used.
- Addr. Fault. Inst.: IP for the instruction on which the processor faulted

Saved IP: Not used.

Prog. State Changes:

No changes in the program’s control flow accompany the constraint-range fault. This fault occurs after the fault-if instruction has been executed, but the instruction has no effect on the program state.
Floating-Point Faults

<table>
<thead>
<tr>
<th>Fault Type:</th>
<th>416</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault Subtype:</td>
<td>Bit Number</td>
</tr>
<tr>
<td>Bit 0</td>
<td>Floating Overflow</td>
</tr>
<tr>
<td>Bit 1</td>
<td>Floating Underflow</td>
</tr>
<tr>
<td>Bit 2</td>
<td>Floating Invalid-Operation</td>
</tr>
<tr>
<td>Bit 3</td>
<td>Floating Zero-Divide</td>
</tr>
<tr>
<td>Bit 4</td>
<td>Floating Inexact</td>
</tr>
<tr>
<td>Bit 5</td>
<td>Floating Reserved-Encoding</td>
</tr>
<tr>
<td>Bits 6 and 7</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Function:

Indicates that there is a problem with an operand or the result of a floating-point instruction. Each floating-point fault is assigned a bit in the fault-subtype field. Multiple floating-point faults can only occur simultaneously, however, with the floating-overflow, floating-underflow, and floating-inexact faults.

The floating-point faults are described in detail in the section in Chapter 12 titled "Exceptions and Fault Handling." The following paragraphs give a brief description of each floating-point fault.

A floating-overflow fault occurs when (1) the floating-point overflow mask is clear and (2) the infinitely precise result of a floating-point instruction exceeds the largest allowable finite value for the specified destination format. This fault interacts with the floating-inexact fault (as described in Chapter 12).

A floating-underflow fault occurs when (1) the floating-point underflow mask is clear and (2) the infinitely precise result of a floating-point instruction is less than the smallest possible normalized, finite value for the specified destination format. This fault interacts with the floating-inexact fault (as described in Chapter 12).

The floating invalid-operation fault occurs when (1) the floating-point invalid-operation mask is clear and (2) one of the source operands for a floating-point instruction is inappropriate for the type of operation being performed.

The floating zero-divide fault occurs when (1) the floating-point zero-divide mask is clear and (2) the divisor operand of a floating-point divide instruction is zero.

The floating-inexact fault occurs when (1) the floating-point inexact mask is clear and (2) an infinitely precise result cannot be encoded in the format specified for the destination operand. This fault interacts with the floating-overflow and floating-underflow faults (as described in Chapter 12).

The floating reserved-encoding fault occurs when a denormalized value is used as an operand in a floating-point instruction and the normalizing-mode bit in the arithmetic controls is clear.
### Fault Record

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flags:</strong></td>
</tr>
<tr>
<td><strong>F0</strong> — Used if inexact fault occurs in conjunction with overflow or underflow fault. If set, F0 indicates that the adjusted result has been rounded toward +∞; if clear, F0 indicates that the adjusted result has been rounded toward -∞.</td>
</tr>
<tr>
<td><strong>F1</strong> — Used with overflow and underflow faults only. If set, F1 indicates that the adjusted result has been bias adjusted, because its exponent was outside the range of the extended-real format.</td>
</tr>
</tbody>
</table>

| **Fault Data:** |
| Used only with overflow and underflow faults. Adjusted result is stored in this field in extended-real format (as shown in Figure 12-5). |

| **Addr. Fault. Inst.:** |
| IP for the instruction on which the processor faulted |

| **Saved IP:** |
| IP for the instruction that would have been executed next, if the fault had not occurred. |

| **Prog. State Changes:** |
| Changes in the program’s control flow accompany the floating-overflow, floating-underflow, and floating-inexact faults, because a result is stored in the destination before the fault is signaled. The faulting instruction can thus not be reexecuted. |

| Changes in the program’s control flow do not accompany the floating invalid-operation, floating zero-divide, and floating reserved-encoding faults, because the faults occur before the execution of the faulting instruction. |
Machine Faults

Fault Type: $8_{16}$

Fault Subtype:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Bad Access</td>
</tr>
<tr>
<td>2-F</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Function: Indicates that the processor has detected a hardware or memory-system error.

The bad-access fault is the only one of this fault type. This fault occurs whenever an unrecoverable memory error occurs on a memory operation. In the 80960KB processor, the processor receives a signal on its bad access pin (BADAC) to indicate an unrecoverable memory error. Upon receiving this signal, the processor signals a machine bad access fault. There is one exception to this action. The processor will not signal a machine bad access fault while executing any of the synchronous load or move instructions. Instead, it sets the condition code bits to indicate whether or not the memory access was completed successfully.

Fault Record:

<table>
<thead>
<tr>
<th>Flags:</th>
<th>Not used.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault Data:</td>
<td>Not used.</td>
</tr>
<tr>
<td>Addr. Fault. Inst.:</td>
<td>Not used.</td>
</tr>
</tbody>
</table>

Saved IP: Not used.

Prog. State Changes: This fault may occur at any time. When it does occur, the accompanying state of the program’s control flow is undefined. As a result, the processor is not able to return predictably from the fault handler to the point in the program where the fault occurred.

If this fault occurs during an atomic operation, there is no guarantee that the locking mechanism that memory uses for synchronization is unlocked.
Operation Faults

Fault Type: \( \frac{16}{2} \)

Fault Subtype: Number Name
0 Reserved
1 Invalid Opcode
2 Unimplemented
3 Reserved
4 Invalid Operand
5 - F Reserved

Function: Indicates that the processor cannot execute the current instruction because of invalid instruction syntax or operand semantics.

The invalid-opcode fault occurs when the processor attempts to execute an instruction that contains an undefined opcode or addressing mode.

The unimplemented fault occurs when unaligned memory accesses are not allowed and the processor attempts to access an unaligned word or group of words in memory. (The 80960KB processor does allow unaligned memory accesses, so this fault never occurs.)

The invalid-operand fault occurs when the processor attempts to execute an instruction for which one or more of the operands have special requirements and one or more of the operands do not meet these requirements. This fault subtype is not generated on floating-point instructions.

Fault Record: Flags: Not used.

Fault Data: Not used.

Addr. Fault. Inst.: IP for the last instruction executed in the process.

Saved IP: IP for the instruction that would have been executed next, if the fault had not occurred.

Prog. State Changes: A change in the program’s control flow does not accompany the operation faults, because the faults occur before the execution of the faulting instruction.
Protection Faults

Fault Type: $7_{16}$

Fault Subtype:

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0</td>
<td>Reserved</td>
</tr>
<tr>
<td>Bit 1</td>
<td>Length</td>
</tr>
<tr>
<td>Bit 2-7</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Function:
Indicates that the index operand used in a `calls` instruction points to an entry beyond the extent of the system procedure table.

Fault Flags: Not used.

Fault Data: Not used.

Addr. Fault. Inst.: IP for the instruction on which the processor faulted.

Saved IP: Same as the address-of-faulting-instruction field.

Prog. State Changes: A change in the program’s control flow does not accompany the protection length fault.
Trace Faults

Fault Type: $1_{16}$

Fault Subtype: Bit Number Name
Bit 0 Reserved
Bit 1 Instruction Trace
Bit 2 Branch Trace
Bit 3 Call Trace
Bit 4 Return Trace
Bit 5 Prereturn Trace
Bit 6 Supervisor Trace
Bit 7 Breakpoint Trace

Function: Indicates that the processor has detected one or more trace events. The processor's event tracing mechanism is described in detail in Chapter 10.

A trace event is the occurrence of a particular instruction or type of instruction in the instruction stream. The processor recognizes seven different trace events (instruction, branch, call, return, prereturn, supervisor, and breakpoint). It detects these events, however, only if a mode bit is set for the event in the trace controls word, which is cached in the processor chip. If, in addition, the trace-enable flag in the process controls is set, the processor generates a fault when a trace event is detected.

The fault is generated following the instruction that causes a trace event (or prior to the instruction for the prereturn trace event).

The following trace modes are available:

- **Instruction** — Generate trace event following any instruction.
- **Branch** — Generate trace event following any branch instruction when branch is taken.
- **Call** — Generate trace event following any call or branch-and-link instruction, or implicit procedure call (i.e., call to fault or interrupt handler).
- **Return** — Generate trace event following any return instruction.
- **Prereturn** — Generate trace event prior to any return instruction.
- **Supervisor** — Generate trace event following any call-system instruction.
- **Breakpoint** — Generate trace event following any processor action that causes a breakpoint condition.

There is a trace fault subtype and a bit in the fault-subtype field associated with each of these modes. Multiple fault subtypes can
occur simultaneously, with the fault-subtype bit set for each subtype that occurs.

When a fault type other than a trace fault occurs during the execution of an instruction that causes a trace event, the non-trace-fault is handled before the trace fault. An exception to this rule is the prereturn trace fault. The prereturn trace fault will occur before the processor has a chance to detect a non-trace-fault, so it is handled first.

Likewise, if an interrupt occurs during an instruction that causes a trace event, the interrupt is serviced before the trace fault is handled. Again, the prereturn trace fault is an exception. Since it occurs before the instruction, it will be handled before any interrupt that might occur during the execution of the instruction.

Fault Record:  
Flags:  
Not used.

Fault Data:  
Not used.

Addr. Fault. Inst.:  
IP for the instruction that caused the trace event, except for the prereturn trace fault. For the prereturn trace fault, this field has no defined value.

Saved IP:  
IP for the instruction that would have been executed next, if the fault had not occurred.

Prog. State Changes:  
A change in the program’s control flow accompanies all the trace faults (except the prereturn trace fault), because the events that can cause a trace fault occur after the faulting instruction is completed. As a result, the faulting instruction cannot be reexecuted upon returning from the fault handler.

Since the prereturn trace fault occurs before the return instruction is executed, a change in the program’s control flow does not accompany this fault and the faulting instruction can be executed upon returning from the fault handler.
Type Faults

Fault Type: \( A_{16} \)

Fault Subtype:  

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Type Mismatch</td>
</tr>
<tr>
<td>2-F</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Function: Indicates that an attempt was made to execute the modpc instruction while the processor was in the user mode.

Fault Record:  

Flags: Not used.

Fault Data: Not used.

Addr. Fault. Inst.: IP for the instruction on which the processor faulted

Saved IP: Not used.

Prog. State Changes: When a type mismatch fault occurs, the accompanying state of the program is undefined. The processor is thus not able to return predictably from the fault handler to the point in the program where the fault occurred.
9.0 DEBUGGING

This section describes the tracing facilities of the 80960KB processor, which allow the monitoring of instruction execution.

9.1 OVERVIEW OF THE TRACE-CONTROL FACILITIES

The 80960KB processor provides facilities for monitoring the activity of the processor by means of trace events. A trace event in the 80960KB is a condition where the processor has just completed executing a particular instruction or type of instruction, or where the processor is about to execute a particular instruction.

By monitoring trace events, debugging software is able to display or analyze the activity of the processor or of a program. This analysis can be used to locate software or hardware bugs or for general system monitoring during the development of system or applications programs.

The typical way to use this tracing capability is to set the processor to detect certain trace events either by means of the trace-controls word or a set of breakpoint registers. An alternate method of creating a trace event is with the mark and force mark (fmark) instructions. These instructions cause an explicit trace event to be generated when the processor detects them in the instruction stream.

If tracing is enabled, the processor signals a trace fault when it detects a trace event. The fault handler for trace faults can then call the debugging monitor software to display or analyze the state of the processor when the trace event occurred.

9.2 REQUIRED SOFTWARE SUPPORT FOR TRACING

To use the processor’s tracing facilities, software must provide trace-fault handling procedures, perhaps interfaced with a debugging monitor. Software must also manipulate several control flags to enable the various tracing modes and to enable or disable tracing in general. These control flags are located in the system-data structures described in the next section.

9.3 TRACE CONTROLS

The following flags or fields control tracing:

- Trace controls
- Trace-enable flag in the process controls
- Trace-fault-pending flag in the process controls
- Trace flag (bit 0) in the return-status field of register r0
- Trace-control flag in the supervisor-stack-pointer field of the system table or a procedure table
9.3.1 Trace-Controls Word

The trace-controls word is cached internally in the processor.

The trace controls allow software to define the conditions under which trace events are generated. Figure 22 shows the structure of the trace-controls word.

![Figure 22. Trace-Controls Word](image)

This word contains two sets of bits: the mode flags and the event flags. The mode flags define a set of trace modes that the processor can use to generate trace events. A mode represents a subset of instructions that will cause trace events to be generated. For example, when the call-trace mode is enabled, the processor generates a trace event whenever a call or branch-and-link operation is executed. To enable a trace mode, the kernel sets the mode flag for the selected trace mode in the trace controls. The trace modes are described later in this section.

The processor uses the event flags to keep track of which trace events (for those trace modes that have been enabled) have been detected.

A special instruction, the modify-trace-controls (modtc) instruction, allows software to set or clear flags in the trace controls. On initialization, all the flags in the processor’s internal trace controls are cleared. The modtc instruction can then be used to set or clear trace mode flags as required.

Software can access the event flags using the modtc instruction, however, there is no reason to. The processor modifies these flags as part of its trace-handling mechanism.

Bits 0, 8 through 06, and 24 through 31 of the trace controls are reserved. Software should initialize these bits to zero and not modify them.
9.3.2 Trace-Enable and Trace-Fault-Pending Flags

The trace-enable flag and the trace-fault-pending flag, in the process controls (shown in Figure 14), control tracing. The trace-enable flag enables the processor’s tracing facilities. When this flag is set, the processor generates trace faults on all trace events.

Typically, software selects the trace modes to be used through the trace controls. It then sets the trace-enable flag when tracing is to begin. This flag is also altered as part of some of the call and return operations that the processor carries out, as described at the end of this section.

The trace-fault-pending flag allows the processor to keep track of the fact that an enabled trace event has been detected. The processor uses this flag as follows. When the processor detects an enabled trace event, it sets this flag. Before executing an instruction, the processor checks this flag. If the flag is set, it signals a trace fault. By providing a means of recording the occurrence of a trace event, the trace-fault-pending flag allows the processor to service an interrupt or handle a fault other than a trace fault, before handling the trace fault. Software should not modify this flag.

9.3.4 Trace Control on Supervisor Calls

The trace flag and the trace-control flag allow tracing to be enabled or disabled when a call-system instruction (calls) is executed that results in a switch to supervisor mode. This action occurs independent of whether or not tracing is enabled prior to the call.

When a supervisor call is executed (calls instruction that references an entry in the system procedure table with an entry type 11x), the processor saves the current state of the trace-enable flag (from the process controls) in the trace flag (bit 0) of the return-status field of register r0.

Then, when the processor selects the supervisor procedure from the procedure table, it sets the trace-enable flag in the process controls according to the setting in the trace-control flag in the procedure table (bit 0 of the word that contains the supervisor-stack pointer).

On a return from the supervisor procedure, the trace-enable flag in the process controls is restored to the value saved in the return-status field of register r0.

9.4 TRACE MODE

The following trace modes can be enabled through the trace controls:

- Instruction trace
- Branch trace
- Call trace
- Return trace
- Prereturn trace
• Supervisor trace
• Breakpoint trace

These modes can be enabled individually or several modes can be enabled at once. Some of these modes overlap, such as the call-trace mode and the supervisor-trace mode. The section “Handling Multiple Trace Events” describes what the processor does when multiple trace events occur.

The following sections describe each of the trace modes.

9.4.1 Instruction Trace

When the instruction-trace mode is enabled, the processor generates an instruction-trace event each time an instruction is executed. This mode can be used within a debugging monitor to single-step the processor.

9.4.2 Branch Trace

When the branch-trace mode is enabled, the processor generates a branch-trace event any time a branch instruction that branches is executed. A branch-trace event is not generated for conditional-branch instructions that do not branch. Also, branch-and-link, call, and return instructions do not cause branch-trace events to be generated.

9.4.3 Call Trace

When the call-trace mode is enabled, the processor generates a call-trace event any time a call instruction (call, callx, or calls) or a branch-and-link instruction (bal or balx) is executed. An implicit call, such as the action used to invoke a fault handler or an interrupt handler, also causes a call-trace event to be generated.

When the processor detects a call-trace event, it also sets the prereturn-trace flag (bit 3 of register r0) in the new frame created by the call operation or in the current frame if a branch-and-link operation was performed. The processor uses this flag to determine whether or not to signal a prereturn-trace event on a return instruction.

9.4.4 Return Trace

When the return-trace mode is enabled, the processor generates a return-trace event any time a ret instruction is executed.

9.4.5 Prereturn Trace

The prereturn-trace mode causes the processor to generate a prereturn-trace event prior to the execution of any ret instruction, providing the prereturn-trace flag in r0 is set. (Prereturn tracing cannot be used without enabling call tracing.)
The processor sets the prereturn-trace flag whenever it detects a call-trace event (as described above for the call-trace mode). This flag performs a prereturn-trace-pending function. If another trace event occurs at the same time as the prereturn-trace event, the prereturn-trace flag allows the processor to fault on the non-prereturn-trace event first, then come back and fault again on the prereturn-trace event. The prereturn trace is the only trace event that can cause two successive trace faults to be generated between instruction boundaries.

9.4.6 Supervisor Trace

When the supervisor-trace mode is enabled, the processor generates a supervisor-trace event any time (1) a call-system instruction (calls) is executed, where the procedure table entry is a supervisor procedure, or (2) when a ret instruction is executed and the return-status field is set 0102 or 0112 (i.e., return from supervisor mode).

This trace mode allows a debugging program to determine the boundaries of kernel procedure calls within the instruction stream.

9.4.7 Breakpoint Trace

The breakpoint-trace mode allows trace events to be generated at places other than those specified with the other trace modes. This mode is used in conjunction with the mark and force-mark (fmark) instructions, and the breakpoint registers.

The mark and fmark instructions allow breakpoint-trace events to be generated at specific points in the instruction stream. When the breakpoint-trace mode is enabled, the processor generates a breakpoint-trace event any time it encounters a mark instruction. The fmark causes the processor to generate a breakpoint-trace event regardless of whether the breakpoint-trace mode is enabled or not.

The processor has two, one-word breakpoint registers, designated as breakpoint 0 and breakpoint 1. Using the set-breakpoint-register IAC, one instruction pointer can be loaded into each register. The processor then generates a breakpoint trace any time it executes an instruction referenced in a breakpoint register.

9.5 TRACE-FAULT HANDLER

A fault handler is a procedure that the processor calls to handle faults that occur. The requirements for fault handlers are given in Section 8, "Fault-Handler Procedures."

A trace-fault handler has one additional restriction. It must be called with an implicit supervisor call, and the trace-control flag in the system-procedure-table entry must be clear. This restriction insures that tracing is turned off when a trace fault is being handled, which is necessary to prevent an endless loop.
9.6 SIGNALING A TRACE EVENT

To summarize the information presented in the previous sections, the processor signals a trace event when it detects any of the following conditions:

- An instruction included in a trace-mode group is executed or about to be executed (in the case of a prereturn trace event) and the trace mode for that instruction is enabled.
- An implicit call operation has been executed and the call-trace mode is enabled.
- A mark instruction has been executed and the breakpoint-trace mode is enabled.
- An fmark instruction has been executed.
- An instruction specified in a breakpoint register is executed and the breakpoint-trace mode is enabled.

When the processor detects a trace event and the trace-enable flag in the process controls is set, the processor performs the following action:

1. The processor sets the appropriate trace-event flag in the trace controls. If a trace event meets the conditions of more than one of the enabled trace modes, a trace-event flag is set for each trace mode condition that is met.
2. The processor sets the trace-fault-pending flag in the process controls.

Note

The processor may set a trace-event flag and the trace-fault-pending flag before it has completed execution of the instruction that caused the event. However, the processor only handles trace events in between the execution of instructions.

If, when the processor detects a trace event, the trace-enable flag in the process controls is clear, the processor sets the appropriate event flags, but does not set the trace-fault-pending flag.

9.7 HANDLING MULTIPLE TRACE EVENTS

If the processor detects multiple trace events, it records one or more of them based on the following precedence, where 1 is the highest precedence:

1. Supervisor-trace event
2. Breakpoint- (from mark or fmark instruction, or from a breakpoint register), branch-, call-, or return-trace event
3. Instruction-trace event

When multiple trace events are detected, the processor may not signal each event; however, it will signal at least the one with the highest precedence.
9.8 TRACE HANDLING ACTION

Once a trace event has been signaled, the processor determines how to handle the trace event, according to the setting of the trace-enable and trace-fault-pending flags in the process controls and to other events that might occur simultaneously with the trace event such as an interrupt or a non-trace fault.

The following sections describe how the processor handles trace events for various situations.

9.8.1 Normal Handling of Trace Events

Prior to executing an instruction, the processor performs the following action regarding trace events:

1. The processor checks the state of the trace-fault pending flag. If this flag is clear, the processor begins execution of the next instruction. If the flag is set, the processor performs the following actions.

2. The processor checks the state of the trace-enable flag. If the trace-enable flag is clear, the processor clears any trace event flags that have been set, prior to starting execution of the next instruction. If the trace-enable flag is set, the processor performs the following action.

3. The processor signals a trace fault and begins the fault handling action, as described in Section 8.

9.8.2 Prereturn Trace Handling

The processor handles a prereturn-trace event the same as described above except when it occurs at the same time as a non-trace fault. Here, the non-trace fault is handled first.

On returning from the fault handler for the non-trace fault, the processor checks the prereturn-trace flag in register r0. If this flag is set, the processor generates a prereturn-trace event, then handles it as described above.

9.8.3 Tracing and Interrupt Handlers

When the processor invokes an interrupt handler to service an interrupt, it disables tracing. It does this by saving the current state of the process controls, then clearing the trace-enable and trace-fault-pending flags in the current process controls.

On returning from the interrupt handler, the processor restores the process controls to the state they were in prior to handling the interrupt, which restores the state of the trace-enable and trace-fault-pending flags. If these two flags were set prior to calling the interrupt handler, a trace fault will be signaled on the return from the interrupt handler.
9.8.4 Tracing and Fault Handlers

The processor can invoke a fault handler with either an implicit local call or an implicit supervisor call. On a local call, the trace-enable and trace-fault-pending flags are neither saved on the call nor restored on the return. The state of these flags on the return is thus dependent on the action of the fault handler.

On a supervisor call, the trace-enable and trace-fault-pending flags are saved, as part of the saved process controls, and restored on the return. So, if these two flags were set prior to calling the fault handler, a trace fault will be signaled on the return from the fault handler.

Note
On a return from an interrupt handler or a fault handler (other than the trace-fault handler), the trace-fault-pending flag is restored. If this flag is set as a result of the handler’s ret instruction, the detected trace event is lost.

10. INSTRUCTION SET REFERENCE

This section provides detailed information about each of the instructions for the 80960KB processor. To provide quick access to information on a particular instruction, the instructions are listed alphabetically by assembly-language mnemonic. An explanation of the format and abbreviations used in this section is given later.

10.1 INTRODUCTION

The information in this section is oriented toward programmers who are writing assembly-language code for the 80960KB processor. The information provided for each instruction includes the following:

- Alphabetic reference
- Assembly-language mnemonic and name
- Assembly-language format
- Description of the instruction’s operation
- Action the instruction carries out when executed (generally presented in the form of an algorithm)
- Faults that can occur during execution
- Assembly-language example
- Opcode and instruction format
- Related instructions

Additional information about the instruction set can be found in the following sections and appendices in this chapter:
10.2 NOTATION

To simplify the presentation of information about the instructions, a simple notation has been adopted in this section. The following paragraphs describe this notation.

10.2.1 Alphabetic Reference

The instructions are listed alphabetically by assembly-language mnemonic. If several instructions are related and fall together alphabetically, they are described as a group on a single page.

The reference at the top of each page gives the assembly-language mnemonics for the instructions covered on that page (e.g., \texttt{subc}). Occasionally, there are so many instructions covered on the page that it is not practical to give all the mnemonics in the page reference. In these cases, the name of the instruction group is given in capital letters (e.g., \texttt{BRANCH} or \texttt{FAULT IF}). A box around the alphabetic reference indicates that the instruction or group of instructions are extensions to the 80960 architecture instruction set.

10.2.3 Mnemonic

The Mnemonic section gives the complete mnemonic (in bold-face type) and instruction name for each instruction covered on the page, for example:

\begin{verbatim}
subi     Subtract Integer
\end{verbatim}

10.2.4 Format

The Format section gives the assembly-language format of the instruction and the type of operands allowed. The format is given in two or three lines. The following is an example of a two line format:

\begin{verbatim}
sub*     src1    src2,   dst
         reg/lit  reg/lit  reg
\end{verbatim}

The first line gives the assembly-language mnemonic (bold-face type) and the operands (italics). When the format is used for two or more instructions, an abbreviated form of the mnemonic is used. The "*" sign at the end of the mnemonic indicates that the mnemonic has been abbreviated.

The operand names are designed to describe the functions of the operands (e.g., \texttt{src}, \texttt{len}, \texttt{mask}).

3-113
The second line of the format shows what is allowed to be entered for each operand. The notation used on this line is as follows:

- **reg**: Global (g0 ... g5) or local (r0 ... r5) register
- **freg**: Global (g0 ... g5) or local (r0 ... r5) register, or floating-point (fp0 ... fp3) register, where the registers contain floating-point numbers
- **lit**: Integer or ordinal literal of the range 0 ... 31
- **flit**: Floating-point literal of value 1.0 or 0.0
- **disp**: Signed displacement of range \(-2^{22} \ldots (2^{22} - 1)\)
- **mem**: Address defined with the full range of addressing modes

In some cases, a third line will be added to show specifically what will be in a register or memory location. For example, it may be useful to know that a register is to contain an address. The notation used in this line is as follows:

- **addr**: Address
- **efa**: Effective address

### 10.2.5 Description

The Description section describes what the instruction does and the functions of the operands. It also gives programming hints when appropriate.

### 10.2.6 Action

The Action section gives an algorithm written in a pseudo-code that describes in detail what actions the processor takes when executing the instruction and the precise order of these actions. The main purpose of this section is to show the possible side effects of the instruction. The following is an example of the action algorithm for the alterbit instruction:

```plaintext
if (AC.cc and 2#010#) = 0
    then dist ← src and not (2^(bitpos mod 32));
else dst ← src or 2^(bitpos mod 32);
end if;
```

In these action statements, the term AC.cc means the condition-code bits in the arithmetic controls. The notation 2#value# means that the value enclosed in the “#” signs is in base 2.

### 10.2.7 Faults

The Faults section lists the faults that can be signaled as the result of execution of the instruction. Faults listed with all-capital letters refer to a group of faults; faults listed with initial-capital letters refer to a specific fault.
All instructions can signal a group of general faults which are referred to as STANDARD FAULTS. The standard faults include the trace-instruction and machine-bad-access faults. In addition, for all instructions have a MEM machine-format (such as load, store, call extended), the invalid-opcode and operation-unimplemented faults are standard faults.

The following list shows the various fault groups and the individual faults in each group:

TRACE FAULT
- Instruction Trace
- Branch Trace
- Call Trace
- Return Trace
- Prereturn Trace
- Supervisor Trace
- Breakpoint Trace

OPERATION
- Invalid Opcode
- Unimplemented
- Invalid Operand

ARITHMETIC
- Integer Overflow
- Arithmetic Zero-Divide

FLOATING-POINT
- Floating Overflow
- Floating Underflow
- Floating Invalid-Operation
- Floating Zero-Divide
- Floating Inexact
- Floating Reserved-Encoding

CONSTRAINT
- Constraint Range
- Privileged
10.2.8 Example

The Example section gives an assembly-language example of an application of the instruction.

10.2.9 Opcode and Instruction Format

The Opcode and Instruction Format section gives the opcode and machine language instruction format for each instruction, for example:

```
subi 593 REG
```

The opcode is given in hexadecimal format.

The machine language format is one of four possible formats: REG, COBR, CTRL, and MEM. Refer to Appendix B for more information on the machine-language instruction formats.

10.2.10 See Also

The See Also section gives the mnemonics of related instructions, which can then be looked up alphabetically in this section for comparison. For instructions that are grouped on one page (such as `addr` and `addri`) only the first mnemonic is given.

10.2 INSTRUCTION

This section contains reference information on the processor’s instructions. It is arranged alphabetically by instruction or instruction group.
addc

Mnemonic: addc  Add Ordinal With Carry

Format: addc  
         src1,  src2,  dst  
         reg/lit  reg/lit  reg

Description: Adds the src2 and src1 values, and bit 1 of the condition code (used here as a carry in), and stores the result in dst. If the ordinal addition results in a carry, bit 1 of the condition code is set; otherwise, bit 1 is cleared. If integer addition results in an overflow, bit 0 of the condition code is set; otherwise, bit 0 is cleared. Regardless of the results of the addition, bits 0 and 1 of the arithmetic controls are always written.

The addc instruction can be used for either ordinal or integer arithmetic. The instruction does not distinguish between ordinal and integer source operands. Instead, the processor evaluates the result for both data types and sets bits 0 and 1 of the condition code accordingly.

An integer overflow fault is never signaled with this instruction.

Action: 
# Let the value of the condition code be xCn.  
dst ← src2 + src1 + C;  
AC.cc ← 2#OCV#;  
# C is carry from ordinal addition.  
# V is 1 if integer addition would have generated an overflow.

Faults: STANDARD

Example:  
# Example of double-precision arithmetic  
# Assume 64-bit source operands  
# in g0,g1 and g2,g3  
cmpo 1, 0  # clears Bit 1 (carry bit) of  
        # the AC.cc  
addc g0, g2, g0  # add low-order 32 bits;  
        # g0 ← g2 + g0 + Carry Bit  
addc g1, g3, g1  # add high-order 32 bits;  
        # g1 ← g3 + g1 + Carry Bit  
# 64-bit result is in g0, g1

Opcode:  
addc  5B0  REG

See Also: addo, subc
addi, addo

Mnemonic:  
addi  Add Integer  
addo  Add Ordinal

Format:  
add*  src1,  src2,  dst  
reg/lit  reg/lit  reg

Description:  Adds the src2 and src1 values and stores the result in dst.

Action:  \( dst \leftarrow src2 + src1; \)

Faults:  
STANDARD  
Integer Overflow  
Refer to discussion of faults at the beginning of this chapter.  
Result is too large for destination format.  
This fault is signaled only when executing the addi instruction and if both of the following conditions are met: (1) the integer-overflow mask in the arithmetic-controls registers is clear and (2) the source operands have like signs and the sign of the result operand is different than the signs of the source operands.

Example:  
addi r4, g5, r9  \# r9 \leftarrow g5 + r4

Opcode:  
addi  591  REG  
addo  590  REG

See Also:  addc, addr, subi, subo
addr, addrl

Mnemonics:  
addr  Add Real
addrl  Add Long Real

Format:  
addr*  srcl,  src2,  dst
        freg/flit  freg/flit  freg

Description:  Adds the src2 and src1 values and stores the result in dst.

For the addrl instruction, if the src1, src2, or dst operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>src1</th>
<th>-∞</th>
<th>-F</th>
<th>-0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>-F</td>
<td>-∞</td>
<td>-F</td>
<td>src2</td>
<td>src2</td>
<td>±F or ±0</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>-0</td>
<td>-∞</td>
<td>src1</td>
<td>-0</td>
<td>±0</td>
<td>src1</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>-∞</td>
<td>src1</td>
<td>±0</td>
<td>+0</td>
<td>src1</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>-∞</td>
<td>±F or ±0</td>
<td>src2</td>
<td>src2</td>
<td>+F</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Notes:
F  Means finite-real number
*  Indicates floating invalid-operation exception

When the sum of two operands with opposite signs is zero, the result is +0, except for the round toward -∞ mode, in which case, the result is -0. When zero is added to itself (e.g. src1 + src1, where src1 is 0), the result retains the sign of the source.

Action:  
dst ← src2 + src1;
Faults: STANDARD

Floating Reserved Encoding
One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow
Result is too large for destination format.

Floating Underflow
Normalized result is too small for destination format.

Floating Invalid Operation
Source operands are infinities of unlike sign.
One or more operands is an SNaN value.

Floating Inexact
Result cannot be represented exactly in destination format.
Floating overflow occurred and the overflow exception was masked.

Example: addrl g6, g8, fp3  #fp3 ← g6,g7 + g8,g9

Opcode: addr 78F REG
addrl 79F REG

See Also: addi, subr
alterbit

Mnemonic: alterbit Alter Bit

Format: alterbit bitpos, src, dst
        reg/lit reg/lit reg

Description: Copies the src value to dst with one bit altered. The bitpos operand specifies
the bit to be changed; the condition code determines the value the bit is to be
changed to. If the condition code is X1X2, the selected bit is set; otherwise,
it is cleared.

Action: if (AC.cc and 2#010#) = 0
        then dst <- src and not (2^(bitpos mod 32));
        else dst <- src or 2^(bitpos mod 32);
        end if;

Faults: STANDARD

Example: # assume condition code is 2#010#
         alterbit 24, g4, g9  # g9 <- g4, with bit 24 set

Opcode: alterbit 58F REG

See Also: checkbit, clearbit, notbit, setbit
and, andnot

Mnemonics:  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and</td>
<td>src1, src2, dst</td>
<td>src2 and src1 values and stores the result in dst.</td>
</tr>
<tr>
<td>andnot</td>
<td>andnot</td>
<td>src1, src2, dst</td>
<td>src2 and src1 values and stores the result in dst.</td>
</tr>
</tbody>
</table>

Format:  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and</td>
<td>src1, src2, dst</td>
<td>src2 and src1 values and stores the result in dst.</td>
</tr>
<tr>
<td>andnot</td>
<td>andnot</td>
<td>src1, src2, dst</td>
<td>src2 and src1 values and stores the result in dst.</td>
</tr>
</tbody>
</table>

Description:  

Performs a bitwise AND (and instruction) or AND NOT (andnot instruction) operation on the src2 and src1 values and stores the result in dst. Note in the action expressions below, the src2 operand comes first, so that with the andnot instruction the expression is evaluated as

\[ \{ \text{src2 andnot (src1)} \} \]

rather than

\[ \{ \text{src1 andnot (src2)} \} \].

Action:  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and</td>
<td>dst ← src2 and src1;</td>
<td>dst ← src2 and src1;</td>
</tr>
<tr>
<td>andnot</td>
<td>andnot</td>
<td>dst ← src2 and not (src1);</td>
<td>dst ← src2 and not (src1);</td>
</tr>
</tbody>
</table>

Faults:  

STANDARD

Example:  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and</td>
<td>0x17, g8, g2</td>
<td>g2 ← g8 AND 0x17</td>
</tr>
<tr>
<td>andnot</td>
<td>andnot</td>
<td>r3, r12, r9</td>
<td>r9 ← r12 AND NOT r3</td>
</tr>
</tbody>
</table>

Opcode:  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and</td>
<td>581</td>
<td>REG</td>
</tr>
<tr>
<td>andnot</td>
<td>andnot</td>
<td>582</td>
<td>REG</td>
</tr>
</tbody>
</table>

See Also:  

nand, nor, not, notand, notor, or, ornot, xnor, xor
atadd

Mnemonic: atadd Atomic Add

Format: atadd src/dst, src, dst
reg reg/lit reg addr

Description: Adds the src value (full word) to the value in the memory location specified with the src/dst operand. The initial value from memory is stored in dst.

The read and write of memory are done atomically (i.e., other processors are prevented from accessing the word of memory specified with the src/dst operand until the operation has been completed).

The memory location in src/dst is the address of the first byte (least significant byte) of the word. The address is automatically aligned to a word boundary.

Action: tempa ← src/dst and not (3); # force alignment to word boundary
temp ← atomic_read (tempa);
atomic_write (tempa) ← temp + src;
dst ← temp;

Faults: STANDARD

Example: atadd r8, r2, r11 # r8 ← r2 + address r8,
# where r8 specifies the
# address of a word in
# memory; r11 ← initial
# value stored at address
# r8 in memory

Opcode: atadd 612 REG

See Also: atmod
Mnemonics: atanr Arctangent Real
atanrl Arctangent Long Real

Format: atanr* srcl, src2, dst
freg/flit freg/flit freg

Description: Calculates the arctangent of the quotient of src2/srcl and stores the result in dst. The result is returned in radians and is in the range of \(-\pi\) to \(+\pi\), inclusive. The sign of the result is always the sign of src2.

For the atanrl instruction, if the srcl, src2, or dst operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

These instructions are commonly used as part of an algorithm to convert rectangular coordinates to polar coordinates. They can also be used to implement the FORTRAN intrinsic functions ATAN and ATAN2. If srcl is the floating-point literal value +1.0, then these instructions return a result in the range of \(-\pi/2\) to \(+\pi/2\).

The following table gives the range of results for various values of src2 and srcl, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>Src1</th>
<th>-∞</th>
<th>-F</th>
<th>-0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>-3n/4</td>
<td>-n/2</td>
<td>-n/2</td>
<td>-n/2</td>
<td>-n/2</td>
<td>-n/2</td>
<td>NaN</td>
</tr>
<tr>
<td>-F</td>
<td>-n</td>
<td>-n to -n/2</td>
<td>-n/2</td>
<td>-n/2</td>
<td>-n/2 to -0</td>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>-0</td>
<td>-n</td>
<td>-n</td>
<td>-n</td>
<td>-0</td>
<td>-0</td>
<td>-0</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>+n</td>
<td>+n</td>
<td>+n</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>+n</td>
<td>+n to +n/2</td>
<td>+n/2</td>
<td>+n/2</td>
<td>+n/2 to +0</td>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>+3n/4</td>
<td>+n/2</td>
<td>+n/2</td>
<td>+n/2</td>
<td>+n/2</td>
<td>+n/2</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Notes:
F Means finite-real number.
atanr, atanrl

Action:  \( dst \leftarrow \arctan (src2/src1); \)

Faults:  

- STANDARD
  - Refer to the discussion of faults at the beginning of this chapter.

- Floating Reserved Encoding
  - One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- Floating Underflow
  - Result is too small for destination format.

- Floating Invalid Operation
  - One or more operands are an SNaN value.

- Floating Inexact
  - Result cannot be represented exactly in destination format.

Example:  

\[
\begin{align*}
\text{atanrl } & \ g8, \ g10, \ fp3 \ \# \ \text{fp3} \leftarrow \\
& \ g0, \ g0 \ \# \ g0,g1 \leftarrow \arctan (g0,g1)
\end{align*}
\]

Opcode:  

\[
\begin{align*}
\text{atanr} & \quad 680 \quad \text{REG} \\
\text{atanrl} & \quad 690 \quad \text{REG}
\end{align*}
\]

See Also:  tanr
Mnemonic: atmod  Atomic Modify

Format: atmod src, mask, src/dst
         reg    reg/lit    reg
         addr

Description: Copies the src/dst value into the memory location specified in src. The bits set in the mask operand select the bits to be modified in memory. The initial value from memory is stored in src/dst.

The read and write of memory are done atomically (i.e., other processors are prevented from accessing the word of memory specified with the src/dst operand until the operation has been completed).

The memory location in src is the address of the first byte (least significant byte) of the word to be modified. The address is automatically aligned to a word boundary.

Action: tempa ← src and not (3);  # force alignment to word boundary
temp ← atomic_read (tempa);
atOMIC_write (tempa) ← (src/dst and mask)
         or (temp and not (mask));
src/dst ← temp;

Faults: STANDARD

Example: atmod g5, g7, g10  # g5 ← g5 masked by g7,
         # where g5 specifies the
         # address of a word in
         # memory;
         # g10 ← initial value
         # stored at address g5
         # in memory.

Opcode: atmod  610  REG

See Also: atadd
### bal, balx

**Mnemonic:**
- `bal` : Branch And Link
- `balx` : Branch And Link Extended

**Format:**
- `bal` : `targ`  
  `disp`
- `balx` : `targ, dst`
  `mem reg`

**Description:**
Stores the address of the next instruction (the instruction following the `bal` or `balx` instruction) and branches to the instruction specified with the `targ` operand.

With the `bal` instruction, the address of the next instruction is stored in register g14. The `targ` operand can be either a label or an absolute address that specifies the IP of the target instruction. This value can be no farther than $-2^{23}$ to $(2^{23} - 4)$ from the current IP.

The `balx` instruction performs almost the same operation as the `bal` instruction except that the target instruction can be farther than $-2^{23}$ to $(2^{23} - 4)$ from the current IP. With the `balx` instruction, the address of the next instruction is stored in `dst`. The `targ` operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. Here, the "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect branching can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.

**Note**
At the machine level, the `bal` instruction uses the CTRL instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by `displacement` in the following action statement for the `bal` instruction), which can range from $-2^{21}$ to $(2^{21} - 1)$. To determine the IP of the target instruction, the processor converts this `displacement` value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.
bal, balx

To allow labels or absolute addresses to be used in the assembly-language version of the bal instruction, the Intel 80960KB Assembler performs the following calculation to convert the targ value in an assembly-language instruction to the displacement value required by the machine instruction format:

\[ \text{displacement} = \frac{\text{targ}}{4} - \text{IP} \]

For further information about the CTRL instruction format, refer to Appendix B.

**Action:**

- **bal:**  \( \text{G14} \leftarrow \text{IP} + 4; \) # destination next IP is always g14
  \( \text{IP} \leftarrow \text{IP} + \text{targ}; \) # resume execution at the new IP

- **balx:**  \( \text{dst} \leftarrow \text{IP} + \text{inst length}; \) # instruction length
  # is 4 or 8 bytes
  \( \text{IP} \leftarrow \text{targ}; \) # resume execution at the new IP

**Faults:**  STANDARD

**Example:**

- **bal xyz:**  \# IP \( \leftarrow \text{xyz}; \)

- **balx (g2), g4:**  \# IP \( \leftarrow (\text{g2}); \)
  # address of return instruction
  # is stored in g4; example of
  # indirect addressing.

**Opcode:**

- **bal:**  0B  CTRL
- **balx:**  85  MEM

**See Also:** b, bx
b, bx

Mnemonic:

<table>
<thead>
<tr>
<th>b</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>bx</td>
<td>Branch Extended</td>
</tr>
</tbody>
</table>

Format:

<table>
<thead>
<tr>
<th>b</th>
<th>targ</th>
</tr>
</thead>
<tbody>
<tr>
<td>bx</td>
<td>targ mem</td>
</tr>
</tbody>
</table>

Description: Branches to the instruction specified with the targ operand.

With the b instruction, the targ operand can be either a label or an absolute address that specifies the IP of the target instruction. This value can be no farther than $-2^{23}$ to $(2^{23} - 4)$ from the current IP.

The bx instruction performs the same operation as the b instruction except that the target instruction can be farther than $-2^{23}$ to $(2^{23} - 4)$ from the current IP. With the bx instruction, the targ operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. Here, the "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect branching can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.

Note

At the machine level, the b instruction uses the CTRL instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by displacement in the following action statement for the b instruction), which can range from $-2^{21}$ to $(2^{21} - 1)$. To determine the IP of the target instruction, the processor converts this displacement value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

To allow labels or absolute addresses to be used in the assembly-language version of the b instruction, the Intel 80960KB Assembler performs the following calculation to convert the targ value in an assembly-language instruction to the displacement value required by the machine instruction format:

$$displacement = (targ/4) - IP$$

For further information about the CTRL instruction format, refer to Appendix B.
b, bx

**Action:**

- **b:** IP ← IP + displacement; # resume execution at the new IP
- **bx:** IP ← targ; # resume execution at the new IP

**Faults:**

STANDARD

**Example:**

- b xyz  # IP ← xyz;
- bx 1332 (ip)  # IP ← IP + 1332;
  # this example uses ip-relative addressing.

**Opcode:**

- b 08 CTRL
- bx 84 MEM

**See Also:**

bal, balx, BRANCH IF, COMPARE INTEGER AND BRANCH, COMPARE ORDINAL AND BRANCH
**bbc, bbs**

**Mnemonic:**
- **bbc** Check Bit and Branch If Clear
- **bbs** Check Bit and Branch If Set

**Format:**
```
bb*                bitpos,          src,            targ
  reg/lit          reg
```

**Description:** Checks the bit in `src` (designated by `bitpos`) and sets the condition code in the arithmetic controls according to the value found. The processor then performs a conditional branch based on the value of the condition code.

For the **bbc** instruction, if the selected bit in `src` is clear, the processor sets the condition code to `010_2` and branches to the instruction specified with the `targ` operand; otherwise, it sets the condition code to `000_2` and goes to the next instruction.

For the **bbs** instruction, if the selected bit is set, the processor sets the condition code to `010_2` and branches to `targ`; otherwise, it sets the condition code to `000_2` and goes to the next instruction.

When using the Intel 80960KB Assembler, the `targ` operand can be either a label or an absolute address that is no farther than \(-2^{12}\) to \((2^{12} - 4)\) from the current IP.

**Note**

At the machine level, the **bbc** and **bbs** instructions use the COBR instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by `displacement` in the following action statement), which can range from \(-2^{10}\) to \((2^{10} - 1)\). To determine the IP of the target instruction, the processor converts this `displacement` value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the IP of the next instruction.

To allow labels or absolute addresses to be used in the assembly-language versions of the **bbc** and **bbs** instructions, the Intel 80960KB Assembler performs the following calculation to convert the `targ` value in an assembly-language instruction to the `displacement` value required by the machine instruction format:

```
displacement = (targ/4) - (IP + 4)
```

For further information about the COBR instruction format, refer to Appendix B.
Action:

bbc:

if (src and $2^{(bitpos \mod 32)} = 0$
then AC.cc ← 2#010#;
    IP ← IP + 4 + (displacement * 4);
    # resume execution at the new IP
else AC.cc ← 2#000#;
    IP ← IP + 4; # resume execution at the next IP
end if;

bbs:

if (src and $2^{(bitpos \mod 32)} = 1$
then AC.cc ← 2#010#;
    IP ← IP + 4 + (displacement * 4);
    # resume execution at the new IP
else AC.cc ← 2#000#;
    IP ← IP + 4; # resume execution at the next IP
end if;

Faults:

STANDARD

Example:

# assume bit 10 of r6 is clear
bbc 10, r6, xyz  # bit 10 of r6 is checked
    # and found clear;
    # AC.cc ← 2#010#
    # IP ← xyz;

Opcode:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
<th>COBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbc</td>
<td>30</td>
<td>COBR</td>
</tr>
<tr>
<td>bbs</td>
<td>37</td>
<td>COBR</td>
</tr>
</tbody>
</table>

See Also:

chkbit
BRANCH IF

Mnemonics:  
be   Branch If Equal  
bne  Branch If Not Equal  
bl   Branch If Less  
ble  Branch If Less Or Equal  
bg   Branch If Greater  
bge  Branch If Greater Or Equal  
bo   Branch If Ordered  
bno  Branch If Unordered

Format:  
b*  
targ  
disp

Description:  Branches to a new instruction according to the state of the condition code in the arithmetic controls.

For all branch-if instructions except the bno instruction, the processor branches to the instruction specified with the targ operand, if the logical AND of the condition code and the mask-part of the opcode is not zero. Otherwise, it goes to the next instruction.

For the bno instruction, the processor branches to the instruction specified with targ, if the logical AND of the condition code and the mask-part of the opcode is zero. Otherwise, it goes to the next instruction.

When using the Intel 80960KB Assembler, the targ operand can be either a label or an absolute address that specifies the IP of the target instruction. This value can be no farther than \(-2^{23}\) to \((2^{23} - 4)\) from the current IP.

Note

At the machine level, the branch-if instructions use the CTRL instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by displacement in the following action statements), which can range from \(-2^{21}\) to \((2^{21} - 1)\). To determine the IP of the target instruction, the processor converts this displacement value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.
To allow labels or absolute addresses to be used in the assembly-language version of the branch-if instructions, the Intel 80960KB Assembler performs the following calculation to convert the `targ` value in an assembly-language instruction to the `displacement` value required by the machine instruction format:

\[
\text{displacement} = \frac{\text{targ}}{4} - \text{IP}
\]

For further information about the CTRL instruction format, refer to Appendix B.

The following table shows the condition-code mask for each instruction:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mask</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bno</code></td>
<td>000</td>
<td>Unordered</td>
</tr>
<tr>
<td><code>bg</code></td>
<td>001</td>
<td>Greater</td>
</tr>
<tr>
<td><code>be</code></td>
<td>010</td>
<td>Equal</td>
</tr>
<tr>
<td><code>bge</code></td>
<td>011</td>
<td>Greater or equal</td>
</tr>
<tr>
<td><code>bl</code></td>
<td>100</td>
<td>Less</td>
</tr>
<tr>
<td><code>bne</code></td>
<td>101</td>
<td>Not equal</td>
</tr>
<tr>
<td><code>ble</code></td>
<td>110</td>
<td>Less or equal</td>
</tr>
<tr>
<td><code>bo</code></td>
<td>111</td>
<td>Ordered</td>
</tr>
</tbody>
</table>

For the `bno` instruction (unordered), the branch is taken if the condition code is equal to `000_2`.

The mask is in bits 0-2 of the opcode.

**Action:**

For All Instructions Except `bno`:

\[
\text{if (mask and AC.cc) \neq 2#000#} \\
\quad \text{then IP ← IP + displacement}; \quad \# \text{resume execution at new IP} \\
\quad \text{end if;}
\]

`bno`:

\[
\text{if AC.cc = 2#000#} \\
\quad \text{then IP ← IP + displacement}; \quad \# \text{resume execution at new IP} \\
\quad \text{end if;}
\]
BRANCH IF

Faults: STANDARD

Example: # assume AC.cc AND 2#100# are ≠ 0
bl xyz # IP ← xyz;

Opcode:
- be 12 CTRL
- bne 15 CTRL
- bl 14 CTRL
- ble 16 CTRL
- bg 11 CTRL
- bge 13 CTRL
- bo 17 CTRL
- bno 10 CTRL

See Also: b, bx
**call**

**Mnemonic:** call

**Format:** call targ

**Description:** Calls a new procedure. The processor performs a local call operation as described in Chapter 4 in the section titled "Local Calls." As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. The processor then goes to the instruction specified with the targ argument and begins execution of the new procedure.

When using the Intel 80960KB Assembler, the targ operand can be either a label or an absolute address that specifies the IP of the first instruction in the called procedure. This value can be no farther than $-2^{23}$ to $(2^{23} - 4)$ from the current IP.

**Note**

At the machine level, the call instruction uses the CTRL instruction format. With this format, the first instruction of the called procedure is specified by means of a word-displacement (represented by displacement in the following action statement), which can range from $-2^{21}$ to $(2^{21} - 1)$. To determine the IP of the target instruction, the processor converts this displacement value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

To allow labels or absolute addresses to be used in the assembly-language version of the call instruction, the Intel 80960KB Assembler performs the following calculation to convert the targ value in an assembly-language instruction to the displacement value required by the machine instruction format:

$$displacement = (targ/4) - IP$$

For further information about the CTRL instruction format, refer to Appendix B.
call

**Action:**
wait for any uncompleted instructions to finish;
temp ← (SP + 63) and not (63); # round to next boundary
RIP ← IP;
if register_set_available
   then allocate as new frame;
   else save a register_set in memory at its FP;
      allocate as new frame;
# local register references now refer to new frame
IP ← IP + displacement;
PFP ← FP;
FP ← temp;
SP ← temp + 64;

**Faults:**
STANDARD

**Example:**
call xyz  # IP ← xyz

**Opcode:**
call 09  CTRL

**See Also:**
bal, calls, callx
calls

**Mnemonic:** calls  Call System

**Format:**

```
calls  targ  reg/lit
```

**Description:**

Calls a system procedure. The `targ` operand gives the number of the procedure being called.

For this instruction, the processor performs the system call operation described in Chapter 4 in the section titled "System Calls." The `targ` operand provides an index to an entry in the system procedure table. From this entry, the processor gets the IP of the called procedure.

The procedure called can be either a local procedure or a supervisor procedure, depending on the entry type in the procedure table. If it is a supervisor procedure, the processor also switches to supervisor mode (if it is not already in this mode).

As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. If the processor switches to the supervisor mode, the new stack frame is created on the supervisor stack.
calls

Action:  
if $targ > 259$ then raise Protection Length Fault;  
wait for any uncompleted instructions to finish;  
$temp_p.e \leftarrow$ memory ($SPT, 48 + (4 \times targ)$);  
# $SPT$ is pointer to system procedure table from IMI  
$RIP \leftarrow IP$;  
$IP \leftarrow temp_p.e.address$; if $(temp_p.e.type = local)$ or  
execution_mode = supervisor  
then temp $\leftarrow (SP + 63)$ and not(63);  
$tempRRR \leftarrow 2\#000#$;  
else temp $\leftarrow$ memory ($SPTSS, 12$);  
# supervisor call  
$tempRRR \leftarrow 2\#01T#$;  
# $T$ is process_controls.$T$  
execution_mode $\leftarrow$ supervisor;  
process_controls.$T \leftarrow temp.T$;  
endif;  
if frame_available  
then allocate as new frame;  
else save a frame in memory at its $FP$;  
allocate as new frame;  
# local register references now refer to new frame  
endif;  
$PFP \leftarrow FP$;  
$L0.RRR \leftarrow tempRRR$;  
$FP \leftarrow temp$;  
$SP \leftarrow temp + 64$;

Faults:  
STANDARD

Example:  
calls r12  
# $IP \leftarrow$ value obtained from  
# procedure table for procedure  
# number given in r12

Opcode:  
calls 660  
REG

See Also:  
bal, call, callx
Mnemonic: callx  Call Extended
Format: callx  \texttt{targ}  \texttt{mem}

Description: Calls a new procedure. The processor performs a local call operation as described in Chapter 4 in the section titled "Local Calls." As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. The processor then goes to the instruction specified with the \texttt{targ} argument and begins execution of the new procedure.

This instruction performs the same operation as the call instruction except that the target instruction can be farther than $-2^{23}$ to $(2^{23} - 4)$ from the current IP.

The \texttt{targ} operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect calls can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.

Action:
wait for any uncompleted instructions to finish;
temp $\leftarrow (SP + 63)$ \texttt{and not} (63); # round to next boundary
RIP $\leftarrow$ IP;
if register\_set\_available
  then allocate as new frame;
  else save a register\_set in memory at its FP;
  allocate as new frame;
# local register references now refer to new frame
endif;
IP $\leftarrow$ \texttt{targ};
PFP $\leftarrow$ FP;
FP $\leftarrow$ temp;
SP $\leftarrow$ temp + 64;
callx

Faults: STANDARD

Example: callx (g5) # IP ← (g5), where the address
# in g5 is the address of the new
# procedure

Opcode: bal, callx 86 MEM

See Also: call, calls
Mnemonic: chkbit

Format: chkbit bitpos, src reg/lit reg/lit

Description: Checks the bit in src designated by bitpos and sets the condition code according to the value found. If the bit is set, the condition code is set to 010₂; if the bit is clear, the condition code is set to 000₂.

Action: if (src and 2^(bitpos mod 32)) = 0 then AC.cc ← 2#000#; else AC.cc ← 2#010#; end if;

Faults: STANDARD

Example: chkbit 13, g8 # checks bit 13 in g8

Opcode: chkbit 5AE REG

See Also: alterbit, clrbit, notbit, setbit
**classr, classrl**

Mnemonic:  
- **classr**: Classify Real  
- **classrl**: Classify Long Real

Format:  
- `classr* src freg/flit`

Description:  
Checks the classification of the real number in `src` and stores the class in arithmetic-status bits (3 through 6) of the arithmetic controls.

For the **classrl** instruction, if the `src` operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the setting of the arithmetic-status bits depending on the classification of the operand.

<table>
<thead>
<tr>
<th>AStatus</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>s000</td>
<td>Zero</td>
</tr>
<tr>
<td>s001</td>
<td>Denormalized number</td>
</tr>
<tr>
<td>s010</td>
<td>Normal finite number</td>
</tr>
<tr>
<td>s011</td>
<td>Infinity</td>
</tr>
<tr>
<td>s100</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>s101</td>
<td>Signaling NaN</td>
</tr>
<tr>
<td>s110</td>
<td>Reserved operand</td>
</tr>
</tbody>
</table>

The "s" bit is set to the sign of the `src` operand.

Refer to Chapter 7 for a discussion of the different real number classifications.
Action:  
\[ s \leftarrow \text{sign}_\text{of}(src) \]
\[ \text{if} \ src = 0 \]
\[ \quad \text{then} \ \text{arithmetic}\_\text{status} \leftarrow \text{s00}0; \]
\[ \text{elseif} \ src = \text{denormalized} \]
\[ \quad \text{then} \ \text{arithmetic}\_\text{status} \leftarrow \text{s00}1; \]
\[ \text{elseif} \ src = \text{normal finite} \]
\[ \quad \text{then} \ \text{arithmetic}\_\text{status} \leftarrow \text{s01}0; \]
\[ \text{elseif} \ src = \infty \]
\[ \quad \text{then} \ \text{arithmetic}\_\text{status} \leftarrow \text{s01}1; \]
\[ \text{elseif} \ src = \text{QNaN} \]
\[ \quad \text{then} \ \text{arithmetic}\_\text{status} \leftarrow \text{s10}0; \]
\[ \text{elseif} \ src = \text{SNaN} \]
\[ \quad \text{then} \ \text{arithmetic}\_\text{status} \leftarrow \text{s10}1; \]
\[ \text{elseif} \ src = \text{reserved operand} \]
\[ \quad \text{then} \ \text{arithmetic}\_\text{status} \leftarrow \text{s11}0; \]
\[ \text{end if} \]

Faults:  
STANDARD  
Refer to the discussion of faults at the beginning of this chapter.

None of the floating-point exceptions can be raised.

Example:  
classrl g12  # classifies long real in g12, g13

Opcode:  
classr   68F   REG
classrl  69F   REG
clrbit

Mnemonic: clrbit Clear Bit

Format: clrbit bitpos, src, dst
        reg/lit reg/lit reg

Description: Copies the src value to dst with one bit cleared. The bitpos operand specifies the bit to be cleared.

Action: \[ dst \leftarrow src \text{ and } \text{not}(2^{\text{bitpos mod 32}}); \]

Faults: STANDARD

Example: clrbit 23, g3, g6 # g6 \leftarrow g3 with bit 23 # cleared

Opcode: clrbit 58C REG

See Also: alterbit, chkbit, notbit, setbit
Mnemonics: 
- cmpi Compare Integer
- cmpo Compare Ordinal

Format: 
```
cmp* src1, reg/lit src2 reg/lit
```

Description: 
Compares the `src2` and `src1` values and sets the condition code according to the results of the comparison. The following table shows the setting of the condition code for the three possible results of the comparison.

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td><code>src1 &lt; src2</code></td>
</tr>
<tr>
<td>010</td>
<td><code>src1 = src2</code></td>
</tr>
<tr>
<td>001</td>
<td><code>src1 &gt; src2</code></td>
</tr>
</tbody>
</table>

The `cmpi` instruction followed by one of the branch-if instructions is equivalent to one of the compare-integer-and-branch instructions. The latter method of comparing and branching produces more compact code; however, the former method can result in faster running code because it takes advantage of the processor's pipelined architecture. The same is true for the `cmpo` instruction and the compare-ordinal-and-branch instructions.

Action:
```
if src1 < src2 then AC.cc ← 2#100#
elseif src1 = src2 then AC.cc ← 2#010#
else AC.cc ← 2#001#
end if;
```

Faults: STANDARD

Example: 
```
cmpo 0x10, r9  # compare values in r9 and 0x10
              # and set condition code
```

Opcode: 
- cmpi 5A1 REG
- cmpo 5A0 REG

See Also: cmpibe, cmpr, cmpdeci, cmpdeco
**cmpdeci, cmpdeco**

**Mnemonics:**
- `cmpdeci` Compare and Decrement Integer
- `cmpdeco` Compare and Decrement Ordinal

**Format:**
```
cmpdeci* src1, src2, dst
    reg/lit    reg/lit    reg
```

**Description:**
Compares the `src2` and `src1` values and sets the condition code according to the results of the comparison. The `src2` operand is then decremented by one and the result is stored in `dst`.

The following table shows the setting of the condition code for the three possible results of the comparison.

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td><code>src1 &lt; src2</code></td>
</tr>
<tr>
<td>010</td>
<td><code>src1 = src2</code></td>
</tr>
<tr>
<td>001</td>
<td><code>src1 &gt; src2</code></td>
</tr>
</tbody>
</table>

These instructions are intended for use in ending iterative loops. For the `cmpdeci` instruction, integer overflow is ignored to allow looping down through the minimum integer values.

**Action:**
```
if src1 < src2 then AC.cc ← 2#100#;
elseif src1 = src2 then AC.cc ← 2#010#;
elseif src1 > src2 then AC.cc ← 2#001#;
end if;
dst ← src2 - 1; #overflow suppressed for cmpdeci
    # instruction
```

**Faults:**
STANDARD

**Example:**
```
cmpdeci 12, g7, gl  # g7 and 12 are compared;
    # gl ← g7 - 1
```

**Opcode:**
- `cmpdeci` 5A7 REG
- `cmpdeco` 5A6 REG

**See Also:**
- cmpinco, cmpo
cmpinci, cmpinco

Mnemonics:  
cmpinci    Compare and Increment Integer  
cmpinco    Compare and Increment Ordinal

Format:  
cmpinc*    src1,    src2,    dst  
reg/lit    reg/lit  reg

Description:  
Compares the src2 and src1 values and sets the condition code according to 
the results of the comparison. The src2 operand is then incremented by one 
and the result is stored in dst.

The following table shows the setting of the condition code for the three 
possible results of the comparison.

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>src1 &lt; src2</td>
</tr>
<tr>
<td>010</td>
<td>src1 = src2</td>
</tr>
<tr>
<td>001</td>
<td>src1 &gt; src2</td>
</tr>
</tbody>
</table>

These instructions are intended for use in ending iterative loops. For the 
cmpinci instruction, integer overflow is ignored to allow looping up through 
the maximum integer values.

Action:  
if src1 < src2 then AC.cc ← 2#100#;  
elseif src1 = src2 then AC.cc ← 2#010#;  
elseif src1 > src2 then AC.cc ← 2#001#;  
end if;  
dst ← src2 + 1;  # overflow suppressed for cmpinci  
            # instruction

Faults:  
STANDARD

Example:  
cmpinco r8, g2, g9    # g2 and r8 are compared;  
                # g9 ← g2 + 1

Opcode:  
cmpinci    5A5    REG  
cmpinco    5A4    REG

See Also:  
cmpdeco, cmpo
**cmpor, cmporl**

**Mnemonics:**
- `cmpor`  Compare Ordered Real
- `cmporl`  Compare Ordered Long Real

**Format:**

```
cmpor*  src1,  src2
  freg/flit  freg/flit
```

**Description:**

Compares the `src2` and `src1` values and sets the condition code according to the results of the comparison.

For the `cmporl` instruction, if the `src1` or `src2` operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the setting of the condition code for the four possible results of the comparison.

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>src1 &lt; src2</td>
</tr>
<tr>
<td>010</td>
<td>src1 = src2</td>
</tr>
<tr>
<td>001</td>
<td>src1 &gt; src2</td>
</tr>
<tr>
<td>000</td>
<td>if either src1 or src2 is a NaN</td>
</tr>
</tbody>
</table>

The algorithm for these instructions checks the classification of the operands. If either is in the NaN class, the condition code is set to 0000 and a floating invalid-operation exception is raised. The `cmpor` and `cmporl` instructions operate the same as the `cmpr` and `cmprl` instructions, except that the latter instructions do not signal an exception if a NaN value is detected.

If a floating-reserved-encoding fault occurs, the condition code results are undefined.

**Action:**

```
if src1 < src2
    then AC.cc ← 2#100#
elseif src1 = src2
    then AC.cc ← 2#010#
elseif src1 > src2
    then AC.cc ← 2#001#
else AC.cc ← 2#000#; # indicates one number is a NaN
raise floating invalid operation fault
end if;
```
Faults:  
- **STANDARD**  
  Refer to the discussion of faults at the beginning of this chapter.
- **Floating Reserved Encoding**  
  One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exception can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- **Floating Invalid Operation**  
  One or more operands are a NaN value.

Example:  
```
cmporl g6, g12  # compare value in g12,g13
                # with value in g6,g7
```

Opcode:  
- `cmpor`  
  684  
  REG
- `cmporl`  
  694  
  REG

See Also:  
- `cmp, cmpi, BRANCH IF`
**cmpr, cmprl**

**Mnemonics:**
- cmpr: Compare Real
- cmprl: Compare Long Real

**Format:**
- `cmpr* src1, src2 freg/flit freg/flit`

**Description:**
Compares the `src2` and `src1` values and sets the condition code according to the results of the comparison. For the `cmprl` instruction, if the `src1` or `src2` operand references a global or local register, this register is the first (lowest numbered) of two successive registers.

The following table shows the setting of the condition code for the four possible results of the comparison.

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td><code>src1 &lt; src2</code></td>
</tr>
<tr>
<td>010</td>
<td><code>src1 = src2</code></td>
</tr>
<tr>
<td>001</td>
<td><code>src1 &gt; src2</code></td>
</tr>
<tr>
<td>000</td>
<td>if either <code>src1</code> or <code>src2</code> is a NaN</td>
</tr>
</tbody>
</table>

The algorithm for these instructions checks the classification of the operands. If either is in the NaN class, the condition code is set to 000₂, but no fault is raised. The `cmpr` and `cmprl` instructions operate the same as the `cmpor` and `cmporl` instructions, except that the latter instructions raise an invalid-operand exception if a NaN value is detected.

If a floating-reserved-encoding fault occurs, the condition code results are undefined.

**Action:**

```plaintext
if src1 < src2
    then AC.cc ← 2#100#;
else if src1 = src2
    then AC.cc ← 2#010#;
else if src1 > src2
    then AC.cc ← 2#001#;
else AC.cc ← 2#000#; # indicates one number is a NaN
end if;
```
Faults: STANDARD

Floating Reserved Encoding
One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exception can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Invalid Operation
One or more operands are an SNaN value.

Example: cmprl g2, g6  # compare values in g6,g7
          # and g2,g3

Opcode: cmpr 685 REG
        cmprl 695 REG

See Also: cmpor, cmpi, BRANCH IF
COMPARE AND BRANCH

Mnemonics:
- cmpibe: Compare Integer And Branch If Equal
- cmpibne: Compare Integer And Branch If Not Equal
- cmpibl: Compare Integer And Branch If Less
- cmpible: Compare Integer And Branch If Less Or Equal
- cmpibg: Compare Integer And Branch If Greater
- cmpibge: Compare Integer And Branch If Greater Or Equal
- cmpibo: Compare Integer And Branch If Ordered
- cmpibno: Compare Integer And Branch If Unordered
- cmpobe: Compare Ordinal And Branch If Equal
- cmpobne: Compare Ordinal And Branch If Not Equal
- cmpobl: Compare Ordinal And Branch If Less
- cmpoble: Compare Ordinal And Branch If Less Or Equal
- cmpobg: Compare Ordinal And Branch If Greater
- cmpobge: Compare Ordinal And Branch If Greater Or Equal

Format:
- cmpib*: src1, src2, targ
  reg/lit reg
- cmpob*: src1, src2, targ
  reg/lit reg disp

Description: Compares the src2 and src1 values and sets the condition code according to the results of the comparison. If the logical AND of the condition code and the mask-part of the opcode is not zero, the processor branches to the instruction specified with the targ operand; otherwise, the processor goes to the next instruction.

When using the Intel 80960KB Assembler, the targ operand can be either a label or an absolute address that is no farther than $-2^{10}$ to $(2^{10} - 4)$ from the current IP.

Note

At the machine level, the compare-and-branch instructions use the COBR instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by displacement in the following action statement), which can range from $-2^{10}$ to $(2^{10} - 1)$. To determine the IP of the target instruction, the processor converts this displacement value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the IP of the next instruction.
COMPARE AND BRANCH

To allow labels or absolute addresses to be used in the assembly-language versions of these instructions, the Intel 80960KB Assembler performs the following calculation to convert the \( \text{targ} \) value in an assembly-language instruction to the \( \text{displacement} \) value required by the machine instruction format:

\[
\text{displacement} = (\text{targ}/4) - (\text{IP} + 4)
\]

For further information about the COBR instruction format, refer to Appendix B.

The following table shows the condition-code mask for each instruction:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mask</th>
<th>Branch Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpibno</td>
<td>000</td>
<td>No Condition</td>
</tr>
<tr>
<td>cmpibg</td>
<td>001</td>
<td>( \text{src1} &gt; \text{src2} )</td>
</tr>
<tr>
<td>cmpibe</td>
<td>010</td>
<td>( \text{src1} = \text{src2} )</td>
</tr>
<tr>
<td>cmpibge</td>
<td>011</td>
<td>( \text{src1} \geq \text{src2} )</td>
</tr>
<tr>
<td>cmpibl</td>
<td>100</td>
<td>( \text{src1} &lt; \text{src2} )</td>
</tr>
<tr>
<td>cmpibne</td>
<td>101</td>
<td>( \text{src1} \neq \text{src2} )</td>
</tr>
<tr>
<td>cmpible</td>
<td>110</td>
<td>( \text{src1} \leq \text{src2} )</td>
</tr>
<tr>
<td>cmpibo</td>
<td>111</td>
<td>Any Condition</td>
</tr>
<tr>
<td>cmpobg</td>
<td>001</td>
<td>( \text{src1} &gt; \text{src2} )</td>
</tr>
<tr>
<td>cmpobe</td>
<td>010</td>
<td>( \text{src1} = \text{src2} )</td>
</tr>
<tr>
<td>cmpobge</td>
<td>011</td>
<td>( \text{src1} \geq \text{src2} )</td>
</tr>
<tr>
<td>cmpobl</td>
<td>100</td>
<td>( \text{src1} &lt; \text{src2} )</td>
</tr>
<tr>
<td>cmpobne</td>
<td>101</td>
<td>( \text{src1} \neq \text{src2} )</td>
</tr>
<tr>
<td>cmpoble</td>
<td>110</td>
<td>( \text{src1} \leq \text{src2} )</td>
</tr>
</tbody>
</table>

The \text{cmpibo} instruction always branches; the \text{cmpibno} instruction never branches.

The functions that these instructions perform can be duplicated with a \text{cmpi} instruction followed by a branch-if instruction, as described in the description of the \text{cmpi} instruction in this chapter.
COMPARE AND BRANCH

Action:

if src1 < src2 then AC.cc ← 2#100#;
else if src1 = src2 then AC.cc ← 2#010#;
else AC.cc ← 2#001#;
end if;
if mask and AC.cc ≠ 2#000#
    then IP ← IP + 4 + (displacement * 4);
        # resume execution at the new IP
else IP ← IP + 4;
    # resume execution at the next IP
end if;

Faults: STANDARD

Example:

# assume g3 < g9
cmpibl g3, g9, xyz  # g9 is compared with g3;
                    # IP ← xyz.

# assume r7 ≥ 19
cmpobge r7, 19, xyz  # 19 is compared with r7
                    # IP ← xyz.

Opcode:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpibe</td>
<td>3A</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpibne</td>
<td>3D</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpibl</td>
<td>3C</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpible</td>
<td>3E</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpibg</td>
<td>39</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpibge</td>
<td>3B</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpiboe</td>
<td>3F</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpibno</td>
<td>38</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpobep</td>
<td>32</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpobne</td>
<td>35</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpobtl</td>
<td>34</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpoble</td>
<td>36</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpobge</td>
<td>31</td>
<td>COBR</td>
</tr>
<tr>
<td>cmpobge</td>
<td>33</td>
<td>COBR</td>
</tr>
</tbody>
</table>

See Also: BRANCH IF, cmpi
**Mnemonics:**
- `concmpi` Conditional Compare Integer
- `concmpo` Conditional Compare Ordinal

**Format:**
```concmp* src1, src2
reg/lit reg/lit```

**Description:**
Compares the `src2` and `src1` values if bit 2 of the condition code is not set. If the comparison is performed, the condition code is set according to the results of the comparison.

These instructions are provided to facilitate bounds checking by means of two-sided range comparisons (e.g., is A between B and C?). They are generally used after a compare instruction to test whether a value is inclusively between two other values.

The example below illustrates this application by testing whether the value in `g3` is between the values in `g5` and `g6`, where `g5` is assumed to be less than `g6`. First a comparison (`cmpo`) of `g3` and `g6` is performed. If `g3` is less than or equal to `g6` (i.e., condition code is either 010₂ or 001), a conditional comparison (`concmpo`) of `g3` and `g5` is then performed. If `g3` is greater than or equal to `g5` (indicating that `g3` is within the bounds of `g5` and `g6`), the condition code is set to 010₂; otherwise, it is set to 001₂.

**Action:**
```java
if (AC.cc and 2#100#) = 0 then
  if src1 ≤ src2
    then AC.cc ← 2#010;
    else AC.cc ← 2#001;
  endif;
endif;
```

**Faults:**
- STANDARD

**Example:**
```assembly
cmpo g6, g3  # compares g6 and g3 and sets # condition code
concmpo g5, g3  # if condition code is not # 2#1xx#, g5 is compared # with g3```

**Opcode:**
- `concmpi` 5A3  REG
- `concmpo` 5A2  REG

**See Also:**
- `cmpo`, `cmpi`
**cosr, cosrl**

Mnemonics:  
- **cosr**: Cosine Real  
- **cosrl**: Cosine Long Real

Format:  
- `cosr* src, freg/flit dst freg`

Description:  
Calculates the cosine of the value in `src` and stores the result in `dst`. The `src` value is an angle given in radians. The resulting `dst` value is in the range -1 to +1, inclusive.

For the `cosrl` instruction, if the `src` or `dst` operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the cosine of various classes of numbers with neither overflow nor underflow.

<table>
<thead>
<tr>
<th>Src</th>
<th>Dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>*</td>
</tr>
<tr>
<td>-F</td>
<td>-1 to +1</td>
</tr>
<tr>
<td>-0</td>
<td>+1</td>
</tr>
<tr>
<td>+0</td>
<td>+1</td>
</tr>
<tr>
<td>+F</td>
<td>-1 to +1</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Notes:  
- F  Means finite-real number  
- * Indicates floating invalid-operation exception

In the trigonometric instructions, the 80960KB uses a value for π with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. The section in Chapter 12 titled "Pi" gives this π value, along with some suggestions for representing this value in a program.

Action:  
`dst ← cosine (src);`
Faults: STANDARD  
Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding  
One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Invalid Operation  
The src operand is $\infty$.
One or more operands are an SNaN value.

Floating Inexact  
Result cannot be represented exactly in destination format.

Example:  
```
cosl r8, g2  # cosine of value in r8,r9 is  
# stored in g2,g3
```

Opcode:  
```
cosr 68D  REG

cosl 69D  REG
```

See Also:  
sinr, sinrl, tanr, tanrl
cpyrsre, cpysre

Mnemonics:  
cpysre  Copy Sign Real Extended  
cpyrsre  Copy Reversed Sign Real Extended

Format:  
cpy*  
src1,  src2,  dst  
freg/flit  freg/flit  freg

Description:  
Copies the absolute value of src1 into dst. For the cpysre instruction, the sign of src2 is copied to dst; for the cpyrsre instruction, the opposite of the sign of src2 is copied to dst.

If the src1, src2, or dst operand references a global or local register, this register is the first (lowest numbered) of three successive registers. Also, the number of this register must be a multiple of four (e.g., g0, g4, g8).

These instructions only operate on values in the extended-real format. The same operations can be performed on real- and long-real values using the setbit and clearbit instructions, or a combination of the chkbit and alterbit instructions.

Action:  
cpysre  if src2 is positive  
then  dst ← abs (src1)  
else  dst ← -abs (src1)

cpyrsre  if src2 is negative  
then  dst ← abs (src1)  
else  dst ← -abs (src1)

Faults:  
STANDARD  Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding  One or more operands is a denormalized value and the normalizing-mode bit in the arithmetic controls is set.

Example:  
cpysre  fp0, fp1, fp2  
# absolute value from fp0 is copied to  
# fp2; sign from fp1 is copied to fp2

Opcode:  
cpysre  6E2  REG  
cpyrsre  6E3  REG
Mnemonics:  
- `cvtilr` Convert Long Integer to Real  
- `cvtir` Convert Integer to Real

Format:  
`cvti*`  
```
src, dst
reg/lit freg
```

Description:  
Converts the integer in `src` to a real and stores the result in `dst`. For the `cvtilr` instruction, the `src` operand references the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

Converting an integer to long real format requires two instructions. First, the integer is converted to extended real format by using the `cvtir` or `cvtilr` instruction with a floating-point register as a destination. Then the `movrl` instruction is used to move the value from the floating-point register to two global or local registers, causing an explicit conversion to long real format. (Note that this conversion is always exact.) The example section below illustrates this conversion.

Action:  
\[dst \leftarrow \text{real}(src);\]

Faults:  
- STANDARD  
Refer to the discussion of faults at the beginning of this chapter.

The following floating-point exception can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- Floating Inexact  
Can only be signaled when converting an integer to real (32-bit) format

Example:  
```
# Conversion of an integer to a long real value
 cvtir g6, fp3
 movrl fp3, g8  # result stored in g8,g9
```

Opcode:  
```
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvtir</td>
<td>674</td>
<td>REG</td>
</tr>
<tr>
<td>cvtilr</td>
<td>675</td>
<td>REG</td>
</tr>
</tbody>
</table>
```

See Also:  
- `cvtri`, `movr`
**cvtri, cvtril, cvtzri, cvtzril**

**Mnemonics:**
- `cvtri` Convert Real To Integer
- `cvtril` Convert Real To Integer Long
- `cvtzri` Convert Truncated Real To Integer
- `cvtzril` Convert Truncated Real To Long Integer

**Format:**

```
cvtri*       src,       dst
from/flit   reg
```

**Description:** Converts the real value in `src` to an integer and stores the result in `dst`.

For the `cvtril` and `cvtzril` instructions, the `dst` operand references the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The nontruncated versions of these instructions round according to the current rounding mode in the Arithmetic Controls register. The truncated versions always round toward zero.

Converting a long real value to an integer requires two instructions. First, the long real value is converted to extended real format by using the `movrl` instruction with a floating-point register as a destination. (Note that this operation is always exact.) Then one of the convert real-to-integer instructions is used to move the value from the floating-point register to one or two global or local registers. The example section below illustrates this conversion.

If the magnitude of the result cannot be represented in the destination, an integer-overflow fault is raised, and the maximum positive or maximum negative value is stored in the destination (depending on whether the real value was positive or negative, respectively).

**Action:**

```
dst ← integer (src1);
# src1 is rounded to integer value
```
cvtri, cvtrl, cvtzri, cvtzril

Faults: STANDARD

Refer to the discussion of faults at the beginning of this chapter.

The following exception can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls register.

Integer Overflow

Result is too large for destination format.

Example:

# Conversion of long real value to an integer
movrl g4, fp2  # long-real source is
               # converted to extended-real
               # format and moved to fp2

  cvtril fp2, g12  # extended-real value is
                 # converted to long integer

Opcode:

cvtri  6C0  REG
cvtril 6C1  REG
cvtzri 6C2  REG
cvtzril 6C3  REG

See Also: cvtir, movr
**daddc**

**Mnemonic:** daddc  
Decimal Add With Carry

**Format:**

```
daddc src1, src2, dst  
reg    reg    reg
```

**Description:** Adds bits 0 through 3 of src2 and src1 and bit 1 of the condition code (used here as a carry bit). The result is stored in bits 0 through 3 of dst. If the addition results in a carry, bit 1 of the condition code is set. Bits 4 through 31 of src are copied to dst unchanged.

This instruction is intended to be used iteratively to add binary-coded-decimal (BCD) values in which the least-significant four bits of the operands represent the decimal numbers 0 to 9. The instruction assumes that the least significant 4 bits of both operands are valid BCD numbers. If these bits are not valid BCD numbers, the resulting value in dst is unpredictable.

**Action:**

- `# Let the value of the condition code be xCx.
  dst ← src2 + src1 + C;
  AC.cc ← 2#0C0#;
  # C is carry from addition of bits 0 through 4 of operands
  # Bits 4 - 31 of dst are same as bits 4 - 31 of src2`

**Faults:** STANDARD

**Example:**

```
daddc g5, g9, g10  # g10 ← g9 + g5 + Carry Bit,  
# where arithmetic is  
# carried out only on bits 0  
# through 3 of the operands
```

**Opcode:**

```
daddc 642 REG
```

**See Also:** dsubc, dmovt
divi, divo

Mnemonic:  

- divi: Divide Integer
- divo: Divide Ordinal

Format:  

\[
\text{div}^* \quad src1, \quad src2, \quad dst \\
\text{reg/lit} \quad \text{reg/lit} \quad \text{reg}
\]

Description:  
Divides the src2 value by the src1 value and stores the result in dst.

For the divi instruction, and integer-overflow fault can be signaled.

Action:  
\[dst \leftarrow src2 / src1;\]

Faults:  

- STANDARD: Refer to discussion of faults at the beginning of this chapter.
- Arithmetic Zero Divide: The src1 operand is 0.

The following fault condition can be raised with the divi instruction. Whether or not a fault is raised depends on the state of its associated mask bit in the arithmetic-controls register.

- Integer Overflow: Result is too large for destination format.

Example:  
\[\text{divo} \quad r3, \quad r8, \quad r13 \# \quad r13 \leftarrow r8/r3\]

Opcode:  

- divi: 74B REG
- divo: 70B REG

See Also:  
ediv, mulo
**divr, divrl**

**Mnemonic:**
- divr: Divide Real
- divrl: Divide Long Real

**Format:**
```
divr* src1, freg/flit
      src2, freg/flit
      dst  freg
```

**Description:** Divides the `src2` value by the `src1` value and stores the result in `dst`.

For the `divrl` instruction, if the `src1`, `src2`, or `dst` operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the source values is 0, ∞, or a NaN.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>Src2</th>
<th>-∞</th>
<th>-F</th>
<th>-0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>-∞</td>
<td>-∞</td>
<td>NaN</td>
</tr>
<tr>
<td>-F</td>
<td>+0</td>
<td>+F</td>
<td>**</td>
<td>**</td>
<td>-F</td>
<td>-0</td>
<td>NaN</td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td>+0</td>
<td>*</td>
<td>*</td>
<td>-0</td>
<td>-0</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>-0</td>
<td>-0</td>
<td>*</td>
<td>*</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>-0</td>
<td>-F</td>
<td>**</td>
<td>**</td>
<td>+F</td>
<td>+0</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
<td>-∞</td>
<td>-∞</td>
<td>+∞</td>
<td>+∞</td>
<td>*</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**Notes:**
- F Means finite-real number.
- * Indicates floating invalid-operation exception.
- ** Indicates floating zero-divide exception.

**Action:**
```
dst ← src2 / src1;
```
### Faults:

<table>
<thead>
<tr>
<th>Fault</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STANDARD</td>
<td>Refer to the discussion of faults at the beginning of this chapter.</td>
</tr>
<tr>
<td>Floating Reserved Encoding</td>
<td>One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.</td>
</tr>
</tbody>
</table>

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

| Floating Overflow | Result is too large for destination format.                                   |
| Floating Underflow| Result is too small for destination format.                                   |
| Floating Zero Divide | The src1 operand is 0 and the src2 operand is numeric and finite. |
| Floating Invalid Operation | Both source operands are 0 or both are \( \infty \). |
| Floating Inexact    | One or more operands are an SNaN value.                                       |
| Result cannot be represented exactly in destination format. |

### Example:

\[
\text{divr} \quad \text{divrl} \\
\text{divr} \quad g_0, g_0, fpl \# fpl \leftarrow g_0, g_1 / g_10, g_11 \\
\]

### Opcode:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>divr</td>
<td>78B</td>
<td>REG</td>
</tr>
<tr>
<td>divrl</td>
<td>79B</td>
<td>REG</td>
</tr>
</tbody>
</table>

### See Also:

ediv, mulr, mulrl
**dmovt**

**Mnemonic:** dmovt  
Decimal Move And Test

**Format:**

```
dmovt src, dst
reg reg
```

**Description:** Copies the src value into dst. The least-significant eight bits of the src value are tested to determine whether or not they constitute a valid ASCII decimal (00110000_2 .. 00111001_2), and the condition code is set accordingly. If the value is a valid ASCII decimal, the condition code is set to 000_2; otherwise, it is set to 010_2.

This instruction is intended to be used iteratively to validate decimal strings.

**Action:**

```
dst <-- src;
if src = 2#0011000# .. 2#00111001#
    then AC.cc <-- 2#000#;
else AC.cc <-- 2#010#;
end if;
```

**Faults:** STANDARD

**Example:**

```
dmovt g1, g6  # g6 <-- g1;
# g1 tested for decimal value
```

**Opcode:**

```
dmovt 644  REG
```

**See Also:** daddc, dsbdc
Mnemonic: dsubc

Decimal Subtract With Carry

Format: dsubc src1, reg src2, reg dst

Description: Subtracts bits 0 through 3 of src2 and src1 and bit 1 of the condition code (used here as a carry bit). The result is stored in bits 0 through 3 of dst. If the subtraction results in a carry, bit 1 of the condition code is set. Bits 4 through 31 of src are copied to dst unchanged.

This instruction is intended to be used iteratively to subtract binary-coded-decimal (BCD) values in which the least-significant four bits of the operands represent the decimal numbers 0 to 9. The instruction assumes that the least significant 4 bits of both operands are valid BCD numbers. If these bits are not valid BCD numbers, the resulting value in dst is unpredictable.

Action: # Let the value of the condition code be xCx.
dst ← src2 - src1 - 1 + C;
AC.cc ← 2#0C0#;
# C is carry from subtraction of bits 0 through 4 of operands
# Bits 4 - 31 of dst are same as bits 4 - 31 of src2

Faults: STANDARD

Example: dsubc r1, r2, r12 # r12 ← r2 - r1 - 1 + Carry
# Bit, where arithmetic is
# carried out only on bits 0
# through 3 of the operands

Opcode: dsubc 643 REG

See Also: daddc, dmovt
ediv

Mnemonic: ediv Extended Divide

Format: ediv src1, src2, dst
         reg/lit   reg/lit   reg

Description: Divides src2 by src1 and stores the result in dst. The src2 value is a long ordinal (i.e., 64 bits), which is contained in two adjacent registers. The src2 operand specifies the lower numbered register, which contains the least significant bits of the operand. The src2 operand must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ...). The src1 value is a normal ordinal (i.e., 32 bits).

The remainder is stored in the register designated by dst and the quotient is stored in the next highest numbered register. The dst operand must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ...).

This instruction performs ordinal arithmetic.

If this operation overflows (i.e., the quotient or remainder do not fit in 32-bits), no fault is raised and the result is undefined.

Action: dst ← (src2 - (src2 / src1) * src1); # remainder
dst + 1 ← (src2 / src1); # quotient

Faults: STANDARD, Arithmetic Integer Divide

Example: ediv g3, g4, g10  # g10 ← remainder of g4,g5/g3
         # g11 ← quotient of g4,g5/g3

Opcode: ediv 671 REG

See Also: emul
Mnemonic:       emul       Extended Multiply

Format:         emul       src1, src2, dst
               reg/lit       reg/lit       reg

Description:    Multiplies src2 by src1 and stores the result in dst. The result is a long
                ordinal (i.e., 64 bits), which is stored in two adjacent registers. The dst
                operand specifies the lower numbered register, which receives the least sig-
                nificant bits of the result. The dst operand must be an even numbered
                register (i.e., r0, r2, r4, ... or g0, g2, ...).

                This instruction performs ordinal arithmetic.

Action:         dst ← (src1 * src2) mod 2^32;
                dst + 1 ← (src * src2)/mod 2^32;

Faults:         STANDARD

Example:        emul r4, r5, g2       # g2, g3 ← r4 * r5

Opcode:         emul       670       REG

See Also:       ediv
**expr, exprl**

**Mnemonic:**
- `expr` : Exponent Real
- `exprl` : Exponent Long Real

**Format:**
- `exp*` : `src, dst`  
- `freg/flit` : `freg`

**Description:** Calculates an approximation of the exponential value of 2 to the `src` power, minus 1, and stores the result in `dst`. The `src` value must be within the range of -0.5 to +0.5, inclusive. If the `src` value is outside this range, the result is undefined.

For the `exprl` instruction, if the `src` or `dst` operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., `g0`, `g2`, `g4`).

The following table shows the results obtained when computing the exponent of various classes of numbers.

<table>
<thead>
<tr>
<th>Src</th>
<th>Dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.5 to -0</td>
<td>-(1/√2) - 1 to -0</td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>+0 to +0.5</td>
<td>+0 to √2 - 1</td>
</tr>
</tbody>
</table>

**Notes:**
*** Results are unpredictable

**Action:**

\[ dst \leftarrow (2^{src}) - 1; \]
**Faults:**

STANDARD

Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding

One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- **Floating Underflow**
  - Result is too small for destination format.

- **Floating Invalid Operation**
  - One or more operands are an SNaN value.

- **Floating Inexact**
  - Result cannot be represented exactly in destination format.

**Example:**

```plaintext
# y = 2^x  (y and x in g0)
# uses identity
#   2^x = 2^{I+f}
#   = 2^I * ((2^f - 1)+1)
# where:  I integer, -0.5 <= f <= +0.5
# assumes round-to-nearest
# does not handle infinities or NaNs
_pow2x:
   roundr g0,fp0       # I in fp0
   subr fp0,g0,g0     # f in g0
   expr g0,g0
   addr 0f1.0,g0,g0
   cvtri fp0,gl
   scaler gl,fp0,g0
```

**Opcode:**

- `expr` 689 REG
- `exprr` 699 REG

**See Also:**

scaler, logr
extract

Mnemonic: extract  Extract

Format: extract  
          bitpos, len, src/dst  
          reg/lit  reg/lit  reg

Description: Shifts a specified bit field in src/dst right and fills the bits to the left of the shifted bit field with zeros. The bitpos value specifies the least significant bit of the bit field to be shifted, and the len value specifies the length of the bit field.

Action:  
          src/dst ← (src/dst / 2^((bitpos mod 32)))  
                   and (2^len - 1);  

Faults:  STANDARD

Example:  extract 5, 12, g4  # g4 ← g4 with bits 5  
          # through 16 shifted right

Opcode:  extract  651  REG

See Also: modify
FAULT IF

Mnemonic:  
- faulte: Fault If Equal
- faultne: Fault If Not Equal
- faultl: Fault If Less
- faultle: Fault If Less Or Equal
- faultg: Fault If Greater
- faultge: Fault If Greater Or Equal
- faulto: Fault If Ordered
- faultno: Fault If Unordered

Format: fault\*

Description: Raises a constraint-range fault if the logical AND of the condition code and the mask-part of the opcode is not zero.

The following table shows the condition-code mask for each instruction:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mask</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>faultno</td>
<td>000</td>
<td>Unordered</td>
</tr>
<tr>
<td>faultg</td>
<td>001</td>
<td>Greater</td>
</tr>
<tr>
<td>faultl</td>
<td>010</td>
<td>Equal</td>
</tr>
<tr>
<td>faultge</td>
<td>011</td>
<td>Greater or equal</td>
</tr>
<tr>
<td>faultle</td>
<td>100</td>
<td>Less</td>
</tr>
<tr>
<td>faultne</td>
<td>101</td>
<td>Not equal</td>
</tr>
<tr>
<td>faulto</td>
<td>110</td>
<td>Less or equal</td>
</tr>
<tr>
<td>faultno</td>
<td>111</td>
<td>Ordered</td>
</tr>
</tbody>
</table>

For the faultno instruction (unordered), the fault is raised if the condition code is equal to 2#000#.

Action:  
For all instructions except faultno:

```plaintext
if (mask and AC.cc) ≠ 2#000#
    then raise constraint-range fault;
end if;
```

faultno:

```plaintext
if AC.cc = 2#000#
    then raise constraint-range fault;
end if;
```
FAULT IF

Faults: STANDARD, Constraint Range

Example: # assume 2#110# AND AC.cc ≠ 2#000#
faultle # raises Constraint Range Fault

Opcode: 

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>faulte</td>
<td>1A</td>
<td>CTRL</td>
</tr>
<tr>
<td>faultne</td>
<td>1D</td>
<td>CTRL</td>
</tr>
<tr>
<td>faultl</td>
<td>1C</td>
<td>CTRL</td>
</tr>
<tr>
<td>faultle</td>
<td>1E</td>
<td>CTRL</td>
</tr>
<tr>
<td>faultg</td>
<td>19</td>
<td>CTRL</td>
</tr>
<tr>
<td>faultge</td>
<td>1B</td>
<td>CTRL</td>
</tr>
<tr>
<td>faulto</td>
<td>1F</td>
<td>CTRL</td>
</tr>
<tr>
<td>faultno</td>
<td>18</td>
<td>CTRL</td>
</tr>
</tbody>
</table>

See Also: be, teste
flushreg

Mnemonic:      flushreg  Flush Local Registers

Format:        flushreg

Description:  Copies the contents of all the cached local-register sets into their associated register-save areas in the procedure stack. The contents of all the local-register sets except for the current set are then marked as invalid. On a return, the local registers for the frame being returned to are then loaded from the stack.

The flushreg instruction is provided to allow a compiler or applications program to circumvent the normal call/return mechanism of the processor. For example, a compiler may need to back up several frames in the stack on the next return, rather than using the normal return mechanism that returns one frame at a time. Here, the compiler uses the flushreg instruction to update the stack with the current states of the saved register sets. The compiler can then return to any frame in the stack without losing the contents of the saved local-register sets. To return to a frame other than the frame directly below the current frame, the compiler merely modifies the PFP in register r0 of the current frame to point to the frame that it wishes to return to.

Action:        Each register set except the current set is flushed to its associated stack frame in memory and marked as purged, meaning that they will be reloaded from memory if and when they become the current local register set.

Faults:        STANDARD

Example:       flushreg

Opcode:        flushreg  66D  REG
**fmark**

**Mnemonic:**  
\texttt{fmark} \quad \text{Force Mark}

**Format:**  
\texttt{fmark}

**Description:**  
Generates a breakpoint trace-event, regardless of the setting of the breakpoint trace mode flag.

When a breakpoint trace event is detected, the trace-fault-pending flag (bit 10) of the process controls word and the breakpoint-trace-event flag (bit 23) of the trace controls are set. Before the next instruction is executed, a trace fault is generated.

For more information on trace-fault generation, refer to Chapter 12.

**Action:**  
\begin{verbatim}
if process.trace_controls and breakpoint_trace_flag
  then
    raise trace breakpoint fault
endif
\end{verbatim}

**Faults:**  
STANDARD, Breakpoint Trace

**Example:**  
\begin{verbatim}
ld xyz, r4
addi r4, r5, r6
fmark
# Breakpoint trace event is generated at
# this point in the instruction stream.
\end{verbatim}

**Opcode:**  
\texttt{fmark} \quad \texttt{66C} \quad \text{REG}

**See Also:**  
mark
LOAD

Mnemonic:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>Load</td>
</tr>
<tr>
<td>ldob</td>
<td>Load Ordinal Byte</td>
</tr>
<tr>
<td>ldos</td>
<td>Load Ordinal Short</td>
</tr>
<tr>
<td>ldib</td>
<td>Load Integer Byte</td>
</tr>
<tr>
<td>ldis</td>
<td>Load Integer Short</td>
</tr>
<tr>
<td>ldl</td>
<td>Load Long</td>
</tr>
<tr>
<td>ldt</td>
<td>Load Triple</td>
</tr>
<tr>
<td>ldq</td>
<td>Load Quad</td>
</tr>
</tbody>
</table>

Format: \( ld^* \quad src, \quad dst \quad mem \quad reg \)

Description: Copies a byte or string of bytes from memory into a register or group of successive registers. The \( src \) operand specifies the address of the first byte to be loaded. The full range of addressing modes may be used in specifying \( src \). (Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.)

The \( dst \) operand specifies a register or the first (lowest numbered) register of successive registers.

The \( ldob \) and \( ldib \), and \( ldos \) and \( ldis \) instructions load a byte and half word, respectively, and convert it to a full 32-bit word. The \( ld \), \( ldl \), \( ldt \), and \( ldq \) instructions copy 4, 8, 12, and 16 bytes, respectively, from memory into successive registers.

For the \( ldl \) instruction, \( dst \) must specify an even numbered register (e.g., \( g0 \), \( g2 \), \( g4 \), \( g8 \)). For the \( ldt \) and \( ldq \) instructions, \( dst \) must specify a register number that is a multiple of four (e.g., \( g0 \), \( g4 \), \( g8 \)). If the data extends beyond register \( g15 \) or \( r15 \) for the \( ldl \), \( ldt \), or \( ldq \) instruction, the results are unpredictable.

Action: \( dst \leftarrow \text{memory} (src); \)

Faults: STANDARD

Example: \( ldl \ 2456 \ (r3), \ r10 \)  # \( r10 \), \( r11 \leftarrow \text{value of two} \)
          # words beginning at offset
          # 2456 plus the address in
          # \( r3 \) in memory
LOAD

<table>
<thead>
<tr>
<th>Opcode</th>
<th>OPCODE</th>
<th>MODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>90</td>
<td>MEM</td>
</tr>
<tr>
<td>ldob</td>
<td>80</td>
<td>MEM</td>
</tr>
<tr>
<td>ldos</td>
<td>88</td>
<td>MEM</td>
</tr>
<tr>
<td>ldib</td>
<td>C0</td>
<td>MEM</td>
</tr>
<tr>
<td>ldis</td>
<td>C8</td>
<td>MEM</td>
</tr>
<tr>
<td>ldl</td>
<td>98</td>
<td>MEM</td>
</tr>
<tr>
<td>ltd</td>
<td>A0</td>
<td>MEM</td>
</tr>
<tr>
<td>ldq</td>
<td>B0</td>
<td>MEM</td>
</tr>
</tbody>
</table>

See Also: MOVE, STORE
**Mnemonic:** lda

**Format:**

```
Ida src               dst
   mem               reg
   efa
```

**Description:** Computes the effective address specified with `src` and stores it in `dst`. The `src` address is not checked for validity.

An important application of this instruction is to load a constant longer than 5 bits into a register. (To load a register with a constant of 5 bits or less, the move instruction (mov) can be used with a literal as the `src` operand.)

**Action:**

```
dst ← efa (src);
```

**Faults:** STANDARD

**Example:**

```
Ida 58 (g9), gl  # Computes the effective address specified with 58 (g9) and stores it in gl
Ida 0x749, r8    # loads the constant 16#749# in r8
```

**Opcode:**

```
Ida 8C      MEM
```
**logbnr, logbnrl**

**Mnemonic:**
- `logbnr` Log Binary Real
- `logbnrl` Log Binary Long Real

**Format:**

```
logbnr* src, dst
freg/flit freg
```

**Description:**
Calculates the $\log_2(src)$ and stores the integral part of this value (i.e., the part to the left of the binary point) as a real number in `dst`. The result of this operation is an unbiased exponent. When `src` is a denormalized number, `dst` is the unbiased exponent that `src` would have if the format had unlimited exponent range.

(The fractional part of $\log_2(src)$ is ignored. If the fractional part is needed, use the `logr` or `logrl` instruction.)

This instruction implements the IEEE recommended function $\log b$. It is useful for calculating the order of magnitude of a number.

For the `logbnrl` instruction, if the `src2` or `dst` operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., `g0`, `g2`, `g4`).

The following table shows the results obtained when taking the log binary of various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>Src</th>
<th>Dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>$-F$</td>
<td>$\pm F$</td>
</tr>
<tr>
<td>$-0$</td>
<td>**</td>
</tr>
<tr>
<td>$+0$</td>
<td>**</td>
</tr>
<tr>
<td>$+F$</td>
<td>$\pm F$</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**Notes:**
- `F` Means finite-real number
- `**` Indicates floating zero-divide exception
logbnr, logbnrl

Note that the significand of the src operand can be extracted by using the scaler or scalerl instruction.

**Action:**

\[ dst \leftarrow (\log_2 \text{(unbiased exponent (src)}) - \text{fraction}); \]

# the integral part of the unbiased exponent of src
# is stored in dst as a biased real

**Faults:**

- **STANDARD**
  Refer to the discussion of faults at the beginning of this chapter.

- **Floating Reserved Encoding**
  One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- **Floating Underflow**
  Result is too small for destination format.

- **Floating Invalid Operation**
  One or more operands are an SNaN value.

- **Floating Inexact**
  Result cannot be represented exactly in destination format.

- **Floating Zero Divide**
  The src operand is 0.

**Example:**

\[ \logbnrl \ g12, \ fp3 \]

# fp3 \leftarrow \text{integral part}
# of \log_2 (g12,g13)

**Opcode:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>logbnr</td>
<td>68A</td>
<td>REG</td>
</tr>
<tr>
<td>logbnrl</td>
<td>69A</td>
<td>REG</td>
</tr>
</tbody>
</table>

**See Also:** logr, scaler
logep, logepfl

Mnemonic: logep Log Epsilon Real
logepfl Log Epsilon Long Real

Format: logep* src1, src2, dst
freg/flit freg/flit freg

Description: Calculates (src2 * log2 (src1 + 1)), and stores the result in dst.

For the logepfl instruction, if the src1, src2, or dst operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>Src1</th>
<th>(1/√2) -1 to -0</th>
<th>-0</th>
<th>+0</th>
<th>+0 to √2 -1</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>-∞</td>
<td>*</td>
<td>*</td>
<td>-∞</td>
<td>NaN</td>
</tr>
<tr>
<td>-F</td>
<td>+F</td>
<td>+0</td>
<td>-0</td>
<td>-F</td>
<td>NaN</td>
</tr>
<tr>
<td>0</td>
<td>+0</td>
<td>+0</td>
<td>-0</td>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td>*</td>
<td>*</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Notes:
F Means finite-real number.
* Indicates floating invalid-operation exception.

This instruction offers optimal accuracy for values of src1 + 1 close to 1 (i.e., for values of src1 close to 0). This expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in src2.
The following equation is used to calculate the scale factor for a particular logarithm base, where \( n \) is the logarithm base desired for the result stored in \( dst \):

\[
\text{scale factor} = \log_n 2
\]

The range of \( src1 \) is restricted to the following:

\[
\frac{1}{\sqrt{2}} \leq src1 + 1 \leq \sqrt{2}
\]

When the \( src1 \) operand is outside this range, the \texttt{logr} or \texttt{logrl} instruction can be used with very insignificant loss of accuracy by adding 1.0 to \( src1 \).

**Action:**

\[
dst \leftarrow src2 \times \log_2 (src1 + 1);
\]

**Faults:**

- **STANDARD**
  - Refer to the discussion of faults at the beginning of this chapter.
- **Floating Reserved Encoding**
  - One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- **Floating Overflow**
  - Result is too large for destination format.
- **Floating Underflow**
  - Result is too small for destination format.
- **Floating Invalid Operation**
  - The \( src1 \) operand is 0 and the \( src2 \) operand is \( \infty \).
  - The \( src1 \) operand does not fall within the range defined in the above description section.
  - One or more operands are an SNaN value.
- **Floating Inexact**
  - Result cannot be represented exactly in destination format.
**logepr, logeprl**

**Example:**

```plaintext
logepr g8, g4, fp2
# fp2 ← g4,g5 * log2 (g8,g9 + 1)
```

**Opcode:**

- `logepr` 681 REG
- `logeprl` 691 REG

**See Also:** logr
Mnemonic: logr  Log Real
         logrl Log Long Real

Format:  logr*   src1,  src2,  dst
         freg/flit  freg/flit  freg

Description: Calculates \((src2 \times \log_2 (src1))\), and stores the result in \(dst\). (The \textbf{logbnr} and \textbf{logbnrl} instructions perform this function more efficiently, if only an estimate is needed.)

For the \textbf{logrl} instruction, if the \(src1, src2,\) or \(dst\) operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th></th>
<th>-∞</th>
<th>-F</th>
<th>-0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>**</td>
<td>±∞</td>
<td>-∞</td>
<td>NaN</td>
</tr>
<tr>
<td>-F</td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>**</td>
<td>±F</td>
<td>-∞</td>
<td>NaN</td>
</tr>
<tr>
<td>-0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>±0</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>±0</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>**</td>
<td>±F</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>**</td>
<td>±∞</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Notes:

- \(F\)  Means finite-real number.
- * Indicates floating invalid-operation exception.
- ** Indicates floating zero-divide exception.

The \textbf{logr} instruction combined with the \textbf{expr} instruction forms the basis for the power function \(x^y\).
Adding 1.0 to a number to be used as the src1 operand will cause information to be lost. To perform this function, use the logepr or logeprl instruction.

These instructions provide a simple method of converting the result of the \( \log_2 \) arithmetic into a value in another logarithm base by including a scale factor in src2. The following equation is used to calculate the scale factor for a particular logarithm base, where \( n \) is the logarithm base desired for the result stored in dst:

\[
\text{scale factor} = \log_n 2
\]

**Action:**
\[
dst \leftarrow src2 \times \log_2 (src1);
\]

**Faults:**
- STANDARD: Refer to the discussion of faults at the beginning of this chapter.
- Floating Reserved Encoding: One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- Floating Overflow: Result is too large for destination format.
- Floating Underflow: Result is too small for destination format.
- Floating Zero Divide: The src1 operand is 0 and src2 is non-zero.
- Floating Invalid Operation: The src1 and src2 operands are both 0.
  - The src1 operand is \( \infty \) and the src2 operand is 0.
  - The src1 operand is 1 and the src2 operand is \( \infty \).
  - The src1 operand is negative and non-zero.
- Floating Inexact: One or more operands are an SNaN value.
- Result cannot be represented exactly in destination format.
Example: \[ \text{logrl } r2, g8, g2 \] # g2,g3 \( \leftarrow \) g8,g9 \( \times \) \( \log_2(r2,r3) \)

Opcode:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Code</th>
<th>REG</th>
</tr>
</thead>
<tbody>
<tr>
<td>logr</td>
<td>682</td>
<td>REG</td>
</tr>
<tr>
<td>logrl</td>
<td>692</td>
<td>REG</td>
</tr>
</tbody>
</table>

See Also: expr, logepr
mark

Mnemonic: mark    Mark

Format: mark

Description: Generates a breakpoint trace event if the breakpoint trace mode has been enabled. The breakpoint trace mode is enabled if the trace-enable bit (bit 0) of the process controls and the breakpoint-trace mode bit (bit 7) of the trace controls have been set. Both these words are located in the PCB.

When a breakpoint trace event is detected, the trace-fault-pending flag (bit 10) of the process controls and the breakpoint-trace-event flag (bit 23) of the trace controls are set. Before the next instruction is executed, a trace fault is generated.

If the breakpoint-trace mode has not been enabled, the **mark** instruction behaves like a no-op.

For more information on trace-fault generation, refer to Chapter 12.

Action: raise trace breakpoint fault

Faults: STANDARD, Breakpoint Trace

Example: # Assume that the breakpoint trace mode is enabled.
ld xyz, r4
addi r4, r5, r6
mark
# Breakpoint trace event is generated at this point in the instruction stream.

Opcode: mark  66B  REG

See Also: fmark, modpc, modtc
modac

Mnemonic: modac  Modify AC

Format: modac  mask,  src,  dst
        reg/lit  reg/lit  reg

Description: Reads and modifies the arithmetic controls. The src operand contains the value to be placed in the arithmetic controls and the mask operand specifies the bits that may be changed. Only the bits set in mask are modified in the arithmetic controls. Once the arithmetic controls have been changed, their initial state is copied into dst.

Action: temp ← AC
        AC ← (src and mask) or
              (AC and not (mask));
        dst ← temp;

Faults: STANDARD

Example: g1, g9, g12  # AC ← g9, masked by g1
         # g12 ← initial value of AC

Opcode: modac  645  REG

See Also: modpc, modtc
**modi**

**Mnemonic:** modi  Modulo Integer

**Format:**

```
modi src1, src2, dst
reg/lit reg/lit reg
```

**Description:** Divides src2 by src1, where both are integers, and stores the modulo remainder of the result in dst. If the result is nonzero, dst is given the same sign as src1.

**Action:**

```
dst ← src2 - ((src2/src1) * src1);
if src2 * src1 < 0
    then dst ← dst + src1;
end if;
```

**Faults:** STANDARD, Arithmetic Zero Divide

**Example:**

```
modi r9, r2, r5,  # r5 ← modulo (r2/r9)
```

**Opcode:**

```
modi 749 REG
```

**See Also:** div, remi
modify

Mnemonic: modify

Format: modify mask, src, src/dst
        reg/lit   reg/lit   reg

Description: Modifies selected bits in src/dst with bits from src. The mask operand selects the bits to be modified: only the bits set in the mask are modified in src/dst.

Action: src/dst ← (src and mask) or (src/dst and not (mask));

Faults: STANDARD

Example: modify g8, g10, r4 # r4 ← g10 masked by g8

Opcode: modify 650 REG

See Also: alterbit, extract
modpc

Mnemonic: modpc  Modify Process Controls

Format: modpc src, mask, src/dst
         reg/lit   reg/lit   reg

Description: Reads and modifies the processor’s internally cached process controls as specified with mask and src/dst. The src/dst operand contains the value to be placed in the process controls and the mask operand specifies the bits that may be changed. Only the bits set in the mask are modified in the process controls. Once the process controls have been changed, their initial value is copied into src/dst. The src operand is a dummy operand that should be set equal to the mask operand.

The processor must be in the supervisor mode to modify the process controls using this instruction. If the mask operand is set to 0, this instruction can be used to read the process controls, without the processor being in the supervisor mode.

If the action of this instruction results in the priority of the processor being lowered, the interrupt table is checked for pending interrupts.

Changing the state, resume, internal state, and trace enable fields of the process controls can lead to unpredictable behavior, as described in Chapter 7 in the section titled "Changing the Process-Controls Word."

Action:

if mask ≠ 0
    then if process.process_controls.execution_mode ≠ supervisor
        then raise type-mismatch fault;
        end if;
        temp ← process.process_controls;
        process.process_controls ←
        (mask and src/dst) or
        (process.process_controls and not (mask));
        src/dst ← temp;
        if (temp.priority > process.process_controls.priority
            then check_pending_interrupts;
            # if continue here, no interrupt to do
        end if;
    else src/dst ← process.process_controls;
end if;
**modpc**

**Faults:** STANDARD, Type Mismatch

**Example:**
```
modpc g9, g9, g8  # process controls ← g8
# masked by g9
```

**Opcode:**
```
modpc  655  REG
```

**See Also:** modac, modtc
modtc

Mnemonic: modtc Modify Trace Controls

Format: modtc mask, src, dst
         reg/lit   reg/lit   reg

Description: Reads and modifies the trace controls for the current process. The processor changes its internally cached trace controls as specified with mask and src. The src operand contains the value to be placed in the trace controls and the mask operand specifies the bits that may be changed. Only the bits set in the mask are modified in the trace controls. Once the trace controls have been changed, their initial state is copied into dst.

This instruction only affects the trace controls cached in processor. The trace controls in the PCB for the current process are not affected.

Since bits 8 through 15 and 24 through 31 of the trace-controls word are reserved, the mask operand is ANDed with 00FF00FF16 to insure that these bits are not set in the mask.

The changed trace controls take effect on the first non-branching instruction fetched from memory. Since instructions are prefetched four at a time, the trace controls may not take effect for up to the next four instructions executed.

For more information on the trace controls, refer to Chapters 12 and 16.

Action: temp ← process.trace_controls;
temp1 ← 16#00FF00FF# and mask;
process.trace_controls ←
    (temp1 and src) or
    (process.trace_controls and not(temp1));
dst ← temp;

Faults: STANDARD

Example: modtc g12, g10, g2
         # trace controls ← g10 masked by g12;
         # previous trace controls stored in g2

Opcode: modtc 654 REG

See Also: modac, modpc
MOVE

Mnemonic: 
mov Move
movl Move Long
movt Move Triple
movq Move Quad

Format: mov* src, dst
        reg/lit reg

Description: Copies the content of one or more source registers (specified with the src operand) to one or more destination registers (specified with the dst operand).

For the movl, movt, and movq instructions, the src and dst operands specify the first (lowest numbered) register of several successive registers. The src and dst registers must be even numbered (e.g., g0, g2) for the movl instruction and an integral multiple of four (e.g., g0, g4) for the movt and movq instructions.

When the src and dst operands overlap, the value moved is unpredictable.

Action: dst ← src;

Faults: STANDARD

Example: movt g8, r4  # r4, r5, r6 ← g8, g9, g10

Opcode: mov 5CC REG
         movl 5DC REG
         movt 5EC REG
         movq 5FC REG

See Also: ld, movr, st
movr, movre, movrl

Mnemonic:
- movr: Move Real
- movrl: Move Long Real
- movre: Move Extended Real

Format:

\[
\text{movr*} \quad \text{src}, \quad \text{dst} \\
\text{freg/flit} \quad \text{freg}
\]

Description:
Copies a real value from one or more source registers (specified with the src operand) to one or more destination registers (specified with the dst operand).

For the movrl instruction, if the src or dst operand references a global or local register, this register is the first (lowest numbered) of two successive registers. For the movre instruction, if the src or dst operand references a global or local register, this register is the first (lowest numbered) of three successive registers.

When copying real numbers between global or local registers and floating-point registers, conversion between real or long-real format to extended-real format is performed implicitly. Conversion between real and long-real formats must be done through floating-point registers and requires two instructions, as illustrated in the example below.

When the movre instruction moves an operand from global or local registers to a floating-point register, it automatically truncates the most-significant 16 bits of the word in the third register (refer to Figure 12-5). Likewise, when this instruction is used to move an operand from a floating-point register to global or local registers, it adds 16 zeros to the third word. The movre instruction is not a numeric instruction; it merely manipulates bits.

The movr and movrl instructions can cause a floating-point exception to be raised, which might result in a fault being raised, as is explained in the section below on faults. The movre instruction can never raise an exception and thus never faults.

Action:

\[
dst \leftarrow src;
\]
Faults: STANDARD  Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding  One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow  Result is too large for destination format.
Floating Underflow  Result is too small for destination format.
Floating Invalid Operation  Source operand is an SNaN value.
Floating Inexact  Result cannot be represented exactly in destination format.

Example:  # Conversion of real value in g3 to a # to a long real value, which is stored # in g4,g5
          movr g3, fp2
          movrl fp2, g4

Opcode:  movr  6C9  REG
          movrl  6D9  REG
          movre  6E9  REG

See Also:  mov
muli, mulo

**Mnemonic:**
- **muli** Multiply Integer
- **mulo** Multiply Ordinal

**Format:**

```
mul*       src1,   src2,   dst
    reg/lit  reg/lit  reg
```

**Description:** Multiplies the `src2` value by the `src1` value and stores the result in `dst`.

**Action:**

```
dst ← src2 * src1;
```

**Faults:** STANDARD, Integer Overflow

**Example:**

```
muli r3, r4, r9  # r9 ← r4 TIMES r3
```

**Opcode:**

- **muli** 741 REG
- **mulo** 701 REG

**See Also:** emul, mulr
Mnemonic: \textbf{mulr} \hspace{1em} Multiply Real
\textbf{mulrl} \hspace{1em} Multiply Long Real

Format: \texttt{mulr} \hspace{1em} \textit{src1}, \hspace{1em} \textit{src2}, \hspace{1em} \textit{dst}
\hspace{1em} \texttt{freg/flit} \hspace{1em} \texttt{freg/flit} \hspace{1em} \texttt{freg}

Description: Multiplies the \textit{src2} value by the \textit{src1} value and stores the result in \textit{dst}.

For the \texttt{mulrl} instruction, if the \textit{src1}, \textit{src2}, or \textit{dst} operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the source values is 0, \(\infty\), or a NaN.

The following table shows the results obtained when multiplying various classes of numbers together, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>\textbf{Src1}</th>
<th>-\infty</th>
<th>-F</th>
<th>-0</th>
<th>+0</th>
<th>+F</th>
<th>+\infty</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-\infty</td>
<td>+\infty</td>
<td>+\infty</td>
<td>*</td>
<td>*</td>
<td>-\infty</td>
<td>-\infty</td>
<td>NaN</td>
</tr>
<tr>
<td>-F</td>
<td>+\infty</td>
<td>+F</td>
<td>+0</td>
<td>-0</td>
<td>-F</td>
<td>-\infty</td>
<td>NaN</td>
</tr>
<tr>
<td>-0</td>
<td>*</td>
<td>+0</td>
<td>+0</td>
<td>-0</td>
<td>-0</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>*</td>
<td>-0</td>
<td>-0</td>
<td>+0</td>
<td>+0</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>-\infty</td>
<td>-F</td>
<td>-0</td>
<td>+0</td>
<td>+F</td>
<td>+\infty</td>
<td>NaN</td>
</tr>
<tr>
<td>+\infty</td>
<td>-\infty</td>
<td>-\infty</td>
<td>*</td>
<td>*</td>
<td>+\infty</td>
<td>+\infty</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Notes:
\begin{itemize}
  \item F \hspace{1em} Means finite-real number.
  \item * \hspace{1em} Indicates floating invalid-operation exception.
\end{itemize}

When you need to multiply by the power of 2, the \texttt{scaler} and \texttt{scalerl} instructions can also be used.

\textbf{Action:} \hspace{1em} \texttt{dst} \leftarrow \texttt{src2} \times \texttt{src1};
### mulr, mulrl

**Faults:**
- STANDARD: Refer to the discussion of faults at the beginning of this chapter.
- Floating Reserved Encoding: One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- **Floating Overflow:** Result is too large for destination format.
- **Floating Underflow:** Result is too small for destination format.
- **Floating Invalid Operation:** One source operand is 0 and the other is \( \infty \).
  - One or more operands are an SNaN value.
- **Floating Inexact:** Result cannot be represented exactly in destination format.

**Example:**
```
mulrl g12, g4, fp2  # fp2 ← g4, g5 * g12, g13
```

**Opcode:**
- mulr: 78C REG
- mulrl: 79C REG

**See Also:** emul, muli, scaler
<table>
<thead>
<tr>
<th>Mnemonic:</th>
<th>nand</th>
<th>Nand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format:</td>
<td>nand</td>
<td>src1, src2, dst</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg/lit, reg/lit, reg</td>
</tr>
<tr>
<td>Description:</td>
<td>Performs a bitwise NAND operation on the src2 and src1 values and stores the result in dst.</td>
<td></td>
</tr>
<tr>
<td>Action:</td>
<td>dst ← (not (src2)) or not (src1);</td>
<td></td>
</tr>
<tr>
<td>Faults:</td>
<td>STANDARD</td>
<td></td>
</tr>
<tr>
<td>Example:</td>
<td>nand g5, r3, r7 # r7 ← r3 NAND g5</td>
<td></td>
</tr>
<tr>
<td>Opcode:</td>
<td>nand</td>
<td>58E REG</td>
</tr>
<tr>
<td>See Also:</td>
<td>and, andnot, nor, not, notand, notor, or, ornot, xnor, xor</td>
<td></td>
</tr>
</tbody>
</table>
nor

Mnemonic: nor Nor

Format: nor   src1,     src2,     dst
         reg/lit  reg/lit  reg

Description: Performs a bitwise NOR operation on the src2 and src1 values and stores the result in dst.

Action: dst ← not (src2) and not (src1);

Faults: STANDARD

Example: nor g8, 28, r5  # r5 ← 28 NOR g8

Opcode: nor   588     REG

See Also: and, andnot, nand, not, notand, notor, or, ornot, xnor, xor
### not, notand

**Mnemonic:**
- not: Not
- notand: Not And

**Format:**
- not: 
  - src, dst
  - reg/lit, reg
- notand: 
  - src1, src2, dst
  - reg/lit, reg/lit, reg

**Description:** Performs a bitwise NOT (not instruction) or NOT AND (notand instruction) operation on the src2 and src1 values and stores the result in dst.

**Action:**
- not: 
  - dst ← not (src1);
- notand: 
  - dst ← (not (src2)) and src1;

**Faults:** STANDARD

**Example:**
- not g2, g4  
  # g4 ← NOT g2
- notand r5, r6, r7  
  # r7 ← NOT r6 AND r5

**Opcode:**
- not: 58A REG
- notand: 584 REG

**See Also:** and, andnot, nand, nor, notor, or, ornot, xnor, xor
notbit

Mnemonic: notbit  Not Bit

Format: notbit  bitpos,  src,  dst
         reg/lit   reg/lit   reg

Description: Copies the src value to dst with one bit toggled. The bitpos operand specifies the bit to be toggled.

Action: \( dst \leftarrow src \ xor 2^{(bitpos \mod 32)}; \)

Faults: STANDARD

Example: notbit r3, r12, r7  # r7 \leftarrow r12 with the bit
         # specified in r3 toggled

Opcode: notbit  580  REG

See Also: alterbit, chkbit, clrbit, setbit
Mnemonic: notor
Not Or

Format: notor src1, src2, dst
reg/lit reg/lit reg

Description: Performs a bitwise NOT OR operation on the src2 and src1 values and stores the result in dst.

Action: dst ← (not (src2)) or src1;

Faults: STANDARD

Example: notor g12, g3, g6  # g6 ← NOT g3 OR g12

Opcode: notor 58D REG

See Also: and, andnot, nand, nor, not, notand, or, ornot, xnor, xor
**or, ornot**

**Mnemonic:**
- or
- ornot

**Format:**
- or: `src1, src2, dst`  
  ```
  reg/lit reg/lit reg
  ```
- ornot: `src1, src2, dst`  
  ```
  reg/lit reg/lit reg
  ```

**Description:** Performs a bitwise OR (or instruction) or ORNOT (ornot instruction) operation on the `src2` and `src1` values and stores the result in `dst`.

**Action:**
- or: \[ dst \leftarrow src2 \text{ or } src1; \]
- ornot: \[ dst \leftarrow src2 \text{ or not } (src1); \]

**Faults:** STANDARD

**Example:**
- or: `or 14, g9, g3`  
  ```
  # g3 \leftarrow g9 \text{ OR } 14
  ```
- ornot: `ornot r3, r8, r11`  
  ```
  # r11 \leftarrow r8 \text{ OR NOT } r3
  ```

**Opcode:**
- or: 587, REG
- ornot: 58B, REG

**See Also:** and, andnot, nand, nor, not, notand, notor, xnor, xor
Mnemonic: \( \text{remi} \) \( \text{remo} \)  

Remainder Integer  
Remainder Ordinal  

Format: \( \text{rem}^* \) \( \text{srcl} \), \( \text{src2} \), \( \text{dst} \) \( \text{reg/lit} \) \( \text{reg/lit} \) \( \text{reg} \)  

Description: Divides \( \text{src2} \) by \( \text{srcl} \) and stores the remainder in \( \text{dst} \). The sign of the result (if nonzero) is the same as the sign of \( \text{src2} \).  

Action: \( \text{dst} \leftarrow \text{src2} - (\text{src2} / \text{srcl}) \times \text{srcl}; \)  

Faults: \( \text{STANDARD} \) \( \text{Integer Overflow} \)  

Refer to discussion of faults at the beginning of this chapter.  

Result is too large for destination format. This fault is signaled only when executing the \( \text{remi} \) instruction and if both of the following conditions are met: (1) the integer-overflow mask in the arithmetic-controls registers is clear and (2) the source operands have like signs and the sign of the result operand is different than the signs of the source operands.  

Example: \( \text{remo} \text{r4}, \text{r5}, \text{r6} \) \# \( \text{r6} \leftarrow \text{r5} \text{rem} \text{r4} \)  

Opcode: \( \text{remi} \) 748 \( \text{REG} \)  
\( \text{remo} \) 708 \( \text{REG} \)  

See Also: \( \text{remr}, \text{modi} \)
### remr, remrl

**Mnemonic:**
- `remr`  Remainder Real
- `remrl`  Remainder Long Real

**Format:**
```
remr*  src1,  src2,  dst
freg/flit  freg/flit  freg
```

**Description:** Divides `src2` by `src1` and stores the remainder in `dst`. The sign of the result (if nonzero) is the same as the sign of `src2`.

For the `remrl` instruction, if the `src1`, `src2`, or `dst` operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>Src1</th>
<th>-∞</th>
<th>-F</th>
<th>-0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NaN</td>
</tr>
<tr>
<td>-∞</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>-F</td>
<td>src2</td>
<td>-F or -0</td>
<td>**</td>
<td>**</td>
<td>-F or -0</td>
<td>src2</td>
<td>NaN</td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td>-0</td>
<td>*</td>
<td>*</td>
<td>-0</td>
<td>-0</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>*</td>
<td>*</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>src2</td>
<td>+F or +0</td>
<td>**</td>
<td>**</td>
<td>+F or +0</td>
<td>src2</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**Notes:**
- `F` Means finite-real number.
- `*` Indicates floating invalid-operation exception.
- `**` Indicates floating zero-divide exception.

When the result is 0, its sign is the same as that of `src2`. When the `src1` is ∞, the result is equal to the `src2`.

The result of this operation is always exact if the destination format is at least as wide as the `src2` and `src1`.
The remainder provided with the `remr` and `remrl` instructions is different from the remainder described in the IEEE floating-point standard. The difference is related to how the quotient (N) of the expression `(src2/src1)` is determined.

As shown below in the action statement, N for the `remr` and `remrl` instructions is the nearest integer value obtained when the exact result (E) of the expression `(src2/src1)` is truncated toward zero. N will always be less than or equal to the absolute value of E.

For the IEEE standard, N is simply the nearest integer value to E. Here, N may be less than, equal to, or greater than the absolute value of E.

To help determine the IEEE remainder from the result given by the `remr` and `remrl` instructions, the following information about the quotient is given in the arithmetic-status field in the arithmetic:

<table>
<thead>
<tr>
<th>Arithmetic Status Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Q1, the next-to-last quotient bit</td>
</tr>
<tr>
<td>5</td>
<td>Q0, the last quotient bit</td>
</tr>
<tr>
<td>4</td>
<td>QR, the value the next quotient bit would have if one more reduction were performed (the &quot;round&quot; bit of the quotient)</td>
</tr>
<tr>
<td>3</td>
<td>QS, set if the remainder after the QR reduction would be nonzero (the &quot;sticky&quot; bit of the quotient)</td>
</tr>
</tbody>
</table>

The information can then be used to determine the IEEE standard remainder, as shown in the example below:

```
Action:  dst ← src2 - (N * src1);
# where N = truncate (src2/src1).
# Here, (src2/src1) is truncated
# toward zero to the nearest integer.
```
### remr, remrl

**Faults:**

<table>
<thead>
<tr>
<th>Fault</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STANDARD</td>
<td>Refer to the discussion of faults at the beginning of this chapter.</td>
</tr>
<tr>
<td>Floating Reserved Encoding</td>
<td>One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.</td>
</tr>
</tbody>
</table>

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating Overflow</td>
<td>Result is too large for destination format.</td>
</tr>
<tr>
<td>Floating Underflow</td>
<td>Result is too small for destination format.</td>
</tr>
<tr>
<td>Floating Zero Divide</td>
<td>The src1 operand is 0.</td>
</tr>
<tr>
<td>Floating Invalid Operation</td>
<td>The src2 operand is ∞.</td>
</tr>
<tr>
<td></td>
<td>The src1 operand is 0.</td>
</tr>
<tr>
<td></td>
<td>One or more operands are an SNaN value.</td>
</tr>
<tr>
<td>Floating Inexact</td>
<td>Result cannot be represented exactly in destination format.</td>
</tr>
</tbody>
</table>

**Example:**

```
remrl g6, g8, fp1
# fp1 ← g8,g9 rem g6,g7
```

**Opcode:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>remr</td>
<td>683 REG</td>
</tr>
<tr>
<td>remrl</td>
<td>693 REG</td>
</tr>
</tbody>
</table>

**See Also:** remi, modi
Mnemonic: ret  Return

Format: ret

Description: Returns process control to the calling procedure. The current stack frame (i.e., that of the called procedure) is deallocated and the FP is changed to point to the stack frame of the calling procedure. Instruction execution is continued at the instruction pointed to by the RIP in the calling procedure’s stack frame, which is the instruction immediately following the call instruction.

As shown in the action statement below, the action that the processor takes on the return is determined by the return status and prereturn trace bits. These bits are contained in bits 0, through 3 of register r0 of the current set of local registers.

Refer to Chapter 4 for further discussion of the return instruction.

Action: wait for any uncompleted instructions to finish;
case frame_status is

2#000#:  FP ← PFP;
         free current register_set;
         if register_set (FP) not allocated
         then retrieve from memory(FP);
         end if;
         IP ← RIP;

2#001#:  x ← memory(FP-16);
         y ← memory(FP-12);
         do case 000 action;
         arithmetic_controls ← y;
         if execution_mode = supervisor
         then process_controls ← x;
         end if;

2#010#:  if execution_mode ≠ supervisor
         then go to case 000;
         else process_controls.T ← 0;
         execution_mode ← user;
         go to case 000;
         end if;
ret

2#111#: 
if execution_mode ≠ supervisor
    then free current register set;
    check_pending_interruptions;
    # if continue here, no interrupt to do
    do case 000 action;
end if;

2#110#: 
    if execution_mode = supervisor
        then free current register set;
        check_pending_interruptions;
        # if continue here, no interrupt to do
        do case 000 action;
end if;

2#100#: undefined

2#101#: undefined

Faults: STANDARD

Example: ret # process control returns to
          # calling procedure
          # environment

Opcode: ret 0A CTRL

See Also: call, calls, callx
rotate

Mnemonic: rotate    Rotate

Format: rotate    len, src, dst
            reg/lit   reg/lit   reg

Description: Copies src to dst and rotates the bits in the resulting dst operand to the left (toward higher significance). (The bits shifted off the left end of the word are inserted at the right end of the word.) The len operand specifies the number of bits that the dst operand is rotated. The len operand can range from 0 to 31.

This instruction can also be used to rotate bits to the right. Here, the number of bits the word is to be rotated right is subtracted from 32 to get the len operand.

Action:       dst ← rotate (len mod 32 (src))

Faults:       STANDARD

Example:       rotate r4, r8, r12  # r12 ← r8
               # with bits rotated
               # r4 bits to left

Opcode:        rotate  59D    REG

See Also:      SHIFT
roundr, roundrl

Mnemonic:  
roundr  Round Real  
roundrl Round Long Real

Format:  
roundr*  src,  dst  
freg/flit  freg

Description:  Rounds src to the nearest integral value, depending on the rounding mode, and stores the result in dst.

For the roundrl instruction, if the src or dst operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

If the src operand is $\infty$ the result is src. If the src operand is not an integral value, a floating-inexact exception is raised.

Action:  
dst <- round_to_integral_value(src);

Faults:  
STANDARD  Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding  One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow  Result is too large for destination format.
Floating Underflow  Result is too small for destination format.
Floating Invalid Operation  One or more operands are an SNaN value.
Floating Inexact  Result cannot be represented exactly in destination format.

Example:  
roundrl r4, r10  
# r10,r11 <- r4,r5 rounded

Opcode:  
roundr  68B  REG  
roundrl  69B  REG
Mnemonic: scaler   Scale Real
        scalerl  Scale Long Real

Format: scaler*  src1,      src2,       dst
           reg/lit   freg/flit   freg

Description: Multiplies src2 by 2 to the power of src1 and stores the result in dst. The src1 operand is an integer; whereas, src2 and dst are reals.

For the scalerl instruction, if the src2 or dst operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>Src1</th>
<th>-N</th>
<th>0</th>
<th>+ N</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
</tr>
<tr>
<td>-F</td>
<td>-F</td>
<td>-F</td>
<td>-F</td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td>-0</td>
<td>-0</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>+F</td>
<td>+F</td>
<td>+F</td>
<td>+F</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Notes:
F Means finite-real number.
N Means integer.

In most cases, only the exponent is changed and the mantissa (fraction) remains unchanged. However, when the src1 operand is a denormalized value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source’s mantissa.
scaler, scalerl

Refer to the sections titled "Floating Overflow Exception" and "Floating Underflow Exception" in Chapter 12 for further discussion of how overflow and underflow are handled.

Action: \[ dst \leftarrow src2 \times (2^{src1}) \]

Faults:
- STANDARD: Refer to the discussion of faults at the beginning of this chapter.
- Floating Reserved Encoding: One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- Floating Overflow: Result is too large for destination format.
- Floating Underflow: Result is too small for destination format.
- Floating Zero Divide: The \( src1 \) operand is 0.
- Floating Invalid Operation: One or more operands are an SNaN value.
- Floating Inexact: Result cannot be represented exactly in destination format.

Example:

```assembly
scalerl g6, g2, fp0
# fp0 \leftarrow g2, g3 \times 2^{g6}
```

Opcode:

- scaler: 677 REG
- scalerl: 676 REG

See Also: mulr
scanbit

Mnemonic: scanbit  Scan For Bit

Format:    

<table>
<thead>
<tr>
<th>src</th>
<th>dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg/lit</td>
<td>reg</td>
</tr>
</tbody>
</table>

Description: Searches the src value for the most-significant set bit (1 bit). If a most-significant 1 bit is found, its bit number is stored in dst and the condition code is set to 010₂. If the src value is zero, all 1’s are stored in dst and the condition code is set to 000₂.

Action: 

\[
dst \leftarrow 16\#FFFFFFFF\#; \\
AC.cc \leftarrow 2\#000\#; \\
\text{for } i \text{ in } 31..0 \text{ reverse loop} \\
\quad \text{if } (src \text{ and } 2^i) \neq 0 \\
\quad \quad \text{then} \\
\quad \quad \quad dst \leftarrow i; \\
\quad \quad \quad AC.cc \leftarrow 2\#010\#; \\
\quad \quad \quad \text{exit}; \\
\quad \text{end if}; \\
\end{align*}

end loop;

Faults: STANDARD

Example:  

```
# assume g8 is nonzero  
scanbit g8, g10  
# g10 \leftarrow bit number of  
# most-significant set bit  
# in g8; AC.cc \leftarrow 2\#010\#
```

Opcode:    

scanbit 641  REG

See Also:  spanbit
scanbyte

Mnemonic:      scanbyte    Scan Byte Equal

Format:        scanbyte    src1,    src2
                reg/lit     reg/lit

Description:   Performs a byte-by-byte comparison of src1 and src2 and sets the condition code to 2#010# if any two corresponding bytes are equal. If no corresponding bytes are equal, the condition code is set to 0002.

Action:        if (src1 and 16#000000FF#) = (src2 and 16#000000FF#) or
                (src1 and 16#0000FF00#) = (src2 and 16#0000FF00#) or
                (src1 and 16#00FF0000#) = (src2 and 16#00FF0000#) or
                (src1 and 16#FF000000#) = (src2 and 16#FF000000#)
                then AC.cc ← 2#010#;
                else AC.cc ← 2#000#;
                endif;

Faults:        STANDARD

Example:       # assume r9 = 0x11AB1100
                scanbyte 0x00AB0011, r9
                # AC.cc ← 2#010#

Opcode:        scanbyte    5AC    REG
Mnemonic: setbit

Format: setbit bitpos, src, dst
         reg/lit   reg/lit   reg

Description: Copies the src value to dst with one bit set. The bitpos operand specifies the bit to be set.

Action: dst ← src \text{ or } 2^{\text{bitpos mod 32}};

Faults: STANDARD

Example: setbit 15, r9, r1
         # r1 ← r9 with bit 15 set

Opcode: setbit 583 REG

See Also: alterbit, chkbit, clrbit, notbit,
SHIFT

Mnemonic:  shlo Shift Left Ordinal
            shro Shift Right Ordinal
            shli Shift Left Integer
            shri Shift Right Integer
            shrdi Shift Right Dividing Integer

Format:  \( sh^* \)  \( len, \quad src, \quad dst \)
         \( reg/lit \quad .\quad reg/lit \quad .\quad reg \)

Description: Shifts \( src \) left or right by the number of digits indicated with the \( len \) operand and stores the result in \( dst \). This operation (with the exception of the shri instruction, as described below) is equivalent to multiplying (shift left) or dividing (shift right) the \( src \) value by \( 2^{len} \).

The shri instruction performs a conventional arithmetic right shift, which, when used as a divide, produces an incorrect quotient for negative \( src \) values. To get a correct quotient for a negative \( src \) value, use the shrdi instruction, which performs correct rounding of negative results.

Action:  shlo:  if \( len < 32 \)
            then \( dst \leftarrow src \cdot 2^{len} \)
            else \( dst \leftarrow 0 \);
            end if;

            shro:  if \( len < 32 \)
            then \( dst \leftarrow src/2^{len} \)
            else \( dst \leftarrow 0 \);
            end if;

            shli:  \( dst \leftarrow src \cdot 2^{len} \)

            shri:  if \( src \geq 0 \)
            then if \( len < 32 \)
                then \( dst \leftarrow src/2^{len} \)
                else \( dst \leftarrow 0 \);
            else if \( len < 32 \)
                then \( dst \leftarrow (src - 2^{len} + 1)/2^{len} \)
                else \( dst \leftarrow -1 \);
            end if;
            end if;

            shrdi: \( dst \leftarrow src/2^{len} \)
SHIFT

Faults: STANDARD, Integer Overflow

Example: shli 13, g4, r6
# g6 ← g4 shifted left 13 bits

Opcode: shlo  59C   REG
        shro  598   REG
        shli  59E   REG
        shri  59B   REG
        shrdi 59A   REG

See Also: divi, muli, rotate
**sinr, sinrl**

Mnemonics:  
- `sinr` Sine Real  
- `sinrl` Sine Long Real

Format:  
- `sinr* src, dst`  
- `freg/flit`  
- `freg`

Description:  
Calculates the sine of `src` and stores the result in `dst`. The `src` value is an angle given in radians. The resulting `dst` value is in the range -1 to +1, inclusive.

For the `sinrl` instruction, if the `src` or `dst` operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the sine of various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>Src</th>
<th>Dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>*</td>
</tr>
<tr>
<td>-F</td>
<td>-1 to +1</td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>+F</td>
<td>-1 to +1</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Notes:  
- F Means finite-real number  
- * Indicates floating invalid-operation exception

In the trigonmetric instructions, the 80960KB uses a value for π with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. The section in Chapter 12 titled "Pi" gives this π value, along with some suggestions for representing this value in a program.

Action:  

```
dst ← sin (src);```

3-224
**Faults:**

STANDARD

Refer to the discussion of faults at the beginning of this chapter.

Floating Reserved Encoding

One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Underflow

Result is too small for destination format.

Floating Invalid Operation

The *src* operand is $\infty$.

One or more operands is an SNaN value.

Floating Inexact

Result cannot be represented exactly in destination format.

**Example:**

```
sinrl g6, g0
# sine of value in g6,g7
# is stored in g0,g1
```

**Opcode:**

```
sinr 68C  REG
sinrl 69C  REG
```

**See Also:**

```
cosr, tanr
```
spanbit

Mnemonic: spanbit  Span Over Bit

Format: spanbit  src,  dst
          reg/lit  reg

Description: Searches the src value for the most-significant clear bit (0 bit). If a most-significant 0 bit is found, its bit number is stored in dst and the condition code is set to 010₂. If the src value is all 1’s, all 1’s are stored in dst and the condition code is set to 000₂.

Action:  dst ← 16#FFFFFFFF#
         AC.cc ← 2#000#
         for i in 31..0 reverse loop
            if (src and 2^i) = 0
               then
                  dst ← i;
                  AC.cc ← 2#010#;
                  exit;
               end if;
         end loop;

Faults: STANDARD

Example:  # assume r2 is not 16#FFFFFFFF#
          spanbit r2 r9
          # r9 ← bit number of
          # most-significant clear bit
          # in r2; AC.cc ← 2#010#

Opcode:  spanbit  640  REG

See Also: scanbit
Mnemonic:  sqrtr  Square Root Real
          sqrtrl  Square Root Long Real

Format:  sqrtr*  src, dst
         freg/flit  freg

Description: Calculates the square root of src and stores it in dst.

For the sqrtrl instruction, if the src or dst operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>Src</th>
<th>Dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>*</td>
</tr>
<tr>
<td>-F</td>
<td>*</td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>+F</td>
<td>+F</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Notes:
- F  Means finite-real number
- *  Indicates floating invalid-operation exception

With these instructions, it is not possible to raise a floating overflow or floating underflow fault unless the src operand is in a floating-point register and the dst operand is not.

Action:  dst ← sqrt (src);
sqrtr, sqrtrl

Faults:  
STANDARD  
Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding  
One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow  
Result is too large for destination format.
Floating Underflow  
Result is too small for destination format.
Floating Invalid Operation  
The src operand is less than -0.
The src operand is an SNaN value.
Floating Inexact  
Result cannot be represented exactly in destination format.

Example:  
sqrtrl g6, fp0  
# fp0 ← sqrt of g6, g7

Opcode:  
sqrtr  
688  REG
sqrtrl  
698  REG
STORE

Mnemonic:  
st  Store  
stob  Store Ordinal Byte  
stos  Store Ordinal Short  
stib  Store Integer Byte  
stis  Store Integer Short  
stl  Store Long  
stt  Store Triple  
stq  Store Quad  

Format:  
\texttt{st* } & src, & dst  
\texttt{reg/lit} & mem  

Description:  Copies a byte or string of bytes from a register or group of registers to memory. The \texttt{src} operand specifies a register or the first (lowest numbered) register of successive registers.

The \texttt{dst} operand specifies the address of the memory location where the byte or the first byte of a string of bytes is to be stored. The full range of addressing modes may be used in specifying \texttt{dst}. (Refer to Chapter 5 for a complete discussion of the addressing modes available with memory-type operands.)

The \texttt{stob} and \texttt{stib}, and \texttt{stos} and \texttt{stis} instructions store a byte and half word, respectively, from the low order bytes of the \texttt{src} register. The \texttt{st}, \texttt{stl}, \texttt{stt}, and \texttt{stq} instructions copy 4, 8, 12, and 16 bytes, respectively, from successive registers to memory.

For the \texttt{stl} instruction, \texttt{dst} must specify an even numbered register (e.g., g0, g2, ..., g12). For the \texttt{stt} and \texttt{stq} instructions, \texttt{dst} must specify a register number that is a multiple of four (e.g., g0, g4, g8).

Action:  memory (\texttt{dst}) ← \texttt{src};

Faults:  STANDARD, Integer Overflow Fault (\texttt{stib} and \texttt{stis} instructions only)

Example:  
\texttt{st g2, 1256 (g6)}
\texttt{# word beginning at offset}
\texttt{# 1256 + (g6) ← g2}
### STORE

**Opcode:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>st</code></td>
<td>92</td>
<td>MEM</td>
</tr>
<tr>
<td><code>stob</code></td>
<td>82</td>
<td>MEM</td>
</tr>
<tr>
<td><code>stos</code></td>
<td>8A</td>
<td>MEM</td>
</tr>
<tr>
<td><code>stib</code></td>
<td>C2</td>
<td>MEM</td>
</tr>
<tr>
<td><code>stis</code></td>
<td>CA</td>
<td>MEM</td>
</tr>
<tr>
<td><code>stl</code></td>
<td>9A</td>
<td>MEM</td>
</tr>
<tr>
<td><code>stt</code></td>
<td>A2</td>
<td>MEM</td>
</tr>
<tr>
<td><code>stq</code></td>
<td>B2</td>
<td>MEM</td>
</tr>
</tbody>
</table>

**See Also:** LOAD, MOVE
Mnemonic: subc
Subtract Ordinal With Carry

Format: subc src1, src2, dst
         reg/lit    reg/lit    reg

Description: Subtracts (src1 - 1) from src2, adds bit 1 of the condition code (used here as a carry bit), and stores the result in dst. If the ordinal subtraction results in a carry, bit 1 of the condition code is set.

This instruction can also be used for integer subtraction. Here, if integer subtraction results in an overflow, bit 0 of the condition code is set.

The subc instruction does not distinguish between ordinals and integers: it sets bits 0 and 1 of the condition code regardless of the data type.

Action: # Let the value of the condition code be xCx.
         dst ← src2 - (src1 - 1) + C;
         AC.cc ← 2#0CV#;
         # C is carry from ordinal subtraction.
         # V is 1 if integer subtraction would have generated
         # an overflow.

Faults: STANDARD

Example: subc g5, g6, g7
         # g7 ← g6 - (g5 - 1)
         # + Carry Bit

Opcode: subc 5B2 REG

See Also: addc
subi, subo

Mnemonic:  
subi  Subtract Integer  
subo  Subtract Ordinal

Format:  
\[ \text{sub}^* \quad \text{src1,} \quad \text{src2,} \quad \text{dst} \]  
\[ \text{reg/lit} \quad \text{reg/lit} \quad \text{reg} \]

Description:  Subtract src1 from src2 and stores the result in dst. The binary results from these two instructions are identical. The only difference is that subi can signal an integer overflow.

Action:  
\[ \text{dst} \leftarrow \text{src2} - \text{src1}; \]

Faults:  
STANDARD, Integer Overflow (subi instruction only)

Example:  
\[ \text{subi g6, g9, g12} \quad \# \text{g12} \leftarrow \text{g9} - \text{g6} \]

Opcode:  
\[ \text{subi} \quad 593 \quad \text{REG} \]  
\[ \text{subo} \quad 592 \quad \text{REG} \]

See Also:  
addi, addr, subc, subr
**Mnemonic:**  
subr  Subtract Real  
subrl Subtract Long Real  

**Format:**  
subr* srcl, src2, dst  
freg/flit freg/flit freg  

**Description:**  
Subtracts src1 from src2 and stores the result in dst.

For the subrl instruction, if the src1, src2, or dst operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when subtracting various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>Src1</th>
<th>-∞</th>
<th>-F</th>
<th>-0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>*</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td>NaN</td>
</tr>
<tr>
<td>-F</td>
<td>+∞</td>
<td>±F or ±0 src2</td>
<td>src2</td>
<td>-F</td>
<td>-∞</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+∞</td>
<td>src1</td>
<td>±0</td>
<td>-0</td>
<td>src1</td>
<td>-∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>+∞</td>
<td>src1</td>
<td>+0</td>
<td>±0</td>
<td>src1</td>
<td>-∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>+∞</td>
<td>+F</td>
<td>src2</td>
<td>src2</td>
<td>±F or ±0</td>
<td>-∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>*</td>
<td>NaN</td>
</tr>
</tbody>
</table>

| NaN    | NaN | NaN | NaN | NaN | NaN | NaN | NaN   |

Notes:  
F Means finite-real number.  
* Indicates floating invalid-operation exception.

When the difference between two operands of like sign is zero, the result is +0, except for the round toward -∞ mode, in which case the result is -0. This instruction also guarantees that +0 - (-0) = +0, and that -0 - (+0) = -0.

When one source operand is ∞, the result is ∞ of the expected sign. If both source operands are ∞ of the same sign, an invalid-operation exception is raised.
subr, subrl

**Action:**

\[ \text{dst} \leftarrow \text{src2} - \text{src1}; \]

**Faults:**

- **STANDARD**
- **Floating Reserved Encoding**

Refer to the discussion of faults at the beginning of this chapter.

One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- **Floating Overflow**
- **Floating Underflow**
- **Floating Invalid Operation**
- **Floating Inexact**

Result is too large for destination format.

Result is too small for destination format.

Source operands are infinities of like sign.

One or more operands are an SNaN value.

Result cannot be represented exactly in destination format.

**Example:**

```
subrl g6, fp0, fpl
# fpl \leftarrow \text{fp0} - g6,g7
```

**Opcode:**

- `subr` 78D  REG
- `subrl` 79D  REG

**See Also:**

`subi`, `subc`, `addr`
Mnemonic: syncf   Synchronize Faults

Format: syncf

Description: Waits for any faults to be generated associated with any prior uncompleted instructions.

Action: if arithmetic_controls.nif
         then;
         else wait until no imprecise faults can occur
         associated with any uncompleted instructions;
         end if;

Faults: STANDARD

Example:  ld xyz, g6
          addi r6, r8, r8
          syncf
          and g6, 0xFFFF, g8
          # the syncf instruction insures that any faults
          # that may occur during the execution of the
          # ld and addi instructions occur before the
          # and instruction is executed

Opcode: syncf 66F REG

See Also: mark, fmark
# synld

**Mnemonic:** synld  Synchronous Load

**Format:**

```
synld    src,     dst
          reg     reg
          addr    addr
```

**Description:** Copies a word from the memory location specified with `src` into `dst` and waits for the completion of all memory operations, including those initiated prior to the `synld` instruction. When the load has been successfully completed, the condition code is set to `2#010#`.

The primary function of this instruction is for reading IAC messages, the IAC Message Control word, or the IAC Interrupt Control Register. However, this instruction is not restricted to IAC applications. It may be used when it is important to guarantee the completion of the load operation before proceeding or to avoid a bad-access fault.

The setting of the condition code indicates whether or not the load was completed successfully. If the load operation results in a bad access condition (e.g., reading an AP-bus interconnect register), the condition code is set to `000_2`, but the bad-access fault is not raised.

**Action:**

```plaintext
if PRCB.addressing_mode = physical
    then tempa ← src;
    else tempa ← physical_address (src);
end if;
tempa ← tempa and 16#FFFFFFFC#; # force alignment
if tempa = 16#FF000004#
    then dst ← interrupt_control_reg;
        AC.cc ← 2#010#;
    else dst ← memory (tempa);
        if bad_access
            then AC.cc ← 2#000#;
            else AC.cc ← 2#010#;
        end if;
end if;
```

**Faults:** STANDARD
Example:

```
lda 16#FF000010#, g8
synld g8, g9  # g9 ← word from IAC
              # message buffer;
              # AC.cc = 2#010#
```

 Opcode:       synld     615     REG

See Also:      synmov
**synmov, synmovl, synmovq**

**Mnemonic:**
- `synmov`  Synchronous Move
- `synmovl`  Synchronous Move Long
- `synmovq`  Synchronous Move Quad

**Format:**
```
synmov*  dst, src
    reg    reg
    addr   addr
```

**Description:**
Copies 1 (`synmov`), 2 (`synmovl`), or 4 (`synmovq`) words from the memory location specified with `src` to the memory location specified with `dst` and waits for the completion of all memory operations, including those initiated prior to this instruction. When the move has been successfully completed, the condition code is set to `010_2`.

The `src` and `dst` operands specify the address of the first (lowest address) word. These addresses should be for word boundaries (`synmov`), double-word boundaries (`synmovl`), or quad-word boundaries (`synmovq`). If not, the processor forces alignment to these boundaries.

The primary function of these instructions is for sending IAC messages. However, this instruction is not restricted to IAC applications. It may be used when it is important to guarantee the completion of the move operation before proceeding or to avoid a Bad Access Fault.

The setting of the condition code indicates whether or not the move was completed successfully. If the move operation results in a bad access condition (e.g., sending an IAC message to a non-existent agent on the AP-bus), the condition code is set to `000_2`, but the Bad Access Fault is not raised.

Address `FF000010_{16}` is used to send an IAC message to the processor upon which the instruction is executed. Refer to Chapter 11 for further information about sending internal IAC messages.
Action: synmov:

if PRCB.addressing_mode = physical
    then tempa ← dst;
    # dst is used as a physical address
else tempa ← physical_address (dst);
    # dst translated into a physical address
end if;
tempa ← tempa and 16#FFFFFFFC#
    # force alignment
if tempa = 16#FF000004#
    then interrupt_control_reg ← memory (src)
        AC.cc ← 2#010#;
    else temp ← memory (src);
        memory (tempa) ← temp;
        # write operations into memory (tempa) are
        # interpreted as noncacheable
        wait for completion;
        if bad_access
            then AC.cc ← 2#000#;
            else AC.cc ← 2#010#;
        end if;
    end if;
end if;
synmovl:

if PRCB.addressing_mode = physical
    then tempa ← dst;
    # dst is used as a physical address
else tempa ← physical_address (dst);
    # dst translated into as a physical address
end if;
tempa ← tempa and 16#FFFFFFF8#
    # force alignment
temp ← memory (src);
memory (tempa) ← temp;
    # write operations into memory (tempa) are interpreted
    # as noncacheable
    wait for completion;
if bad_access
    then AC.cc ← 2#000#;
    else AC.cc ← 2#010#;
end if;
synmov, synmovl, synmovq

**synmovq:**

```plaintext
if PRCB.addressing_mode = physical
  then tempa ← dst;
  # dst is used as a physical address
else tempa ← physical_address (dst);
  # dst is translated into as a physical address
end if;
tempa ← tempa and 16#FFFFFFF0#; # force alignment
temp ← memory (src);
if tempa = 16#FF000010#
  then AC.cc ← 2#010#;
      use temp as a received iac message;
else memory (tempa) ← temp;
  # write operations into memory (tempa) are interpreted
  # as noncacheable
  wait for completion;
  if bad_access
    then AC.cc ← 2#000#;
    else AC.cc ← 2#010#;
  end if;
end if;
```

**Faults:** STANDARD

**Example:**

```
lda 16#FF000010#, g7
  # g7 ← 16#FF000010
synmovq g7, g8
  # g7 ← IAC message from g8
```

**Opcode:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>synmov</td>
<td>600</td>
<td>REG</td>
</tr>
<tr>
<td>synmovl</td>
<td>601</td>
<td>REG</td>
</tr>
<tr>
<td>synmovq</td>
<td>602</td>
<td>REG</td>
</tr>
</tbody>
</table>

**See Also:** synld
Mnemonics: \texttt{tanr} \hspace{1em} \texttt{Tangent Real} \\
\texttt{tanrl} \hspace{1em} \texttt{Tangent Long Real}

Format: \texttt{tanr*} \hspace{1em} src, \hspace{1em} dst \\
\hspace{2em} \hspace{1em} freg/flit \hspace{1em} freg

Description: Calculates the tangent of \textit{src} and stores the result in \textit{dst}. The \textit{src} value is an angle given in radians. The resulting \textit{dst} value is in the range of $-\infty$ to $+\infty$, inclusive; a result of $-\infty$ or $+\infty$ will result in a floating invalid-operation exception being signaled.

For the \texttt{tanrl} instruction, if the \textit{src} or \textit{dst} operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the tangent of various classes of numbers, assuming that neither overflow nor underflow occurs.

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{Src} & \textbf{Dst} \\
\hline
$-\infty$ & * \\
-F & -F to +F \\
-0 & -0 \\
+0 & +0 \\
+F & -F to +F \\
$+\infty$ & * \\
NaN & NaN \\
\hline
\end{tabular}
\end{center}

Notes:
\begin{itemize}
\item \textit{F} \hspace{1em} \text{Means finite-real number}
\item * \hspace{1em} \text{Indicates floating invalid-operation exception}
\end{itemize}

If the source operand is a finite value, the result will be finite, unless the \textit{src} operand is in a floating-point register and the \textit{dst} operand is not.

In the trigonmetric instructions, the 80960KB uses a value for $\pi$ with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. The section in Chapter 12 titled "Pi" gives this $\pi$ value, along with some suggestions for representing this value in a program.
**tanh, tanhl**

**Action:** \( dst \leftarrow \text{tangent} (src); \)

**Faults:**
- STANDARD
- Floating Reserved Encoding

Refer to the discussion of faults at the beginning of this chapter.

One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

- **Floating Overflow**
  - Result is too large for destination format.

- **Floating Underflow**
  - Result is too small for destination format.

- **Floating Invalid Operation**
  - The src operand is \( \infty \).
  - One or more operands are an SNaN value.

- **Floating Inexact**
  - Result cannot be represented exactly in destination format.

**Example:**
```
tanh g4, fp0  # tangent of value in g4,g5 is stored in fp0
```

**Opcode:**
- `tanh` \(68E\) REG
- `tanhl` \(69E\) REG

**See Also:** cosh, sinh
**TEST**

**Mnemonic:**
- **test:** Test For Equal
- **testne:** Test For Not Equal
- **testl:** Test For Less
- **testle:** Test For Less or Equal
- **testg:** Test For Greater
- **testge:** Test For Greater or Equal
- **testo:** Test For Ordered
- **testno:** Test For Unordered

**Format:**
```
test* dst
reg
```

**Description:** Stores a true (1) in **dst** if the logical AND of the condition code and the mask-part of the opcode is not zero. Otherwise, the instruction stores a false (0) in **dst**.

The following table shows the condition-code mask for each instruction:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mask</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>testno</td>
<td>000</td>
<td>Unordered</td>
</tr>
<tr>
<td>testg</td>
<td>001</td>
<td>Greater</td>
</tr>
<tr>
<td>teste</td>
<td>010</td>
<td>Equal</td>
</tr>
<tr>
<td>testge</td>
<td>011</td>
<td>Greater or equal</td>
</tr>
<tr>
<td>testl</td>
<td>100</td>
<td>Less</td>
</tr>
<tr>
<td>testne</td>
<td>101</td>
<td>Not equal</td>
</tr>
<tr>
<td>testle</td>
<td>110</td>
<td>Less or equal</td>
</tr>
<tr>
<td>testo</td>
<td>111</td>
<td>Ordered</td>
</tr>
</tbody>
</table>

For the **testno** instruction (Unordered), a true is stored if the condition code is 2#000#; otherwise a false is stored.
TEST

Action: For All Instructions Except testno:

\[
\text{if (mask and AC.cc) } \neq 2\#000# \\
\quad \text{then } dst \leftarrow 1; \ # dst \text{ set for true} \\
\quad \text{else } dst \leftarrow 0; \ # dst \text{ set for false} \\
\text{end if;}
\]

\[
\text{testno:}
\]

\[
\text{if AC.cc } = 2\#000# \\
\quad \text{then } dst \leftarrow 1; \ # dst \text{ set for true} \\
\quad \text{else } dst \leftarrow 0; \ # dst \text{ set for false} \\
\text{end if;}
\]

Faults: STANDARD

Example: \# assume AC.cc = 2\#100#

testl g9 \# g9 \leftarrow 16\#00000001#

Opcode:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>teste</td>
<td>22</td>
<td>COBR</td>
</tr>
<tr>
<td>testne</td>
<td>25</td>
<td>COBR</td>
</tr>
<tr>
<td>testl</td>
<td>24</td>
<td>COBR</td>
</tr>
<tr>
<td>testle</td>
<td>26</td>
<td>COBR</td>
</tr>
<tr>
<td>testg</td>
<td>21</td>
<td>COBR</td>
</tr>
<tr>
<td>testge</td>
<td>23</td>
<td>COBR</td>
</tr>
<tr>
<td>testo</td>
<td>27</td>
<td>COBR</td>
</tr>
<tr>
<td>testno</td>
<td>20</td>
<td>COBR</td>
</tr>
</tbody>
</table>

See Also: cmpi, cmpdeci, cmpinci
xnor, xor

Mnemonic:  

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>xnor</th>
<th>xor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exclusive Nor</td>
<td>Exclusive Or</td>
</tr>
</tbody>
</table>

Format:  

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>xnor</td>
<td>src1, src2, dst</td>
</tr>
<tr>
<td></td>
<td>reg/lit reg/lit reg</td>
</tr>
<tr>
<td>xor</td>
<td>src1, src2, dst</td>
</tr>
<tr>
<td></td>
<td>reg/lit reg/lit reg</td>
</tr>
</tbody>
</table>

Description: Performs a bitwise XNOR (xnor instruction) or XOR (xor instruction) operation on the src2 and src1 values and stores the result in dst.

Action:  

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>xnor</td>
<td>dst ← not (src2 or src1) or (src2 and src1);</td>
</tr>
<tr>
<td>xor</td>
<td>dst ← (src2 or src1) and not (src2 and src1);</td>
</tr>
</tbody>
</table>

Faults: STANDARD

Example:  

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>xnor</td>
<td>xnor r3, r9, r12  # r12 ← r9 XNOR r3</td>
</tr>
<tr>
<td>xor</td>
<td>xor g1, g7, g4   # g4 ← g7 XOR g1</td>
</tr>
</tbody>
</table>

Opcode:  

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>xnor</td>
<td>589    REG</td>
</tr>
<tr>
<td>xor</td>
<td>586    REG</td>
</tr>
</tbody>
</table>

See Also: and, andnot, nand, nor, not, notand, notor, or, ornot
11.0 FLOATING-POINT OPERATION

This section describes the floating-point processing capabilities of the 80960KB processor. The subjects discussed include the real number data types, the execution environment for floating-point operations, the floating-point instructions, and fault and exception handling.

11.1 INTRODUCING THE 80960KB FLOATING-POINT ARCHITECTURE

The floating-point architecture used in the 80960KB processor is designed to allow a convenient implementation of the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic. This hardware architecture, along with a small amount of software support, conforms to the IEEE standard and provides support for the following data structures and operations:

- Real (32-bit), long real (64-bit), and extended real (80-bit) floating-point number formats.
- Add, subtract, multiply, divide, square root, remainder, and compare operations
- Conversion between integer and floating-point formats
- Conversion between different floating-point formats
- Handling of floating-point exceptions, including non-numbers (NaNs)

The software to support the 80960KB floating-point architecture is needed primarily to handle conversions between real numbers and decimal strings.

In addition, the 80960KB floating-point architecture supports several functions that go beyond the IEEE standard. These functions fall into two categories:

- functions recommended in the appendix to the IEEE standard, such as copy sign and classify, and
- commonly used transcendental functions, including trigonometric, logarithmic, and exponential functions.

11.2 REAL NUMBERS AND FLOATING-POINT FORMAT

This section provides an introduction to real numbers and how they are represented in floating-point format. Readers who are already familiar with numeric processing techniques and the IEEE standard may wish to skip this section.

11.2.1 Real Number System

As shown at the top of Figure 23, the real-number system comprises the continuum of real numbers from minus infinity (-\infty) to plus infinity (+\infty).
Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in real-number calculations. As shown at the bottom of Figure 23, the subset of real numbers that a particular processor supports represents an approximation of the real number system. The range and precision of this real-number subset is determined by the format that the processor uses to represent real numbers.

11.2.2 Floating-Point Format

To increase the speed and efficiency of real number computations, computers or numeric processors typically represent real numbers in a binary floating-point format. In this format, a real number has three parts: a sign, a significand, and an exponent. Figure 24 shows the binary floating-point format that the processor uses. This format conforms to the IEEE standard.

The sign is a binary value that indicates whether the number is positive (0) or negative (1). The significand has two parts: a one-bit binary integer (also referred to as the j-bit) and a binary fraction. The j-bit is often not represented, but instead is an implied value. The exponent is a binary integer that represents the base-2 power that the significand is raised to.
Table 16 shows how the real number 201.187 (in ordinary decimal format) is stored in floating-point format. The table lists a progression of real number notations that leads to the format that the 80960KB processor uses. In this format, the binary real number is normalized and the exponent is biased.

**Table 16. Real Number Notation**

<table>
<thead>
<tr>
<th>NOTATION</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDINARY DECIMAL</td>
<td>201.187</td>
</tr>
<tr>
<td>SCIENTIFIC DECIMAL</td>
<td>2.01187E102</td>
</tr>
<tr>
<td>SCIENTIFIC BINARY</td>
<td>1.1001001001011111E211</td>
</tr>
<tr>
<td>SCIENTIFIC BINARY (BIASED EXPONENT)</td>
<td>1.1001001001011111E210000110</td>
</tr>
<tr>
<td>32-BIT FLOATING-POINT FORMAT (NORMALIZED)</td>
<td></td>
</tr>
<tr>
<td>SIGN</td>
<td>0</td>
</tr>
<tr>
<td>BIASED EXPONENT</td>
<td>10000110</td>
</tr>
<tr>
<td>SIGNIFICAND</td>
<td>1001001001011111</td>
</tr>
<tr>
<td>1. (IMPLIED)</td>
<td>11.2.3 Normalized Numbers</td>
</tr>
</tbody>
</table>

In most cases, the processor represents real numbers in normalized form. This means that except for zero, the significand is always made up of an integer of 1 and a fraction as follows:

1.ffe...ff
For values less than 1, leading zeros are eliminated. (For each leading zero eliminated, the exponent is decremented by one.)

Representing numbers in normalized form maximizes the number of significant digits that can be accommodated in a significand of a given width. To summarize, a normalized real number consists of a normalized significand that represents a real number between 1 and 2 and an exponent that gives the number’s binary point.

11.2.4 Biased Exponent

The processor represents exponents in a biased form. This means that a constant is added to the actual exponent so that the biased exponent is always a positive number. The value of the biasing constant depends on the number of bits available for representing exponents in the floating-point format being used. The biasing constant is chosen so that the smallest normalized number can be reciprocated without overflow.

11.2.5 Real Number and Non-Number Encodings

The real numbers that are encoded in the floating-point format described above are generally divided into three classes: +0, +nonzero-finite number, and +∞. Encodings for non-numbers (NaNs) are also defined. The term NaN stands for “Not a Number.” Figure 25 shows how the encodings for these numbers and non-numbers fit into the real number continuum. The encodings shown here are for the IEEE single-precision (32-bit) format, where the term “s” indicates the sign bit, “e” the biased exponent, and “f” the fraction. (The exponent values are given in decimal.)

11.2.6 Signed Zeros

Zero can be represented as a +0 or a -0 depending on the sign bit. Both encodings are equal in value. The sign of a zero result depends on the operation being performed and the rounding mode being used. Signed zeros have been provided to aid in implementing interval arithmetic. The sign of a zero may indicate the direction from which underflow occurred, or it may indicate the sign of an ∞ that has been reciprocated.

11.2.7 Signed, Nonzero, Finite Values

The class of signed, nonzero, finite values is divided into two groups: normalized and denormalized. The normalized finite numbers comprise all the nonzero finite values that can be encoded in a normalized real number format from zero to ∞. In the 32-bit form shown in Figure 25, this group of numbers includes all the numbers with biased exponents ranging from 1 to 254 (unbiased, the exponent range is from -126 to +127).
11.2.8 Denormalized Numbers

When real numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros.

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called denormalized numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization causes a loss of precision (the number of significant bits in the fraction is reduced by the leading zeros).

When performing normalized floating-point computations, a processor normally operates on normalized numbers and produces normalized numbers as results. Denormalized numbers represent an underflow condition.

A denormalized number is computed through a technique called gradual underflow. Table 17 gives an example of gradual underflow in the denormalization process. Here the 32-bit format is being used, so the minimum exponent (unbiased) is $-126_{10}$. The true result in this example requires an
exponent of \(-129\) in order to have a normalized number. Since \(-129\) is beyond the allowable exponent range, the result is denormalized by inserting leading zeros until the minimum exponent of \(-126\) is reached.

Table 17. Denormalization Process

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sign</th>
<th>Exponent*</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Result</td>
<td>0</td>
<td>-129</td>
<td>1.01011100...00</td>
</tr>
<tr>
<td>Denormalize</td>
<td>0</td>
<td>-128</td>
<td>0.101011100...00</td>
</tr>
<tr>
<td>Denormalize</td>
<td>0</td>
<td>-127</td>
<td>0.0101011100...00</td>
</tr>
<tr>
<td>Denormalize</td>
<td>0</td>
<td>-126</td>
<td>0.00101011100...00</td>
</tr>
<tr>
<td>Denormal Result</td>
<td>0</td>
<td>-126</td>
<td>0.00101011100...00</td>
</tr>
</tbody>
</table>

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result.

11.2.9 Signed Infinities

The two infinities, \(+\infty\) and \(-\infty\), represent the maximum positive and negative real numbers, respectively, that can be represented in the floating-point format. Infinity is always represented by a zero fraction and the maximum biased exponent allowed in the specified format (e.g., 255\(_{10}\) for the 32-bit format).

Whereas denormalized numbers represent an underflow condition, the two infinity numbers represent the result of an overflow condition. Here, the normalized result of a computation has a biased exponent greater than the largest allowable exponent for the selected result format.

11.2.10 NaNs

Since NaNs are non-numbers, they are not part of the real number line. In Figure 25, the encoding space for NaNs in the 80960KB floating-point format is shown above the ends of the real number line. This space includes any value with the maximum allowable biased exponent and a non-zero fraction. (The sign bit is ignored for NaNs.)

The IEEE standard defines two specific NaN values: a quiet NaN (QNaN) and a signaling NaN (SNaN). A QNaN is a NaN with the most significant fraction bit set; an SNaN is a NaN with the most significant bit clear. QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs signal an invalid-operation exception whenever they appear as operands in arithmetic operations. Exceptions are discussed later in section titled “Exceptions and Fault Handling.”
The section “Operations on NaNs” provides detailed information on how the processor handles NaNs.

### 11.3 REAL DATA TYPES

The processor supports three real-number data formats: real, long real, and extended real. These formats correspond directly to the single-precision, double-precision, and double-extended precision formats in the IEEE standard. Figure 26 shows these data formats and gives the resolution that each provides.

For the real and long-real formats, only the fraction is given for the significand. The integer is assumed to be 1 for all numbers except 0 and denormalized finite numbers.

For the extended-real format, the integer is contained in bit 63, and the most-significant fraction bit is bit 62. Here, the integer is explicitly set to 1 for normalized numbers, infinities, and NaNs, and to 0 for zero and denormalized numbers.

Table 18 shows the encodings for all the classes of real numbers (i.e., zero, denormalized finite, normalized finite, and \( \infty \)) and NaNs, for each of the three real data-types.
### Table 18. Real Numbers and NaN Encodings

<table>
<thead>
<tr>
<th>Class</th>
<th>Sign</th>
<th>Biased Exponent</th>
<th>Integer</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ ∞</td>
<td>0</td>
<td>11...11</td>
<td>1</td>
<td>00...00</td>
</tr>
<tr>
<td>+ NORMALS</td>
<td>0</td>
<td>11...10</td>
<td>1</td>
<td>11...11</td>
</tr>
<tr>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>00...01</td>
<td>1</td>
<td>00...00</td>
</tr>
<tr>
<td>+ DENORMALS</td>
<td>0</td>
<td>00...00</td>
<td>0</td>
<td>11...11</td>
</tr>
<tr>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>00...00</td>
<td>0</td>
<td>00...01</td>
</tr>
<tr>
<td>+ ZERO</td>
<td>0</td>
<td>00...00</td>
<td>0</td>
<td>00...00</td>
</tr>
<tr>
<td>- ZERO</td>
<td>1</td>
<td>00...00</td>
<td>0</td>
<td>00...00</td>
</tr>
<tr>
<td>- DENORMALS</td>
<td>1</td>
<td>00...00</td>
<td>0</td>
<td>00...01</td>
</tr>
<tr>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>00...00</td>
<td>0</td>
<td>11...11</td>
</tr>
<tr>
<td>- NORMALS</td>
<td>1</td>
<td>00...01</td>
<td>1</td>
<td>00...00</td>
</tr>
<tr>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11...10</td>
<td>1</td>
<td>11...11</td>
</tr>
<tr>
<td>- ∞</td>
<td>1</td>
<td>11...11</td>
<td>1</td>
<td>00...00</td>
</tr>
<tr>
<td>NaN</td>
<td>X</td>
<td>11...11</td>
<td>1</td>
<td>0X...XX</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>11...11</td>
<td>1</td>
<td>1X...XX</td>
</tr>
</tbody>
</table>

**REAL:** 8 BITS 23 BITS

**LONG REAL:** 11 BITS 52 BITS

**EXTENDED REAL:** 15 BITS 63 BITS

**Notes:**
1. Integer is implied for real and long real formats and is not stored.
2. Fraction for SNaN must be non-zero.
11.4 EXECUTION ENVIRONMENT FOR FLOATING-POINT OPERATIONS

An important feature of the 80960KB processor is that the floating-point processing capabilities have been integrated into the execution environment of the processor. Operations on floating-point numbers are carried out using the same registers that are used for ordinals and integers. In addition, four floating-point registers have been provided for extended-precision floating-point arithmetic.

The following sections describe how floating-point operations are handled in the processor's execution environment.

11.4.1 Registers

All of the registers in the processor's execution environment, (i.e., global, local, and floating point) can be used for floating-point operations. When using global or local registers, real values (i.e., 32 bits) are contained in one register; long-real values (i.e., 64 bits) are contained in two successive registers; and extended-real values (i.e., 80 bits) are contained in three successive registers.

Figure 27 shows how the three forms of the real data type are encoded when stored in global and local registers. Note that long-real values must be aligned on even-numbered register boundaries (e.g., g0, g2, ...). Extended-real values must be aligned on register boundaries that are an integral multiple of four (e.g., g0, g4, ...).

Figure 27. Storage of Real Values in Global and Local Registers
Real values in the floating-point registers are always in the extended-real format. When a real or long-real value is moved from global or local registers to floating-point register, the processor automatically reformats it for the extended-real format.

11.4.2 Loading and Storing Floating-Point Values

Floating-point values are loaded from memory into global or local registers using the load (ld), load long (ldl), and load triple (ldt) instructions. Likewise, floating-point values in global or local registers are stored in memory using the store (st), store long (stl), and store triple (stt) instructions.

Loading a floating-point value into a floating-point register requires two steps (two instructions). First, a floating-point value must be loaded from memory into one or more global or local registers. Then, the value must be moved to the floating-point register using a move real (movr), move long-real (movrl), or move extended-real (movre) instruction.

A similar two-step procedure is required to store a value from a floating-point register into memory. The value must first be moved into one or more global or local registers (using a movr, movrl, or movre instruction), then stored in memory.

This two-step method for moving values from memory into floating-point registers and vice versa may seem a little cumbersome; however, in practice it generally is not. Floating-point registers are most often used to store and accumulate intermediate results of computations. The contents of these registers are not normally stored in memory.

For example, the following instruction

\[ \text{divr } r3, r4, fp2 \]

causes the real value in local register r4 to be divided by the value in r3, with the extended-real result stored in floating-point register fp2. Here, a move operation from the local registers to the floating-point registers is not required, since it is implicit in the divide operation.

11.4.3 Moving Floating-Point Values

Either the move instructions (mov, movl, or movt) or the move-real instructions (movr, movrl, or movre) can be used to move real values among global and local registers. The move real instructions are generally used to convert a real value from one format to another or for moving real values between the global or local registers and floating-point registers. The move instructions are used to move real values while keeping them in the same format.

When using the movr and movrl instructions to move floating-point numbers between the global or local registers and the floating-point registers, the processor automatically converts values from real and long-real format, respectively, into the extended-real format and vice versa.
For example, the following instruction

\[
\text{movr g3, fp1}
\]

causes a 32-bit, real value in global register g3 to be converted to 80-bit, extended-real format and placed in floating-point register fp1.

Going the opposite direction, the instruction

\[
\text{movrl fp0, r4}
\]

causes an extended-real value in floating-point register fp0 to be converted to 64-bit, long-real format and placed in local registers r4 and r5.

The movre instruction moves 80-bit, extended-real values between registers, without format conversion. When this instruction is used to move a value from three global or local registers to a floating-point register, the processor extracts the 80-bit value from the three word extended-real format. When moving a value from a floating-point register to global or local registers, the processor inserts the 80-bit value into the three registers in the three-word format.

11.4.4 Arithmetic Controls

The arithmetic controls are used extensively to control the arithmetic and faulting properties of floating-point operations. Table 19 shows the bits in the arithmetic controls that are used in floating-point operations.

The condition code flags are used to indicate the results of comparisons of real numbers, just as they are for integers and ordinals.

The arithmetic status field is used to record results from the classify real (clcsr and clssrl) and remainder real (remr and remrl) instructions. These instructions are discussed later in this section.

The floating-point flags indicate exceptions to floating-point operations. Here, the term exception refers to a potentially undesirable operation (such as dividing a number by zero) or an undesirable result (such as underflow). The flags provide a means of recording the occurrence of specific exceptions.

The floating-point masks provide a method of inhibiting the processor from invoking a fault handler when an exception is detected.

Use of the floating-point flag and mask bits are discussed later in this section in "Exceptions and Fault Handling."
Table 19. Arithmetic Controls Used in Floating-Point Operations

<table>
<thead>
<tr>
<th>Arithmetic Control Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 2</td>
<td>Condition code</td>
</tr>
<tr>
<td>3 - 6</td>
<td>Arithmetic status field</td>
</tr>
<tr>
<td>8</td>
<td>Integer overflow flag</td>
</tr>
<tr>
<td>12</td>
<td>Integer overflow mask</td>
</tr>
<tr>
<td>16</td>
<td>Floating overflow flag</td>
</tr>
<tr>
<td>17</td>
<td>Floating underflow flag</td>
</tr>
<tr>
<td>18</td>
<td>Floating invalid-operation flag</td>
</tr>
<tr>
<td>19</td>
<td>Floating zero-divide flag</td>
</tr>
<tr>
<td>20</td>
<td>Floating inexact flag</td>
</tr>
<tr>
<td>24</td>
<td>Floating overflow mask</td>
</tr>
<tr>
<td>25</td>
<td>Floating underflow mask</td>
</tr>
<tr>
<td>26</td>
<td>Floating invalid-operation mask</td>
</tr>
<tr>
<td>27</td>
<td>Floating zero-divide mask</td>
</tr>
<tr>
<td>28</td>
<td>Floating inexact mask</td>
</tr>
<tr>
<td>29</td>
<td>Normalizing mode flag</td>
</tr>
<tr>
<td>30 - 31</td>
<td>Rounding control</td>
</tr>
</tbody>
</table>

11.4.5 Normalizing Mode

The normalizing-mode flag specifies whether the processor operates in normalizing mode (set) or not (clear).

Normalizing mode is the most common mode of operation. Here, the processor operates on valid floating-point operands, regardless of whether they are normalized or denormalized values.

When the processor is not operating in normalizing mode, it signals a reserved-encoding exception whenever it encounters a denormalized floating-point value as a source operand. In either mode, denormalized numbers are be produced if the underflow exception is masked.

There are no flag or mask bits in the arithmetic controls for this exception. When a reserved-encoding exception is detected, the processor generates a floating reserved-encoding fault and leaves the destination operand unchanged (i.e., no result is stored).

The unnormalized mode of operation is provided to allow unnormalized arithmetic to be simulated with software. Here, a fault handler routine can be used to perform unnormalized arithmetic whenever a reserved-encoding exception is signaled.
11.4.6 Rounding Control

Often the infinitely precise result of an arithmetic operation cannot be encoded exactly in the format of the destination operand. For example, the following value has a 24-bit fraction. The least-significant bit of this fraction (the underlined bit) cannot be encoded exactly in the real (32-bit) format:

\[ 1.0001\ 0000\ 1000\ 0011\ 1001\ 001\E_2 \ 101 \]

The processor must then round the result to one of the following two values:

\[ 1.001\ 0000\ 1000\ 0011\ 1001\ 011\E_2 \ 101 \]
\[ 1.001\ 0000\ 1000\ 0011\ 1001\ 100\E_2 \ 101 \]

A rounded result is called an inexact result. When an inexact result is produced, the floating-point inexact flag bit in the arithmetic controls is set.

The processor rounds results according to the destination format (real, long real, or extended real) and the setting of the rounding-mode flags of the arithmetic controls. Four types of rounding are allowed, as described in Table 20.

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round up (toward +∞)</td>
<td>Rounded result is close to but no less than the infinitely precise result</td>
</tr>
<tr>
<td>Round down (toward -∞)</td>
<td>Rounded result is close to but no greater than the infinitely precise result</td>
</tr>
<tr>
<td>Round toward zero (Truncate)</td>
<td>Rounded result is close to but no greater in absolute value than the infinitely precise result</td>
</tr>
<tr>
<td>Round to nearest (even)</td>
<td>Rounded result is close to the infinitely precise result. If two values are equally close, the result is the even value (i.e., the one with the least-significant bit of zero).</td>
</tr>
</tbody>
</table>

When the infinitely precise result is between the largest positive finite value allowed in a particular format and +∞, the processor rounds the result as shown in Table 21.
Table 21. Rounding of Positive Numbers

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round up (toward +(\infty))</td>
<td>+(\infty)</td>
</tr>
<tr>
<td>Round down (toward -(\infty))</td>
<td>Maximum, positive finite value</td>
</tr>
<tr>
<td>Round toward zero (Truncate)</td>
<td>Maximum, positive finite value</td>
</tr>
<tr>
<td>Round to nearest (even)</td>
<td>+(\infty)</td>
</tr>
</tbody>
</table>

When the infinitely precise result is between the largest negative finite value allowed in a particular format and -\(\infty\), the processor rounds the result as shown in Table 22.

Table 22. Rounding of Negative Numbers

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round up (toward +(\infty))</td>
<td>Maximum, negative finite value</td>
</tr>
<tr>
<td>Round down (toward -(\infty))</td>
<td>-(\infty)</td>
</tr>
<tr>
<td>Round toward zero (Truncate)</td>
<td>Maximum, negative finite value</td>
</tr>
<tr>
<td>Round to nearest (even)</td>
<td>-(\infty)</td>
</tr>
</tbody>
</table>

The rounding modes have no effect on comparison operations, operations that produce exact results, or operations that produce NaN results.

The floating-point instructions allow a result to be stored in a shorter destination than the source operands. For example, the instruction

```
addr fp1, fp2, g5
```

produces a real (32-bit) result from two extended-real (80-bit) source operands. In all such operations, only one rounding error occurs: the error that occurs when rounding the infinitely precise result to the size of the destination format.

Technically, an operation which computes a narrow result from wide operands is in violation of the IEEE standard. However, systems that are designed to conform to the IEEE standard do not need to use this capability of the processor.

### 11.5 INSTRUCTION FORMAT

The instruction format for floating-point instructions is the same as for the other processor instructions. When programming in assembly language, an assembly language statement begins with an instruction mnemonic and is followed by from one to three operands. For example, the multiply-real instruction `mulr` might be used as follows:

```
mulr r8, r9, fp3
```
Here, real operands in local registers r8 and r9 are multiplied together and the result is stored in floating-point register fp3.

From the machine level point of view, all floating-point instructions use the REG format. Refer to Appendix B for details on the REG format instructions.

### 11.6 INSTRUCTION OPERANDS

Operands for floating-point instructions can be either floating-point literals or registers. The processor recognizes two encodings for floating-point literals: +0.0 and +1.0.

All of the registers in the processor’s execution environment (global registers g0 through g15, local registers r0 through r15, and floating-point registers fp0 through fp3) can be used as operands in floating-point instructions. (Of course, registers g15, r0, r1, and r2 would generally not be used for storing floating-point numbers, since they are reserved for stack management functions.)

When global or local registers are specified as operands, the instruction mnemonic (or opcode) determines how the values in these registers are interpreted. For example, there are two floating-point divide instructions: divide real (divr) and divide long real (divrl). When using the divr instruction, the processor assumes that global- or local-register operands contain real (32-bit) values. When using the divrl instruction, global- or local-register operands are assumed to contain long-real (64-bit) values.

With either instruction, floating-point registers (containing extended-real values) can also be used as operands.

Using floating-point registers as operands allows mixed format or mixed precision arithmetic to be performed with either real and extended-real values or long-real and extended-real values. Mixed-format operations with real and long-real values are not supported.

### 11.7 SUMMARY OF FLOATING-POINT INSTRUCTIONS

The processor’s floating-point instructions consist of all instructions for which at least one operand is a real data type.

These instructions can be divided into the following groups:

- Data Movement
- Data Type Conversion
- Basic Arithmetic
- Comparison and Classification
- Trigonometric
- Logarithmic and Exponential
The following sections give a brief overview of the instructions in each group. Detailed descriptions of the operations of these instructions are given in Section 10.

11.7.1 Data Movement

As has been described earlier in this section, the non-floating-point load and store instructions are used to move real values between registers and memory. Once in registers, the non-floating-point move instructions (mov, movl, and movt) are used to move real values between global and local registers without format conversion; whereas, the floating-point move instructions (movr, movrl, and movre) are used to move real values between global and local registers and floating-point registers.

The copy-sign-real extended (cpysre) and copy-reverse-sign real-extended (cpyrsre) instructions provide a means of copying the sign of one extended-real value to another, if one of the values is in a floating-point register. This operation is best performed on real and long-real values using the bit instructions chkbit and alterbit.

11.7.2 Data Type Conversion

Two types of data type conversions are provided: conversion from one floating-point format to another (e.g., real to extended real) and conversion between integer and real.

Conversion between floating-point formats is handled in either of two ways: explicitly by move instructions or implicitly by using the floating-point registers as operands in instructions.

As described earlier in this section, the movr instruction implicitly converts values from real to extended real, and vice versa, when moving values between global or local registers and floating-point registers. Likewise, the movrl instruction implicitly converts values from long real to extended real, and vice versa.

Conversion between real and long-real formats requires the use of both instructions. For example, the following two instructions convert a real value in global register g6 to a long-real value contained in g6 and g7, using a floating-point register for intermediate storage of the value:

```plaintext
movr g6, fpl
movrl fpl, g6
```

Implicit format conversion is also provided through the arithmetic, trigonometric, logarithmic, and exponential instructions. For example, the instruction

```plaintext
addr r4, r5, fp2
```

adds two real values together and produces an extended-real result.
The following six instructions allow conversion between integers and reals:

- cvtir: convert integer to real
- cvtilr: convert long integer to long real
- cvtri: convert real to integer
- cvtril: convert real to long integer
- cvtzri: convert truncated real to integer
- cvtzril: convert truncated real to long integer

Both the cvtir and cvtilr instructions can be used to convert an integer to an extended-real value by specifying that the result be placed in a floating-point register.

The convert real-to-integer instructions round off the real value to the nearest integer or long-integer value. For the cvtri and cvtril instructions, the rounding mode determines the direction the real number is rounded. For the convert truncated real-to-integer instructions (cvtzri and cvtzril), rounding is always toward zero. The latter two instructions are provided to allow efficient implementation of FORTRAN-like truncation semantics.

Extended-real values can be converted to integers by using a floating-point register as a source operand in either of the convert real-to-integer instructions.

Converting long-real values to integers requires two instructions, as in the following example:

```
movrl g6, fp3
cvtzri fp3, g6
```

The first instruction moves the long-real value to a floating-point register. The second instruction converts the extended-real value to an integer.

### 11.7.3 Basic Arithmetric

The following instructions perform the basic arithmetic operations specified in the IEEE standard:

- addr: add real
- addrl: add long real
- subr: subtract real
- subrl: subtract long real
- mulr: multiply real
- mulrl: multiply long real
- divr: divide real
- divrl: divide long real
- remr: remainder real
The round instructions round the floating-point operand to its nearest integral (i.e., integer) value, based on the current rounding mode. These instructions perform a function similar to the convert real-to-integer instructions except that the result is in floating-point format.

### 11.7.4 Comparison, Branching, and Classification

Comparison of floating-point values differs from comparison of integers or ordinals because with floating-point values there are four, rather than the usual three, mutually exclusive relationships: less than, equal to, greater than, and unordered.

The unordered relationship is true when at least one of the two values being compared is a NaN. This additional relationship is required because, by definition, NaNs are not numbers, so they cannot have greater than, equal, or less than relationships with other floating-point values.

The following instructions are provided for comparing floating-point values:

- `cmpr`: compare real
- `cmprl`: compare long real
- `cmppor`: compare ordered real
- `cmpporl`: compare ordered long real

All of these instructions set the condition code flags in the arithmetic controls to indicate the results of the comparison. With the compare instructions (`cmpr` and `cmprl`), the condition code flags are set to 0002 for the unordered condition. With the compare ordered instructions (`cmppor` and `cmpporl`), the condition code flags are set to 0002 and an invalid-operation exception is signaled for the unordered condition.

Two branch instructions (`bo` and `bno`) allow conditional branching to be performed on an ordered or unordered condition, respectively. With these instructions, the processor checks the condition code flags for unordered (0002) or ordered (1112) and branches accordingly.

The classify-real instructions (`classr` and `classrl`) provide a means of determining the class of a floating-point value (i.e., zero, denormalized finite, normalized finite, $\infty$, SNaN, or QNaN). The result of this operation is stored in the arithmetic status field of the arithmetic controls.
11.7.5 Trigonometric

The following instructions provide four common trigonometric functions:

\[
\begin{align*}
\text{sin} & \quad \text{sine real} \\
\text{sinrl} & \quad \text{sine long real} \\
\text{cosr} & \quad \text{cosine real} \\
\text{cosrl} & \quad \text{cosine long real} \\
\text{tanr} & \quad \text{tangent real} \\
\text{tanrl} & \quad \text{tangent long real} \\
\text{atanr} & \quad \text{arctangent real} \\
\text{atanrl} & \quad \text{arctangent long real}
\end{align*}
\]

The arctangent instructions facilitate conversion from rectangular to polar coordinates.

11.7.6 Pi

The processor uses the following value for \( \pi \) in its computations:

\[ \pi = 0.f \times 2^e \]

where

\[
\begin{align*}
f &= C90FDAAA2 \ 2168C234 \ C_{16} \\
e &= 2 \text{ if significand is } 0.
\end{align*}
\]

(The spaces in the fraction above indicate 32-bit boundaries.)

This value has a 66-bit mantissa, which is 2 bits more than is allowed in the significand of an extended-real value. (Since 66 bits is not an even number of hex digits, two additional zeros have been added to the value so that it can be represented in a hexadecimal format. The least-significant hex digit (C16) is thus 1100\(_2\), where the two least significant bits represent bits 67 and 68 of the mantissa.)

If the results of computations that explicitly use \( \pi \) are to be used in the sine, cosine, or tangent instructions, the full 66-bit fraction for \( \pi \) should be used. This insures that the results are consistent with the argument reduction algorithms that these instructions use. Using a rounded version of \( \pi \) can cause inaccuracies in result values, which if propagated through several calculations, might result in meaningless results.

A common method of representing the full 66-bit fraction of \( \pi \) is to separate the value into two numbers. For example, the following two long-real values added together give the value for \( \pi \) shown above with the full 66-bit fraction:
\[ \pi = \text{higher } \pi + \text{low } \pi \]

where

\[
\text{higher } \pi = 400921FB \ 54400000_{16} \\
\text{low } \pi = 3DD0B461 \ 1A600000_{16}
\]

Here \( \pi \) gives the most significant 33 bits of \( \pi \) and \( \text{low } \pi \) gives the least significant 33 bits. Similar versions of \( \pi \) can also be written in the extended-real format.

When using this two-part \( \pi \) value in an algorithm, parallel computations should be performed on each part, with the results kept separate. When all the computations are complete, the two results can be added together to form the final result.

### 11.7.7 Logarithmic, Exponential, and Scale

The following instructions provide three different logarithmic functions, an exponential function, and a scale function:

- \text{logbnr} \quad \text{log binary real}
- \text{logbnrl} \quad \text{log binary long real}
- \text{logr} \quad \text{log real}
- \text{logrl} \quad \text{log long real}
- \text{logepr} \quad \text{log epsilon real}
- \text{logeprl} \quad \text{log epsilon long real}
- \text{expr} \quad \text{exponent real}
- \text{exprl} \quad \text{exponent long real}
- \text{scaler} \quad \text{scale real}
- \text{scalerl} \quad \text{scale long real}

These instructions are described in detail in Section 10. The following is a brief description of their functions.

The log binary instructions compute the IEEE recommended function \( \logb (X) \). The result is an integral value that is the binary log of \( X \).

The log instructions compute the function \( Y * \log (X) \), where the log of \( X \) is the base-logarithm.

The log epsilon instructions compute the function \( Y * \log (X + 1) \), where the log of \( X + 1 \) is a base-2 logarithm.

The exponent instructions compute the value \( 2^x - 1 \).

The scale instructions perform a multiplication of a floating-point value by a power of 2.
11.7.8 Arithmetic Versus Nonarithmetic Instructions

The floating-point instructions can be divided into two groups: arithmetic and nonarithmetic. Arithmetic instructions are those that are sensitive to real values, meaning that they distinguish among NaN, ∞, normalized finite, denormalized finite, and zero values.

All but five of the floating-point instructions are arithmetic. The five nonarithmetic instructions are move-real extended (movre), copy-sign real extended (cpyrsre), copy-reversed-sign real extended (cprysre), and classify real (classr and classrl). These nonarithmetic instructions are insensitive to real values and cannot generate floating-point exceptions or faults.

This distinction between arithmetic and nonarithmetic instructions is important because floating-point exceptions and faults can be signaled only during the execution of arithmetic instructions.

11.8 OPERATIONS ON NANS

As was described earlier in this section, the processor supports two types of NaNs: QNaN and SNaN. An SNaN is any NaN value with its most-significant fraction bit set to 0 and at least one other fraction bit set to 1. (If all the fraction bits are set to 0, the value is an ∞.) A QNaN is any NaN value with the most-significant fraction bit set to 1. The sign bit of a NaN is not interpreted.

In general, when a QNaN is used in one or more arithmetic floating-point instructions, it is allowed to propagate through a computation. An SNaN on the other hand causes a floating invalid-operation exception to be signaled.

The floating invalid-operation exception has a flag and a mask bit associated with it in the arithmetic controls. The mask bit determines how the processor handles an SNaN value. If the floating invalid-operation mask bit is set, the SNaN is converted to a QNaN by setting the most significant fraction bit of the value to a 0. The result is then stored in the destination and the floating invalid-operation flag is set. If the invalid operation mask is clear, a floating invalid-operation fault is signaled and no result is stored in the destination.

When the result is a QNaN, the format of the result is as shown in Table 23, depending on the form of the source operands.

In some cases, a QNaN result is returned when none of the source operands are NaNs. Here, a standard QNaN is returned. The significand for the standard QNaN is as follows:

\[ 1.1000...00 \]

(For real and long-real destinations, the integer bit will be an implied 1.)

Other than the rules specified above, software is free to use the other bits of a NaN for any purpose.
Table 23. Format of QNaN Results

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>QNaN Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only one operand is NaN, destination is same width</td>
<td>QNaN version of NaN source</td>
</tr>
<tr>
<td>Only one operand is NaN, destination is longer</td>
<td>QNaN version of NaN source, with fraction extended with zeros</td>
</tr>
<tr>
<td>Only one operand is NaN, destination is shorter</td>
<td>QNaN version of NaN source, with fraction truncated</td>
</tr>
<tr>
<td>Both operands are NaNs</td>
<td>QNaN version of source whose fraction field has greatest magnitude, with fraction extended or truncated as described above</td>
</tr>
</tbody>
</table>

11.9 EXCEPTIONS AND FAULT HANDLING

Occasionally, a floating-point instruction can result in an exception being signaled. The processor recognizes six floating-point exceptions:

- Floating Reserved Encoding
- Floating Invalid Operation
- Floating Zero Divide
- Floating Overflow
- Floating Underflow
- Floating Inexact

These exceptions can be divided into two categories:

1. Situations in which one or more source operands are inappropriate for an operation and would cause an exception to be signaled.
2. Situations in which the result of an operation is exceptional.

The reserved encoding, invalid operation, and division-by-zero exceptions fall in the first category; the overflow, underflow, and inexact exceptions fall in the second category.

Except for the floating reserved-encoding exception, each of these exceptions has a flag and a mask bit associated with it in the arithmetic controls. When an exception condition occurs, the processor performs one of the following operations:

- If the mask bit for the exception is set, the flag for the exception is set and instruction execution continues, substituting a default value in place of the result.
If the mask bit for the exception is clear, the flag for the exception is not set and a floating-point arithmetic fault is raised. The processor then stores diagnostic information in the fault information area and diverts instruction execution to a fault handler.

Since the floating reserved-encoding exception does not have a flag or mask bit, it always results in a fault.

Note
The floating-point exception flags are "sticky," which means that the processor does not implicitly clear them while carrying out floating-point operations. They may be cleared by software.

11.9.1 Fault Handler

As is described in Section 9, when a floating-point fault is signaled, the processor calls a single fault handler. This fault handler determines how to handle the specific fault subtype by interpreting the floating-point exception flags and the information in the fault record.

11.9.2 Floating Reserved-Encoding Exception

A reserved encoding exception occurs as a result of either of the following two conditions:

- When a reserved encoding is used as an operand in a floating-point instruction, or
- When a denormalized value is used as an operand in a floating-point instruction and the normalizing-mode bit in the arithmetic controls is clear.

The first condition is rare. It can only occur if a program presents an extended-real value to the processor that has a zero j-bit (integer part) and a non-zero biased exponent.

The second condition was discussed earlier in the section titled "Normalizing Mode." This condition is also rare, since the vast majority of programs run with the normalizing mode enabled.

There is neither a flag nor a mask bit for this exception. When a reserved-encoding exception occurs, the processor raises a floating reserved-encoding fault and does not store a result.

11.9.3 Floating Invalid-Operation Exception

The invalid-operation exception indicates that one of the source operands is inappropriate for the type of operation being performed. The following conditions cause this exception to be signaled:

- Any arithmetic operation on an SNaN
- Addition of infinities of unlike sign
- Subtraction of infinities of like sign
- Multiplication of zero by $\infty$
• Division of zero by zero or \( \infty \) by \( \infty \)
• Remainder of \( x \) by \( y \), if \( y \) is zero or \( x \) is \( \infty \)
• Square root of a negative, nonzero value
• Conversion of a NaN from floating-point format to integer format
• Sine, cosine, or tangent of \( \infty \)
• \( y \times \log(x) \), if:
  - \( x \) is negative and nonzero,
  - \( y \) is zero and \( x \) is \( \infty \)
  - \( y \) and \( x \) are zero, or
  - \( y \) is \( \infty \) and \( x \) is 1
• Log epsilon of \( (y, x) \), if \( y \) is \( \infty \) and \( x \) is 0
• Compare ordered, if a source operand is a NaN

When a floating invalid-operation exception occurs and its mask is set, the following occurs:

• When the result is a floating-point value, the standard QNaN value is stored in the destination and the floating invalid-operation flag is set. (A discussion of how the processor handles NaNs was provided earlier in the section titled “Operations on NaNs.”)

• When the result is an integer, the maximum negative integer is stored in the destination and the floating invalid-operation flag is set.

When the mask is clear, no result is stored; the floating invalid-operation flag is not set; and the floating invalid-operation fault is signaled.

11.9.4 Floating Zero-Divide Exception

The floating zero-divide exception is signaled when an exact non-finite result would be produced from finite operands. (Note that a different exception, overflow, is signaled when an infinite result is produced inexactly from finite operands.) The most common example of this exception is a division operation, where the divisor is zero and the dividend is a nonzero, finite value.

When the floating zero-divide mask is set: a correctly signed \( \infty \) is stored in the destination and the floating zero-divide flag is set. When the mask is clear, no result is stored; the floating zero-divide flag is not set; and a floating zero-divide fault is signaled.

11.9.5 Floating Overflow Exception

The overflow exception occurs when the infinitely precise result of a floating-point instruction exceeds the largest allowable finite value for the specified destination format. For example, if the destination format is real (32 bits), overflow occurs when the infinitely precise result falls outside the range \(-1.0 \times 2^{126}\) to \(1.0 \times 2^{126}\) (exclusive), where 126 is the unbiased exponent of the result.
When the floating overflow mask is set, a rounded result is stored in the destination and the floating overflow flag is set. The current rounding mode determines the method used to round the result.

When the mask is clear: no result is stored in the destination and the floating overflow flag is not set. Instead, the processor stores the result in extended-real format in the fault information area. The fraction of the extended-real value is rounded to the instruction’s destination precision. For example, if the destination operand’s format is real (32 bits), the extended-real fraction is rounded to 23 bits, with the 40 least-significant bits filled with zeros.

If the exponent exceeds the range of the extended-real format (16383 unbiased), then the exponent is divided by $2^{24576}$ and a flag (bit 1 of the fault flags byte or override flags byte) is set in the fault information area to indicate that the exponent has been bias adjusted. After this fault information is stored, a floating overflow fault is signaled.

When using the scale instructions (scaler or scalerl), massive overflow can occur, where the infinitely precise result is too large to be represented, even with a bias adjusted exponent. Here, a properly signed $\infty$ is stored in the fault record.

The floating overflow exception cannot occur on a conversion from floating-point format to integer format (although an integer overflow exception can occur).

11.9.6 Floating Underflow Exception

An underflow condition occurs when the infinitely precise result of a floating-point instruction is less than the smallest possible normalized, finite value for the specified destination format. For example, for the real (32-bit) format, underflow occurs when an infinitely precise result falls in the range $-1.0 \times 2^{126}$ to $1.0 \times 2^{126}$ (exclusive), where -126 is the unbiased exponent.

When a floating underflow condition occurs, the setting of the floating underflow mask determines how the processor handles the condition.

If the mask is set when an underflow condition occurs, the processor goes ahead and denormalizes the result. Then if the result is exact, it is stored in the destination and the floating underflow exception is not signaled, nor is the floating underflow flag set. If, on the other hand, the denormalized result is inexact, the floating underflow flag is set and the processor goes on to handle the inexact condition as described in the next section.

If the floating underflow mask is clear when an underflow-condition occurs, no result is stored in the destination and the floating underflow flag is not set. Instead, the processor stores the result in extended-real format in the fault information area, with the fraction of the extended-real value rounded to the instruction’s destination precision. For example, if the destination precision is real (23-bit fraction) the 40 least-significant bits of the fraction are set to 0.

If the exponent of the value stored is less than the minimum allowable value in the extended-real format (-16,382 unbiased), then the exponent is multiplied by $2^{24576}$ and a flag (bit 1 of the fault or
override flags byte) is set in the fault information area to indicate that the exponent has been bias adjusted. After this information is stored, a floating underflow fault is signaled.

The scale instructions can cause massive underflow to occur, where the infinitely precise result is too small to be represented, even with a bias adjusted exponent. Here, a properly signed zero is stored in the fault record.

Refer to the section later titled “Floating-Point Underflow Condition” for more information on the interaction of the floating underflow and inexact exceptions.

11.9.7 Floating Inexact Exception

The floating inexact exception occurs when an infinitely precise result cannot be encoded in the format specified for the destination operand. Either of the following two conditions can cause an inexact exception to be signaled:

- When a result is rounded and the result is not exact
- When overflow occurs and the floating overflow mask is set

If the floating inexact mask is set when an inexact condition occurs and an unmasked overflow or underflow condition does not occur, the rounded result is stored in the destination and the floating-point inexact flag is set. The current rounding mode determines the method used to round the result.

If the floating inexact mask is clear when an inexact condition occurs, the floating inexact flag is not set and one of the following operations is carried out:

- If only the inexact condition has occurred, the processor stores the rounded result in the specified destination, then raises a floating-inexact fault.
- If the inexact condition occurs along with overflow or underflow, no result is stored in the destination. Instead, the processor stores the result in extended-real format in the fault information area, as described for the floating overflow and underflow exceptions, then raises a floating inexact fault.

Refer to the following section for more information on the interaction of the floating underflow and inexact exceptions.

11.9.8 Floating-Point Underflow Condition

Two aspects of underflow are important in numeric processings: the “tininess” of a number and “loss of accuracy.” A result is tiny when it is nonzero and its exponent is between $+2^{E_{\text{min}}}$, where $E_{\text{min}}$ is the smallest unbiased exponent allowed in the destination format. For example, if the destination format is long-real (64-bit format), a result is tiny if it is nonzero and in the range of $+1 \times 2^{1022}$ to $-1 \times 2^{21022}$. The ability to detect a tiny result is important because such a result may cause an exception to be signaled in a later operation (e.g., overflow on a division).
Loss of accuracy occurs when a tiny result is approximated as part of the denormalization process so that it will fit into the destination format.

In the 80960KB processor, tininess is detected after rounding as an underflow condition. Loss of accuracy is detected as an inexact condition.

The algorithm in Figure 27 shows how the processor responds to these two conditions, when a floating-point operation produces a tiny result.

An important point to note in this algorithm is that if the underflow mask is set, an underflow exception is signaled only if the denormalized result is inexact. If the denormalized number is exact, no flags are set and no faults are signaled.
generate infinitely precise result  # exponent and significand;
if exponent < underflow threshold
  then
    if underflow fault mask clear
      then
        goto underflow fault handler;
        exit algorithm;
      else generate denormalized number
        if denormalized significand equals infinitely precise significand
          then
            store denormalized result in destination;
            # no underflow is signaled;
          else
            set underflow flag in AC;
            if inexact fault mask is clear
              then
                goto inexact fault handler;
                exit algorithm;
              else
                set inexact flag in AC;
                store denormalized result in destination;
          end if;
        end if;
      end if;
  end if;
else
  if infinitely precise result is inexact
    then
      if inexact fault mask is clear
        then
          goto inexact fault handler;
          exit algorithm;
        else
          set inexact flag in AC;
          store normalized result in destination;
        end if;
      else
        store normalized result in destination;
      end if;
  end if;
exit algorithm

Figure 27. Interaction of Floating Underflow and Inexact Exceptions
12.0 INTERAGENT COMMUNICATION

This section describes the interagent communication (IAC) mechanism of the 80960KB processor. Included is a description of the IAC message structure, the IAC message sending and receiving mechanism, and reference information on the available IAC messages.

Note
The 80960KB processor's interagent communication mechanism is an extension to the architecture and may not be supported in other processors based on this architecture.

12.1 INTRODUCTION TO IAC MESSAGES

The IAC facilities provide a mechanism for agents connected to the processor's bus to communicate with the processor by means of messages. The agents that use these facilities may be other 80960KB processors, I/O processors, or special purpose hardware. Programs running on the 80960KB processor can also use this message-passing mechanism to send messages internally to the processor.

The primary function of these facilities is to provide an alternative to the interrupt mechanism for external hardware to communicate with the processor. Also, certain processor functions like reinitialization, purging the instruction cache, and setting breakpoint registers can only be carried out with this mechanism.

IAC messages (referred to here as IACs) are four words in length and are exchanged by means of message buffers that are mapped to memory. All the usable IACs are predefined. The processor handles an IAC in much the same way as it handles an instruction.

The processor provides two mechanisms for exchanging IACs: external and internal. The external IAC mechanism is used to pass IACs between two agents on the processor's bus. A processor uses the internal IAC mechanism to pass an IAC to itself.

12.2 IAC MESSAGE FORMAT

Figure 28 shows the format for an IAC message. Each message consists of a message-type field and up to five parameter fields.

![Figure 28. IAC Message Format](image-url)
The message type is an 8-bit binary code. Each IAC has a unique message type.

The parameters can be 8, 16, or 32-bits in length, depending on the specified field. Many of the IACs do not require parameters. When a message type does require one or more parameters, the processor only looks at the required parameter fields. Those fields not used are ignored.

### 12.3 SOFTWARE REQUIREMENTS FOR HANDLING IACS

No special software, such as dedicated data structures or stacks, are required to handle IACs. An IAC is sent with a quad synchronous move instruction (synmovq). When the processor receives an IAC, it handles it independently from the program execution environment. It does not use the instruction execution unit, the registers (global or local), the stack, or memory. Thus, the state of the processor when the IAC is received does not need to be saved.

Some IACs, such as the purge instruction cache IAC, do not affect the processor’s state. The processor treats these IACs as if they were an instruction inserted in the control flow of the process. When the IAC action is complete, the processor resumes work on the program it is currently running.

Other IACs, such as the reinitialize processor IAC, cause the state of the processor or the control of the currently running program to be permanently changed. In these instances, the processor resumes activity in its new processor state, following the execution of the IAC.

All IACs are assumed to have a priority of 31, so the processor executes the action requested in the IAC message immediately, even if the processor's current priority is 31. While the processor is handling an IAC, it will not respond to interrupts signaled on the interrupt pins.

### 12.4 INTERNAL IACS

Internal IACs are used for functions such as setting breakpoint registers, purging the instruction cache, or sending software initiated interrupts.

To send an internal IAC, software must perform the following steps:

1. Load the message into four consecutive words in memory, with the first word aligned on a word boundary.
2. Execute a synmovq instruction to move the message from its source address to destination address FF00001016.

When the destination operand of a1 synmovq instruction is FF00001016, the processor interprets the instruction as a send internal-IAC instruction. The processor then receives the IAC by moving the message from memory into an internal message buffer.

The action of the synmovq move instruction insures that the loading of the message into the processor is completed before the processor is allowed to perform any other chores.
Note
The address range of FF000000₁₆ through FFFFFFFF₁₆ is reserved for interrupt handling and IAC message passing.

12.5 EXTERNAL IACS

External IACs are used by agents external to the processor to initiate processor actions such as testing for pending interrupts or freezing the processor. External IACs can be sent between two 80960KB processors that are connected to the same bus or by external logic that duplicates the external IAC sending mechanism. The following sections describe how one processor sends an IAC to another processor. The 80960KB Hardware Designer’s Reference Manual describes the requirements that external logic must meet to perform these same functions.

12.5.1 Sending External IACs

Sending an external IAC message is similar to sending an internal IAC message, except that the address of the receiving agent is specified in a slightly different way. Figure 29 shows the required encoding of the address for the receiving agent.

![Figure 29. Encoding of Address for Processor Receiving an IAC](image)

At initialization each agent on the bus is assigned a unique address in the range of FF000C00₁₆ to FFFFCC00₁₆. To send an IAC to an agent, the sending agent sends the message to the address assigned to the receiving agent. As shown in Figure 29, only bits 14 through 23 of this address are interpreted to determine the address of the receiving agent. Bits 4 through 8 of this address are used to encode the priority of the message.

For example, to send a priority 25₁₀ IAC to the agent at address 000000001₂, the message address would be FF004D90₁₆.

To send an external IAC from one 80960KB processor to another, software must perform the following steps:

1. Load the message into four consecutive words in memory, with the first word aligned on a word boundary.
2. Execute a **synmovq** instruction to move the message from its source address to the address of the receiving agent (encoded in the form shown in Figure 29).
3. Check the condition code in the arithmetic controls to determine if the message was received (0102) or rejected (0002).

The action of the synmovq move instruction insures that the sending processor does not execute any other instructions until the synmovq instruction is complete. It also sets the condition code bits to indicate whether or not the move was successful. A successful move is interpreted as the IAC being received by the processor.

### 12.5.2 Receiving and Handling an External IACs

A processor receives and handles an external IAC in somewhat the same manner as it receives and handles an interrupt. To configure a processor to receive external IACs, vector INTO of the interrupt-control register (shown in Figure 19) is set to 0. The INTO pin on the processor chip then becomes the IAC pin. (Refer to Section 7, “Interrupts From Interrupt Pins” for further discussion of the interrupt pins and interrupt-control register.)

When the processor receives a signal on the IAC pin, it handles it initially as if it were receiving an interrupt. It reads the vector number associated with this pin (bits 0 through 7 of the interrupt-control register). If it is zero, the processor recognizes that it is receiving an external IAC. It then reads the four-word IAC message from the bus and performs the requested IAC.

The processor acts immediately on any IAC that it receives. For efficient system operation, external logic must thus be provided to insure that low priority IAC messages do not interrupt the processor while it is handling a higher priority task. The handshaking for this operation is provided by the write-external-priority mechanism described in Section 6.

Using the write-external-priority mechanism, the processor keeps the external logic updated regarding the processor's current priority. When an IAC is sent to the processor, the external logic intercepts it and reads the priority. The external logic then determines whether the IAC priority is above that of the processor or not. If the IAC has a higher priority, the external logic sends an acknowledge signal to the sending processor, then signals the receiving processor by asserting the IAC pin. If the IAC has an equal or lower priority, the external logic sends a non-acknowledge signal to the sending processor.

The sending processor uses the acknowledge or non-acknowledge signals to set the condition codes to complete the synmovq instruction.

While the processor is servicing an IAC, it performs some handshaking with the external logic so that the logic knows when the processor has finished work on an IAC. The external logic is then able to reject any IAC that it receives while the processor is servicing another IAC.

### 12.6 SUMMARY OF IAC MESSAGES

Table 24 gives a list of the IAC messages that the processor can send either internally or externally. The following section provides detailed reference information on these messages.
Table 24. IAC Messages

<table>
<thead>
<tr>
<th>Interrupt Handling</th>
<th>Processor Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>Purge Instruction Cache</td>
</tr>
<tr>
<td>Test Pending</td>
<td>Set Breakpoint Register</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Store System Base</td>
</tr>
<tr>
<td>Test Pending</td>
<td>Freeze</td>
</tr>
<tr>
<td>Freeze</td>
<td>Continue Initialization</td>
</tr>
<tr>
<td>Store System Base</td>
<td>Reinitialize Processor</td>
</tr>
<tr>
<td>Continue Initialization</td>
<td></td>
</tr>
</tbody>
</table>

12.7 IAC MESSAGE REFERENCE

The following section provides detailed descriptions of the operations carried out for each of the IACs. This section is organized alphabetically by IAC title for easy reference.
Continue Initialization

Message Type: 92₁₆
Function: Carries out the initialization procedure that follows the processor self test. The processor executes the initialization procedure beginning with reading the initial memory image from ROM. The self test is not performed.

Refer to the section in Chapter 7 titled "Processor Initialization" for further details on the initialization process.
Freeze

Message Type: $91_{16}$

Function: Stops the processor. The processor puts itself in the stopped state.
Interrupt

Message Type: \(40_{16}\)

Parameters:
- Field 1: Interrupt vector
- Fields 2 - 5: Not Used

Function:
Generates an interrupt request. The interrupt vector is given in field 1 of the IAC message. The processor handles the interrupt request just as it does interrupts received from other sources. If the interrupt priority is higher than the processor's current priority, the processor services the interrupt request immediately. Otherwise, it posts the interrupt in the pending interrupts section of the interrupt table.

Refer to Chapter 8 for further information on the servicing of interrupt IACs.
Purge Instruction Cache

Message Type: \[89_{16}\]

Function: Invalidates all entries in the processor's internal instruction cache.
Reinitialize Processor

Message Type: \(93_{16}\)

Parameters:
- Fields 1 - 2: Not Used
- Field-3: Address of System Address Table
- Field-4: Address of Processor Control Block
- Field 5: Start Instruction IP

Function:
Reestablishes the processor state. In reinitializing itself, the processor first locates the system address table and the processor control block in the IMI from the addresses given in fields 3 and 4. The processor then begins executing the instruction list beginning with the IP given in field 5.
Set Breakpoint Register

Message Type: $8F_{16}$

Parameters:
- Fields 1 - 2: Not Used
- Field 3: Breakpoint IP
- Field 4: Breakpoint IP
- Field 5: Not Used

Function:
Enables or disables two breakpoints. When the processor receives this IAC, it conditionally loads the parameters from fields 3 and 4 into breakpoint registers 0 and 1, respectively. Field 3 provides a breakpoint IP for breakpoint register 0, and field 4 provides a breakpoint IP for breakpoint register 1. Bit 1 in each of these fields is a breakpoint disable flag.

If the disable flag in one of these fields is set, the breakpoint for the corresponding breakpoint register is disabled. Otherwise, the IP value in the field is loaded into the corresponding breakpoint register and the breakpoint is enabled.

Breakpoints are described in the section in Chapter 10 titled "Breakpoint-Trace Mode."
Store System Base

Message Type: 80_{16}

Parameters:
- Fields 1 - 2: Not Used
- Field 3: Destination Address
- Fields 4 - 5: Not Used

Function:
Stores the current locations of the system address table and the PRCB in a specified location in memory. The address of the system address table is stored in the word starting at the byte specified in field 3, and the address of the PRCB is stored in the next word in memory (field 3 address plus 4).
Test Pending Interrupts

Message Type: \( 41_{16} \)

Function: Tests for pending interrupts. The processor checks the pending interrupt section of the interrupt table for a pending interrupt with a priority higher than the processor's current priority. If a higher priority interrupt is found, it is serviced immediately. Otherwise, no action is taken.
APPENDIX A
INSTRUCTION AND DATA STRUCTURE QUICK REFERENCE

This appendix provides quick reference for the 80960KB instructions and data structures.

INSTRUCTION QUICK REFERENCE

This section provides two lists of 80960KB instructions: one sorted by assembly-language mnemonic and another sorted by machine-level opcode. In these lists, each entry includes the assembly-language mnemonic for an instruction; the operands (given in the required order); the machine-level opcode and instruction type (i.e., REG, MEM, COBR, CTRL); and the page number in Chapter 11 where the detailed description of the instruction is given.
### Instruction List by Assembler Mnemonic

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Opcode</th>
<th>Inst. Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>addc</td>
<td>src1, src2, dst</td>
<td>5B0</td>
<td>REG</td>
<td>11-6</td>
</tr>
<tr>
<td>addi</td>
<td>src1, src2, dst</td>
<td>591</td>
<td>REG</td>
<td>11-7</td>
</tr>
<tr>
<td>addo</td>
<td>src1, src2, dst</td>
<td>590</td>
<td>REG</td>
<td>11-7</td>
</tr>
<tr>
<td>addr</td>
<td>src1, src2, dst</td>
<td>78F</td>
<td>REG</td>
<td>11-8</td>
</tr>
<tr>
<td>addr1</td>
<td>src1, src2, dst</td>
<td>79F</td>
<td>REG</td>
<td>11-8</td>
</tr>
<tr>
<td>alterbit</td>
<td>bitpos, src, dst</td>
<td>58F</td>
<td>REG</td>
<td>11-10</td>
</tr>
<tr>
<td>and</td>
<td>src1, src2, dst</td>
<td>581</td>
<td>REG</td>
<td>11-11</td>
</tr>
<tr>
<td>andnot</td>
<td>src1, src2, dst</td>
<td>582</td>
<td>REG</td>
<td>11-11</td>
</tr>
<tr>
<td>atadd</td>
<td>src/dst, src, dst</td>
<td>612</td>
<td>REG</td>
<td>11-12</td>
</tr>
<tr>
<td>atanr</td>
<td>src1, src2, dst</td>
<td>680</td>
<td>REG</td>
<td>11-13</td>
</tr>
<tr>
<td>atanrl</td>
<td>src1, src2, dst</td>
<td>690</td>
<td>REG</td>
<td>11-13</td>
</tr>
<tr>
<td>atmod</td>
<td>src, mask, src/dst</td>
<td>610</td>
<td>REG</td>
<td>11-15</td>
</tr>
<tr>
<td>b</td>
<td>targ</td>
<td>08</td>
<td>CTRL</td>
<td>11-18</td>
</tr>
<tr>
<td>bal</td>
<td>targ</td>
<td>0B</td>
<td>CTRL</td>
<td>11-16</td>
</tr>
<tr>
<td>balx</td>
<td>targ, dst</td>
<td>85</td>
<td>MEM</td>
<td>11-16</td>
</tr>
<tr>
<td>bbc</td>
<td>bitpos, src, targ</td>
<td>30</td>
<td>COBR</td>
<td>11-20</td>
</tr>
<tr>
<td>bbs</td>
<td>bitpos, src, targ</td>
<td>37</td>
<td>COBR</td>
<td>11-20</td>
</tr>
<tr>
<td>be</td>
<td>targ</td>
<td>12</td>
<td>CTRL</td>
<td>11-22</td>
</tr>
<tr>
<td>bg</td>
<td>targ</td>
<td>11</td>
<td>CTRL</td>
<td>11-22</td>
</tr>
<tr>
<td>bge</td>
<td>targ</td>
<td>13</td>
<td>CTRL</td>
<td>11-22</td>
</tr>
<tr>
<td>bl</td>
<td>targ</td>
<td>14</td>
<td>CTRL</td>
<td>11-22</td>
</tr>
<tr>
<td>ble</td>
<td>targ</td>
<td>16</td>
<td>CTRL</td>
<td>11-22</td>
</tr>
<tr>
<td>bne</td>
<td>targ</td>
<td>15</td>
<td>CTRL</td>
<td>11-22</td>
</tr>
<tr>
<td>bno</td>
<td>targ</td>
<td>10</td>
<td>CTRL</td>
<td>11-22</td>
</tr>
<tr>
<td>bo</td>
<td>targ</td>
<td>11</td>
<td>CTRL</td>
<td>11-22</td>
</tr>
<tr>
<td>bx</td>
<td>targ</td>
<td>84</td>
<td>MEM</td>
<td>11-18</td>
</tr>
<tr>
<td>call</td>
<td>targ</td>
<td>09</td>
<td>CTRL</td>
<td>11-25</td>
</tr>
<tr>
<td>calls</td>
<td>targ</td>
<td>660</td>
<td>REG</td>
<td>11-27</td>
</tr>
<tr>
<td>callx</td>
<td>targ</td>
<td>86</td>
<td>MEM</td>
<td>11-29</td>
</tr>
<tr>
<td>chkbit</td>
<td>bitpos, src</td>
<td>5AE</td>
<td>REG</td>
<td>11-31</td>
</tr>
<tr>
<td>classr</td>
<td>src</td>
<td>68F</td>
<td>REG</td>
<td>11-32</td>
</tr>
<tr>
<td>classrl</td>
<td>src</td>
<td>69F</td>
<td>REG</td>
<td>11-32</td>
</tr>
<tr>
<td>clrbit</td>
<td>bitpos, src, dst</td>
<td>58C</td>
<td>REG</td>
<td>11-34</td>
</tr>
<tr>
<td>cmpdeci</td>
<td>src1, src2, dst</td>
<td>5A7</td>
<td>REG</td>
<td>11-36</td>
</tr>
<tr>
<td>cmpdeco</td>
<td>src1, src2, dst</td>
<td>5A6</td>
<td>REG</td>
<td>11-36</td>
</tr>
<tr>
<td>cmpi</td>
<td>src1, src2</td>
<td>5A1</td>
<td>REG</td>
<td>11-35</td>
</tr>
<tr>
<td>cmpibe</td>
<td>src1, src2, targ</td>
<td>3A</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpibg</td>
<td>src1, src2, targ</td>
<td>39</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpibge</td>
<td>src1, src2, targ</td>
<td>3B</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpibl</td>
<td>src1, src2, targ</td>
<td>3C</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpible</td>
<td>src1, src2, targ</td>
<td>3E</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpibne</td>
<td>src1, src2, targ</td>
<td>3D</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpibno</td>
<td>src1, src2, targ</td>
<td>38</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpibo</td>
<td>src1, src2, targ</td>
<td>3F</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operands</td>
<td>Opcode</td>
<td>Inst. Type</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------</td>
<td>--------</td>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>cmpinci</td>
<td>src1, src2, dst</td>
<td>5A5</td>
<td>REG</td>
<td>11-37</td>
</tr>
<tr>
<td>cmpinco</td>
<td>src1, src2, dst</td>
<td>5A4</td>
<td>REG</td>
<td>11-37</td>
</tr>
<tr>
<td>cmpo</td>
<td>src1, src2</td>
<td>5A0</td>
<td>REG</td>
<td>11-35</td>
</tr>
<tr>
<td>cmpobe</td>
<td>src1, src2, targ</td>
<td>32</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpobg</td>
<td>src1, src2, targ</td>
<td>31</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpobje</td>
<td>src1, src2, targ</td>
<td>33</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpobbl</td>
<td>src1, src2, targ</td>
<td>34</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpoble</td>
<td>src1, src2, targ</td>
<td>36</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpobne</td>
<td>src1, src2, targ</td>
<td>35</td>
<td>COBR</td>
<td>11-42</td>
</tr>
<tr>
<td>cmpor</td>
<td>src1, src2</td>
<td>684</td>
<td>REG</td>
<td>11-38</td>
</tr>
<tr>
<td>cmporl</td>
<td>src1, src2</td>
<td>694</td>
<td>REG</td>
<td>11-38</td>
</tr>
<tr>
<td>cmpor</td>
<td>src1, src2</td>
<td>685</td>
<td>REG</td>
<td>11-40</td>
</tr>
<tr>
<td>cmporl</td>
<td>src1, src2</td>
<td>695</td>
<td>REG</td>
<td>11-40</td>
</tr>
<tr>
<td>concmpi</td>
<td>src1, src2</td>
<td>5A3</td>
<td>REG</td>
<td>11-45</td>
</tr>
<tr>
<td>concmpo</td>
<td>src1, src2</td>
<td>5A2</td>
<td>REG</td>
<td>11-45</td>
</tr>
<tr>
<td>cosr</td>
<td>src, dst</td>
<td>68D</td>
<td>REG</td>
<td>11-46</td>
</tr>
<tr>
<td>cosrl</td>
<td>src, dst</td>
<td>69D</td>
<td>REG</td>
<td>11-46</td>
</tr>
<tr>
<td>cpyprsre</td>
<td>src1, src2, dst</td>
<td>6E3</td>
<td>REG</td>
<td>11-48</td>
</tr>
<tr>
<td>cpysre</td>
<td>src1, src2, dst</td>
<td>6E2</td>
<td>REG</td>
<td>11-48</td>
</tr>
<tr>
<td>cvtllr</td>
<td>src, dst</td>
<td>675</td>
<td>REG</td>
<td>11-49</td>
</tr>
<tr>
<td>cvtllr</td>
<td>src, dst</td>
<td>674</td>
<td>REG</td>
<td>11-49</td>
</tr>
<tr>
<td>cvtril</td>
<td>src, dst</td>
<td>6C0</td>
<td>REG</td>
<td>11-50</td>
</tr>
<tr>
<td>cvttril</td>
<td>src, dst</td>
<td>6C1</td>
<td>REG</td>
<td>11-50</td>
</tr>
<tr>
<td>cvtzrli</td>
<td>src, dst</td>
<td>6C2</td>
<td>REG</td>
<td>11-50</td>
</tr>
<tr>
<td>cvtzrli</td>
<td>src, dst</td>
<td>6C3</td>
<td>REG</td>
<td>11-50</td>
</tr>
<tr>
<td>daddc</td>
<td>src1, src2, dst</td>
<td>642</td>
<td>REG</td>
<td>11-52</td>
</tr>
<tr>
<td>divi</td>
<td>src1, src2, dst</td>
<td>74B</td>
<td>REG</td>
<td>11-53</td>
</tr>
<tr>
<td>divo</td>
<td>src1, src2, dst</td>
<td>70B</td>
<td>REG</td>
<td>11-53</td>
</tr>
<tr>
<td>divr</td>
<td>src1, src2, dst</td>
<td>78B</td>
<td>REG</td>
<td>11-54</td>
</tr>
<tr>
<td>divrl</td>
<td>src1, src2, dst</td>
<td>79B</td>
<td>REG</td>
<td>11-54</td>
</tr>
<tr>
<td>dmovt</td>
<td>src, dst</td>
<td>644</td>
<td>REG</td>
<td>11-56</td>
</tr>
<tr>
<td>dsubc</td>
<td>src1, src2, dst</td>
<td>643</td>
<td>REG</td>
<td>11-57</td>
</tr>
<tr>
<td>ediv</td>
<td>src1, src2, dst</td>
<td>671</td>
<td>REG</td>
<td>11-58</td>
</tr>
<tr>
<td>emul</td>
<td>src1, src2, dst</td>
<td>670</td>
<td>REG</td>
<td>11-59</td>
</tr>
<tr>
<td>expr</td>
<td>src, dst</td>
<td>689</td>
<td>REG</td>
<td>11-60</td>
</tr>
<tr>
<td>exprl</td>
<td>src, dst</td>
<td>699</td>
<td>REG</td>
<td>11-60</td>
</tr>
<tr>
<td>extract</td>
<td>bitpos, len, src/dst</td>
<td>651</td>
<td>REG</td>
<td>11-62</td>
</tr>
<tr>
<td>faulite</td>
<td>1A</td>
<td>CTRL</td>
<td>11-63</td>
<td></td>
</tr>
<tr>
<td>faultg</td>
<td>19</td>
<td>CTRL</td>
<td>11-63</td>
<td></td>
</tr>
<tr>
<td>faultge</td>
<td>1B</td>
<td>CTRL</td>
<td>11-63</td>
<td></td>
</tr>
<tr>
<td>faultl</td>
<td>1C</td>
<td>CTRL</td>
<td>11-63</td>
<td></td>
</tr>
<tr>
<td>faultle</td>
<td>1E</td>
<td>CTRL</td>
<td>11-63</td>
<td></td>
</tr>
<tr>
<td>faultne</td>
<td>1D</td>
<td>CTRL</td>
<td>11-63</td>
<td></td>
</tr>
<tr>
<td>faultno</td>
<td>18</td>
<td>CTRL</td>
<td>11-63</td>
<td></td>
</tr>
<tr>
<td>fausto</td>
<td>1F</td>
<td>CTRL</td>
<td>11-63</td>
<td></td>
</tr>
<tr>
<td>flushreg</td>
<td>66D</td>
<td>REG</td>
<td>11-65</td>
<td></td>
</tr>
<tr>
<td>fmark</td>
<td>66C</td>
<td>REG</td>
<td>11-66</td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operands</td>
<td>Opcode</td>
<td>Inst. Type</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>--------</td>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>ld</td>
<td>src, dst</td>
<td>90</td>
<td>MEM</td>
<td>11-67</td>
</tr>
<tr>
<td>lda</td>
<td>src, dst</td>
<td>8C</td>
<td>MEM</td>
<td>11-69</td>
</tr>
<tr>
<td>ldib</td>
<td>src, dst</td>
<td>C0</td>
<td>MEM</td>
<td>11-67</td>
</tr>
<tr>
<td>ldis</td>
<td>src, dst</td>
<td>C8</td>
<td>MEM</td>
<td>11-67</td>
</tr>
<tr>
<td>ldl</td>
<td>src, dst</td>
<td>98</td>
<td>MEM</td>
<td>11-67</td>
</tr>
<tr>
<td>ldob</td>
<td>src, dst</td>
<td>80</td>
<td>MEM</td>
<td>11-67</td>
</tr>
<tr>
<td>ldos</td>
<td>src, dst</td>
<td>88</td>
<td>MEM</td>
<td>11-67</td>
</tr>
<tr>
<td>ldq</td>
<td>src, dst</td>
<td>B0</td>
<td>MEM</td>
<td>11-67</td>
</tr>
<tr>
<td>ldt</td>
<td>src, dst</td>
<td>A0</td>
<td>MEM</td>
<td>11-67</td>
</tr>
<tr>
<td>logbnr</td>
<td>src, dst</td>
<td>68A</td>
<td>REG</td>
<td>11-70</td>
</tr>
<tr>
<td>logbnrl</td>
<td>src, dst</td>
<td>69A</td>
<td>REG</td>
<td>11-70</td>
</tr>
<tr>
<td>logepr</td>
<td>src1, src2, dst</td>
<td>681</td>
<td>REG</td>
<td>11-72</td>
</tr>
<tr>
<td>logeprl</td>
<td>src1, src2, dst</td>
<td>691</td>
<td>REG</td>
<td>11-72</td>
</tr>
<tr>
<td>logr</td>
<td>src1, src2, dst</td>
<td>682</td>
<td>REG</td>
<td>11-75</td>
</tr>
<tr>
<td>logrl</td>
<td>src1, src2, dst</td>
<td>692</td>
<td>REG</td>
<td>11-75</td>
</tr>
<tr>
<td>mark</td>
<td>src, dst</td>
<td>66B</td>
<td>REG</td>
<td>11-78</td>
</tr>
<tr>
<td>modac</td>
<td>mask, src, dst</td>
<td>645</td>
<td>REG</td>
<td>11-79</td>
</tr>
<tr>
<td>modi</td>
<td>src1, src2, dst</td>
<td>749</td>
<td>REG</td>
<td>11-80</td>
</tr>
<tr>
<td>modify</td>
<td>mask, src, src/dst</td>
<td>650</td>
<td>REG</td>
<td>11-81</td>
</tr>
<tr>
<td>modpc</td>
<td>src, mask, src/dst</td>
<td>655</td>
<td>REG</td>
<td>11-82</td>
</tr>
<tr>
<td>modtc</td>
<td>mask, src, dst</td>
<td>654</td>
<td>REG</td>
<td>11-84</td>
</tr>
<tr>
<td>mov</td>
<td>src, dst</td>
<td>5CC</td>
<td>REG</td>
<td>11-85</td>
</tr>
<tr>
<td>movl</td>
<td>src, dst</td>
<td>5DC</td>
<td>REG</td>
<td>11-85</td>
</tr>
<tr>
<td>movq</td>
<td>src, dst</td>
<td>5FC</td>
<td>REG</td>
<td>11-85</td>
</tr>
<tr>
<td>movr</td>
<td>src, dst</td>
<td>6C9</td>
<td>REG</td>
<td>11-86</td>
</tr>
<tr>
<td>movre</td>
<td>src, dst</td>
<td>6E9</td>
<td>REG</td>
<td>11-86</td>
</tr>
<tr>
<td>movrl</td>
<td>src, dst</td>
<td>6D9</td>
<td>REG</td>
<td>11-86</td>
</tr>
<tr>
<td>movt</td>
<td>src, dst</td>
<td>5EC</td>
<td>REG</td>
<td>11-85</td>
</tr>
<tr>
<td>multi</td>
<td>src1, src2, dst</td>
<td>741</td>
<td>REG</td>
<td>11-88</td>
</tr>
<tr>
<td>mulo</td>
<td>src1, src2, dst</td>
<td>701</td>
<td>REG</td>
<td>11-88</td>
</tr>
<tr>
<td>mulr</td>
<td>src1, src2, dst</td>
<td>78C</td>
<td>REG</td>
<td>11-89</td>
</tr>
<tr>
<td>mulrl</td>
<td>src1, src2, dst</td>
<td>79C</td>
<td>REG</td>
<td>11-89</td>
</tr>
<tr>
<td>nand</td>
<td>src1, src2, dst</td>
<td>58E</td>
<td>REG</td>
<td>11-91</td>
</tr>
<tr>
<td>nor</td>
<td>src1, src2, dst</td>
<td>588</td>
<td>REG</td>
<td>11-92</td>
</tr>
<tr>
<td>not</td>
<td>src, dst</td>
<td>58A</td>
<td>REG</td>
<td>11-93</td>
</tr>
<tr>
<td>notand</td>
<td>src, dst</td>
<td>584</td>
<td>REG</td>
<td>11-93</td>
</tr>
<tr>
<td>notbit</td>
<td>bitpos, src, dst</td>
<td>580</td>
<td>REG</td>
<td>11-94</td>
</tr>
<tr>
<td>notor</td>
<td>src1, src2, dst</td>
<td>58D</td>
<td>REG</td>
<td>11-95</td>
</tr>
<tr>
<td>or</td>
<td>src1, src2, dst</td>
<td>587</td>
<td>REG</td>
<td>11-96</td>
</tr>
<tr>
<td>ornor</td>
<td>src1, src2, dst</td>
<td>58B</td>
<td>REG</td>
<td>11-96</td>
</tr>
<tr>
<td>remi</td>
<td>src1, src2, dst</td>
<td>748</td>
<td>REG</td>
<td>11-97</td>
</tr>
<tr>
<td>remo</td>
<td>src1, src2, dst</td>
<td>708</td>
<td>REG</td>
<td>11-97</td>
</tr>
<tr>
<td>remr</td>
<td>src1, src2, dst</td>
<td>683</td>
<td>REG</td>
<td>11-98</td>
</tr>
<tr>
<td>remrl</td>
<td>src1, src2, dst</td>
<td>693</td>
<td>REG</td>
<td>11-98</td>
</tr>
<tr>
<td>ret</td>
<td>src, dst</td>
<td>0A</td>
<td>CTRL</td>
<td>11-101</td>
</tr>
<tr>
<td>rotate</td>
<td>len, src, dst</td>
<td>59D</td>
<td>REG</td>
<td>11-103</td>
</tr>
<tr>
<td>roundr</td>
<td>src, dst</td>
<td>68B</td>
<td>REG</td>
<td>11-104</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operands</td>
<td>Opcode</td>
<td>Inst. Type</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>--------</td>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>roundrl</td>
<td>src, dst</td>
<td>69B</td>
<td>REG</td>
<td>11-104</td>
</tr>
<tr>
<td>scaler</td>
<td>src1, dst</td>
<td>677</td>
<td>REG</td>
<td>11-105</td>
</tr>
<tr>
<td>scalerl</td>
<td>src1, dst</td>
<td>676</td>
<td>REG</td>
<td>11-105</td>
</tr>
<tr>
<td>scanbit</td>
<td>src, dst</td>
<td>641</td>
<td>REG</td>
<td>11-107</td>
</tr>
<tr>
<td>scanbyte</td>
<td>src1, src2</td>
<td>5AC</td>
<td>REG</td>
<td>11-108</td>
</tr>
<tr>
<td>setbit</td>
<td>bitpos, src, dst</td>
<td>583</td>
<td>REG</td>
<td>11-109</td>
</tr>
<tr>
<td>shli</td>
<td>len, src, dst</td>
<td>593</td>
<td>REG</td>
<td>11-110</td>
</tr>
<tr>
<td>shlo</td>
<td>len, src, dst</td>
<td>59C</td>
<td>REG</td>
<td>11-110</td>
</tr>
<tr>
<td>shri</td>
<td>len, src, dst</td>
<td>59A</td>
<td>REG</td>
<td>11-110</td>
</tr>
<tr>
<td>shro</td>
<td>len, src, dst</td>
<td>59B</td>
<td>REG</td>
<td>11-110</td>
</tr>
<tr>
<td>sinr</td>
<td>src, dst</td>
<td>68C</td>
<td>REG</td>
<td>11-112</td>
</tr>
<tr>
<td>sinrl</td>
<td>src, dst</td>
<td>69C</td>
<td>REG</td>
<td>11-112</td>
</tr>
<tr>
<td>spanbit</td>
<td>src, dst</td>
<td>640</td>
<td>REG</td>
<td>11-114</td>
</tr>
<tr>
<td>sqrtr</td>
<td>src, dst</td>
<td>688</td>
<td>REG</td>
<td>11-115</td>
</tr>
<tr>
<td>sqrtrl</td>
<td>src, dst</td>
<td>698</td>
<td>REG</td>
<td>11-115</td>
</tr>
<tr>
<td>st</td>
<td>src, dst</td>
<td>92</td>
<td>MEM</td>
<td>11-117</td>
</tr>
<tr>
<td>stib</td>
<td>src, dst</td>
<td>C2</td>
<td>MEM</td>
<td>11-117</td>
</tr>
<tr>
<td>stis</td>
<td>src, dst</td>
<td>CA</td>
<td>MEM</td>
<td>11-117</td>
</tr>
<tr>
<td>stl</td>
<td>src, dst</td>
<td>9A</td>
<td>MEM</td>
<td>11-117</td>
</tr>
<tr>
<td>stob</td>
<td>src, dst</td>
<td>82</td>
<td>MEM</td>
<td>11-117</td>
</tr>
<tr>
<td>stos</td>
<td>src, dst</td>
<td>8A</td>
<td>MEM</td>
<td>11-117</td>
</tr>
<tr>
<td>stq</td>
<td>src, dst</td>
<td>B2</td>
<td>MEM</td>
<td>11-117</td>
</tr>
<tr>
<td>stt</td>
<td>src, dst</td>
<td>A2</td>
<td>MEM</td>
<td>11-117</td>
</tr>
<tr>
<td>subc</td>
<td>src1, src2, dst</td>
<td>5B2</td>
<td>REG</td>
<td>11-119</td>
</tr>
<tr>
<td>subi</td>
<td>src1, src2, dst</td>
<td>593</td>
<td>REG</td>
<td>11-120</td>
</tr>
<tr>
<td>subo</td>
<td>src1, src2, dst</td>
<td>592</td>
<td>REG</td>
<td>11-120</td>
</tr>
<tr>
<td>subbr</td>
<td>src1, src2, dst</td>
<td>78D</td>
<td>REG</td>
<td>11-121</td>
</tr>
<tr>
<td>subrl</td>
<td>src1, src2, dst</td>
<td>79D</td>
<td>REG</td>
<td>11-121</td>
</tr>
<tr>
<td>syncf</td>
<td></td>
<td>66F</td>
<td>REG</td>
<td>11-123</td>
</tr>
<tr>
<td>synld</td>
<td>src, dst</td>
<td>615</td>
<td>REG</td>
<td>11-124</td>
</tr>
<tr>
<td>synmov</td>
<td>dst, src</td>
<td>600</td>
<td>REG</td>
<td>11-126</td>
</tr>
<tr>
<td>synmovl</td>
<td>dst, src</td>
<td>601</td>
<td>REG</td>
<td>11-126</td>
</tr>
<tr>
<td>synmovq</td>
<td>dst, src</td>
<td>602</td>
<td>REG</td>
<td>11-126</td>
</tr>
<tr>
<td>tanr</td>
<td>src, dst</td>
<td>68E</td>
<td>REG</td>
<td>11-129</td>
</tr>
<tr>
<td>tanrl</td>
<td>src, dst</td>
<td>69E</td>
<td>REG</td>
<td>11-129</td>
</tr>
<tr>
<td>teste</td>
<td>dst</td>
<td>22</td>
<td>COBR</td>
<td>11-131</td>
</tr>
<tr>
<td>testg</td>
<td>dst</td>
<td>21</td>
<td>COBR</td>
<td>11-131</td>
</tr>
<tr>
<td>testge</td>
<td>dst</td>
<td>23</td>
<td>COBR</td>
<td>11-131</td>
</tr>
<tr>
<td>testl</td>
<td>dst</td>
<td>24</td>
<td>COBR</td>
<td>11-131</td>
</tr>
<tr>
<td>testle</td>
<td>dst</td>
<td>26</td>
<td>COBR</td>
<td>11-131</td>
</tr>
<tr>
<td>testne</td>
<td>dst</td>
<td>25</td>
<td>COBR</td>
<td>11-131</td>
</tr>
<tr>
<td>testno</td>
<td>dst</td>
<td>20</td>
<td>COBR</td>
<td>11-131</td>
</tr>
<tr>
<td>testo</td>
<td>dst</td>
<td>27</td>
<td>COBR</td>
<td>11-131</td>
</tr>
<tr>
<td>xnor</td>
<td>src1, src2, dst</td>
<td>589</td>
<td>REG</td>
<td>11-133</td>
</tr>
<tr>
<td>xor</td>
<td>src1, src2, dst</td>
<td>586</td>
<td>REG</td>
<td>11-133</td>
</tr>
</tbody>
</table>
### Instruction List by Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Inst. Type</th>
<th>Mnemonic</th>
<th>Operands</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>CTRL</td>
<td>b</td>
<td>targ</td>
<td>11-18</td>
</tr>
<tr>
<td>09</td>
<td>CTRL</td>
<td>call</td>
<td>targ</td>
<td>11-25</td>
</tr>
<tr>
<td>0A</td>
<td>CTRL</td>
<td>ret</td>
<td></td>
<td>11-101</td>
</tr>
<tr>
<td>0B</td>
<td>CTRL</td>
<td>bal</td>
<td>targ</td>
<td>11-16</td>
</tr>
<tr>
<td>10</td>
<td>CTRL</td>
<td>bno</td>
<td>targ</td>
<td>11-22</td>
</tr>
<tr>
<td>11</td>
<td>CTRL</td>
<td>bg</td>
<td>targ</td>
<td>11-22</td>
</tr>
<tr>
<td>12</td>
<td>CTRL</td>
<td>be</td>
<td>targ</td>
<td>11-22</td>
</tr>
<tr>
<td>13</td>
<td>CTRL</td>
<td>bge</td>
<td>targ</td>
<td>11-22</td>
</tr>
<tr>
<td>14</td>
<td>CTRL</td>
<td>bl</td>
<td>targ</td>
<td>11-22</td>
</tr>
<tr>
<td>15</td>
<td>CTRL</td>
<td>bne</td>
<td>targ</td>
<td>11-22</td>
</tr>
<tr>
<td>16</td>
<td>CTRL</td>
<td>ble</td>
<td>targ</td>
<td>11-22</td>
</tr>
<tr>
<td>17</td>
<td>CTRL</td>
<td>bo</td>
<td></td>
<td>11-22</td>
</tr>
<tr>
<td>18</td>
<td>CTRL</td>
<td>faultno</td>
<td></td>
<td>11-63</td>
</tr>
<tr>
<td>19</td>
<td>CTRL</td>
<td>faultg</td>
<td></td>
<td>11-63</td>
</tr>
<tr>
<td>1A</td>
<td>CTRL</td>
<td>faulte</td>
<td></td>
<td>11-63</td>
</tr>
<tr>
<td>1B</td>
<td>CTRL</td>
<td>faultge</td>
<td></td>
<td>11-63</td>
</tr>
<tr>
<td>1C</td>
<td>CTRL</td>
<td>faultl</td>
<td></td>
<td>11-63</td>
</tr>
<tr>
<td>1D</td>
<td>CTRL</td>
<td>faultne</td>
<td></td>
<td>11-63</td>
</tr>
<tr>
<td>1E</td>
<td>CTRL</td>
<td>faultle</td>
<td></td>
<td>11-63</td>
</tr>
<tr>
<td>1F</td>
<td>CTRL</td>
<td>faulto</td>
<td></td>
<td>11-63</td>
</tr>
<tr>
<td>20</td>
<td>COBR</td>
<td>testno</td>
<td>dst</td>
<td>11-131</td>
</tr>
<tr>
<td>21</td>
<td>COBR</td>
<td>testg</td>
<td>dst</td>
<td>11-131</td>
</tr>
<tr>
<td>22</td>
<td>COBR</td>
<td>teste</td>
<td>dst</td>
<td>11-131</td>
</tr>
<tr>
<td>23</td>
<td>COBR</td>
<td>testge</td>
<td>dst</td>
<td>11-131</td>
</tr>
<tr>
<td>24</td>
<td>COBR</td>
<td>testl</td>
<td>dst</td>
<td>11-131</td>
</tr>
<tr>
<td>25</td>
<td>COBR</td>
<td>testne</td>
<td>dst</td>
<td>11-131</td>
</tr>
<tr>
<td>26</td>
<td>COBR</td>
<td>testle</td>
<td>dst</td>
<td>11-131</td>
</tr>
<tr>
<td>27</td>
<td>COBR</td>
<td>testo</td>
<td>dst</td>
<td>11-131</td>
</tr>
<tr>
<td>30</td>
<td>COBR</td>
<td>bbc</td>
<td>bitpos,</td>
<td>11-20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>COBR</td>
<td>cmpobg</td>
<td>src1,</td>
<td>11-18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>COBR</td>
<td>cmpobe</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>COBR</td>
<td>cmpobge</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>COBR</td>
<td>cmpobl</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>COBR</td>
<td>cmpobne</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>COBR</td>
<td>cmpoble</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>COBR</td>
<td>bbs</td>
<td>bitpos,</td>
<td>11-20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>COBR</td>
<td>cmpibno</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>COBR</td>
<td>cmpibg</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>3A</td>
<td>COBR</td>
<td>cmpibp</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>3B</td>
<td>COBR</td>
<td>cmpipbge</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td>COBR</td>
<td>cmpipbl</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>3D</td>
<td>COBR</td>
<td>cmpiibne</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>3E</td>
<td>COBR</td>
<td>cmpible</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>3F</td>
<td>COBR</td>
<td>cmpipibo</td>
<td>src1,</td>
<td>11-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>targ</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>MEM</td>
<td>ldbb</td>
<td>src,</td>
<td>11-67</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dst</td>
<td></td>
</tr>
<tr>
<td>Opcode</td>
<td>Inst. Type</td>
<td>Mnemonic</td>
<td>Operands</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>82</td>
<td>MEM</td>
<td>stob</td>
<td>src, dst</td>
<td>11-117</td>
</tr>
<tr>
<td>84</td>
<td>MEM</td>
<td>bx</td>
<td>targ</td>
<td>11-18</td>
</tr>
<tr>
<td>85</td>
<td>MEM</td>
<td>balx</td>
<td>targ, dst</td>
<td>11-16</td>
</tr>
<tr>
<td>86</td>
<td>MEM</td>
<td>callx</td>
<td>targ</td>
<td>11-29</td>
</tr>
<tr>
<td>88</td>
<td>MEM</td>
<td>ldos</td>
<td>src, dst</td>
<td>11-67</td>
</tr>
<tr>
<td>8A</td>
<td>MEM</td>
<td>stos</td>
<td>src, dst</td>
<td>11-117</td>
</tr>
<tr>
<td>8C</td>
<td>MEM</td>
<td>lda</td>
<td>src</td>
<td>11-69</td>
</tr>
<tr>
<td>90</td>
<td>MEM</td>
<td>ld</td>
<td>src, dst</td>
<td>11-67</td>
</tr>
<tr>
<td>92</td>
<td>MEM</td>
<td>st</td>
<td>src, dst</td>
<td>11-117</td>
</tr>
<tr>
<td>98</td>
<td>MEM</td>
<td>ldl</td>
<td>src, dst</td>
<td>11-67</td>
</tr>
<tr>
<td>9A</td>
<td>MEM</td>
<td>stl</td>
<td>src, dst</td>
<td>11-67</td>
</tr>
<tr>
<td>A0</td>
<td>MEM</td>
<td>ldt</td>
<td>src, dst</td>
<td>11-117</td>
</tr>
<tr>
<td>A2</td>
<td>MEM</td>
<td>stt</td>
<td>src, dst</td>
<td>11-67</td>
</tr>
<tr>
<td>B0</td>
<td>MEM</td>
<td>ldq</td>
<td>src, dst</td>
<td>11-117</td>
</tr>
<tr>
<td>B2</td>
<td>MEM</td>
<td>stq</td>
<td>src, dst</td>
<td>11-67</td>
</tr>
<tr>
<td>C0</td>
<td>MEM</td>
<td>ldib</td>
<td>src, dst</td>
<td>11-67</td>
</tr>
<tr>
<td>C2</td>
<td>MEM</td>
<td>stib</td>
<td>src, dst</td>
<td>11-117</td>
</tr>
<tr>
<td>C8</td>
<td>MEM</td>
<td>ldis</td>
<td>src, dst</td>
<td>11-67</td>
</tr>
<tr>
<td>CA</td>
<td>MEM</td>
<td>stis</td>
<td>src, dst</td>
<td>11-117</td>
</tr>
<tr>
<td>580</td>
<td>REG</td>
<td>notbit</td>
<td>bitpos, src, dst</td>
<td>11-94</td>
</tr>
<tr>
<td>581</td>
<td>REG</td>
<td>and</td>
<td>src1, src2, dst</td>
<td>11-11</td>
</tr>
<tr>
<td>582</td>
<td>REG</td>
<td>andnot</td>
<td>src1, src2, dst</td>
<td>11-11</td>
</tr>
<tr>
<td>583</td>
<td>REG</td>
<td>setbit</td>
<td>bitpos, src, dst</td>
<td>11-109</td>
</tr>
<tr>
<td>584</td>
<td>REG</td>
<td>notand</td>
<td>src, dst</td>
<td>11-93</td>
</tr>
<tr>
<td>586</td>
<td>REG</td>
<td>xor</td>
<td>src1, src2, dst</td>
<td>11-133</td>
</tr>
<tr>
<td>587</td>
<td>REG</td>
<td>or</td>
<td>src1, src2, dst</td>
<td>11-96</td>
</tr>
<tr>
<td>588</td>
<td>REG</td>
<td>nor</td>
<td>src1, src2, dst</td>
<td>11-92</td>
</tr>
<tr>
<td>589</td>
<td>REG</td>
<td>xnor</td>
<td>src1, src2, dst</td>
<td>11-133</td>
</tr>
<tr>
<td>58A</td>
<td>REG</td>
<td>not</td>
<td>src, dst</td>
<td>11-93</td>
</tr>
<tr>
<td>58B</td>
<td>REG</td>
<td>ornot</td>
<td>src1, src2, dst</td>
<td>11-96</td>
</tr>
<tr>
<td>58C</td>
<td>REG</td>
<td>clrbits</td>
<td>bitpos, src, dst</td>
<td>11-34</td>
</tr>
<tr>
<td>58D</td>
<td>REG</td>
<td>notor</td>
<td>src1, src2, dst</td>
<td>11-95</td>
</tr>
<tr>
<td>58E</td>
<td>REG</td>
<td>nand</td>
<td>src1, src2, dst</td>
<td>11-91</td>
</tr>
<tr>
<td>58F</td>
<td>REG</td>
<td>alterbit</td>
<td>bitpos, src, dst</td>
<td>11-10</td>
</tr>
<tr>
<td>590</td>
<td>REG</td>
<td>addo</td>
<td>src1, src2, dst</td>
<td>11-7</td>
</tr>
<tr>
<td>591</td>
<td>REG</td>
<td>addi</td>
<td>src1, src2, dst</td>
<td>11-7</td>
</tr>
<tr>
<td>592</td>
<td>REG</td>
<td>subo</td>
<td>src1, src2, dst</td>
<td>11-120</td>
</tr>
<tr>
<td>593</td>
<td>REG</td>
<td>subi</td>
<td>src1, src2, dst</td>
<td>11-120</td>
</tr>
<tr>
<td>598</td>
<td>REG</td>
<td>shro</td>
<td>len, src, dst</td>
<td>11-110</td>
</tr>
<tr>
<td>59A</td>
<td>REG</td>
<td>shrdi</td>
<td>len, src, dst</td>
<td>11-110</td>
</tr>
<tr>
<td>59B</td>
<td>REG</td>
<td>shri</td>
<td>len, src, dst</td>
<td>11-110</td>
</tr>
<tr>
<td>59C</td>
<td>REG</td>
<td>shlo</td>
<td>len, src, dst</td>
<td>11-110</td>
</tr>
<tr>
<td>59D</td>
<td>REG</td>
<td>rotate</td>
<td>len, src, dst</td>
<td>11-103</td>
</tr>
<tr>
<td>59E</td>
<td>REG</td>
<td>shli</td>
<td>len, src, dst</td>
<td>11-110</td>
</tr>
<tr>
<td>5A0</td>
<td>REG</td>
<td>cmpo</td>
<td>src1, src2</td>
<td>11-35</td>
</tr>
<tr>
<td>5A1</td>
<td>REG</td>
<td>cmpi</td>
<td>src1, src2</td>
<td>11-35</td>
</tr>
<tr>
<td>5A2</td>
<td>REG</td>
<td>concmpo</td>
<td>src1, src2</td>
<td>11-45</td>
</tr>
<tr>
<td>Opcode</td>
<td>Inst. Type</td>
<td>Mnemonic</td>
<td>Operands</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>5A3</td>
<td>REG</td>
<td>concmpi</td>
<td>src1, src2</td>
<td>11-45</td>
</tr>
<tr>
<td>5A4</td>
<td>REG</td>
<td>cmpinco</td>
<td>src1, src2, dst</td>
<td>11-37</td>
</tr>
<tr>
<td>5A5</td>
<td>REG</td>
<td>cmpincli</td>
<td>src1, src2, dst</td>
<td>11-37</td>
</tr>
<tr>
<td>5A6</td>
<td>REG</td>
<td>cmpdeco</td>
<td>src1, src2, dst</td>
<td>11-36</td>
</tr>
<tr>
<td>5A7</td>
<td>REG</td>
<td>cmpdeci</td>
<td>src1, src2, dst</td>
<td>11-36</td>
</tr>
<tr>
<td>5A8</td>
<td>REG</td>
<td>scanbyte</td>
<td>src1, src2</td>
<td>11-108</td>
</tr>
<tr>
<td>5A9</td>
<td>REG</td>
<td>chkbit</td>
<td>bitpos, src</td>
<td>11-31</td>
</tr>
<tr>
<td>5B0</td>
<td>REG</td>
<td>adde</td>
<td>src1, src2, dst</td>
<td>11-6</td>
</tr>
<tr>
<td>5B1</td>
<td>REG</td>
<td>subc</td>
<td>src1, src2, dst</td>
<td>11-119</td>
</tr>
<tr>
<td>5C8</td>
<td>REG</td>
<td>mov</td>
<td>src, dst</td>
<td>11-85</td>
</tr>
<tr>
<td>5D2</td>
<td>REG</td>
<td>movt</td>
<td>src, dst</td>
<td>11-85</td>
</tr>
<tr>
<td>5E8</td>
<td>REG</td>
<td>movt</td>
<td>src, dst</td>
<td>11-85</td>
</tr>
<tr>
<td>5F2</td>
<td>REG</td>
<td>synmov</td>
<td>dst, src</td>
<td>11-126</td>
</tr>
<tr>
<td>600</td>
<td>REG</td>
<td>synmovl</td>
<td>dst, src</td>
<td>11-126</td>
</tr>
<tr>
<td>601</td>
<td>REG</td>
<td>synmovq</td>
<td>dst, src</td>
<td>11-126</td>
</tr>
<tr>
<td>610</td>
<td>REG</td>
<td>atmmod</td>
<td>src, mask, src/dst</td>
<td>11-15</td>
</tr>
<tr>
<td>612</td>
<td>REG</td>
<td>atadd</td>
<td>src/dst, src, dst</td>
<td>11-12</td>
</tr>
<tr>
<td>615</td>
<td>REG</td>
<td>synld</td>
<td>src, dst</td>
<td>11-124</td>
</tr>
<tr>
<td>640</td>
<td>REG</td>
<td>spanbit</td>
<td>src, dst</td>
<td>11-114</td>
</tr>
<tr>
<td>641</td>
<td>REG</td>
<td>scanbit</td>
<td>src, dst</td>
<td>11-107</td>
</tr>
<tr>
<td>642</td>
<td>REG</td>
<td>dadde</td>
<td>src1, src2, dst</td>
<td>11-52</td>
</tr>
<tr>
<td>643</td>
<td>REG</td>
<td>dsbdc</td>
<td>src1, src2, dst</td>
<td>11-57</td>
</tr>
<tr>
<td>644</td>
<td>REG</td>
<td>dmovt</td>
<td>src, dst</td>
<td>11-56</td>
</tr>
<tr>
<td>645</td>
<td>REG</td>
<td>modac</td>
<td>mask, src, dst</td>
<td>11-79</td>
</tr>
<tr>
<td>650</td>
<td>REG</td>
<td>modify</td>
<td>mask, src, src/dst</td>
<td>11-81</td>
</tr>
<tr>
<td>651</td>
<td>REG</td>
<td>extract</td>
<td>bitpos, len, src/dst</td>
<td>11-62</td>
</tr>
<tr>
<td>654</td>
<td>REG</td>
<td>modt</td>
<td>mask, src, dst</td>
<td>11-84</td>
</tr>
<tr>
<td>655</td>
<td>REG</td>
<td>modtp</td>
<td>mask, src/dst</td>
<td>11-82</td>
</tr>
<tr>
<td>660</td>
<td>REG</td>
<td>calls</td>
<td>targ</td>
<td>11-27</td>
</tr>
<tr>
<td>66B</td>
<td>REG</td>
<td>mark</td>
<td></td>
<td>11-78</td>
</tr>
<tr>
<td>66C</td>
<td>REG</td>
<td>fmark</td>
<td></td>
<td>11-66</td>
</tr>
<tr>
<td>66D</td>
<td>REG</td>
<td>flushreg</td>
<td></td>
<td>11-65</td>
</tr>
<tr>
<td>66F</td>
<td>REG</td>
<td>syncf</td>
<td></td>
<td>11-123</td>
</tr>
<tr>
<td>670</td>
<td>REG</td>
<td>emul</td>
<td>src1, src2, dst</td>
<td>11-59</td>
</tr>
<tr>
<td>671</td>
<td>REG</td>
<td>ediv</td>
<td>src1, src2, dst</td>
<td>11-58</td>
</tr>
<tr>
<td>674</td>
<td>REG</td>
<td>cvtir</td>
<td>src, dst</td>
<td>11-49</td>
</tr>
<tr>
<td>675</td>
<td>REG</td>
<td>cvtflr</td>
<td>src, dst</td>
<td>11-49</td>
</tr>
<tr>
<td>676</td>
<td>REG</td>
<td>scalerl</td>
<td>src1, src2, dst</td>
<td>11-105</td>
</tr>
<tr>
<td>677</td>
<td>REG</td>
<td>scaler</td>
<td>src1, src2, dst</td>
<td>11-105</td>
</tr>
<tr>
<td>680</td>
<td>REG</td>
<td>atanr</td>
<td>src1, src2, dst</td>
<td>11-13</td>
</tr>
<tr>
<td>681</td>
<td>REG</td>
<td>logepr</td>
<td>src1, src2, dst</td>
<td>11-72</td>
</tr>
<tr>
<td>682</td>
<td>REG</td>
<td>logr</td>
<td>src1, src2, dst</td>
<td>11-75</td>
</tr>
<tr>
<td>683</td>
<td>REG</td>
<td>remr</td>
<td>src1, src2, dst</td>
<td>11-98</td>
</tr>
<tr>
<td>684</td>
<td>REG</td>
<td>cmpor</td>
<td>src1, src2</td>
<td>11-38</td>
</tr>
<tr>
<td>685</td>
<td>REG</td>
<td>cmpr</td>
<td>src1, src2</td>
<td>11-40</td>
</tr>
<tr>
<td>688</td>
<td>REG</td>
<td>sqrtr</td>
<td>src, dst</td>
<td>11-115</td>
</tr>
<tr>
<td>Opcode</td>
<td>Inst. Type</td>
<td>Mnemonic</td>
<td>Operands</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>689</td>
<td>REG REPORT</td>
<td>expr</td>
<td>src, dst</td>
<td>11-60</td>
</tr>
<tr>
<td>68A</td>
<td>REG REPORT</td>
<td>logbnr</td>
<td>src, dst</td>
<td>11-70</td>
</tr>
<tr>
<td>68B</td>
<td>REG REPORT</td>
<td>roundr</td>
<td>src, dst</td>
<td>11-104</td>
</tr>
<tr>
<td>68C</td>
<td>REG REPORT</td>
<td>sinn</td>
<td>src, dst</td>
<td>11-112</td>
</tr>
<tr>
<td>68D</td>
<td>REG REPORT</td>
<td>cosr</td>
<td>src, dst</td>
<td>11-46</td>
</tr>
<tr>
<td>68E</td>
<td>REG REPORT</td>
<td>tanr</td>
<td>src, dst</td>
<td>11-129</td>
</tr>
<tr>
<td>68F</td>
<td>REG REPORT</td>
<td>clsrcr</td>
<td>src</td>
<td>11-32</td>
</tr>
<tr>
<td>690</td>
<td>REG REPORT</td>
<td>atanrl</td>
<td>src1, src2, dst</td>
<td>11-13</td>
</tr>
<tr>
<td>691</td>
<td>REG REPORT</td>
<td>logeprl</td>
<td>src1, src2, dst</td>
<td>11-72</td>
</tr>
<tr>
<td>692</td>
<td>REG REPORT</td>
<td>logrl</td>
<td>src1, src2, dst</td>
<td>11-75</td>
</tr>
<tr>
<td>693</td>
<td>REG REPORT</td>
<td>remrl</td>
<td>src1, src2, dst</td>
<td>11-98</td>
</tr>
<tr>
<td>694</td>
<td>REG REPORT</td>
<td>cmporl</td>
<td>src1, src2</td>
<td>11-38</td>
</tr>
<tr>
<td>695</td>
<td>REG REPORT</td>
<td>cmprl</td>
<td>src1, src2</td>
<td>11-40</td>
</tr>
<tr>
<td>698</td>
<td>REG REPORT</td>
<td>sqrrl</td>
<td>src, dst</td>
<td>11-115</td>
</tr>
<tr>
<td>699</td>
<td>REG REPORT</td>
<td>exprl</td>
<td>src, dst</td>
<td>11-60</td>
</tr>
<tr>
<td>69A</td>
<td>REG REPORT</td>
<td>logbnrl</td>
<td>src, dst</td>
<td>11-70</td>
</tr>
<tr>
<td>69B</td>
<td>REG REPORT</td>
<td>roundrl</td>
<td>src, dst</td>
<td>11-104</td>
</tr>
<tr>
<td>69C</td>
<td>REG REPORT</td>
<td>sinrl</td>
<td>src, dst</td>
<td>11-112</td>
</tr>
<tr>
<td>69D</td>
<td>REG REPORT</td>
<td>cosrl</td>
<td>src, dst</td>
<td>11-46</td>
</tr>
<tr>
<td>69E</td>
<td>REG REPORT</td>
<td>tanrl</td>
<td>src, dst</td>
<td>11-129</td>
</tr>
<tr>
<td>69F</td>
<td>REG REPORT</td>
<td>clsrcrl</td>
<td>src</td>
<td>11-32</td>
</tr>
<tr>
<td>6C0</td>
<td>REG REPORT</td>
<td>cvtri</td>
<td>src, dst</td>
<td>11-50</td>
</tr>
<tr>
<td>6C1</td>
<td>REG REPORT</td>
<td>cvtril</td>
<td>src, dst</td>
<td>11-50</td>
</tr>
<tr>
<td>6C2</td>
<td>REG REPORT</td>
<td>cvtzri</td>
<td>src, dst</td>
<td>11-50</td>
</tr>
<tr>
<td>6C3</td>
<td>REG REPORT</td>
<td>cvtzrli</td>
<td>src, dst</td>
<td>11-50</td>
</tr>
<tr>
<td>6C9</td>
<td>REG REPORT</td>
<td>movr</td>
<td>src, dst</td>
<td>11-86</td>
</tr>
<tr>
<td>6D9</td>
<td>REG REPORT</td>
<td>movrl</td>
<td>src, dst</td>
<td>11-86</td>
</tr>
<tr>
<td>6E2</td>
<td>REG REPORT</td>
<td>cpysre</td>
<td>src1, src2, dst</td>
<td>11-48</td>
</tr>
<tr>
<td>6E3</td>
<td>REG REPORT</td>
<td>cpysrse</td>
<td>src1, src2, dst</td>
<td>11-48</td>
</tr>
<tr>
<td>6E9</td>
<td>REG REPORT</td>
<td>movre</td>
<td>src, dst</td>
<td>11-86</td>
</tr>
<tr>
<td>701</td>
<td>REG REPORT</td>
<td>mulo</td>
<td>src1, src2, dst</td>
<td>11-88</td>
</tr>
<tr>
<td>708</td>
<td>REG REPORT</td>
<td>remo</td>
<td>src1, src2, dst</td>
<td>11-97</td>
</tr>
<tr>
<td>70B</td>
<td>REG REPORT</td>
<td>divo</td>
<td>src1, src2, dst</td>
<td>11-53</td>
</tr>
<tr>
<td>741</td>
<td>REG REPORT</td>
<td>muli</td>
<td>src1, src2, dst</td>
<td>11-88</td>
</tr>
<tr>
<td>748</td>
<td>REG REPORT</td>
<td>remi</td>
<td>src1, src2, dst</td>
<td>11-97</td>
</tr>
<tr>
<td>749</td>
<td>REG REPORT</td>
<td>modi</td>
<td>src1, src2, dst</td>
<td>11-80</td>
</tr>
<tr>
<td>74B</td>
<td>REG REPORT</td>
<td>divi</td>
<td>src1, src2, dst</td>
<td>11-53</td>
</tr>
<tr>
<td>78B</td>
<td>REG REPORT</td>
<td>divr</td>
<td>src1, src2, dst</td>
<td>11-54</td>
</tr>
<tr>
<td>78C</td>
<td>REG REPORT</td>
<td>mulr</td>
<td>src1, src2, dst</td>
<td>11-89</td>
</tr>
<tr>
<td>78D</td>
<td>REG REPORT</td>
<td>subr</td>
<td>src1, src2, dst</td>
<td>11-121</td>
</tr>
<tr>
<td>78F</td>
<td>REG REPORT</td>
<td>addr</td>
<td>src1, src2, dst</td>
<td>11-8</td>
</tr>
<tr>
<td>79B</td>
<td>REG REPORT</td>
<td>divrl</td>
<td>src1, src2, dst</td>
<td>11-54</td>
</tr>
<tr>
<td>79C</td>
<td>REG REPORT</td>
<td>mulrl</td>
<td>src1, src2, dst</td>
<td>11-89</td>
</tr>
<tr>
<td>79D</td>
<td>REG REPORT</td>
<td>subrl</td>
<td>src1, src2, dst</td>
<td>11-121</td>
</tr>
<tr>
<td>79F</td>
<td>REG REPORT</td>
<td>addrl</td>
<td>src1, src2, dst</td>
<td>11-8</td>
</tr>
</tbody>
</table>
SUMMARY OF SYSTEM DATA STRUCTURES

The following pages provide a collection of the system data structures presented in this manual. They are grouped by function. The chapter reference below each data structure shows where in this manual this data structure is described.

Execution Environment

![Diagram of Execution Environment](image)

Figure A-1: Arithmetic Controls (Chapter 3)
Figure A-2: Registers Available to a Single Procedure (Chapter 3)
Figure A-3: Procedure Stack Structure (Chapter 4)
Processor Management

Figure A-4: Process Controls (Chapter 7)
## Check-Sum Words

<table>
<thead>
<tr>
<th>Physical Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT Pointer</td>
</tr>
<tr>
<td>PRCB Pointer</td>
</tr>
<tr>
<td>Check Word</td>
</tr>
<tr>
<td>Instruction Pointer</td>
</tr>
<tr>
<td>4 Check Words</td>
</tr>
</tbody>
</table>

## System Address Table (SAT)

<table>
<thead>
<tr>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>136</td>
</tr>
<tr>
<td>140</td>
</tr>
<tr>
<td>144</td>
</tr>
<tr>
<td>148</td>
</tr>
<tr>
<td>152</td>
</tr>
<tr>
<td>156</td>
</tr>
</tbody>
</table>

- Pointer to Offset 0
- 00FC 00FB16
- System Procedure Pointer
- 3044 00FB16

## Processor Control Block (PRCB)

<table>
<thead>
<tr>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>28</td>
</tr>
</tbody>
</table>

- W 0000 000016
- Interrupt Table Pointer
- Interrupt Stack Pointer
- 0000 027F16
- 0000 027F16
- Fault Table Pointer
- 0000 00016
- Scratch Space

## Initialization Code

<table>
<thead>
<tr>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Figure A-5: Initial Memory Image (Chapter 7)
Interrupt Handling

Figure A-6: Interrupt Table (Chapter 8)
*If the interrupt is serviced while the processor is working on another interrupt procedure, the new stack pointer (NSP) will be the same as the SP.*
IACs

![IAC Message Format](image1)

Figure A-8: IAC Message Format (Chapter 13)

Fault Handling

![Fault Record](image2)

Figure A-9: Fault Record (Chapter 9)
### Figure A-10: Fault Table and Fault-Table Entries (Chapter 9)

The table illustrates the distribution of fault entries in a 256-word fault table. Each entry is 4 bytes long, with the address field occupying 32 bits (`n`), and the procedure number or handler address field occupying 16 bits (`n+4`). The entries are categorized as follows:

- **Trace Fault Entry**
- **Operation Fault Entry**
- **Arithmetic Fault Entry**
- **Floating-Point Fault Entry**
- **Constraint Fault Entry**
- **Protection Fault Entry**
- **Machine Fault Entry**
- **Type Fault Entry**

The local procedure fault-table entry and system procedure table fault-table entry are shown with their respective fields:

**Local Procedure Fault-Table Entry**

- **Fault-Handler Procedure Address**
  - 31 bits: `n`
  - 16 bits: `n+4`

**System Procedure Table Fault-Table Entry**

- **Fault-Handler Procedure Number**
  - 31 bits: `n`
  - 16 bits: `n+4`
  - `0000027F16`

**Reserved (Initialize to 0)**
Figure A-11: Trace Controls (Chapter 10)
APPENDIX B
MACHINE-LEVEL INSTRUCTION FORMATS

This appendix describes the machine-level format for 80960KB instructions. Included is a description of the four instruction formats and how the addressing modes relate to these formats. Also, a table is given that shows the relationship between the machine-level instruction operands and the assembly-language-level instruction operands.

GENERAL INSTRUCTION FORMAT

At the machine-level, all the 80960KB instructions are one word long and begin on word boundaries. (One group of instructions allows a second word, which contains a 32-bit displacement.)

There are four basic instruction formats: REG, COBR, CTRL, and MEM. Figure B-1 shows these formats. Each instruction has only one format, which is defined by the opcode field of the instruction.

The following sections describe the fields in the instruction word for each format.
REG FORMAT

The REG format is for operations that are performed on data contained in the global, local, and floating-point registers. The majority of the 80960KB instructions use this format.

The opcode for the REG instructions is 12 bits long (3 hexadecimal digits) and is split between bits 7 through 10 and bits 24 through 31. For example, the opcode for the `addi` instruction is $59_{16}$. Here, $59_{16}$ is contained in bits 24 through 31 and $1_{16}$ is contained in bits 7 through 10.

The `src1` and `src2` fields specify source operands for the instruction. The operands can be either registers or literals. The mode bits (m1 for `src1` and m2 for `src2`) and the instruction type (non-floating point or floating point) determine whether an operand is a register or a literal. Table B-1 shows the relationship between the instruction type, the mode bits, and the `src1` and `src2` operands.

### Table B-1: Encoding of Src1 and Src2 Fields in REG Format

<table>
<thead>
<tr>
<th>Inst. Type</th>
<th>M1 or M2</th>
<th>Src1 or Src2 Operand Value</th>
<th>Register Number</th>
<th>Literal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-FP</td>
<td>0</td>
<td>00000</td>
<td>r0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01111</td>
<td>r15</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10000</td>
<td>g0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11111</td>
<td>g15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>00000</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11111</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>FP</td>
<td>0</td>
<td>00000</td>
<td>r0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01111</td>
<td>r15</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10000</td>
<td>g0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11111</td>
<td>g15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>00000</td>
<td>fp0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00011</td>
<td>fp3</td>
<td>+0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00100 to 01111</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10000</td>
<td>reserved</td>
<td>+1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10001 to 10101</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10110</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10111 to 1111</td>
<td>reserved</td>
<td></td>
</tr>
</tbody>
</table>
For non-floating-point instructions, if a mode bit is set to 0, the respective src1 or src2 field specifies a global or local register. If the mode bit is set to 1, the field specifies an ordinal literal in the range of 0 to 31.

For floating-point instructions, if the mode bit is set to 0, the respective src1 or src2 field specifies a global or local register (just as it does for non-floating-point instructions). If the mode bit is set to 1, the field specifies either a floating-point register or one of two real-number literals (+0.0 or +1.0). All of the other encoding when the mode bit is set to 1 are reserved. When a reserved encoding is used as a source, the processor either signals an invalid opcode fault or produces an undefined value.

The src/dst field can specify either a source operand or a destination operand or both, depending on the instruction. Here again, the mode bit (m3) and the instruction type (non-floating point or floating point) determine how this field is used. Table B-2 shows this relationship.

<table>
<thead>
<tr>
<th>Inst. Type</th>
<th>m3</th>
<th>Src/Dst</th>
<th>Src Only</th>
<th>Dst Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-FP</td>
<td>0</td>
<td>g0 .. g15</td>
<td>g0 .. g15</td>
<td>g0 .. g15</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>NA</td>
<td>Literal</td>
<td>NA</td>
</tr>
<tr>
<td>FP</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>g0 .. g15</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>fp0 .. fp4</td>
</tr>
</tbody>
</table>

Note: NA means not allowed

For non-floating-point instructions, if M3 is clear, the src/dst operand is a global or local register that is encoded as shown in Table B-1. If M3 is set, the src/dst operand can be used only as a src operand that is an ordinal literal.

For floating-point instructions, the src/dst field is only used to encode destination operands. Here, the encoding is the same as shown in Table B-1, except that the encodings for floating-point literals are not allowed. That is, if M3 is clear, the destination operand is a global or local register; if M3 is set, the destination operand is a floating-point register. When a reserved encoding or literal encoding is used as a destination, the processor either signals an invalid opcode fault or produces an undefined result.

**COBR FORMAT**

The COBR format is used primarily for control-and-branch instructions. (The test-if instructions also use this format.) The opcode field for this format is 8 bits (two hexadecimal digits).

The src1 and src2 fields specify source operands for the instruction. The src1 field can specify either a global or local register or a literal as determined by mode bit m1. (The encoding of the src1 field is the same as is shown in Table B-1 for the non-floating point instructions.) The src2 field can only specify a local or global register.
The displacement field contains a signed, two's complement number that specifies a word displacement. The processor uses this value to compute the address of a target instruction that the processor goes to as the result of a comparison. The displacement field can range from $-2^{10}$ to $(2^{10} - 1)$. To determine the IP of the target instruction, the processor converts the displacement value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the IP of the next instruction.

**Note**

To allow labels or absolute addresses to be used in the assembly-language version of the COBR format instructions, the Intel 80960KB Assembler converts a `targ` (target) operand value in an assembly-language instruction into the displacement value required for the COBR format, using the following calculation:

$$\text{displacement} = \frac{\text{targ}}{4} - (\text{IP} + 4)$$

For the test-if instructions, only the src1 field is used. Here, this field specifies a destination global or local register (m1 is ignored).

**CTRL FORMAT**

The CTRL format is used for instructions that branch to a new IP, including the branch, branch-if, `bal`, and `call` instructions. The `return` instruction also uses this format. The opcode field for this format is 8 bits (two hexadecimal digits).

The instructions that use this format have no operands. The target address for a branch is specified with the displacement field in the same manner as is done with the COBR format instructions. Here, the displacement field specifies a word displacement (also a signed, two's complement number) that can range from $-2^{11}$ to $2^{21} - 1$.

The processor ignores the displacement field for the `return` instruction.

**MEM FORMAT**

The MEM format is used for instructions that require a memory address to be computed. These instructions include the load, store, and `lda` instructions. Also, the extended versions of the branch, branch-and-link, and call instructions (`bx`, `balx`, and `callx`) uses this format.

There are two MEM formats, MEMA and MEMB. The MEMB format offers the option of including a 32-bit displacement (contained in a second word) to the instruction. Bit 12 of the first word of the instruction determines whether the format is MEMA (clear) or MEMB (set).

For both formats the opcode field is 8 bits long. The src/dst field specifies a global or local register. For load instructions, the src/dst field specifies the destination register for a word loaded into the processor from memory or, for operands larger than one word, the first of successive destination registers. For store instructions, this field specifies the register or group of registers that contain the source operand to be stored in memory.
The mode bit (or bits for the MEMB format) determine the address mode used for the instruction. Table B-3 summarizes the addressing modes for the two versions of the MEM format. The fields used in these addressing modes are described in the following sections.

Table B-3: Addressing Modes for MEM Format Instructions

<table>
<thead>
<tr>
<th>Format</th>
<th>Mode Bit(s)</th>
<th>Address Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMA</td>
<td>0</td>
<td>offset</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>(abase) + offset</td>
</tr>
<tr>
<td>MEMB</td>
<td>0100</td>
<td>(abase)</td>
</tr>
<tr>
<td></td>
<td>0101</td>
<td>(IP) + displacement + 8</td>
</tr>
<tr>
<td></td>
<td>0110</td>
<td>reserved</td>
</tr>
<tr>
<td></td>
<td>0111</td>
<td>(abase) + (index) * 2&lt;sup&gt;scale&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>1100</td>
<td>displacement</td>
</tr>
<tr>
<td></td>
<td>1101</td>
<td>(abase) + displacement</td>
</tr>
<tr>
<td></td>
<td>1110</td>
<td>(index) * 2&lt;sup&gt;scale&lt;/sup&gt; + displacement</td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td>(abase) + (index) * 2&lt;sup&gt;scale&lt;/sup&gt; + displacement</td>
</tr>
</tbody>
</table>

Notes:
1. In the address computations above, a field in parentheses (e.g., (abase)) indicates that the value in the specified register is used in the computation.
2. The use of a reserved encoding causes an invalid opcode fault to be signaled.

MEMA Format Addressing

The MEMA format provides two addressing modes:

- absolute offset
- register indirect with offset

The offset field specifies an unsigned byte offset from 0 to 4096. The abase field specifies a global or local register that contains an address in memory. The address is interpreted as either a virtual address or a physical address depending on whether the processor is operating in virtual-addressing or physical-addressing mode, respectively.

For the absolute offset addressing mode (the mode bit is clear), the processor interprets the offset field as an offset from byte 0 of the current process address space. The abase field is ignored. Using this addressing mode along with the **lda** instruction allows a constant of from 0 to 4096 to be loaded into a register.

For the register indirect with offset addressing mode (the md bit is set), the value in the offset field is added to the address in the abase register. Setting the offset value to zero creates a register indirect addressing mode, however, this operation can generally be carried out faster by using the MEMB version of this addressing mode.
MEMB Format Addressing

The MEMB format provides the following seven addressing modes:

- absolute displacement
- register indirect
- register indirect with displacement
- register indirect with index
- register indirect with index and displacement
- index with displacement
- IP with displacement

The abase and index fields specify local or global registers, the contents of which are used in the address computation. When the index field is used in an addressing mode, the processor automatically scales the value in the index register by the amount specified in the scale field. Table B-4 gives the encoding of the scale field. The optional displacement field is contained in the word following the instruction word. The displacement is a 32-bit, signed, twos complement value.

<table>
<thead>
<tr>
<th>Scale</th>
<th>Scale Factor (Multiplier)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>2</td>
</tr>
<tr>
<td>010</td>
<td>4</td>
</tr>
<tr>
<td>011</td>
<td>8</td>
</tr>
<tr>
<td>100</td>
<td>16</td>
</tr>
<tr>
<td>101 to 111</td>
<td>reserved</td>
</tr>
</tbody>
</table>

Table B-4: Encoding of Scale Field

Note:
The use of a reserved encoding causes an invalid opcode fault to be signaled.

For the IP with displacement mode, the value of the displacement field plus 8 is added to the address of the current instruction.
APPENDIX C
INSTRUCTION TIMING

This appendix describes the 80960KB processor’s instruction pipeline and how it affects the timing of instructions. The number of clock cycles required for each instruction are also given here.

INTRODUCTION

The 80960 architecture defines several mechanisms for increasing processor performance through the use of pipelining and parallel execution of instructions. This appendix describes how these mechanisms have been incorporated into the design of the 80960KB processor and provides information to help programmers maximize the performance of the processor.

INTERNAL STRUCTURE OF THE 80960KB PROCESSOR

The 80960KB processor is composed of the following six major functional units (shown in Figure C-1):

- Bus Control Logic
- Instruction Fetch Unit and Instruction Cache
- Instruction Decoder
- Micro-Instruction Sequencer and ROM
- Instruction Execution Unit
- Floating Point Unit
These units function independently from one another, but in close cooperation. The functions of each of these units is described in the following sections.

**Bus Control Logic**

The Bus Control Logic (BCL) provides the interface between the processor and the external world. This interface consists of a multiplexed, burst bus, which is capable of memory-access rates of over 53 Megabytes/second (with a 20 Mhz CPU clock). The BCL accepts requests from other units within the 80960KB, prioritizes them, and executes them. It attempts to maximize bus access efficiency through buffering and burst accesses.

The BCL provides a queuing mechanism that can buffer up to three outstanding requests at any given time. This mechanism, coupled with other 80960KB features (such as scoreboard, which is discussed later), allow other units in the 80960KB to continue operation without waiting for bus requests to be completed. As a result, the execution of most memory reference instructions require little or no delay in the instruction execution pipeline.
The BCL generates burst cycles on the external bus, which allow from one to 16 bytes of data to be read or written in a single operation. The processor takes advantage of burst transfers in several ways. First, multiple-register load or store operations can be carried out in a single bus operation, using the `ldl` (load long), `ldt` (load triple), and `ldq` (load quad) instructions and the corresponding `stl` (store long), `sst` (store triple), and `stq` (store quad) instructions. Second, instructions can be fetched in 16-byte bursts, thereby reducing bus traffic for instruction fetches. Third, floating-point values of 32, 64 or 80 bits can be stored in a single bus operation.

**Instruction Fetch Unit and Instruction Cache**

The Instruction Fetch Unit (IFU) acts as an intelligent "buffer" for the Instruction Decoder (ID). Its purpose is to present the instruction stream to the ID in the fastest and most transparent way possible. The IFU uses several mechanisms to accomplish this goal, as described in the following paragraphs.

The IFU maintains a 512 byte, direct-mapped instruction cache. This cache allows very fast access to instructions. While the other units in the processor are executing instructions, the IFU looks ahead in flow of instructions stored in the instruction cache. If a cache miss is detected (that is, an instruction that will soon be needed is not in the instruction cache), the IFU issues a prefetch request to the BCL. Upon receiving the requested instruction, the IFU updates the instruction cache. In most cases, this fetch and load will take place before the ID requires the instruction. The major exception to this rule happens on branch conditions.

The IFU works closely with the ID in handling branch conditions. The ID informs the IFU of any branch operations that are about to take place. Such notifications take place on unconditional branches and on conditional branches in which the condition code is valid. When the IFU is notified of a branch, it checks for a cache hit on the desired instruction. If the instruction is not present, the IFU begins fetching instructions for the new control path.

To further minimize delays in the instruction pipeline, the ID sends a special signal to the IFU whenever instructions are required immediately. The IFU then passes the fetched instructions to the ID directly, rather than writing them to the cache and reading them back out again. This technique is called an instruction-cache bypassing.

The instruction pointer (IP) register in the processor and the IFU maintain several instruction pointers. These pointers point to instructions at various stages of the fetch-decode-execute pipeline. If a fault is signaled from any unit, the processor uses these pointers to determine the problem and preserve the state of the processor.

**Instruction Decoder**

The ID decodes the instructions it receives from the IFU and routes them to the appropriate execution units. In doing this, it attempts to keep the computing resources of the processor working at the highest possible levels.

Instructions are decoded into the following four groups, according to how the instructions are executed:
Simple Instructions

The instructions in the simple-instruction group require very little decoding. These instructions include logical; comparison; shift; integer add and subtract; and ordinal add and subtract instructions. The ID decodes these instructions and passes them to the instruction execution unit (IEU), where they are executed, usually in a single clock period.

Floating Point and Branch Instructions

All floating-point instructions are executed by the floating-point unit (FPU). Often, the execution of floating-point instructions requires interaction between the FPU, ID, and Micro-Instruction Sequencer (MIS). For example, the FPU may require access to the general-purpose registers (maintained by the IEU). Here, the ID assists in supplying data to the FPU. Also, many of the floating-point instructions are executed by means of microcode. The FPU gets the microcode from the MIS.

The ID executes branch instructions directly. If the branches are unconditional, no interaction with the processor's other execution units is required.

On conditional branch instructions, the ID uses a condition code scoreboard to streamline the branching process. Scoreboarding is a mechanism by which various resources within the processor can be marked as in use (or pending a result). When one of the execution units in the processor is in the process of altering the condition code, it marks the condition code scoreboard. When the ID prepares to execute a conditional branch instruction, it checks the condition code scoreboard. If the scoreboard is marked as in use, the ID waits for the result before proceeding. If the condition code scoreboard is clear, the ID signals the IFU immediately if a change in program flow is about to happen.

Conditional fault instructions (fault-if instructions) are also executed in the ID. These operations differ from conditional branches in that they result in a fault event being generated, followed by an implicit call to the appropriate fault-handler routine.

As a result of the pipelining described above, branches can often be carried out in zero clock cycles. For example, the branch instruction (b) shown below will execute in zero cycles, since the branch time is overlapped completely by the execution time of the floating-point instruction (sinr).
The branch-if instruction (be) in the following example is also executed in zero cycles:

```
cmp 0x10, r9
divi r10, r11, r10
be go_here
```

Here, the comparison instruction (cmp) is placed early in the instruction stream, allowing the branch condition based on the value of r9 to take place while the integer divide instruction (divi) is being executed.

**Complex Instructions**

Complex instructions are those that are executed using one or more microcode instructions. Examples of such instructions are the `flushreg` (flush local registers), `mark`, and `fmark` (force mark) instructions. The ID decodes complex instructions and forwards them to the MIS unit. The MIS then sends the equivalent microcode to the IEU.

**Load and Store Instructions**

Load and store instructions are those that request data to be read from or written into memory. The ID sends these instructions directly to the BCL, which executes them.

The ID is responsible for converting the addressing information encoded in load, store, branch, and call instructions into an effective memory addresses. The circuitry that actually performs effective-address calculations resides in the IFU, but the ID oversees these operations. The generation of effective addresses is performed within a separate carry look-ahead adder, used with hardware shift logic. The ability to calculate effective addresses independently from instruction execution allows address calculation to be overlapped with computation. The time required to calculate an effective address ranges from zero to four cycles; but, for the most commonly used addressing modes, this time is less than two cycles.

Instructions that require effective addresses are executed by either the ID or the BCL, thus preserving the pipeline and eliminating delays or resource constraints on the IEU or FPU.
Micro-Instruction Sequencer and ROM

The MIS is a multipurpose unit designed to help in the execution of instructions that use microcode. All of the processor’s microcode is stored in ROM, which is accessed through the MIS. When the ID receives a complex instruction (one that requires microcode to be executed), the MIS supplies the microcode to the IEU as described earlier in the discussion of complex instructions.

The MIS also supplies microcode for floating-point instructions; the power-up and self-test performed during processor initialization; interrupt handling; and fault handling.

Instruction Execution Unit

The IEU contains the Arithmetic Logic Unit (ALU) and the mechanism for register and condition-code scoreboard. It also manages the 16 global registers and the 4 sets of 16 local registers.

The ALU performs the following functions for the IEU:

- Addition and subtraction of integers and ordinals
- Moves between registers
- Logical operations
- Bit operations
- Shifts and rotates
- Comparisons

It is capable of performing any of these operations in a single clock cycle.

The IEU can also work with integer literals in the range of -16 to +31, which are encoded in the REG instruction format. This method of encoding literals performs two functions. First, it provides a more compact instruction stream. Second, when a literal is used as an argument for an instruction, the IEU is able to execute the instruction in one less clock cycle.

The IEU handles the reading and writing of global and local registers. It also handles the allocation of local registers sets on procedure calls. The IEU allocates a new set of local registers on each procedure call. If all four register sets become allocated, the IEU automatically flushes the oldest frame to the stack on the next procedure call. The IEU also automatically retrieves any local register frame from the stack when required by a return operation. The majority of procedure calls or returns do not require the processor to flush local registers to memory. Call instructions that can be executed without flushing a register set require only 9 cycles to complete, with the corresponding return taking only 7 cycles.

The register scoreboard provides scoreboard for the global and local registers. When, one or more registers are being used in an operation, they are marked as in use. The register scoreboard mechanism allows the processor to continue executing subsequent instructions, as long as those instructions do not require the contents of the scoreboarded registers.
A typical event that would cause scoreboardding is a load operation. For a load from memory, the contents of the affected registers are not valid until the BCL fetches the data and the registers are loaded. For example, consider the sequence:

```
Id   g0, (g1)
addi g2, g3, g4
addi g5, g4, g6
subi g0, g6, g6
```

Here, when the BCL initiates the `Id` operation, register `g0` is scoreboarded. As long as subsequent instructions do not require the contents of `g0`, the ID continues to dispatch instructions. For example, the two `addi` instructions above are executed while the BCL is fetching the data for `g0`. If `g0` is not loaded by the time the `subi` instruction is ready to be executed, the IEU delays execution of the instruction until the loading of `g0` has been completed.

If an operation accesses a single register, only that register is scoreboarded. However, if multiple registers are accessed (such as, with the `ldl`, `lit`, or `ldq` instructions), registers are scoreboarded as shown in Table C-1, according to the base register of the the group being accessed.

<table>
<thead>
<tr>
<th>Base Register Accessed</th>
<th>Block of Registers Scoreboarded</th>
</tr>
</thead>
<tbody>
<tr>
<td>g0</td>
<td>0-3</td>
</tr>
<tr>
<td>g2</td>
<td>0-3</td>
</tr>
<tr>
<td>g4</td>
<td>0-7</td>
</tr>
<tr>
<td>g6</td>
<td>0-7</td>
</tr>
<tr>
<td>g8</td>
<td>8-11</td>
</tr>
<tr>
<td>g10</td>
<td>8-11</td>
</tr>
<tr>
<td>g12</td>
<td>12-15</td>
</tr>
<tr>
<td>g14</td>
<td>12-14</td>
</tr>
</tbody>
</table>

Instruction Execution Unit Performance Enhancements

The execution times of instructions in the IEU are dependent on the instruction flow. Two features in the IEU that can enhance the performance of instruction execution are:

- Register Bypassing
- Condition Code Scoreboarding

Register Bypassing. Register bypassing is a mechanism that allows an instruction that would ordinarily require source operands to be placed in registers to be executed without accessing one or both of the source registers. Register bypassing occurs in either of two circumstances. First, when the IEU executes an instruction with two source operands, register bypassing occurs if one or both of the operands are literals. Second, register bypassing will also occur
when the second of two source operands is the result of the previous instruction. The net result of register bypassing is the saving of one clock cycle. Most instructions that the IEU executes can be executed in a single cycle when register bypassing occurs.

**Condition Code Scoreboarding.** The processor requires one clock cycle to set the condition code bits as the result of an instruction. If one of the instructions that follows depends on the condition code, condition-code scoreboarding can be used to save one cycle of execution time. The following example illustrates this technique:

**Case 1 — 5 cycles**

```
addc r4, r5, r10
mov g10, g12
addc r6, r7, r11
```

**Case 2 — 6 cycles**

```
addc r4, r5, r10
addc r6, r7, r11
mov g10, g12
```

Here, both Case 1 and Case 2 accomplish the same task. However, Case 2 requires a wait of one clock cycle between the first and second `addc` instruction, while the condition code is set. Case 1, on the other hand, takes advantage of condition code scoreboarding by executing the move (`mov`) instruction while the condition code is being set. The code in Case 1 thus executes one clock cycle faster than the code in Case 2.

**Floating Point Unit**

The FPU performs all the floating-point computations for the processor, as well as the integer multiply and divide operations. It also manages the four 80-bit floating-point registers, which it uses for extended-precision, floating-point calculations.

The FPU shares the resources of the processor. For example, it can use the global and local registers as operands for floating-point operations. It also gets microcode for the execution of complex floating-point instructions from the MIS.

To perform integer multiplication and several floating-point calculations, the FPU contains a 32-bit integer Booth-Multiplier. This multiplier performs integer multiplication operation in a variable amount of time, depending on the number of significant bits. It is used for integer multiplications and several floating-point calculations.

**EXECUTION TIMES**

The following section describes the execution times that can be expected for the various instructions in the 80960KB processor. As illustrated in the previous sections of this appendix, the execution time for each instruction can vary considerably, for two reasons. First, many instructions can vary in execution time, depending on their arguments and the state of the
on-chip resources being used. Second, by taking advantage of pipelining and overlapping of operations, a program can be written in which some instructions, in effect, take no clock cycles to execute.

In the following discussion of instruction timing, the execution time of an instruction is defined as the time between the beginning of actual execution of a decoded instruction and the beginning of execution for the next decoded instruction. For example, the illustration in Figure C-2 shows the execution time of a two operand instruction to be two clocks, with respect to the next instruction to be executed.

![Figure C-2: Execution Time of an Instruction](image)

**Execution times for the 80960 Architecture Instructions**

The following paragraphs show the instruction times for the instructions defined in the 80960 architecture.

**Logical instructions**

The timing of the logical instructions depends on the IEU bypass mechanism described earlier in this appendix, in particular for any instruction of the form:

```
alu_instruction  src1, src2, dst
```

If `src1` or `src2` is a literal or if `src2` is the result of the previous operation, a bypass hit occurs. Otherwise, there is no bypass hit and the instruction requires an extra clock to load the second operand. Table C-2 shows the timing of the logical instructions depending on whether or not a bypass hit occurs.

**Note**

In all the following tables, execution time is given in number of clock cycles.
Table C-2: Logical Instruction Timing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Normal Case Execution Time (Bypass Hit)</th>
<th>Worst Case Execution Time (Bypass Miss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>nand</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>or</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>nor</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>xor</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>xnor</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>andnot</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>notand</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>not</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>notor</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ornot</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>rotate</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>shlo</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>shro</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>shli</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>shri</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>shrdi</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Bit Instructions

The execution times for the bit instructions are also dependent on whether or not a register bypass has occurred or not, as is shown in Table C-3.

Table C-3: Bit Instruction Timing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Normal Case Execution Time (Bypass Hit)</th>
<th>Worst Case Execution Time (Bypass Miss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>notbit</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>setbit</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>clrbit</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>alterbit</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>chkbit</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>extract</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>modify</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
The execution times of the scanbit and spanbit instructions (shown in Table C-4) depend on condition code scoreboard. If the condition code is not set by the previous instruction execution, the instruction will complete in one less clock cycle. Execution time is also dependent on the number of bits operated upon.

Table C-4: Scan and Span Bit Instruction Timing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Best Case Execution Time</th>
<th>Normal Case Execution Time</th>
<th>Worst Case Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>scanbit</td>
<td>8</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>spanbit</td>
<td>8</td>
<td>11</td>
<td>14</td>
</tr>
</tbody>
</table>

Register Moves

The timing of instructions that move data between registers is directly related to the number of words moved. One clock cycle is required to move one (as shown in Table C-5).

Table C-5: Move Instruction Timing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>1</td>
</tr>
<tr>
<td>movl</td>
<td>2</td>
</tr>
<tr>
<td>movt</td>
<td>3</td>
</tr>
<tr>
<td>movq</td>
<td>4</td>
</tr>
</tbody>
</table>

Integer and Ordinal Arithmetic

The execution times for the basic add, subtract, and comparison instructions (as shown in Table C-6) depend on register bypass. The normal-case results are achieved when a register bypass occurs.
Table C-6: Integer and Ordinal Arithmetic Instruction Timing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Normal Case Execution Time (Bypass Hit)</th>
<th>Worst Case Execution Time (Bypass Miss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addo</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>addi</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>subo</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>subi</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>cmпо</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>cmpi</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>cmрinco</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>cmрdecо</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>cmрinci</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>cmрdeci</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

The execution times for the add and subtract with carry and conditional compare instructions (shown in Table C-7) depend on condition code scoreboarding. If the instruction executed prior to any of these instructions sets the condition code (CC), the worst case instruction execution time occurs; if an instruction is inserted between the instruction that sets the condition code and one of the instructions listed in Table C-7, the instruction is executed in the normal case time.

Table C-7: Add/Subtract With Carry, Conditional Compare Instruction Timing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Normal Case Execution Time (CC Available)</th>
<th>Worst Case Execution Time (CC Not Available)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addc</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>subc</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>subi</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>concmpi</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Multiply and Divide Instructions

Table C-8 shows the typical instruction execution times for the multiply and divide instructions:
Since the processor contains a Booth Multiplier with early out, the execution times on the multiply and divide instructions (shown in Table C-8) depend on the number of significant bits in the src1 operand. For example, Table C-9 shows the execution times based on the number of significant bits in src1:

Table C-9: Multiply/Divide Execution Times Based on Significant Bits

<table>
<thead>
<tr>
<th>Src1 Significant Bits</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>32</td>
<td>21</td>
</tr>
</tbody>
</table>

Note that the shift instructions or the add and subtract instructions may be faster than the multiply instructions in certain instances (for example, when multiplying by 3, 5, 15, etc.).

**Branching**

Branch instructions are executed directly by the ID and do not require IEU or FPU resources. Because of this, branch instructions can in most cases be programmed so that their execution is overlapped with other operations. Table C-10 lists the ranges of times for execution of branch instructions, from best (maximum overlap) to worst (no overlap). (The instructions in capital letters indicate groups of instructions that branch on condition codes, such the BRANCH IF instructions, be, bg, bl, etc.)
Table C-10: Branch Instruction Timing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Best Case Execution Time (CC Available)</th>
<th>Worst Case Execution Time (CC Not Available)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0 to 2 (0 to 2)</td>
<td>0 to 2 (0 to 2)</td>
</tr>
<tr>
<td>BRANCH IF</td>
<td>0 to 2 (0 to 1)</td>
<td>0 to 3 (0 to 2)</td>
</tr>
<tr>
<td>bx</td>
<td>0 to 6 (0 to 6)</td>
<td>0 to 6 (0 to 6)</td>
</tr>
<tr>
<td>BRANCH AND LINK</td>
<td>2 to 8 (2 to 8)</td>
<td>2 to 8 (2 to 8)</td>
</tr>
<tr>
<td>COMPARE AND BRANCH</td>
<td>3 to 5 (3 to 4)</td>
<td>3 to 5 (3 to 4)</td>
</tr>
<tr>
<td>TEST IF</td>
<td>0 to 3 (0 to 2)</td>
<td>0 to 4 (0 to 3)</td>
</tr>
<tr>
<td>FAULT IF</td>
<td>0 to 2 (0 to 1)</td>
<td>0 to 3 (0 to 2)</td>
</tr>
</tbody>
</table>

The second column of numbers lists execution-time ranges for conditional branches in which the condition code was not set in the previous instruction, and the third column lists ranges for branches in which the condition code was set by the previous instruction. Also, the first range in each column is for the case in which the branch is taken, and the range in parentheses is for the case in which the branch is not taken.

When writing optimized code for the 80960KB processor, it is best to perform conditional tests at least one instruction before a conditional branch. This practice allows the execution times in column two to be achieved. It is also important to note that the "not taken" branch case executes in one less cycle, because there is no break in the pipeline. (Remember, instruction time is defined as the time from the start of execution of one instruction to the start of execution of the next instruction. If the pipeline is stalled, the fetch of the next instruction will be delayed one clock. This delay may or may not be hidden by the parallelism of the 80960KB processor).

**Call/Return Instructions**

As described earlier in this appendix, the 80960KB processor provides four sets of local registers. When a call instruction is executed, the processor allocates a new set of local registers to the called procedure or interrupt routine. If, when a call or callx instruction is executed, a set of local registers is available, the processor executes the instruction in 9 clock cycles.

If a set of local registers is not available, the processor flushes the oldest set of registers to the stack in memory to free up a register set. Flushing a set of local registers requires four quad-word stores to memory. Assuming zero-wait-state memory, this operation adds 24 clocks to the 9 clocks normally required to execute a call.

The ret (return) instruction normally requires 7 clock cycles, If the local registers being returned to have been flushed to the stack, an additional 24 clocks must be added to this execution time (with zero-wait-state memory) for the processor to reload the local registers.
from the stack. It is important to note that the processor only reloads the local registers when they are required, thus eliminating unnecessary memory cycles.

**Load Instructions**

A load instruction requires the following steps:

1. Instruction Fetch
2. Decode
3. Compute Effective Address/Scoreboard Register(s)
4. Place Address on Bus
5. Wait State(s)
6. Receive Data on Bus
7. Place Data in target register

Of these steps, only steps 3 through 7 are included in the definition of execution time for an instruction. The following figures show several examples of load instruction timing depending on where the load instruction is placed in the instruction stream.

The example in Figure C-3 illustrates a load instruction where the instruction that follows requires the fetched data. Here, the pipeline is stalled while the processor waits for the load to complete. Assuming a one-clock-cycle effective-address calculation, the load will require 4 or 5 clock cycles to be executed, depending on whether or not zero-wait-state memory is used.

![Figure C-3: Load Where the Next Instruction Requires the Fetched Data](image-url)
Figure C-4 gives an example of a load instruction where the instruction that follows does not require the data being fetched from memory. Here, the unrelated instruction can be executed while the load is being completed. The 2 clock cycles required to execute the unrelated instruction are then overlapped with the 4 or 5 cycles required to execute the load (again depending on whether or not zero-wait-state memory is used). The load instruction thus requires a net of 1 or 2 clock cycles from the pipeline to be executed.

![Diagram of load instruction with unrelated instruction](image)

Figure C-4: Load Where the Next Instruction Does Not Require the Fetched Data

Finally, Figure C-5 shows an example of two load instructions being executed back-to-back. These two instructions can be executed in 5 or 6 clock cycles, as long as the number of BCL requests is limited to 3 or less (which is the size of the output request FIFO in the BCL’s control queue). Here, the second load is almost completely overlapped by the first load. Times for multiple word loads will be lengthened 1 cycle plus wait states for each additional word. If more than 3 requests become outstanding, the processor will wait until the number of outstanding load operations goes below the size of the output FIFO.

![Diagram of back-to-back load instructions](image)

Figure C-5: Back-to-Back Load Instructions
Store Operations

Store instructions involve a posting of an address and data request to the BCL and are usually executed in 2 to 3 clock cycles. (They do not require register scoreboarding.) If the instruction following a store instruction is another store instruction, the second store instruction is usually executed in 2 clock cycles. If the following instruction uses the IEU, the execution time is 3 clock cycles. The only case in which this time will increase is when the three-request output FIFO in the BCL becomes full. Here, if another store instruction is issued, the processor waits for the BCL to complete its operations before other instructions can execute.

Execution times for the Extended Instructions

The following paragraphs show the execution times for those 80960KB instructions that are extensions to the 80960 architecture.

Decimal Instructions

Table C-11 shows the instruction times for the decimal instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmove</td>
<td>7</td>
</tr>
<tr>
<td>daddc</td>
<td>8</td>
</tr>
<tr>
<td>dsubc</td>
<td>8</td>
</tr>
</tbody>
</table>

Floating-Point Instructions

Table C-12 shows the instruction execution times for the simple floating-point instructions. Where applicable, a range and a typical observed average are given.
Table C-12: Simple Floating-Point Instruction Timing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>movr</td>
<td>5</td>
</tr>
<tr>
<td>movrl</td>
<td>5 to 7</td>
</tr>
<tr>
<td>movre</td>
<td>7 to 8</td>
</tr>
<tr>
<td>cpysre</td>
<td>8</td>
</tr>
<tr>
<td>cpysrsre</td>
<td>8</td>
</tr>
<tr>
<td>addr</td>
<td>9 to 17 (typical 10)</td>
</tr>
<tr>
<td>addrl</td>
<td>12 to 20 (typical 13)</td>
</tr>
<tr>
<td>subr</td>
<td>9 to 17 (typical 10)</td>
</tr>
<tr>
<td>subrl</td>
<td>12 to 20 (typical 13)</td>
</tr>
<tr>
<td>mulr</td>
<td>11 to 22 (typical 20)</td>
</tr>
<tr>
<td>mulrl</td>
<td>14 to 43 (typical 36)</td>
</tr>
<tr>
<td>divr</td>
<td>35</td>
</tr>
<tr>
<td>divrl</td>
<td>77</td>
</tr>
<tr>
<td>cmprr</td>
<td>10</td>
</tr>
<tr>
<td>cmprl</td>
<td>12</td>
</tr>
<tr>
<td>cmpor</td>
<td>10</td>
</tr>
<tr>
<td>cmporl</td>
<td>12</td>
</tr>
<tr>
<td>cvttri</td>
<td>25 to 33</td>
</tr>
<tr>
<td>cvtril</td>
<td>26 to 35</td>
</tr>
<tr>
<td>cvtilr</td>
<td>41 to 45</td>
</tr>
<tr>
<td>cvtilr</td>
<td>42 to 46</td>
</tr>
<tr>
<td>cvtzri</td>
<td>41 to 45</td>
</tr>
<tr>
<td>cvtzrill</td>
<td>42 to 46</td>
</tr>
<tr>
<td>roundr</td>
<td>56 to 69</td>
</tr>
<tr>
<td>roundrl</td>
<td>56 to 70</td>
</tr>
<tr>
<td>scaler</td>
<td>28</td>
</tr>
<tr>
<td>scalerl</td>
<td>30</td>
</tr>
<tr>
<td>logbnr</td>
<td>32 to 41</td>
</tr>
<tr>
<td>logbnrl</td>
<td>32 to 43</td>
</tr>
</tbody>
</table>

The instructions given in Table C-13 consist of the complex floating point instructions. Only typical instruction execution rates are given here. In many cases, the clock count can vary by 30-40%. Execution time is dependent on the operands.
Table C-13: Complex Floating-Point Instruction Timing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrtrl</td>
<td>104</td>
</tr>
<tr>
<td>expr</td>
<td>300</td>
</tr>
<tr>
<td>exprl</td>
<td>334</td>
</tr>
<tr>
<td>logepr</td>
<td>400</td>
</tr>
<tr>
<td>logeprl</td>
<td>420</td>
</tr>
<tr>
<td>logr</td>
<td>438</td>
</tr>
<tr>
<td>logrl</td>
<td>438</td>
</tr>
<tr>
<td>remr</td>
<td>(67 to 75878)</td>
</tr>
<tr>
<td>remrl</td>
<td>(67 to 75878)</td>
</tr>
<tr>
<td>atanr</td>
<td>267</td>
</tr>
<tr>
<td>atanrl</td>
<td>350</td>
</tr>
<tr>
<td>cosr</td>
<td>406</td>
</tr>
<tr>
<td>cosrl</td>
<td>441</td>
</tr>
<tr>
<td>tanr</td>
<td>293</td>
</tr>
<tr>
<td>tanrl</td>
<td>323</td>
</tr>
</tbody>
</table>

It is important to note that these floating-point instructions are interruptible. When an interrupt is received while one of these instructions is being executed, the processor can suspend execution, service the external request, then resume execution of the instruction.
APPENDIX D
INITIALIZATION CODE

This appendix provides an example of the initialization code required to initialize the 80960KB processor.

OVERVIEW

The code given in this appendix demonstrates one of the methods that can be used to initialize the 80960KB processor. To use this code, the programmer must assemble (and compile, in the case of the C program modules) the individual files into object modules. These modules must then be loaded into ROM (generally EPROM). The resulting EPROM will contain an IMI (as shown in Figure 7-3; an interrupt table; a fault table; and a system procedure table; a set of dummy interrupt and fault handler routines; and a set of dummy system procedures. (The dummy interrupt and fault handler routines merely perform a return to the initialization code if an interrupt or fault occurs during initialization. Likewise, the dummy system procedures perform returns. These routines may be changed to suit the needs of a particular application.)

When the RESET pin on the processor is asserted, the processor performs its self test, then begins executing the initialization code. This code directs the processor to perform the following rudimentary steps of initialization:

1. Copy the PRCB from the IMI into RAM.
2. Copy the interrupt table into RAM.
3. Execute a reinitialize processor IAC, to enable the processor to load the new pointers to the PRCB and interrupt table.

The PRCB and interrupt table are copied into RAM because both of these data structures have fields that the processor must be able to write.

Once these first steps of initialization have been completed, the processor is able to execute additional initialization steps to configure the processor for a particular application. The following items are examples of further initialization actions that might be included in the initialization code:

- Copy new interrupt handler routines into RAM and change the pointers in the interrupt table to point to these new routines.
- Copy the fault table into RAM; copy new fault handler routines into RAM; change the pointers in the fault table to point to the new fault handler routines; and change the pointer in the PRCB to point to the relocated fault table.
- Create a new system procedure table in RAM; copy the system procedures into RAM; change the pointer in the PRCB to point to the new system procedure table.

Alternatively, the interrupt handler routines, fault handler routines, and system procedures can all be loaded into ROM. Here, execution of an application program can begin directly following the reinitialization of the processor.

3-331
EXAMPLE CODE

The example code consists of the following six files:

- example.lst
- f_table.lst
- i_table.lst
- f_handler.c
- i_handler.c
- cold.ld

The first three files are listings from the Intel 80960KB Assembler. These listings include assembly code (such as would be included in an "s" file) and the resulting object code. The fourth and fifth files are C program modules. The sixth file is a load module.

The following steps describe how to use the code in these files:

1. Assemble the assembly code in files example.s, f_table.s, and i_table.s. (Here the "s" files are made up of the assembly code only from the "lst" files listed above.)
2. Compile the C code in files f_handler.c and i_handler.c.
3. Link the object modules (example.o, f_table.o, i_table.o, f_handler.o, and i_handler.o), using the 80960 Linker and the script in the cold.ld file. The script in cold.ld directs the linker to locate the linked code at address 0.
4. Burn the output file from the linker in an EPROM.

---

example.lst

```
1 0000
2 0000
3 0000
4 0000
5 0000
6 0000
7 0000
8 0000
9 0000
10 0000
11 0000
12 0000
13 0000
14 0000
15 0000
16 0000
17 0000
18 0000

# ------ declare the below symbols public
.globl system_address_table
.globl proc_ptr
.globl start_ip
.globl csl
.globl user_stack
.globl sup_stack
```

---

Below is example system initialization code and tables. The code builds the proc in memory, sets up the stack frame, the interrupt, fault, and system procedure tables, and then vectors to a user defined routine.

---
80960KB PROGRAMMER'S REFERENCE

19 0000  .globl intr_stack
20 0000  
21 0000  # ------- define IAC address
22 0000  .set local_IAC, 0xff000010
23 0000  
24 0000  # ------- core initialization block (located at address 0)
25 0000  # ------- (8 words)
26 0000  
27 0000  .text
28 0000  
29 0000  .word system_address_table  # SAT pointer
30 0004  .word proc_ptr  # PRCB pointer
31 0008  .word 0
32 000c  .word start_ip  # Pointer to first IP
33 0010  .word cs1  # calculated at link time
34 0014  .word 0  # cs1 = -(segtab + PRCB + startup)
35 0018  .word 0
36 001c  ffffffff
37 001c  
38 001c  
39 001c  
40 001c  
41 001c  # ------- initial PRCB
42 001c  
43 0020  
44 0020  .word 0x0  # 0 - reserved
45 0024  .word 0x0  # 4 - initialize to 0
46 0028  .word 0x0  # 8 - reserved
47 002c  .word 0x0  # 12 - reserved
48 0030  .word 0x0  # 16 - reserved
49 0034  .word intr_table  # 20 - interrupt table address
50 0038  .word intr_stack  # 24 - interrupt stack pointer
51 003c  .word 0x0  # 28 - reserved
52 0040  .word 0x0000027f  # 32 -
53 0044  .word 0x0000027f  # 36 -
54 0048  .word fault_table  # 40 - fault table
55 004c  .word 0x0  # 44 - reserved
56 0050  .space 12  # 48 - reserved
57 005c  .word 0x0  # 60 - reserved
58 0060  .space 8  # 64 - reserved
59 0068  .word 0x0  # 72 - reserved
60 006c  .word 0x0  # 76 - reserved
61 0070  .space 48  # 80 - scratch space (resumption)
62 00a0  .space 44  # 128 - scratch space (error)
63 00a0  
64 00a0  
65 00a0  # The system procedure table will only be used if software puts the
66 00a0  # processor into user mode and makes a supervisor procedure call
67 00a0  
68 00cc  .align 6
69 00cc  
70 0100  .word 0x00000000  # Reserved 0
71 0104  .word 0x00000000  # Reserved 4
72 0108  .word 0x00000000  # Reserved 8
73 010c  .word 0x0001f0  # Supervisor stack pointer 12
74 0110  .word 0x00000000  # Reserved
75 0114  .word 0x00000000  # Reserved
76 0118  .word 0x00000000  # Reserved
77 011c  .word 0x00000000  # Reserved
78 0120  .word 0x00000000  # Reserved
79 0124  .word 0x00000000  # Reserved
80 0128  .word 0x00000000  # Reserved
81 012c  .word 0x00000000  # Reserved
82 0130  .word proc_entry_0  # Procedure entry 0 (user)
83 0134  .word (proc_entry_1 + 0x2)  # Procedure entry 1 (sup.)
84 0138  
85 0138  
86 0138  
87 0138  
88 0138  
89 0140  .space 136  # reserve 136 bytes
90 0140  
91 0140  
92 0140  
93 0140  
94 0140  .space 8  # initialization words
95 0140

3-333
# initialization words

```
.word sys_proc_table

.align 4
.text

proc_entry_0:  
    ret

proc_entry_1:  
    ret

# These pointers are to dummy supervisor routines. They are for example only.

# -- Processor starts execution at this spot after reset.

start_ip:

# -- copy the interrupt table to RAM

# -- Processor will copy PRCB to ram space, located at prcb_ram

# -- fix up the prcb to point to a new interrupt table

# -- the instruction pointer labeled start_again_ip

# -- The processor will begin execution here after being reinitialized. We will now set up the stacks and continue

start_again_ip:

  lda 1024, g0  
    load length of int. table

  lda 0, g4  
    initialize offset to 0

  lda intr_table, g1  
    load source

  lda intr_ram, g2  
    load address of new table

  bal loop_here  
    branch to move routine

  lda 176, g0  
    load length of prcb

  lda 0, g4  
    initialize offset to 0

  lda prcb_ptr, g1  
    load source

  lda prcb_ram, g2  
    load destination

  bal loop_here  
    branch to move routine

  lda intr_ram, g12  
    load address

  st g12, 20(g2)  
    store into PReB

  lda local_IAC, g5

  lda reinitialize_iac, g6

  symmovq g5, g6

# -- Below is the software loop to move data

# -- The processor will begin execution here after being reinitialized. We will now set up the stacks and continue

170 0254 8cf83000 00000750

  lda user_stack, fp  
    set up user stack space

  lda -0x40(fp), fp  
    load pfp (just in case)

```

---
173 0264 8c0fe040 lda 0x40(fp), sp # set up current stack ptr
174 0264
175 0264
176 0268 5cf01e00 mov 0, g14 # g14 used by C compiler
177 0268 # for argument lists past
178 0268 13 arguments.
179 0268 # Initialize to 0
180 0268
181 026c 8c203000 3b001000 lda 0x3b001000, g0 # set up arith. controls
182 0274 6840290 modac g0, g0, g0 # to mask unwanted
183 0274 0750 exceptions
184 0274
185 0274
186 0274
187 0274
# -- call main code from here
188 0274 # -- Note: This setup assumes a main module "main()" written in
189 0274 # -- C. Also, no opens are done for stdin, stdout, or stderr.
190 0274 # -- If I/O is required, the devices would need to be opened
191 0274 # -- before the call to main.
192 0274
193 0278 86003000 00000000 callx _main
194 0278
195 0278
196 0278
197 0280 93000000 reinitialize iac:
198 0284 00000000 # reinitialize iac message
199 0284 .word 0x93000000
200 0288 00000000 # word system_address_table
201 028c 00000000 # word prcb_ram # use newly copied prcb
202 028c .word start_again_ip # start here
203 028c
204 028c # ---------------- other misc. stuff
205 0290 .data
206 0290 # -- define RAM area to copy the prcb & intr to after initial bootup
207 0290
208 0290
209 0290
210 0290 .align 6
211 0290 intr_ram:
212 0290 .space 1024
213 0290
214 0290 prcb_ram:
215 0290 .space 176
216 0290
217 0290 .align 6
218 0290
219 0290 user_stack: # reserved area for the user stack
220 0290 # this can be located anywhere in memory
221 0290 .space 0x800
222 0290
223 0290 intr_stack: # reserved area for the interrupt stack
224 0290 # this can be located anywhere in memory
225 0290 .space 0x200
226 0290
227 0290 sup_stack: # Reserve stack space for
228 0290 # supervisor stack
229 0290 .space 0x400
230 0290
231 0290 # the end
232 0290

3-335
f_table.lst

1 0000
2 0000
3 0000
4 0000
5 0000
6 0000 00000000
7 0004 00000000
8 0008 00000000
9 000c 00000000
10 0010 00000000
11 0014 00000000
12 0018 00000000
13 001c 00000000
14 0020 00000000
15 0024 00000000
16 0028 00000000
17 002c 00000000
18 0030 00000000
19 0034 00000000
20 0038 00000000
21 003c 00000000
22 0040 00000000
23 0044 00000000
24 0048 00000000
25 004c 00000000
26 0050 00000000
27 0054 00000000
28 0058 00000000
29 005c 00000000
30 0060 00000000
31 0064 00000000
32 0068 00000000
33 006c 00000000
34 0070 00000000
35 0074 00000000
36 0078 00000000
37 007c 00000000
38 0080 00000000
39 0084 00000000
40 0088 00000000
41 008c 00000000
42 0090 00000000
43 0094 00000000
44 0098 00000000
45 009c 00000000
46 00a0 00000000
47 00a4 00000000
48 00a8 00000000
49 00ac 00000000
50 00b0 00000000
51 00b4 00000000
52 00b8 00000000
53 00bc 00000000
54 00c0 00000000
55 00c4 00000000
56 00c8 00000000
57 00cc 00000000
58 00d0 00000000
59 00d4 00000000
60 00d8 00000000
61 00dc 00000000
62 00e0 00000000
63 00e4 00000000
64 00e8 00000000
65 00ec 00000000
66 00f0 00000000
67 00f4 00000000
68 00f8 00000000
69 00fc 00000000

```c
/* User Fault Table */
.globl fault_table
.align 8

fault_table:
.word _user_reserved # Type 0 Reserved Fault Handler
.word 0
.word _user_trace; # 8
.word 0 #
.word _user_operation; #
.word 0 #
.word _user_arithmetic; #
.word 0 #
.word _user_real_arithmetic; #
.word 0 #
.word _user_constraint; #
.word 0 #
.word _user_reserved # Type 6 Reserved Fault Handler
.word 0 #
.word _user_protection; #
.word 0 #
.word _user_machine; #
.word 0 #
.word _user_reserved; #
.word 0 #
.word _user_type; #
.word 0 #
.word _user_reserved # Type 11 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 12 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 13 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 14 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 15 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 16 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 17 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 18 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 19 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 20 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 21 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 22 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 23 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 24 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 25 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 26 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 27 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 28 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 29 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 30 Reserved Fault Handler
.word 0 #
.word _user_reserved # Type 31 Reserved Fault Handler
```

3-336
```
/* Initial Interrupt Table */
.globl intr_table
.align 6
intr_table:
    .word 0  # Pending Priorities 0
.fill 8,4,0  # pending Interrupts 4 + (0->7)*4
    .word _user_intrh;  # interrupt table entry 8
    .word _user_intrh;  # interrupt table entry 9
    .word _user_intrh;  # interrupt table entry 10
    .word _user_intrh;  # interrupt table entry 11
    .word _user_intrh;  # interrupt table entry 12
    .word _user_intrh;  # interrupt table entry 13
    .word _user_intrh;  # interrupt table entry 14
    .word _user_intrh;  # interrupt table entry 15
    .word _user_intrh;  # interrupt table entry 16
    .word _user_intrh;  # interrupt table entry 17
    .word _user_intrh;  # interrupt table entry 18
    .word _user_intrh;  # interrupt table entry 19
    .word _user_intrh;  # interrupt table entry 20
    .word _user_intrh;  # interrupt table entry 21
    .word _user_intrh;  # interrupt table entry 22
    .word _user_intrh;  # interrupt table entry 23
    .word _user_intrh;  # interrupt table entry 24
    .word _user_intrh;  # interrupt table entry 25
    .word _user_intrh;  # interrupt table entry 26
    .word _user_intrh;  # interrupt table entry 27
    .word _user_intrh;  # interrupt table entry 28
    .word _user_intrh;  # interrupt table entry 29
    .word _user_intrh;  # interrupt table entry 30
    .word _user_intrh;  # interrupt table entry 31
    .word _user_intrh;  # interrupt table entry 32
    .word _user_intrh;  # interrupt table entry 33
    .word _user_intrh;  # interrupt table entry 34
    .word _user_intrh;  # interrupt table entry 35
    .word _user_intrh;  # interrupt table entry 36
    .word _user_intrh;  # interrupt table entry 37
    .word _user_intrh;  # interrupt table entry 38
    .word _user_intrh;  # interrupt table entry 39
    .word _user_intrh;  # interrupt table entry 40
    .word _user_intrh;  # interrupt table entry 41
    .word _user_intrh;  # interrupt table entry 42
    .word _user_intrh;  # interrupt table entry 43
    .word _user_intrh;  # interrupt table entry 44
    .word _user_intrh;  # interrupt table entry 45
    .word _user_intrh;  # interrupt table entry 46
    .word _user_intrh;  # interrupt table entry 47
    .word _user_intrh;  # interrupt table entry 48
    .word _user_intrh;  # interrupt table entry 49
    .word _user_intrh;  # interrupt table entry 50
    .word _user_intrh;  # interrupt table entry 51
    .word _user_intrh;  # interrupt table entry 52
    .word _user_intrh;  # interrupt table entry 53
    .word _user_intrh;  # interrupt table entry 54
    .word _user_intrh;  # interrupt table entry 55
    .word _user_intrh;  # interrupt table entry 56
    .word _user_intrh;  # interrupt table entry 57
    .word _user_intrh;  # interrupt table entry 58
    .word _user_intrh;  # interrupt table entry 59
    .word _user_intrh;  # interrupt table entry 60
    .word _user_intrh;  # interrupt table entry 61
    .word _user_intrh;  # interrupt table entry 62
    .word _user_intrh;  # interrupt table entry 63
    .word _user_intrh;  # interrupt table entry 64
    .word _user_intrh;  # interrupt table entry 65
    .word _user_intrh;  # interrupt table entry 66
    .word _user_intrh;  # interrupt table entry 67
    .word _user_intrh;  # interrupt table entry 68
    .word _user_intrh;  # interrupt table entry 69
    .word _user_intrh;  # interrupt table entry 70
    .word _user_intrh;  # interrupt table entry 71
    .word _user_intrh;  # interrupt table entry 72
    .word _user_intrh;  # interrupt table entry 73
    .word _user_intrh;  # interrupt table entry 74
    .word _user_intrh;  # interrupt table entry 75
```

| Offset 75 | 0134 | 00000000 | .word _user_intrh; | # interrupt table entry 76 |
| Offset 76 | 0138 | 00000000 | .word _user_intrh; | # interrupt table entry 77 |
| Offset 77 | 013C | 00000000 | .word _user_intrh; | # interrupt table entry 78 |
| Offset 78 | 0140 | 00000000 | .word _user_intrh; | # interrupt table entry 79 |
| Offset 79 | 0144 | 00000000 | .word _user_intrh; | # interrupt table entry 80 |
| Offset 80 | 0148 | 00000000 | .word _user_intrh; | # interrupt table entry 81 |
| Offset 81 | 014C | 00000000 | .word _user_intrh; | # interrupt table entry 82 |
| Offset 82 | 0150 | 00000000 | .word _user_intrh; | # interrupt table entry 83 |
| Offset 83 | 0154 | 00000000 | .word _user_intrh; | # interrupt table entry 84 |
| Offset 84 | 0158 | 00000000 | .word _user_intrh; | # interrupt table entry 85 |
| Offset 85 | 015C | 00000000 | .word _user_intrh; | # interrupt table entry 86 |
| Offset 86 | 0160 | 00000000 | .word _user_intrh; | # interrupt table entry 87 |
| Offset 87 | 0164 | 00000000 | .word _user_intrh; | # interrupt table entry 88 |
| Offset 88 | 0168 | 00000000 | .word _user_intrh; | # interrupt table entry 89 |
| Offset 89 | 016C | 00000000 | .word _user_intrh; | # interrupt table entry 90 |
| Offset 90 | 0170 | 00000000 | .word _user_intrh; | # interrupt table entry 91 |
| Offset 91 | 0174 | 00000000 | .word _user_intrh; | # interrupt table entry 92 |
| Offset 92 | 0178 | 00000000 | .word _user_intrh; | # interrupt table entry 93 |
| Offset 93 | 017C | 00000000 | .word _user_intrh; | # interrupt table entry 94 |
| Offset 94 | 0180 | 00000000 | .word _user_intrh; | # interrupt table entry 95 |
| Offset 95 | 0184 | 00000000 | .word _user_intrh; | # interrupt table entry 96 |
| Offset 96 | 0188 | 00000000 | .word _user_intrh; | # interrupt table entry 97 |
| Offset 97 | 018C | 00000000 | .word _user_intrh; | # interrupt table entry 98 |
| Offset 98 | 0190 | 00000000 | .word _user_intrh; | # interrupt table entry 99 |
| Offset 99 | 0194 | 00000000 | .word _user_intrh; | # interrupt table entry 100 |
| Offset 100 | 0198 | 00000000 | .word _user_intrh; | # interrupt table entry 101 |
| Offset 101 | 019C | 00000000 | .word _user_intrh; | # interrupt table entry 102 |
| Offset 102 | 01A0 | 00000000 | .word _user_intrh; | # interrupt table entry 103 |
| Offset 103 | 01A4 | 00000000 | .word _user_intrh; | # interrupt table entry 104 |
| Offset 104 | 01A8 | 00000000 | .word _user_intrh; | # interrupt table entry 105 |
| Offset 105 | 01AC | 00000000 | .word _user_intrh; | # interrupt table entry 106 |
| Offset 106 | 01B0 | 00000000 | .word _user_intrh; | # interrupt table entry 107 |
| Offset 107 | 01B4 | 00000000 | .word _user_intrh; | # interrupt table entry 108 |
| Offset 108 | 01B8 | 00000000 | .word _user_intrh; | # interrupt table entry 109 |
| Offset 109 | 01BC | 00000000 | .word _user_intrh; | # interrupt table entry 110 |
| Offset 110 | 01C0 | 00000000 | .word _user_intrh; | # interrupt table entry 111 |
| Offset 111 | 01C4 | 00000000 | .word _user_intrh; | # interrupt table entry 112 |
| Offset 112 | 01C8 | 00000000 | .word _user_intrh; | # interrupt table entry 113 |
| Offset 113 | 01CC | 00000000 | .word _user_intrh; | # interrupt table entry 114 |
| Offset 114 | 01D0 | 00000000 | .word _user_intrh; | # interrupt table entry 115 |
| Offset 115 | 01D4 | 00000000 | .word _user_intrh; | # interrupt table entry 116 |
| Offset 116 | 01DA | 00000000 | .word _user_intrh; | # interrupt table entry 117 |
| Offset 117 | 01DC | 00000000 | .word _user_intrh; | # interrupt table entry 118 |
| Offset 118 | 01E0 | 00000000 | .word _user_intrh; | # interrupt table entry 119 |
| Offset 119 | 01E4 | 00000000 | .word _user_intrh; | # interrupt table entry 120 |
| Offset 120 | 01E8 | 00000000 | .word _user_intrh; | # interrupt table entry 121 |
| Offset 121 | 01EC | 00000000 | .word _user_intrh; | # interrupt table entry 122 |
| Offset 122 | 0200 | 00000000 | .word _user_intrh; | # interrupt table entry 123 |
| Offset 123 | 0204 | 00000000 | .word _user_intrh; | # interrupt table entry 124 |
| Offset 124 | 0208 | 00000000 | .word _user_intrh; | # interrupt table entry 125 |
| Offset 125 | 020C | 00000000 | .word _user_intrh; | # interrupt table entry 126 |
| Offset 126 | 0210 | 00000000 | .word _user_intrh; | # interrupt table entry 127 |
| Offset 127 | 0214 | 00000000 | .word _user_intrh; | # interrupt table entry 128 |
| Offset 128 | 0218 | 00000000 | .word _user_intrh; | # interrupt table entry 129 |
| Offset 129 | 0220 | 00000000 | .word _user_intrh; | # interrupt table entry 130 |
| Offset 130 | 0224 | 00000000 | .word _user_intrh; | # interrupt table entry 131 |
| Offset 131 | 0228 | 00000000 | .word _user_intrh; | # interrupt table entry 132 |
| Offset 132 | 022C | 00000000 | .word _user_intrh; | # interrupt table entry 133 |
| Offset 133 | 0230 | 00000000 | .word _user_intrh; | # interrupt table entry 134 |
| Offset 134 | 0234 | 00000000 | .word _user_intrh; | # interrupt table entry 135 |
| Offset 135 | 0238 | 00000000 | .word _user_intrh; | # interrupt table entry 136 |
| Offset 136 | 023C | 00000000 | .word _user_intrh; | # interrupt table entry 137 |
| Offset 137 | 0240 | 00000000 | .word _user_intrh; | # interrupt table entry 138 |
| Offset 138 | 0244 | 00000000 | .word _user_intrh; | # interrupt table entry 139 |
| Offset 139 | 0248 | 00000000 | .word _user_intrh; | # interrupt table entry 140 |
| Offset 140 | 024C | 00000000 | .word _user_intrh; | # interrupt table entry 141 |
| Offset 141 | 0250 | 00000000 | .word _user_intrh; | # interrupt table entry 142 |
interrupt table entry 192
interrupt table entry 189
interrupt table entry 188
interrupt table entry 187
interrupt table entry 186
interrupt table entry 185
interrupt table entry 184
interrupt table entry 183
interrupt table entry 182
interrupt table entry 181
interrupt table entry 179
interrupt table entry 177
interrupt table entry 175
interrupt table entry 173
interrupt table entry 170
interrupt table entry 168
interrupt table entry 166
interrupt table entry 164
interrupt table entry 162
interrupt table entry 160
interrupt table entry 158
interrupt table entry 156
interrupt table entry 154
interrupt table entry 152
interrupt table entry 150
interrupt table entry 148
interrupt table entry 146
interrupt table entry 144
interrupt table entry 142
interrupt table entry 140
interrupt table entry 138
interrupt table entry 136
interrupt table entry 134
interrupt table entry 132
interrupt table entry 130
interrupt table entry 128
interrupt table entry 126
interrupt table entry 124
interrupt table entry 122
interrupt table entry 120
interrupt table entry 118
interrupt table entry 116
interrupt table entry 114
interrupt table entry 112
interrupt table entry 110
interrupt table entry 108
interrupt table entry 106
interrupt table entry 104
interrupt table entry 102
interrupt table entry 100
interrupt table entry 98
interrupt table entry 96
interrupt table entry 94
interrupt table entry 92
interrupt table entry 90
interrupt table entry 88
interrupt table entry 86
interrupt table entry 84
interrupt table entry 82
interrupt table entry 80
interrupt table entry 78
interrupt table entry 76
interrupt table entry 74
interrupt table entry 72
interrupt table entry 70
interrupt table entry 68
interrupt table entry 66
interrupt table entry 64
interrupt table entry 62
interrupt table entry 60
interrupt table entry 58
interrupt table entry 56
interrupt table entry 54
interrupt table entry 52
interrupt table entry 50
interrupt table entry 48
interrupt table entry 46
interrupt table entry 44
interrupt table entry 42
interrupt table entry 40
interrupt table entry 38
interrupt table entry 36
interrupt table entry 34
interrupt table entry 32
interrupt table entry 30
interrupt table entry 28
interrupt table entry 26
interrupt table entry 24
interrupt table entry 22
interrupt table entry 20
interrupt table entry 18
interrupt table entry 16
interrupt table entry 14
interrupt table entry 12
interrupt table entry 10
interrupt table entry 8
interrupt table entry 6
interrupt table entry 4
interrupt table entry 2
interrupt table entry 0
f_handler.c

user_reserved() {}
user_machine() {}
user_trace() {}
user_operation() {}
user_arithmetic() {}
user_real_arithmetic() {}
user_constraint() {}
user_protection() {}
user_type() {}

i_handler.c

user_intrh()
{
}

cold.ld

MEMORY
{
    rom: o=0x0, l=0x40000
    ram: o=0x40000, l=0x40000
}

SECTIONS
{
    .text :
    {
    } >rom
    .data :
    {
    } >ram
    .bss :
    {
    } >ram
}

csl = -(system_address_table + prcb_ptr + start_ip);
This appendix describes those parts of the 80960KB processor design that are implementation dependent. This information is provided to facilitate the design of programs and kernel code that will be portable to other implementations of the 80960 architecture.

ARCHITECTURE RESTRICTIONS

The following aspects of the 80960KB's operation are deviations from the 80960KB architecture:

1. On all bus write operations except those of the synmov, synmovl, and synmovq instructions, the processor ignores the BADAC pin (i.e., errors signaled on "normal" writes are ignored).

2. The check for out-of-range input values for the expr, exprl, logepr, and logeprl instructions is omitted; out-of-range inputs yield an undefined result.

3. Bits 5 and 6 of a machine-level instruction word in the REG and MEMB formats and bits 0 and 1 of the CTRL format are provided to designate special function registers. The 80960KB processor has no special function registers.

4. The 80960KB processor does not guarantee that the value in register r2 of the current frame is predictable.

5. (The following is a note rather than a restriction.) When using the REG-format instructions, the m bit for every operand that is not defined by the instruction should be set (e.g., code the unused operand as an arbitrary literal). This practice may reduce overhead in some situations.

SALIGN PARAMETER

Stack frames in the 80960KB architecture are aligned on (SALIGN*16) byte boundaries. SALIGN is an implementation defined parameter. For the 80960KB processor, SALIGN is 4. Stack frames for this processor are thus aligned on 64 byte boundaries.

The low-order N bits of the FP are ignored and always interpreted to be zero. The N parameter is defined by the following expression: SALIGN*16 = 2^N. Thus for the 80960KB processor, N is 6.

BOUNDARY ALIGNMENT

The physical-address boundaries on which an operand begins has an impact on processor performance. For the 80960KB processor, the following is true:

- An operand that spans more word boundaries than necessary (e.g., addressing a 32-bit operand on a nonword boundary) suffers a moderate cost in speed because of extra bus and memory cycles.
• An operand that spans a 16-byte boundary suffers a large cost in speed.
• String operands that begin on nonword boundaries suffer a moderate cost in speed. String operands that begin on word boundaries but not on 16-byte boundaries suffer a small cost in speed.

FAULTS
The size of resumption records conditionally placed on the stack during faults and interrupts is 16 bytes.

PHYSICAL MEMORY
The upper 16M bytes of physical memory are reserved for special functions of local-bus components and IACs.

IACS
The mechanism for sending, receiving, and handling IAC messages is not defined in the 80960 architecture. It is a special implementation of the 80960KB processor.

The write-external-priority flag in the IMI controls is not defined in the 80960 architecture.

INTERRUPTS
The interrupt IAC message, the interrupt pins, and the interrupt register are not defined in the 80960 architecture. They are special implementations for the 80960KB processor.

INITIALIZATION
The 80960 architecture does not define an initialization mechanism. The initialization mechanism and procedures described in this manual are implementation dependent for the 80960KB processor.

BREAKPOINTS
The breakpoint registers in the 80960KB processor are not defined in the 80960 architecture.

IMPLEMENTATION DEPENDENT INSTRUCTIONS
The `synmov`, `synmovl`, `synmovq`, and `synld` instructions are not defined in the 80960 architecture and are implementation dependent in the 80960KB processor.
LOCK PIN

The LOCK pin is not defined in the 80960 architecture and is implementation dependent in the 80960KB processor.
80960KB
EMBEDDED 32-BIT MICROPROCESSOR
WITH INTEGRATED FLOATING-POINT UNIT

- High-Performance Embedded Architecture
  - 20 MIPS Burst Execution at 20 MHz
  - 7.5 MIPS* Sustained Execution at 20 MHz
- On-Chip Floating-Point Unit
  - Supports IEEE 754 Floating-Point Standard
  - Four 80-Bit Registers
  - 4 Million Whetstones/Second at 20 MHz
- 512-Byte On-Chip Instruction Cache
  - Direct Mapped
  - Parallel Load/Decode for Uncached Instructions
- Multiple Register Sets
  - Sixteen Global 32-Bit Registers
  - Sixteen Local 32-Bit Registers
  - Four Local Register Sets Stored On-Chip
  - Register Scoreboarding
- Built-In Interrupt Controller
  - 32 Priority Levels
  - 256 Vectors
  - Supports 8259A
- Easy to Use, High Bandwidth 32-Bit Bus
  - 53.3 MBytes/s Burst
  - Up to 16-Bytes Transferred per Burst
- 4 Gigabyte, Linear Address Space
- 132-Lead Pin Grid Array (PGA) Package

The 80960KB is the first member of Intel's new 32-bit microprocessor family, the 960 series, which is designed especially for embedded applications. It is based on the family's high performance, common core architecture, and includes a 512-byte instruction cache, a built-in interrupt controller, and an integrated floating-point unit. The 80960KB has a large register set, multiple parallel execution units, and a high-bandwidth, burst bus. Using advanced RISC technology, this high performance processor is capable of execution rates in excess of 7.5 million instructions per second.* The 80960KB is well-suited for a wide range of embedded applications, including image processing, industrial control, robotics, and telecommunications.

*Relative to Digital Equipment Corporation's VAX-11/780** at 1 MIPS

**VAX-11™ is a trademark of Digital Equipment Corporation.
THE 960 SERIES

The 80960KB is the first member of a new family of 32-bit microprocessors from Intel known as the 960 Series. This series was especially designed to serve the needs of embedded applications. The embedded market includes applications as diverse as industrial automation, avionics, image processing, graphics, robotics, telecommunications, and automobiles. These types of applications require high integration, low power consumption, quick interrupt response times, and high performance. Since time to market is critical, embedded microprocessors need to be easy to use in both hardware and software designs.

All members of the 80960 series share a common core architecture which utilizes RISC technology so that, except for special functions, the family members are object code compatible. Each new processor in the series will add its own special set of functions to the core to satisfy the needs of a specific application or range of applications in the embedded market. For example, future processors may include a DMA controller, a timer, or an A/D converter.

The 80960KB includes an integrated floating-point unit. Also available is the 80960MC, a military-grade version of the processor, and in the near future, the 80960KA, another commercial version without floating-point will be available.

Figure 2. Register Set

NOTES:
1. Register g15 is reserved for stack management functions.
2. Registers r0, r1, and r2 are reserved for stack management functions.
KEY PERFORMANCE FEATURES
The 80960KB's architecture is based on the most recent advances in RISC technology and is grounded in Intel's long experience in designing embedded controllers. Many features contribute to the 80960KB's exceptional performance:

1. Large Register Set. Having a large number of registers reduces the number of times that a processor needs to access memory. Modern compilers can take advantage of this feature to optimize execution speed. For maximum flexibility, the 80960KB provides 32 32-bit registers and four 80-bit floating-point registers. (See Figure 2.)

2. Fast Instruction Execution. Simple functions make up the bulk of instructions in most programs, so that execution speed can be greatly improved by ensuring that these core instructions execute in as short a time as possible. The most-frequently executed instructions such as register-register moves, add/subtract, logical operations, and shifts execute in one to two cycles (Table 1 contains a list of instructions.)

3. Load/Store Architecture. Like other processors based on RISC technology, the 80960KB has a Load/Store architecture, only the LOAD and STORE instructions reference memory; all other instructions operate on registers. This type of architecture simplifies instruction decoding and is used in combination with other techniques to increase parallelism.

<table>
<thead>
<tr>
<th>Control</th>
<th>Opcode</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare and Branch</td>
<td>Opcode</td>
<td>Reg/Lit</td>
</tr>
<tr>
<td>Register to Register</td>
<td>Opcode</td>
<td>Reg</td>
</tr>
<tr>
<td>Memory Access—Short</td>
<td>Opcode</td>
<td>Reg</td>
</tr>
<tr>
<td>Memory Access—Long</td>
<td>Opcode</td>
<td>Reg</td>
</tr>
</tbody>
</table>
Table 1. 80960KB Instruction Set

<table>
<thead>
<tr>
<th>Data Movement</th>
<th>Arithmetic</th>
<th>Logical</th>
<th>Bit and Bit Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Add</td>
<td>And</td>
<td>Set Bit</td>
</tr>
<tr>
<td>Store</td>
<td>Subtract</td>
<td>Not And</td>
<td>Clear Bit</td>
</tr>
<tr>
<td>Move</td>
<td>Multiply</td>
<td>And Not</td>
<td>Not Bit</td>
</tr>
<tr>
<td>Load Address</td>
<td>Divide</td>
<td>Or</td>
<td>Check Bit</td>
</tr>
<tr>
<td></td>
<td>Remainder</td>
<td>Exclusive Or</td>
<td>Alter Bit</td>
</tr>
<tr>
<td></td>
<td>Modulo</td>
<td>Not Or</td>
<td>Scan for Bit</td>
</tr>
<tr>
<td></td>
<td>Shift</td>
<td>Or Not</td>
<td>Scan over Bit</td>
</tr>
<tr>
<td></td>
<td>Extended Multiply</td>
<td>Nor</td>
<td>Extract</td>
</tr>
<tr>
<td></td>
<td>Extended Divide</td>
<td>Exclusive Nor</td>
<td>Modify</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Branch</th>
<th>Call/Return</th>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>Unconditional</td>
<td>Call</td>
<td>Conditional Fault</td>
</tr>
<tr>
<td>Conditional</td>
<td>Branch</td>
<td>Call Extended</td>
<td>Synchronize Faults</td>
</tr>
<tr>
<td>Compare</td>
<td>Conditional Branch</td>
<td>Call System</td>
<td></td>
</tr>
<tr>
<td>Compare and Increment</td>
<td>Compare and Branch</td>
<td>Return</td>
<td></td>
</tr>
<tr>
<td>Compare and Decrement</td>
<td></td>
<td>Branch and Link</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Debug</th>
<th>Miscellaneous</th>
<th>Decimal</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Modify Trace</td>
<td>Atomic Add</td>
<td>Move</td>
<td></td>
</tr>
<tr>
<td>Controls</td>
<td>Atomic Modify</td>
<td>Add with Carry</td>
<td></td>
</tr>
<tr>
<td>Mark</td>
<td>Flush Local Registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Force Mark</td>
<td>Modify Arithmetic Controls</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scan Byte for Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Test Condition Code</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Floating-Point</th>
<th>Synchronous</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert Real to Integer</td>
<td>Move Real</td>
<td>Synchronous Load</td>
<td></td>
</tr>
<tr>
<td>Convert Integer to Real</td>
<td>Add</td>
<td>Synchronous Move</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Remainder</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scale</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Round</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Square Root</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sine</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cosine</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tangent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arctangent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Log</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Log Binary</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Log Natural</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exponent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Classify</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Copy Real Extended</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Compare</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. Simple Instruction Formats. All instructions in the 80960KB are 32-bits long and must be aligned on word boundaries. This alignment makes it possible to eliminate the instruction-alignment stage in the pipeline. To simplify the instruction decoder further, there are only five instruction formats and each instruction uses only one format. (See Figure 3.)

5. Overlapped Instruction Execution. A load operation allows execution of subsequent instructions to continue before the data has been returned from memory, so that these instructions can overlap the load. The 80960KB manages this process transparently to software through the use of a register scoreboard. Conditional instructions also make use of a scoreboard so that subsequent unrelated instructions can be executed while the conditional instruction is pending.

6. Integer Execution Optimization. When the result of an operation is used as an operand in a subsequent calculation, the value is sent immediately to its destination register. Yet at the same time, the value is put back on a bypass path to the ALU, thereby saving the time that otherwise would be required to retrieve the value for the next operation.

7. Bandwidth Optimizations. The 80960KB gets optimal use of its memory bus bandwidth because the bus is tuned for use with the cache: the line size of the instruction cache matches the maximum burst size for instruction fetches. The 80960KB automatically fetches four words in a burst and stores them directly in the cache. Due to the size of the cache and the fact that it is continually filled in anticipation of needed instructions in the program flow, the 80960KB is exceptionally insensitive to memory wait states. In fact, each wait state causes only a 7% degradation in system performance. The benefit is that the 80960KB will deliver outstanding performance even with a low cost memory system.

8. Cache Bypass. If there is a cache miss, the processor fetches the needed instruction, then sends it on to the instruction decoder at the same time it updates the cache. Thus, no extra time is taken to load and read the cache.

Memory Space and Addressing Modes

The 80960KB offers a linear programming environment so that all programs running on the processor are contained in a single address space. The maximum size of the address space is 4 Gigabytes ($2^{32}$ bytes).

For ease of use, the 80960KB has a small number of addressing modes, but includes all those necessary to ensure efficient compiler implementations of high-level languages such as C, Fortran and Ada. Table 2 lists the memory addressing modes.

Data Types

The 80960 KB recognizes the following data types:

Numeric:
- 8-, 16-, 32- and 64-bit ordinals
- 8-, 16, 32- and 64-bit integers
- 32-, 64- and 80-bit real numbers

Non-Numeric:
- Bit
- Bit Field
- Triple-Word (96 bits)
- Quad-Word (128 bits)

Large Register Set

The programming environment of the 80960KB includes a large number of registers. In fact, 36 registers are available at any time. The availability of this many registers greatly reduces the number of memory accesses required to execute most programs, which leads to greater instruction processing speed.

There are two types of general-purpose registers: local and global. The 20 global registers consist of sixteen 32-bit registers (G0 through G15) and four 80-bit registers (FP0 through FP3). These registers perform the same function as the general-purpose registers provided in other popular microprocessors. The term global refers to the fact that these registers retain their contents across procedure calls.

The local registers, on the other hand, are procedure specific. For each procedure call, the 80960KB allocates 16 local registers (R0 through R15). Each local register is 32 bits wide. Any register can also be used for single or double-precision floating-point operations; the 80-bit floating-point registers are provided for extended precision.

Multiple Register Sets

To further increase the efficiency of the register set, multiple sets of local registers are stored on-chip. This cache holds up to four local register frames, which means that up to three procedure calls can be made without having to access the procedure stack resident in memory.

Although programs may have procedure calls nested many calls deep, a program typically oscillates back and forth between only two or three levels. As
Table 2. Memory Addressing Modes

- 12-Bit Offset
- 32-Bit Offset
- Register-Indirect
- Register + 12-Bit Offset
- Register + 32-Bit Offset
- Register + (Index-Register × Scale-Factor)
- Register × Scale Factor + 32-Bit Displacement
- Register + (Index-Register × Scale-Factor) + 32-Bit Displacement

Scale-Factor is 1, 2, 4, 8 or 16

As a result, with four stack frames in the cache, the probability of there being a free frame on the cache when a call is made is very high. In fact, runs of representative C-language programs show that 80% of the calls are handled without needing to access memory.

If there are four or more active procedures and a new procedure is called, the processor moves the oldest set of local registers in the register cache to a procedure stack in memory to make room for a new set of registers. Global register G15 is used by the processor as the frame pointer (FP) for the procedure stack.

Note that the global and floating-point registers are not exchanged on a procedure call, but retain their contents, making them available to all procedures for fast parameter passing. An illustration of the register cache is shown in Figure 4.

Figure 4. Multiple Register Sets Are Stored On-Chip
Instruction Cache

To further reduce memory accesses, the 80960KB includes a 512-byte on-chip instruction cache. The instruction cache is based on the concept of locality of reference; that is, most programs are not usually executed in a steady stream but consist of many branches and loops that lead to jumping back and forth within the same small section of code. Thus, by maintaining a block of instructions in a cache, the number of memory references required to read instructions into the processor can be greatly reduced.

To load the instruction cache, instructions are fetched in 16-byte blocks, so that up to four instructions can be fetched at one time. An efficient prefetch algorithm increases the probability that an instruction will already be in the cache when it is needed.

Code for small loops will often fit entirely within the cache, leading to a great increase in processing speed since further memory references might not be necessary until the program exits the loop. Similarly, when calling short procedures, the code for the calling procedure is likely to remain in the cache, so it will be there on the procedure's return.

Register Scoreboarding

The instruction decoder has been optimized in several ways. One of these optimizations is the ability to do instruction overlapping by means of register scoreboarding.

Register scoreboarding occurs when a LOAD instruction is executed to move a variable from memory into a register. When the instruction is initiated, a scoreboard bit on the target register is set. When the register is actually loaded, the bit is reset. In between, any reference to the register contents is accompanied by a test of the scoreboard bit to insure that the load has completed before processing continues. Since the processor does not have to wait for the LOAD to be completed, it can go on to execute additional instructions placed in between the LOAD instruction and the instruction that uses the register contents, as shown in the following example:

LOAD R4, address 1
LOAD R5, address 2
Unrelated instruction
Unrelated instruction
ADD R4, R5, R6

In essence, the two unrelated instructions between the LOAD and ADD instructions are executed for free (i.e., take no apparent time to execute) because they are executed while the register is being loaded. Up to three LOAD instructions can be pending at one time with three corresponding scoreboard bits set. By exploiting this feature, system programmers and compilers have a useful tool for optimizing execution speed.

Floating-Point Arithmetic

In the 80960KB, floating-point arithmetic has been made an integral part of the architecture. Having the floating-point unit integrated on-chip provides two advantages. First, it improves the performance of the chip for floating-point applications, since no additional bus overhead is associated with floating-point calculations, thereby leaving more time for other bus operations such as I/O. Second, the cost of using floating-point operations is reduced because a separate coprocessor chip is not required.

The 80960KB floating-point (real number) data types include single-precision (32-bit), double-precision (64-bit), and extended precision (80-bit) floating-point numbers. Any register may be used to execute floating-point operations.

The processor provides hardware support for both mandatory and recommended portions of IEEE Standard 754 for floating-point arithmetic, including all arithmetic, exponential, logarithmic, and other transcendental functions. Table 3 shows execution times for some representative instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>32-Bit</th>
<th>64-Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>0.5</td>
<td>0.7</td>
</tr>
<tr>
<td>Subtract</td>
<td>0.5</td>
<td>0.7</td>
</tr>
<tr>
<td>Multiply</td>
<td>1.0</td>
<td>1.8</td>
</tr>
<tr>
<td>Divide</td>
<td>1.8</td>
<td>3.8</td>
</tr>
<tr>
<td>Square Root</td>
<td>5.0</td>
<td>5.2</td>
</tr>
<tr>
<td>Arctangent</td>
<td>13.4</td>
<td>17.5</td>
</tr>
<tr>
<td>Exponent</td>
<td>15.0</td>
<td>16.7</td>
</tr>
<tr>
<td>Sine</td>
<td>20.3</td>
<td>22.1</td>
</tr>
<tr>
<td>Cosine</td>
<td>20.3</td>
<td>22.1</td>
</tr>
</tbody>
</table>
High Bandwidth Local Bus

An 80960KB CPU resides on a high-bandwidth address/data bus known as the local bus (L-Bus). The L-Bus provides a direct communication path between the processor and the memory and I/O subsystem interfaces. The processor uses the local bus to fetch instructions, manipulate memory, and respond to interrupts. Its features include:

- 32-bit multiplexed address/data path
- Four-word burst capability, which allows transfers from 1 to 16 bytes at a time
- High bandwidth reads and writes at 53 MBytes per second
- Special signal to indicate whether a memory transaction can be cached

Figure 5 identifies the groups of signals which constitute the L-Bus. Table 4 lists the function of the L-Bus and other processor-support signals, such as the interrupt lines.

Interrupt Handling

The 80960KB can be interrupted in one of two ways: by the activation of one of four interrupt pins or by sending a message on the processor's data bus.

The 80960KB is unusual in that it automatically handles interrupts on a priority basis and tracks pending interrupts through its on-chip interrupt controller. Two of the interrupt pins can be configured to provide 8259A handshaking for expansion beyond four interrupt lines.

Debug Features

The 80960KB has built-in debug capabilities. There are two types of breakpoints and six different trace modes. The debug features are controlled by two internal 32-bit registers, the Process-Controls Word and the Trace-Controls Word. By setting bits in these control words, a software debug monitor can closely control how the processor responds during program execution.

The 80960KB has both hardware and software breakpoints. It provides two hardware breakpoint registers on-chip which can be set by a special command to any value. When the instruction pointer matches the value in one of the breakpoint registers, the breakpoint will fire, and a breakpoint handling routine is called automatically.

The 80960KB also provides software breakpoints through the use of two instructions, MARK and FMARK. These instructions can be placed at any point in a program and will cause the processor to halt execution at that point and call the breakpoint handling routine. The breakpoint mechanism is easy to use and provides a powerful debugging tool.

Tracing is available for instructions (single-step execution), calls and returns, and branching. Each different type of trace may be enabled separately by a special debug instruction. In each case, the 80960KB executes the instruction first and then calls a trace handling routine (usually part of a software debug monitor). Further program execution is halted until the trace routine is completed. When the trace event handling routine is completed, instruction execution resumes at the next instruction. The

![Figure 5. Local Bus Signal Groups](image-url)
80960KB’s tracing mechanisms, which are implemented completely in hardware, greatly simplify the task of testing and debugging software.

**FAULT DETECTION**

The 80960KB has an automatic mechanism to handle faults. There are ten fault types including trace, arithmetic, and floating-point faults. When the processor detects a fault, it automatically calls the appropriate fault handling routine and saves the current instruction pointer and necessary state information to make efficient recovery possible. The processor posts diagnostic information on the type of fault to a Fault Record. Like interrupt handling routines, fault handling routines are usually written to meet the needs of a specific application and are often included as part of the operating system or kernel.

For each of the ten fault types, there are numerous subtypes that provide specific information about a fault. For example, a floating-point fault may have its subtype set to an Overflow or Zero-Divide fault. The fault handler can use this specific information to respond correctly to the fault.

**BUILT-IN TESTABILITY**

Upon reset, the 80960KB automatically conducts an extensive internal test of its major blocks of logic. Then, before executing its first instruction, it does a zero check sum on the first eight words in memory to ensure that the system has been loaded correctly. If a problem is discovered at any point during the self-test, the 80960KB will assert its FAILURE pin and will not begin program execution. The self-test takes approximately 47,000 cycles to complete.

System manufacturers can use the 80960KB’s self-test feature during incoming parts inspection. No special diagnostic programs need to be written, and the test is both thorough and fast. The self-test capability helps ensure that defective parts will be discovered before systems are shipped, and once in the field, the self-test makes it easier to distinguish between problems caused by processor failure and problems resulting from other causes.

**CHMOS**

The 80960KB is fabricated using Intel’s CHMOS III (Complementary High Speed Metal Oxide Semiconductor) process. This advanced technology eliminates the frequency and reliability limitations of older CMOS processes and opens a new era in microprocessor performance. It combines the high performance capabilities of Intel’s industry-leading HMOS III technology with the high density and low power characteristics of CMOS. The 80960KB is available at 16 MHz and 20 MHz. A 25 MHz version will be available in the near future.

Table 4a. 80960KB Pin Description: L-Bus Signals

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Name and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK2</td>
<td>I</td>
<td>SYSTEM CLOCK provides the fundamental timing for 80960KB systems. It is divided by two inside the 80960KB to generate the internal processor clock.</td>
</tr>
<tr>
<td>LAD₃₁ - LAD₀</td>
<td>I/O T.S.</td>
<td>LOCAL ADDRESS/DATA BUS carries 32-bit physical addresses and data to and from memory. During an address (Tₐ) cycle, bits 2–31 contain a physical word address (bits 0–1 indicate SIZE; see below). During a data (Tₚ) cycle, bits 0–31 contain read or write data. The LAD lines are active HIGH and float to a high impedance state when not active. SIZE, which is comprised of bits 0–1 of the LAD lines during a Tₐ cycle, specifies the size of a burst transfer in words.</td>
</tr>
<tr>
<td>ALE</td>
<td>O T.S.</td>
<td>ADDRESS-LATCH ENABLE indicates the transfer of a physical address. ALE is asserted during a Tₐ cycle and deasserted before the beginning of the Tₚ state. It is active LOW and floats to a high impedance state when the processor is idle or is at the end of any bus access.</td>
</tr>
</tbody>
</table>

I/O = Input/Output, O = Output, I = Input, O.D. = Open-Drain, T.S. = tri-state
Table 4a. 80960KB Pin Description: L-Bus Signals (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Name and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADS</td>
<td>O</td>
<td>ADDRESS/DATA STATUS indicates an address state. ADS is asserted every $T_a$ state and deasserted during the following $T_d$ state. For a burst transaction, ADS is asserted again every $T_d$ state where READY was asserted in the previous cycle.</td>
</tr>
<tr>
<td>W/R</td>
<td>O</td>
<td>WRITE/READ specifies, during a $T_a$ cycle, whether the operation is a write or read. It is latched on-chip and remains valid during $T_d$ cycles.</td>
</tr>
<tr>
<td>DT/R</td>
<td>O</td>
<td>DATA TRANSMIT/RECEIVE indicates the direction of data transfer to and from the L-Bus. It is low during $T_a$ and $T_d$ cycles for a read or interrupt acknowledgement; it is high during $T_a$ and $T_d$ cycles for a write. DT/R never changes state when DEN is asserted (see Timing Diagrams).</td>
</tr>
<tr>
<td>DEN</td>
<td>O</td>
<td>DATA ENABLE is asserted during $T_d$ cycles and indicates transfer of data on the LAD bus lines.</td>
</tr>
<tr>
<td>READY</td>
<td>I</td>
<td>READY indicates that data on LAD lines can be sampled or removed. If READY is not asserted during a $T_d$ cycle, the $T_d$ cycle is extended to the next cycle by inserting a wait state ($T_w$), and ADS is not asserted in the next cycle.</td>
</tr>
<tr>
<td>LOCK</td>
<td>I/O</td>
<td>BUS LOCK prevents other bus masters from gaining control of the L-Bus following the current cycle (if they would assert LOCK to do so). LOCK is used by the processor or any bus agent when it performs indivisible Read/Modify/Write (RMW) operations. For a read that is designated as a RMW-read, LOCK is examined. If asserted, the processor waits until it is not asserted; if not asserted, the processor asserts LOCK during the $T_a$ cycle and leaves it asserted. A write that is designated as an RMW-write deasserts LOCK in the $T_a$ cycle. During the time LOCK is asserted, a bus agent can perform a normal read or write but no RMW operations. LOCK is also held asserted during an interrupt-acknowledge transaction.</td>
</tr>
<tr>
<td>BE3–BE0</td>
<td>O</td>
<td>BYTE ENABLE LINES specify which data bytes (up to four) on the bus take part in the current bus cycle. BE3 corresponds to LAD31–LAD24 and BE0 corresponds to LAD7–LAD0. The byte enables are provided in advance of data. The byte enables asserted during $T_a$ specify the bytes of the first data word. The byte enables asserted during $T_d$ specify the bytes of the next data word (if any), that is, the word to be transmitted following the next assertion of READY. The byte enables during the $T_d$ cycles preceding the last assertion of READY are undefined. The byte enables are latched on-chip and remain constant from one $T_d$ cycle to the next when READY is not asserted. For reads, the byte enables specify the byte(s) that the processor will actually use. L-Bus agents are required to assert only adjacent byte enables (e.g., asserting just BE0 and BE2 is not permitted), and are required to assert at least one byte enable. Accesses must also be naturally aligned (e.g., asserting BE0 and BE2 is not allowed even though they are adjacent). To produce address bits A0 and A1 externally, they can be decoded from the byte enables.</td>
</tr>
</tbody>
</table>

I/O = Input/Output, O = Output, I = Input, O.D. = Open-Drain, T.S. = tri-state
### Table 4a. 80960KB Pin Description: L-Bus Signals (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Name and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOLD/ HLDAR</td>
<td>I</td>
<td>HOLD: If the processor is the primary bus master (PBM), the input is interpreted as HOLD, a request from a secondary bus master to acquire the bus. When the processor receives HOLD and grants another master control of the bus, it floats its tri-state bus lines and then asserts HLDA and enters the $T_1$ state. When HOLD is deasserted, the processor will deassert HLDA and go to either the $T_1$ or $T_a$ state.</td>
</tr>
<tr>
<td>HLDA/ HOLDR</td>
<td>O</td>
<td>HOLD ACKNOWLEDGE: If the processor is a primary bus master (PBM), the input is HLDAR, which indicates, when HOLDR output is high, that the processor has acquired the bus. Processors and other agents can be told at reset if they are the primary bus master (PBM).</td>
</tr>
<tr>
<td>CACHE</td>
<td>O</td>
<td>CACHE indicates if an access is cacheable during a $T_a$ cycle. It is not asserted during any synchronous access, such as a synchronous load/store instruction used for sending an IAC message. The CACHE signal floats to a high impedance state when the processor is idle.</td>
</tr>
</tbody>
</table>

### Table 4b. 80960KB Pin Description: Module Support Signals

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Name and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BADAC</td>
<td>I</td>
<td>BAD ACCESS, if asserted in the cycle following the one in which the last READY of a transaction is asserted, indicates that an unrecoverable error has occurred on the current bus transaction, or that a synchronous load/store instruction has not been acknowledged.</td>
</tr>
<tr>
<td>RESET</td>
<td>I</td>
<td>RESET clears the internal logic of the processor and causes it to re-initialize.</td>
</tr>
<tr>
<td>FAILURE</td>
<td>O</td>
<td>INITIALIZATION FAILURE indicates that the processor has failed to initialize correctly. After RESET is deasserted and before the first bus transaction begins, FAILURE is asserted while the processor performs a self-test. If the self-test completes successfully, then FAILURE is deasserted. Next, the processor performs a zero checksum on the first eight words of memory. If it fails, FAILURE is asserted for a second time and remains asserted; if it passes, system initialization continues and FAILURE remains deasserted.</td>
</tr>
<tr>
<td>N.C.</td>
<td>N/A</td>
<td>NOT CONNECTED indicates pins should not be connected. Never connect any pin marked N.C.</td>
</tr>
</tbody>
</table>

1/0 = Input/Output, O = Output, I = Input, O.D. = Open-Drain, T.S. = tri-state
Table 4b. 80960KB Pin Description: Module Support Signals (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Name and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAC</td>
<td>I</td>
<td>INTERAGENT COMMUNICATION REQUEST/INTERRUPT 0 indicates either that there is a pending IAC message for the processor or an interrupt. The bus interrupt control register determines in which way the signal should be interpreted. To signal an interrupt or IAC request in a synchronous system, this pin (as well as the other interrupt pins) must be enabled by being deasserted for at least one bus cycle and then asserted for at least one additional bus cycle; in an asynchronous system, the pin must remain deasserted for at least two bus cycles and then be asserted for at least two more bus cycles.</td>
</tr>
<tr>
<td>INTO</td>
<td>I</td>
<td>LOCAL PROCESSOR NUMBER: This signal is interpreted differently during system reset. If the signal is at a high voltage level, it indicates that this processor is a primary bus master (Local Processor Number = 0); if it is at a low voltage level, it indicates that this processor is a secondary bus master (Local Processor Number = 1).</td>
</tr>
<tr>
<td>INT1</td>
<td>I</td>
<td>INTERRUPT 1, like INTO, provides direct interrupt signaling.</td>
</tr>
<tr>
<td>INT2/INTR</td>
<td>I</td>
<td>INTERRUPT 2/INTERRUPT REQUEST: The bus control registers determines how this pin is interpreted. If INT2, it has the same interpretation as the INTO and INT1 pins. If INTR, it is used to receive an interrupt request from an external interrupt controller.</td>
</tr>
<tr>
<td>INT3/INTA</td>
<td>I/O</td>
<td>INTERRUPT 3/INTERRUPT ACKNOWLEDGE: The bus interrupt control register determines how this pin is interpreted. If INT3, it has the same interpretation as the INTO, INT1, and INT2 pins. If INTA, it is used as an output to control interrupt-acknowledge bus transactions. The INTA output is latched on-chip and remains valid during ( T_d ) cycles; as an output, it is open-drain.</td>
</tr>
</tbody>
</table>

I/O = Input/Output, O = Output, I = Input, O.D. = Open-Drain, T.S. = tri-state

ELECTRICAL SPECIFICATIONS

Power and Grounding

The 80960KB is implemented in CHMOS III technology and has modest power requirements. Its high clock frequency and numerous output buffers (address/data, control, error, and arbitration signals) can cause power surges as multiple output buffers drive new signal levels simultaneously. For clean on-chip power distribution at high frequency, 11 \( V_{CC} \) and 13 \( V_{SS} \) pins separately feed functional units of the 80960KB.

Power and ground connections must be made to all power and ground pins of the 80960KB. On the circuit board, all \( V_{CC} \) pins must be strapped closely together, preferably on a power plane. Likewise, all \( V_{SS} \) pins should be strapped together, preferably on a ground plane. These pins may not be connected together within the chip.

Power Decoupling Recommendations

Liberal decoupling capacitance should be placed near the 80960KB. The processor can cause transient power surges when driving the L-Bus, particularly when it is connected to a large capacitive load.

Low inductance capacitors and interconnects are recommended for best high frequency electrical performance. Inductance can be reduced by shortening the board traces between the processor and decoupling capacitors as much as possible. Capacitors specifically designed for PGA packages are also commercially available and offer the lowest possible inductance.

Connection Recommendations

For reliable operation, always connect unused inputs to an appropriate signal level. In particular, if one or more interrupt lines are not used, they should be pulled up. No inputs should ever be left floating.
All open-drain outputs require a pullup device. While in some cases a simple pullup resistor will be adequate, we recommend a network of pullup and pulldown resistors biased to a valid $V_{IH} (\geq 3.4V)$ and terminated in the characteristic impedance of the circuit board. Figure 6 shows our recommendations for the resistor values for both a low and high current drive network, which assumes that the circuit board has a characteristic impedance of 100Ω. The advantage of terminating the output signals in this fashion is that it limits signal swing and reduces AC power consumption.

**Characteristic Curves**

Figure 7 shows the typical supply current requirements over the operating temperature range of the processor at supply voltage ($V_{CC}$) of 5V. Figure 8 shows the typical power supply current ($I_{CC}$) required by the 80960KB at various operating frequencies when measured at three input voltage ($V_{CC}$) levels.

For a given output current ($I_{OL}$), the curve in Figure 9 shows the worst case output low voltage ($I_{OL}$).

Figure 10 shows the typical capacitive derating curve for the 80960KB measured from 1.5V on the system clock (CLK) to 0.8V on the falling edge and 2.0V on the rising edge of the L-Bus address/data (LAD) signals.

**Test Load Circuit**

Figure 13 illustrates the load circuit used to test the 80960KB's tristate pins, and Figure 14 shows the load circuit used to test the open drain outputs. The open drain test uses an active load circuit in the form of a matched diode bridge. Since the open-drain only sink current, however, only the $I_{OL}$ legs of the bridge are necessary and the $I_{OH}$ legs are not used. When the 80960KB driver under test is turned off, the output pin is pulled up to $V_{REF}$ (i.e., $V_{OH}$). Diode $D_1$ is turned off and the $I_{OL}$ current source flows through diode $D_2$.

When the 80960KB open-drain driver under test is on, diode $D_1$ is also on, and the voltage on the pin being tested drops to $V_{OL}$. Diode $D_2$ turns off and $I_{OL}$ flows through diode $D_1$.

![Diagram](270565-25)

**Figure 6. Connection Recommendations for Low and High Current Drive Networks**

![Diagram](270565-26)
**Figure 7.** Typical Supply Current (I\textsubscript{CC})

**Figure 8.** Typical Current vs Frequency

**Figure 9.** Worst Case Voltage vs Output Current on Open-Drain Pins

**Figure 10.** Capacitive Derating Curve
**ABSOLUTE MAXIMUM RATINGS**

Operating Temperature .......... 0°C to +85°C
Storage Temperature .......... -65°C to +150°C
Voltage on Any Pin .......... -0.5V to VCC + 0.5V
Power Dissipation .......... 2.9W (20 MHz)

*Notice: Stresses above those listed under “Absolute Maximum Ratings” may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

**D.C. CHARACTERISTICS**

80960KB (16 MHz): TCASE = 0°C to +85°C, VCC = 5V ± 10%
80960KB (20 MHz): TCASE = 0°C to +85°C, VCC = 5V ± 5%

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
<th>Test Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIL</td>
<td>Input Low Voltage</td>
<td>-0.3</td>
<td>+0.8</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>VIH</td>
<td>Input High Voltage</td>
<td>2.0</td>
<td>VCC + 0.3</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>VCL</td>
<td>CLK2 Input Low Voltage</td>
<td>-0.3</td>
<td>+1.0</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>VCH</td>
<td>CLK2 Input High Voltage</td>
<td>0.55 VCC</td>
<td>VCC + 0.3</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>VOL</td>
<td>Output Low Voltage</td>
<td>0.45(5)</td>
<td>0.60(6)</td>
<td>V</td>
<td>(1)</td>
</tr>
<tr>
<td>VOH</td>
<td>Output High Voltage</td>
<td>2.4</td>
<td>V</td>
<td>(2, 4)</td>
<td></td>
</tr>
<tr>
<td>ICC</td>
<td>Power Supply Current:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 MHz</td>
<td>475</td>
<td>mA</td>
<td>TA = 0°C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 MHz</td>
<td>545</td>
<td>mA</td>
<td>TA = 0°C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILI</td>
<td>Input Leakage Current</td>
<td>±15</td>
<td>µA</td>
<td>0 ≤ VO ≤ VCC</td>
<td></td>
</tr>
<tr>
<td>ILO</td>
<td>Output Leakage Current</td>
<td>±15</td>
<td>µA</td>
<td>0.45 ≤ VO ≤ VCC</td>
<td></td>
</tr>
<tr>
<td>CIN</td>
<td>Input Capacitance</td>
<td>10</td>
<td>pF</td>
<td>fC = 1 MHz(3)</td>
<td></td>
</tr>
<tr>
<td>CO</td>
<td>I/O or Output Capacitance</td>
<td>12</td>
<td>pF</td>
<td>fC = 1 MHz(3)</td>
<td></td>
</tr>
<tr>
<td>CLK</td>
<td>Clock Capacitance</td>
<td>10</td>
<td>pF</td>
<td>fC = 1 MHz(3)</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

1. For tri-state outputs, this parameter is measured at:
   Address/Data ................................................................. 4.0 mA
   Controls ................................................................. 5.0 mA
2. This parameter is measured at:
   Address/Data ................................................................. -1.0 mA
   Controls ................................................................. -0.9 mA
   ALE ................................................................. -5.0 mA
3. Input, output, and clock capacitance are not tested.
4. Not measured on open-drain outputs.
5. For open-drain outputs ............................................... 25 mA
6. For open-drain outputs ............................................... 40 mA
AC SPECIFICATIONS

This section describes the AC specifications for the 80960KB pins. All input and output timings are specified relative to the 1.5V level of the rising edge of CLK2, and refer to the time at which the signal reaches (for output delay and input setup) or leaves (for hold time) the TTL levels of LOW (0.8V) or HIGH (2.0V). All AC testing should be done with input voltages of 0.4V and 2.4V, except for the clock (CLK2), which should be tested with input voltages of 0.45V and 0.55 VCC.

Figure 11. Drive Levels and Timing Relationships for 80960KB Signals
Figure 12. Timing Relationship of L-Bus Signals
## A.C. Specification Tables

### 80960KB A.C. Characteristics (16 MHz)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
<th>Test Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>Processor Clock</td>
<td>31.25</td>
<td>125</td>
<td>ns</td>
<td>(V_{IN} = 1.5)V</td>
</tr>
<tr>
<td></td>
<td>Period (CLK2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td>Processor Clock</td>
<td>11</td>
<td></td>
<td>ns</td>
<td>(V_{IL} = 10%) Point\n( = 1.2)V</td>
</tr>
<tr>
<td></td>
<td>Low Time (CLK2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₃</td>
<td>Processor Clock</td>
<td>11</td>
<td></td>
<td>ns</td>
<td>(V_{IL} = 90%) Point\n( = 0.1)V (+ 0.5 )V₃₃</td>
</tr>
<tr>
<td></td>
<td>High Time (CLK2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₄</td>
<td>Processor Clock</td>
<td>10</td>
<td></td>
<td>ns</td>
<td>(V_{IN} = 90%) Point to 10% \Point</td>
</tr>
<tr>
<td></td>
<td>Fall Time (CLK2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₅</td>
<td>Processor Clock</td>
<td>10</td>
<td></td>
<td>ns</td>
<td>(V_{IN} = 10%) Point to 90% \Point</td>
</tr>
<tr>
<td></td>
<td>Rise Time (CLK2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₆</td>
<td>Output Valid Delay</td>
<td>5</td>
<td>35</td>
<td>ns</td>
<td>(C_L = 100) pF (LAD) \n(C_L = 75) pF (Controls)</td>
</tr>
<tr>
<td>T₇</td>
<td>ALE Width</td>
<td>15</td>
<td></td>
<td>ns</td>
<td>(C_L = 75) pF</td>
</tr>
<tr>
<td>T₈</td>
<td>ALE Output Valid Delay</td>
<td>5</td>
<td>20</td>
<td>ns</td>
<td>(C_L = 75) pF( )²</td>
</tr>
<tr>
<td>T₉</td>
<td>Output Float Delay</td>
<td>5</td>
<td>20</td>
<td>ns</td>
<td>(C_L = 100) pF (LAD) \n(C_L = 75) pF (Controls)( )²</td>
</tr>
<tr>
<td>T₁₀</td>
<td>Input Setup 1</td>
<td>3</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>T₁₁</td>
<td>Input Hold</td>
<td>10</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>T₁₂</td>
<td>Input Setup 2</td>
<td>8</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>T₁₃</td>
<td>Setup to ALE Inactive</td>
<td>10</td>
<td></td>
<td>ns</td>
<td>(C_L = 100) pF (LAD) \n(C_L = 75) pF (Controls)</td>
</tr>
<tr>
<td>T₁₄</td>
<td>Hold after ALE Inactive</td>
<td>8</td>
<td></td>
<td>ns</td>
<td>(C_L = 100) pF (LAD) \n(C_L = 75) pF (Controls)</td>
</tr>
<tr>
<td>T₁₅</td>
<td>Reset Hold</td>
<td>5</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>T₁₆</td>
<td>Reset Setup</td>
<td>8</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>T₁₇</td>
<td>Reset Width</td>
<td>1281</td>
<td></td>
<td>ns</td>
<td>(41) CLK2 Periods Minimum</td>
</tr>
</tbody>
</table>

### NOTES:

1. \(\text{IAC/INT}_0, \text{INT}_1, \text{INT}_2/\text{INTR}, \text{INT}_3\) can be asynchronous.
2. A float condition occurs when the maximum output current becomes less than \(I_{LO}\). Float delay is not tested, but should be no longer than the valid delay.
# 80960KB AC Characteristics (20 MHz)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
<th>Test Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&lt;sub&gt;1&lt;/sub&gt;</td>
<td>Processor Clock Period (CLK2)</td>
<td>25</td>
<td>125</td>
<td>ns</td>
<td>( V_{IN} = 1.5V )</td>
</tr>
<tr>
<td>T&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Processor Clock Low Time (CLK2)</td>
<td>8</td>
<td></td>
<td>ns</td>
<td>( V_{IL} = 10% ) Point ( = 1.2V )</td>
</tr>
<tr>
<td>T&lt;sub&gt;3&lt;/sub&gt;</td>
<td>Processor Clock High Time (CLK2)</td>
<td>8</td>
<td></td>
<td>ns</td>
<td>( V_{IL} = 90% ) Point ( = 0.1V + 0.5V_{CC} )</td>
</tr>
<tr>
<td>T&lt;sub&gt;4&lt;/sub&gt;</td>
<td>Processor Clock Fall Time (CLK2)</td>
<td>10</td>
<td></td>
<td>ns</td>
<td>( V_{IN} = 90% ) Point to 10% Point</td>
</tr>
<tr>
<td>T&lt;sub&gt;5&lt;/sub&gt;</td>
<td>Processor Clock Rise Time (CLK2)</td>
<td>10</td>
<td></td>
<td>ns</td>
<td>( V_{IN} = 10% ) Point to 90% Point</td>
</tr>
<tr>
<td>T&lt;sub&gt;6&lt;/sub&gt;</td>
<td>Output Valid Delay</td>
<td>5</td>
<td>30</td>
<td>ns</td>
<td>( C_{L} = 60 \text{ pF (LAD)} ) ( C_{L} = 50 \text{ pF (Controls)} )</td>
</tr>
<tr>
<td>T&lt;sub&gt;7&lt;/sub&gt;</td>
<td>ALE Width</td>
<td>12</td>
<td></td>
<td>ns</td>
<td>( C_{L} = 50 \text{ pF} )</td>
</tr>
<tr>
<td>T&lt;sub&gt;8&lt;/sub&gt;</td>
<td>ALE Output Valid Delay</td>
<td>5</td>
<td>20</td>
<td>ns</td>
<td>( C_{L} = 50 \text{ pF (2)} )</td>
</tr>
<tr>
<td>T&lt;sub&gt;9&lt;/sub&gt;</td>
<td>Output Float Delay</td>
<td>5</td>
<td>20</td>
<td>ns</td>
<td>( C_{L} = 60 \text{ pF (LAD)} ) ( C_{L} = 50 \text{ pF (Controls)} )</td>
</tr>
<tr>
<td>T&lt;sub&gt;10&lt;/sub&gt;</td>
<td>Input Setup 1</td>
<td>3</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>T&lt;sub&gt;11&lt;/sub&gt;</td>
<td>Input Hold</td>
<td>10</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>T&lt;sub&gt;12&lt;/sub&gt;</td>
<td>Input Setup 2</td>
<td>7</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>T&lt;sub&gt;13&lt;/sub&gt;</td>
<td>Setup to ALE Inactive</td>
<td>10</td>
<td></td>
<td>ns</td>
<td>( C_{L} = 60 \text{ pF (LAD)} ) ( C_{L} = 50 \text{ pF (Controls)} )</td>
</tr>
<tr>
<td>T&lt;sub&gt;14&lt;/sub&gt;</td>
<td>Hold after ALE Inactive</td>
<td>8</td>
<td></td>
<td>ns</td>
<td>( C_{L} = 60 \text{ pF (LAD)} ) ( C_{L} = 50 \text{ pF (Controls)} )</td>
</tr>
<tr>
<td>T&lt;sub&gt;15&lt;/sub&gt;</td>
<td>Reset Hold</td>
<td>5</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>T&lt;sub&gt;16&lt;/sub&gt;</td>
<td>Reset Setup</td>
<td>7</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>T&lt;sub&gt;17&lt;/sub&gt;</td>
<td>Reset Width</td>
<td>1025</td>
<td></td>
<td>ns</td>
<td>41 CLK2 Periods Minimum</td>
</tr>
</tbody>
</table>

**NOTES:**
1. \( IAC/INT_0, INT_1, INT_2/INTR, INT_3 \) can be asynchronous.
2. A float condition occurs when the maximum output current becomes less than \( I_{LO} \). Float delay is not tested, but should be no longer than the valid delay.

---

**Figure 13. Test Load Circuit for TRI-STATE Output Pins**

**Figure 14. Test Load Circuit for Open-Drain Output Pins**

---

4-19
Figure 15. Processor Clock Pulse (CLK2)

Figure 16. RESET Signal Timing

INIT PARAMETERS (BADAC, IAC) MUST BE SETUP 8 CLOCKS PRIOR TO THIS CLK2 EDGE
INIT PARAMETERS MUST BE HELD BEYOND THIS CLK2 EDGE

T15 = RESET HOLD
T16 = RESET SETUP
T17 = RESET WIDTH
Design Considerations

Input hold times can be disregarded by the designer whenever the input is removed because a subse­quent output from the processor is deasserted (e.g., DEN becomes deasserted).

In other words, whenever the processor generates an output that indicates a transition into a subse­quent state, the processor must have sampled any inputs for the previous state. As an example, in the T<sub>r</sub> cycle following a read, the minimum time that DEN becomes deasserted is 5 ns, but the minimum hold time on the data is 10 ns. When DEN is deas­serted, however, the data is guaranteed to have been sampled.

Similarly, whenever the processor generates an output that indicates a transition into a subsequent state, any outputs that are specified to be tri-stated in this new state are guaranteed to be tri-stated. For example, in the T<sub>ch</sub> cycle following a T<sub>ch</sub> cycle for a read, the minimum output delay of DEN is 5 ns, but the maximum float time of LAD is 20 ns. When DEN is asserted, however, the LAD outputs are guaran­teed to have been tri-stated.

Designing for the ICE-960KB

The 80960KB In-Circuit Emulator assists in debug­ging 80960KB hardware and software designs. The product consists of a probe module, cable, and control unit. Because of the high operating frequency of 80960KB systems, the probe module connects di­rectly to the 80960KB socket.

When designing an 80960KB hardware system that uses the ICE-960KB to debug the system, several electrical and mechanical characteristics should be considered. These considerations include capacitive loading, drive requirement, power requirement, and physical layout.

The ICE-960KB probe module increases the load capacitance of each line by up to 25 pF. It also adds one standard Schottky TTL load on the CLK2 line, up to one advanced low-power Schottky TTL load for each control signal line, and one advanced low­power Schottky TTL load for each address/data and byte enable line. These loads originate from the probe module and are driven by the 80960KB processor.

To achieve high noise immunity, the ICE-960KB probe is powered by the user's system. The high­speed probe circuitry draws up to 1.1A plus the max­imum current (I<sub>cc</sub>) of the 80960KB processor.

The mechanical considerations are shown in Figure 18, which illustrates the lateral clearance require­ments for the ICE-960KB probe as viewed from above the socket of the 80960KB processor.
MECHANICAL DATA

Pin Assignment

The 80960KB pinout as viewed from the substrate side of the component is shown in Figure 19 and from the pin side in Figure 20.

VCC and GND connections must be made to multiple VCC and GND pins. Each VCC and GND pin must be connected to the appropriate voltage or ground and externally strapped close to the package. Preferably, the circuit board should include power and ground planes for power distribution. Table 5 and Table 6 list the function of each pin.

NOTE:
Pins identified as N.C., “No Connect,” should never be connected under any circumstances. The 80960KB component contains 54 N.C. pins.

Package Dimensions and Mounting

The 80960KB is packaged in a 132-lead ceramic pin-grid array (PGA). Pins in this package are arranged 0.100 inch (2.54mm) center-to-center, in a 14 by 14 matrix, three rows around. (See Figure 21.)

A wide variety of available sockets allow low-insertion or zero-insertion force mountings, and a choice of terminals such as soldertail, surface mount, or wire wrap. Several applicable sockets are shown in Figure 22.

Package Thermal Specification

The 80960KB is specified for operation when case temperature is within the range 0°C to +85°C. The PGA case temperature should be measured at the center of the top surface of the package opposite the pins as shown in Figure 23.

The ambient temperature can be calculated from the \( \theta_{jc} \) and \( \theta_{ja} \) by using the following equations:

\[
T_J = T_C + P \cdot \theta_{jc} \\
T_A = T_J - P \cdot \theta_{ja} \\
T_C = T_A + P \cdot \left( \theta_{ja} - \theta_{jc} \right)
\]

Values for \( \theta_{ja} \) and \( \theta_{jc} \) are given in Table 7 at various airflows. Note that \( \theta_{ja} \) can be reduced by attaching a heatsink to the package. The maximum allowable ambient temperature (\( T_A \)) permitted without exceeding \( T_C \) is shown by the curve in Figure 25. The curve assumes an \( I_{CC} \) of 545 mA, \( V_{CC} \) of 5.0V, and a \( T_{CASE} \) of +85°C.

WAVEFORMS

Figures 24 through 31 show the waveforms for various transactions on the 80960KB's local bus.
Figure 19. 80960KB PGA Pinout—View from Bottom (Pins Facing Up)
Figure 20. 80960KB PGA Pinout—View from Top (Pins Facing Down)
<table>
<thead>
<tr>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>VCC</td>
<td>C6</td>
<td>LAD20</td>
<td>H1</td>
<td>W/ R</td>
<td>M10</td>
<td>VSS</td>
</tr>
<tr>
<td>A2</td>
<td>VSS</td>
<td>C7</td>
<td>LAD13</td>
<td>H2</td>
<td>BE0</td>
<td>M11</td>
<td>VCC</td>
</tr>
<tr>
<td>A3</td>
<td>LAD19</td>
<td>C8</td>
<td>LAD8</td>
<td>H3</td>
<td>LOCK</td>
<td>M12</td>
<td>N.C.</td>
</tr>
<tr>
<td>A4</td>
<td>LAD17</td>
<td>C9</td>
<td>LAD3</td>
<td>H12</td>
<td>N.C.</td>
<td>M13</td>
<td>N.C.</td>
</tr>
<tr>
<td>A5</td>
<td>LAD16</td>
<td>C10</td>
<td>VCC</td>
<td>H13</td>
<td>N.C.</td>
<td>M14</td>
<td>N.C.</td>
</tr>
<tr>
<td>A6</td>
<td>LAD14</td>
<td>C11</td>
<td>VSS</td>
<td>H14</td>
<td>N.C.</td>
<td>N1</td>
<td>VSS</td>
</tr>
<tr>
<td>A7</td>
<td>LAD11</td>
<td>C12</td>
<td>INT3/INTA</td>
<td>J1</td>
<td>DT/R</td>
<td>N2</td>
<td>N.C.</td>
</tr>
<tr>
<td>A8</td>
<td>LAD9</td>
<td>C13</td>
<td>INT1</td>
<td>J2</td>
<td>BE2</td>
<td>N3</td>
<td>N.C.</td>
</tr>
<tr>
<td>A9</td>
<td>LAD7</td>
<td>C14</td>
<td>IAC/INT0</td>
<td>J3</td>
<td>VSS</td>
<td>N4</td>
<td>N.C.</td>
</tr>
<tr>
<td>A10</td>
<td>LAD5</td>
<td>D1</td>
<td>ALE</td>
<td>J12</td>
<td>N.C.</td>
<td>N5</td>
<td>N.C.</td>
</tr>
<tr>
<td>A11</td>
<td>LAD4</td>
<td>D2</td>
<td>ADS</td>
<td>J13</td>
<td>N.C.</td>
<td>N6</td>
<td>N.C.</td>
</tr>
<tr>
<td>A12</td>
<td>LAD1</td>
<td>D3</td>
<td>HLDA/HLDR</td>
<td>J14</td>
<td>N.C.</td>
<td>N7</td>
<td>N.C.</td>
</tr>
<tr>
<td>A13</td>
<td>INT2/INTR</td>
<td>D12</td>
<td>VCC</td>
<td>K1</td>
<td>BE3</td>
<td>N8</td>
<td>N.C.</td>
</tr>
<tr>
<td>A14</td>
<td>VCC</td>
<td>D13</td>
<td>N.C.</td>
<td>K2</td>
<td>FAILURE</td>
<td>N9</td>
<td>N.C.</td>
</tr>
<tr>
<td>B1</td>
<td>LAD23</td>
<td>D14</td>
<td>N.C.</td>
<td>K3</td>
<td>VSS</td>
<td>N10</td>
<td>N.C.</td>
</tr>
<tr>
<td>B2</td>
<td>LAD24</td>
<td>E1</td>
<td>LAD28</td>
<td>K12</td>
<td>VCC</td>
<td>N11</td>
<td>N.C.</td>
</tr>
<tr>
<td>B3</td>
<td>LAD22</td>
<td>E2</td>
<td>LAD26</td>
<td>K13</td>
<td>N.C.</td>
<td>N12</td>
<td>N.C.</td>
</tr>
<tr>
<td>B4</td>
<td>LAD21</td>
<td>E3</td>
<td>LAD27</td>
<td>K14</td>
<td>N.C.</td>
<td>N13</td>
<td>N.C.</td>
</tr>
<tr>
<td>B5</td>
<td>LAD18</td>
<td>E12</td>
<td>N.C.</td>
<td>L1</td>
<td>DEN</td>
<td>N14</td>
<td>N.C.</td>
</tr>
<tr>
<td>B6</td>
<td>LAD15</td>
<td>E13</td>
<td>VSS</td>
<td>L2</td>
<td>N.C.</td>
<td>P1</td>
<td>VCC</td>
</tr>
<tr>
<td>B7</td>
<td>LAD12</td>
<td>E14</td>
<td>N.C.</td>
<td>L3</td>
<td>VCC</td>
<td>P2</td>
<td>N.C.</td>
</tr>
<tr>
<td>B8</td>
<td>LAD10</td>
<td>F1</td>
<td>LAD29</td>
<td>L12</td>
<td>VSS</td>
<td>P3</td>
<td>N.C.</td>
</tr>
<tr>
<td>B9</td>
<td>LAD6</td>
<td>F2</td>
<td>LAD31</td>
<td>L13</td>
<td>N.C.</td>
<td>P4</td>
<td>N.C.</td>
</tr>
<tr>
<td>B10</td>
<td>LAD2</td>
<td>F3</td>
<td>CACHE</td>
<td>L14</td>
<td>N.C.</td>
<td>P5</td>
<td>N.C.</td>
</tr>
<tr>
<td>B11</td>
<td>CLK2</td>
<td>F12</td>
<td>N.C.</td>
<td>M1</td>
<td>N.C.</td>
<td>P6</td>
<td>N.C.</td>
</tr>
<tr>
<td>B12</td>
<td>LAD0</td>
<td>F13</td>
<td>N.C.</td>
<td>M2</td>
<td>N.C.</td>
<td>P7</td>
<td>N.C.</td>
</tr>
<tr>
<td>B13</td>
<td>RESET</td>
<td>F14</td>
<td>N.C.</td>
<td>M3</td>
<td>VSS</td>
<td>P8</td>
<td>N.C.</td>
</tr>
<tr>
<td>B14</td>
<td>VSS</td>
<td>G1</td>
<td>LAD30</td>
<td>M4</td>
<td>VSS</td>
<td>P9</td>
<td>N.C.</td>
</tr>
<tr>
<td>C1</td>
<td>HOLD/HLDAR</td>
<td>G2</td>
<td>READY</td>
<td>M5</td>
<td>VCC</td>
<td>P10</td>
<td>N.C.</td>
</tr>
<tr>
<td>C2</td>
<td>LAD25</td>
<td>G3</td>
<td>BE1</td>
<td>M6</td>
<td>N.C.</td>
<td>P11</td>
<td>N.C.</td>
</tr>
<tr>
<td>C3</td>
<td>BADAC</td>
<td>G12</td>
<td>N.C.</td>
<td>M7</td>
<td>N.C.</td>
<td>P12</td>
<td>N.C.</td>
</tr>
<tr>
<td>C4</td>
<td>VCC</td>
<td>G13</td>
<td>N.C.</td>
<td>M8</td>
<td>N.C.</td>
<td>P13</td>
<td>VSS</td>
</tr>
<tr>
<td>C5</td>
<td>VSS</td>
<td>G14</td>
<td>N.C.</td>
<td>M9</td>
<td>N.C.</td>
<td>P14</td>
<td>VCC</td>
</tr>
</tbody>
</table>
Table 6. 80960KB PGA Pinout—in Signal Order

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADS</td>
<td>D2</td>
<td>LAD15</td>
<td>B6</td>
<td>N.C.</td>
<td>J14</td>
<td>N.C.</td>
<td>P8</td>
</tr>
<tr>
<td>ALE</td>
<td>D1</td>
<td>LAD16</td>
<td>A5</td>
<td>N.C.</td>
<td>K13</td>
<td>N.C.</td>
<td>P9</td>
</tr>
<tr>
<td>BADAC</td>
<td>C3</td>
<td>LAD17</td>
<td>A4</td>
<td>N.C.</td>
<td>K14</td>
<td>N.C.</td>
<td>P10</td>
</tr>
<tr>
<td>BE0</td>
<td>H2</td>
<td>LAD18</td>
<td>B5</td>
<td>N.C.</td>
<td>L13</td>
<td>N.C.</td>
<td>P11</td>
</tr>
<tr>
<td>BE1</td>
<td>G3</td>
<td>LAD19</td>
<td>A3</td>
<td>N.C.</td>
<td>L14</td>
<td>N.C.</td>
<td>P12</td>
</tr>
<tr>
<td>BE2</td>
<td>J2</td>
<td>LAD20</td>
<td>C6</td>
<td>N.C.</td>
<td>M1</td>
<td>N.C.</td>
<td>L2</td>
</tr>
<tr>
<td>BE3</td>
<td>K1</td>
<td>LAD21</td>
<td>B4</td>
<td>N.C.</td>
<td>M2</td>
<td>READY</td>
<td>G2</td>
</tr>
<tr>
<td>CACHE</td>
<td>F3</td>
<td>LAD22</td>
<td>B3</td>
<td>N.C.</td>
<td>M6</td>
<td>RESET</td>
<td>B13</td>
</tr>
<tr>
<td>CLD2</td>
<td>B11</td>
<td>LAD23</td>
<td>B1</td>
<td>N.C.</td>
<td>M7</td>
<td>VCC</td>
<td>A1</td>
</tr>
<tr>
<td>DEN</td>
<td>L1</td>
<td>LAD24</td>
<td>B2</td>
<td>N.C.</td>
<td>M8</td>
<td>VCC</td>
<td>A14</td>
</tr>
<tr>
<td>DT/R</td>
<td>J1</td>
<td>LAD25</td>
<td>C2</td>
<td>N.C.</td>
<td>M9</td>
<td>VCC</td>
<td>C4</td>
</tr>
<tr>
<td>FAILURE</td>
<td>K2</td>
<td>LAD26</td>
<td>E2</td>
<td>N.C.</td>
<td>M12</td>
<td>VCC</td>
<td>C10</td>
</tr>
<tr>
<td>HLDA/HLDAR</td>
<td>D3</td>
<td>LAD27</td>
<td>E3</td>
<td>N.C.</td>
<td>M13</td>
<td>VCC</td>
<td>D12</td>
</tr>
<tr>
<td>HOLD/HLDAR</td>
<td>C1</td>
<td>LAD28</td>
<td>E1</td>
<td>N.C.</td>
<td>M14</td>
<td>VCC</td>
<td>K12</td>
</tr>
<tr>
<td>IAC/INT0</td>
<td>C14</td>
<td>LAD29</td>
<td>F1</td>
<td>N.C.</td>
<td>N2</td>
<td>VCC</td>
<td>L3</td>
</tr>
<tr>
<td>INT1</td>
<td>C13</td>
<td>LAD30</td>
<td>G1</td>
<td>N.C.</td>
<td>N3</td>
<td>VCC</td>
<td>M5</td>
</tr>
<tr>
<td>INT2/INTR</td>
<td>A13</td>
<td>LAD31</td>
<td>F2</td>
<td>N.C.</td>
<td>N4</td>
<td>VCC</td>
<td>M11</td>
</tr>
<tr>
<td>INT3/INTA</td>
<td>C12</td>
<td>LOCK</td>
<td>H3</td>
<td>N.C.</td>
<td>N5</td>
<td>VCC</td>
<td>P1</td>
</tr>
<tr>
<td>LAD0</td>
<td>B12</td>
<td>N.C.</td>
<td>D13</td>
<td>N.C.</td>
<td>N6</td>
<td>VCC</td>
<td>P14</td>
</tr>
<tr>
<td>LAD1</td>
<td>A12</td>
<td>N.C.</td>
<td>D14</td>
<td>N.C.</td>
<td>N7</td>
<td>VSS</td>
<td>A2</td>
</tr>
<tr>
<td>LAD2</td>
<td>B10</td>
<td>N.C.</td>
<td>E12</td>
<td>N.C.</td>
<td>N8</td>
<td>VSS</td>
<td>B14</td>
</tr>
<tr>
<td>LAD3</td>
<td>C9</td>
<td>N.C.</td>
<td>E14</td>
<td>N.C.</td>
<td>N9</td>
<td>VSS</td>
<td>C5</td>
</tr>
<tr>
<td>LAD4</td>
<td>A11</td>
<td>N.C.</td>
<td>F12</td>
<td>N.C.</td>
<td>N10</td>
<td>VSS</td>
<td>C11</td>
</tr>
<tr>
<td>LAD5</td>
<td>A10</td>
<td>N.C.</td>
<td>F13</td>
<td>N.C.</td>
<td>N11</td>
<td>VSS</td>
<td>E13</td>
</tr>
<tr>
<td>LAD6</td>
<td>B9</td>
<td>N.C.</td>
<td>F14</td>
<td>N.C.</td>
<td>N12</td>
<td>VSS</td>
<td>J3</td>
</tr>
<tr>
<td>LAD7</td>
<td>A9</td>
<td>N.C.</td>
<td>G12</td>
<td>N.C.</td>
<td>N13</td>
<td>VSS</td>
<td>K3</td>
</tr>
<tr>
<td>LAD8</td>
<td>C8</td>
<td>N.C.</td>
<td>G13</td>
<td>N.C.</td>
<td>N14</td>
<td>VSS</td>
<td>L12</td>
</tr>
<tr>
<td>LAD9</td>
<td>A8</td>
<td>N.C.</td>
<td>G14</td>
<td>N.C.</td>
<td>P2</td>
<td>VSS</td>
<td>M3</td>
</tr>
<tr>
<td>LAD10</td>
<td>B8</td>
<td>N.C.</td>
<td>H12</td>
<td>N.C.</td>
<td>P3</td>
<td>VSS</td>
<td>M4</td>
</tr>
<tr>
<td>LAD11</td>
<td>A7</td>
<td>N.C.</td>
<td>H13</td>
<td>N.C.</td>
<td>P4</td>
<td>VSS</td>
<td>M10</td>
</tr>
<tr>
<td>LAD12</td>
<td>B7</td>
<td>N.C.</td>
<td>H14</td>
<td>N.C.</td>
<td>P5</td>
<td>VSS</td>
<td>N1</td>
</tr>
<tr>
<td>LAD13</td>
<td>C7</td>
<td>N.C.</td>
<td>J12</td>
<td>N.C.</td>
<td>P6</td>
<td>VSS</td>
<td>P13</td>
</tr>
<tr>
<td>LAD14</td>
<td>A6</td>
<td>N.C.</td>
<td>J13</td>
<td>N.C.</td>
<td>P7</td>
<td>W/R</td>
<td>H1</td>
</tr>
</tbody>
</table>
Figure 21. A 132-Lead Pin-Grid Array (PGA) Used to Package the 80960KB
• Low insertion force (LIF) soldertail
  55274-1
• Amp tests indicate 50% reduction in insertion force compared to machined sockets

Other socket options
• Zero insertion force (ZIF) soldertail
  55583-1
• Zero insertion force (ZIF) Burn-in version 55573-2

Amp Incorporated
(Harrisburg, PA 17105 U.S.A
Phone 717-564-0100)

Cam handle locks in low profile position when 80960KB is installed
(handle UP for open and DOWN for closed positions).

Courtesy Amp Incorporated

Peel-A-Way* Mylar and Kapton Socket Terminal Carriers
• Low insertion force surface mount CS132-37TG
• Low insertion force soldertail CS132-01TG
• Low insertion force wire-wrap CS132-02TG (two-level)
  CS132-03TG (three-level)
• Low insertion force press-fit CS132-05TG

Advanced Interconnections
(5 Division Street)
Warwick, RI 02881 U.S.A.
Phone 401-885-0485)

Peel-A-Way Carrier No. 132:
Kapton Carrier is KS132
Mylar Carrier is MS132
Molded Plastic Body KS132
is shown below:

*Peel-A-Way is a trademark of Advanced Interconnections.

Figure 22. Several Socket Options for Mounting the 80960KB
MEASURE PGA CASE TEMPERATURE AT CENTER OF TOP SURFACE

Figure 23. Measuring 80960KB PGA Case Temperature

Figure 24. System and Processor Clock Relationship

\[ V_{CC} = 5.0V, \text{Freq.} = 20 \text{ MHz} \]

Figure 25. Maximum Allowable Ambient Temperature
Figure 26. Read Transaction
Table 7. 80960KB PGA Package Thermal Characteristics

<table>
<thead>
<tr>
<th>Parameter</th>
<th>0 (0)</th>
<th>50 (0.25)</th>
<th>100 (0.50)</th>
<th>200 (1.01)</th>
<th>400 (2.03)</th>
<th>600 (3.04)</th>
<th>800 (4.06)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Junction-to-Case (Case Measured as Figure 6-4)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Case-to-Ambient (No Heatsink)</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>Case-to-Ambient (with Omnidirectional Heatsink)</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Case-to-Ambient (with Unidirectional Heatsink)</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

NOTES:
1. Table 7 applies to 80960KB PGA plugged into socket or soldered directly into board.
2. $\theta_{JA} = \theta_{JC} + \theta_{CA}$.
3. $\theta_{J,CAP} = 4^\circ C/w$ (approx.)
   $\theta_{J,PIN} = 4^\circ C/w$ (inner pins) (approx.)
   $\theta_{J,PIN} = 8^\circ C/w$ (outer pins) (approx.)

Figure 27. Write Transaction with One Wait State
Figure 28. Burst Read Transaction

Figure 29. Burst Write Transaction with One Wait State
NOTE:
INTR can go low no sooner than 10 ns (input hold time) following the beginning of interrupt acknowledgement cycle 1. For a second interrupt to be acknowledged, INTR must be low for at least three cycles before it can be reasserted.

Figure 30. Interrupt Acknowledge Transaction
Figure 31. Bus Exchange Transaction (PBM = Primary Bus Master, SBM = Secondary Bus Master)
376™ HIGH PERFORMANCE
32-BIT EMBEDDED PROCESSOR

- Full 32-Bit Internal Architecture
  - 8-, 16-, 32-Bit Data Types
  - 8 General Purpose 32-Bit Registers
  - Extensive 32-Bit Instruction Set
- High Performance 16-Bit Data Bus
  - 16 MHz CPU Clock
  - Two-Clock Bus Cycles
  - 16 Mbytes/Sec Bus Bandwidth
- 16 Mbyte Physical Memory Size
- High Speed Numerics Support with the 80387SX
- Low System Cost with the 82370 Integrated System Peripheral
- On-Chip Debugging Support Including Break Point Registers

Complete Intel Development Support
- C, PL/M, Assembler Translators
- ICETM-376, In-Circuit Emulator
- iRMK Real Time Kernel

Extensive Third-Party Support:
- Software: C, Pascal, FORTRAN, BASIC and ADA*
- Hosts: VMS®, UNIX®, MS-DOS®, and Others
- Real-Time Kernels

High Speed CHMOS Technology

Available in 100 Pin Plastic Quad Flat-Pack Package and 88-Pin Pin Grid Array
(See Packaging Outlines and Dimensions #231369)

INTRODUCTION
The 376 32-bit embedded processor is designed for high performance embedded systems. It provides the performance benefits of a highly pipelined 32-bit internal architecture with the low system cost associated with 16-bit hardware systems. The 80376 is based on the 80386 and offers a high degree of compatibility with the 80386. All 80386 32-bit programs not dependent on paging can be executed on the 80376 and all 80376 programs can be executed on the 80386. All 32-bit 80386 language translators can be used for software development. With proper support software, any 80386-based computer can be used to develop and test 80376 programs. In addition, any 80386-based PC-AT* compatible computer can be used for hardware prototyping for designs based on the 80376 and its companion product the 82370.

*UNIX is a registered trademark of AT&T.
ADA is a registered trademark of the U.S. Government, Ada Joint Program Office.
PC-AT is a registered trademark of IBM Corporation.
VMS is a trademark of Digital Equipment Corporation.
MS-DOS is a trademark of MicroSoft Corporation.
## 1.0 PIN DESCRIPTION

![Figure 1.1. 80376 100-Pin Quad Flat-Pack Pin Out (Top View)](image)

### Table 1.1. 100-Pin Plastic Quad Flat-Pack Pin Assignments

<table>
<thead>
<tr>
<th>A Row</th>
<th>B Row</th>
<th>C Row</th>
<th>D Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin</td>
<td>Label</td>
<td>Pin</td>
<td>Label</td>
</tr>
<tr>
<td>1</td>
<td>$D_0$</td>
<td>26</td>
<td>LOCK#</td>
</tr>
<tr>
<td>2</td>
<td>$V_{SS}$</td>
<td>27</td>
<td>N/C</td>
</tr>
<tr>
<td>3</td>
<td>HLDA</td>
<td>28</td>
<td>N/C</td>
</tr>
<tr>
<td>4</td>
<td>HOLD</td>
<td>29</td>
<td>N/C</td>
</tr>
<tr>
<td>5</td>
<td>$V_{SS}$</td>
<td>30</td>
<td>N/C</td>
</tr>
<tr>
<td>6</td>
<td>NA#</td>
<td>31</td>
<td>N/C</td>
</tr>
<tr>
<td>7</td>
<td>READY#</td>
<td>32</td>
<td>$V_{CC}$</td>
</tr>
<tr>
<td>8</td>
<td>$V_{CC}$</td>
<td>33</td>
<td>RESET</td>
</tr>
<tr>
<td>9</td>
<td>$V_{CC}$</td>
<td>34</td>
<td>BUSY#</td>
</tr>
<tr>
<td>10</td>
<td>$V_{SS}$</td>
<td>35</td>
<td>$V_{SS}$</td>
</tr>
<tr>
<td>11</td>
<td>$V_{SS}$</td>
<td>36</td>
<td>ERROR#</td>
</tr>
<tr>
<td>12</td>
<td>$V_{SS}$</td>
<td>37</td>
<td>PEREQ</td>
</tr>
<tr>
<td>13</td>
<td>$V_{SS}$</td>
<td>38</td>
<td>NMI</td>
</tr>
<tr>
<td>14</td>
<td>$V_{SS}$</td>
<td>39</td>
<td>$V_{CC}$</td>
</tr>
<tr>
<td>15</td>
<td>CLK2</td>
<td>40</td>
<td>INTR</td>
</tr>
<tr>
<td>16</td>
<td>ADS#</td>
<td>41</td>
<td>$V_{SS}$</td>
</tr>
<tr>
<td>17</td>
<td>BLE#</td>
<td>42</td>
<td>$V_{CC}$</td>
</tr>
<tr>
<td>18</td>
<td>$A_1$</td>
<td>43</td>
<td>N/C</td>
</tr>
<tr>
<td>19</td>
<td>BHE#</td>
<td>44</td>
<td>N/C</td>
</tr>
<tr>
<td>20</td>
<td>N/C</td>
<td>45</td>
<td>N/C</td>
</tr>
<tr>
<td>21</td>
<td>$V_{CC}$</td>
<td>46</td>
<td>N/C</td>
</tr>
<tr>
<td>22</td>
<td>$V_{SS}$</td>
<td>47</td>
<td>N/C</td>
</tr>
<tr>
<td>23</td>
<td>M/IO#</td>
<td>48</td>
<td>$V_{CC}$</td>
</tr>
<tr>
<td>24</td>
<td>D/C#</td>
<td>49</td>
<td>$V_{SS}$</td>
</tr>
<tr>
<td>25</td>
<td>W/R#</td>
<td>50</td>
<td>$V_{SS}$</td>
</tr>
</tbody>
</table>
Figure 1.2. 80376 88-Pin Grid Array Pin Out

Table 1.2. 88-Pin Grid Array Pin Assignments

<table>
<thead>
<tr>
<th>Pin</th>
<th>Label</th>
<th>Pin</th>
<th>Label</th>
<th>Pin</th>
<th>Label</th>
<th>Pin</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>2H</td>
<td>CLK2</td>
<td>12D</td>
<td>A18</td>
<td>2L</td>
<td>M/IO#</td>
<td>11A</td>
<td>VCC</td>
</tr>
<tr>
<td>9B</td>
<td>D15</td>
<td>12E</td>
<td>A17</td>
<td>5M</td>
<td>LOCK#</td>
<td>13A</td>
<td>VCC</td>
</tr>
<tr>
<td>8A</td>
<td>D14</td>
<td>13E</td>
<td>A16</td>
<td>1J</td>
<td>ADS#</td>
<td>13C</td>
<td>VCC</td>
</tr>
<tr>
<td>8B</td>
<td>D13</td>
<td>12F</td>
<td>A15</td>
<td>1H</td>
<td>READY#</td>
<td>13L</td>
<td>VCC</td>
</tr>
<tr>
<td>7A</td>
<td>D12</td>
<td>13F</td>
<td>A14</td>
<td>2G</td>
<td>NA#</td>
<td>1N</td>
<td>VCC</td>
</tr>
<tr>
<td>7B</td>
<td>D11</td>
<td>12G</td>
<td>A13</td>
<td>1G</td>
<td>HOLD</td>
<td>13N</td>
<td>VCC</td>
</tr>
<tr>
<td>6A</td>
<td>D10</td>
<td>13G</td>
<td>A12</td>
<td>2F</td>
<td>HLDA</td>
<td>11B</td>
<td>VSS</td>
</tr>
<tr>
<td>6B</td>
<td>D9</td>
<td>13H</td>
<td>A11</td>
<td>7N</td>
<td>PEREQ</td>
<td>2C</td>
<td>VSS</td>
</tr>
<tr>
<td>5A</td>
<td>D8</td>
<td>12H</td>
<td>A10</td>
<td>7M</td>
<td>BUSY#</td>
<td>1D</td>
<td>VSS</td>
</tr>
<tr>
<td>5B</td>
<td>D7</td>
<td>13J</td>
<td>A9</td>
<td>8N</td>
<td>ERROR#</td>
<td>1M</td>
<td>VSS</td>
</tr>
<tr>
<td>4B</td>
<td>D6</td>
<td>12J</td>
<td>A8</td>
<td>9M</td>
<td>INTR</td>
<td>4N</td>
<td>VSS</td>
</tr>
<tr>
<td>4A</td>
<td>D5</td>
<td>12K</td>
<td>A7</td>
<td>6M</td>
<td>RESET</td>
<td>11N</td>
<td>VSS</td>
</tr>
<tr>
<td>3B</td>
<td>D4</td>
<td>13K</td>
<td>A6</td>
<td>8M</td>
<td>NMI</td>
<td>9N</td>
<td>VSS</td>
</tr>
<tr>
<td>2D</td>
<td>D3</td>
<td>12L</td>
<td>A5</td>
<td>2B</td>
<td>VCC</td>
<td>2A</td>
<td>VSS</td>
</tr>
<tr>
<td>1E</td>
<td>D2</td>
<td>12M</td>
<td>A4</td>
<td>12B</td>
<td>VCC</td>
<td>12A</td>
<td>VSS</td>
</tr>
<tr>
<td>2E</td>
<td>D1</td>
<td>11M</td>
<td>A3</td>
<td>1C</td>
<td>VCC</td>
<td>1B</td>
<td>VSS</td>
</tr>
<tr>
<td>1F</td>
<td>D0</td>
<td>10M</td>
<td>A2</td>
<td>2M</td>
<td>VCC</td>
<td>13B</td>
<td>VSS</td>
</tr>
<tr>
<td>9A</td>
<td>A23</td>
<td>1K</td>
<td>A1</td>
<td>3N</td>
<td>VCC</td>
<td>13M</td>
<td>VSS</td>
</tr>
<tr>
<td>10A</td>
<td>A22</td>
<td>2J</td>
<td>BLE#</td>
<td>5N</td>
<td>VCC</td>
<td>2N</td>
<td>VSS</td>
</tr>
<tr>
<td>10B</td>
<td>A21</td>
<td>2K</td>
<td>BHE#</td>
<td>10N</td>
<td>VCC</td>
<td>6N</td>
<td>VSS</td>
</tr>
<tr>
<td>12C</td>
<td>A20</td>
<td>4M</td>
<td>W/R#</td>
<td>1A</td>
<td>VCC</td>
<td>12N</td>
<td>VSS</td>
</tr>
<tr>
<td>13D</td>
<td>A19</td>
<td>3M</td>
<td>D/C#</td>
<td>3A</td>
<td>VCC</td>
<td>1L</td>
<td>N/C</td>
</tr>
</tbody>
</table>

Note: The table represents the pin assignments for the 88-Pin Grid Array.
The following table lists a brief description of each pin on the 80376. The following definitions are used in these descriptions:

- # The named signal is active LOW.
- I Input signal.
- O Output signal.
- I/O Input and Output signal.
- — No electrical connection.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Name and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK2</td>
<td>I</td>
<td>CLK2 provides the fundamental timing for the 80376. For additional information see Clock (page 33).</td>
</tr>
<tr>
<td>RESET</td>
<td>I</td>
<td>RESET suspends any operation in progress and places the 80376 in a known reset state. See Interrupt Signals (page 38) for additional information.</td>
</tr>
<tr>
<td>D_{15}−D_0</td>
<td>I/O</td>
<td>DATA BUS inputs data during memory, I/O and interrupt acknowledge read cycles and outputs data during memory and I/O write cycles. See Data Bus (page 34) for additional information.</td>
</tr>
<tr>
<td>A_{23}−A_1</td>
<td>O</td>
<td>ADDRESS BUS outputs physical memory or port I/O addresses. See Address Bus (page 34) for additional information.</td>
</tr>
<tr>
<td>W/R#</td>
<td>O</td>
<td>WRITE/READ is a bus cycle definition pin that distinguishes write cycles from read cycles. See Bus Cycle Definition Signals (page 35) for additional information.</td>
</tr>
<tr>
<td>D/C#</td>
<td>O</td>
<td>DATA/CONTROL is a bus cycle definition pin that distinguishes data cycles, either memory or I/O, from control cycles which are: interrupt acknowledge, halt, and instruction fetching. See Bus Cycle Definition Signals (page 35) for additional information.</td>
</tr>
<tr>
<td>M/IO#</td>
<td>O</td>
<td>MEMORY I/O is a bus cycle definition pin that distinguishes memory cycles from input/output cycles. See Bus Cycle Definition Signals (page 35) for additional information.</td>
</tr>
<tr>
<td>LOCK#</td>
<td>O</td>
<td>BUS LOCK is a bus cycle definition pin that indicates that other system bus masters are denied access to the system bus while it is active. See Bus Cycle Definition Signals (page 35) for additional information.</td>
</tr>
<tr>
<td>ADS#</td>
<td>O</td>
<td>ADDRESS STATUS indicates that a valid bus cycle definition and address (W/R#, D/C#, M/IO#, BHE#, BLE# and A_{23}−A_1) are being driven at the 80376 pins. See Bus Control Signals (page 35) for additional information.</td>
</tr>
<tr>
<td>NA#</td>
<td>I</td>
<td>NEXT ADDRESS is used to request address pipelining. See Bus Control Signals (page 35) for additional information.</td>
</tr>
<tr>
<td>READY#</td>
<td>I</td>
<td>BUS READY terminates the bus cycle. See Bus Control Signals (page 35) for additional information.</td>
</tr>
<tr>
<td>BHE#, BLE#</td>
<td>O</td>
<td>BYTE ENABLES indicate which data bytes of the data bus take part in a bus cycle. See Address Bus (page 34) for additional information.</td>
</tr>
<tr>
<td>HOLD</td>
<td>I</td>
<td>BUS HOLD REQUEST input allows another bus master to request control of the local bus. See Bus Arbitration Signals (page 36) for additional information.</td>
</tr>
</tbody>
</table>
Symbol | Type | Name and Function
--- | --- | ---
HLDA | O | BUS HOLD ACKNOWLEDGE output indicates that the 80376 has surrendered control of its local bus to another bus master. See Bus Arbitration Signals (page 36) for additional information.
INTR | I | INTERRUPT REQUEST is a maskable input that signals the 80376 to suspend execution of the current program and execute an interrupt acknowledge function. See Interrupt Signals (page 38) for additional information.
NMI | I | NON-MASKABLE INTERRUPT REQUEST is a non-maskable input that signals the 80376 to suspend execution of the current program and execute an interrupt acknowledge function. See Interrupt Signals (page 38) for additional information.
BUSY # | I | BUSY signals a busy condition from a processor extension. See Coprocessor Interface Signals (page 37) for additional information.
ERROR # | I | ERROR signals an error condition from a processor extension. See Coprocessor Interface Signals (page 37) for additional information.
PEREQ | I | PROCESSOR EXTENSION REQUEST indicates that the processor extension has data to be transferred by the 80376. See Coprocessor Interface Signals (page 37) for additional information.
N/C | — | NO CONNECT should always remain unconnected. Connection of a N/C pin may cause the processor to malfunction or be incompatible with future steppings of the 80376.
VCC | I | SYSTEM POWER provides the +5V nominal D.C. supply input.
VSS | I | SYSTEM GROUND provides 0V connection from which all inputs and outputs are measured.

2.0 ARCHITECTURE OVERVIEW

The 80376 supports the protection mechanisms needed by sophisticated multitasking embedded systems and real-time operating systems. The use of these protection mechanisms is completely optional. For embedded applications not needing protection, the 80376 can easily be configured to provide a 16 Mbyte physical address space.

Instruction pipelining, high bus bandwidth, and a very high performance ALU ensure short average instruction execution times and high system throughput. The 80376 is capable of execution at sustained rates of 2.5–3.0 million instructions per second.

The 80376 offers on-chip testability and debugging features. Four break point registers allow conditional or unconditional break point traps on code execution or data accesses for powerful debugging of even ROM based systems. Other testability features include self-test and tri-stating of output buffers during RESET.

The Intel 80376 embedded processor consists of a central processing unit, a memory management unit and a bus interface. The central processing unit consists of the execution unit and instruction unit. The execution unit contains the eight 32-bit general registers which are used for both address calculation and data operations and a 64-bit barrel shifter used to speed shift, rotate, multiply, and divide operations. The instruction unit decodes the instruction opcodes and stores them in the decoded instruction queue for immediate use by the execution unit.

The Memory Management Unit (MMU) consists of a segmentation and protection unit. Segmentation allows the managing of the logical address space by providing an extra addressing component, one that allows easy code and data relocatability, and efficient sharing.

The protection unit provides four levels of protection for isolating and protecting applications and the operating system from each other. The hardware enforced protection allows the design of systems with a high degree of integrity and simplifies debugging.

Finally, to facilitate high performance system hardware designs, the 80376 bus interface offers address pipelining and direct Byte Enable signals for each byte of the data bus.
2.1 Register Set

The 80376 has twenty-nine registers as shown in Figure 2.1. These registers are grouped into the following six categories:

<table>
<thead>
<tr>
<th>Category</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Purpose Registers</td>
<td>AH, AX, AL, EAX</td>
</tr>
<tr>
<td></td>
<td>BH, BX, BL, EBX</td>
</tr>
<tr>
<td></td>
<td>CH, CX, CL, ECX</td>
</tr>
<tr>
<td></td>
<td>DH, DK, DL, EDX</td>
</tr>
<tr>
<td></td>
<td>SI, EDI</td>
</tr>
<tr>
<td></td>
<td>DI, EDI</td>
</tr>
<tr>
<td></td>
<td>BP, EBX</td>
</tr>
<tr>
<td></td>
<td>SP, ESP</td>
</tr>
<tr>
<td>Segment Registers</td>
<td>CS, SS, DS, ES, FS</td>
</tr>
<tr>
<td></td>
<td>GS</td>
</tr>
<tr>
<td>Flags and Instruction Pointer</td>
<td>EFLAGS, EIP</td>
</tr>
<tr>
<td>Control Register</td>
<td>CR0</td>
</tr>
<tr>
<td>System Address Registers</td>
<td>GDTR, IDTR, LDTR</td>
</tr>
<tr>
<td></td>
<td>TR</td>
</tr>
<tr>
<td>Debug Registers</td>
<td>DR0, DR1, DR2, DR3, DR4, DR5, DR6, DR7</td>
</tr>
</tbody>
</table>

Figure 2.1. 80376 Base Architecture Registers
**General Registers:** The eight 32-bit general purpose registers are used to contain arithmetic and logical operands. Four of these (EAX, EBX, ECX and EDX) can be used either in their entirety as 32-bit registers, as 16-bit registers, or split into pairs of separate 8-bit registers.

**Segment Registers:** Six 16-bit special purpose registers select, at any given time, the segments of memory that are immediately addressable for code, stack, and data.

**Flags and Instruction Pointer Registers:** These two 32-bit special purpose registers in Figure 2.1 record or control certain aspects of the 80376 processor state. The EFLAGS register includes status and control bits that are used to reflect the outcome of many instructions and modify the semantics of some instructions. The Instruction Pointer, called EIP, is 32 bits wide. The Instruction Pointer controls instruction fetching and the processor automatically increments it after executing an instruction.

**Control Register:** The 32-bit control register, CR0, is used to control Coprocessor Emulation.

**System Address Registers:** These four special registers reference the tables or segments supported by the 80376/80386 protection model. These tables or segments are:

- GDTR (Global Descriptor Table Register)
- IDTR (Interrupt Descriptor Table Register)
- LDTR (Local Descriptor Table Register)
- TR (Task State Segment Register)

**Debug Registers:** The six programmer accessible debug registers provide on-chip support for debugging. The use of the debug registers is described in Section 2.11 **Debugging Support.**

**EFLAGS REGISTER**
The flag Register is a 32-bit register named EFLAGS. The defined bits and bit fields within EFLAGS, shown in Figure 2.2, control certain operations and indicate the status of the 80376 processor. The function of the flag bits is given in Table 2.1.

---

**Figure 2.2. Status and Control Register Bit Functions**

---

4-41
Table 2.1. Flag Definitions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CF</td>
<td>Carry Flag—Set on high-order bit carry or borrow; cleared otherwise.</td>
</tr>
<tr>
<td>1</td>
<td>PF</td>
<td>Parity Flag—Set if low-order 8 bits of result contain an even number of 1-bits; cleared otherwise.</td>
</tr>
<tr>
<td>2</td>
<td>AF</td>
<td>Auxiliary Carry Flag—Set on carry from or borrow to the low order four bits of AL; cleared otherwise.</td>
</tr>
<tr>
<td>3</td>
<td>ZF</td>
<td>Zero Flag—Set if result is zero; cleared otherwise.</td>
</tr>
<tr>
<td>4</td>
<td>SF</td>
<td>Sign Flag—Set equal to high-order bit of result (0 if positive, 1 if negative).</td>
</tr>
<tr>
<td>5</td>
<td>TF</td>
<td>Single Step Flag—Once set, a single step interrupt occurs after the next instruction executes. TF is cleared by the single step interrupt.</td>
</tr>
<tr>
<td>6</td>
<td>IF</td>
<td>Interrupt-Enable Flag—When set, external interrupts signaled on the INTR pin will cause the CPU to transfer control to an interrupt vector specified location.</td>
</tr>
<tr>
<td>7</td>
<td>DF</td>
<td>Direction Flag—Causes string instructions to auto-increment (default) the appropriate index registers when cleared. Setting DF causes auto-decrement.</td>
</tr>
<tr>
<td>8</td>
<td>OF</td>
<td>Overflow Flag—Set if the operation resulted in a carry/borrow into the sign bit (high-order bit) of the result but did not result in a carry/borrow out of the high-order bit or vice-versa.</td>
</tr>
<tr>
<td>9, 10</td>
<td>IOPL</td>
<td>I/O Privilege Level—Indicates the maximum CPL permitted to execute I/O instructions without generating an exception 13 fault or consulting the I/O permission bit map. It also indicates the maximum CPL value allowing alteration of the IF bit.</td>
</tr>
<tr>
<td>11</td>
<td>TS</td>
<td>Task Switched—Indicates that the execution of the current task is nested within another task (see Task Switching).</td>
</tr>
<tr>
<td>12, 13</td>
<td>RF</td>
<td>Resume Flag—Used in conjunction with debug register breakpoints. It is checked at instruction boundaries before breakpoint processing. If set, any debug fault is ignored on the next instruction. It is reset at the successful completion of any instruction except IRET, POPF, and those instructions causing task switches.</td>
</tr>
</tbody>
</table>

CONTROL REGISTER

The 80376 has a 32-bit control register called CR0 that is used to control coprocessor emulation. This register is shown in Figures 2.1 and 2.2. The defined CR0 bits are described in Table 2.2.

Table 2.2. CR0 Definitions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MP</td>
<td>Monitor Coprocessor Extension—Allows WAIT instructions to cause a processor extension not present exception (number 7).</td>
</tr>
<tr>
<td>2</td>
<td>EM</td>
<td>Emulate Processor Extension—When set, this bit causes a processor extension not present exception (number 7) on ESC instructions to allow processor extension emulation.</td>
</tr>
<tr>
<td>3</td>
<td>TS</td>
<td>Task Switched—When set, this bit indicates the next instruction using a processor extension will cause exception 7, allowing software to test whether the current processor extension context belongs to the current task (see Task Switching).</td>
</tr>
</tbody>
</table>
2.2 Instruction Set

The instruction set is divided into nine categories of operations:

- Data Transfer
- Arithmetic
- Shift/Rotate
- String Manipulation
- Bit Manipulation
- Control Transfer
- High Level Language Support
- Operating System Support
- Processor Control

These 80376 processor instructions are listed in Table 8.1 **80376 Instruction Set and Clock Count Summary**.

All 80376 processor instructions operate on either 0, 1, 2 or 3 operands; an operand resides in a register, in the instruction itself, or in memory. Most zero operand instructions (e.g. CLI, STI) take only one byte. One operand instructions generally are two bytes long. The average instruction is 3.2 bytes long. Since the 80376 has a 16-byte prefetch instruction queue an average of 5 instructions can be prefetched. The use of two operands permits the following types of common instructions:

- Register to Register
- Memory to Register
- Immediate to Register
- Memory to Memory
- Register to Memory
- Immediate to Memory

The operands are either 8-, 16- or 32-bit long.

2.3 Memory Organization

Memory on the 80376 is divided into 8-bit quantities (bytes), 16-bit quantities (words), and 32-bit quantities (dwords). Words are stored in two consecutive bytes in memory with the low-order byte at the lowest address. Dwords are stored in four consecutive bytes in memory with the low-order byte at the lowest address. The address of a word or Dword is the byte address of the low-order byte.

In addition to these basic data types the 80376 processor supports segments. Memory can be divided up into one or more variable length segments, which can be shared between programs.

ADDRESS SPACES

The 80376 has three types of address spaces: **logical**, **linear**, and **physical**. A **logical** address (also known as a **virtual** address) consists of a selector and an offset. A selector is the contents of a segment register. An offset is formed by summing all of the addressing components (BASE, INDEX, and DISPLACEMENT), discussed in Section 2.4 **Addressing Modes**, into an effective address.

Every selector has a **logical base** address associated with it that can be up to 32 bits in length. This 32-bit logical base address is added to either a 32-bit offset address or a 16-bit offset address (by using the **address length prefix**) to form a final 32-bit **linear** address. This final linear address is then truncated so that only the lower 24 bits of this address are used to address the 16 Mbytes physical memory address space. The **logical base** address is stored in one of two operating system tables (i.e. the Local Descriptor Table or Global Descriptor Table).

Figure 2.3 shows the relationship between the various address spaces.
SEGMENT REGISTER USAGE

The main data structure used to organize memory is the segment. On the 80376, segments are variable sized blocks of linear addresses which have certain attributes associated with them. There are two main types of segments, code and data. The simplest use of segments is to have one code and data segment. Each segment is 16 Mbytes in size overlapping each other. This allows code and data to be directly addressed by the same offset.

In order to provide compact instruction encoding and increase processor performance, instructions do not need to explicitly specify which segment register is used. The segment register is automatically chosen according to the rules of Table 2.3 (Segment Register Selection Rules). In general, data references use the selector contained in the DS register, stack references use the SS register and instruction fetches use the CS register. The contents of the Instruction Pointer provide the offset. Special segment override prefixes allow the explicit use of a given segment register, and override the implicit rules listed in Table 2.3. The override prefixes also allow the use of the ES, FS and GS segment registers.

There are no restrictions regarding the overlapping of the base addresses of any segments. Thus, all 6 segments could have the base address set to zero. Further details of segmentation are discussed in Section 3.0 Architecture.
Table 2.3. Segment Register Selection Rules

<table>
<thead>
<tr>
<th>Type of Memory Reference</th>
<th>Implied (Default) Segment Use</th>
<th>Segment Override Prefixes Possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Fetch</td>
<td>CS</td>
<td>None</td>
</tr>
<tr>
<td>Destination of PUSH, PUSHF, INT, CALL, PUSHA Instructions</td>
<td>SS</td>
<td>None</td>
</tr>
<tr>
<td>Source of POP, POPA, POPF, IRET, RET Instructions</td>
<td>SS</td>
<td>None</td>
</tr>
<tr>
<td>Destination of STOS, MOVS, REP STOS, REP MOVS Instructions (DI is Base Register)</td>
<td>ES</td>
<td>None</td>
</tr>
<tr>
<td>Other Data References, with Effective Address Using Base Register of:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[EAX]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[EBX]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[ECX]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[EDX]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[ESI]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[EDI]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[EBP]</td>
<td>SS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[ESP]</td>
<td>SS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
</tbody>
</table>

2.4 Addressing Modes

The 80376 provides a total of 8 addressing modes for instructions to specify operands. The addressing modes are optimized to allow the efficient execution of high level languages such as C and FORTRAN, and they cover the vast majority of data references needed by high-level languages.

Two of the addressing modes provide for instructions that operate on register or immediate operands:

Register Operand Mode: The operand is located in one of the 8-, 16- or 32-bit general registers.

Immediate Operand Mode: The operand is included in the instruction as part of the opcode.

The remaining 6 modes provide a mechanism for specifying the effective address of an operand. The linear address consists of two components: the segment base address and an effective address. The effective address is calculated by summing any combination of the following three address elements (see Figure 2.3):

DISPLACEMENT: an 8-, 16- or 32-bit immediate value following the instruction.

BASE: The contents of any general purpose register. The base registers are generally used by compilers to point to the start of the local variable area. Note that if the Address Length Prefix is used, only BX and BP can be used as a BASE register.

INDEX: The contents of any general purpose register except for ESP. The index registers are used to access the elements of an array, or a string of characters. The index register's value can be multiplied by a scale factor, either 1, 2, 4 or 8. The scaled index is especially useful for accessing arrays or structures. Note that if the Address Length Prefix is used, no Scaling is available and only the registers SI and DI can be used to INDEX.
Combinations of these 3 components make up the 6 additional addressing modes. There is no performance penalty for using any of these addressing combinations, since the effective address calculation is pipelined with the execution of other instructions. The one exception is the simultaneous use of BASE and INDEX components which requires one additional clock.

As shown in Figure 2.4, the effective address (EA) of an operand is calculated according to the following formula:

\[ EA = BASE_{Register} + (INDEX_{Register} \times scaling) + DISPLACEMENT \]

1. Direct Mode: The operand's offset is contained as part of the instruction as an 8-, 16- or 32-bit DISPLACEMENT.

2. Register Indirect Mode: A BASE register contains the address of the operand.

3. Based Mode: A BASE register's contents is added to a DISPLACEMENT to form the operand's offset.

4. Scaled Index Mode: An INDEX register's contents is multiplied by a SCALING factor which is added to a DISPLACEMENT to form the operand's offset.

5. Based Scaled Index Mode: The contents of an INDEX register is multiplied by a SCALING factor and the result is added to the contents of a BASE register to obtain the operand's offset.

6. Based Scaled Index Mode with Displacement: The contents of an INDEX register are multiplied by a SCALING factor, and the result is added to the contents of a BASE register and a DISPLACEMENT to form the operand's offset.

Figure 2.4. Addressing Mode Calculations
GENERATING 16-BIT ADDRESSES

The 80376 executes code with a default length for operands and addresses of 32 bits. The 80376 is also able to execute operands and addresses of 16 bits. This is specified through the use of override prefixes. Two prefixes, the Operand Length Prefix and the Address Length Prefix, override the default 32-bit length on an individual instruction basis. These prefixes are automatically added by assemblers. The Operand Length and Address Length Prefixes can be applied separately or in combination to any instruction.

The 80376 normally executes 32-bit code and uses either 8- or 32-bit displacements, and any register can be used as based or index registers. When executing 16-bit code (by prefix overrides), the displacements are either 8 or 16 bits, and the base and index register conform to the 16-bit model. Table 2.4 illustrates the differences.

Table 2.4. BASE and INDEX Registers for 16- and 32-Bit Addresses

<table>
<thead>
<tr>
<th></th>
<th>16-Bit Addressing</th>
<th>32-Bit Addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE REGISTER</td>
<td>BX, BP</td>
<td>Any 32-Bit GP Register</td>
</tr>
<tr>
<td>INDEX REGISTER</td>
<td>SI, DI</td>
<td>Any 32-Bit GP Register except ESP</td>
</tr>
<tr>
<td>SCALE FACTOR</td>
<td>None</td>
<td>1, 2, 4, 8</td>
</tr>
<tr>
<td>DISPLACEMENT</td>
<td>0, 8, 16 Bits</td>
<td>0, 8, 32 Bits</td>
</tr>
</tbody>
</table>

2.5 Data Types

The 80376 supports all of the data types commonly used in high level languages:

- **Bit:** A single bit quantity.
- **Bit Field:** A group of up to 32 contiguous bits, which spans a maximum of four bytes.
- **Bit String:** A set of contiguous bits, on the 80376 bit strings can be up to 16 Mbits long.
- **Byte:** A signed 8-bit quantity.
- **Unsigned Byte:** An unsigned 8-bit quantity.
- **Integer (Word):** A signed 16-bit quantity.
- **Long Integer (Double Word):** A signed 32-bit quantity. All operations assume a 2's complement representation.
- **Unsigned Integer (Word):** An unsigned 16-bit quantity.
- **Unsigned Long Integer (Double Word):** An unsigned 32-bit quantity.
- **Signed Quad Word:** A signed 64-bit quantity.
- **Unsigned Quad Word:** An unsigned 64-bit quantity.
- **Pointer:** A 16- or 32-bit offset only quantity which indirectly references another memory location.
- **Long Pointer:** A full pointer which consists of a 16-bit segment selector and either a 16- or 32-bit offset.
- **Char:** A byte representation of an ASCII Alphanumeric or control character.
- **String:** A contiguous sequence of bytes, words or dwords. A string may contain between 1 byte and 16 Mbytes.
- **BCD:** A byte (unpacked) representation of decimal digits 0–9.
- **Packed BCD:** A byte (packed) representation of two decimal digits 0–9 storing one digit in each nibble.
When the 80376 is coupled with a numerics Coprocessor such as the 80387SX then the following common Floating Point types are supported.

Floating Point: A signed 32-, 64- or 80-bit real number representation. Floating point numbers are supported by the 80387SX numerics coprocessor.

Figure 2.5 illustrates the data types supported by the 80376 processor and the 80387SX coprocessor.

**Figure 2.5. 80376 Supported Data Types**
2.6 I/O Space

The 80376 has two distinct physical address spaces: physical memory and I/O. Generally, peripherals are placed in I/O space although the 80376 also supports memory-mapped peripherals. The I/O space consists of 64 Kbytes which can be divided into 64K 8-bit ports, 32K 16-bit ports, or any combination of ports which add to no more than 64 Kbytes. The M/IO# pin acts as an additional address line, thus allowing the system designer to easily determine which address space the processor is accessing. Note that the I/O address refers to a physical address.

The I/O ports are accessed by the IN and OUT instructions, with the port address supplied as an immediate 8-bit constant in the instruction or in the DX register. All 8-bit and 16-bit port addresses are zero extended on the upper address lines. The I/O instructions cause the M/IO# pin to be driven LOW. I/O port addresses 00F8H through 00FFH are reserved for use by Intel.

2.7 Interrupts and Exceptions

Interrupts and exceptions alter the normal program flow in order to handle external events, report errors or exceptional conditions. The difference between interrupts and exceptions is that interrupts are used to handle asynchronous external events while exceptions handle instruction faults. Although a program can generate a software interrupt via an INT N instruction, the processor treats software interrupts as exceptions.

Hardware interrupts occur as the result of an external event and are classified into two types: maskable or non-maskable. Interrupts are serviced after the execution of the current instruction. After the interrupt handler is finished servicing the interrupt, execution proceeds with the instruction immediately after the interrupted instruction.

Exceptions are classified as faults, traps, or aborts depending on the way they are reported, and whether or not restart of the instruction causing the exception is supported. Faults are exceptions that are detected and serviced before the execution of the faulting instruction. Traps are exceptions that are reported immediately after the execution of the instruction which caused the problem. Aborts are exceptions which do not permit the precise location of the instruction causing the exception to be determined. Thus, when an interrupt service routine has been completed, execution proceeds from the instruction immediately following the interrupted instruction. On the other hand the return address from an exception/fault routine will always point at the instruction causing the exception and include any leading instruction prefixes. Table 2.5 summarizes the possible interrupts for the 80376 and shows where the return address points to.

The 80376 has the ability to handle up to 256 different interrupts/exceptions. In order to service the interrupts, a table with up to 256 interrupt vectors must be defined. The interrupt vectors are simply pointers to the appropriate interrupt service routine. The interrupt vectors are 8-byte quantities, which are put in an Interrupt Descriptor Table. Of the 256 possible interrupts, 32 are reserved for use by Intel and the remaining 224 are free to be used by the system designer.

INTERRUPT PROCESSING

When an interrupt occurs the following actions happen. First, the current program address and the Flags are saved on the stack to allow resumption of the interrupted program. Next, an 8-bit vector is supplied to the 80376 which identifies the appropriate entry in the interrupt table. The table contains either an Interrupt Gate, a Trap Gate or a Task Gate that will point to an interrupt procedure or task. The user supplied interrupt service routine is executed. Finally, when an IRET instruction is executed the old processor state is restored and program execution resumes at the appropriate instruction.

The 8-bit interrupt vector is supplied to the 80376 in several different ways: exceptions supply the interrupt vector internally; software INT instructions contain or imply the vector; maskable hardware interrupts supply the 8-bit vector via the interrupt acknowledge bus sequence. Non-Maskable hardware interrupts are assigned to interrupt vector 2.

Maskable Interrupt

Maskable interrupts are the most common way to respond to asynchronous external hardware events. A hardware interrupt occurs when the INTR is pulled HIGH and the Interrupt Flag bit (IF) is enabled. The processor only responds to interrupts between instructions (string instructions have an “interrupt window” between memory moves which allows interrupts during long string moves). When an interrupt occurs the processor reads an 8-bit vector supplied by the hardware which identifies the source of the interrupt (one of 224 user defined interrupts).
<table>
<thead>
<tr>
<th>Function</th>
<th>Interrupt Number</th>
<th>Instruction Which Can Cause Exception</th>
<th>Return Address Points to Faulting Instruction</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide Error</td>
<td>0</td>
<td>DIV, IDIV</td>
<td>Yes</td>
<td>FAULT</td>
</tr>
<tr>
<td>Debug Exception</td>
<td>1</td>
<td>Any Instruction</td>
<td>Yes</td>
<td>TRAP*</td>
</tr>
<tr>
<td>NMI Interrupt</td>
<td>2</td>
<td>INT 2 or NMI</td>
<td>No</td>
<td>NMI</td>
</tr>
<tr>
<td>One-Byte Interrupt</td>
<td>3</td>
<td>INT</td>
<td>No</td>
<td>TRAP</td>
</tr>
<tr>
<td>Interrupt on Overflow</td>
<td>4</td>
<td>INTO</td>
<td>No</td>
<td>TRAP</td>
</tr>
<tr>
<td>Array Bounds Check</td>
<td>5</td>
<td>BOUND</td>
<td>Yes</td>
<td>FAULT</td>
</tr>
<tr>
<td>Invalid OP-Code Check</td>
<td>6</td>
<td>Any Illegal Instruction</td>
<td>Yes</td>
<td>FAULT</td>
</tr>
<tr>
<td>Device Not Available</td>
<td>7</td>
<td>ESC, WAIT</td>
<td>Yes</td>
<td>FAULT</td>
</tr>
<tr>
<td>Double Fault</td>
<td>8</td>
<td>Any Instruction That Can Generate an Exception</td>
<td></td>
<td>ABORT</td>
</tr>
<tr>
<td>Coprocessor Segment Overrun</td>
<td>9</td>
<td>ESC</td>
<td>No</td>
<td>ABORT</td>
</tr>
<tr>
<td>Invalid TSS</td>
<td>10</td>
<td>JMP, CALL, IRET, INT</td>
<td>Yes</td>
<td>FAULT</td>
</tr>
<tr>
<td>Segment Not Present</td>
<td>11</td>
<td>Segment Register Instructions</td>
<td>Yes</td>
<td>FAULT</td>
</tr>
<tr>
<td>Stack Fault</td>
<td>12</td>
<td>Stack References</td>
<td>Yes</td>
<td>FAULT</td>
</tr>
<tr>
<td>General Protection Fault</td>
<td>13</td>
<td>Any Memory Reference</td>
<td>Yes</td>
<td>FAULT</td>
</tr>
<tr>
<td>Intel Reserved</td>
<td>14–15</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Coprocessor Error</td>
<td>16</td>
<td>ESC, WAIT</td>
<td>Yes</td>
<td>FAULT</td>
</tr>
<tr>
<td>Intel Reserved</td>
<td>17–32</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Two-Byte Interrupt</td>
<td>0–255</td>
<td>INT n</td>
<td>No</td>
<td>TRAP</td>
</tr>
</tbody>
</table>

*Some debug exceptions may report both traps on the previous instruction, and faults on the next instruction.

Interrupts through Interrupt Gates automatically reset IF, disabling INTR requests. Interrupts through Trap Gates leave the state of the IF bit unchanged. Interrupts through a Task Gate change the IF bit according to the image of the EFLAGS register in the task’s Task State Segment (TSS). When an IRET instruction is executed, the original state of the IF bit is restored.

**Non-Maskable Interrupt**

Non-maskable interrupts provide a method of servicing very high priority interrupts. When the NMI input is pulled HIGH it causes an interrupt with an internally supplied vector value of 2. Unlike a normal hardware interrupt no interrupt acknowledgement sequence is performed for an NMI.

While executing the NMI servicing procedure, the 80376 will not service any further NMI request, or INT requests, until an interrupt return (IRET) instruction is executed or the processor is reset. If NMI occurs while currently servicing an NMI, its presence will be saved for servicing after executing the first IRET instruction. The disabling of INTR requests depends on the gate in IDT location 2.

**Software Interrupts**

A third type of interrupt/exception for the 80376 is the software interrupt. An INT n instruction causes the processor to execute the interrupt service routine pointed to by the nth vector in the interrupt table.

A special case of the two byte software interrupt INT 3 is the one byte INT 3, or breakpoint interrupt. By inserting this one byte instruction in a program, the user can set breakpoints in his program as a debugging tool.
A final type of software interrupt, is the single step interrupt. It is discussed in Single-Step Trap (page 22).

**INTERRUPT AND EXCEPTION PRIORITIES**

Interrupts are externally-generated events. Maskable Interrupts (on the INTR input) and Non-Maskable Interrupts (on the NMI input) are recognized at instruction boundaries. When NMI and maskable INTR are **both** recognized at the **same** instruction boundary, the 80376 invokes the NMI service routine first. If, after the NMI service routine has been invoked, maskable interrupts are still enabled, then the 80376 will invoke the appropriate interrupt service routine.

As the 80376 executes instructions, it follows a consistent cycle in checking for exceptions, as shown in Table 2.6. This cycle is repeated as each instruction is executed, and occurs in parallel with instruction decoding and execution.

**INSTRUCTION RESTART**

The 80376 fully supports restarting all instructions after faults. If an exception is detected in the instruction to be executed (exception categories 4 through 9 in Table 2.6), the 80376 device invokes the appropriate exception service routine. The 80376 is in a state that permits restart of the instruction.

**DOUBLE FAULT**

A Double fault (exception 8) results when the processor attempts to invoke an exception service routine for the segment exceptions (10, 11, 12 or 13), but in the process of doing so, detects an exception.

**2.8 Reset and Initialization**

When the processor is Reset the registers have the values shown in Table 2.7. The 80376 will then start executing instructions near the top of physical memory, at location OFFFFFOH. A short JMP should be executed within the segment defined for power-up (see Table 2.7). The GDT should then be initialized for a start-up data and code segment followed by a far JMP that will load the segment descriptor cache with the new descriptor values. The IDT table, after reset, is located at physical address 0H, with a limit of 256 entries.

RESET forces the 80376 to terminate all execution and local bus activity. No instruction execution or bus activity will occur as long as Reset is active. Between 350 and 450 CLK2 periods after Reset becomes inactive, the 80376 will start executing instructions at the top of physical memory.

---

**Table 2.6. Sequence of Exception Checking**

Consider the case of the 80376 having just completed an instruction. It then performs the following checks before reaching the point where the next instruction is completed:

1. Check for Exception 1 Traps from the instruction just completed (single-step via Trap Flag, or Data Breakpoints set in the Debug Registers).
2. Check for external NMI and INTR.
3. Check for Exception 1 Faults in the next instruction (Instruction Execution Breakpoint set in the Debug Registers for the next instruction).
4. Check for Segmentation Faults that prevented fetching the entire next instruction (exceptions 11 or 13).
5. Check for Faults decoding the next instruction (exception 6 if illegal opcode; or exception 13 if instruction is longer than 15 bytes, or privilege violation (i.e. not at IOPL or at CPL = 0).
6. If WAIT opcode, check if $TS = 1$ and $MP = 1$ (exception 7 if both are 1).
7. If ESCape opcode for numeric coprocessor, check if $EM = 1$ or $TS = 1$ (exception 7 if either are 1).
8. If WAIT opcode or ESCape opcode for numeric coprocessor, check ERROR# input signal (exception 16 if ERROR# input is asserted).
9. Check for Segmentation Faults that prevent transferring the entire memory quantity (exceptions 11, 12, 13).
Table 2.7. Register Values after Reset

<table>
<thead>
<tr>
<th>Flag Word (EFLAGS)</th>
<th>uuuu0002H</th>
<th>(Note 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Status Word (CR0)</td>
<td>uuuuuu1H</td>
<td>(Note 2)</td>
</tr>
<tr>
<td>Instruction Pointer (EIP)</td>
<td>0000FFFF0H</td>
<td></td>
</tr>
<tr>
<td>Code Segment (CS)</td>
<td>F000H</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>Data Segment (DS)</td>
<td>0000H</td>
<td>(Note 4)</td>
</tr>
<tr>
<td>Stack Segment (SS)</td>
<td>0000H</td>
<td></td>
</tr>
<tr>
<td>Extra Segment (ES)</td>
<td>0000H</td>
<td>(Note 4)</td>
</tr>
<tr>
<td>Extra Segment (FS)</td>
<td>0000H</td>
<td></td>
</tr>
<tr>
<td>Extra Segment (GS)</td>
<td>0000H</td>
<td></td>
</tr>
<tr>
<td>EAX Register</td>
<td>0000H</td>
<td>(Note 5)</td>
</tr>
<tr>
<td>EDX Register</td>
<td>Component and Stepping ID</td>
<td>(Note 6)</td>
</tr>
<tr>
<td>All Other Registers</td>
<td>Undefined</td>
<td>(Note 7)</td>
</tr>
</tbody>
</table>

NOTES:
1. EFLAG Register. The upper 14 bits of the EFLAGS register are undefined, all defined flag bits are zero.
2. CR0: The defined 4 bits in the CR0 is equal to 1H.
3. The Code Segment Register (CS) will have its Base Address set to 0FFFF0000H and Limit set to 0FFFFH.
4. The Data and Extra Segment Registers (DS and ES) will have their Base Address set to 000000000H and Limit set to 0FFFFH.
5. If self-test is selected, the EAX should contain a 0 value. If a value of 0 is not found the self-test has detected a flaw in the part.
6. EDX register always holds component and stepping identifier.
7. All unidentified bits are Intel Reserved and should not be used.

2.9 Initialization

Because the 80376 processor starts executing in protected mode, certain precautions need be taken during initialization. Before any far jumps can take place the GDT and/or LDT tables need to be setup and their respective registers loaded. Before interrupts can be initialized the IDT table must be setup and the IDTR must be loaded. The example code is shown below:

```assembly
; ***************************************************************
; This is an example of startup code to put either an 80376, 80386SX or 80386 into flat mode. All of memory is treated as simple linear RAM. There are no interrupt routines. The Builder creates the GDT-alias and IDT-alias and places them, by default, in GDT[1] and GDT[2]. Other entries in the GDT are specified in the Build file. After initialization it jumps to a C startup routine. To use this template, change this jmp address to that of your code, or make the label of your code "c_start".
; This code was assembled and built using version 1.2 of the Intel RLL utilities and Intel 386ASM assembler.

*** This code was tested ***
```

4-52
**NAME FLAT** ; name of the object module

**EXTRN** c_startup:near ; this is the label jumped to after init

pe_flag equ 1
data_selc equ 20h ; assume code is GDT[3], data GDT[4]

**INIT_CODE SEGMENT** PUBLIC USE32 ; Segment base at 0ffffff80h

PUBLIC GDT_DESC

gdt_desc dq ?

PUBLIC START

start:
   cld ; clear direction flag
   smsw bx ; check for processor (80376) at reset
   test bl,1 ; use SMSW rather than MOV for speed
   jnz pestart
realstart ; is an 80386 and in real mode
   db 66h ; force the next operand into 32-bit mode.
   mov eax,offset gdt_desc ; move address of the GDT descriptor into eax
   xor ebx,ebx ; clear ebx
   mov bh,ah ; load 8 bits of address into bh
   move bl,al ; load 8 bits of address into bl
   db 67h
   db 66h ; use the 32-bit form of LGDT to load
   lgdt cs:[ebx] ; the 32-bits of address into the GDTR
   smsw ax ; go into protected mode (set PE bit)
   or al,pe_flag
   lmsw ax
   jmp next ; flush prefetch queue
pestart:
   mov ebx,offset gdt_desc
   xor eax,eax
   mov ax,bx ; lower portion of address only
   lgdt cs:[eax]
   xor ebx,ebx ; initialize data selectors
   mov bl,data_selc ; GDT[3]
   mov ds,bx
   mov ss,bx
   mov es,bx
   mov fs,bx
   mov gs,bx
   jmp pejump
next:
   xor ebx,ebx ; initialize data selectors
   mov bl,data_selc ; GDT[3]
   mov ds,bx
   mov ss,bx
   mov es,bx
   mov fs,bx
   mov gs,bx
   db 66h ; for the 80386, need to make a 32-bit jump
pejump:
   jmp far ptr c_startup ; but the 80376 is already 32-bit.
   org 70h
   jmp short start

**INIT_CODE ENDS**

END
This code should be linked into your application for boot loadable code. The following build file illustrates how this is accomplished.

**FLAT**; -- build program id

**SEGMENT**
*segments (dpl=0), -- Give all user segments a DPL of 0.
    _phantom_code_ (dpl=0), -- These two segments are created by
    _phantom_data_ (dpl=0), -- the builder when the FLAT control is used.
    init_code (base=0xffff800h); -- Put startup code at the reset vector area.

**GATE**
g13 (entry=13, dpl=0, trap), -- trap gate disables interrupts
i32 (entry=32, dpl=0, interrupt), -- interrupt gates doesn't

**TABLE**
-- create GDT

**GDT** (LOCATION = GDT_DESC, -- In a buffer starting at GDT_DESC,
    ENTRY' = (3:_phantom_code_,
        4:_phantom_data_,
        5:code32,
        6:data,
        7:init_code)
    );

**TASK**

**MAIN_TASK**

    (DPL = 0, -- Task privilege level is 0.
        DATA = DATA, -- Points to a segment that
            -- indicates initial DS value.
        CODE = main, -- Entry point is main, which
            -- must be a public id.
        STACKS = (DATA), -- Segment id points to stack
            -- segment. Sets the initial SS:ESP.
        NO INTENABLED, -- Disable interrupts.
        PRESENT = (ENTRY = (3:_phantom_code_,
        4:_phantom_data_,
        5:code32,
        6:data,
        7:init_code)
        ));

**MEMORY**

    (RANGE = (EPROM = ROM(0xffff800h..0xfffff),
        DRAM = RAM(0..0xffff)),
        ALLOCATE = (EPROM = (MAIN_TASK)));

**END**

asm386 flatsim.a38 debug
asm386 application.a38 debug
bnd386 application.obj,flatsim.obj nolo debug oj (application.bnd)
bld386 application.bnd bf (flatsim.bld) bl flat

Commands to assemble and build a boot-loadable application named “application.a38”. The initialization code is called “flatsim.a38”, and build file is called “application.bld”.
2.10 Self-Test

The 80376, like the 80386, has the capability to perform a self-test. The self-test checks the function of all of the Control ROM and most of the non-random logic of the part. Approximately one-half of the 80376 can be tested during self-test.

Self-Test is initiated on the 80376 when the RESET pin transitions from HIGH to LOW, and the BUSY # pin is LOW. The self-test takes about 220 clocks, or approximately 33 ms with a 16 MHz 80376 processor. At the completion of self-test the processor performs reset and begins normal operation. The part has successfully passed self-test if the contents of the EAX register is zero. If the EAX register is not zero then the self-test has detected a flaw in the part. If self-test is not selected after reset, EAX may be non-zero after reset.

2.11 Debugging Support

The 80376 provides several features which simplify the debugging process. The three categories of on-chip debugging aids are:
1. The code execution breakpoint opcode (OCCh).
2. The single-step capability provided by the TF bit in the flag register, and
3. The code and data breakpoint capability provided by the Debug Registers DR0–3, DR6, and DR7.

**BREAKPOINT INSTRUCTION**

A single-byte software interrupt (Int 3) breakpoint instruction is available for use by software debuggers. The breakpoint opcode is OCCh, and generates an exception 3 trap when executed.
SINGLE-STEP TRAP

If the single-step flag (TF, bit 8) in the EFLAG register is found to be set at the end of an instruction, a single-step exception occurs. The single-step exception is auto vectored to exception number 1.

The Debug Registers are an advanced debugging feature of the 80376. They allow data access breakpoints as well as code execution breakpoints. Since the breakpoints are indicated by on-chip registers, an instruction execution breakpoint can be placed in ROM code or in code shared by several tasks, neither of which can be supported by the INT 3 breakpoint opcode.

The 80376 contains six Debug Registers, consisting of four breakpoint address registers and two breakpoint control registers. Initially after reset, breakpoints are in the disabled state; therefore, no breakpoints will occur unless the debug registers are programmed. Breakpoints set up in the Debug Registers are auto-vectored to exception 1. Figure 2.6 shows the breakpoint status and control registers.

3.0 ARCHITECTURE

The Intel 80376 Embedded Processor has a physical address space of 16 Mbytes (2^24 bytes) and allows the running of virtual memory programs of almost unlimited size (16 Kbytes × 16 Mbytes or 256 Gbytes (2^38 bytes)). In addition the 80376 provides a sophisticated memory management and a hardware-assisted protection mechanism.

3.1 Addressing Mechanism

The 80376 uses two components to form the logical address, a 16-bit selector which determines the linear base address of a segment, and a 32-bit effective address. The selector is used to specify an index into an operating system defined table (see Figure 3.1). The table contains the 32-bit base address of a given segment. The linear address is formed by adding the base address obtained from the table to the 32-bit effective address. This value is truncated to 24 bits to form the physical address, which is then placed on the address bus.

![Figure 3.1. Address Calculation](image-url)

**Figure 3.1. Address Calculation**
3.2 Segmentation

Segmentation is one method of memory management and provides the basis for protection in the 80376. Segments are used to encapsulate regions of memory which have common attributes. For example, all of the code of a given program could be contained in a segment, or an operating system table may reside in a segment. All information about each segment, is stored in an 8-byte data structure called a descriptor. All of the descriptors in a system are contained in tables recognized by hardware.

TERMINOLOGY

The following terms are used throughout the discussion of descriptors, privilege levels and protection:

PL: Privilege Level—One of the four hierarchical privilege levels. Level 0 is the most privileged level and level 3 is the least privileged.

RPL: Requestor Privilege Level—The privilege level of the original supplier of the selector. RPL is determined by the least two significant bits of a selector.

DPL: Descriptor Privilege Level—This is the least privileged level at which a task may access that descriptor (and the segment associated with that descriptor). Descriptor Privilege Level is determined by bits 6:5 in the Access Right Byte of a descriptor.

CPL: Current Privilege Level—The privilege level at which a task is currently executing, which equals the privilege level of the code segment being executed. CPL can also be determined by examining the lowest 2 bits of the CS register, except for conforming code segments.

EPL: Effective Privilege Level—The effective privilege level is the least privileged of the RPL and the DPL. EPL is the numerical maximum of RPL and DPL.

Task: One instance of the execution of a program. Tasks are also referred to as processes.

DESCRIPTOR TABLES

The descriptor tables define all of the segments which are used in an 80376 system. There are three types of tables on the 80376 which hold descriptors: the Global Descriptor Table, Local Descriptor Table, and the Interrupt Descriptor Table. All of the tables are variable length memory arrays, they can range in size between 8 bytes and 64 Kbytes. Each table can hold up to 8192 8-byte descriptors. The upper 13 bits of a selector are used as an index into the descriptor table. The tables have registers associated with them which hold the 32-bit linear base address, and the 16-bit limit of each table.

Each of the tables have a register associated with it: GDTR, LDTR and IDTR; see Figure 3.2. The LGDT, LLDT and LIDT instructions load the base and limit of the Global, Local and Interrupt Descriptor Tables into the appropriate register. The SGDT, SLDT and SIDT store these base and limit values. These are privileged instructions.

Global Descriptor Table

The Global Descriptor Table (GDT) contains descriptors which are possibly available to all of the tasks in a system. The GDT can contain any type of segment descriptor except for interrupt and trap descriptors. Every 80376 system contains a GDT. A simple 80376 system contains only 2 entries in the GDT; a code and a data descriptor.

The first slot of the Global Descriptor Table corresponds to the null selector and is not used. The null selector defines a null pointer value.

Local Descriptor Table

LDTs contain descriptors which are associated with a given task. Generally, operating systems are designed so that each task has a separate LDT. The LDT may contain only code, data, stack, task gate, and call gate descriptors. LDTs provide a mechanism for isolating a given task’s code and data segments from the rest of the operating system, while the GDT contains descriptors for segments which are common to all tasks. A segment cannot be accessed by a task if its segment descriptor does not exist in either the current LDT or the GDT. This pro-
provides both isolation and protection for a task's segments, while still allowing global data to be shared among tasks.

Unlike the 6-byte GDT or IDT registers which contain a base address and limit, the visible portion of the LDT register contains only a 16-bit selector. This selector refers to a Local Descriptor Table descriptor in the GDT (see Figure 2.1).

**INTERRUPT DESCRIPTOR TABLE**

The third table needed for 80376 systems is the Interrupt Descriptor Table. The IDT contains the descriptors which point to the location of up to 256 interrupt service routines. The IDT may contain only task gates, interrupt gates and trap gates. The IDT should be at least 256 bytes in size in order to hold the descriptors for the 32 Intel Reserved Interrupts. Every interrupt used by a system must have an entry in the IDT. The IDT entries are referenced by INT instructions, external interrupt vectors, and exceptions.

**DESCRIPTORS**

The object to which the segment selector points to is called a descriptor. Descriptors are eight-byte quantities which contain attributes about a given region of linear address space. These attributes include the 32-bit logical base address of the segment, the 20-bit length and granularity of the segment, the protection level, read, write or execute privileges, and the type of segment. All of the attribute information about a segment is contained in 12 bits in the segment descriptor. Figure 3.3 shows the general format of a descriptor. All segments on the the 80376 have three attribute fields in common: the Present bit (P), the Descriptor Privilege Level bits (DPL) and the Segment bit (S). P=1 if the segment is loaded in physical memory, if P = 0 then any attempt to access the segment causes a not present exception (exception 11). The DPL is a two-bit field which specifies the protection level, 0–3, associated with a segment.

The 80376 has two main categories of segments: system segments, and non-system segments (for code and data). The segment bit, S, determines if a given segment is a system segment, a code segment or a data segment. If the S bit is 1 then the segment is either a code or data segment, if it is 0 then the segment is a system segment.

Note that although the 80376 is limited to a 16-Mbyte Physical address space (2^24), its base address allows a segment to be placed anywhere in a 4-Gbyte linear address space. When writing code for the 80376, users should keep code portability to an 80386 processor (or other processors with a larger physical address space) in mind. A segment base address can be placed anywhere in this 4-Gbyte linear address space, but a physical address will be

---

**Figure 3.3. Segment Descriptors**

<table>
<thead>
<tr>
<th>31</th>
<th>0 BYTE ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGMENT BASE 15...0</td>
<td>SEGMENT LIMIT 15...0</td>
</tr>
<tr>
<td>BASE 31...24</td>
<td>G 1 0 A V L LIMIT 19...16</td>
</tr>
</tbody>
</table>

- **BASE**: Base Address of the segment
- **LIMIT**: The length of the segment
- **P**: Present Bit 1 = Present 0 = Not Present
- **DPL**: Descriptor Privilege Level 0–3
- **S**: Segment Descriptor: 0 = System Descriptor, 1 = Code or Data Descriptor
- **TYPE**: Type of Segment
- **A**: Accessed Bit
- **G**: Granularity Bit 1 = Segment length is 4 Kbyte Granular 0 = Segment length is byte granular
- **0**: Bit must be zero (0) for compatibility with future processors
- **AVL**: Available field for user or OS

---

**Figure 3.4. Code and Data Descriptors**

<table>
<thead>
<tr>
<th>31</th>
<th>0 BYTE ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGMENT BASE 15...0</td>
<td>SEGMENT LIMIT 15...0</td>
</tr>
<tr>
<td>BASE 31...24</td>
<td>G 1 0 A V L LIMIT 19...16</td>
</tr>
</tbody>
</table>

- **G**: Granularity Bit 1 = Segment length is 4 Kbyte Granular 0 = Segment length is byte granular
- **0**: Bit must be zero (0) for compatibility with future processors
- **AVL**: Available field for user or OS

4-58
Table 3.1. Access Rights Byte Definition for Code and Data Descriptors

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Present (P)</td>
<td>P = 1 Segment is mapped into physical memory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P = 0 No mapping to physical memory exits</td>
</tr>
<tr>
<td>6–5</td>
<td>Descriptor Privilege Level (DPL)</td>
<td>Segment privilege attribute used in privilege tests.</td>
</tr>
<tr>
<td>4</td>
<td>Segment Descriptor (S)</td>
<td>S = 1 Code or Data (includes stacks) segment descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S = 0 System Segment Descriptor or Gate Descriptor</td>
</tr>
<tr>
<td>3</td>
<td>Executable (E)</td>
<td>E = 0 Descriptor type is data segment:</td>
</tr>
<tr>
<td></td>
<td>Expansion</td>
<td>ED = 0 Expand up segment, offsets must be ≤ limit.</td>
</tr>
<tr>
<td></td>
<td>Direction (ED)</td>
<td>ED = 1 Expand down segment, offsets must be &gt; limit.</td>
</tr>
<tr>
<td></td>
<td>Writable (W)</td>
<td>W = 0 Data segment may not be written into.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>W = 1 Data segment may be written into.</td>
</tr>
<tr>
<td>2</td>
<td>Executable (E)</td>
<td>E = 1 Descriptor type is code segment:</td>
</tr>
<tr>
<td></td>
<td>Conforming (C)</td>
<td>C = 1 Code segment may only be executed when CPL ≥ DPL and CPL remains unchanged.</td>
</tr>
<tr>
<td>1</td>
<td>Readable (R)</td>
<td>R = 0 Code segment may not be read.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R = 1 Code segment may be read.</td>
</tr>
<tr>
<td>0</td>
<td>Accessed (A)</td>
<td>A = 0 Segment has not been accessed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A = 1 Segment selector has been loaded into segment register or used by selector test instructions.</td>
</tr>
</tbody>
</table>

generated that is a truncated version of this linear address. Truncation will be to the maximum number of address bits. It is recommended to place EPROM at the highest physical address and DRAM at the lowest physical addresses.

Code and Data Descriptors (S = 1)

Figure 3.4 shows the general format of a code and data descriptor and Table 3.1 illustrates how the bits in the Access Right Byte are interpreted.

Code and data segments have several descriptor fields in common. The accessed bit, A, is set whenever the processor accesses a descriptor. The granularity bit, G, specifies if a segment length is 1-byte-granular or 4-Kbyte-granular. Base address bits 31–24, which are normally found in 80386 descriptors, are not made externally available on the 80376. They do not affect the operation of the 80376. The A31–A24 field should be set to allow an 80386 to correctly execute with EPROM at the upper 4096 Mbytes of physical memory.

System Descriptor Formats (S = 0)

System segments describe information about operating system tables, tasks, and gates. Figure 3.5 shows the general format of system segment descriptors, and the various types of system segments.

80376 system descriptors (which are the same as 80386 descriptor types 2, 5, 9, B, C, E and F) contain a 32-bit logical base address and a 20-bit segment limit.

Selector Fields

A selector has three fields: Local or Global Descriptor Table Indicator (TI), Descriptor Entry Index (Index), and Requestor (the selector’s) Privilege Level (RPL) as shown in Figure 3.6. The TI bit selects either the Global Descriptor Table or the Local Descriptor Table. The Index selects one of 8K descriptors in the appropriate descriptor table. The RPL bits allow high speed testing of the selector’s privilege attributes.

Segment Descriptor Cache

In addition to the selector value, every segment register has a segment descriptor cache register associated with it. Whenever a segment register’s contents are changed, the 8-byte descriptor associated with that selector is automatically loaded (cached) on the chip. Once loaded, all references to that segment use the cached descriptor information instead of reaccessing the descriptor. The contents of the descriptor cache are not visible to the programmer. Since descriptor caches only change when a segment register is changed, programs which modify the descriptor tables must reload the appropriate segment registers after changing a descriptor’s value.
3.3 Protection

The 80376 offers extensive protection features. These protection features are particularly useful in sophisticated embedded applications which use multitasking real-time operating systems. For simpler embedded applications these protection capabilities can be easily bypassed by making all applications run at privilege level (PL) 0.

RULES OF PRIVILEGE

The 80376 controls access to both data and procedures between levels of a task, according to the following rules.

--- Data stored in a segment with privilege level \( p \) can be accessed only by code executing at a privilege level at least as privileged as \( p \).

--- A code segment/procedure with privilege level \( p \) can only be called by a task executing at the same or a lesser privilege level than \( p \).

PRIVILEGE LEVELS

At any point in time, a task on the 80376 always executes at one of the four privilege levels. The Current Privilege Level (CPL) specifies what the task's privilege level is. A task's CPL may only be changed
by control transfers through gate descriptors to a code segment with a different privilege level. Thus, an application program running at PL = 3 may call an operating system routine at PL = 1 (via a gate) which would cause the task's CPL to be set to 1 until the operating system routine was finished.

**Selector Privilege (RPL)**

The privilege level of a selector is specified by the RPL field. The selector's RPL is only used to establish a less trusted privilege level than the current privilege level of the task for the use of a segment. This level is called the task's effective privilege level (EPL). The EPL is defined as being the least privileged (numerically larger) level of a task's CPL and a selector's RPL. The RPL is most commonly used to verify that pointers passed to an operating system procedure do not access data that is of higher privilege than the procedure that originated the pointer. Since the originator of a selector can specify any RPL value, the Adjust RPL (ARPL) instruction is provided to force the RPL bits to the originator's CPL.

**I/O Privilege**

The I/O privilege level (IOPL) lets the operating system code executing at CPL = 0 define the least privileged level at which I/O instructions can be used. An exception 13 (General Protection Violation) is generated if an I/O instruction is attempted when the CPL of the task is less privileged than the IOPL. The IOPL is stored in bits 13 and 14 of the EFLAGS register. The following instructions cause an exception 13 if the CPL is greater than IOPL: IN, INS, OUT, OUTS, STI, CLI and LOCK prefix.

**Descriptor Access**

There are basically two types of segment accesses: those involving code segments such as control transfers, and those involving data accesses. Determining the ability of a task to access a segment involves the type of segment to be accessed, the instruction used, the type of descriptor used and CPL, RPL, and DPL as described above.

Any time an instruction loads a data segment register (DS, ES, FS, GS) the 80376 makes protection validation checks. Selectors loaded in the DS, ES, FS, GS registers must refer only to data segment or readable code segments.

Finally the privilege validation checks are performed. The CPL is compared to the EPL and if the EPL is more privileged than the CPL, an exception 13 (general protection fault) is generated.

The rules regarding the stack segment are slightly different than those involving data segments. Instructions that load selectors into SS must refer to data segment descriptors for writeable data segments. The DPL and RPL must equal the CPL of all other descriptor types or a privilege level violation will cause an exception 13. A stack not present fault causes an exception 12.

**PRIVILEGE LEVEL TRANSFERS**

Inter-segment control transfers occur when a selector is loaded in the CS register. For a typical system most of these transfers are simply the result of a call or a jump to another routine. There are five types of control transfers which are summarized in Table 3.2. Many of these transfers result in a privilege level transfer. Changing privilege levels is done only by control transfers, using gates, task switches, and interrupt or trap gates.

Control transfers can only occur if the operation which loaded the selector references the correct descriptor type. Any violation of these descriptor usage rules will cause an exception 13.

**CALL GATES**

Gates provide protected indirect CALLs. One of the major uses of gates is to provide a secure method of privilege transfers within a task. Since the operating system defines all of the gates in a system, it can ensure that all gates only allow entry into a few trusted procedures.
Table 3.2. Descriptor Types Used for Control Transfer

<table>
<thead>
<tr>
<th>Control Transfer Types</th>
<th>Operation Types</th>
<th>Descriptor Referenced</th>
<th>Descriptor Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersegment within the same privilege level</td>
<td>JMP, CALL, RET, IRET*</td>
<td>Code Segment</td>
<td>GDT/LDT</td>
</tr>
<tr>
<td>Intersegment to the same or higher privilege level</td>
<td>CALL</td>
<td>Call Gate</td>
<td>GDT/LDT</td>
</tr>
<tr>
<td>Interrupt within task may change CPL</td>
<td>Interrupt Instruction, Exception, External Interrupt</td>
<td>Trap or Interrupt Gate</td>
<td>IDT</td>
</tr>
<tr>
<td>Intersegment to a lower privilege level (changes task CPL)</td>
<td>RET, IRET*</td>
<td>Code Segment</td>
<td>GDT/LDT</td>
</tr>
<tr>
<td>Task Switch</td>
<td>CALL, JMP</td>
<td>Task State Segment</td>
<td>GDT</td>
</tr>
<tr>
<td></td>
<td>IRET**, Interrupt Instruction, Exception, External Interrupt</td>
<td>Task Gate</td>
<td>IDT</td>
</tr>
</tbody>
</table>

*NT (Nested Task bit of flag register) = 0
**NT (Nested Task bit of flag register) = 1
NOTE:
BIT_MAP_OFFSET must be ≤ DFFFH

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000</td>
<td>BACK LINK</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ESP0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP1</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP2</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS0</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS1</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS2</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR3</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIP</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EFLAGS</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECX</td>
<td>44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDX</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBX</td>
<td>52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP</td>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBP</td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDI</td>
<td>68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP</td>
<td>72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDI</td>
<td>76</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Type = 9: Available 80376 TSS.
Type = B: Busy 80376 TSS.

Figure 3.7. 80376 TSS And TSS Registers
TASK SWITCHING

A very important attribute of any multi-tasking operating system is its ability to rapidly switch between tasks or processes. The 80376 directly supports this operation by providing a task switch instruction in hardware. The 80376 task switch operation saves the entire state of the machine (all of the registers, tasks or processes). The operating system thus has the information as the reason the task is inactive, the time the task has spent running, and open files belonging to the task.

The TSS descriptor points to a segment (see Figure 3.7) containing the entire 80376 execution state. A task gate descriptor contains a TSS selector. The limit of an 80376 TSS must be greater than 64H, and can be as large as 16 Mbytes. In the additional TSS space, the operating system is free to store additional information as the reason the task is inactive, the time the task has spent running, and open files belonging to the task.

Each Task must have a TSS associated with it. The current TSS is identified by a special register in the 80376 called the Task State Segment Register (TSS). This register contains a selector referring to the task state segment descriptor that defines the current TSS. A hidden base and limit register associated with the TSS descriptor is loaded whenever TR is loaded with a new selector. Returning from a task is accomplished by the IRET instruction. When IRET is executed, control is returned to the task which was interrupted. The current executing task's state is saved in the TSS and the old task state is restored from its TSS.

Several bits in the flag register and CR0 register give information about the state of a task which is useful to the operating system. The Nested Task bit, NT, controls the function of the IRET instruction. If NT = 0 the IRET instruction performs the regular return. If NT = 1, IRET performs a task switch operation back to the previous task. The NT bit is set or reset in the following fashion:

When a CALL or INT instruction initiates a task switch, the new TSS will be marked busy and the back link field of the new TSS set to the old TSS selector. The NT bit of the new task is set by CALL or INT initiated task switches. An interrupt that does not cause a task switch will clear NT (The NT bit will be restored after execution of the interrupt handler). NT may also be set or cleared by POPF or IRET instructions.

The 80376 task state segment is marked busy by changing the descriptor type field from TYPE 9 to TYPE OBH. Use of a selector that references a busy task state segment causes an exception 13.

The coprocessor's state is not automatically saved when a task switch occurs. The Task Switched Bit, TS, in the CR0 register helps deal with the processor's state in a multi-tasking environment. Whenever the 80376 switches tasks, it sets the TS bit. The 80376 detects the first use of a processor extension instruction after a task switch and causes the processor extension not available exception 7. The exception handler for exception 7 may then decide whether to save the state of the coprocessor.

The T bit in the 80376 TSS indicates that the processor should generate a debug exception when switching to a task. If T = 1 then upon entry to a new task a debug exception 1 will be generated.
PROTECTION AND I/O PERMISSION BIT MAP

The I/O instructions that directly refer to addresses in the processor's I/O space are IN, INS, OUT and OUTS. The 80376 has the ability to selectively trap references to specific I/O addresses. The structure that enables selective trapping is the I/O Permission Bit Map in the TSS segment (see Figures 3.7 and 3.8). The I/O permission map is a bit vector. The size of the map and its location in the TSS segment are variable. The processor locates the I/O permission map by means of the I/O map base field in the fixed portion of the TSS. The I/O map base field is 16 bits wide and contains the offset of the beginning of the I/O permission map.

If an I/O instruction (IN, INS, OUT or OUTS) is encountered, the processor first checks whether CPL \( \leq \) IOPL. If this condition is true, the I/O operation may proceed. If not true, the processor checks the I/O permission map.

Each bit in the map corresponds to an I/O port byte address; for example, the bit for port 41 is found at I/O map base + 5 linearly, \((5 \times 8 = 40)\), bit offset 1. The processor tests all the bits that correspond to the I/O addresses spanned by an I/O operation; for example, a double word operation tests four bits corresponding to four adjacent byte addresses. If any tested bit is set, the processor signals a general protection exception. If all the tested bits are zero, the I/O operations may proceed.

It is not necessary for the I/O permission map to represent all the I/O addresses. I/O addresses not spanned by the map are treated as if they had one-bits in the map. The I/O map base should be at least one byte less than the TSS limit and the last byte beyond the I/O mapping information must contain all 1's.

Because the I/O permission map is in the TSS segment, different tasks can have different maps. Thus, the operating system can allocate ports to a task by changing the I/O permission map in the task's TSS.

IMPORTANT IMPLEMENTATION NOTE:
Beyond the last byte of I/O mapping information in the I/O permission bit map must be a byte containing all 1's. The byte of all 1's must be within the limit of the 80376's TSS segment (see Figure 3.7).

4.0 FUNCTIONAL DATA

The Intel 80376 embedded processor features a straightforward functional interface to the external hardware. The 80376 has separate parallel buses for data and address. The data bus is 16 bits in width, and bidirectional. The address bus outputs 24-bit address values using 23 address lines and two-byte enable signals.

The 80376 has two selectable address bus cycles: pipelined and non-pipelined. The pipelining option allows as much time as possible for data access by
starting the pending bus cycle before the present bus cycle is finished. A non-pipelined bus cycle gives the highest bus performance by executing every bus cycle in two processor clock cycles. For maximum design flexibility, the address pipelining option is selectable on a cycle-by-cycle basis.

The processor’s bus cycle is the basic mechanism for information transfer, either from system to processor, or from processor to system. 80376 bus cycles perform data transfer in a minimum of only two clock periods. On a 16-bit data bus, the maximum 80376 transfer bandwidth at 16 MHz is therefore 16 Mbytes/sec. However, any bus cycle will be extended for more than two clock periods if external hardware withholds acknowledgement of the cycle.

The 80376 can relinquish control of its local buses to allow mastership by other devices, such as direct memory access (DMA) channels. When relinquished, HLDA is the only output pin driven by the 80376, providing near-complete isolation of the processor from its system (all other output pins are in a float condition).

4.1 Signal Description Overview

Ahead is a brief description of the 80376 input and output signals arranged by functional groups. Note the # symbol at the end of a signal name indicates the active, or asserted, state occurs when the signal is at a LOW voltage. When no # is present after the signal name, the signal is asserted when at the HIGH voltage level.

Example signal: M/IO#—HIGH voltage indicates Memory selected
              —LOW voltage indicates I/O selected

The signal descriptions sometimes refer to A.C. timing parameters, such as “t25 Reset Setup Time” and “t26 Reset Hold Time.” The values of these parameters can be found in Table 6.4.

CLOCK (CLK2)

CLK2 provides the fundamental timing for the 80376. It is divided by two internally to generate the internal processor clock used for instruction execution. The internal clock is comprised of two

![Figure 4.2. CLK2 Signal and Internal Processor Clock](image-url)
phases, "phase one" and "phase two". Each CLK2 period is a phase of the internal clock. Figure 4.2 illustrates the relationship. If desired, the phase of the internal processor clock can be synchronized to a known phase by ensuring the falling edge of the RESET signal meets the applicable setup and hold times $t_{25}$ and $t_{26}$.

**DATA BUS (D15–D0)**

These three-state bidirectional signals provide the general purpose data path between the 80376 and other devices. The data bus outputs are active HIGH and will float during bus hold acknowledge. Data bus reads require that read-data setup and hold times $t_{21}$ and $t_{22}$ be met relative to CLK2 for correct operation.

**ADDRESS BUS (BHE#, BLE#, A23–A1)**

These three-state outputs provide physical memory addresses or I/O port addresses. $A_{23}–A_{16}$ are LOW during I/O transfers except for I/O transfers automatically generated by coprocessor instructions. During coprocessor I/O transfers, $A_{22}–A_{16}$ are driven LOW, and $A_{23}$ is driven HIGH so that this address line can be used by external logic to generate the coprocessor select signal. Thus, the I/O address driven by the 80376 for coprocessor commands is 8000F8H, and the I/O address driven by the 80376 processor for coprocessor data is 8000FCH or 8000FEH.

The address bus is capable of addressing 16 Mbytes of physical memory space (000000H through 0FFFFFFH), and 64 Kbytes of I/O address space (000000H through 00FFFFH) for programmed I/O. The address bus is active HIGH and will float during bus hold acknowledge.

The Byte Enable outputs BHE# and BLE# directly indicate which bytes of the 16-bit data bus are involved with the current transfer. BHE# applies to $D_{15}–D_8$ and BLE# applies to $D_7–D_0$. If both BHE# and BLE# are asserted, then 16 bits of data are being transferred. See Table 4.1 for a complete decoding of these signals. The byte enables are active LOW and will float during bus hold acknowledge.

<table>
<thead>
<tr>
<th>BHE#</th>
<th>BLE#</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Word Transfer</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Byte Transfer on Upper Byte of the Data Bus, $D_{15}–D_8$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Byte Transfer on Lower Byte of the Data Bus, $D_7–D_0$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Never Occurs</td>
</tr>
</tbody>
</table>

Table 4.1. Byte Enable Definitions
BUS CYCLE DEFINITION SIGNALS
(W/R#, D/C#, M/IO#, LOCK#)

These three-state outputs define the type of bus cycle being performed: W/R# distinguishes between write and read cycles, D/C# distinguishes between data and control cycles, M/IO# distinguishes between memory and I/O cycles, and LOCK# distinguishes between locked and unlocked bus cycles. All of these signals are active LOW and will float during bus acknowledge.

The primary bus cycle definition signals are W/R#, D/C# and M/IO#, since these are the signals driven valid as ADS# (Address Status output) becomes active. The LOCK# signal is driven valid at the same time the bus cycle begins, which due to address pipelining, could be after ADS# becomes active. Exact bus cycle definitions, as a function of W/R#, D/C# and M/IO# are given in Table 4.2.

LOCK# indicates that other system bus masters are not to gain control of the system bus while it is active. LOCK# is activated on the CLK2 edge that begins the first locked bus cycle (i.e., it is not active at the same time as the other bus cycle definition pins) and is deactivated when ready is returned to the end of the last bus cycle which is to be locked. The beginning of a bus cycle is determined when READY# is returned in a previous bus cycle and another is pending (ADS# is active) or the clock in which ADS# is driven active if the bus was idle. This means that it follows more closely with the write data rules when it is valid, but may cause the bus to be locked longer than desired. The LOCK# signal may be explicitly activated by the LOCK prefix on certain instructions. LOCK# is always asserted when executing the XCHG instruction, during descriptor updates, and during the interrupt acknowledge sequence.

Table 4.2. Bus Cycle Definition

<table>
<thead>
<tr>
<th>M/IO#</th>
<th>D/C#</th>
<th>W/R#</th>
<th>Bus Cycle Type</th>
<th>Locked?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>INTERRUPT ACKNOWLEDGE</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Does Not Occur</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>I/O DATA READ</td>
<td>No</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>I/O DATA WRITE</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>MEMORY CODE READ</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>HALT:</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SHUTDOWN:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Address = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Address = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BHE# = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BHE# = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BLE# = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BLE# = 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>MEMORY DATA READ</td>
<td>Some Cycles</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>MEMORY DATA WRITE</td>
<td>Some Cycles</td>
</tr>
</tbody>
</table>

BUS CONTROL SIGNALS
(ADS#, READY#, NA#)

The following signals allow the processor to indicate when a bus cycle has begun, and allow other system hardware to control address pipelining and bus cycle termination.

Address Status (ADS#)

This three-state output indicates that a valid bus cycle definition and address (W/R#, D/C#, M/IO#, BHE#, BLE# and A23–A1) are being driven at the 80376 pins. ADS# is an active LOW output. Once ADS# is driven active, valid address, byte enables, and definition signals will not change. In addition, ADS# will remain active until its associated bus cycle begins (when READY# is returned for the previous bus cycle when running pipelined bus cycles). ADS# will float during bus hold acknowledge. See sections Non-Pipelined Bus Cycles (page 43) and Pipelined Bus Cycles (page 45) for additional information on how ADS# is asserted for different bus states.

Transfer Acknowledge (READY#)

This input indicates the current bus cycle is complete, and the active bytes indicated by BHE# and BLE# are accepted or provided. When READY# is sampled active during a read cycle or interrupt acknowledge cycle, the 80376 latches the input data and terminates the cycle. When READY# is sampled active during a write cycle, the processor terminates the bus cycle.
READY# is ignored on the first bus state of all bus cycles, and sampled each bus state thereafter until asserted. READY# must eventually be asserted to acknowledge every bus cycle, including Halt Indication and Shutdown Indication bus cycles. When being sampled, READY# must always meet setup and hold times t19 and t20 for correct operation.

**Next Address Request (NA#)**

This is used to request pipelining. This input indicates the system is prepared to accept new values of BHE#, BLE#, A23-A1, W/R#, D/C# and M/IO# from the 80376 even if the end of the current cycle is not being acknowledged on READY#. If this input is active when sampled, the next bus cycle’s address and status signals are driven onto the bus, provided the next bus request is already pending internally. NA# is ignored in clock cycles in which ADS# or READY# is activated. This signal is active LOW and must satisfy setup and hold times t15 and t16 for correct operation. See *Pipelined Bus Cycles* (page 45) and *Read and Write Cycles* (page 42) for additional information.

**BUS ARBITRATION SIGNALS (HOLD, HLDA)**

This section describes the mechanism by which the processor relinquishes control of its local buses when requested by another bus master device. See *Entering and Exiting Hold Acknowledge* (page 52) for additional information.

**Bus Hold Request (HOLD)**

This input indicates some device other than the 80376 requires bus mastership. When control is granted, the 80376 floats A23-A1, BHE#, BLE#, D15-D0, LOCK#, M/IO#, D/C#, W/R# and ADS#, and then activates HLDA, thus entering the bus hold acknowledge state. The local bus will remain granted to the requesting master until HOLD becomes inactive. When HOLD becomes inactive, the 80376 will deactivate HLDA and drive the local bus (at the same time), thus terminating the hold acknowledge condition.

HOLD must remain asserted as long as any other device is a local bus master. External pull-up resistors may be required when in the hold acknowledge state since none of the 80376 floated outputs have internal pull-up resistors. See *Resistor Recommendations* (page 59) for additional information. HOLD is not recognized while RESET is active but is recognized during the time between the high-to-low transition of RESET and the first instruction fetch. If RESET is asserted while HOLD is asserted, RESET has priority and places the bus into an idle state, rather than the hold acknowledge (high-impedance) state.

HOLD is a level-sensitive, active HIGH, synchronous input. HOLD signals must always meet setup and hold times t23 and t24 for correct operation.

**Bus Hold Acknowledge (HLDA)**

When active (HIGH), this output indicates the 80376 has relinquished control of its local bus in response to an asserted HOLD signal, and is in the bus Hold Acknowledge state.

The Bus Hold Acknowledge state offers near-complete signal isolation. In the Hold Acknowledge state, HLDA is the only signal being driven by the 80376. The other output signals or bidirectional signals (D15-D0, BHE#, BLE#, A23-A1, W/R#, D/C#, M/IO#, LOCK# and ADS#) are in a high-impedance state so the requesting bus master may control them. These pins remain OFF throughout the time that HLDA remains active (see Table 4.3). Pull-up resistors may be desired on several signals to avoid spurious activity when no bus master is driving them. See *Resistor Recommendations* (page 59) for additional information.

When the HOLD signal is made inactive, the 80376 will deactivate HLDA and drive the bus. One rising edge on the NMI input is remembered for processing after the HOLD input is negated.

<table>
<thead>
<tr>
<th>Pin Value</th>
<th>Pin Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HLDA</td>
</tr>
<tr>
<td>Float</td>
<td>LOCK#, M/IO#, D/C#, W/R#, ADS#, A23-A1, BHE#, BLE#, D15-D0</td>
</tr>
</tbody>
</table>

In addition to the normal usage of Hold Acknowledge with DMA controllers or master peripherals, the near-complete isolation has particular attractiveness during system test when test equipment drives the system, and in hardware-fault-tolerant applications.

**Hold Latencies**

The maximum possible HOLD latency depends on the software being executed. The actual HOLD latency at any time depends on the current bus activity, the state of the LOCK# signal (internal to the CPU) activated by the LOCK# prefix, and interrupts. The 80376 will not honor a HOLD request until the current bus operation is complete. Table 4.4 shows the types of bus operations that can affect HOLD latency, and indicates the types of delays that
these operations may introduce. When considering maximum HOLD latencies, designers must select which of these bus operations are possible, and then select the maximum latency form among them.

The 80376 breaks 32-bit data or I/O accesses into 2 internally locked 16-bit bus cycles; the LOCK# signal is not asserted. The 80376 breaks unaligned 16-bit or 32-bit data or I/O accesses into 2 or 3 internally locked 16-bit bus cycles. Again the LOCK# signal is not asserted but a HOLD request will not be recognized until the end of the entire transfer.

As indicated in Table 4.4, wait states affect HOLD latency. The 80376 will not honor a HOLD request until the end of the current bus operation, no matter how many wait states are required. Systems with DMA where data transfer is critical must insure that READY# returns sufficiently soon.

Table 4.4. Locked Bus Operations Affecting HOLD Latency in Systems Clocks

}***Not Available At This Time***{

COPROCESSOR INTERFACE SIGNALS
(PEREQ, BUSY#, ERROR#)

In the following sections are descriptions of signals dedicated to the numeric coprocessor interface. In addition to the data bus, address bus, and bus cycle definition signals, these following signals control communication between the 80376 and the 80387SX processor extension.

Coprocessor Request (PEREQ)

When asserted (HIGH), this input signal indicates a coprocessor request for a data operand to be transferred to/from memory by the 80376. In response, the 80376 transfers information between the coprocessor and memory. Because the 80376 has internally stored the coprocessor opcode being executed, it performs the requested data transfer with the correct direction and memory address.

PEREQ is a level-sensitive active HIGH asynchronous signal. Setup and hold times, t29 and t30, relative to the CLK2 signal must be met to guarantee recognition at a particular clock edge. This signal is provided with a weak internal pull-down resistor of around 20 KΩ to ground so that it will not float active when left unconnected.

Coprocessor Busy (BUSY#)

When asserted (LOW), this input indicates the coprocessor is still executing an instruction, and is not yet able to accept another. When the 80376 encounters any coprocessor instruction which operates on the numerics stack (e.g. load, pop, or arithmetic operation), or the WAIT instruction, this input is first automatically sampled until it is seen to be inactive. This sampling of the BUSY# input prevents overrunning the execution of a previous coprocessor instruction.

The F(N)INIT, F(N)CLEX coprocessor instructions are allowed to execute even if BUSY# is active, since these instructions are used for coprocessor initialization and exception-clearing.

BUSY# is an active LOW, level-sensitive asynchronous signal. Setup and hold times, t29 and t30, relative to the CLK2 signal must be met to guarantee recognition at a particular clock edge. This pin is provided with a weak internal pull-up resistor of around 20 KΩ to VCC so that it will not float active when left unconnected.

BUSY# serves an additional function. If BUSY# is sampled LOW at the falling edge of RESET, the 80376 processor performs an internal self-test (see Bus Activity During and Following Reset on page 54). If BUSY# is sampled HIGH, no self-test is performed.

Coprocessor Error (ERROR#)

When asserted (LOW), this input signal indicates that the previous coprocessor instruction generated a coprocessor error of a type not masked by the coprocessor's control register. This input is automatically sampled by the 80376 when a coprocessor instruction is encountered, and if active, the 80376 generates exception 16 to access the error-handling software.

Several coprocessor instructions, generally those which clear the numeric error flags in the coprocessor or save coprocessor state, do execute without the 80376 generating exception 16 even if ERROR# is active. These instructions are FNINIT, FNACLEX, FNSAVE, FNSAVEAX, FNSTCW, FNSAVEV and FNSAVE.
ERROR # is an active LOW, level-sensitive asynchronous signal. Setup and hold times t₁₂₉ and t₁₃₀, relative to the CLK2 signal must be met to guarantee recognition at a particular clock edge. This pin is provided with a weak internal pull-up resistor of around 20 KΩ to VCC so that it will not float active when left unconnected.

INTERRUPT SIGNALS (INTR, NMI, RESET)

The following descriptions cover inputs that can interrupt or suspend execution of the processor's current instruction stream.

Maskable Interrupt Request (INTR)

When asserted, this input indicates a request for interrupt service, which can be masked by the 80376 Flag Register IF bit. When the 80376 responds to the INTR input, it performs two interrupt acknowledge bus cycles and, at the end of the second, latches an 8-bit interrupt vector on D₇-D₀ to identify the source of the interrupt.

INTR is an active HIGH, level-sensitive asynchronous signal. Setup and hold times, t₁₂₇ and t₁₂₈, relative to the CLK2 signal must be met to guarantee recognition at a particular clock edge. To assure recognition of an INTR request, INTR should remain active until the first interrupt acknowledge bus cycle begins. INTR is sampled at the beginning of every instruction. In order to be recognized at a particular instruction boundary, INTR must be active at least eight CLK2 clock periods before the beginning of the execution of the instruction. If recognized, the 80376 will begin execution of the interrupt.

Non-Maskable Interrupt Request (NMI)

This input indicates a request for interrupt service which cannot be masked by software. The non-maskable interrupt request is always processed according to the pointer or gate in slot 2 of the interrupt table. Because of the fixed NMI slot assignment, no interrupt acknowledge cycles are performed when processing NMI.

NMI is an active HIGH, rising edge-sensitive asynchronous signal. Setup and hold times, t₁₂₇ and t₁₂₈, relative to the CLK2 signal must be met to guarantee recognition at a particular clock edge. To assure recognition of NMI, it must be inactive for at least eight CLK2 periods, and then be active for at least eight CLK2 periods before the beginning of the execution of an instruction.

Once NMI processing has begun, no additional NMI's are processed until after the next IRET instruction, which is typically the end of the NMI service routine. If NMI is re-asserted prior to that time, however, one rising edge on NMI will be remembered for processing after executing the next IRET instruction.

Interrupt Latency

The time that elapses before an interrupt request is serviced (interrupt latency) varies according to several factors. This delay must be taken into account by the interrupt source. Any of the following factors can affect interrupt latency:

1. If interrupts are masked, and INTR request will not be recognized until interrupts are reenabled.
2. If an NMI is currently being serviced, an incoming NMI request will not be recognized until the 80376 encounters the IRET instruction.
3. An interrupt request is recognized only on an instruction boundary of the 80376 Execution Unit except for the following cases:
   - Repeat string instructions can be interrupted after each iteration.
   - If the instruction loads the Stack Segment register, an interrupt is not processed until after the following instruction, which should be an ESP load. This allows the entire stack pointer to be loaded without interruption.
   - If an instruction sets the interrupt flag (enabling interrupts), an interrupt is not processed until after the next instruction.

The longest latency occurs when the interrupt request arrives while the 80376 processor is executing a long instruction such as multiplication, division or a task-switch.

4. Saving the Flags register and CS:EIP registers.
5. If interrupt service routine requires a task switch, time must be allowed for the task switch.
6. If the interrupt service routine saves registers that are not automatically saved by the 80376.

RESET

This input signal suspends any operation in progress and places the 80376 in a known reset state. The 80376 is reset by asserting RESET for 15 or more CLK2 periods (80 or more CLK2 periods before requesting self-test). When RESET is active, all other input pins are ignored, and all other bus pins are driven to an idle bus state as shown in Table 4.5. If RESET and HOLD are both active at a point in time, RESET takes priority even if the 80376 was in a Hold Acknowledge state prior to RESET active.

RESET is an active HIGH, level-sensitive synchronous signal. Setup and hold times, t₁₄₆ and t₁₄₈, must be met in order to assure proper operation of the 80376.
Table 4.5. Pin State (Bus Idle) during RESET

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Signal Level during RESET</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADS#</td>
<td>1</td>
</tr>
<tr>
<td>D15–D0</td>
<td>Float</td>
</tr>
<tr>
<td>BHE#, BLE#</td>
<td>0</td>
</tr>
<tr>
<td>A23–A1</td>
<td>1</td>
</tr>
<tr>
<td>W/R#</td>
<td>0</td>
</tr>
<tr>
<td>D/C#</td>
<td>1</td>
</tr>
<tr>
<td>M/IO#</td>
<td>0</td>
</tr>
<tr>
<td>LOCK#</td>
<td>1</td>
</tr>
<tr>
<td>HLDA</td>
<td>0</td>
</tr>
</tbody>
</table>

4.2 Bus Transfer Mechanism

All data transfers occur as a result of one or more bus cycles. Logical data operands of byte and word lengths may be transferred without restrictions on physical address alignment. Any byte boundary may be used, although two physical bus cycles are performed as required for unaligned operand transfers.

The 80376 processor address signals are designed to simplify external system hardware. BHE# and BLE# provide linear selects for the two bytes of the 16-bit data bus.

Byte Enable outputs BHE# and BLE# are asserted when their associated data bus bytes are involved with the present bus cycle, as listed in Table 4.6.

Table 4.6. Byte Enables and Associated Data and_operand Bytes

<table>
<thead>
<tr>
<th>Byte Enable</th>
<th>Associated Data Bus Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>BHE#</td>
<td>D15–D8 (Byte 1—Most Significant)</td>
</tr>
<tr>
<td>BLE#</td>
<td>D7–D0 (Byte 0—Least Significant)</td>
</tr>
</tbody>
</table>

Each bus cycle is composed of at least two bus states. Each bus state requires one processor clock period. Additional bus states added to a single bus cycle are called wait states. See Bus Functional Description (page 39) for additional information.

4.3 Memory and I/O Spaces

Bus cycles may access physical memory space or I/O space. Peripheral devices in the system may either be memory-mapped, or I/O-mapped, or both. As shown in Figure 4.3, physical memory addresses range from 000000H to 0FFFFFFH (16 Mbytes) and I/O addresses from 000000H to 00FFFFH (64 Kbytes). Note the I/O addresses used by the automatic I/O cycles for coprocessor communication are 8000F0H to 8000FFH, beyond the address range of programmed I/O, to allow easy generation of a coprocessor chip select signal using the A23 and M/IO# signals.

OPERAND ALIGNMENT

With the flexibility of memory addressing on the 80376, it is possible to transfer a logical operand that spans more than one physical Dword or word of memory or I/O. Examples are 32-bit Dword or 16-bit word operands beginning at addresses not evenly divisible by 2.

Operand alignment and size dictate when multiple bus cycles are required. Table 4.6a describes the transfer cycles generated for all combinations of logical operand lengths and alignment.

Table 4.6a. Transfer Bus Cycles for Bytes, Words and Dwords

<table>
<thead>
<tr>
<th>Physical Byte Address in Memory (Low-Order Bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transfer Cycles</td>
</tr>
<tr>
<td>Byte-Length of Logical Operand</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>

Key:  
- b = byte transfer  
- w = word transfer  
- l = low-order portion  
- m = mid-order portion  
- x = don't care  
- h = high-order portion
4.4 Bus Functional Description

The 80376 has separate, parallel buses for data and address. The data bus is 16 bits in width, and bidirectional. The address bus provides a 24-bit value using 23 signals for the 23 upper-order address bits and 2 Byte Enable signals to directly indicate the active bytes. These buses are interpreted and controlled by several definition signals.

The definition of each bus cycle is given by three signals: M/IO#, W/R# and D/C#. At the same time, a valid address is present on the byte enable signals, BHE# and BLE#, and the other address signals A23-A1. A status signal, ADS#, indicates when the 80376 issues a new bus cycle definition and address.

Collectively, the address bus, data bus and all associated control signals are referred to simply as “the bus”. When active, the bus performs one of the bus cycles below:

1. Read from memory space
2. Locked read from memory space
3. Write to memory space
4. Locked write to memory space
5. Read from I/O space (or coprocessor)
6. Write to I/O space (or coprocessor)
7. Interrupt acknowledge (always locked)
8. Indicate halt, or indicate shutdown

Table 4.2 shows the encoding of the bus cycle definition signals for each bus cycle. See Bus Cycle Definition Signals (page 35) for additional information.

When the 80376 bus is not performing one of the activities listed above, it is either Idle or in the Hold Acknowledge state, which may be detected by external circuitry. The idle state can be identified by the 80376 giving no further assertions on its address strobe output (ADS#) since the beginning of its most recent bus cycle, and the most recent bus cycle having been terminated. The hold acknowledge state is identified by the 80376 asserting its hold acknowledge (HLDA) output.

The shortest time unit of bus activity is a bus state. A bus state is one processor clock period (two CLK2 periods) in duration. A complete data transfer occurs during a bus cycle, composed of two or more bus states.
The fastest 80376 bus cycle requires only two bus states. For example, three consecutive bus read cycles, each consisting of two bus states, are shown by Figure 4.4. The bus states in each cycle are named T1 and T2. Any memory or I/O address may be accessed by such a two-state bus cycle, if the external hardware is fast enough.

Every bus cycle continues until it is acknowledged by the external system hardware, using the 80376 READY# input. Acknowledging the bus cycle at the end of the first T2 results in the shortest bus cycle, requiring only T1 and T2. If READY# is not immediately asserted however, T2 states are repeated indefinitely until the READY# input is sampled active.

The pipelining option provides a choice of bus cycle timings. Pipelined or non-pipelined cycles are selectable on a cycle-by-cycle basis with the Next Address (NA#) input.

When pipelining is selected the address (BHE#, BLE#, A1–A23, M/IO#, D/C#, W/R#) and definition (W/R#, D/C#, M/IO# and LOCK#) of the next cycle are available before the end of the current cycle. To signal their availability, the 80376 address status output (ADS#) is asserted. Figure 4.5 illustrates the fastest read cycles with pipelined timing.

Note from Figure 4.5 the fastest bus cycles using pipelining require only two bus states, named T1P and T2P. Therefore pipelined cycles allow the same data bandwidth as non-pipelined cycles, but address-to-data access time is increased by one T-state time compared to that of a non-pipelined cycle.
### READ AND WRITE CYCLES

Data transfers occur as a result of bus cycles, classified as read or write cycles. During read cycles, data is transferred from an external device to the processor. During write cycles, data is transferred from the processor to an external device.

Two choices of bus cycle timing are dynamically selectable: non-pipelined or pipelined. After an idle bus state, the processor always uses non-pipelined timing. However, the NA# (Next Address) input may be asserted to select pipelined timing for the next bus cycle. When pipelining is selected and the 80376 has a bus request pending internally, the address and definition of the next cycle is made available even before the current bus cycle is acknowledged by READY#.

Terminating a read or write cycle, like any bus cycle, requires acknowledging the cycle by asserting the READY# input. Until acknowledged, the processor inserts wait states into the bus cycle, to allow adjustment for the speed of any external device. External hardware, which has decoded the address and bus cycle type, asserts the READY# input at the appropriate time.

At the end of the second bus state within the bus cycle, READY# is sampled. At that time, if external hardware acknowledges the bus cycle by asserting READY#, the bus cycle terminates as shown in Figure 4.6. If READY# is negated as in Figure 4.7, the 80376 executes another bus state (a wait state) and READY# is sampled again at the end of that state. This continues indefinitely until the cycle is acknowledged by READY# asserted.

When the current cycle is acknowledged, the 80376 terminates it. When a read cycle is acknowledged, the 80376 latches the information present at its data pins. When a write cycle is acknowledged, the write data of the 80376 remains valid throughout phase one of the next bus state, to provide write data hold time.

---

#### Figure 4.5. Fastest Read Cycles with Pipelined Timing

<table>
<thead>
<tr>
<th>CYCLE 1 PIPELINED (READ)</th>
<th>CYCLE 2 PIPELINED (READ)</th>
<th>CYCLE 3 PIPELINED (READ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1P</td>
<td>T2P</td>
<td>T1P</td>
</tr>
<tr>
<td>φ1</td>
<td>φ2</td>
<td>φ1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BHE#, BLE#, A1–A23 M#/O#, D/C#, W/# (OUTPUTS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALID 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ADS# (OUTPUT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALID 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NA# (INPUT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALID 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>READY# (INPUT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALID 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOCK# (OUTPUT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALID 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D0–D15 (INPUT DURING READ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
</tr>
</tbody>
</table>

240182–20
### Non-Pipelined Bus Cycles

Any bus cycle may be performed with non-pipelined timing. For example, Figure 4.6 shows a mixture of non-pipelined read and write cycles. Figure 4.6 shows that the fastest possible non-pipelined cycles have two bus states per bus cycle. The states are named T1 and T2. In phase one of T1, the address signals and bus cycle definition signals are driven valid and, to signal their availability, address strobe (ADS#) is simultaneously asserted.

During read or write cycles, the data bus behaves as follows. If the cycle is a read, the 80376 floats its data signals to allow driving by the external device being addressed. The **80376 requires that all data bus pins be at a valid logic state (HIGH or LOW) at the end of each read cycle, when READY# is asserted. The system MUST be designed to meet this requirement.** If the cycle is a write, data signals are driven by the 80376 beginning in phase two of T1 until phase one of the bus state following cycle acknowledgement.
Figure 4.7. Various Non-Pipelined Bus Cycles (Various Number of Wait States)

Figure 4.7 illustrates non-pipelined bus cycles with one wait state added to Cycles 2 and 3. READY# is sampled inactive at the end of the first T2 in Cycles 2 and 3. Therefore Cycles 2 and 3 have T2 repeated again. At the end of the second T2, READY# is sampled active.

When address pipelining is not used, the address and bus cycle definition remain valid during all wait states. When wait states are added and it is desirable to maintain non-pipelined timing, it is necessary to negate NA# during each T2 state except the last one, as shown in Figure 4.7, Cycles 2 and 3. If NA# is sampled active during a T2 other than the last one, the next state would be T2I or T2P instead of another T2.

When address pipelining is not used, the bus states and transitions are completely illustrated by Figure 4.8. The bus transitions between four possible states, T1, T2, T1, and T1h. Bus cycles consist of T1 and T2, with T2 being repeated for wait states. Otherwise the bus may be idle, T1, or in the hold acknowledge state T1h.
Bus States:
T1—first clock of a non-pipelined bus cycle (80376 drives new address and asserts ADS#).
T2—subsequent clocks of a bus cycle when NA# has not been sampled asserted in the current bus cycle.
Ti—idle state.
Th—hold acknowledge state (80376 asserts HLDA).
The fastest bus cycle consists of two states: T1 and T2.
Four basic bus states describe bus operation when not using pipelined address.

Figure 4.8. 80376 Bus States (Not Using Pipelined Address)

Bus cycles always begin with T1. T1 always leads to T2. If a bus cycle is not acknowledged during T2 and NA# is inactive, T2 is repeated. When a cycle is acknowledged during T2, the following state will be T1 of the next bus cycle if a bus request is pending internally, or Ti if there is no bus request pending, or Th if the HOLD input is being asserted.

Use of pipelining allows the 80376 to enter three additional bus states not shown in Figure 4.8. Figure 4.12 on page 49 is the complete bus state diagram, including pipelined cycles.

Pipeline Bus Cycles

Pipelining is the option of requesting the address and the bus cycle definition of the next internally pending bus cycle before the current bus cycle is acknowledged with READY# asserted. ADS# is asserted by the 80376 when the next address is issued. The pipelining option is controlled on a cycle-by-cycle basis with the NA# input signal.

Once a bus cycle is in progress and the current address has been valid for at least one entire bus state, the NA# input is sampled at the end of every phase one until the bus cycle is acknowledged. During non-pipelined bus cycles NA# is sampled at the end of phase one in every T2. An example is Cycle 2 in Figure 4.9, during which NA# is sampled at the end of phase one of every T2 (it was asserted once during the first T2 and has no further effect during that bus cycle).
Following any idle bus state (Ti), bus cycles are non-pipelined. Within non-pipelined bus cycles, NA# is only sampled during wait states. Therefore, to begin pipelining during a group of non-pipelined bus cycles requires a non-pipelined cycle with at least one wait state (Cycle 2 above).

**Figure 4.9. Transitioning to Pipelining during Burst of Bus Cycles**

If NA# is sampled active, the 80376 is free to drive the address and bus cycle definition of the next bus cycle, and assert ADS#, as soon as it has a bus request internally pending. It may drive the next address as early as the next bus state, whether the current bus cycle is acknowledged at that time or not.

Regarding the details of pipelining, the 80376 has the following characteristics:

1. The next address and status may appear as early as the bus state after NA# was sampled active (see Figures 4.9 or 4.10). In that case, state T2P is entered immediately. However, when there is not an internal bus request already pending, the next address and status will not be available immediately after NA# is asserted and T2I is entered instead of T2P (see Figure 4.11 Cycle 3). Provided the current bus cycle isn't yet acknowledged by READY# asserted, T2P will be entered as soon as the 80376 does drive the next address and status. External hardware should therefore observe the ADS# output as confirmation the next address and status are actually being driven on the bus.

2. Any address and status which are validated by a pulse on the 80376 ADS# output will remain stable on the address pins for at least two processor clock periods. The 80376 cannot produce a new address and status more frequently than every two processor clock periods (see Figures 4.9, 4.10 and 4.11).

3. Only the address and bus cycle definition of the very next bus cycle is available. The pipelining capability cannot look further than one bus cycle ahead (see Figure 4.11, Cycle 1).
Following any idle bus state (Ti) the bus cycle is always non-pipelined and NA# is only sampled during wait states. To start, address pipelining after an idle state requires a non-pipelined cycle with at least one wait state (cycle 1 above). The pipelined cycles (2, 3, 4 above) are shown with various numbers of wait states.

Figure 4.10. Fastest Transition to Pipelined Bus Cycle Following Idle Bus State

The complete bus state transition diagram, including pipelining is given by Figure 4.12. Note it is a superset of the diagram for non-pipelined only, and the three additional bus states for pipelining are drawn in bold.

The fastest bus cycle with pipelining consists of just two bus states, T1P and T2P (recall for non-pipelined it is T1 and T2). T1P is the first bus state of a pipelined cycle.

**Initiating and Maintaining Pipelined Bus Cycles**

Using the state diagram Figure 4.12, observe the transitions from an idle state, Ti, to the beginning of a pipelined bus cycle T1P. From an idle state, Ti, the first bus cycle must begin with T1, and is therefore a non-pipelined bus cycle. The next bus cycle will be pipelined, however, provided NA# is asserted and the first bus cycle ends in a T2P state (the address and status for the next bus cycle is driven during T2P). The fastest path from an idle state to a pipelined bus cycle is shown in bold below:

Ti, T1, T1-T2-T2P, T1P-T2P,
Figure 4.11. Details of Address Pipelining during Cycles with Wait States

T1-T2-T2P are the states of the bus cycle that establishes address pipelining for the next bus cycle, which begins with T1P. The same is true after a bus hold state, shown below:

- T1-T2-T2P, hold acknowledge non-pipelined states
- T1P-T2P, pipelined cycle

The transition to pipelined address is shown functionally by Figure 4.10, Cycle 1. Note that Cycle 1 is used to transition into pipelined address timing for the subsequent Cycles 2, 3 and 4, which are pipelined. The NA# input is asserted at the appropriate time to select address pipelining for Cycles 2, 3 and 4.

Once a bus cycle is in progress and the current address and status has been valid for one entire bus state, the NA# input is sampled at the end of every phase one until the bus cycle is acknowledged.
Bus States:
T1—first clock of a non-pipelined bus cycle (80376 drives new address, status and asserts ADS#).
T2—subsequent clocks of a bus cycle when NA# has not been sampled asserted in the current bus cycle.
T2I—subsequent clocks of a bus cycle when NA# has been sampled asserted in the current bus cycle but there is not yet an internal bus request pending (80376 will not drive new address, status or assert ADS#).
T2P—subsequent clocks of a bus cycle when NA# has been sampled asserted in the current bus cycle and there is an internal bus request pending (80376 drives new address, status and asserts ADS#).
T1P—first clock of a pipelined bus cycle.
Th—idle state.
Th—hold acknowledge state (80376 asserts HLDA).
Asserting NA# for pipelined bus cycles gives access to three more bus states: T2I, T2P and T1P.
Using pipelining the fastest bus cycle consists of T1P and T2P.

Figure 4.12. 80376 Processor Complete Bus States (Including Pipelining)
Sampling begins in T2 during Cycle 1 in Figure 4.10. Once NA# is sampled active during the current cycle, the 80376 is free to drive a new address and bus cycle definition on the bus as early as the next bus state. In Figure 4.10, Cycle 1 for example, the next address and status is driven during state T2P. Thus Cycle 1 makes the transition to pipelined timing, since it begins with T1 but ends with T2P. Because the address for Cycle 2 is available before Cycle 2 begins, Cycle 2 is called a pipelined bus cycle, and it begins with T1P. Cycle 2 begins as soon as READY# asserted terminates Cycle 1.

Examples of transition bus cycles are Figure 4.10, Cycle 1 and Figure 4.9, Cycle 2. Figure 4.10 shows transition during the very first cycle after an idle bus state, which is the fastest possible transition into address pipelining. Figure 4.9, Cycle 2 shows a transition cycle occurring during a burst of bus cycles. In any case, a transition cycle is the same whenever it occurs: it consists at least of T1, T2 (NA# is asserted at that time), and T2P (provided the 80376 has an internal bus request already pending, which it almost always has). T2P states are repeated if wait states are added to the cycle.

Note that only three states (T1, T2 and T2P) are required in a bus cycle performing a transition from non-pipelined into pipelined timing, for example Figure 4.10, Cycle 1. Figure 4.10, Cycles 2, 3 and 4 show that pipelining can be maintained with two-state bus cycles consisting only of T1P and T2P.

Once a pipelined bus cycle is in progress, pipelined timing is maintained for the next cycle by asserting NA# and detecting that the 80376 enters T2P during the current bus cycle. The current bus cycle must end in state T2P for pipelining to be maintained in the next cycle. T2P is identified by the assertion of ADS#. Figures 4.9 and 4.10 however, each show pipelining ending after Cycle 4 because Cycle 4 ends in T2I. This indicates the 80376 didn't have an internal bus request prior to the acknowledgement of Cycle 4. If a cycle ends with a T2 or T2I, the next cycle will not be pipelined.

Realistically, pipelining is almost always maintained as long as NA# is sampled asserted. This is so because in the absence of any other request, a code prefetch request is always internally pending until the instruction decoder and code prefetch queue are completely full. Therefore pipelining is maintained for long bursts of bus cycles, if the bus is available (i.e., HOLD inactive) and NA# is sampled active in each of the bus cycles.

**INTERRUPT ACKNOWLEDGE (INTA) CYCLES**

In response to an interrupt request on the INTR input when interrupts are enabled, the 80376 performs two interrupt acknowledge cycles. These bus cycles are similar to read cycles in that bus definition signals define the type of bus activity taking place, and each cycle continues until acknowledged by READY# sampled active.

The state of A2 distinguishes the first and second interrupt acknowledge cycles. The byte address driven during the first interrupt acknowledge cycle is 4 (A23–A3, A1, BLE# LOW, A2 and BHE# HIGH). The byte address driven during the second interrupt acknowledge cycle is 0 (A23–A1, BLE# LOW and BHE# HIGH).

The LOCK# output is asserted from the beginning of the first interrupt acknowledge cycle until the end of the second interrupt acknowledge cycle. Four idle bus states, T1, are inserted by the 80376 between the two interrupt acknowledge cycles for compatibility with the interrupt specification T\text{RHRL} of the 8259A Interrupt Controller and the 82370 Integrated Peripheral.
Interrupt Vector (0–255) is read on D0–D7 at end of second Interrupt Acknowledge bus cycle. Because each Interrupt Acknowledge bus cycle is followed by idle bus states, asserting NA# has no practical effect. Choose the approach which is simplest for your system hardware design.

Figure 4.13. Interrupt Acknowledge Cycles

During both interrupt acknowledge cycles, D15–D0 float. No data is read at the end of the first interrupt acknowledge cycle. At the end of the second interrupt acknowledge cycle, the 80376 will read an external interrupt vector from D7–D0 of the data bus. The vector indicates the specific interrupt number (from 0–255) requiring service.

HALT INDICATION CYCLE

The 80376 execution unit halts as a result of executing a HLT instruction. Signaling its entrance into the halt state, a halt indication cycle is performed. The halt indication cycle is identified by the state of the bus definition signals shown on page 34, Bus Cycle Definition Signals, and a byte address of 2. The halt indication cycle must be acknowledged by READY# asserted. A halted 80376 resumes execution when INTR (if interrupts are enabled), NMI or RESET is asserted.
**SHUTDOWN INDICATION CYCLE**

The 80376 shuts down as a result of a protection fault while attempting to process a double fault. Signaling its entrance into the shutdown state, a shutdown indication cycle is performed. The shutdown indication cycle is identified by the state of the bus definition signals shown on page 34 Bus Cycle Definition Signals and a byte address of 0. The shutdown indication cycle must be acknowledged by READY# asserted. A shutdown 80376 resumes execution when NMI or RESET is asserted.

**ENTERING AND EXITING HOLD ACKNOWLEDGE**

The bus hold acknowledge state, \( T_h \), is entered in response to the HOLD input being asserted. In the bus hold acknowledge state, the 80376 floats all outputs or bidirectional signals, except for HLDA. HLDA is asserted as long as the 80376 remains in the bus hold acknowledge state. In the bus hold acknowledge state, all inputs except HOLD and RESET are ignored.
Figure 4.15. Example Shutdown Indication Cycle from Non-Pipelined Cycle

$T_h$ may be entered from a bus idle state as in Figure 4.16 or after the acknowledgement of the current physical bus cycle if the LOCK# signal is not asserted, as in Figures 4.17 and 4.18.

$T_h$ is exited in response to the HOLD input being negated. The following state will be $T_i$ as in Figure 4.16 if no bus request is pending. The following bus state will be $T_1$ if a bus request is internally pending, as in Figures 4.17 and 4.18. $T_h$ is exited in response to RESET being asserted.

If a rising edge occurs on the edge-triggered NMI input while in $T_h$, the event is remembered as a non-maskable interrupt 2 and is serviced when $T_h$ is exited unless the 80376 is reset before $T_h$ is exited.
NOTE:
For maximum design flexibility the 80376 has no internal pull-up resistors on its outputs. Your design may require an external pullup on ADS# and other 80376 outputs to keep them negated during float periods.

Figure 4.16. Requesting Hold from Idle Bus

RESET DURING HOLD ACKNOWLEDGE
RESET being asserted takes priority over HOLD being asserted. If RESET is asserted while HOLD remains asserted, the 80376 drives its pins to defined states during reset, as in Table 4.5, Pin State During Reset, and performs internal reset activity as usual.

If HOLD remains asserted when RESET is inactive, the 80376 enters the hold acknowledge state before performing its first bus cycle, provided HOLD is still asserted when the 80376 processor would otherwise perform its first bus cycle. If HOLD remains asserted when RESET is inactive, the BUSY# input is still sampled as usual to determine whether a self test is being requested.

BUS ACTIVITY DURING AND FOLLOWING RESET
RESET is the highest priority input signal, capable of interrupting any processor activity when it is asserted. A bus cycle in progress can be aborted at any stage, or idle states or bus hold acknowledge states discontinued so that the reset state is established.
<table>
<thead>
<tr>
<th><strong>NOTE:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>HOLD is a synchronous input and can be asserted at any CLK2 edge, provided setup and hold requirements are met. This waveform is useful for determining Hold Acknowledge latency.</td>
</tr>
</tbody>
</table>

**Figure 4.17. Requesting Hold from Active Bus (NA# Inactive)**

RESET should remain asserted for at least 15 CLK2 periods to ensure it is recognized throughout the 80376, and at least 80 CLK2 periods if a 80376 self-test is going to be requested at the falling edge. RESET asserted pulses less than 15 CLK2 periods may not be recognized. RESET pulses less than 80 CLK2 periods followed by a self-test may cause the self-test to report a failure when no true failure exists.

Provided the RESET falling edge meets setup and hold times $t_{125}$ and $t_{126}$, the internal processor clock phase is defined at that time as illustrated by Figure 4.19 and Figure 6.7.
NOTE:
HOLD is a synchronous input and can be asserted at any CLK2 edge, provided setup and hold (t23 and t24) requirements are met. This waveform is useful for determining Hold Acknowledge latency.

Figure 4.18. Requesting Hold from Idle Bus (NA# Active)

An 80376 self-test may be requested at the time RESET goes inactive by having the BUSY# input at a LOW level as shown in Figure 4.19. The self-test requires \((2^{20} + \text{approximately 60})\) CLK2 periods to complete. The self-test duration is not affected by the test results. Even if the self-test indicates a problem, the 80376 attempts to proceed with the reset sequence afterwards.

After the RESET falling edge (and after the self-test if it was requested) the 80376 performs an internal initialization sequence for approximately 350 to 450 CLK2 periods.
4.5 Self-Test Signature

Upon completion of self-test (if self-test was requested by driving BUSY# LOW at the falling edge of RESET) the EAX register will contain a signature of 00000000H indicating the 80376 passed its self-test of microcode and major PLA contents with no problems detected. The passing signature in EAX, 00000000H, applies to all 80376 revision levels. Any non-zero signature indicates the 80376 unit is faulty.

4.6 Component and Revision Identifiers

To assist 80376 users, the 80376 after reset holds a component identifier and revision identifier in its DX register. The upper 8 bits of DX hold 33H as identification of the 80376 component. (The lower nibble, 03H, refers to the Intel386™ architecture. The upper nibble, 30H, refers to the third member of the Intel386 family). The lower 8 bits of DX hold an 8-bit unsigned binary number related to the...
component revision level. The revision identifier will, in general, chronologically track those component stepping which are intended to have certain improvements or distinction from previous stepping. The 80376 revision identifier will track that of the 80386 where possible.

The revision identifier is intended to assist 80376 users to a practical extent. However, the revision identifier value is not guaranteed to change with every stepping revision, or to follow a completely uniform numerical sequence, depending on the type or intention of revision, or manufacturing materials required to be changed. Intel has sole discretion over these characteristics of the component.

<table>
<thead>
<tr>
<th>Table 4.7. Component and Revision Identifier History</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>80376 Stepping Name</strong></td>
</tr>
<tr>
<td>A0</td>
</tr>
</tbody>
</table>

4.7 Coprocessor Interfacing

The 80376 provides an automatic interface for the Intel 80387SX numeric floating-point coprocessor. The 80387SX coprocessor uses an I/O mapped interface driven automatically by the 80376 and assisted by three dedicated signals: BUSY #, ERROR # and PEREQ.

As the 80376 begins supporting a coprocessor instruction, it tests the BUSY # and ERROR # signals to determine if the coprocessor can accept its next instruction. Thus, the BUSY # and ERROR # inputs eliminate the need for any "preamble" bus cycles for communication between processor and coprocessor. The 80387SX can be given its command opcode immediately. The dedicated signals provide instruction synchronization, and eliminate the need of using the 80376 WAIT opcode (9BH) for 80387SX instruction synchronization (the WAIT opcode was required when the 80387SX was used with the 8037 coprocessor).

Custom coprocessors can be included in 80376 based systems by memory-mapped or I/O-mapped interfaces. Such coprocessor interfaces allow a completely custom protocol, and are not limited to a set of coprocessor protocol "primitives". Instead, memory-mapped or I/O-mapped interfaces may use all applicable 80376 instructions for high-speed coprocessor communication. The BUSY # and ERROR # inputs of the 80376 may also be used for the custom coprocessor interface, if such hardware assist is desired. These signals can be tested by the 80376 WAIT opcode (9BH). The WAIT instruction will wait until the BUSY # input is inactive (interruptable by an NMI or enabled INTR input), but generates an exception 16 fault if the ERROR # pin is active when the BUSY # goes (or is) inactive. If the custom coprocessor interface is memory-mapped, protection of the addresses used for the interface can be provided with the segmentation mechanism of the 80376. If the custom interface is I/O-mapped, protection of the interface can be provided with the 80376 IOPL (I/O Privilege Level) mechanism.

The 80387SX numeric coprocessor interface is I/O mapped as shown in Table 4.8. Note that the 80387SX coprocessor interface addresses are beyond the 0H-0FFFFH range for programmed I/O. When the 80376 supports the 80387SX coprocessor, the 80376 automatically generates bus cycles to the coprocessor interface addresses.

<table>
<thead>
<tr>
<th>Table 4.8 Numeric Coprocessor Port Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address in 80376</strong></td>
</tr>
<tr>
<td>I/O Space</td>
</tr>
<tr>
<td>8000F8H</td>
</tr>
<tr>
<td>8000FCH</td>
</tr>
<tr>
<td>8000FEH</td>
</tr>
</tbody>
</table>

SOFTWARE TESTING FOR COPROCESSOR PRESENCE

When software is used to test coprocessor (80387SX) presence, it should use only the following coprocessor opcodes: FNINIT, FNSTCW and FNSTSW. To use other coprocessor opcodes when a coprocessor is known to be not present, first set EM = 1 in the 80376 CR0 register.

5.0 PACKAGE THERMAL SPECIFICATIONS

The Intel 80376 embedded processor is specified for operation when case temperature is within the range of 0°C–115°C for the ceramic 88-pin PGA package, and 0°C–110°C for the 100-pin plastic package. The case temperature may be measured in any environment, to determine whether the 80376 is within specified operating range. The case temperature should be measured at the center of the top surface.
The ambient temperature is guaranteed as long as $T_e$ is not violated. The ambient temperature can be calculated from the $\theta_{jc}$ and $\theta_{ja}$ from the following equations:

$$
T_j = T_e + P*\theta_{jc}
$$

$$
T_A = T_j - P*\theta_{ja}
$$

$$
T_C = T_A + P*[\theta_{ja} - \theta_{jc}]
$$

Values for $\theta_{ja}$ and $\theta_{ja}$ are given in Table 5.1 for the 100-lead fine pitch. $\theta_{ja}$ is given at various airflows. Table 5.2 shows the maximum $T_A$ allowable (without exceeding $T_C$) at various airflows. Note that $T_A$ can be improved further by attaching “fins” or a “heat sink” to the package. $P$ is calculated using the maximum $I_{CC}$.

### Table 5.1. 80376 Package Thermal Characteristics

<table>
<thead>
<tr>
<th>Package</th>
<th>$\theta_{ja}$ Versus Airflow-ft/min (m/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
</tr>
<tr>
<td>100-Lead Fine Pitch</td>
<td>7</td>
</tr>
<tr>
<td>88-Pin PGA</td>
<td>2</td>
</tr>
</tbody>
</table>

Assuming $I_{CC}$ hot of 360 mA, $V_{CC}$ of 5.0V, and a $T_{CASE}$ of 110°C for plastic and 115°C for the 88-Pin PGA Package:

### Table 5.2. 80376 Maximum Allowable Ambient Temperature at Various Airflows

<table>
<thead>
<tr>
<th>Package</th>
<th>$T_A$ (°C) vs Airflow-ft/min (m/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
</tr>
<tr>
<td>100-Lead Fine Pitch</td>
<td>7</td>
</tr>
<tr>
<td>88-Pin PGA</td>
<td>2</td>
</tr>
</tbody>
</table>

### 6.0 ELECTRICAL SPECIFICATIONS

The following sections describe recommended electrical connections for the 80376, and its electrical specifications.

#### 6.1 Power and Grounding

The 80376 is implemented in CHMOS III technology and has modest power requirements. However, its high clock frequency and 47 output buffers (address, data, control, and HLDA) can cause power surges as multiple output buffers drive new signal levels simultaneously. For clean on-chip power distribution at high frequency, 14 $V_{CC}$ and 18 $V_{SS}$ pins separately feed functional units of the 80376.

Power and ground connections must be made to all external $V_{CC}$ and GND pins of the 80376. On the circuit board, all $V_{CC}$ pins should be connected on a $V_{CC}$ plane and all $V_{SS}$ pins should be connected on a GND plane.

### POWER DECOUPLING RECOMMENDATIONS

Liberal decoupling capacitors should be placed near the 80376. The 80376 driving its 24-bit address bus and 16-bit data bus at high frequencies can cause transient power surges, particularly when driving large capacitive loads. Low inductance capacitors and interconnects are recommended for best high frequency electrical performance. Inductance can be reduced by shortening circuit board traces between the 80376 and decoupling capacitors as much as possible.

### RESISTOR RECOMMENDATIONS

The ERROR # and BUSY # inputs have internal pull-up resistors of approximately 20 KΩ and the PEREQ input has an internal pull-down resistor of approximately 20 KΩ built into the 80376 to keep these signals inactive when the 80387SX is not present in the system (or temporarily removed from its socket).
In typical designs, the external pull-up resistors shown in Table 6.1 are recommended. However, a particular design may have reason to adjust the resistor values recommended here, or alter the use of pull-up resistors in other ways.

Table 6.1. Recommended Resistor Pull-Ups to \( V_{CC} \)

<table>
<thead>
<tr>
<th>Pin</th>
<th>Signal</th>
<th>Pull-Up Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>ADS#</td>
<td>20 KΩ ± 10%</td>
<td>Lightly Pull ADS# Inactive during 80376 Hold Acknowledge States</td>
</tr>
<tr>
<td>26</td>
<td>LOCK#</td>
<td>20 KΩ ± 10%</td>
<td>Lightly Pull LOCK# Inactive during 80376 Hold Acknowledge States</td>
</tr>
</tbody>
</table>

OTHER CONNECTION RECOMMENDATIONS

For reliable operation, always connect unused inputs to an appropriate signal level. N/C pins should always remain unconnected. Connection of N/C pins to \( V_{CC} \) or \( V_{SS} \) will result in incompatibility with future stepping of the 80376.

Particularly when not using interrupts or bus hold (as when first prototyping), prevent any chance of spurious activity by connecting these associated inputs to GND:

- INTR
- NMI
- HOLD

If not using address pipelining connect the NA# pin to a pull-up resistor in the range of 20 KΩ to \( V_{CC} \).

6.2 Absolute Maximum Ratings

Table 6.2. Maximum Ratings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Maximum Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage Temperature</td>
<td>(-65°C ) \text{ to } (+150°C)</td>
</tr>
<tr>
<td>Case Temperature under Bias</td>
<td>(-65°C ) \text{ to } (+120°C)</td>
</tr>
<tr>
<td>Supply Voltage with Respect to ( V_{SS} )</td>
<td>(-0.5V ) \text{ to } (+6.5V)</td>
</tr>
<tr>
<td>Voltage on Other Pins</td>
<td>(-0.5V ) \text{ to } (( V_{CC} ) \text{ + } 0.5V)</td>
</tr>
</tbody>
</table>

Table 6.2 gives a stress ratings only, and functional operation at the maximums is not guaranteed. Functional operating conditions are given in Section 6.3, D.C. Specifications, and Section 6.4, A.C. Specifications.

Extended exposure to the Maximum Ratings may affect device reliability. Furthermore, although the 80376 contains protective circuitry to resist damage from static electric discharge, always take precautions to avoid high static voltages or electric fields.
# 6.3 D.C. Specifications

## Table 6.3: 80376 D.C. Characteristics

Table 6.3: 80376 D.C. Characteristics

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Min</th>
<th>Max</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{IL}$</td>
<td>Input LOW Voltage</td>
<td>-0.3</td>
<td>+0.8</td>
<td>V(1)</td>
</tr>
<tr>
<td>$V_{IH}$</td>
<td>Input HIGH Voltage</td>
<td>2.0</td>
<td>$V_{CC} + 0.3$</td>
<td>V(1)</td>
</tr>
<tr>
<td>$V_{ILC}$</td>
<td>CLK2 Input LOW Voltage</td>
<td>-0.3</td>
<td>+0.8</td>
<td>V(1)</td>
</tr>
<tr>
<td>$V_{IH2}$</td>
<td>CLK2 Input HIGH Voltage</td>
<td>$V_{CC} - 0.8$</td>
<td>$V_{CC} + 0.3$</td>
<td>V(1)</td>
</tr>
<tr>
<td>$V_{OL}$</td>
<td>Output LOW Voltage</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_{OL} = 4$ mA</td>
<td></td>
<td>0.45</td>
<td></td>
<td>V(1)</td>
</tr>
<tr>
<td>$I_{OL} = 5$ mA</td>
<td></td>
<td>0.45</td>
<td></td>
<td>V(1)</td>
</tr>
<tr>
<td>$V_{OH}$</td>
<td>Output High Voltage</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_{OH} = -1$ mA</td>
<td></td>
<td>2.4</td>
<td></td>
<td>V(1)</td>
</tr>
<tr>
<td>$I_{OH} = -0.2$ mA</td>
<td></td>
<td>$V_{CC} - 0.05$</td>
<td></td>
<td>V(1)</td>
</tr>
<tr>
<td>$I_{OH} = -0.9$ mA</td>
<td></td>
<td>2.4</td>
<td></td>
<td>V(1)</td>
</tr>
<tr>
<td>$I_{OH} = -0.18$ mA</td>
<td></td>
<td>$V_{CC} - 0.5$</td>
<td></td>
<td>V(1)</td>
</tr>
<tr>
<td>$I_{IL}$</td>
<td>Input Leakage Current</td>
<td>$\pm 15$</td>
<td>$\mu A$, $0V \leq V_{IN} \leq V_{CC}(1)$</td>
<td></td>
</tr>
<tr>
<td>$I_{IH}$</td>
<td>Input Leakage Current</td>
<td>200</td>
<td>$\mu A$, $V_{IH} = 2.4V(1, 2)$</td>
<td></td>
</tr>
<tr>
<td>$I_{IL}$</td>
<td>Input Leakage Current</td>
<td>-400</td>
<td>$\mu A$, $V_{IL} = 0.45V(3)$</td>
<td></td>
</tr>
<tr>
<td>$I_{IO}$</td>
<td>Output Leakage Current</td>
<td>$\pm 15$</td>
<td>$\mu A$, $0.45V \leq V_{OUT} \leq V_{CC}(1)$</td>
<td></td>
</tr>
<tr>
<td>$I_{CC}$</td>
<td>Supply Current</td>
<td>400</td>
<td>mA(4)</td>
<td></td>
</tr>
<tr>
<td>$C_{IN}$</td>
<td>Input Capacitance</td>
<td>10</td>
<td>pF, $F_C = 1$ MHz(6)</td>
<td></td>
</tr>
<tr>
<td>$C_{OUT}$</td>
<td>Output or I/O Capacitance</td>
<td>12</td>
<td>pF, $F_C = 1$ MHz(6)</td>
<td></td>
</tr>
<tr>
<td>$C_{CLK}$</td>
<td>CLK2 Capacitance</td>
<td>20</td>
<td>pF, $F_C = 1$ MHz(6)</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

1. Tested at the minimum operating frequency of the part.
2. PEREQ input has an internal pull-down resistor.
3. BUSY # and ERROR # inputs each have an internal pull-up resistor.
4. $I_{CC}$ max measurement at worse case frequency, $V_{CC}$ and temperature (0°C).
5. Not 100% tested.
6. $I_{CC}$ HOT max measurement at worse case frequency, $V_{CC}$ and max temperature.
The A.C. specifications given in Table 6.4 consist of output delays, input setup requirements and input hold requirements. All A.C. specifications are relative to the CLK2 rising edge crossing the 2.0V level.

A.C. specification measurement is defined by Figure 6.1. Inputs must be driven to the voltage levels indicated by Figure 6.1 when A.C. specifications are measured. 80376 output delays are specified with minimum and maximum limits measured as shown. The minimum 80376 delay times are hold times provided to external circuitry. 80376 input setup and hold times are specified as minimums, defining the smallest acceptable sampling window. Within the sampling window, a synchronous input signal must be stable for correct 80376 processor operation.

Outputs NA#, W/R#, D/C#, M/IO#, LOCK#, BHE#, BLE#, A23–A1 and HLDA only change at the beginning of phase one. D15–D0 (write cycles) only change at the beginning of phase two. The READY#, HOLD, BUSY#, ERROR#, PEREQ and D15–D0 (read cycles) inputs are sampled at the beginning of phase one. The NA#, INTR and NMI inputs are sampled at the beginning of phase two.

**Figure 6.1. Drive Levels and Measurement Points for A.C. Specifications**
### 6.4 A.C. Specifications

**ADVANCE INFORMATION SUBJECT TO CHANGE**

Table 6.4. 80376 A.C. Characteristics at 16 MHz

Functional Operating Range: \( V_{CC} = 5V \pm 10\% \); \( T_{CASE} = 0^\circ C \) to 115°C for 88-pin PGA, 0°C to 110°C for 100-pin plastic

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Min</th>
<th>Max</th>
<th>Unit</th>
<th>Figure</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>t(_1)</td>
<td>Operating Frequency</td>
<td>4</td>
<td>16</td>
<td>MHz</td>
<td>6.3</td>
<td>half CLK2 freq</td>
</tr>
<tr>
<td>t(_2a)</td>
<td>CLK2 HIGH Time</td>
<td>9</td>
<td>ns</td>
<td>6.3</td>
<td>( t_2a )</td>
<td></td>
</tr>
<tr>
<td>t(_2b)</td>
<td>CLK2 HIGH Time</td>
<td>5</td>
<td>ns</td>
<td>6.3</td>
<td>( t_2b )</td>
<td></td>
</tr>
<tr>
<td>t(_3a)</td>
<td>CLK2 LOW Time</td>
<td>9</td>
<td>ns</td>
<td>6.3</td>
<td>( t_3a )</td>
<td></td>
</tr>
<tr>
<td>t(_3b)</td>
<td>CLK2 LOW Time</td>
<td>7</td>
<td>ns</td>
<td>6.3</td>
<td>( t_3b )</td>
<td></td>
</tr>
<tr>
<td>t(_4)</td>
<td>CLK2 Fall Time</td>
<td>8</td>
<td>ns</td>
<td>6.6</td>
<td>( t_4 )</td>
<td></td>
</tr>
<tr>
<td>t(_5)</td>
<td>CLK2 Rise Time</td>
<td>8</td>
<td>ns</td>
<td>6.6</td>
<td>( t_5 )</td>
<td></td>
</tr>
<tr>
<td>t(_6)</td>
<td>A(_{23}-A_1) Valid Delay</td>
<td>4</td>
<td>36</td>
<td>ns</td>
<td>6.5</td>
<td>( t_6 )</td>
</tr>
<tr>
<td>t(_7)</td>
<td>A(_{23}-A_1) Float Delay</td>
<td>1</td>
<td>40</td>
<td>ns</td>
<td>6.6</td>
<td>(1)</td>
</tr>
<tr>
<td>t(_8)</td>
<td>BHE#, BLE#, LOCK# Valid Delay</td>
<td>3</td>
<td>36</td>
<td>ns</td>
<td>6.5</td>
<td>( t_8 )</td>
</tr>
<tr>
<td>t(_9)</td>
<td>BHE#, BLE#, LOCK# Float Delay</td>
<td>4</td>
<td>46</td>
<td>ns</td>
<td>6.6</td>
<td>(1)</td>
</tr>
<tr>
<td>t(_10)</td>
<td>W/R#, M/IO#, D/C#, ADS# Valid Delay</td>
<td>6</td>
<td>33</td>
<td>ns</td>
<td>6.5</td>
<td>( t_{10} )</td>
</tr>
<tr>
<td>t(_11)</td>
<td>W/R#, M/IO#, D/C#, ADS# Float Delay</td>
<td>6</td>
<td>35</td>
<td>ns</td>
<td>6.6</td>
<td>(1)</td>
</tr>
<tr>
<td>t(_12)</td>
<td>D(_{15}-D_0) Write Data Valid Delay</td>
<td>4</td>
<td>40</td>
<td>ns</td>
<td>6.5</td>
<td>( t_{12} )</td>
</tr>
<tr>
<td>t(_13)</td>
<td>D(_{15}-D_0) Write Data Float Delay</td>
<td>4</td>
<td>35</td>
<td>ns</td>
<td>6.6</td>
<td>(1)</td>
</tr>
<tr>
<td>t(_14)</td>
<td>HLDA Valid Delay</td>
<td>6</td>
<td>33</td>
<td>ns</td>
<td>6.6</td>
<td>( t_{14} )</td>
</tr>
<tr>
<td>t(_15)</td>
<td>NA# Setup Time</td>
<td>5</td>
<td>ns</td>
<td>6.6</td>
<td>( t_{15} )</td>
<td></td>
</tr>
<tr>
<td>t(_16)</td>
<td>NA# Hold Time</td>
<td>21</td>
<td>ns</td>
<td>6.6</td>
<td>( t_{16} )</td>
<td></td>
</tr>
<tr>
<td>t(_19)</td>
<td>READY# Setup Time</td>
<td>19</td>
<td>ns</td>
<td>6.6</td>
<td>( t_{19} )</td>
<td></td>
</tr>
<tr>
<td>t(_20)</td>
<td>READY# Hold Time</td>
<td>4</td>
<td>ns</td>
<td>6.6</td>
<td>( t_{20} )</td>
<td></td>
</tr>
<tr>
<td>t(_21)</td>
<td>Setup Time D(_{15}-D_0) Read Data</td>
<td>9</td>
<td>ns</td>
<td>6.6</td>
<td>( t_{21} )</td>
<td></td>
</tr>
<tr>
<td>t(_22)</td>
<td>Hold Time D(_{15}-D_0) Read Data</td>
<td>6</td>
<td>ns</td>
<td>6.6</td>
<td>( t_{22} )</td>
<td></td>
</tr>
<tr>
<td>t(_23)</td>
<td>HOLD Setup Time</td>
<td>26</td>
<td>ns</td>
<td>6.6</td>
<td>( t_{23} )</td>
<td></td>
</tr>
<tr>
<td>t(_24)</td>
<td>HOLD Hold Time</td>
<td>5</td>
<td>ns</td>
<td>6.6</td>
<td>( t_{24} )</td>
<td></td>
</tr>
<tr>
<td>t(_25)</td>
<td>RESET Setup Time</td>
<td>13</td>
<td>ns</td>
<td>6.7</td>
<td>( t_{25} )</td>
<td></td>
</tr>
<tr>
<td>t(_26)</td>
<td>RESET Hold Time</td>
<td>4</td>
<td>ns</td>
<td>6.7</td>
<td>( t_{26} )</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:**
The 80376 does not have \( t_{17} \) or \( t_{18} \) timing specifications.
Table 6.4. 80376 A.C. Characteristics at 16 MHz

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Min</th>
<th>Max</th>
<th>Unit</th>
<th>Figure</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>t27</td>
<td>NMI, INTR Setup Time</td>
<td>16</td>
<td></td>
<td>ns</td>
<td>6.4</td>
<td>(2)</td>
</tr>
<tr>
<td>t28</td>
<td>NMI, INTR Hold Time</td>
<td>16</td>
<td></td>
<td>ns</td>
<td>6.4</td>
<td>(2)</td>
</tr>
<tr>
<td>t29</td>
<td>PEREQ, ERROR #, BUSY # Setup Time</td>
<td>16</td>
<td></td>
<td>ns</td>
<td>6.4</td>
<td>(2)</td>
</tr>
<tr>
<td>t30</td>
<td>PEREQ, ERROR #, BUSY # Hold Time</td>
<td>5</td>
<td></td>
<td>ns</td>
<td>6.4</td>
<td>(2)</td>
</tr>
</tbody>
</table>

NOTES:
1. Float condition occurs when maximum output current becomes less than ILO in magnitude. Float delay is not 100% tested.
2. These inputs are allowed to be asynchronous to CLK2. The setup and hold specifications are given for testing purposes, to assure recognition within a specific CLK2 period.
3. These are not tested. They are guaranteed by design characterization.
4. Tested with CL set to 50 pF and derated to support the indicated distributed capacitive load. See Figure 6.8 for the capacitive derating curve.

A.C. TEST LOADS

![A.C. Test Loads Diagram](240182-36)

Figure 6.2. A.C. Test Loads

A.C. TIMING WAVEFORMS

![CLK2 Waveform](240182-37)

Figure 6.3. CLK2 Waveform

![A.C. Timing Waveforms—Input Setup and Hold Timing](240182-38)

Figure 6.4. A.C. Timing Waveforms—Input Setup and Hold Timing
Figure 6.5. A.C. Timing Waveforms—Output Valid Delay Timing

Figure 6.6. A.C. Timing Waveforms—Output Float Delay and HLDA Valid Delay Timing
The second internal processor phase following RESET high-to-low transition (provided $t_{25}$ and $t_{26}$ are met) is $\Phi_2$.

**Figure 6.7. A.C. Timing Waveforms—RESET Setup and Hold Timing, and Internal Phase**

**Figure 6.8. Capacitive Derating Curve**

**Figure 6.9. CMOS Level Slew Rates for Output Buffers**

**Figure 6.10. TTL Level Slew Rates for Output Buffers**

**Figure 6.11. Typical $I_{CC}$ vs Frequency**
6.5 Designing for ICE™-376 Emulator (Advanced Data)

The 376 embedded processor in-circuit emulator product is the ICE-376 emulator. Use of the emulator requires the target system to provide a socket that is compatible with the ICE-376 emulator. The 80376 offers two different probes for emulating user systems: an 88-pin PGA probe and a 100-pin fine pitch flat-pack probe. The 100-pin fine pitch flat-pack probe requires a socket, called the 100-pin PQFP, which is available from 3-M text-tool (part number 2-0100-07243-000). The ICE-376 emulator probe attaches to the target system via an adapter which replaces the 80376 component in the target system. Because of the high operating frequency of 80376 systems and of the ICE-376 emulator, there is no buffering between the 80376 emulation processor in the ICE-376 emulator probe and the target system. A direct result of the non-buffered interconnect is that the ICE-376 emulator shares the address and data bus with the user's system, and the RESET signal is intercepted by the ICE emulator hardware. In order for the ICE-376 emulator to be functional in the user's system without the Optional Isolation Board (OIB) the designer must be aware of the following conditions:

1. The bus controller must only enable data transceivers onto the data bus during valid read cycles of the 80376, other local devices or other bus masters.

2. Before another bus master drives the local processor address bus, the other master must gain control of the address bus by asserting HOLD and receiving the HLDA response.

3. The emulation processor receives the RESET signal 2 or 4 CLK2 cycles later than an 80376 would, and responds to RESET later. Correct phase of the response is guaranteed.

In addition to the above considerations, the ICE-376 emulator processor module has several electrical and mechanical characteristics that should be taken into consideration when designing the 80376 system.

Capacitive Loading: ICE-376 adds up to 27 pF to each 80376 signal.

Drive Requirements: ICE-376 adds one FAST TTL load on the CLK2, control, address, and data lines. These loads are within the processor module and are driven by the 80376 emulation processor, which has standard drive and loading capability listed in Tables 6.3 and 6.4.

Power Requirements: For noise immunity and CMOS latch-up protection the ICE-376 emulator processor module is powered by the user system. The circuitry on the processor module draws up to 1.4A including the maximum 80376 ICC from the user 80376 socket.

80376 Location and Orientation: The ICE-376 emulator processor module may require lateral clearance. Figure 6.12 shows the clearance requirements of the IMP adapter and Figure 6.13 shows the clearance requirements of the 88-pin PGA adapter.

![Figure 6.12. Preliminary ICE™-376 Emulator User Cable with PQFP Adapter](image)
optional isolation board (OIB), which provides extra electrical buffering and has the same lateral clearance requirements as Figures 6.12 and 6.13, adds an additional 0.5 inches to the vertical clearance requirement. This is illustrated in Figure 6.14.

Optional Isolation Board (OIB) and the CLK2 speed reduction: Due to the unbuffered probe design, the ICE-376 emulator is susceptible to errors on the user’s bus. The OIB allows the ICE-376 emulator to function in user systems with faults (shorted signals, etc.). After electrical verification the OIB may be removed. When the OIB is installed, the user system must have a maximum CLK2 frequency of 20 MHz.
7.0 DIFFERENCES BETWEEN THE 80376 AND THE 80386

The following are the major differences between the 80376 and the 80386.

1. The 80376 generates byte selects on BHE# and BLE# (like the 8086 and 80286 microprocessors) to distinguish the upper and lower bytes on its 16-bit data bus. The 80386 uses four-byte selects, BE0#–BE3#, to distinguish between the different bytes on its 32-bit bus.

2. The 80376 has no bus sizing option. The 80386 can select between either a 32-bit bus or a 16-bit bus by use of the BS16# input. The 80376 has a 16-bit bus size.

3. The NA# pin operation in the 80376 is identical to that of the NA# pin on the 80386 with one exception: the NA# pin of the 80386 cannot be activated on 16-bit bus cycles (where BS16# is LOW in the 80386 case), whereas NA# can be activated on any 80376 bus cycle.

4. The contents of all 80376 registers at reset are identical to the contents of the 80386 registers at reset, except the DX register. The DX register contains a component-stepping identifier at reset, i.e.

   in 80386, after reset DH = 3 indicates 80386
   DL = revision number;
   in 80376, after reset DH = 33H indicates 80376
   DL = revision number.

5. The 80386 uses A31 and M/IO# as a select for numerics coprocessor. The 80376 uses the A23 and M/IO# to select its numerics coprocessor.

6. The 80386 prefetch unit fetches code in four-byte units. The 80376 prefetch unit reads two bytes as one unit (like the 80286 microprocessor). In BS16# mode, the 80386 takes two consecutive bus cycles to complete a prefetch request. If there is a data read or write request after the prefetch starts, the 80386 will fetch all four bytes before addressing the new request.

7. The 80376 has no paging mechanism.

8. The 80376 starts executing code in what corresponds to the 80386 protected mode. The 80386 starts execution in real mode, which is then used to enter protected mode.

9. The 80386 has a virtual-86 mode that allows the execution of a real mode 8086 program as a task in protected mode. The 80376 has no virtual-86 mode.

10. The 80386 maps a 48-bit logical address into a 32-bit physical address by segmentation and paging. The 80376 maps its 48-bit logical address into a 24-bit physical address by segmentation only.

11. The 80376 uses the 80387SX numerics coprocessor for floating point operations, while the 80386 uses the 80387 coprocessor.

12. The 80386 can execute from 16-bit code segments. The 80376 can only execute from 32-bit code Segments.

8.0 INSTRUCTION SET

This section describes the 376 embedded processor instruction set. Table 8.1 lists all instructions along with instruction encoding diagrams and clock counts. Further details of the instruction encoding are then provided in the following sections, which completely describe the encoding structure and the definition of all fields occurring within 80376 instructions.

8.1 80376 Instruction Encoding and Clock Count Summary

To calculate elapsed time for an instruction, multiply the instruction clock count, as listed in Table 8.1 below, by the processor clock period (e.g. 62.5 ns for an 80376 operating at 16 MHz). The actual clock count of an 80376 program will average 10% more
than the calculated clock count due to instruction sequences which execute faster than they can be fetched from memory.

**Instruction Clock Count Assumptions:**
1. The instruction has been prefetched, decoded, and is ready for execution.
2. Bus cycles do not require wait states.
3. There are no local bus HOLD requests delaying processor access to the bus.
4. No exceptions are detected during instruction execution.
5. If an effective address is calculated, it does not use two general register components. One register, scaling and displacement can be used within the clock counts shown. However, if the effective address calculation uses two general register components, add 1 clock to the clock count shown.
6. Memory reference instruction accesses byte or aligned 16-bit operands.

**Instruction Clock Count Notation**
- If two clock counts are given, the smaller refers to a register operand and the larger refers to a memory operand.

---

- **n** = number of times repeated.
- **m** = number of components in the next instruction executed, where the entire displacement (if any) counts as one component, the entire immediate data (if any) counts as one component, and all other bytes of the instruction and prefix(es) each count as one component.

**Misaligned or 32-Bit Operand Accesses:**
- If instructions accesses a misaligned 16-bit operand or 32-bit operand on even address add:
  - 2* clocks for read or write.
  - 4** clocks for read and write.
- If instructions accesses a 32-bit operand on odd address add:
  - 4* clocks for read or write.
  - 8** clocks for read and write.

**Wait States:**
Wait states add 1 clock per wait state to instruction execution for each data access.
Table 8.1. 80376 Instruction Set Clock Count Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock Counts</th>
<th>Number of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GENERAL DATA TRANSFER</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV = Move:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register to Register/Memory</td>
<td>000100 w mod reg r/m</td>
<td>2/2*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td>Register/Memory to Register</td>
<td>000101 w mod reg r/m</td>
<td>2/4*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td>Immediate to Register/Memory</td>
<td>110011 w mod 000 r/m</td>
<td>2/1*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td>Immediate to Register (Short Form)</td>
<td>1011 w reg r/0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory to Accumulator (Short Form)</td>
<td>101000 w disp full</td>
<td>4*</td>
<td>1*</td>
<td>a</td>
</tr>
<tr>
<td>Accumulator to Memory (Short Form)</td>
<td>101000 w disp full</td>
<td>2</td>
<td>1*</td>
<td>a</td>
</tr>
<tr>
<td>Register/Memory to Segment Register</td>
<td>10001110 mod sreg3 r/m</td>
<td>2/2*</td>
<td>0/6*</td>
<td>a,b,c</td>
</tr>
<tr>
<td>Segment Register to Register/ Memory</td>
<td>10001100 mod sreg3 r/m</td>
<td>2/2*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td><strong>MOVSX = Move with Sign Extension</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register from Register/ Memory</td>
<td>00101111 1011111 w mod reg r/m</td>
<td>3/6*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td><strong>MOVZX = Move with Zero Extension</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register from Register/ Memory</td>
<td>00101111 1011111 w mod reg r/m</td>
<td>3/6*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td><strong>PUSH = Push</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/ Memory</td>
<td>111111 mod 110 r/m</td>
<td>7/9*</td>
<td>2/4*</td>
<td>a</td>
</tr>
<tr>
<td>Register (Short Form)</td>
<td>010100 reg</td>
<td>4</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>Segment Register (ES, SS, or DF)</td>
<td>000 sreg 110</td>
<td>4</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>Segment Register (FS or GS)</td>
<td>00001111 10 sreg3001</td>
<td>4</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>Immediate</td>
<td>01110000</td>
<td>4</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td><strong>PUSHA = Push All</strong></td>
<td>01100000</td>
<td>34</td>
<td>16</td>
<td>a</td>
</tr>
<tr>
<td><strong>POP = Pop</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/ Memory</td>
<td>00011111 mod 000 r/m</td>
<td>7/8*</td>
<td>2/4*</td>
<td>a</td>
</tr>
<tr>
<td>Register (Short Form)</td>
<td>010110 reg</td>
<td>6</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>Segment Register (ES, SS, or DF)</td>
<td>000 sreg 2111</td>
<td>25</td>
<td>6</td>
<td>a,b,c</td>
</tr>
<tr>
<td>Segment Register (FS or GS)</td>
<td>00001111 10 sreg3001</td>
<td>25</td>
<td>6</td>
<td>a,b,c</td>
</tr>
<tr>
<td>POPA = Pop All</td>
<td>01100001</td>
<td>40</td>
<td>16</td>
<td>a</td>
</tr>
<tr>
<td><strong>XCHG = Exchange</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/ Memory with Register</td>
<td>10000111 w mod reg r/m</td>
<td>3/5**</td>
<td>0/2**</td>
<td>a,m</td>
</tr>
<tr>
<td>Register with Accumulator (Short Form)</td>
<td>10010 reg</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>IN = Input from:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed Port</td>
<td>1110110 w port number</td>
<td>6*</td>
<td>1*</td>
<td>f,k</td>
</tr>
<tr>
<td>Variable Port</td>
<td>1110110 w</td>
<td>26*</td>
<td>1*</td>
<td>f,l</td>
</tr>
<tr>
<td><strong>OUT = Output to:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed Port</td>
<td>11100111 w port number</td>
<td>4*</td>
<td>1*</td>
<td>f,k</td>
</tr>
<tr>
<td>Variable Port</td>
<td>1110111 w</td>
<td>24*</td>
<td>1*</td>
<td>f,l</td>
</tr>
<tr>
<td>LEA = Load EA to Register</td>
<td>10001101 mod reg r/m</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a, b, c are valid for 80386, 80387, or 80386A, respectively.
Table 8.1. 80376 Instruction Set Clock Count Summary (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock</th>
<th>Number of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGMENT CONTROL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDS = Load Pointer to DS</td>
<td>11000101 mod reg r/m</td>
<td>26*</td>
<td>6*</td>
<td>a, b, c</td>
</tr>
<tr>
<td>LES = Load Pointer to ES</td>
<td>11000100 mod reg r/m</td>
<td>26*</td>
<td>6*</td>
<td>a, b, c</td>
</tr>
<tr>
<td>LFS = Load Pointer to FS</td>
<td></td>
<td>26*</td>
<td>6*</td>
<td>a, b, c</td>
</tr>
<tr>
<td>LGS = Load Pointer to GS</td>
<td></td>
<td>26*</td>
<td>6*</td>
<td>a, b, c</td>
</tr>
<tr>
<td>LSS = Load Pointer to SS</td>
<td></td>
<td>26*</td>
<td>6*</td>
<td>a, b, c</td>
</tr>
<tr>
<td>FLAG CONTROL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLC = Clear Carry Flag</td>
<td>11111000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLD = Clear Direction Flag</td>
<td>11111000</td>
<td>8</td>
<td></td>
<td>f</td>
</tr>
<tr>
<td>CLI = Clear Interrupt Enable Flag</td>
<td>11110101</td>
<td>6</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>CLTS = Clear Task Switched Flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMC = Complement Carry Flag</td>
<td>11110101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAHF = Load AH Into Flag</td>
<td>10011111</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POPF = Pop Flags</td>
<td>10011101</td>
<td>5</td>
<td></td>
<td>a, g</td>
</tr>
<tr>
<td>PUSHF = Push Flags</td>
<td>10011100</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAHF = Store AH Into Flags</td>
<td>10011110</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STC = Set Carry Flag</td>
<td>11111001</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STD = Set Direction Flag</td>
<td>11111011</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STI = Set Interrupt Enable Flag</td>
<td>11111011</td>
<td>8</td>
<td></td>
<td>f</td>
</tr>
<tr>
<td>ARITHMETIC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD = Add</td>
<td>00000d mod reg r/m</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register to Register</td>
<td>00000000 mod reg r/m</td>
<td>7**</td>
<td>2**</td>
<td>a</td>
</tr>
<tr>
<td>Register to Memory</td>
<td>0000001w mod reg r/m</td>
<td>6*</td>
<td>1*</td>
<td>a</td>
</tr>
<tr>
<td>Memory to Register</td>
<td>00000000 mod reg r/m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate to Register/Memory</td>
<td>1000000w mod 000 r/m</td>
<td>2/7**</td>
<td>0/2**</td>
<td>a</td>
</tr>
<tr>
<td>Immediate to Accumulator</td>
<td>0000100w</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Short Form)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC = Add with Carry</td>
<td>0001000 mod reg r/m</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register to Register</td>
<td>00010000 mod reg r/m</td>
<td>7**</td>
<td>2**</td>
<td>a</td>
</tr>
<tr>
<td>Register to Memory</td>
<td>00010001 mod reg r/m</td>
<td>6*</td>
<td>1*</td>
<td>a</td>
</tr>
<tr>
<td>Memory to Register</td>
<td>00010000 mod reg r/m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate to Register/Memory</td>
<td>1000000w mod 010 r/m</td>
<td>2/7**</td>
<td>0/2**</td>
<td>a</td>
</tr>
<tr>
<td>Immediate to Accumulator</td>
<td>0001010w</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Short Form)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC = Increment</td>
<td>1111111w mod 000 r/m</td>
<td>2/6**</td>
<td>0/2**</td>
<td>a</td>
</tr>
<tr>
<td>Register/Memory</td>
<td>11111111w mod 000 r/m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB = Subtract</td>
<td>011000 reg</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:
- a, b, c indicate specific conditions or notes related to the instructions.
### Table 8.1 80376 Instruction Set Clock Count Summary (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock Counts</th>
<th>Number Of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARITHMETIC (Continued)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register from Memory</td>
<td>0010100 w mod reg r/m</td>
<td>7**</td>
<td>2**</td>
<td>a</td>
</tr>
<tr>
<td>Memory from Register</td>
<td>0010101 w mod reg r/m</td>
<td>6**</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>Immediate from Register/Memory</td>
<td>100000 s w mod 1 1 r/m</td>
<td>2/7**</td>
<td>0/1**</td>
<td>a</td>
</tr>
<tr>
<td>Immediate from Accumulator (Short Form)</td>
<td>0010110 w</td>
<td>immediate data</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SBB = Subtract with Borrow</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register from Register</td>
<td>000110 d w mod reg r/m</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register from Memory</td>
<td>001100 w mod reg r/m</td>
<td>7**</td>
<td>2**</td>
<td>a</td>
</tr>
<tr>
<td>Memory from Register</td>
<td>0001101 w mod reg r/m</td>
<td>1**</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>Immediate from Register/Memory</td>
<td>100000 s w mod 1 1 r/m</td>
<td>2/7**</td>
<td>0/2**</td>
<td>a</td>
</tr>
<tr>
<td>Immediate from Accumulator (Short Form)</td>
<td>0011110 w</td>
<td>immediate data</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DEC = Decrement</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/Memory</td>
<td>1111111 w reg 00 1</td>
<td>2/6**</td>
<td>0/2**</td>
<td>a</td>
</tr>
<tr>
<td>Register (Short Form)</td>
<td>01001 reg</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CMP = Compare</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register with Register</td>
<td>001100 d w mod reg r/m</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory with Register</td>
<td>001100 w mod reg r/m</td>
<td>5**</td>
<td>1**</td>
<td>a</td>
</tr>
<tr>
<td>Register with Memory</td>
<td>0001101 w mod reg r/m</td>
<td>6**</td>
<td>2**</td>
<td>a</td>
</tr>
<tr>
<td>Immediate with Register/Memory</td>
<td>100000 s w mod 1 1 r/m</td>
<td>2/5**</td>
<td>0/1**</td>
<td>a</td>
</tr>
<tr>
<td>Immediate with Accumulator (Short Form)</td>
<td>0011100 w</td>
<td>immediate data</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NEG = Change Sign</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/Memory</td>
<td>1111111 w mod 0 0 1</td>
<td>2/6**</td>
<td>0/2**</td>
<td>a</td>
</tr>
<tr>
<td><strong>AAA = ASCII Adjust for Add</strong></td>
<td>01110 1 1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AAS = ASCII Adjust for Subtract</strong></td>
<td>0011111 1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DAA = Decimal Adjust for Add</strong></td>
<td>001001 1 1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DAS = Decimal Adjust for Subtract</strong></td>
<td>001011 1 1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MUL = Multiply (Unsigned)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accumulator with Register/Memory</td>
<td>1111011 w mod 1 0 0 r/m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplier—Byte</td>
<td></td>
<td>12-17/15-20</td>
<td>0/1</td>
<td>a,n</td>
</tr>
<tr>
<td>—Word</td>
<td></td>
<td>12-25/15-28*</td>
<td>0/1*</td>
<td>a,n</td>
</tr>
<tr>
<td>—Doubleword</td>
<td></td>
<td>12-41/17-46*</td>
<td>0/2*</td>
<td>a,n</td>
</tr>
<tr>
<td><strong>IMUL = Integer Multiply (Signed)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accumulator with Register/Memory</td>
<td>1111011 w mod 1 0 1 r/m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplier—Byte</td>
<td></td>
<td>12-17/15-20</td>
<td>0/1</td>
<td>a,n</td>
</tr>
<tr>
<td>—Word</td>
<td></td>
<td>12-25/15-28*</td>
<td>0/1*</td>
<td>a,n</td>
</tr>
<tr>
<td>—Doubleword</td>
<td></td>
<td>12-41/17-46*</td>
<td>0/2*</td>
<td>a,n</td>
</tr>
<tr>
<td>Register with Register/Memory</td>
<td>0001111 1 1 0 1 1 1 1 mod reg r/m</td>
<td>12-17/15-20</td>
<td>0/1</td>
<td>a,n</td>
</tr>
<tr>
<td>Multiplier—Byte</td>
<td></td>
<td>12-25/15-28*</td>
<td>0/1*</td>
<td>a,n</td>
</tr>
<tr>
<td>—Word</td>
<td></td>
<td>12-41/17-46*</td>
<td>0/2*</td>
<td>a,n</td>
</tr>
<tr>
<td>Register/Memory with Immediate to Register</td>
<td>011010 s 1 mod reg r/m</td>
<td>immediate data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>—Word</td>
<td></td>
<td>13-26/14-27*</td>
<td>0/1*</td>
<td>a,n</td>
</tr>
<tr>
<td>—Doubleword</td>
<td></td>
<td>13-42/16-45*</td>
<td>0/2*</td>
<td>a,n</td>
</tr>
</tbody>
</table>
Table 8.1. 80376 Instruction Set Clock Count Summary (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock Counts</th>
<th>Number Of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARITHMETIC (Continued)</td>
<td>DIV = Divide (Unsigned)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accumulator by Register/Memory</td>
<td>11110111w mod 110 r/m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divisor—Byte</td>
<td></td>
<td>15/17</td>
<td>0/1</td>
<td>a, o</td>
</tr>
<tr>
<td>—Word</td>
<td></td>
<td>22/25*</td>
<td>0/1*</td>
<td>a, o</td>
</tr>
<tr>
<td>—Doubleword</td>
<td></td>
<td>58/43*</td>
<td>0/2*</td>
<td>a, o</td>
</tr>
<tr>
<td>IDIV = Integer Divide (Signed)</td>
<td>Accumulator by Register/Memory</td>
<td>11110111w mod 111 r/m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divisor—Byte</td>
<td></td>
<td>15/22</td>
<td>0/1</td>
<td>a, o</td>
</tr>
<tr>
<td>—Word</td>
<td></td>
<td>27/30*</td>
<td>0/1*</td>
<td>a, o</td>
</tr>
<tr>
<td>—Doubleword</td>
<td></td>
<td>43/48*</td>
<td>0/2*</td>
<td>a, o</td>
</tr>
<tr>
<td>AAD = ASCII Adjust for Divide</td>
<td></td>
<td>11010101 00001010</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>AAM = ASCII Adjust for Multiply</td>
<td></td>
<td>11010100 00001010</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>CBW = Convert Byte to Word</td>
<td></td>
<td>10011000</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>CWD = Convert Word to Double Word</td>
<td></td>
<td>10011001</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>LOGIC</td>
<td>Shift Rotate Instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not Through Carry (ROL, ROR, SAL, SAR, SHL, and SHR)</td>
<td>Register/Memory by 1</td>
<td>1101000w mod 1TT r/m</td>
<td>3/7**</td>
<td>0/2**</td>
</tr>
<tr>
<td>Register/Memory by CL</td>
<td></td>
<td>1101001w mod 1TT r/m</td>
<td>3/7**</td>
<td>0/2**</td>
</tr>
<tr>
<td>Register/Memory by Immediate Count</td>
<td></td>
<td>1100000w mod 1TT r/m</td>
<td>3/7**</td>
<td>0/2**</td>
</tr>
<tr>
<td>Through Carry (RCL and RCR)</td>
<td>Register/Memory by 1</td>
<td>1101000w mod 1TT r/m</td>
<td>9/10**</td>
<td>0/2**</td>
</tr>
<tr>
<td>Register/Memory by CL</td>
<td></td>
<td>1101001w mod 1TT r/m</td>
<td>9/10**</td>
<td>10/2**</td>
</tr>
<tr>
<td>Register/Memory by Immediate Count</td>
<td></td>
<td>110000w mod 1TT r/m</td>
<td>9/10**</td>
<td>0/2**</td>
</tr>
<tr>
<td>T T T Instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>ROL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>ROR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>RCL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>RCR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>SHL/SAL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>SHR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>SAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHLD = Shift Left Double</td>
<td>Register/Memory by Immediate</td>
<td>000011111 10100100 mod reg r/m</td>
<td>3/7**</td>
<td>0/2**</td>
</tr>
<tr>
<td>Register/Memory by CL</td>
<td></td>
<td>000011111 10100101 mod reg r/m</td>
<td>3/7**</td>
<td>0/2**</td>
</tr>
<tr>
<td>SHRD = Shift Right Double</td>
<td>Register/Memory by Immediate</td>
<td>000011111 10101100 mod reg r/m</td>
<td>3/7**</td>
<td>0/2**</td>
</tr>
<tr>
<td>Register/Memory by CL</td>
<td></td>
<td>000011111 10101101 mod reg r/m</td>
<td>3/7**</td>
<td>0/2**</td>
</tr>
<tr>
<td>AND = And</td>
<td>Register to Register</td>
<td>0010000w mod reg r/m</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Format</td>
<td>Clock Counts</td>
<td>Number of Data Cycles</td>
<td>Notes</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------</td>
<td>--------------</td>
<td>-----------------------</td>
<td>-------</td>
</tr>
<tr>
<td>LOGIC (Continued)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register to Memory</td>
<td>00100000w mod reg r/m</td>
<td>7*</td>
<td>2**</td>
<td>a</td>
</tr>
<tr>
<td>Memory to Register</td>
<td>00100011w mod reg r/m</td>
<td>6*</td>
<td>1*</td>
<td>a</td>
</tr>
<tr>
<td>Immediate to Register/Memory</td>
<td>10000000w mod 100 r/m</td>
<td>Immediate data</td>
<td>2/7**</td>
<td>0/2**</td>
</tr>
<tr>
<td>Immediate to Accumulator (Short Form)</td>
<td>0010010w immediate data</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST = And function to Flags, No Result</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/Memory and Register</td>
<td>1000010w mod reg r/m</td>
<td>2/5*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td>Immediate Data and Register/Memory</td>
<td>1111011w mod 000 r/m</td>
<td>Immediate data</td>
<td>2/5*</td>
<td>0/1*</td>
</tr>
<tr>
<td>Immediate Data and Accumulator (Short Form)</td>
<td>1010100w immediate data</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR = Or</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register to Register</td>
<td>0000100w mod reg r/m</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register to Memory</td>
<td>00001001w mod reg r/m</td>
<td>7**</td>
<td>2**</td>
<td>a</td>
</tr>
<tr>
<td>Memory to Register</td>
<td>00001002w mod reg r/m</td>
<td>6*</td>
<td>1*</td>
<td>a</td>
</tr>
<tr>
<td>Immediate to Register/Memory</td>
<td>10000000w mod 010 r/m</td>
<td>Immediate data</td>
<td>2/7**</td>
<td>0/2**</td>
</tr>
<tr>
<td>Immediate to Accumulator (Short Form)</td>
<td>0000110w immediate data</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR = Exclusive Or</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register to Register</td>
<td>0011000w mod reg r/m</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register to Memory</td>
<td>00110001w mod reg r/m</td>
<td>7**</td>
<td>2**</td>
<td>a</td>
</tr>
<tr>
<td>Memory to Register</td>
<td>00110002w mod reg r/m</td>
<td>6*</td>
<td>1*</td>
<td>a</td>
</tr>
<tr>
<td>Immediate to Register/Memory</td>
<td>10000000w mod 110 r/m</td>
<td>Immediate data</td>
<td>2/7**</td>
<td>0/2**</td>
</tr>
<tr>
<td>Immediate to Accumulator (Short Form)</td>
<td>0011010w immediate data</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT = Invert Register/Memory</td>
<td>1111011w mod 010 r/m</td>
<td>2/6**</td>
<td>0/2**</td>
<td>a</td>
</tr>
<tr>
<td>STRING MANIPULATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPS = Compare Byte Word</td>
<td>1010011w</td>
<td>10*</td>
<td>2*</td>
<td>a</td>
</tr>
<tr>
<td>INS = Input Byte/Word from DX Port</td>
<td>0110110w</td>
<td>9*</td>
<td>1*</td>
<td>a,l,k</td>
</tr>
<tr>
<td>LODS = Load Byte/Word to AL/AX/EAX</td>
<td>1010110w</td>
<td>29*</td>
<td>1*</td>
<td>a,l,i</td>
</tr>
<tr>
<td>MOVES = Move Byte Word</td>
<td>1010011w</td>
<td>5*</td>
<td>1*</td>
<td>a</td>
</tr>
<tr>
<td>OUTS = Output Byte/Word to DX Port</td>
<td>0110111w</td>
<td>7**</td>
<td>2**</td>
<td>a</td>
</tr>
<tr>
<td>SCAS = Scan Byte Word</td>
<td>1010111w</td>
<td>8**</td>
<td>1*</td>
<td>a,l,k</td>
</tr>
<tr>
<td>STOS = Store Byte/Word from AL/AX/EAX</td>
<td></td>
<td>28**</td>
<td>1*</td>
<td>a,l,i</td>
</tr>
<tr>
<td>STOS = Store Byte/Word from AL/AX/EAX</td>
<td></td>
<td>7*</td>
<td>1*</td>
<td>a</td>
</tr>
<tr>
<td>XLAT = Translate String</td>
<td>11010111</td>
<td>4*</td>
<td>1*</td>
<td>a</td>
</tr>
<tr>
<td>REPEATED STRING MANIPULATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Repeated by Count in CX or ECX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPE CMPS = Compare String</td>
<td>11110011 1010011w</td>
<td>5 + 9n**</td>
<td>2n**</td>
<td>a</td>
</tr>
</tbody>
</table>

4-108
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock Counts</th>
<th>Number of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPEATED STRING MANIPULATION (Continued)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPNE CMP$ = Compare String</td>
<td>11110010 1010011w</td>
<td>5 + 9n**</td>
<td>2n**</td>
<td>a</td>
</tr>
<tr>
<td>REP INS$ = Input String</td>
<td>11110011 0110110w</td>
<td>7 + 16n*</td>
<td>1n*</td>
<td>a,f,k</td>
</tr>
<tr>
<td>REP LODS$ = Load String</td>
<td>11110011 1010110w</td>
<td>27 + 6n*</td>
<td>1n*</td>
<td>a,f,l</td>
</tr>
<tr>
<td>REP MOV$ = Move String</td>
<td>11110011 1010010w</td>
<td>6 + 6n*</td>
<td>1n*</td>
<td>a</td>
</tr>
<tr>
<td>REP OUTS$ = Output String</td>
<td>11110011 0110111w</td>
<td>7 + 4n**</td>
<td>2n**</td>
<td>a</td>
</tr>
<tr>
<td>REP SCAS$ = Scan String</td>
<td>11110011 1010111w</td>
<td>6 + 6n*</td>
<td>1n*</td>
<td>a,f,k</td>
</tr>
<tr>
<td>REPNE SCAS$ = Scan String (Find Non-AL/AX/EAX)</td>
<td>11110010 1010111w</td>
<td>26 + 5n*</td>
<td>1n*</td>
<td>a,f,l</td>
</tr>
<tr>
<td>REP STOS$ = Store String</td>
<td>11110011 1010101w</td>
<td>5 + 5n*</td>
<td>1n*</td>
<td>a</td>
</tr>
<tr>
<td>BIT MANIPULATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSF$ = Scan Bit Forward</td>
<td>00001111 0111100 mod reg r/m</td>
<td>10 + 3n**</td>
<td>2n**</td>
<td>a</td>
</tr>
<tr>
<td>BSR$ = Scan Bit Reverse</td>
<td>00001111 0111101 mod reg r/m</td>
<td>10 + 3n**</td>
<td>2n**</td>
<td>a</td>
</tr>
<tr>
<td>BT$ = Test Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/Memory, Immediate</td>
<td>00001111 0111100 mod 00 &lt;r/m&gt; imm8-bit data</td>
<td>3/6*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td>Register/Memory, Register</td>
<td>00001111 1010001 mod reg r/m</td>
<td>3/12*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td>BTC$ = Test Bit and Complement</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/Memory, Immediate</td>
<td>00001111 0111101 mod 11 r/m imm8-bit data</td>
<td>6/12*</td>
<td>0/2*</td>
<td>a</td>
</tr>
<tr>
<td>Register/Memory, Register</td>
<td>00001111 1011101 mod reg r/m</td>
<td>6/13*</td>
<td>0/2*</td>
<td>a</td>
</tr>
<tr>
<td>BTR$ = Test Bit and Reset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/Memory, Immediate</td>
<td>00001111 0111100 mod 110 r/m imm8-bit data</td>
<td>6/12*</td>
<td>0/2*</td>
<td>a</td>
</tr>
<tr>
<td>Register/Memory, Register</td>
<td>00001111 1011001 mod reg r/m</td>
<td>6/13*</td>
<td>0/2*</td>
<td>a</td>
</tr>
<tr>
<td>BTS$ = Test Bit and Set</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/Memory, Immediate</td>
<td>00001111 0111101 mod 101 r/m imm8-bit data</td>
<td>6/12*</td>
<td>0/2*</td>
<td>a</td>
</tr>
<tr>
<td>Register/Memory, Register</td>
<td>00001111 1010101 mod reg r/m</td>
<td>6/13*</td>
<td>0/2*</td>
<td>a</td>
</tr>
<tr>
<td>CONTROL TRANSFER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL$ = Call</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct within Segment</td>
<td>11101000 full displacement</td>
<td>9 + m*</td>
<td>2</td>
<td>j</td>
</tr>
<tr>
<td>Register/Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirect within Segment</td>
<td>11111111 mod 010 r/m</td>
<td>9 + m/12 + m</td>
<td>2/3</td>
<td>a,j</td>
</tr>
<tr>
<td>Direct Intersegment</td>
<td>10011010 unsigned full offset, selector</td>
<td>42 + m</td>
<td>9</td>
<td>c,d,j</td>
</tr>
</tbody>
</table>
Table 8.1. 80376 Instruction Set Clock Count Summary (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock Counts</th>
<th>Number of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL TRANSFER (Continued)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Direct Intersgment)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Via Call Gate to Same Privilege Level</td>
<td></td>
<td>64 + m</td>
<td>13</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>Via Call Gate to Different Privilege Level, (No Parameters)</td>
<td></td>
<td>96 + m</td>
<td>13</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>Via Call Gate to Different Privilege Level, (x Parameters)</td>
<td></td>
<td>106 + 8x + m</td>
<td>13 + 4x</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>From 386 Task to 386 TSS</td>
<td></td>
<td>395</td>
<td>124</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>Indirect Intersgment</td>
<td>11111111 mod 011 r/m</td>
<td>46 + m</td>
<td>10</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>Via Call Gate to Same Privilege Level</td>
<td></td>
<td>68 + m</td>
<td>14</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>Via Call Gate to Different Privilege Level, (No Parameters)</td>
<td></td>
<td>102 + m</td>
<td>14</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>Via Call Gate to Different Privilege Level, (x Parameters)</td>
<td></td>
<td>110 + 8x + m</td>
<td>14 + 4x</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>From 386 Task to 386 TSS</td>
<td></td>
<td>399</td>
<td>130</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>JMP = Unconditional Jump</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Short</td>
<td>1110 001 8-bit displacement</td>
<td>7 + m</td>
<td></td>
<td>j</td>
</tr>
<tr>
<td>Direct within Segment</td>
<td>1110 001 full displacement</td>
<td>7 + m</td>
<td></td>
<td>j</td>
</tr>
<tr>
<td>Register/Memory Indirect within Segment</td>
<td>11111111 mod 100 r/m</td>
<td>9 + m/14 + m</td>
<td>2/4</td>
<td>a,j</td>
</tr>
<tr>
<td>Direct Intersgment</td>
<td>1111 010 100, unsigned 8-bit offset, selector</td>
<td>37 + m</td>
<td>5</td>
<td>c,d,j</td>
</tr>
<tr>
<td>Via Call Gate to Same Privilege Level</td>
<td></td>
<td>53 + m</td>
<td>9</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>From 386 Task to 386 TSS</td>
<td></td>
<td>395</td>
<td>124</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>Indirect Intersgment</td>
<td>11111111 mod 101 r/m</td>
<td>37 + m</td>
<td>9</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>Via Call Gate to Same Privilege Level</td>
<td></td>
<td>59 + m</td>
<td>13</td>
<td>a,c,d,j</td>
</tr>
<tr>
<td>From 386 Task to 386 TSS</td>
<td></td>
<td>401</td>
<td>124</td>
<td>a,c,d,j</td>
</tr>
</tbody>
</table>
### Table 8.1. 80376 Instruction Set Clock Count Summary (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock Counts</th>
<th>Number of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL TRANSFER (Continued)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET = Return from CALL:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Within Segment</td>
<td>11 00 00 11</td>
<td>12 + m</td>
<td>2</td>
<td>a,j,p</td>
</tr>
<tr>
<td>Within Segment Adding Immediate to SP</td>
<td>11 00 00 10 16-bit displ</td>
<td>12 + m</td>
<td>2</td>
<td>a,j,p</td>
</tr>
<tr>
<td>Intersegment</td>
<td>11 00 01 11</td>
<td>16</td>
<td>4</td>
<td>ac,d,j,p</td>
</tr>
<tr>
<td>Intersegment Adding Immediate to SP</td>
<td>11 00 10 10 16-bit displ</td>
<td>16 + m</td>
<td>4</td>
<td>ac,d,j,p</td>
</tr>
<tr>
<td>to Different Privilege Level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intersegment</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intersegment Adding Immediate to SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONDITIONAL JUMPS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTE: Times Are Jump “Taken or Not Taken”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JO = Jump on Overflow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01 11 00 00 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00 00 11 11 10 00 00 00 00 00 10 16-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JNO = Jump on Not Overflow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01 11 00 00 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00 00 11 11 10 00 00 00 00 00 10 16-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JB/JNAE = Jump on Below/Not Above or Equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01 11 00 10 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00 00 11 11 10 00 00 00 00 00 10 16-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JNB/JAЕ = Jump on Not Below/Above or Equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01 11 00 11 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00 00 11 11 10 00 00 00 00 00 10 16-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JE/JZ = Jump on Equal/Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01 11 01 00 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00 00 11 11 10 00 00 00 00 00 10 16-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JNE/JNZ = Jump on Not Equal/Not Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01 11 01 01 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00 00 11 11 10 00 00 00 00 00 10 16-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JBE/JNA = Jump on Below or Equal/Not Above</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01 11 10 10 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00 00 11 11 10 00 00 00 00 00 10 16-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JNBE/JA = Jump on Not Below or Equal/Above</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01 11 10 11 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00 00 11 11 10 00 00 00 00 00 10 16-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JS = Jump on Sign</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01 11 11 00 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00 00 11 11 10 00 00 00 00 00 10 16-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
</tbody>
</table>
Table 8.1. 80376 Instruction Set Clock Count Summary (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock Counts</th>
<th>Number of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONDITIONAL JUMPS (Continued)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNS = Jump on Not Sign</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01111001 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00001111 10001001 full displacement</td>
<td>7 + m or 3/2</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JP/JPE = Jump on Parity/Parity Even</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01111010 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00001111 10001010 full displacement</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JNP/JPO = Jump on Not Parity/Parity Odd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01111011 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00001111 10001011 full displacement</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JL/JLGE = Jump on Less/Not Greater or Equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01111100 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00001111 10001111 full displacement</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JNE/JG = Jump on Not Less/ Greater or Equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01111101 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00001111 10001101 full displacement</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JLE/JNG = Jump on Less/Equal/Not Greater</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01111110 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00001111 10001110 full displacement</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JNLE/JG = Jump on Not Less/ Equal/Greater</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-Bit Displacement</td>
<td>01111111 8-bit displ</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>Full Displacement</td>
<td>00001111 10001111 full displacement</td>
<td>7 + m or 3</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JCXZ = Jump on CX Zero</td>
<td>01100011 8-bit displ</td>
<td>9 + m or 5</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>JECKX = Jump on ECX Zero</td>
<td>01100011 8-bit displ</td>
<td>9 + m or 5</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>(Address Size Prefix Differentiates JZ on JECXZ)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOP = Loop CX Times</td>
<td>11100010 8-bit displ</td>
<td>11 + m</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>LOOPZ/LOOPE = Loop with</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero/Equal</td>
<td>11100001 8-bit displ</td>
<td>11 + m</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>LOOPNZ/LOOPNE = Loop White</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not Zero</td>
<td>11100000 8-bit displ</td>
<td>11 + m</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td><strong>CONDITIONAL BYTE SET</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTE: Times Are Register/Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETO = Set Byte on Overflow</td>
<td>00001111 10010000 mod 000 r/m</td>
<td>4/5*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td>SETNO = Set Byte on Not Overflow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>00001111 10010001 mod 000 r/m</td>
<td>4/5*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td>SETB/SETBNE = Set Byte on Below/Not Above or Equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>00001111 10010010 mod 000 r/m</td>
<td>4/5*</td>
<td>0/1*</td>
<td>a</td>
</tr>
<tr>
<td>Instruction</td>
<td>Format</td>
<td>Clock Counts</td>
<td>Number of Data Cycles</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>--------------</td>
<td>-----------------------</td>
<td>-------</td>
</tr>
<tr>
<td><strong>CONDITIONAL BYTE SET (Continued)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETNB = Set Byte on Not Below/Above or Equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 0 0 1 0 0 1 1</td>
<td></td>
<td>4/5*</td>
<td>0/1*</td>
</tr>
<tr>
<td>SETE/SETZ = Set Byte on Equal/Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 0 0 1 0 0 0</td>
<td></td>
<td>4/5</td>
<td>0/1</td>
</tr>
<tr>
<td>SETNE/SETNZ = Set Byte on Not Equal/Not Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 0 0 1 0 1 0</td>
<td></td>
<td>4/5*</td>
<td>0/1*</td>
</tr>
<tr>
<td>SETBE/SETNA = Set Byte on Below or Equal/Not Above</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 0 0 1 0 0 1</td>
<td></td>
<td>4/5</td>
<td>0/1</td>
</tr>
<tr>
<td>SETNB/SETNA = Set Byte on Below or Equal/Not Above</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 0 0 1 0 0 1</td>
<td></td>
<td>4/5</td>
<td>0/1</td>
</tr>
<tr>
<td>SETS = Set Byte on Sign</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 1 0 0 1 0 1</td>
<td></td>
<td>4/5*</td>
<td>0/1*</td>
</tr>
<tr>
<td>SETNS = Set Byte on Not Sign</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 1 0 0 1 0 1</td>
<td></td>
<td>4/5*</td>
<td>0/1*</td>
</tr>
<tr>
<td>SETP/SETPE = Set Byte on Parity/Parity Even</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 1 0 0 1 0 1</td>
<td></td>
<td>4/5*</td>
<td>0/1*</td>
</tr>
<tr>
<td>SETNP/SETPO = Set Byte on Parity/Parity Odd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 1 0 0 1 0 1</td>
<td></td>
<td>4/5*</td>
<td>0/1*</td>
</tr>
<tr>
<td>SETL/SETNGE = Set Byte on Less/Not Greater or Equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 1 0 0 1 0 1</td>
<td></td>
<td>4/5*</td>
<td>0/1*</td>
</tr>
<tr>
<td>SETNL/SETGE = Set Byte on Not Less/Greater or Equal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 1 0 0 1 0 1</td>
<td></td>
<td>4/5*</td>
<td>0/1*</td>
</tr>
<tr>
<td>SETLE/SETNG = Set Byte on Less or Equal/Not Greater</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 1 1 0 0 1 0 1</td>
<td></td>
<td>4/5*</td>
<td>0/1*</td>
</tr>
<tr>
<td>SETNL/SETG = Set Byte on Not Less or Equal/Greater</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/Memory</td>
<td>0 0 0 0 1 1 1 1 1 1 1 0 0 1 0 1</td>
<td></td>
<td>4/5*</td>
<td>0/1*</td>
</tr>
<tr>
<td>ENTER = Enter Procedure</td>
<td>1 1 0 0 1 0 0 0</td>
<td>16-bit displacement, 8-bit level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L = 0</td>
<td></td>
<td></td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>L = 1</td>
<td></td>
<td></td>
<td>14</td>
<td>a</td>
</tr>
<tr>
<td>L &gt; 1</td>
<td></td>
<td></td>
<td>17 + 8(n - 1)</td>
<td>4(n - 1)</td>
</tr>
<tr>
<td>LEAVE = Leave Procedure</td>
<td>1 1 0 0 1 0 0 1</td>
<td></td>
<td>6</td>
<td>a</td>
</tr>
</tbody>
</table>
### Table 8.1. 80376 Instruction Set Clock Count Summary (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock Counts</th>
<th>Number of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTERRUPT INSTRUCTIONS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INT = Interrupt:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type Specified</td>
<td>11001101 type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Via Interrupt or Trap Gate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to Same Privilege Level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Via Interrupt or Trap Gate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to Different Privilege Level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>From 386 Task to 386 TSS via Task Gate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type 3</strong></td>
<td>11001100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Via Interrupt or Trap Gate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to Same Privilege Level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Via Interrupt or Trap Gate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to Different Privilege Level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>From 386 Task to 386 TSS via Task Gate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INTO = Interrupt 4 If Overflow Flag Set</strong></td>
<td>11001110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If OF = 1:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If OF = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Via Interrupt or Trap Gate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to Same Privilege Level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Via Interrupt or Trap Gate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to Different Privilege Level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>From 386 Task to 386 TSS via Task Gate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Table 8.1. 80376 Instruction Set Clock Count Summary (Continued)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock Counts</th>
<th>Number Of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTERRUPT INSTRUCTIONS</strong> (Continued)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bound = Out of Range</td>
<td>0 1 0 0 0 0 0 1 0 mod reg r/m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt 5 if Detect Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If In Range</td>
<td></td>
<td>0</td>
<td>a,c,d,i,o,p</td>
<td></td>
</tr>
<tr>
<td>If Out of Range:</td>
<td></td>
<td>14</td>
<td>c,d,i,p</td>
<td></td>
</tr>
<tr>
<td>Via Interrupt or Trap Gate</td>
<td></td>
<td>14</td>
<td>c,d,i,p</td>
<td></td>
</tr>
<tr>
<td>to Same Privilege Level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Via Interrupt or Trap Gate</td>
<td></td>
<td>14</td>
<td>c,d,i,p</td>
<td></td>
</tr>
<tr>
<td>to Different Privilege Level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>From 386 Task to 386 TSS via Task Gate</td>
<td>398</td>
<td>138</td>
<td>c,d,i,p</td>
<td></td>
</tr>
<tr>
<td><strong>INTERRUPT RETURN</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRET = Interrupt Return</td>
<td>1 1 0 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To the Same Privilege Level (within Task)</td>
<td>42</td>
<td>5</td>
<td>a,c,d,i,p</td>
<td></td>
</tr>
<tr>
<td>To Different Privilege Level (within Task)</td>
<td>86</td>
<td>5</td>
<td>a,c,d,i,p</td>
<td></td>
</tr>
<tr>
<td>From 386 Task to 386 TSS</td>
<td>328</td>
<td>138</td>
<td>c,d,i,p</td>
<td></td>
</tr>
<tr>
<td><strong>PROCESSOR CONTROL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLT = HALT</td>
<td>1 1 1 0 0 1 0 0 0</td>
<td>5</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>MOV = Move to and from Control/Debug/Test Registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR0 from register</td>
<td>0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 0</td>
<td>10</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>Register from CR0</td>
<td>0 0 0 1 1 1 1 1 1 0 0 0 0 0 0</td>
<td>6</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>DR0–3 from Register</td>
<td>0 0 0 1 1 1 1</td>
<td>22</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>DR6–7 from Register</td>
<td>0 0 0 1 1 1 1</td>
<td>16</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>Register from DR6–7</td>
<td>0 0 0 1 1 1 1</td>
<td>14</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>Register from DR0–3</td>
<td>0 0 0 1 1 1 1</td>
<td>22</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>NOP = No Operation</td>
<td>1 0 0 1 0 0 0 0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAIT = Wait until BUSY # Pin is Negated</td>
<td>1 0 0 1 1 0 1 1 1</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 8.1. 80376 Instruction Set Clock Count Summary (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock Counts</th>
<th>Number of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROCESSOR EXTENSION INSTRUCTIONS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor Extension Escape</td>
<td>11011TTT mod LLL r/m</td>
<td></td>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>TTT and LLL bits are opcode information for coprocessor.</td>
<td>See 80387SX Data Sheet</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PREFIX BYTES</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address Size Prefix</td>
<td>01100111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCK = Bus Lock Prefix</td>
<td>11110000</td>
<td>0</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>Operand Size Prefix</td>
<td>01100110</td>
<td>0</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>Segment Override Prefix</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS:</td>
<td>00101110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS:</td>
<td>00111110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES:</td>
<td>00100110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FS:</td>
<td>01100100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GS:</td>
<td>01100101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS:</td>
<td>00110110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PROTECTION CONTROL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARPL = Adjust Requested Privilege Level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>From Register/Memory</td>
<td>01100011 mod reg r/m</td>
<td>20/21**</td>
<td>2**</td>
<td>a</td>
</tr>
<tr>
<td>LAR = Load Access Rights</td>
<td>00001111 mod reg r/m</td>
<td>17/18*</td>
<td>1*</td>
<td>a,c,i,p</td>
</tr>
<tr>
<td>LGDT = Load Global Descriptor</td>
<td>00001111 00000001 mod 010 r/m</td>
<td>13**</td>
<td>3*</td>
<td>a,e</td>
</tr>
<tr>
<td>LIDT = Load Interrupt Descriptor</td>
<td>00001111 00000001 mod 011 r/m</td>
<td>13**</td>
<td>3*</td>
<td>a,e</td>
</tr>
<tr>
<td>LLDT = Load Local Descriptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table Register to Register/ Memory</td>
<td>00001111 00000000 mod 010 r/m</td>
<td>24/28*</td>
<td>5*</td>
<td>a,c,e,p</td>
</tr>
<tr>
<td>LMSW = Load Machine Status Word</td>
<td>00001111 00000001 mod 110 r/m</td>
<td>10/13*</td>
<td>1*</td>
<td>a,e</td>
</tr>
<tr>
<td>LSL = Load Segment Limit</td>
<td>00001111 00000011 mod reg r/m</td>
<td>24/27*</td>
<td>2*</td>
<td>a,c,i,p</td>
</tr>
<tr>
<td>LTR = Load Task Register</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>From Register/Memory</td>
<td>00001111 00000000 mod 001 r/m</td>
<td>27/31*</td>
<td>4*</td>
<td>a,c,e,p</td>
</tr>
<tr>
<td>SGDT = Store Global Descriptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table Register</td>
<td>00001111 00000001 mod 000 r/m</td>
<td>11*</td>
<td>3*</td>
<td>a</td>
</tr>
<tr>
<td>SIDT = Store Interrupt Descriptor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table Register</td>
<td>00001111 00000001 mod 001 r/m</td>
<td>11*</td>
<td>3*</td>
<td>a</td>
</tr>
<tr>
<td>SLDT = Store Local Descriptor Table Register</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Register/ Memory</td>
<td>00001111 00000000 mod 000 r/m</td>
<td>2/2*</td>
<td>4*</td>
<td>a</td>
</tr>
</tbody>
</table>
Table 8.1. 80376 Instruction Set Clock Count Summary  (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Clock Counts</th>
<th>Number of Data Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROTECTION CONTROL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMSW = Store Machine</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status Word</td>
<td>0000111</td>
<td>000000001 mod 100 r/m</td>
<td>2/2*</td>
<td>1*</td>
</tr>
<tr>
<td>STR = Store Task</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register/Memory</td>
<td>0000111</td>
<td>000000000 mod 01 r/m</td>
<td>2/2*</td>
<td>1*</td>
</tr>
<tr>
<td>VERR = Verity Read</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access Register/Mem</td>
<td>0000111</td>
<td>000000000 mod 100 r/m</td>
<td>10/11**</td>
<td>2**</td>
</tr>
<tr>
<td>VERW = Verity Write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access Register/Mem</td>
<td>0000111</td>
<td>000000000 mod 101 r/m</td>
<td>15/16**</td>
<td>2**</td>
</tr>
</tbody>
</table>

NOTES:

a. Exception 13 fault (general violation) will occur if the memory operand in CS, DS, ES, FS or GS cannot be used due to either a segment limit violation or access rights violation. If a stack limit is violated, and exception 12 (stack segment limit violation or not present) occurs.

b. For segment load operations, the CPL, RPL and DPL must agree with the privilege rules to avoid an exception 13 fault (general protection violation). The segment's descriptor must indicate "present" or exception 11 (CS, DS, ES, FS, GS not present). If the SS register is loaded and a stack segment not present is detected, an exception 12 (stack segment limit violation or not present occurs).

c. All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert LOCK# to maintain descriptor integrity in multiprocessor systems.

d. JMP, CALL, INT, RET and IRET instructions referring to another code segment will cause an exception 13 (general protection violation) if an applicable privilege rule is violated.

e. An exception 13 fault occurs if CPL is greater than 0.

f. An exception 13 fault occurs if CPL is greater than IOPL.

g. The IF bit of the flag register is not updated if CPL is greater than IOPL. The IOPL field of the flag register is updated only if CPL = 0.

h. Any violation of privilege rules as applied to the selector operand does not cause a protection exception; rather, the zero flag is cleared.

i. If the coprocessor's memory operand violates a segment limit or segment access rights, an exception 13 fault (general protection exception) will occur before the ESC instruction is executed. An exception 12 fault (stack segment limit violation or no present) will occur if the stack limit is violated by the operand's starting address.

j. The destination of a JMP, CALL, INT, RET or IRET must be in the defined limit of a code segment or an exception 13 fault (general protection violation) will occur.

k. If CPL ≤ IOPL

l. If CPL > IOPL

m. LOCK# is automatically asserted, regardless of the presence or absence of the LOCK# prefix.

n. The 80376 uses an early-out multiply algorithm. The actual number of clocks depends on the position of the most significant bit in the operand (multiplier). Clock counts given are minimum to maximum. To calculate actual clocks use the following formula:

Actual Clock = if \( m > 0 \) then \( \log_2 |m| \), 3) + 9 clocks:
if \( m = 0 \) then 12 clocks (where \( m \) is the multiplier)

o. An exception may occur, depending on the value of the operand.

p. LOCK# is asserted during descriptor table accesses.
8.2 INSTRUCTION ENCODING

Overview

All instruction encodings are subsets of the general instruction format shown in Figure 8.1. Instructions consist of one or two primary opcode bytes, possibly an address specifier consisting of the "mod r/m" byte and "scaled index" byte, a displacement if required, and an immediate data field if required.

Within the primary opcode or opcodes, smaller encoding fields may be defined. These fields vary according to the class of operation. The fields define such information as direction of the operation, size of the displacements, register encoding, or sign extension.

Almost all instructions referring to an operand in memory have an addressing mode byte following the primary opcode byte(s). This byte, the mod r/m byte, specifies the address mode to be used. Certain encodings of the mod r/m byte indicate a second addressing byte, the scale-index-base byte, follows the mod r/m byte to fully specify the addressing mode.

Addressing modes can include a displacement immediately following the mod r/m byte, or scaled index byte. If a displacement is present, the possible sizes are 8, 16 or 32 bits.

If the instruction specifies an immediate operand, the immediate operand follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction.

Figure 8.1 illustrates several of the fields that can appear in an instruction, such as the mod field and the r/m field, but the Figure does not show all fields. Several smaller fields also appear in certain instructions, sometimes within the opcode bytes themselves. Table 8.2 is a complete list of all fields appearing in the 80376 instruction set. Further ahead, following Table 8.2, are detailed tables for each field.

```
TTTTTTTT TTTTTTTT mod TTTT r/m ss index base d32 | 16 | 8 | none data32 | 16 | 8 | none
7 0 7 0 7 6 5 3 2 0 7 6 5 3 2 0

opcode (one or two bytes)
"mod r/m" byte
"s-i-b" byte
address displacement (4, 2, 1 bytes or none)
immediate data (4, 2, 1 bytes or none)

register and address mode specifier

Figure 8.1. General Instruction Format
```

Table 8.2. Fields within 80376 Instructions

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Number of Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>Specifies if Data is Byte or Full Size (Full Size is either 16 or 32 Bits)</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>Specifies Direction of Data Operation</td>
<td>1</td>
</tr>
<tr>
<td>s</td>
<td>Specifies if an Immediate Data Field Must be Sign-Extended</td>
<td>1</td>
</tr>
<tr>
<td>reg</td>
<td>General Register Specifier</td>
<td>3</td>
</tr>
<tr>
<td>mod r/m</td>
<td>Address Mode Specifier (Effective Address can be a General Register)</td>
<td>2 for mod;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 for r/m</td>
</tr>
<tr>
<td>ss</td>
<td>Scale Factor for Scaled Index Address Mode</td>
<td>2</td>
</tr>
<tr>
<td>index</td>
<td>General Register to be used as Index Register</td>
<td>3</td>
</tr>
<tr>
<td>base</td>
<td>General Register to be used as Base Register</td>
<td>3</td>
</tr>
<tr>
<td>sreg2</td>
<td>Segment Register Specifier for CS, SS, DS, ES</td>
<td>2</td>
</tr>
<tr>
<td>sreg3</td>
<td>Segment Register Specifier for CS, SS, DS, ES, FS, GS</td>
<td>3</td>
</tr>
<tr>
<td>ttin</td>
<td>For Conditional Instructions, Specifies a Condition Asserted or a Condition Negated</td>
<td>4</td>
</tr>
</tbody>
</table>

Note: Table 8.1 shows encoding of individual instructions.
16-Bit Extensions of the Instruction Set

Two prefixes, the Operand Size Prefix (66H) and the Effective Address Size Prefix (67H), allow overriding individually the Default selection of operand size and effective address size. These prefixes may precede any opcode bytes and affect only the instruction they precede. If necessary, one or both of the prefixes may be placed before the opcode bytes. The presence of the Operand Size Prefix and the Effective Address Prefix will allow 16-bit data operation and 16-bit effective address calculations.

For instructions with more than one prefix, the order of prefixes is unimportant.

Unless specified otherwise, instructions with 8-bit and 16-bit operands do not affect the contents of the high-order bits of the extended registers.

Encoding of Instruction Fields

Within the instruction are several fields indicating register selection, addressing mode and so on.

ENCODING OF OPERAND LENGTH (w) FIELD

For any given instruction performing a data operation, the instruction will execute as a 32-bit operation. Within the constraints of the operation size, the w field encodes the operand size as either one byte or the full operation size, as shown in the table below.

<table>
<thead>
<tr>
<th>w Field</th>
<th>Operand Size During 16-Bit Data Operations</th>
<th>Operand Size During 32-Bit Data Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 Bits</td>
<td>8 Bits</td>
</tr>
<tr>
<td>1</td>
<td>16 Bits</td>
<td>32 Bits</td>
</tr>
</tbody>
</table>

ENCODING OF THE GENERAL REGISTER (reg) FIELD

The general register is specified by the reg field, which may appear in the primary opcode bytes, or as the reg field of the “mod r/m” byte, or as the r/m field of the “mod r/m” byte.

### Encoding of reg Field When w Field is not Present in Instruction

<table>
<thead>
<tr>
<th>reg Field</th>
<th>Register Selected During 16-Bit Data Operations</th>
<th>Register Selected During 32-Bit Data Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td>001</td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>010</td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>011</td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>100</td>
<td>SP</td>
<td>ESP</td>
</tr>
<tr>
<td>101</td>
<td>BP</td>
<td>EBP</td>
</tr>
<tr>
<td>101</td>
<td>SI</td>
<td>ESI</td>
</tr>
<tr>
<td>101</td>
<td>DI</td>
<td>EDI</td>
</tr>
</tbody>
</table>

### Encoding of reg Field When w Field is Present in Instruction

#### Register Specified by reg Field During 16-Bit Data Operations:

<table>
<thead>
<tr>
<th>reg</th>
<th>Function of w Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(when w = 0)</td>
</tr>
<tr>
<td>000</td>
<td>AL</td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
</tr>
<tr>
<td>100</td>
<td>AH</td>
</tr>
<tr>
<td>101</td>
<td>CH</td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
</tr>
</tbody>
</table>

#### Register Specified by reg Field During 32-Bit Data Operations

<table>
<thead>
<tr>
<th>reg</th>
<th>Function of w Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(when w = 0)</td>
</tr>
<tr>
<td>000</td>
<td>AL</td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
</tr>
<tr>
<td>100</td>
<td>AH</td>
</tr>
<tr>
<td>101</td>
<td>CH</td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
</tr>
</tbody>
</table>
ENCODING OF THE SEGMENT REGISTER (sreg) FIELD

The sreg field in certain instructions is a 2-bit field allowing one of the CS, DS, ES or SS segment registers to be specified. The sreg field in other instructions is a 3-bit field, allowing the FS and GS segment registers to be specified also.

<table>
<thead>
<tr>
<th>2-Bit sreg2 Field</th>
<th>Segment Register Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ES</td>
</tr>
<tr>
<td>01</td>
<td>CS</td>
</tr>
<tr>
<td>10</td>
<td>SS</td>
</tr>
<tr>
<td>11</td>
<td>DS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3-Bit sreg3 Field</th>
<th>Segment Register Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ES</td>
</tr>
<tr>
<td>001</td>
<td>CS</td>
</tr>
<tr>
<td>010</td>
<td>SS</td>
</tr>
<tr>
<td>011</td>
<td>DS</td>
</tr>
<tr>
<td>100</td>
<td>FS</td>
</tr>
<tr>
<td>101</td>
<td>GS</td>
</tr>
<tr>
<td>110</td>
<td>do not use</td>
</tr>
<tr>
<td>111</td>
<td>do not use</td>
</tr>
</tbody>
</table>

ENCODING OF ADDRESS MODE

Except for special instructions, such as PUSH or POP, where the addressing mode is pre-determined, the addressing mode for the current instruction is specified by addressing bytes following the primary opcode. The primary addressing byte is the "mod r/m" byte, and a second byte of addressing information, the "s-i-b" (scale-index-base) byte, can be specified.

The s-i-b byte (scale-index-base byte) is specified when using 32-bit addressing mode and the "mod r/m" byte has r/m = 100 and mod = 00, 01 or 10. When the sib byte is present, the 32-bit addressing mode is a function of the mod, ss, index, and base fields.

The primary addressing byte, the "mod r/m" byte, also contains three bits (shown as TTT in Figure 8.1) sometimes used as an extension of the primary opcode. The three bits, however, may also be used as a register field (reg).

When calculating an effective address, either 16-bit addressing or 32-bit addressing is used. 16-bit addressing uses 16-bit address components to calculate the effective address while 32-bit addressing uses 32-bit address components to calculate the effective address. When 16-bit addressing is used, the "mod r/m" byte is interpreted as a 16-bit addressing mode specifier. When 32-bit addressing is used, the "mod r/m" byte is interpreted as a 32-bit addressing mode specifier.

Tables on the following three pages define all encodings of all 16-bit addressing modes and 32-bit addressing modes.
Encoding of Normal Address Mode with "mod r/m" byte (no "s-i-b" byte present):

<table>
<thead>
<tr>
<th>mod r/m</th>
<th>Effective Address</th>
<th>mod r/m</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 000</td>
<td>DS:[EAX]</td>
<td>10 000</td>
<td>DS:[EAX + d32]</td>
</tr>
<tr>
<td>00 001</td>
<td>DS:[ECX]</td>
<td>10 001</td>
<td>DS:[ECX + d32]</td>
</tr>
<tr>
<td>00 010</td>
<td>DS:[EDX]</td>
<td>10 010</td>
<td>DS:[EDX + d32]</td>
</tr>
<tr>
<td>00 011</td>
<td>DS:[EBX]</td>
<td>10 011</td>
<td>DS:[EBX + d32]</td>
</tr>
<tr>
<td>00 100</td>
<td>s-i-b is present</td>
<td>10 100</td>
<td>s-i-b is present</td>
</tr>
<tr>
<td>00 101</td>
<td>DS:d32</td>
<td>10 101</td>
<td>SS:[EBP + d32]</td>
</tr>
<tr>
<td>00 110</td>
<td>DS:[ESI]</td>
<td>10 110</td>
<td>DS:[ESI + d32]</td>
</tr>
<tr>
<td>00 111</td>
<td>DS:[EDI]</td>
<td>10 111</td>
<td>DS:[EDI + d32]</td>
</tr>
<tr>
<td>01 000</td>
<td>DS:[EAX + d8]</td>
<td>11 000</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 001</td>
<td>DS:[ECX + d8]</td>
<td>11 001</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 010</td>
<td>DS:[EDX + d8]</td>
<td>11 010</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 011</td>
<td>DS:[EBX + d8]</td>
<td>11 011</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 100</td>
<td>s-i-b is present</td>
<td>11 100</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 101</td>
<td>SS:[EBP + d8]</td>
<td>11 101</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 110</td>
<td>SS:[ESI + d8]</td>
<td>11 110</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 111</td>
<td>SS:[EDI + d8]</td>
<td>11 111</td>
<td>register—see below</td>
</tr>
</tbody>
</table>

Register Specified by reg or r/m during Normal Data Operations:

<table>
<thead>
<tr>
<th>mod r/m</th>
<th>function of w field (when w = 0)</th>
<th>function of w field (when w = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 000</td>
<td>AL</td>
<td>EAX</td>
</tr>
<tr>
<td>11 001</td>
<td>CL</td>
<td>ECX</td>
</tr>
<tr>
<td>11 010</td>
<td>DL</td>
<td>EDX</td>
</tr>
<tr>
<td>11 011</td>
<td>BL</td>
<td>EBX</td>
</tr>
<tr>
<td>11 100</td>
<td>AH</td>
<td>ESP</td>
</tr>
<tr>
<td>11 101</td>
<td>CH</td>
<td>EBP</td>
</tr>
<tr>
<td>11 110</td>
<td>DH</td>
<td>ESI</td>
</tr>
<tr>
<td>11 111</td>
<td>BH</td>
<td>EDI</td>
</tr>
</tbody>
</table>

Register Specified by reg or r/m during 16-Bit Data Operations: (66H Prefix)

<table>
<thead>
<tr>
<th>mod r/m</th>
<th>function of w field (when w = 0)</th>
<th>function of w field (when w = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 000</td>
<td>AL</td>
<td>AX</td>
</tr>
<tr>
<td>11 001</td>
<td>CL</td>
<td>CX</td>
</tr>
<tr>
<td>11 010</td>
<td>DL</td>
<td>DX</td>
</tr>
<tr>
<td>11 011</td>
<td>BL</td>
<td>BX</td>
</tr>
<tr>
<td>11 100</td>
<td>AH</td>
<td>SP</td>
</tr>
<tr>
<td>11 101</td>
<td>CH</td>
<td>BP</td>
</tr>
<tr>
<td>11 110</td>
<td>DH</td>
<td>SI</td>
</tr>
<tr>
<td>11 111</td>
<td>BH</td>
<td>DI</td>
</tr>
</tbody>
</table>

4-121
Encoding of 16-bit Address Mode with "mod r/m" Byte Using 67H Prefix

<table>
<thead>
<tr>
<th>mod r/m</th>
<th>Effective Address</th>
<th>mod r/m</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 000</td>
<td>DS:[BX + SI]</td>
<td>10 000</td>
<td>DS:[BX + SI + d16]</td>
</tr>
<tr>
<td>00 001</td>
<td>DS:[BX + DI]</td>
<td>10 001</td>
<td>DS:[BX + DI + d16]</td>
</tr>
<tr>
<td>00 010</td>
<td>SS:[BP + SI]</td>
<td>10 010</td>
<td>SS:[BP + SI + d16]</td>
</tr>
<tr>
<td>00 011</td>
<td>SS:[BP + DI]</td>
<td>10 011</td>
<td>SS:[BP + DI + d16]</td>
</tr>
<tr>
<td>00 100</td>
<td>DS:[SI]</td>
<td>10 100</td>
<td>DS:[SI + d16]</td>
</tr>
<tr>
<td>00 101</td>
<td>DS:[DI]</td>
<td>10 101</td>
<td>DS:[DI + d16]</td>
</tr>
<tr>
<td>00 110</td>
<td>DS:d16</td>
<td>10 110</td>
<td>SS:[BP + d16]</td>
</tr>
<tr>
<td>00 111</td>
<td>DS:[BX]</td>
<td>10 111</td>
<td>DS:[BX + d16]</td>
</tr>
<tr>
<td>01 000</td>
<td>DS:[BX + SI + d8]</td>
<td>11 000</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 001</td>
<td>DS:[BX + DI + d8]</td>
<td>11 001</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 010</td>
<td>SS:[BP + SI + d8]</td>
<td>11 010</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 011</td>
<td>SS:[BP + DI + d8]</td>
<td>11 011</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 100</td>
<td>DS:[SI + d8]</td>
<td>11 100</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 101</td>
<td>DS:[DI + d8]</td>
<td>11 101</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 110</td>
<td>SS:[BP + d8]</td>
<td>11 110</td>
<td>register—see below</td>
</tr>
<tr>
<td>01 111</td>
<td>DS:[BX + d8]</td>
<td>11 111</td>
<td>register—see below</td>
</tr>
</tbody>
</table>
## Encoding of 32-bit Address Mode (“mod r/m” byte and “s-i-b” byte present):

<table>
<thead>
<tr>
<th>mod base</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 000</td>
<td>DS:[EAX + (scaled index)]</td>
</tr>
<tr>
<td>00 001</td>
<td>DS:[ECX + (scaled index)]</td>
</tr>
<tr>
<td>00 010</td>
<td>DS:[EDX + (scaled index)]</td>
</tr>
<tr>
<td>00 011</td>
<td>DS:[EBX + (scaled index)]</td>
</tr>
<tr>
<td>00 100</td>
<td>SS:[ESP + (scaled index)]</td>
</tr>
<tr>
<td>00 101</td>
<td>DS:[d32 + (scaled index)]</td>
</tr>
<tr>
<td>00 110</td>
<td>DS:[ESI + (scaled index)]</td>
</tr>
<tr>
<td>00 111</td>
<td>DS:[EDI + (scaled index)]</td>
</tr>
<tr>
<td>01 000</td>
<td>DS:[EAX + (scaled index) + d8]</td>
</tr>
<tr>
<td>01 001</td>
<td>DS:[ECX + (scaled index) + d8]</td>
</tr>
<tr>
<td>01 010</td>
<td>DS:[EDX + (scaled index) + d8]</td>
</tr>
<tr>
<td>01 011</td>
<td>DS:[EBX + (scaled index) + d8]</td>
</tr>
<tr>
<td>01 100</td>
<td>SS:[ESP + (scaled index) + d8]</td>
</tr>
<tr>
<td>01 101</td>
<td>SS:[EBP + (scaled index) + d8]</td>
</tr>
<tr>
<td>01 110</td>
<td>DS:[ESI + (scaled index) + d8]</td>
</tr>
<tr>
<td>01 111</td>
<td>DS:[EDI + (scaled index) + d8]</td>
</tr>
<tr>
<td>10 000</td>
<td>DS:[EAX + (scaled index) + d32]</td>
</tr>
<tr>
<td>10 001</td>
<td>DS:[ECX + (scaled index) + d32]</td>
</tr>
<tr>
<td>10 010</td>
<td>DS:[EDX + (scaled index) + d32]</td>
</tr>
<tr>
<td>10 011</td>
<td>DS:[EBX + (scaled index) + d32]</td>
</tr>
<tr>
<td>10 100</td>
<td>SS:[ESP + (scaled index) + d32]</td>
</tr>
<tr>
<td>10 101</td>
<td>SS:[EBP + (scaled index) + d32]</td>
</tr>
<tr>
<td>10 110</td>
<td>DS:[ESI + (scaled index) + d32]</td>
</tr>
<tr>
<td>10 111</td>
<td>DS:[EDI + (scaled index) + d32]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ss</th>
<th>Scale Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x1</td>
</tr>
<tr>
<td>01</td>
<td>x2</td>
</tr>
<tr>
<td>10</td>
<td>x4</td>
</tr>
<tr>
<td>11</td>
<td>x8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>Index Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>EAX</td>
</tr>
<tr>
<td>001</td>
<td>ECX</td>
</tr>
<tr>
<td>010</td>
<td>EDX</td>
</tr>
<tr>
<td>011</td>
<td>EBX</td>
</tr>
<tr>
<td>100</td>
<td>no index reg**</td>
</tr>
<tr>
<td>101</td>
<td>EBP</td>
</tr>
<tr>
<td>110</td>
<td>ESI</td>
</tr>
<tr>
<td>111</td>
<td>EDI</td>
</tr>
</tbody>
</table>

**IMPORTANT NOTE:**
When index field is 100, indicating "no index register," then ss field MUST equal 00. If index is 100 and ss does not equal 00, the effective address is undefined.

**NOTE:**
Mod field in “mod r/m” byte; ss, index, base fields in “s-i-b” byte.
ENCODING OF OPERATION DIRECTION (d) FIELD

In many two-operand instructions the d field is present to indicate which operand is considered the source and which is the destination.

<table>
<thead>
<tr>
<th>d</th>
<th>Direction of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Register/Memory &lt; - Register</td>
</tr>
<tr>
<td></td>
<td>&quot;reg&quot; Field Indicates Source Operand;</td>
</tr>
<tr>
<td></td>
<td>&quot;mod r/m&quot; or &quot;mod ss index base&quot; Indicates Destination Operand</td>
</tr>
<tr>
<td>1</td>
<td>Register &lt; - Register/Memory</td>
</tr>
<tr>
<td></td>
<td>&quot;reg&quot; Field Indicates Destination Operand;</td>
</tr>
<tr>
<td></td>
<td>&quot;mod r/m&quot; or &quot;mod ss index base&quot; Indicates Source Operand</td>
</tr>
</tbody>
</table>

ENCODING OF SIGN-EXTEND (s) FIELD

The s field occurs primarily to instructions with immediate data fields. The s field has an effect only if the size of the immediate data is 8 bits and is being placed in a 16-bit or 32-bit destination.

| s | Effect on Immediate Data8 | Effect on Immediate Data 16|32 |
|---|---------------------------|-----------------------------|
| 0 | None                      | None                        |
| 1 | Sign-Extend Data8 to Fill | None                        |

ENCODING OF CONDITIONAL TEST (tttn) FIELD

For the conditional instructions (conditional jumps and set on condition), tttn is encoded with n indicating to use the condition (n = 0) or its negation (n = 1), and tt giving the condition to test.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Condition</th>
<th>tttn</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>Overflow</td>
<td>0000</td>
</tr>
<tr>
<td>NO</td>
<td>No Overflow</td>
<td>0001</td>
</tr>
<tr>
<td>B/NAE</td>
<td>Below/Not Above or Equal</td>
<td>0010</td>
</tr>
<tr>
<td>NB/AE</td>
<td>Not Below/Above or Equal</td>
<td>0011</td>
</tr>
<tr>
<td>E/Z</td>
<td>Equal/Zero</td>
<td>0100</td>
</tr>
<tr>
<td>NE/NZ</td>
<td>Not Equal/Not Zero</td>
<td>0101</td>
</tr>
<tr>
<td>BE/NA</td>
<td>Below or Equal/Not Above</td>
<td>0110</td>
</tr>
<tr>
<td>NBE/A</td>
<td>Not Below or Equal/Above</td>
<td>0111</td>
</tr>
<tr>
<td>S</td>
<td>Sign</td>
<td>1000</td>
</tr>
<tr>
<td>NS</td>
<td>Not Sign</td>
<td>1001</td>
</tr>
<tr>
<td>P/PE</td>
<td>Parity/Parity Even</td>
<td>1010</td>
</tr>
<tr>
<td>NP/PO</td>
<td>Not Parity/Parity Odd</td>
<td>1011</td>
</tr>
<tr>
<td>L/NGE</td>
<td>Less Than/Not Greater or Equal</td>
<td>1100</td>
</tr>
<tr>
<td>NL/GE</td>
<td>Not Less Than/Greater or Equal</td>
<td>1101</td>
</tr>
<tr>
<td>LE/NG</td>
<td>Less Than or Equal/Greater Than</td>
<td>1110</td>
</tr>
<tr>
<td>NLE/G</td>
<td>Not Less or Equal/Greater Than</td>
<td>1111</td>
</tr>
</tbody>
</table>

ENCODING OF CONTROL OR DEBUG REGISTER (eee) FIELD

For the loading and storing of the Control and Debug registers.

<table>
<thead>
<tr>
<th>When Interpreted as Control Register Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>eee Code</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>000</td>
</tr>
<tr>
<td>010</td>
</tr>
<tr>
<td>011</td>
</tr>
</tbody>
</table>

Do not use any other encoding

<table>
<thead>
<tr>
<th>When Interpreted as Debug Register Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>eee Code</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>000</td>
</tr>
<tr>
<td>001</td>
</tr>
<tr>
<td>010</td>
</tr>
<tr>
<td>011</td>
</tr>
<tr>
<td>110</td>
</tr>
<tr>
<td>111</td>
</tr>
</tbody>
</table>

Do not use any other encoding
9.0 REVISION HISTORY

This 80376 data sheet, version -002, contains updates and improvements to previous versions. A revision summary is listed here for your convenience.

The sections significantly revised since version -001 are:

**Front Page**  The 80376 Microarchitecture diagram was added.

**Section 1.0**  Figure 1.2 was updated to show both top and bottom views of the 88-pin PGA package.

**Section 2.0**  Figure 2.0 was updated to show the 16-bit registers SI, DI, BP and SP.

**Section 2.1**  Figure 2.2 was updated to show the correct bit polarity for bit 4 in the CR0 register.

**Section 2.1**  Tables 2.1 and 2.2 were updated to include additional information on the EFLAGs and CR0 registers.

**Section 2.3**  Figure 2.3 was updated to more accurately reflect the addressing mechanism of the 80376.

**Section 2.6**  In the subsection Maskable Interrupt a paragraph was added to describe the effect of interrupt gates on the IF EFLAGs bit.

**Section 2.8**  Table 2.7 was updated to reflect the correct power up condition of the CR0 register.

**Section 2.10**  Figure 2.6 was updated to show the correct bit positions of the BT, BS and BD bits in the DR6 register.

**Section 3.0**  Figure 3.1 was updated to clearly show the address calculation process.

**Section 3.2**  The subsection DESCRIPTORS was elaborated upon to clearly define the relationship between the linear address space and physical address space of the 80376.

**Section 3.2**  Figures 3.3 and 3.4 were updated to show the AVL bit field.

**Section 3.3**  The last sentence in the first paragraph of subsection PROTECTION AND I/O PERMISSION BIT MAP was deleted. This was an incorrect statement.

**Section 4.1**  In the Subsection ADDRESS BUS (BHE #, BLE #, A23−A1 last sentence in the first paragraph was updated to reflect the numeric operand addresses as 8000FCH and 8000FEH. Because the 80376 sometimes does a double word I/O access a second access to 8000FEH can be seen.

**Section 4.1**  The Subsection Hold Lantencies was updated to describe how 32-bit and unaligned accesses are internally locked but do not assert the LOCK# signal.

**Section 4.2**  Table 4.6 was updated to show the correct active data bits during a BLE# assertion.

**Section 4.4**  This section was updated to correctly reflect the pipelining of the address and status of the 80376 as opposed to 'Address Pipelining' which occurs on processors such as the 80286.

**Section 4.6**  Table 4.7 was updated to show the correct Revision number, 05H.

**Section 4.7**  Table 4.8 was updated to show the numeric operand register 8000FEH. This address is seen when the 80376 does a DWORD operation to the port address 8000FCH.

**Section 5.0**  In the first paragraph the case temperatures were updated to correctly reflect the 0°C−115°C for the ceramic package and 0°C−110°C for the plastic package.

**Section 6.2**  Table 6.2 was updated to correctly reflect the Case Temperature under Bias specification of −65°C−120°C.

**Section 6.4**  Figure 6.8 vertical axis was updated to reflect "Output Valid Delay (ns)".

**Section 6.4**  Figure 6.11 was updated to show typical ICC vs Frequency for the 80376.

**Section 6.5**  This entire section was updated to reflect the new ICE-376 emulator.

**Section 8.1**  The clock counts and opcodes for various instructions were updated to their correct value.

**Section 8.2**  The section INSTRUCTION ENCODING was appended to the data sheet.
82370 INTEGRATED SYSTEM PERIPHERAL

- High Performance 32-Bit DMA Controller for 16-Bit Bus
  - 16 MBytes/Sec Maximum Data Transfer Rate at 16 MHz
  - 8 Independently Programmable Channels
- 20-Source Interrupt Controller
  - Individually Programmable Interrupt Vectors
  - 15 External, 5 Internal Interrupts
  - 82C59A Superset
- Four 16-Bit Programmable Interval Timers
  - 82C54 Compatible
- Software Compatible to 82380
- Programmable Wait State Generator
  - 0 to 15 Wait States Pipelined
  - 0 to 16 Wait States Non-Pipelined
- DRAM Refresh Controller
- 80376 Shutdown Detect and Reset Control
  - Software/Hardware Reset
- High Speed CHMOS III Technology
- 100-Pin Plastic Quad Flat-Pack Package and 132-Pin Pin Grid Array Package
  (See Packaging Handbook Order #231369)
- Optimized for Use with the 80376 Microprocessor
  - Resides on Local Bus for Maximum Bus Bandwidth

The 82370 is a multi-function support peripheral that integrates system functions necessary in an 80376 environment. It has eight channels of high performance 32-bit DMA (32-bit internal, 16-bit external) with the most efficient transfer rates possible on the 80376 bus. System support peripherals integrated into the 82370 provide Interrupt Control, Timers, Wait State generation, DRAM Refresh Control, and System Reset logic.

The 82370's DMA Controller can transfer data between devices of different data path widths using a single channel. Each DMA channel operates independently in any of several modes. Each channel has a temporary data storage register for handling non-aligned data without the need for external alignment logic.

![Internal Block Diagram]
Pin Descriptions

The 82370 provides all of the signals necessary to interface an 80376 host processor. It has a separate 24-bit address and 16-bit data bus. It also has a set of control signals to support operation as a bus master or a bus slave. Several special function signals exist on the 82370 for interfacing the system support peripherals to their respective system counterparts. Following are the definitions of the individual pins of the 82370. These brief descriptions are provided as a reference. Each signal is further defined within the sections which describe the associated 82370 function.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Name and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1–A23</td>
<td>I/O</td>
<td>ADDRESS BUS: Outputs physical memory or port I/O addresses. See Address Bus (2.2.3) for additional information.</td>
</tr>
<tr>
<td>BHE #</td>
<td>I/O</td>
<td>BYTE ENABLES: Indicate which data bytes of the data bus take part in a bus cycle. See Byte Enable (2.2.4) for additional information.</td>
</tr>
<tr>
<td>BLE #</td>
<td>I/O</td>
<td>DATA BUS: This is the 16-bit data bus. These pins are active outputs during interrupt acknowledges, during Slave accesses, and when the 82370 is in the Master Mode.</td>
</tr>
<tr>
<td>D0–D15</td>
<td>I/O</td>
<td>PROCESSOR CLOCK: This pin must be connected to the processor’s clock, CLK2. The 82370 monitors the phase of this clock in order to remain synchronized with the CPU. This clock drives all of the internal synchronous circuitry.</td>
</tr>
<tr>
<td>CLK2</td>
<td>I</td>
<td>DATA/CONTROL: D/C# is used to distinguish between CPU control cycles and DMA or CPU data access cycles. It is active as an output only in the Master Mode.</td>
</tr>
<tr>
<td>W/R #</td>
<td>I/O</td>
<td>WRITE/READ: W/R # is used to distinguish between write and read cycles. It is active as an output only in the Master Mode.</td>
</tr>
<tr>
<td>M/IO #</td>
<td>I/O</td>
<td>MEMORY/IO: M/IO # is used to distinguish between memory and IO accesses. It is active as an output only in the Master Mode.</td>
</tr>
<tr>
<td>ADS #</td>
<td>I/O</td>
<td>ADDRESS STATUS: This signal indicates presence of a valid address on the address bus. It is active as output only in the Master Mode. ADS # is active during the first T-state where addresses and control signals are valid.</td>
</tr>
<tr>
<td>NA #</td>
<td>I</td>
<td>NEXT ADDRESS: Asserted by a peripheral or memory to begin a pipelined address cycle. This pin is monitored only while the 82370 is in the Master Mode. In the Slave Mode, pipelining is determined by the current and past status of the ADS # and READY # signals.</td>
</tr>
<tr>
<td>HOLD</td>
<td>O</td>
<td>HOLD REQUEST: This is an active-high signal to the Bus Master to request control of the system bus. When control is granted, the Bus Master activates the hold acknowledge signal (HLDA).</td>
</tr>
<tr>
<td>HLDA</td>
<td>I</td>
<td>HOLD ACKNOWLEDGE: This input signal tells the DMA controller that the Bus Master has relinquished control of the system bus to the DMA controller.</td>
</tr>
</tbody>
</table>
Pin Descriptions (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Name and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DREQ (0–3, 5–7)</td>
<td>I</td>
<td><strong>DMA REQUEST</strong>: The DMA Request inputs monitor requests from peripherals requiring DMA service. Each of the eight DMA channels has one DREQ input. These active-high inputs are internally synchronized and prioritized. Upon request, channel 0 has the highest priority and channel 7 the lowest.</td>
</tr>
<tr>
<td>DREQ4/IRQ9#</td>
<td>I</td>
<td><strong>DMA/INTERRUPT REQUEST</strong>: This is the DMA request input for channel 4. It is also connected to the interrupt controller via interrupt request 9. This internal connection is available for DMA channel 4 only. The interrupt input is active low and can be programmed as either edge or level triggered. Either function can be masked by the appropriate mask register. Priorities of the DMA channel and the interrupt request are not related but follow the rules of the individual controllers. Note that this pin has a weak internal pull-up. This causes the interrupt request to be inactive, but the DMA request will be active if there is no external connection made. Most applications will require that either one or the other of these functions be used, but not both. For this reason, it is advised that DMA channel 4 be used for transfers where a software request is more appropriate (such as memory-to-memory transfers). In such an application, DREQ4 can be masked by software, freeing IRQ9# for other purposes.</td>
</tr>
<tr>
<td>EOP#</td>
<td>I/O</td>
<td><strong>END OF PROCESS</strong>: As an output, this signal indicates that the current Requester access is the last access of the currently operating DMA channel. It is activated when Terminal Count is reached. As an input, it signals the DMA channel to terminate the current buffer and proceed to the next buffer, if one is available. This signal may be programmed as an asynchronous or synchronous input. EOP# must be connected to a pull-up resistor. This will prevent erroneous external requests for termination of a DMA process.</td>
</tr>
<tr>
<td>EDACK (0–2)</td>
<td>O</td>
<td><strong>ENCODED DMA ACKNOWLEDGE</strong>: These signals contain the encoded acknowledgment of a request for DMA service by a peripheral. The binary code formed by the three signals indicates which channel is active. Channel 4 does not have a DMA acknowledge. The inactive state is indicated by the code 100. During a Requester access, EDACK presents the code for the active DMA channel. During a Target access, EDACK presents the inactive code 100.</td>
</tr>
<tr>
<td>IRQ (11–23)#</td>
<td>I</td>
<td><strong>INTERRUPT REQUEST</strong>: These are active low interrupt request inputs. The inputs can be programmed to be edge or level sensitive. Interrupt priorities are programmable as either fixed or rotating. These inputs have weak internal pull-up resistors. Unused interrupt request inputs should be tied inactive externally.</td>
</tr>
<tr>
<td>INT</td>
<td>O</td>
<td><strong>INTERRUPT OUT</strong>: INT signals that an interrupt request is pending.</td>
</tr>
<tr>
<td>CLKin</td>
<td>I</td>
<td><strong>TIMER CLOCK INPUT</strong>: This is the clock input signal to all of the 82370's programmable timers. It is independent of the system clock input (CLK2).</td>
</tr>
<tr>
<td>TOUT1/REF#</td>
<td>O</td>
<td><strong>TIMER 1 OUTPUT/REFRESH</strong>: This pin is software programmable as either the direct output of Timer 1, or as the indicator of a refresh cycle in progress. As REF#, this signal is active during the memory read cycle which occurs during refresh.</td>
</tr>
</tbody>
</table>
Pin Descriptions (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Name and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOUT2#/IRQ3#</td>
<td>I/O</td>
<td><strong>TIMER 2 OUTPUT/INTERRUPT REQUEST:</strong> This is the inverted output of Timer 2. It is also connected directly to interrupt request 3. External hardware can use IRQ3# if Timer 2 is programmed as OUT = 0 (TOUT2# = 1).</td>
</tr>
<tr>
<td>TOUT3#</td>
<td>O</td>
<td><strong>TIMER 3 OUTPUT:</strong> This is the inverted output of Timer 3.</td>
</tr>
<tr>
<td>READY#</td>
<td>I</td>
<td><strong>READY INPUT:</strong> This active-low input indicates to the 82370 that the current bus cycle is complete. READY is sampled by the 82370 both while it is in the Master Mode, and while it is in the Slave Mode.</td>
</tr>
<tr>
<td>WSC (0–1)</td>
<td>I</td>
<td><strong>WAIT STATE CONTROL:</strong> WSC0 and WSC1 are inputs used by the Wait-State Generator to determine the number of wait states required by the currently accessed memory or I/O. The binary code on these pins, combined with the M/IO# signal, selects an internal register in which a wait-state count is stored. The combination WSC = 11 disables the wait-state generator.</td>
</tr>
<tr>
<td>READYO#</td>
<td>O</td>
<td><strong>READY OUTPUT:</strong> This is the synchronized output of the wait-state generator. It is also valid during CPU accesses to the 82370 in the Slave Mode when the 82370 requires wait states. READYO# should feed directly the processor’s READY# input.</td>
</tr>
<tr>
<td>RESET</td>
<td>I</td>
<td><strong>RESET:</strong> This synchronous input serves to initialize the state of the 82370 and provides basis for the CPURST output. RESET must be held active for at least 15 CLK2 cycles in order to guarantee the state of the 82370. After Reset, the 82370 is in the Slave Mode with all outputs except timers and interrupts in their inactive states. The state of the timers and interrupt controller must be initialized through software. This input must be active for the entire time required by the host processor to guarantee proper reset.</td>
</tr>
<tr>
<td>CHPSEL#</td>
<td>O</td>
<td><strong>CHIP SELECT:</strong> This pin is driven active whenever the 82370 is addressed in a slave bus read or write cycle. It is also active during interrupt acknowledge cycles when the 82370 is driving the Data Bus. It can be used to control the local bus transceivers to prevent contention with the system bus.</td>
</tr>
<tr>
<td>CPURST</td>
<td>O</td>
<td><strong>CPU RESET:</strong> CPURST provides a synchronized reset signal for the CPU. It is activated in the event of a software reset command, a processor shut-down detect, or a hardware reset via the RESET pin. The 82370 holds CPURST active for 62 clocks in response to either a software reset command or a shut-down detection. Otherwise CPURST reflects the RESET input.</td>
</tr>
<tr>
<td>VCC</td>
<td></td>
<td><strong>POWER:</strong> +5V input power.</td>
</tr>
<tr>
<td>VSS</td>
<td></td>
<td>Ground Reference.</td>
</tr>
</tbody>
</table>

Table 1. Wait-State Select Inputs

<table>
<thead>
<tr>
<th>Port Address</th>
<th>Wait-State Registers</th>
<th>Select Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D7 D4 D3 D0</td>
<td></td>
<td>WSC1 WSC0</td>
</tr>
<tr>
<td>72H</td>
<td>MEMORY 0 I/O 0</td>
<td>0 0</td>
</tr>
<tr>
<td>73H</td>
<td>MEMORY 1 I/O 1</td>
<td>0 1</td>
</tr>
<tr>
<td>74H</td>
<td>MEMORY 2 I/O 2</td>
<td>1 0</td>
</tr>
<tr>
<td></td>
<td>DISABLED</td>
<td>1 1</td>
</tr>
<tr>
<td>M/IO#</td>
<td></td>
<td>1 0</td>
</tr>
</tbody>
</table>
### 100 Pin Quad Flat-Pack Pin Out (Top View)

<table>
<thead>
<tr>
<th>A Row</th>
<th>B Row</th>
<th>C Row</th>
<th>D Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin</td>
<td>Label</td>
<td>Pin</td>
<td>Label</td>
</tr>
<tr>
<td>1</td>
<td>CPURST</td>
<td>26</td>
<td>VCC</td>
</tr>
<tr>
<td>2</td>
<td>INT</td>
<td>27</td>
<td>D11</td>
</tr>
<tr>
<td>3</td>
<td>VCC</td>
<td>28</td>
<td>D4</td>
</tr>
<tr>
<td>4</td>
<td>VSS</td>
<td>29</td>
<td>D12</td>
</tr>
<tr>
<td>5</td>
<td>TOUT2#/IRQ3#</td>
<td>30</td>
<td>D5</td>
</tr>
<tr>
<td>6</td>
<td>TOUT3#</td>
<td>31</td>
<td>D13</td>
</tr>
<tr>
<td>7</td>
<td>D/C#</td>
<td>32</td>
<td>D6</td>
</tr>
<tr>
<td>8</td>
<td>VCC</td>
<td>33</td>
<td>VSS</td>
</tr>
<tr>
<td>9</td>
<td>W/R#</td>
<td>34</td>
<td>D14</td>
</tr>
<tr>
<td>10</td>
<td>M/IO#</td>
<td>35</td>
<td>D7</td>
</tr>
<tr>
<td>11</td>
<td>HOLD</td>
<td>36</td>
<td>D15</td>
</tr>
<tr>
<td>12</td>
<td>TOUT1/REF#</td>
<td>37</td>
<td>A23</td>
</tr>
<tr>
<td>13</td>
<td>CLK2</td>
<td>38</td>
<td>A22</td>
</tr>
<tr>
<td>14</td>
<td>VSS</td>
<td>39</td>
<td>A21</td>
</tr>
<tr>
<td>15</td>
<td>READYO#</td>
<td>40</td>
<td>A20</td>
</tr>
<tr>
<td>16</td>
<td>EOP#</td>
<td>41</td>
<td>A19</td>
</tr>
<tr>
<td>17</td>
<td>CHPSEL#</td>
<td>42</td>
<td>A18</td>
</tr>
<tr>
<td>18</td>
<td>VCC</td>
<td>43</td>
<td>VCC</td>
</tr>
<tr>
<td>19</td>
<td>D0</td>
<td>44</td>
<td>A17</td>
</tr>
<tr>
<td>20</td>
<td>D8</td>
<td>45</td>
<td>A16</td>
</tr>
<tr>
<td>21</td>
<td>D1</td>
<td>46</td>
<td>A15</td>
</tr>
<tr>
<td>22</td>
<td>D9</td>
<td>47</td>
<td>A14</td>
</tr>
<tr>
<td>23</td>
<td>D2</td>
<td>48</td>
<td>VSS</td>
</tr>
<tr>
<td>24</td>
<td>D10</td>
<td>49</td>
<td>A13</td>
</tr>
<tr>
<td>25</td>
<td>D3</td>
<td>50</td>
<td>A12</td>
</tr>
</tbody>
</table>

- **100 Vss**
- **4-130**
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{SS}$</td>
<td>$V_{CC}$</td>
<td>$V_{SS}$</td>
<td>$V_{CC}$</td>
<td>A12</td>
<td>A9</td>
<td>A8</td>
<td>A5</td>
<td>A3</td>
<td>BHE#</td>
<td>DREQ0</td>
<td>EDACK1</td>
<td>$V_{SS}$</td>
<td>$V_{CC}$</td>
</tr>
<tr>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
</tr>
<tr>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
</tr>
<tr>
<td>$V_{CC}$</td>
<td>A19</td>
<td>A17</td>
<td>A15</td>
<td>A13</td>
<td>A10</td>
<td>A7</td>
<td>A4</td>
<td>A1</td>
<td>ADS#</td>
<td>EDACK2</td>
<td>INT</td>
<td>$V_{SS}$</td>
<td>$V_{CC}$</td>
</tr>
<tr>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
</tr>
<tr>
<td>$V_{SS}$</td>
<td>A21</td>
<td>A18</td>
<td>A16</td>
<td>A14</td>
<td>A11</td>
<td>A6</td>
<td>A2</td>
<td>BLE#</td>
<td>DREQ4/IRQ29#</td>
<td>EDACK0</td>
<td>HLDA</td>
<td>DREQ7</td>
<td>DREQ5</td>
</tr>
<tr>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
</tr>
<tr>
<td>$V_{CC}$</td>
<td>A22</td>
<td>A20</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
</tr>
<tr>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
</tr>
<tr>
<td>$V_{CC}$</td>
<td>(NC)</td>
<td>D12</td>
<td>(NC)</td>
<td>D3</td>
<td>D10</td>
<td>(NC)</td>
<td>READYO#</td>
<td>HOLD</td>
<td>CHPSEL#</td>
<td>EOP#</td>
<td>CPUST</td>
<td>RESET</td>
<td>$V_{CC}$</td>
</tr>
<tr>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
</tr>
<tr>
<td>$V_{SS}$</td>
<td>(NC)</td>
<td>D4</td>
<td>(NC)</td>
<td>(NC)</td>
<td>D2</td>
<td>D9</td>
<td>(NC)</td>
<td>(NC)</td>
<td>TOUT1/REF#</td>
<td>M/IO#</td>
<td>TOUT3#/IRQ3</td>
<td>$V_{SS}$</td>
<td></td>
</tr>
<tr>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
<td>(NC)</td>
</tr>
<tr>
<td>$V_{CC}$</td>
<td>$V_{SS}$</td>
<td>$V_{CC}$</td>
<td>$V_{SS}$</td>
<td>$V_{CC}$</td>
<td>CLK2</td>
<td>D1</td>
<td>D0</td>
<td>DB</td>
<td>$V_{SS}$</td>
<td>$V_{CC}$</td>
<td>$V_{SS}$</td>
<td>$V_{CC}$</td>
<td></td>
</tr>
</tbody>
</table>

82370 PGA Pinout
1.0 FUNCTIONAL OVERVIEW

The 82370 contains several independent functional modules. The following is a brief discussion of the components and features of the 82370. Each module has a corresponding detailed section later in this data sheet. Those sections should be referred to for design and programming information.

1.1 82370 Architecture

The 82370 is comprised of several computer system functions that are normally found in separate LSI and VLSI components. These include: a high-performance, eight-channel, 32-bit Direct Memory Access Controller; a 20-level Programmable Interrupt Controller which is a superset of the 82C59A; four 16-bit Programmable Interval Timers which are functionally equivalent to the 82C54 timers; a DRAM Refresh Controller; a Programmable Wait State Generator; and system reset logic. The interface to the 82370 is optimized for high-performance operation with the 80376 microprocessor.

The 82370 operates directly on the 80376 bus. In the Slave Mode, it monitors the state of the processor at all times and acts or idles according to the commands of the host. It monitors the address pipeline status and generates the programmed number of wait states for the device being accessed. The 82370 also has logic to the reset of the 80376 via hardware or software reset requests and processor shutdown status.
After a system reset, the 82370 is in the Slave Mode. It appears to the system as an I/O device. It becomes a bus master when it is performing DMA transfers.

To maintain compatibility with existing software, the registers within the 82370 are accessed as bytes. If the internal logic of the 82370 requires a delay before another access by the processor, wait states are automatically inserted into the access cycle. This allows the programmer to write initialization routines, etc. without regard to hardware recovery times.

Figure 1-1 shows the basic architectural components of the 82370. The following sections briefly discuss the architecture and function of each of the distinct sections of the 82370.

![Figure 1-1. Architecture of the 82370](image-url)
1.1.1 DMA CONTROLLER

The 82370 contains a high-performance, 8-channel DMA Controller. It provides a 32-bit internal data path. Through its 16-bit external physical data bus, it is capable of transferring data in any combination of bytes, words and double-words. The addresses of both source and destination can be independently incremented, decremented or held constant, and cover the entire 16-bit physical address space of the 80376. It can disassemble and assemble non-aligned data via a 32-bit internal temporary data storage register. Data transferred between devices of different data path widths can also be assembled and disassembled using the internal temporary data storage register. The DMA Controller can also transfer aligned data between I/O and memory on the fly, allowing data transfer rates up to 16 megabytes per second for an 82370 operating at 16 MHz. Figure 1-2 illustrates the functional components of the DMA Controller.

There are twenty-four general status and command registers in the 82370 DMA Controller. Through these registers any of the channels may be programmed into any of the possible modes. The operating modes of any one channel are independent of the operation of the other channels.

Each channel has three programmable registers which determine the location and amount of data to be transferred:

- **Byte Count Register**—Number of bytes to transfer. (24-bits)
- **Requester Register**—Byte Address of memory or peripheral which is requesting DMA service. (24-bits)
- **Target Register**—Byte Address of peripheral or memory which will be accessed. (24-bits)

There are also port addresses which, when accessed, cause the 82370 to perform specific functions. The actual data written doesn’t matter, the act of writing to the specific address causes the command to be executed. The commands which operate in this mode are: Master Clear, Clear Terminal Count Interrupt Request, Clear Mask Register, and Clear Byte Pointer Flip-Flop.

DMA transfers can be done between all combinations of memory and I/O; memory-to-memory, memory-to-I/O, I/O-to-memory, and I/O-to-I/O. DMA service can be requested through software and/or hardware. Hardware DMA acknowledge signals are available for all channels (except channel 4) through an encoded 3-bit DMA acknowledge bus (EDACK0–2).

---

**Figure 1-2. 82370 DMA Controller**

4-134
The 82370 DMA Controller transfers blocks of data (buffers) in three modes: Single Buffer, Buffer Auto-Initialize, and Buffer Chaining. In the Single Buffer Process, the 82370 DMA Controller is programmed to transfer one particular block of data. Successive transfers then require reprogramming of the DMA channel. Single Buffer transfers are useful in systems where it is known at the time the transfer begins what quantity of data is to be transferred, and there is a contiguous block of data area available.

The Buffer Auto-Initialize Process allows the same data area to be used for successive DMA transfers without having to reprogram the channel.

The Buffer Chaining Process allows a program to specify a list of buffer transfers to be executed. The 82370 DMA Controller, through interrupt routines, is reprogrammed from the list. The channel is reprogrammed for a new buffer before the current buffer transfer is complete. This pipelining of the channel programming process allows the system to allocate non-contiguous blocks of data storage space, and transfer all of the data with one DMA process. The buffers that make up the chain do not have to be in contiguous locations.

Channel priority can be fixed or rotating. Fixed priority allows the programmer to define the priority of DMA channels based on hardware or other fixed parameters. Rotating priority is used to provide peripherals access to the bus on a shared basis.

With fixed priority, the programmer can set any channel to have the current lowest priority. This allows the user to reset or manually rotate the priority schedule without reprogramming the command registers.

1.1.2 PROGRAMMABLE INTERVAL TIMERS

Four 16-bit programmable interval timers reside within the 82370. These timers are identical in function to the timers in the 82C54 Programmable Interval Timer. All four of the timers share a common clock input which can be independent of the system clock. The timers are capable of operating in six different modes. In all of the modes, the current count can be latched and read by the 80376 at any time, making these very versatile event timers. Figure 1-3 shows the functional components of the Programmable Interval Timers.

The outputs of the timers are directed to key system functions, making system design simpler. Timer 0 is routed directly to an interrupt input and is not available externally. This timer would typically be used to generate time-keeping interrupts.

![Figure 1-3. Programmable Interval Timers—Block Diagram](image-url)
Timers 1 and 2 have outputs which are available for general timer/counter purposes as well as special functions. Timer 1 is routed to the refresh control logic to provide refresh timing. Timer 2 is connected to an interrupt request input to provide other timer functions. Timer 3 is a general purpose timer/counter whose output is available to external hardware. It is also connected internally to the interrupt request which defaults to the highest priority (IRQ0).

1.1.3 INTERRUPT CONTROLLER

The 82370 has the equivalent of three enhanced 82C59A Programmable Interrupt Controllers. These controllers can all be operated in the Master Mode, but the priority is always as if they were cascaded. There are 15 interrupt request inputs provided for the user, all of which can be inputs from external slave interrupt controllers. Cascading 82C59As to these request inputs allows a possible total of 120 external interrupt requests. Figure 1-4 is a block diagram of the 82370 Interrupt Controller.

Each of the interrupt request inputs can be individually programmed with its own interrupt vector, allowing more flexibility in interrupt vector mapping than was available with the 82C59A. An interrupt is provided to alert the system that an attempt is being made to program the vectors in the method of the 82C59A. This provides compatibility of existing software that used the 82C59A or 8259A with new designs using the 82370.

In the event of an unrequested or otherwise erroneous interrupt acknowledge cycle, the 82370 Interrupt Controller issues a default vector. This vector, programmed by the system software, will alert the system of unsolicited interrupts of the 80376.

The functions of the 82370 Interrupt Controller are identical to the 82C59A, except in regards to programming the interrupt vectors as mentioned above. Interrupt request inputs are programmable as either edge or level triggered and are software maskable. Priority can be either fixed or rotating and interrupt requests can be nested.

Enhancements are added to the 82370 for cascading external interrupt controllers. Master to Slave handshaking takes place on the data bus, instead of dedicated cascade lines.
1.1.4 WAIT STATE GENERATOR

The Wait State Generator is a programmable READY generation circuit for the 80376 bus. A peripheral requiring wait states can request the Wait State Generator to hold the processor's READY input inactive for a predetermined number of bus states. Six different wait state counts can be programmed into the Wait State Generator by software; three for memory accesses and three for I/O accesses. A block diagram of the 82370 Wait State Generator is shown in Figure 1-5.

The peripheral being accessed selects the required wait state count by placing a code on a 2-bit wait state select bus. This code along with the M/IO# signal from the bus master is used to select one of six internal 4-bit wait state registers which has been programmed with the desired number of wait states. From zero to fifteen wait states can be programmed into the wait state registers. The Wait State generator tracks the state of the processor or current bus master at all times, regardless of which device is the current bus master and regardless of whether or not the wait state generator is currently active.

The 82370 Wait State Generator is disabled by making the select inputs both high. This allows hardware which is intelligent enough to generate its own ready signal to be accessed without penalty. As previously mentioned, deselecting the Wait State Generator does not disable its ability to determine the proper number of wait states due to pipeline status in subsequent bus cycles.

The number of wait states inserted into a pipelined bus cycle is the value in the selected wait state register. If the bus master is operating in the non-pipelined mode, the Wait State Generator will increase the number of wait states inserted into the bus cycle by one.

On reset, the Wait State Generator's registers are loaded with the value FFH, giving the maximum number of wait states for any access in which the wait state select inputs are active.

1.1.5 DRAM REFRESH CONTROLLER

The 82370 DRAM Refresh Controller consists of a 24-bit refresh address counter and bus arbitration logic. The output of Timer 1 is used to periodically request a refresh cycle. When the controller receives the request, it requests access to the system bus through the HOLD signal. When bus control is acknowledged by the processor or current bus master, the refresh controller executes a memory read operation at the address currently in the Refresh Address Register. At the same time, it activates a refresh signal (REF#) that the memory uses to force a refresh instead of a normal read. Control of the bus is transferred to the processor at the completion of this cycle. Typically a refresh cycle will take six clock cycles to execute on an 80376 bus.

The 82370 DRAM Refresh Controller has the highest priority when requesting bus access and will interrupt any active DMA process. This allows large blocks of data to be moved by the DMA controller without affecting the refresh function. Also the DMA controller is not required to completely relinquish the bus, the refresh controller simply steals a bus cycle between DMA accesses.

The amount by which the refresh address is incremented is programmable to allow for different bus widths and memory bank arrangements.

1.1.6 CPU RESET FUNCTION

The 82370 contains a special reset function which can respond to hardware reset signals as well as a

![Figure 1-5. 82370 Wait State Generator—Block Diagram](image-url)
software reset command. The circuit will hold the 80376's RESET line active while an external hardware reset signal is present at its RESET input. It can also reset the 80376 processor as the result of a software command. The software reset command causes the 82370 to hold the processor's RESET line active for a minimum of 62 clock cycles. The 80376 requires that its RESET line be held active for a minimum of 80 clock cycles to re-initialize. For a more detailed explanation and solution, see Appendix D (System Notes).

The 82370 can be programmed to sense the shutdown detect code on the status lines from the 80376. If the Shutdown Detect function is enabled, the 82370 will automatically reset the processor. A diagnostic register is available which can be used to determine the cause of reset.

### 1.1.7 REGISTER MAP RELOCATION

After a hardware reset, the internal registers of the 82370 are located in I/O space beginning at port address 0000H. The map of the 82370's registers is relocatable via a software command. The default mapping places the 82370 between I/O addresses 0000H and 00DBH. The relocation register allows this map to be moved to any even 256-byte boundary in the processor's 16-bit I/O address space or any even 64 kbyte boundary in the 24-bit memory address space.

### 1.2 Host Interface

The 82370 is designed to operate efficiently on the local bus of an 80376 microprocessor. The control signals of the 82370 are identical in function to those of the 80376. As a slave, the 82370 operates with all of the features available on the 80376 bus. When the 82370 is in the Master Mode, it looks identical to an 80376 to the connected devices.

The 82370 monitors the bus at all times, and determines whether the current bus cycle is a pipelined or non-pipelined access. All of the status signals of the processor are monitored.

The control, status, and data registers within the 82370 are located at fixed addresses relative to each other, but the group can be relocated to either memory or I/O space and to different locations within those spaces.

As a Slave device, the 82370 monitors the control/status lines of the CPU. The 82370 will generate all of the wait states it needs whenever it is accessed. This allows the programmer the freedom of accessing 82370 registers without having to insert NOPs in the program to wait for slower 82370 internal registers.

The 82370 can determine if a current bus cycle is a pipelined or a non-pipelined cycle. It does this by monitoring the ADS#, NA# and READY# signals and thereby keeping track of the current state of the 80376.

As a bus master, the 82370 looks like an 80376 to the rest of the system. This enables the designer greater flexibility in systems which include the 82370. The designer does not have to alter the interfaces of any peripherals designed to operate with the 80376 to accommodate the 82370. The 82370 will access any peripherals on the bus in the same manner as the 80376, including recognizing pipelined bus cycles.

The 82370 is accessed as an 8-bit peripheral. The 80376 places the data of all 8-bit accesses either on D(0–7) or D(8–15). The 82370 will only accept data on these lines when in the Slave Mode. When in the Master Mode, the 82370 is a full 16-bit machine, sending and receiving data in the same manner as the 80376.

### 2.0 80376 HOST INTERFACE

The 82370 contains a set of interface signals to operate efficiently with the 80376 host processor. These signals were designed so that minimal hardware is needed to connect the 82370 to the 80376. Figure 2-1 depicts a typical system configuration with the 80376 processor. As shown in the diagram, the 82370 is designed to interface directly with the 80376 bus.

Since the 82370 resides on the opposite side of the data bus transceivers with respect to the rest of the system peripherals, it is important to note that the transceivers should be controlled so that contention between the data bus transceivers and the 82370 will not occur. In order to ease the implementation of this, the 82370 activates the CHPSEL# signal which indicates that the 82370 has been addressed and may output data. This signal should be included in the direction and enable control logic of the transceiver. When any of the 82370 internal registers are read, the data bus transceivers should be disabled so that only the 82370 will drive the local bus.

This section describes the basic bus functions of the 82370 to show how this device interacts with the 80376 processor. Other signals which are not directly related to the host interface will be discussed in their associated functional block description.
2.1 Master and Slave Modes

At any time, the 82370 acts as either a Slave device or a Master device in the system. Upon reset, the 82370 will be in the Slave Mode. In this mode, the 80376 processor can read/write into the 82370 internal registers. Initialization information may be programmed into the 82370 during Slave Mode.

When DMA service (including DRAM Refresh Cycles generated by the 82370) is requested, the 82370 will request and subsequently get control of the 80376 local bus. This is done through the HOLD and HLDA (Hold Acknowledge) signals. When the 80376 processor responds by asserting the HLDA signal, the 82370 will switch into Master Mode and perform DMA transfers. In this mode, the 82370 is the bus master of the system. It can read/write data from/to memory and peripheral devices. The 82370 will return to the Slave Mode upon completion of DMA transfers, or when HLDA is negated.

2.2 80376 Interface Signals

As mentioned in the Architecture section, the Bus Interface module of the 82370 (see Figure 1-1) contains signals that are directly connected to the 80376 host processor. This module has separate
16-bit Data and 24-bit Address busses. Also, it has additional control signals to support different bus operations on the system. By residing on the 80376 local bus, the 82370 shares the same address, data and control lines with the processor. The following subsections discuss the signals which interface to the 80376 host processor.

2.2.1 CLOCK (CLK2)

The CLK2 input provides fundamental timing for the 82370. It is divided by two internally to generate the 82370 internal clock. Therefore, CLK2 should be driven with twice the 80376’s frequency. In order to maintain synchronization with the 80376 host processor, the 82370 and the 80376 should share a common clock source. The internal clock consists of two phases: PHI1 and PHI2. Each CLK2 period is a phase of the internal clock. PHI2 is usually used to sample internal data. PHI1 is for latching internal signals. Figure 2-2 illustrates the relationship of CLK2 and the 82370 internal clock signals. The CPURST signal generated by the 82370 guarantees that the 80376 will wake up in phase with PHI1.

2.2.2 DATA BUS (D0–D15)

This 16-bit three-state bidirectional bus provides a general purpose data path between the 82370 and the system. These pins are tied directly to the corresponding Data Bus pins of the 80376 local bus. The Data Bus is also used for interrupt vectors generated by the 82370 in the Interrupt Acknowledge cycle. During Slave I/O operations, the 82370 expects a single byte to be written or read. When the 80376 host processor writes into the 82370, either D0–D7 or D8–D15 will be latched into the 82370, depending upon whether Byte Enable bit BLE# is 0 or 1 (see Table 2-1). When the 80376 host processor reads from the 82370, the single byte data will be duplicated twice on the Data Bus; i.e. on D0–D7 and D8–D15.

During Master Mode, the 82370 can transfer 16- and 8-bit data between memory (or I/O devices) and I/O devices (or memory) via the Data Bus.

2.2.3 ADDRESS BUS (A23–A1)

These three-state bidirectional signals are connected directly to the 80376 Address Bus. In the Slave Mode, they are used as input signals so that the processor can address the 82370 internal ports/registers. In the Master Mode, they are used as output signals by the 82370 to address memory and peripheral devices. The Address Bus is capable of addressing 16 Mbytes of physical memory space (000000H to FFFFFFH), and 64 Kbytes of I/O addresses.

2.2.4 BYTE ENABLE (BHE# , BLE#)

The Byte Enable pins BHE# and BLE# select the specific byte(s) in the word addressed by A1–A23. During Master Mode operation, it is used as an output by the 82370 to address memory and I/O locations. The definition of BHE# and BLE# is further illustrated in Table 2-1.

NOTE:
The 82370 will activate BHE# when output in Master Mode. For a more detailed explanation and its solutions, see Appendix D (System Notes).

![Figure 2-2. CLK2 and 82370 Internal Clock](290164-9)
As an output (Master Mode):

### Table 2-1. Byte Enable Signals

<table>
<thead>
<tr>
<th>BHE #</th>
<th>BLE #</th>
<th>Byte to be Accessed Relative to A23–A1</th>
<th>Logical Byte Presented on Data Bus During WRITE Only*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0, 1</td>
<td>B, A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A, A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>U, A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(Not Used)</td>
<td></td>
</tr>
</tbody>
</table>

*NOTE:*
Actual number of bytes accessed depends upon the programmed data path width.

### Table 2-2. Bus Cycle Definition

<table>
<thead>
<tr>
<th>M/IO #</th>
<th>D/C #</th>
<th>W/R #</th>
<th>As INPUTS</th>
<th>As OUTPUTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Interrupt Acknowledge</td>
<td>NOT GENERATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>UNDEFINED</td>
<td>NOT GENERATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>I/O Read</td>
<td>I/O Read</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>I/O Write</td>
<td>I/O Write</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UNDEFINED</td>
<td>NOT GENERATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>HALT if A1 = 1</td>
<td>NOT GENERATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SHUTDOWN if A1 = 0</td>
<td>Memory Read</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Memory Write</td>
<td>Memory Write</td>
</tr>
</tbody>
</table>

### 2.2.5 BUS CYCLE DEFINITION SIGNALS (D/C#, W/R#, M/IO#)

These three-state bidirectional signals define the type of bus cycle being performed. W/R# distinguishes between write and read cycles. D/C# distinguishes between processor data and control cycles. M/IO# distinguishes between memory and I/O cycles.

During Slave Mode, these signals are driven by the 80376 host processor; during Master Mode, they are driven by the 82370. In either mode, these signals will be valid when the Address Status (ADS#) is driven LOW. Exact bus cycle definitions are given in Table 2-2. Note that some combinations are recognized as inputs, but not generated as outputs. In the Master Mode, D/C# is always HIGH.

### 2.2.6 ADDRESS STATUS (ADS#)

This signal indicates that a valid address (A1–A23, BHE#, BLE#) and bus cycle definition (W/R#, D/C#, M/IO#) is being driven on the bus. In the Master Mode, it is driven by the 82370 as an output. In the Slave Mode, this signal is monitored as an input by the 82370. By the current and past status of ADS# and the READY# input, the 82370 is able to determine, during Slave Mode, if the next bus cycle is a pipelined address cycle. ADS# is asserted during T1 and T2P bus states (see Bus State Definition).

*NOTE:*
ADS# must be qualified with the rising edge of CLK2.

### 2.2.7 TRANSFER ACKNOWLEDGE (READY#)

This input indicates that the current bus cycle is complete. In the Master Mode, assertion of this signal indicates the end of a DMA bus cycle. In the Slave Mode, the 82370 monitors this input and ADS# to detect a pipelined address cycle. This signal should be tied directly to the READY# input of the 80376 host processor.

### 2.2.8 NEXT ADDRESS REQUEST (NA#)

This input is used to indicate to the 82370 in the Master Mode that the system is requesting address
pipelining. When driven LOW by either memory or peripheral devices during Master Mode, it indicates that the system is prepared to accept a new address and bus cycle definition signals from the 82370 before the end of the current bus cycle. If this input is active when sampled by the 82370, the next address is driven onto the bus, provided a bus request is already pending internally.

This input pin is monitored only in the Master Mode. In the Slave Mode, the 82370 uses the ADS# and READY# signals to determine address pipelining cycles, and NA# will be ignored.

2.2.9 RESET (RESET, CPURST)

RESET

This synchronous input suspends any operation in progress and places the 82370 in a known initial state. Upon reset, the 82370 will be in the Slave Mode waiting to be initialized by the 80376 host processor. The 82370 is reset by asserting RESET for 15 or more CLK2 periods. When RESET is asserted, all other input pins are ignored, and all other bus pins are driven to an idle bus state as shown in Table 2-3. The 82370 will determine the phase of its internal clock following RESET going inactive.

RESET is level-sensitive and must be synchronous to the CLK2 signal. The RESET setup and hold time requirements are shown in Figure 2-3.

<table>
<thead>
<tr>
<th>Table 2-3. Output Signals Following RESET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>A1-A23, D0-D15, BHE#, BLE#</td>
</tr>
<tr>
<td>D/C#, W/R#, M/IO#, ADS#</td>
</tr>
<tr>
<td>READY#</td>
</tr>
<tr>
<td>EOP#</td>
</tr>
<tr>
<td>EDACK2-EDACK0</td>
</tr>
<tr>
<td>HOLD</td>
</tr>
<tr>
<td>INT</td>
</tr>
<tr>
<td>TOUT1/REF#</td>
</tr>
<tr>
<td>TOUT2#/IRQ3#, TOUT3#</td>
</tr>
<tr>
<td>CPURST</td>
</tr>
<tr>
<td>CHPSEL#</td>
</tr>
</tbody>
</table>

*NOTE: The Interrupt Controller and Programmable Interval Timer are initialized by software commands.

CPURST

This output signal is used to reset the 80376 host processor. It will go active (HIGH) whenever one of the following events occurs: a) 82370's RESET input is active; b) a software RESET command is issued to the 82370; or c) when the 82370 detects a processor Shutdown cycle and when this detection feature is enabled (see CPU Reset and Shutdown Detect). When activated, CPURST will be held active for 62 clocks. The timing of CPURST is such that the 80376 processor will be in synchronization with the 82370. This timing is shown in Figure 2-4.
2.2.10 INTERRUPT OUT (INT)

This output pin is used to signal the 80376 host processor that one or more interrupt requests (either internal or external) are pending. The processor is expected to respond with an interrupt Acknowledge cycle. This signal should be connected directly to the Maskable Interrupt Request (INTR) input of the 80376 host processor.

2.3 82370 Bus Timing

The 82370 internally divides the CLK2 signal by two to generate its internal clock. Figure 2-2 showed the relationship of CLK2 and the internal clock which consists of two phases: PHI1 and PHI2. Each CLK2 period is a phase of the internal clock.

In the 82370, whether it is in the Master or Slave Mode, the shortest time unit of bus activity is a bus state. A bus state, which is also referred as a 'T-state', is defined as one 82370 PHI2 clock period (i.e. two CLK2 periods). Recall in Table 2-2 various types of bus cycles in the 82370 are defined by the M/IO#, D/C# and W/R# signals. Each of these bus cycles is composed of two or more bus states. The length of a bus cycle depends on when the READY# input is asserted (i.e. driven LOW).

2.3.1 ADDRESS PIPELINING

The 82370 supports Address Pipelining as an option in both the Master and Slave Mode. This feature typically allows a memory or peripheral device to operate with one less wait state than would otherwise be required. This is possible because during a pipelined cycle, the address and bus cycle definition of the next cycle will be generated by the bus master while waiting for the end of the current cycle to be acknowledged. The pipelined bus is especially well suited for an interleaved memory environment. For 16 MHz interleaved memory designs with 100 ns access time DRAMs, zero wait state memory accesses can be achieved when pipelined addressing is selected.

In the Master Mode, the 82370 is capable of initiating, on a cycle-by-cycle basis, either a pipelined or non-pipelined access depending upon the state of the NA# input. If a pipelined cycle is requested (indicated by NA# being driven LOW), the 82370 will drive the address and bus cycle definition of the next cycle as soon as there is an internal bus request pending.

In the Slave Mode, the 82370 is constantly monitoring the ADS# and READY# signals on the processor local bus to determine if the current bus cycle is a pipelined cycle. If a pipelined cycle is detected, the 82370 will request one less wait state from the processor if the Wait State Generator feature is selected. On the other hand, during an 82370 internal register access in a pipelined cycle, it will make use of the advance address and bus cycle information. In all cases, Address Pipelining will result in a savings of one wait state.

2.3.2 MASTER MODE BUS TIMING

When the 82370 is in the Master Mode, it will be in one of six bus states. Figure 2-5 shows the complete bus state diagram of the Master Mode, including pipelined address states. As seen in the figure, the 82370 state diagram is very similar to that of the 80376. The major difference is that in the 82370, there is no Hold state. Also, in the 82370, the conditions for some state transitions depend upon whether it is the end of a DMA process.

NOTE:
The term 'end of a DMA process' is loosely defined here. It depends on the DMA modes of operation as well as the state of the EOP# and DREQ inputs. This is explained in detail in section 3—DMA Controller.

The 82370 will enter the idle state, TI, upon RESET and whenever the internal address is not available at the end of a DMA cycle or at the end of a DMA process. When address pipelining is not used (NA# is not asserted), a new bus cycle always begins with state T1. During T1, address and bus cycle definition signals will be driven on the bus. T1 is always followed by T2.

If a bus cycle is not acknowledged (with READY#) during T2 and NA# is negated, T2 will be repeated. When the end of the bus cycle is acknowledged during T2, the following state will be T1 of the next bus cycle (if the internal address latch is loaded and if this is not the end of the DMA process). Otherwise, the TI state will be entered. Therefore, if the memory or peripheral accessed is fast enough to respond within the first T2, the fastest non-pipelined cycle will take one TI and one T2 state.

Use of the address pipelining feature allows the 82370 to enter three additional bus states: T1P, T2P and T2i. T1P is the first bus state of a pipelined bus cycle. T2P follows T1P (or T2) if NA# is asserted when sampled. The 82370 will drive the bus with the address and bus cycle definition signals of the next cycle during T2P. From the state diagram, it can be seen that after an idle state TI, the first bus cycle must begin with T1, and is therefore a non-pipelined bus cycle. The next bus cycle can be pipelined if
NA# is asserted and the previous bus cycle ended in a T2P state. Once the 82370 is in a pipelined cycle and provided that NA# is asserted in subsequent cycles, the 82370 will be switching between T1P and T2P states. If the end of the current bus cycle is not acknowledged by the READY# input, the 82370 will extend the cycle by adding T2P states. The fastest pipelined cycle will consist of one T1P and one T2P state.

The 82370 will enter state T2i when NA# is asserted and when one of the following two conditions occurs. The first condition is when the 82370 is in state T2. T2i will be entered if READY# is not asserted and there is no next address available. This situation is similar to a wait state. The 82370 will stay in T2i for as long as this condition exists. The second condition which will cause the 82370 to enter T2i is when the 82370 is in state T1P. Before going to state T2P, the 82370 needs to wait in state T2i until the next address is available. Also, in both cases, if the DMA process is complete, the 82370 will enter the T2i state in order to finish the current DMA cycle.

Figure 2-6 is a timing diagram showing non-pipelined bus accesses in the Master Mode. Figure 2-7 shows the timing of pipelined accesses in the Master Mode.  

2.3.3 SLAVE MODE BUS TIMING

Figure 2-8 shows the Slave Mode bus timing in both pipelined and non-pipelined cycles when the 82370 is being accessed. Recall that during Slave Mode, the 82370 will constantly monitor the ADS# and READY# signals to determine if the next cycle is pipelined. In Figure 2-8, the first cycle is non-pipelined and the second cycle is pipelined. In the pipelined cycle, the 82370 will start decoding the address and bus cycle signals one bus state earlier than in a non-pipelined cycle.

The READY# input signal is sampled by the 80376 host processor to determine the completion of a bus cycle. This occurs during the end of every T2, T2i and T2P state. Normally, the output of the 82370 Wait State Generator, READYO#, is directly connected to the READY# input of the 80376 host processor and the 82370. In such case, READYO# and READY# will be identical (see Wait State Generator).

Figure 2-5. Master Mode State Diagram
Figure 2-6. Non-Pipelined Bus Cycles

Figure 2-7. Pipelined Bus Cycles
3.0 DMA CONTROLLER

The 82370 DMA Controller is capable of transferring data between any combination of memory and/or I/O, with any combination of data path widths. The 82370 DMA Controller can be programmed to accommodate 8- or 16-bit devices. With its 16-bit external data path, it can transfer data in units of byte or a word. Bus bandwidth is optimized through the use of an internal temporary register which can disassemble or assemble data to or from either an aligned or non-aligned destination or source. Figure 3-1 is a block diagram of the 82370 DMA Controller.

The 82370 has eight channels of DMA. Each channel operates independently of the others. Within the operation of the individual channels, there are many different modes of data transfer available. Many of the operating modes can be intermixed to provide a very versatile DMA controller.

3.1 Functional Description

In describing the operation of the 82370’s DMA Controller, close attention to terminology is required. Be-
fore entering the discussion of the function of the 82370 DMA Controller, the following explanations of some of the terminology used herein may be of benefit. First, a few terms for clarification:

DMA PROCESS—A DMA process is the execution of a programmed DMA task from beginning to end. Each DMA process requires initial programming by the host 80376 microprocessor.

BUFFER—A contiguous block of data.

BUFFER TRANSFER—The action required by the DMA to transfer an entire buffer.

DATA TRANSFER—The DMA action in which a group of bytes or words are moved between devices by the DMA Controller. A data transfer operation may involve movement of one or many bytes.

BUS CYCLE—Access by the DMA to a single byte or word.

Each DMA channel consists of three major components. These components are identified by the contents of programmable registers which define the memory or I/O devices being serviced by the DMA. They are the Target, the Requester, and the Byte Count. They will be defined generically here and in greater detail in the DMA register definition section.

The Requester is the device which requires service by the 82370 DMA Controller, and makes the request for service. All of the control signals which the DMA monitors or generates for specific channels are logically related to the Requester. Only the Requester is considered capable of initiating or terminating a DMA process.

The Target is the device with which the Requester wishes to communicate. As far as the DMA process is concerned, the Target is a slave which is incapable of control over the process.

The direction of data transfer can be either from Requester to Target or from Target to Requester; i.e. each can be either a source or a destination.

The Requester and Target may each be either I/O or memory. Each has an address associated with it that can be incremented, decremented, or held constant. The addresses are stored in the Requester

---

**Figure 3-1. 82370 DMA Controller Block Diagram**

<table>
<thead>
<tr>
<th>CONTROL/STATUS REGISTERS</th>
<th>CHANNEL REGISTERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND REGISTER I</td>
<td>BASE TARGET ADDRESS</td>
</tr>
<tr>
<td>COMMAND REGISTER II</td>
<td>CURRENT TARGET ADDRESS</td>
</tr>
<tr>
<td>MODE REGISTER I</td>
<td>REQUESTER ADDRESS</td>
</tr>
<tr>
<td>MODE REGISTER II</td>
<td>REQUESTER ADDRESS</td>
</tr>
<tr>
<td>SOFTWARE REQUEST REGISTER</td>
<td>BASE ADDRESS</td>
</tr>
<tr>
<td>MASK REGISTER</td>
<td>CURRENT ADDRESS</td>
</tr>
<tr>
<td>STATUS REGISTER</td>
<td>CHANNEL 1 (SAME AS CH 0)</td>
</tr>
<tr>
<td>BUS SIZE REGISTER</td>
<td>CHANNEL 2 (SAME AS CH 0)</td>
</tr>
<tr>
<td>CHAINING REGISTER</td>
<td>CHANNEL 3 (SAME AS CH 0)</td>
</tr>
<tr>
<td></td>
<td>CHANNEL 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONTROL/STATUS REGISTERS</th>
<th>CHANNEL REGISTERS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CHANNEL 4 (SAME AS CH 0)</td>
</tr>
<tr>
<td></td>
<td>CHANNEL 5 (SAME AS CH 0)</td>
</tr>
<tr>
<td></td>
<td>CHANNEL 6 (SAME AS CH 0)</td>
</tr>
<tr>
<td></td>
<td>CHANNEL 7 (SAME AS CH 0)</td>
</tr>
</tbody>
</table>

290164-16
Address Registers and Target Address Registers, respectively. These registers have two parts: one which contains the current address being used in the DMA process (Current Address Register), and one which holds the programmed base address (Base Address Register). The contents of the Base Registers are never changed by the 82370 DMA Controller. The Current Registers are incremented or decremented according to the progress of the DMA process.

The Byte Count is the component of the DMA process which dictates the amount of data which must be transferred. Current and Base Byte Count Registers are provided. The Current Byte Count Register is decremented once for each byte transferred by the DMA process. When the register is decremented past zero, the Byte Count is considered 'expired' and the process is terminated or restarted, depending on the mode of operation of the channel. The point at which the Byte Count expires is called 'Terminal Count' and several status signals are dependent on this event.

Each channel of the 82370 DMA Controller also contains a 32-bit Temporary Register for use in assembling and disassembling non-aligned data. The operation of this register is transparent to the user, although the contents of it may affect the timing of some DMA handshake sequences. Since there is data storage available for each channel, the DMA Controller can be interrupted without loss of data.

To avoid unexpected results, care should be taken in programming the byte count correctly when assembling and disassembling non-aligned data. For example:

**Words to Bytes:**
Transferring two words to bytes, but setting the byte count to three, will result in three bytes transferred and the final byte flushed.

**Bytes to Words:**
Transferring six bytes to three words, but setting the byte count to five, will result in the sixth byte transferred being undefined.

The 82370 DMA Controller is a slave on the bus until a request for DMA service is received via either a software request command or a hardware request signal. The host processor may access any of the control/status or channel registers at any time the 82370 is a bus slave. Figure 3-2 shows the flow of operations that the DMA Controller performs.

At the time a DMA service request is received, the DMA Controller issues a bus hold request to the host processor. The 82370 becomes the bus master when the host relinquishes the bus by asserting a hold acknowledge signal. The channel to be serviced will be the one with the highest priority at the time the DMA Controller becomes the bus master. The DMA Controller will remain in control of the bus until the hold acknowledge signal is removed, or until the current DMA transfer is complete.

While the 82370 DMA Controller has control of the bus, it will perform the required data transfer(s). The type of transfer, source and destination addresses, and amount of data to transfer are programmed in the control registers of the DMA channel which received the request for service.

At completion of the DMA process, the 82370 will remove the bus hold request. At this time the 82370 becomes a slave again, and the host returns to being a master. If there are other DMA channels with requests pending, the controller will again assert the hold request signal and restart the bus arbitration and switching process.

![Figure 3-2. Flow of DMA Controller Operation](image-url)
knowledge (HOLD, HLDA), and End-of-Process (EOP #). The DREQn inputs and EDACK (0–2) outputs are handshake signals to the devices requiring DMA service. The HOLD output and HLDA input are handshake signals to the host processor. Figure 3-3 shows these signals and how they interconnect between the 82370 DMA Controller, and the Requester and Target devices.

3.2.1 DREQn and EDACK (0–2)

These signals are the handshake signals between the peripheral and the 82370. When the peripheral requires DMA service, it asserts the DREQn signal of the channel which is programmed to perform the service. The 82370 arbitrates the DREQn against other pending requests and begins the DMA process after finishing other higher priority processes.

When the DMA service for the requested channel is in progress, the EDACK (0–2) signals represent the DMA channel which is accessing the Requester. The 3-bit code on the EDACK (0–2) lines indicates the number of the channel presently being serviced. Table 3-2 shows the encoding of these signals. Note that Channel 4 does not have a corresponding hardware acknowledge.

The DMA acknowledge (EDACK) signals indicate the active channel only during DMA accesses to the Requester. During accesses to the Target, EDACK (0–2) has the idle code (100). EDACK (0–2) can thus be used to select a Requester device during a transfer.

DREQn can be programmed as either an Asynchronous or Synchronous input. See section 3.4.1 for details on synchronous versus asynchronous operation of these pins.

### Table 3-2. EDACK Encoding During a DMA Transfer

<table>
<thead>
<tr>
<th>EDACK2</th>
<th>EDACK1</th>
<th>EDACK0</th>
<th>Active Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Target Access</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

The EDACKn signals are always active. They either indicate 'no acknowledge' or they indicate a bus access to the requester. The acknowledge code is either 100, for an idle DMA or during a DMA access to the Target, or 'n' during a Requester access, where n is the binary value representing the channel. A simple 3-line to 8-line decoder can be used to provide discrete acknowledge signals for the peripherals.

3.2.2 HOLD AND HLDA

The Hold Request (HOLD) and Hold Acknowledge (HLDA) signals are the handshake signals between the DMA Controller and the host processor: HOLD is an output from the 82370 and HLDA is an input. HOLD is asserted by the DMA Controller when there is a pending DMA request, thus requesting the processor to give up control of the bus so the DMA process can take place. The 80376 responds by asserting HLDA when it is ready to relinquish control of the bus.
The 82370 will begin operations on the bus one clock cycle after the HLDA signal goes active. For this reason, other devices on the bus should be in the slave mode when HLDA is active.

HOLD and HLDA should not be used to gate or select peripherals requesting DMA service. This is because of the use of DMA-like operations by the DRAM Refresh Controller. The Refresh Controller is arbitrated with the DMA Controller for control of the bus, and refresh cycles have the highest priority. A refresh cycle will take place between DMA cycles without relinquishing bus control. See section 3.4.3 for a more detailed discussion of the interaction between the DMA Controller and the DRAM Refresh Controller.

3.2.3 EOP#

EOP# is a bi-directional signal used to indicate the end of a DMA process. The 82370 activates this as an output during the T2 states of the last Requester bus cycle for which a channel is programmed to execute. The Requester should respond by either withdrawing its DMA request, or interrupting the host processor to indicate that the channel needs to be programmed with a new buffer. As an input, this signal is used to tell the DMA Controller that the peripheral being serviced does not require any more data to be transferred. This indicates that the current buffer is to be terminated.

EOP# can be programmed as either an Asynchronous or a Synchronous input. See section 3.4.1 for details on synchronous versus asynchronous operation of this pin.

3.3 Modes of Operation

The 82370 DMA Controller has many independent operating functions. When designing peripheral interfaces for the 82370 DMA Controller, all of the functions or modes must be considered. All of the channels are independent of each other (except in priority of operation) and can operate in any of the modes. Many of the operating modes, though independently programmable, affect the operation of other modes. Because of the large number of combinations possible, each programmable mode is discussed here with its affects on the operation of other modes. The entire list of possible combinations will not be presented.

Table 3-1 shows the categories of DMA features available in the 82370. Each of the five major categories is independent of the others. The sub-categories are the available modes within the major function or mode category. The following sections explain each mode or function and its relation to other features.

3.3.1 TARGET/REQUESTER DEFINITION

All DMA transfers involve three devices: the DMA Controller, the Requester, and the Target. Since the devices to be accessed by the DMA Controller vary widely, the operating characteristics of the DMA Controller must be tailored to the Requester and Target devices.

The Requester can be defined as either the source or the destination of the data to be transferred. This is done by specifying a Write or a Read transfer, respectively. In a Read transfer, the Target is the data source and the Requester is the destination for the data. In a Write transfer, the Requester is the source and the Target is the destination.

The Requester and Target addresses can each be independently programmed to be incremented, decremented, or held constant. As an example, the 82370 is capable of reversing a string of data by having the Requester address increment and the Target address decrement in a memory-to-memory transfer.
3.3.2 BUFFER TRANSFER PROCESSES

The 82370 DMA Controller allows three programmable Buffer Transfer Processes. These processes define the logical way in which a buffer of data is accessed by the DMA.

The three Buffer Transfer Processes include the Single Buffer Process, the Buffer Auto-Initialize Process, and the Buffer Chaining Process. These processes require special programming considerations. See the DMA Programming section for more details on setting up the Buffer Transfer Processes.

Single Buffer Process

The Single Buffer Process allows the DMA channel to transfer only one buffer of data. When the buffer has been completely transferred (Current Byte Count decremented past zero or EOP# input active), the DMA process ends and the channel becomes idle. In order for that channel to be used again, it must be reprogrammed.

The Single Buffer Process is usually used when the amount of data to be transferred is known exactly, and it is also known that there is not likely to be any data to follow before the operating system can reprogram the channel.

Buffer Auto-Initialize Process

The Buffer Auto-Initialize Process allows multiple groups of data to be transferred to or from a single buffer. This process does not require reprogramming. The Current Registers are automatically reprogrammed from the Base Registers when the current process is terminated, either by an expired Byte Count or by an external EOP# signal. The data transferred will always be between the same Target and Requester.

The auto-initialization/process-execution cycle is repeated until the channel is either disabled or reprogrammed.

Buffer Chaining Process

The Buffer Chaining Process is useful for transferring large quantities of data into non-contiguous buffer areas. In this process, a single channel is used to process data from several buffers, while having to program the channel only once. Each new buffer is programmed in a pipelined operation that provides the new buffer information while the old buffer is being processed. The chain is created by loading new buffer information while the 82370 DMA Controller is processing the Current Buffer. When the Current Buffer expires, the 82370 DMA Controller automatically restarts the channel using the new buffer information.

Loading the new buffer information is done by an interrupt routine which is requested by the 82370. Interrupt Request 1 (IRQ1) is tied internally to the 82370 DMA Controller for this purpose. IRQ1 is generated by the 82370 when the new buffer information is loaded into the channel’s Current Registers, leaving the Base Registers ‘empty’. The interrupt service routine loads new buffer information into the Base Registers. The host processor is required to load the information for another buffer before the current Byte Count expires. The process repeats until the host programs the channel back to single buffer operation, or until the channel runs out of buffers.

The channel runs out of buffers when the Current Buffer expires and the Base Registers have not yet been loaded with new buffer information. When this occurs, the channel must be reprogrammed.

If an external EOP# is encountered while executing a Buffer Chaining Process, the current buffer is considered expired and the new buffer information is loaded into the Current Registers. If the Base Registers are ‘empty’, the chain is terminated.

The channel uses the Base Target Address Register as an indicator of whether or not the Base Registers are full. When the most significant byte of the Base Target Register is loaded, the channel considers all of the Base Registers loaded, and removes the interrupt request. This requires that the other Base Registers (Base Requester Address, Base Byte Count) must be loaded before the Base Target Address Register. The reason for implementing the reloading process this way is that, for most applications, the Byte Count and the Requester will not change from one buffer to the next, and therefore do not need to be reprogrammed. The details of programming the channel for the Buffer Chaining Process can be found in the section on DMA programming.

3.3.3 DATA TRANSFER MODES

Three Data Transfer modes are available in the 82370 DMA Controller. They are the Single Transfer, Block Transfer, and Demand Transfer Modes. These transfer modes can be used in conjunction with any one of three Buffer Transfer modes: Single Buffer, Auto-Initialized Buffer and Buffer Chaining. Any Data Transfer Mode can be used under any of the Buffer Transfer Modes. These modes are independently available for all DMA channels.

Different devices being serviced by the DMA Controller require different handshaking sequences for data transfers to take place. Three handshaking modes are available on the 82370, giving the designer the opportunity to use the DMA Controller as efficiently as possible. The speed at which data can
Single Transfer Mode

In the Single Transfer Mode, one data transfer to or from the Requester is performed by the DMA Controller at a time. The DREQn input is arbitrated and the HOLD/HLDA sequence is executed for each transfer. Transfers continue in this manner until the Byte Count expires, or until EOP# is sampled active. If the DREQn input is held active continuously, the entire DREQ-HOLD- HLDA-DACK sequence is repeated over and over until the programmed number of bytes has been transferred. Bus control is released to the host between each transfer. Figure 3-4 shows the logical flow of events which make up a buffer transfer using the Single Transfer Mode. Refer to section 3.4 for an explanation of the bus control arbitration procedure.

The Single Transfer Mode is used for devices which require complete handshake cycles with each data access. Data is transferred to or from the Requester only when the Requester is ready to perform the transfer. Each transfer requires the entire DREQ- HOLD- HLDA-DACK handshake cycle. Figure 3-5 shows the timing of the Single Transfer Mode cycle.

**NOTE:**
The Single Transfer Mode is more efficient (15%–20%) in the case where the source is the Target. Because of the internal pipeline of the 82370 DMA Controller, two idle states are added at the end of a transfer in the case where the source is the Requester.
Block Transfer Mode

In the Block Transfer Mode, the DMA process is initiated by a DMA request and continues until the Byte Count expires, or until EOP# is activated by the Requester. The DREQn signal need only be held active until the first Requester access. Only a refresh cycle will interrupt the block transfer process.

Figure 3-6 illustrates the operation of the DMA during the Block Transfer Mode. Figure 3-7 shows the timing of the handshake signals during Block Mode Transfers.
**Demand Transfer Mode**

The Demand Transfer Mode provides the most flexible handshaking procedures during the DMA process. A Demand Transfer is initiated by a DMA request. The process continues until the Byte Count expires, or an external EOP# is encountered. If the device being serviced (Requester) desires, it can interrupt the DMA process by de-activating the DREQn line. Action is taken on the condition of DREQn during Requester accesses only. The access during which DREQn is sampled inactive is the last Requester access which will be performed during the current transfer. Figure 3-8 shows the flow of events during the transfer of a buffer in the Demand Mode.

When the DREQn line goes inactive, the DMA Controller will complete the current transfer, including any necessary accesses to the Target, and relinquish control of the bus to the host. The current process information is saved (byte count, Requester and Target addresses, and Temporary Register).

The Requester can restart the transfer process by reasserting DREQn. The 82370 will arbitrate the request with other pending requests and begin the process where it left off. Figure 3-9 shows the timing of handshake signals during Demand Transfer Mode operation.

![Figure 3-8. Buffer Transfer in Demand Transfer Mode](image-url)

![Figure 3-9. Demand Mode Transfers](image-url)
Using the Demand Transfer Mode allows peripherals to access memory in small, irregular bursts without wasting bus control time. The 82370 is designed to give the best possible bus control latency in the Demand Transfer Mode. Bus control latency is defined here as the time form the last active bus cycle of the previous bus master to the first active bus cycle of the new bus master. The 82370 DMA Controller will perform its first bus access cycle two bus states after HLDA goes active. In the typical configuration, bus control is returned to the host one bus state after the DREQn goes inactive.

There are two cases where there may be more than one bus state of bus control latency at the end of a transfer. The first is at the end of an Auto-Initialize process, and the second is at the end of a process where the source is the Requester and Two-Cycle transfers are used.

When a Buffer Auto-Initialize Process is complete, the 82370 requires seven bus states to reload the Current Registers from the Base Registers of the Auto-Initialized channel. The reloading is done while the 82370 is still the bus master so that it is prepared to service the channel immediately after relinquishing the bus, if necessary.

In the case where the Requester is the source, and Two-Cycle transfers are being used, there are two extra idle states at the end of the transfer process. This occurs due to the housekeeping in the DMA's internal pipeline. These two idle states are present only after the very last Requester access, before the DMA Controller de-activates the HOLD signal.

### 3.3.4 CHANNEL PRIORITY ARBITRATION

DMA channel priority can be programmed into one of two arbitration methods: Fixed or Rotating. The four lower DMA channels and the four upper DMA channels operate as if they were two separate DMA controllers operating in cascade. The lower group of four channels (0–3) is always prioritized between channels 7 and 4 of the upper group of channels (4–7). Figure 3-10 shows a pictorial representation of the priority grouping.

The priority can thus be set up as rotating for one group of channels and fixed for the other, or any other combination. While in Fixed Priority, the programmer can also specify which channel has the lowest priority.

![Figure 3-10. DMA Priority Grouping](image)
The 82370 DMA Controller defaults to Fixed Priority. Channel 0 has the highest priority, then 1, 2, 3, 4, 5, 6, 7. Channel 7 has the lowest priority. Any time the DMA Controller arbitrates DMA requests, the requesting channel with the highest priority will be serviced next.

Fixed Priority can be entered into at any time by a software command. The priority levels in effect after the mode switch are determined by the current setting of the Programmable Priority.

Programmable Priority is available for fixing the priority of the DMA channels within a group to levels other than the default. Through a software command, the channel to have the lowest priority in a group can be specified. Each of the two groups of four channels can have the priority fixed in this way. The other channels in the group will follow the natural Fixed Priority sequence. This mode affects only the priority levels while operating with Fixed Priority.

For example, if channel 2 is programmed to have the lowest priority in its group, channel 3 has the highest priority. In descending order, the other channels would have the following priority: (3,0,1,2),4,5,6,7 (channel 2 lowest, channel 3 highest). If the upper group were programmed to have channel 5 as the lowest priority channel, the priority would be (again, highest to lowest): 6,7, (3,0,1,2), 4,5. Figure 3-11 shows this example pictorially. The lower group is always prioritized as a fifth channel of the upper group (between channels 4 and 7).

The DMA Controller will only accept Programmable Priority commands while the addressed group is operating in Fixed Priority. Switching from Fixed to Rotating Priority preserves the current priority levels. Switching from Rotating to Fixed Priority returns the priority levels to those which were last programmed by use of Programmable Priority.

Rotating Priority allows the devices using DMA to share the system bus more evenly. An individual channel does not retain highest priority after being serviced, priority is passed to the next highest priority channel in the group. The channel which was most recently serviced inherits the lowest priority. This rotation occurs each time a channel is serviced. Figure 3-12 shows the sequence of events as priority is passed between channels. Note that the lower group rotates within the upper group, and that servicing a channel within the lower group causes rotation within the group as well as rotation of the upper group.

![Diagram of DMA Channel Priority](image)

**Figure 3-11. Example of Programmed Priority**
3.3.5 COMBINING PRIORITY MODES

Since the DMA Controller operates as two four-channel controllers in cascade, the overall priority scheme of all eight channels can take on a variety of forms. There are four possible combinations of priority modes between the two groups of channels: Fixed Priority only (default), Fixed Priority upper group/Rotating Priority lower group, Rotating Priority upper group/Fixed Priority lower group, and Rotating Priority only. Figure 3-13 illustrates the operation of the two combined priority methods.

Case 1—
0–3 Fixed Priority, 4–7 Rotating Priority

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default priority</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>After servicing channel 2</td>
<td>4 5 6 7 0 1 2 3</td>
</tr>
<tr>
<td>After servicing channel 6</td>
<td>7 0 1 2 3 4 5 6</td>
</tr>
<tr>
<td>After servicing channel 1</td>
<td>4 5 6 7 0 1 2 3</td>
</tr>
</tbody>
</table>

Case 2—
0–3 Rotating Priority, 4–7 Fixed Priority

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default priority</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>After servicing channel 2</td>
<td>3 0 1 2 4 5 6 7</td>
</tr>
<tr>
<td>After servicing channel 6</td>
<td>3 0 1 2 4 5 6 7</td>
</tr>
<tr>
<td>After servicing channel 1</td>
<td>2 3 0 1 4 5 6 7</td>
</tr>
</tbody>
</table>

Figure 3-13. Combining Priority Modes

3.3.6 BUS OPERATION

Data may be transferred by the DMA Controller using two different bus cycle operations: Fly-By (one-cycle) and Two-Cycle. These bus handshake methods are selectable independently for each channel through a command register. Device data path widths are independently programmable for both Target and Requester. Also selectable through software is the direction of data transfer. All of these parameters affect the operation of the 82370 on a bus-cycle by bus-cycle basis.

3.3.6.1 Fly-By Transfers

The Fly-By Transfer Mode is the fastest and most efficient way to use the 82370 DMA Controller to transfer data. In this method of transfer, the data is written to the destination device at the same time it is read from the source. Only one bus cycle is used to accomplish the transfer.
In the Fly-By Mode, the DMA acknowledge signal is used to select the Requester. The DMA Controller simultaneously places the address of the Target on the address bus. The state of M/IO# and W/R# during the Fly-By transfer cycle indicate the type of Target and whether the Target is being written to or read from. The Target’s Bus Size is used as an incrementer for the Byte Count. The Requester address registers are ignored during Fly-By transfers.

Note that memory-to-memory transfers cannot be done using the Fly-By Mode. Only one memory of I/O address is generated by the DMA Controller at a time during Fly-By transfers. Only one of the devices being accessed can be selected by an address. Also, the Fly-By method of data transfer limits the hardware to accesses of devices with the same data bus width. The Temporary Registers are not affected in the Fly-By Mode.

Fly-By transfers also require that the data paths of the Target and Requester be directly connected. This requires that successive Fly-By access be to word boundaries, or that the Requester be capable of switching its connections to the data bus.

3.3.6.2. Two-Cycle Transfers

Two-Cycle transfers can also be performed by the 82370 DMA Controller. These transfers require at least two bus cycles to execute. The data being transferred is read into the DMA Controller’s Temporary Register during the first bus cycle(s). The second bus cycle is used to write the data from the Temporary Register to the destination.

If the addresses of the data being transferred are not word aligned, the 82370 will recognize the situation and read and write the data in groups of bytes, placing them always at the proper destination. This process of collecting the desired bytes and putting them together is called “byte assembly”. The reverse process (reading from aligned locations and writing to non-aligned locations) is called “byte disassembly”.

The assembly/disassembly process takes place transparent to the software, but can only be done while using the Two-Cycle transfer method. The 82370 will always perform the assembly/disassembly process as necessary for the current data transfer. Any data path widths for either the Requester or Target can be used in the Two-Cycle Mode. This is very convenient for interfacing existing 8- and 16-bit peripherals to the 80376’s 16-bit bus.

The 82370 DMA Controller always reads and write data within the word boundaries; i.e. if a word to be read is crossing a word boundary, the DMA Controller will perform two read operations, each reading one byte, to read the 16-bit word into the Temporary Register. Also, the 82370 DMA Controller always attempts to fill the Temporary Register from the source before writing any data to the destination. If the process is terminated before the Temporary Register is filled (TC or EOP#), the 82370 will write the partial data to the destination. If a process is temporarily suspended (such as when DREQn is deactivated during a demand transfer), the contents of a partially filled Temporary Register will be stored within the 82370 until the process is restarted.

For example, if the source is specified as an 8-bit device and the destination as a 32-bit device, there will be four reads as necessary from the 8-bit source to fill the Temporary Register. Then the 82370 will write the 32-bit contents to the destination in two cycles of 16-bit each. This cycle will repeat until the process is terminated or suspended.

With Two-Cycle transfers, the devices that the 82370 accesses can reside at any address within I/O or memory space. The device must be able to decode the byte-enables (BLE#, BHE#). Also, if the device cannot accept data in byte quantities, the programmer must take care not to allow the DMA Controller to access the device on any address other than the device boundary.

3.3.6.3 Data Path Width and Data Transfer Rate Considerations

The number of bus cycles used to transfer a single “word” of data is affected by whether the Two-Cycle or the Fly-By (Single-Cycle) transfer method is used.

The number of bus cycles used to transfer data directly affects the data transfer rate. Inefficient use of bus cycles will decrease the effective data transfer rate that can be obtained. Generally, the data transfer rate is halved by using Two-Cycle transfers instead of Fly-By transfers.

The choice of data path widths of both Target and Requester affects the data transfer rate also. During each bus cycle, the largest pieces of data possible should be transferred.

The data path width of the devices to be accessed must be programmed into the DMA controller. The 82370 defaults after reset to 8-bit-to-8-bit data transfers, but the Target and Requester can have different data path widths, independent of each other and independent of the other channels. Since this is a software programmable function, more discussion of the uses of this feature are found in the section on programming.
3.3.6.4 Read, Write and Verify Cycles

Three different bus cycles types may be used in a data transfer. They are the Read, Write and Verify cycles. These cycle types dictate the way in which the 82370 operates on the data to be transferred.

A Read Cycle transfers data from the Target to the Requester. A Write Cycle transfers data from the Requester to the target. In a Fly-By transfer, the address and bus status signals indicate the access (read of write) to the Target; the access to the Requester is assumed to be the opposite.

The Verify Cycle is used to perform a data read only. No write access is indicated or assumed in a Verify Cycle. The Verify Cycle is useful for validating block fill operations. An external comparator must be provided to do any comparisons on the data read.

3.4 Bus Arbitration and Handshaking

Figure 3-14 shows the flow of events in the DMA request arbitration process. The arbitration sequence starts when the Requester asserts a DREQn (or DMA service is requested by software). Figure 3-15 shows the timing of the sequence of events following a DMA request. This sequence is executed for each channel that is activated. The DREQn signal can be replaced by a software DMA channel request with no change in the sequence.

After the Requester asserts the service request, the 82370 will request control of the bus via the HOLD signal. The 82370 will always assert the HOLD signal one bus state after the service request is asserted. The 80376 responds by asserting the HLDA signal, thus releasing control of the bus to the 82370 DMA Controller.

Priority of pending DMA service requests is arbitrated during the first state after HLDA is asserted by the 80376. The next state will be the beginning of the first transfer access of the highest priority process.

When the 82370 DMA Controller is finished with its current bus activity, it returns control of the bus to the host processor. This is done by driving the HOLD signal inactive. The 82370 does not drive any address or data bus signals after HOLD goes low. It enters the Slave Mode until another DMA process is requested. The processor acknowledges that it has regained control of the bus by forcing the HLDA signal inactive. Note that the 82370's DMA Controller will not re-request control of the bus until the entire HOLD/HLDA handshake sequence is complete.

![Diagram of Bus Arbitration and DMA Sequence](20164-27)

Figure 3-14. Bus Arbitration and DMA Sequence

The 82370 DMA Controller will terminate a current DMA process for one of three reasons: expired byte count, end-of-process command (EOP # activated) from a peripheral, or deactivated DMA request signal. In each case, the controller will de-assert HOLD immediately after completing the data transfer in progress. These three methods of process termination are illustrated in Figures 3-16, 3-19 and 3-18, respectively.

An expired byte count indicates that the current process is complete as programmed and the channel has no further transfers to process. The channel must be restarted according to the currently programmed Buffer Transfer Mode, or reprogrammed completely, including a new Buffer Transfer Mode.
If the peripheral activates the EOP# signal, it is indicating that it will not accept or deliver any more data for the current buffer. The 82370 DMA Controller considers this as a completion of the channel's current process and interprets the condition the same way as if the byte count expired.

The action taken by the 82370 DMA Controller in response to a de-activated DREQn signal depends on the Data Transfer Mode of the channel. In the Demand Mode, data transfers will take place as long as the DREQn is active and the byte count has not expired. In the Block Mode, the controller will complete the entire block transfer without relinquishing the bus, even if DREQn goes inactive before the transfer is complete. In the Single Mode, the controller will execute single data transfers, relinquishing the bus between each transfer, as long as DREQn is active.

Normal termination of a DMA process due to expiration of the byte count (Terminal Count—TC) is shown if Figure 3-16. The condition of DREQn is ignored until after the process is terminated. If the channel is programmed to auto-initialize, HOLD will be held active for an additional seven clock cycles while the auto-initialization takes place.

Table 3-3 shows the DMA channel activity due to EOP# or Byte Count expiring (Terminal Count).

Table 3-3. DMA Channel Activity Due to Terminal Count or External EOP#

<table>
<thead>
<tr>
<th>Buffer Process</th>
<th>Single or Chaining-Base Empty</th>
<th>Auto-Initialize</th>
<th>Chaining-Base Loaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terminal Count</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>EOP#</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RESULTS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Current Registers</td>
<td></td>
<td>Load</td>
<td>Load</td>
</tr>
<tr>
<td>Channel Mask</td>
<td>Set</td>
<td>Load</td>
<td>Load</td>
</tr>
<tr>
<td>EOP# Output</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>Terminal Count Status</td>
<td>Set</td>
<td>Set</td>
<td>Set</td>
</tr>
<tr>
<td>Software Request</td>
<td>CLR</td>
<td>CLR</td>
<td>CLR</td>
</tr>
</tbody>
</table>
The 82370 always relinquishes control of the bus between channel services. This allows the hardware designer the flexibility to externally arbitrate bus hold requests, if desired. If another DMA request is pending when a higher priority channel service is completed, the 82370 will relinquish the bus until the hold acknowledge is inactive. One bus state after the HLDA signal goes inactive, the 82370 will assert HOLD again. This is illustrated in Figure 3-17.

### 3.4.1 SYNCHRONOUS AND ASYNCHRONOUS SAMPLING OF DREQn AND EOP#

As an indicator that a DMA service is to be started, DREQn is always sampled asynchronous. It is sampled at the beginning of a bus state and acted upon at the end of the state. Figure 3-15 illustrates the start of a DMA process due to a DREQn input.

The DREQn and EOP# inputs can be programmed to be sampled either synchronously or asynchronously to signal the end of a transfer.

The synchronous mode affords the Requester one bus state of extra time to react to an access. This means the Requester can terminate a process on the current access, without losing any data. The asynchronous mode requires that the input signal be presented prior to the beginning of the last state of the Requester access.
The timing relationships of the DREQn and EOP# signals to the termination of a DMA transfer are shown in Figures 3-18 and 3-19. Figure 3-18 shows the termination of a DMA transfer due to inactive DREQn. Figure 3-19 shows the termination of a DMA process due to an active EOP# input.

In the Synchronous Mode, DREQn and EOP# are sampled at the end of the last state of every Requester data transfer cycle. If EOP# is active or DREQn is inactive at this time, the 82370 recognizes this access to the Requester as the last transfer. At this point, the 82370 completes the transfer in progress, if necessary, and returns bus control to the host.

In the asynchronous mode, the inputs are sampled at the beginning of every state of a Requester access. The 82370 waits until the end of the state to act on the input.

DREQn and EOP# are sampled at the latest possible time when the 82370 can determine if another transfer is required. In the Synchronous Mode, DREQn and EOP# are sampled on the trailing edge of the last bus state before another data access cycle begins. The Asynchronous Mode requires that the signals be valid one clock cycle earlier.

![Figure 3-18. Termination of a DMA Process due to De-Asserting DREQn](image1)

![Figure 3-19. Termination of a DMA Process due to an External EOP#](image2)
While in the Pipeline Mode, if the NA signal is sampled active during a transfer, the end of the state where NA was sampled active is when the 82370 decides whether to commit to another transfer. The device must de-assert DREQn or assert EOP# before NA is sampled active, otherwise the 82370 will commit to another, possibly undesired, transfer.

Synchronous DREQn and EOP# sampling allows the peripheral to prevent the next transfer from occurring by de-activating DREQn or asserting EOP# during the current Requester access, before the 82370 DMA Controller commits itself to another transfer. The DMA Controller will not perform the next transfer if it has not already begun the bus cycle. Asynchronous sampling allows less stringent timing requirements than the Synchronous Mode, but requires that the DREQn signal be valid at the beginning of the next to last bus state of the current Requester access.

Using the Asynchronous Mode with zero wait states can be very difficult. Since the addresses and control signals are driven by the 82370 near half-way through the first bus state of a transfer, and the Asynchronous Mode requires that DREQn be inactive before the end of the state, the peripheral being accessed is required to present DREQn only a few nanoseconds after the control information is available. This means that the peripheral's control logic must be extremely fast (practically non-causal). An alternative is the Synchronous Mode.

3.4.2 Arbitration of Cascaded Master Requests

The Cascade Mode allows another DMA-type device to share the bus by arbitrating its bus accesses with the 82370's. Seven of the eight DMA channels (0–3 and 5–7) can be connected to a cascaded device. The cascaded device requests bus control through the DREQn line of the channel which is programmed to operate in Cascade Mode. Bus hold acknowledge is signalled to the cascaded device through the EDACK lines. When the EDACK lines are active with the code for the requested cascade channel, the bus is available to the cascaded master device.

A cascade cycle begins the same way a regular DMA cycle begins. The requesting bus master asserts the DREQn line on the 82370. This bus control request is arbitrated as any other DMA request would be. If any channel receives a DMA request, the 82370 requests control of the bus. When the host acknowledges that it has released bus control, the 82370 acknowledges to the requesting master that it may access the bus. The 82370 enters an idle state until the new master relinquishes control.

A cascade cycle will be terminated by one of two events: DREQn going inactive, or HLDA going inactive. The normal way to terminate the cascade cycle...
is for the cascaded master to drop the DREQn signal. Figure 3-21 shows the two cascade cycle termination sequences.

The Refresh Controller may interrupt the cascaded master to perform a refresh cycle. If this occurs, the 82370 DMA Controller will de-assert the EDACK signal (hold acknowledge to cascaded master) and wait for the cascaded master to remove its hold request. When the 82370 regains bus control, it will perform the refresh cycle in its normal fashion. After the refresh cycle has been completed, and if the cascaded device has re-asserted its request, the 82370 will return control to the cascaded master which was interrupted.

The 82370 assumes that it is the only device monitoring the HLDA signal. If the system designer wishes to place other devices on the bus as bus masters, the HLDA from the processor must be intercepted before presenting it to the 82370. Using the Cascade capability of the 82370 DMA Controller offers a much better solution.

### 3.4.3 Arbitration of Refresh Requests

The arbitration of refresh requests by the DRAM Refresh Controller is slightly different from normal DMA channel request arbitration. The 82370 DRAM Refresh Controller always has the highest priority of any DMA process. It also can interrupt a process in progress. Two types of processes in progress may be encountered: normal DMA, and bus master cascade.

In the event of a refresh request during a normal DMA process, the DMA Controller will complete the data transfer in progress and then execute the refresh cycle before continuing with the current DMA process. The priority of the interrupted process is not lost. If the data transfer cycle interrupted by the Refresh Controller is the last of a DMA process, the refresh cycle will always be executed before control of the bus is transferred back to the host.

When the Refresh Controller request occurs during a cascade cycle, the Refresh Controller must be assured that the cascaded master device has relinquished control of the bus before it can execute the refresh cycle. To do this, the DMA Controller drops the EDACK signal to the cascaded master and waits for the corresponding DREQn input to go inactive. By dropping the DREQn signal, the cascaded master relinquishes the bus. The Refresh Controller then performs the refresh cycle. Control of the bus is returned to the cascaded master if DREQn returns to an active state before the end of the refresh cycle, otherwise control is passed to the processor and the cascaded master loses its priority.

![Cascade cycle termination by DREQn inactive](image)

![Cascade cycle termination by HLDA inactive](image)

**Figure 3-21. Cascade Cycle Termination**

4-164
3.5 DMA Controller Register Overview

The 82370 DMA Controller contains 44 registers which are accessible to the host processor. Twenty-four of these registers contain the device addresses and data counts for the individual DMA channels (three per channel). The remaining registers are control and status registers for initiating and monitoring the operation of the 82370 DMA Controller. Table 3-4 lists the DMA Controller's registers and their accessibility.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Control/Status Registers</strong></td>
<td></td>
</tr>
<tr>
<td>Command Register I</td>
<td>write only</td>
</tr>
<tr>
<td>Command Register II</td>
<td>write only</td>
</tr>
<tr>
<td>Mode Register I</td>
<td>write only</td>
</tr>
<tr>
<td>Mode Register II</td>
<td>write only</td>
</tr>
<tr>
<td>Software Request Register</td>
<td>read/write</td>
</tr>
<tr>
<td>Mask Set-Reset Register</td>
<td>write only</td>
</tr>
<tr>
<td>Mask Read-Write Register</td>
<td>read/write</td>
</tr>
<tr>
<td>Status Register</td>
<td>read only</td>
</tr>
<tr>
<td>Bus Size Register</td>
<td>read only</td>
</tr>
<tr>
<td>Chaining Register</td>
<td>read/write</td>
</tr>
<tr>
<td><strong>Channel Registers</strong></td>
<td></td>
</tr>
<tr>
<td>Base Target Address</td>
<td>write only</td>
</tr>
<tr>
<td>Current Target Address</td>
<td>read only</td>
</tr>
<tr>
<td>Base Requester Address</td>
<td>write only</td>
</tr>
<tr>
<td>Current Requester Address</td>
<td>read only</td>
</tr>
<tr>
<td>Base Byte Count</td>
<td>write only</td>
</tr>
<tr>
<td>Current Byte Count</td>
<td>read only</td>
</tr>
</tbody>
</table>

3.5.1 CONTROL/STATUS REGISTERS

The following registers are available to the host processor for programming the 82370 DMA Controller into its various modes and for checking the operating status of the DMA processes. Each set of four DMA channels has one of each of these registers associated with it.

**Command Register I**

Enables or disables the DMA channel as a group. Sets the Priority Mode (Fixed or Rotating) of the group. This write-only register is cleared by a hardware reset, defaulting to all channels enabled and Fixed Priority Mode.

**Command Register II**

Sets the sampling mode of the DREQn and EOP# inputs. Also sets the lowest priority channel for the group in the Fixed Priority Mode. The functions programmed through Command Register II default after a hardware reset to: asynchronous DREQn and EOP#, and channels 3 and 7 lowest priority.

**Mode Registers**

Mode Register I is identical in function to the Mode register of the 8237A. It programs the following functions for an individually selected channel:

- Type of Transfer—read, write, verify
- Auto-Initialize—enable or disable
- Target Address Count—increment or decrement
- Data Transfer Mode—demand, single, block, cascade

Mode Register I functions default to the following after reset: verify transfer, Auto-Initialize disabled, Increment Target address, Demand Mode.

**Mode Register II**

Programs the following functions for an individually selected channel:

- Target Address Hold—enable or disable
- Requester Address Count—increment or decrement
- Requester Address Hold—enable or disable
- Target Device Type—I/O or Memory
- Requester Device Type—I/O or Memory
- Transfer Cycles—Two-Cycle or Fly-By

Mode Register II functions are defined as follows after a hardware reset: Disable Target Address Hold, Increment Requester Address, Target (and Requester) in memory, Fly-By Transfer Cycles. Note: Requester Device Type ignored in Fly-By Transfers.

**Software Request Register**

The DMA Controller can respond to service requests which are initiated by software. Each channel has an internal request status bit associated with it. The host processor can write to this register to set or reset the request bit of a selected channel.

The status of a group's software DMA service requests can be read from this register as well. Each status bit is cleared upon Terminal Count or external EOP#.

The software DMA requests are non-maskable and subject to priority arbitration with all other software and hardware requests. The entire register is cleared by a hardware reset.

**Mask Registers**

Each channel has associated with it a mask bit which can be set/reset to disable/enable that channel. Two methods are available for setting and clear-
ing the mask bits. The Mask Set/Reset Register is a write-only register which allows the host to select an individual channel and either set or reset the mask bit for that channel only. The Mask Read/Write Register is available for reading the mask bit status and for writing mask bits in groups of four.

The mask bits of a group may be cleared in one step by executing the Clear Mask Command. See the DMA Programming section for details. A hardware reset sets all of the channel mask bits, disabling all channels.

**Status Register**

The Status register is a read-only register which contains the Terminal Count (TC) and Service Request status for a group. Four bits indicate the TC status and four bits indicate the hardware request status for the four channels in the group. The TC bits are set when the Byte Count expires, or when and external EOP# is asserted. These bits are cleared by reading from the Status Register. The Service Request bit for a channel indicates when there is a hardware DMA request (DREQn) asserted for that channel. When the request has been removed, the bit is cleared.

**Bus Size Register**

This write-only register is used to define the bus size of the Target and Requester of a selected channel. The bus sizes programmed will be used to dictate the sizes of the data paths accessed when the DMA channel is active. The values programmed into this register affect the operation of the Temporary Register. When 32-bit bus width is programmed, the 82370 DMA Controller will access the device twice through its 16-bit external Data Bus to perform a 32-bit data transfer. Any byte-assembly required to make the transfers using the specified data path widths will be done in the Temporary Register. The Bus Size register of the Target is used as an increment/decrement value for the Byte Counter and Target Address when in the Fly-By Mode. Upon reset, all channels default to 8-bit Targets and 8-bit Requesters.

**Chaining Register**

As a command or write register, the Chaining register is used to enable or disable the Chaining Mode for a selected channel. Chaining can either be disabled or enabled for an individual channel, independently of the Chaining Mode status of other channels. After a hardware reset, all channels default to Chaining disabled.

When read by the host, the Chaining Register provides the status of the Chaining Interrupt of each of the channels. These interrupt status bits are cleared when the new buffer information has been loaded.

### 3.5.2 CHANNEL REGISTERS

Each channel has three individually programmable registers necessary for the DMA process; they are the Base Byte Count, Base Target Address, and Base Requester Address registers. The 24-bit Base Byte Count register contains the number of bytes to be transferred by the channel. The 24-bit Base Target Address Register contains the beginning address (memory or I/O) of the Target device. The 24-bit Base Requester Address register contains the base address (memory or I/O) of the device which is to request DMA service.

Three more registers for each DMA channel exist within the DMA Controller which are directly related to the registers mentioned above. These registers contain the current status of the DMA process. They are the Current Byte Count register, the Current Target Address, and the Current Requester Address. It is these registers which are manipulated (incremented, decremented, or held constant) by the 82370 DMA Controller during the DMA process. The Current registers are loaded from the Base registers at the beginning of a DMA process.

The Base registers are loaded when the host processor writes to the respective channel register addresses. Depending on the mode in which the channel is operating, the Current registers are typically loaded in the same operation. Reading from the channel register addresses yields the contents of the corresponding Current register.

To maintain compatibility with software which accesses an 8237A, a Byte Pointer Flip-Flop is used to control access to the upper and lower bytes of some words of the Channel Registers. These words are accessed as byte pairs at single port addresses. The Byte Pointer Flip-Flop acts as a one-bit pointer which is toggled each time a qualifying Channel Register byte is accessed.

It always points to the next logical byte to be accessed of a pair of bytes.

The Channel registers are arranged as pairs of words, each pair with its own port address. Addressing the port with the Byte Pointer Flip-Flop reset accesses the least significant byte of the pair. The most significant byte is accessed when the Byte Pointer is set.

For compatibility with existing 8237A designs, there is one exception to the above statements about the Byte Pointer Flip-Flop. The third byte (bits 16-23) of
the Target Address is accessed through its own port address. The Byte Pointer Flip-Flop is not affected by any accesses to this byte.

The upper eight bits of the Byte Count Register are cleared when the least significant byte of the register is loaded. This provides compatibility with software which accesses an 8237A. The 8237A has 16-bit Byte Count Registers.

### 3.5.3 TEMPORARY REGISTERS

Each channel has a 32-bit Temporary Register used for temporary data storage during two-cycle DMA transfers. It is this register in which any necessary byte assembly and disassembly of non-aligned data is performed. Figure 3-22 shows how a block of data will be moved between memory locations with different boundaries. Note that the order of the data does not change.

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>20H</td>
<td>50H</td>
</tr>
<tr>
<td>21H</td>
<td>51H</td>
</tr>
<tr>
<td>22H</td>
<td>52H</td>
</tr>
<tr>
<td>23H</td>
<td>53H</td>
</tr>
<tr>
<td>24H</td>
<td>54H</td>
</tr>
<tr>
<td>25H</td>
<td>55H</td>
</tr>
<tr>
<td>26H</td>
<td>56H</td>
</tr>
<tr>
<td>27H</td>
<td>57H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>58H</td>
<td>A</td>
</tr>
<tr>
<td>59H</td>
<td>B</td>
</tr>
<tr>
<td>5AH</td>
<td>C</td>
</tr>
</tbody>
</table>

Target = source = 00000020H
Requester = destination = 00000053H
Byte Count = 000007H

Figure 3-22. Transfer of data between memory locations with different boundaries. This will be the result, independent of data path width.

If the destination is the Requester and an early process termination has been indicated by the EOP# signal or DREQn inactive in the Demand Mode, the Temporary Register is not affected. If data remains in the Temporary Register due to differences in data path widths of the Target and Requester, it will not be transferred or otherwise lost, but will be stored for later transfer.

If the destination is the Target and the EOP# signal is sensed active during the Requester access of a transfer, the DMA Controller will complete the transfer by sending to the Target whatever information is in the Temporary Register at the time of process termination. This implies that the Target could be accessed with partial data in two accesses. For this reason it is advisable to have an I/O device designated as a Requester, unless it is capable of handling partial data transfers.

### 3.6 DMA Controller Programming

Programming a DMA Channel to perform a needed DMA function is in general a four step process. First the global attributes of the DMA Controller are programmed via the two Command Registers. These global attributes include: priority levels, channel group enables, priority mode, and DREQn/EOP# input sampling.

The second step involves setting the operating modes of the particular channel. The Mode Registers are used to define the type of transfer and the handshaking modes. The Bus Size Register and Chaining Register may also need to be programmed in this step.

The third step in setting up the channel is to load the Base Registers in accordance with the needs of the operating modes chosen in step two. The Current Registers are automatically loaded from the Base Registers, if required by the Buffer Transfer Mode in effect. The information loaded and the order in which it is loaded depends on the operating mode. A channel used for cascading, for example, needs no buffer information and this step can be skipped entirely.

The last step is to enable the newly programmed channel using one of the Mask Registers. The channel is then available to perform the desired data transfer. The status of the channel can be observed at any time through the Status Register, Mask Register, Chaining Register, and Software Request register.

Once the channel is programmed and enabled, the DMA process may be initiated in one of two ways, either by a hardware DMA request (DREQn) or a software request (Software Request Register).

Once programmed to a particular Process/Mode configuration, the channel will operate in that configuration until programmed otherwise. For this reason, restarting a channel after the current buffer expires does not require complete reprogramming of the channel. Only those parameters which have changed need to be reprogrammed. The Byte Count Register is always changed and must be reprogrammed. A Target or Requester Address Register which is incremented or decremented should be reprogrammed also.
3.6.1 BUFFER PROCESSES

The Buffer Process is determined by the Auto-Initialize bit of Mode Register I and the Chaining Register. If Auto-Initialize is enabled, Chaining should not be used.

3.6.1.1 Single Buffer Process

The Single Buffer Process is programmed by disabling Chaining via the Chaining Register and programming Mode Register I for non-Auto-Initialize.

3.6.1.2 Buffer Auto-Initialize Process

Setting the Auto-Initialize bit in Mode Register I is all that is necessary to place the channel in this mode. Buffer Auto-Initialize must not be enabled simultaneous to enabling the Buffer Chaining Mode as this will have unpredictable results.

Once the Base Registers are loaded, the channel is ready to be enabled. The channel will reload its Current Registers from the Base Registers each time the Current Buffer expires, either by an expired Byte Count or an external EOP#.

3.6.1.3 Buffer Chaining

The Buffer Chaining Process is entered into from the Single Buffer Process. The Mode Registers should be programmed first, with all of the Transfer Modes defined as if the channel were to operate in the Single Buffer Process. The channel’s Base Registers are then loaded. When the channel has been set up in this way, and the chaining interrupt service routine is in place, the Chaining Process can be entered by programming the Chaining Register. Figure 3-23 illustrates the Buffer Chaining Process.

An interrupt (IRQ1) will be generated immediately after the Chaining Process is entered, as the channel then perceives the Base Registers as empty and in need of reloading. It is important to have the interrupt service routine in place at the time the Chaining Process is entered into. The interrupt request is removed when the most significant byte of the Base Target Address is loaded.

The interrupt will occur again when the first buffer expires and the Current Registers are loaded from the Base Registers. The cycle continues until the Chaining Process is disabled, or the host fails to respond to IRQ1 before the Current Buffer expires.

---

**Figure 3-23. Flow of Events in the Buffer Chaining Process**

```
INSTALL IRQ1 INTERRUPT SERVICE ROUTINE

SET THE CHANNEL TO NON-CHAINING PROCESS

PROGRAM THE MODE REGISTERS

LOAD BASE REGISTERS FOR FIRST BUFFER

SET THE CHANNEL TO CHAINING PROCESS

(IRQ1 WILL BE ACTIVATED)

ENABLE INTERRUPT

(IRQ1 WILL NEED SERVICE-LOAD BASE REGISTERS)

ENABLE THE CHANNEL

FROM THIS POINT, THE HOST CAN PERFORM ANOTHER TASK. THE INTERRUPT SERVICE ROUTINE LEFT BEHIND WILL MAINTAIN THE CHANNEL.
```
Exiting the Chaining Process can be done by resetting the Chaining Mode Register. If an interrupt is pending for the channel when the Chaining Register is reset, the interrupt request will be removed. The Chaining Process can be temporarily disabled by setting the channel's Mask bit in the Mask Register.

The interrupt service routine for IRQ1 has the responsibility of reloading the Base Registers as necessary. It should check the status of the channel to determine the cause of channel expiration, etc. It should also have access to operating system information regarding the channel, if any exists. The IRQ1 service routine should be capable of determining whether the chain should be continued or terminated and act on that information.

3.6.2 DATA TRANSFER MODES

The Data Transfer Modes are selected via Mode Register I. The Demand, Single, and Block Modes are selected by bits D6 and D7. The individual transfer type (Fly-By vs Two-Cycle, Read-Write-Verify, and I/O vs Memory) is programmed through both of the Mode registers.

3.6.3 CASCADED BUS MASTERS

The Cascade Mode is set by writing ones to D7 and D6 of Mode Register I. When a channel is programmed to operate in the Cascade Mode, all of the other modes associated with Mode Registers I and II are ignored. The priority and DREQn/EOP# definitions of the Command Registers will have the same effect on the channel's operation as any other mode.

3.6.4 SOFTWARE COMMANDS

There are five port addresses which, when written to, command certain operations to be performed by the 82370 DMA Controller. The data written to these locations is not of consequence, writing to the location is all that is necessary to command the 82370 to perform the indicated function. Following are descriptions of the command functions.

Clear Byte Pointer Flip-Flop—Location 000CH

Resets the Byte Pointer Flip-Flop. This command should be performed at the beginning of any access to the channel registers in order to be assured of beginning at a predictable place in the register programming sequence.

Master Clear—Location 000DH

All DMA functions are set to their default states. This command is the equivalent of a hardware reset to the DMA Controller. Functions other than those in the DMA Controller section of the 82370 are not affected by this command.

Clear Mask Register—Channels 0–3
— Location 000EH
Channels 4–7
— Location 00CEH

This command simultaneously clears the Mask Bits of all channels in the addressed group, enabling all of the channels in the group.

Clear TC Interrupt Request—Location 001EH

This command resets the Terminal Count Interrupt Request Flip-Flop. It is provided to allow the program which made a software DMA request to acknowledge that it has responded to the expiration of the requested channel(s).

3.7 Register Definitions

The following diagrams outline the bit definitions and functions of the 82370 DMA Controller's Status and Control Registers. The function and programming of the registers is covered in the previous section on DMA Controller Programming. An entry of "X" as a bit value indicates "don't care."
<table>
<thead>
<tr>
<th>Channel</th>
<th>Register Name</th>
<th>Address (hex)</th>
<th>Byte Pointer</th>
<th>Bits Accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel 0</td>
<td>Target Address</td>
<td>00</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>87</td>
<td>x</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Byte Count</td>
<td>01</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>0</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Requester Address</td>
<td>90</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>91</td>
<td>1</td>
<td>8–15</td>
</tr>
<tr>
<td>Channel 1</td>
<td>Target Address</td>
<td>02</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>83</td>
<td>x</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Byte Count</td>
<td>03</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>0</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Requester Address</td>
<td>92</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>93</td>
<td>1</td>
<td>8–15</td>
</tr>
<tr>
<td>Channel 2</td>
<td>Target Address</td>
<td>04</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>81</td>
<td>x</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Byte Count</td>
<td>05</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>0</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Requester Address</td>
<td>94</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>95</td>
<td>1</td>
<td>8–15</td>
</tr>
<tr>
<td>Channel 3</td>
<td>Target Address</td>
<td>06</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>82</td>
<td>x</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Byte Count</td>
<td>07</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
<td>0</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Requester Address</td>
<td>96</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>97</td>
<td>1</td>
<td>8–15</td>
</tr>
<tr>
<td>Channel 4</td>
<td>Target Address</td>
<td>C0</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8F</td>
<td>x</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Byte Count</td>
<td>C1</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D1</td>
<td>0</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Requester Address</td>
<td>98</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>99</td>
<td>1</td>
<td>8–15</td>
</tr>
</tbody>
</table>
Channel Registers (read Current, write Base) (Continued)

<table>
<thead>
<tr>
<th>Channel</th>
<th>Register Name</th>
<th>Address (hex)</th>
<th>Byte Pointer</th>
<th>Bits Accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel 5</td>
<td>Target Address</td>
<td>C2</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8B</td>
<td>1</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Byte Count</td>
<td>C3</td>
<td>x</td>
<td>16–23</td>
</tr>
<tr>
<td></td>
<td>Requester Address</td>
<td>D3</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9A</td>
<td>1</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9B</td>
<td>0</td>
<td>16–23</td>
</tr>
<tr>
<td>Channel 6</td>
<td>Target Address</td>
<td>C4</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>89</td>
<td>1</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Byte Count</td>
<td>C5</td>
<td>x</td>
<td>16–23</td>
</tr>
<tr>
<td></td>
<td>Requester Address</td>
<td>D5</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9C</td>
<td>1</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9D</td>
<td>0</td>
<td>16–23</td>
</tr>
<tr>
<td>Channel 7</td>
<td>Target Address</td>
<td>C6</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8A</td>
<td>1</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td>Byte Count</td>
<td>C7</td>
<td>x</td>
<td>16–23</td>
</tr>
<tr>
<td></td>
<td>Requester Address</td>
<td>D7</td>
<td>0</td>
<td>0–7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9E</td>
<td>1</td>
<td>8–15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9F</td>
<td>0</td>
<td>16–23</td>
</tr>
</tbody>
</table>

Command Register I (write only)

Port Addresses—Channels 0–3—0008H
Channels 4–7—00C8H

```
<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>P</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
```

GROUP MASK
0 = ENABLE CHANNELS
1 = DISABLE CHANNELS

PRIORITY
0 = FIXED PRIORITY
1 = ROTATING PRIORITY

290164–36
Command Register II (write only)

Port Addresses—Channels 0–3—001AH
Channels 4–7—00DAH

Mode Register I (write only)

Port Addresses—Channels 0–3—000BH
Channels 4–7—00CBH

*Target and Requester DECREMENT is allowed only for byte transfers.
Mode Register II (write only)

Port Addresses—Channels 0–3—001BH
Channels 4–7—00DBH

*Target and Requester DECREMENT is allowed only for byte transfers.

Software Request Register (read/write)

Port Addresses—Channels 0–3—0009H
Channels 4–7—00C9H

Write Format: Software DMA Service Request
Read Format: Software Requests Pending

```
D7  D6  D5  D4  D3  D2  D1  D0  1 = REQUEST PENDING
X   X   X   X   SR3  SR2  SR1  SR0
```

Mask Set/Reset Register  Individual Channel Mask (write only)

Port Addresses—Channels 0–3—000AH
Channels 4–7—00CAH

```
D7  D6  D5  D4  D3  D2  D1  D0
X   X   X   X   X   M   C1  C0
```

Mask Read/Write Register  Group Channel Mask (read/write)

Port Addresses—Channels 0–3—000FH
Channels 4–7—00CFH

```
D7  D6  D5  D4  D3  D2  D1  D0
X   X   X   X   M3  M2  M1  M0
```
Status Register  Channel Process Status (read only)

Port Addresses—Channels 0–3—0008H
Channels 4–7—00C8H

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R3</td>
<td>R2</td>
<td>R1</td>
<td>RO</td>
<td>TC3</td>
<td>TC2</td>
<td>TC1</td>
</tr>
</tbody>
</table>

- CHANNEL 0(4) EXPIRED
- CHANNEL 1(5) EXPIRED
- CHANNEL 2(6) EXPIRED
- CHANNEL 3(7) EXPIRED
  1 = EXPIRED
- CHANNEL 0(4) REQUEST
- CHANNEL 1(5) REQUEST
- CHANNEL 2(6) REQUEST
- CHANNEL 3(7) REQUEST
  1 = REQUEST PENDING

Bus Size Register  Set Data Path Width (write only)

Port Addresses—Channels 0–3—0018H
Channels 4–7—00D8H

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RBS1</td>
<td>RBS0</td>
<td>TBS1</td>
<td>TBS0</td>
<td>0</td>
<td>0</td>
<td>C1</td>
</tr>
</tbody>
</table>

- CHANNEL SELECT
- SEE MODE REGISTER

Bus Size Encoding:
- 00 = Reserved by Intel
- 01 = 16-bit Bus
- 11 = 8-bit Bus

*If programmed as 32-bit bus width, the corresponding device will be accessed in two 16-bit cycles provided that the data is aligned within word boundary.

Chaining Register (read/write)

Port Addresses—Channels 0–3—0019H
Channels 4–7—00D9H

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>C1</td>
<td>C0</td>
<td></td>
</tr>
</tbody>
</table>

- CHANNEL SELECT
- SEE MODE REGISTER

- CHAINING ENABLE BIT
  0 = DISABLE CHAINING MODE
  1 = ENABLE CHAINING MODE

290164-44
290164-45
290164-46
3.8 8237A Compatibility

The register arrangement of the 82370 DMA Controller is a superset of the 8237A DMA Controller. Functionally the 82370 DMA Controller is very different from the 8237A. Most of the functions of the 8237A are performed also by the 82370. The following discussion points out the differences between the 8237A and the 82370.

The 8237A is limited to transfers between I/O and memory only (except in one special case, where two channels can be used to perform memory-to-memory transfers). The 82370 DMA Controller can transfer between any combination of memory and I/O. Several other features of the 8237A are enhanced or expanded in the 82370 and other features are added.

The 8237A is an 8-bit only DMA device. For programming compatibility, all of the 8-bit registers are preserved in the 82370. The 82370 is programmed via 8-bit registers. The address registers in the 82370 are 24-bit registers in order to support the 80376's 24-bit bus. The Byte Count Registers are 24-bit registers, allowing support of larger data blocks than possible with the 8237A.

All of the 8237A's operating modes are supported by the 82370 (except the cumbersome two-channel memory-to-memory transfer). The 82370 performs memory-to-memory transfers using only one channel. The 82370 has the added features of buffer pipelining (Buffer Chaining Process) and programmable priority levels.

The 82370 also adds the feature of address registers for both destination and source. These addresses may be incremented, decremented, or held constant, as required by the application of the individual channel. This allows any combination of destination and source device.
the PIC consists of three 82C59A banks: A, B and C. The three banks are cascaded to one another: C is cascaded to B, B is cascaded to A. The INT output of Bank A is used externally to interrupt the 80376.

Bank A has nine interrupt request inputs (two are unused), and Banks B and C have eight interrupt request inputs. Of the fifteen external interrupt request inputs, two are shared by other functions. Specifically, the Interrupt Request 3 input (IRQ3#) can be used as the Timer 2 output (TOUT2#). This pin can be used in three different ways: IRQ3# input only, TOUT2# output only, or using TOUT2# to generate an IRQ3# interrupt request. Also, the Interrupt Request 9 input (IRQ9#) can be used as DMA Request 4 input (DREQ 4). Typically, only IRQ9# or DREQ4 can be used at a time.

4.1.2 INTERRUPT CONTROLLER BANKS

All three banks are identical, with the exception of the IRQ1.5 on Bank A. Therefore, only one bank will be discussed. In the 82370 PIC, all external requests can be cascaded into and each interrupt controller bank behaves like a master. As compared to the 82C59A, the enhancements in the banks are:

- All interrupt vectors are individually programmable. (In the 82C59A, the vectors must be programmed in eight consecutive interrupt vector locations.)
- The cascade address is provided on the Data Bus (D0-D7). (In the 82C59A, three dedicated control signals (CAS0, CAS1, CAS2) are used for master/slave cascading.)

![Interrupt Controller Block Diagram](image-url)

Figure 4-1. Interrupt Controller Block Diagram
The block diagram of a bank is shown in Figure 4-2. As can be seen from this figure, the bank consists of six major blocks: the Interrupt Request Register (IRR), the In-Service Register (ISR), the Interrupt Mask Register (IMR), the Priority Resolver (PR), the Vector Registers (VR), and the Control Logic. The functional description of each block is included below.

**INTERRUPT REQUEST (IRR) AND IN-SERVICE REGISTER (ISR)**

The interrupts at the Interrupt Request (IRQ) input lines are handled by two registers in cascade, the Interrupt Request Register (IRR) and the In-Service Register (ISR). The IRR is used to store all interrupt levels which are requesting service; and the ISR is used to store all interrupt levels which are being serviced.

**PRIORITY RESOLVER (PR)**

This logic block determines the priorities of the bits set in the IRR. The highest priority is selected and strobed into the corresponding bit of the ISR during an Interrupt Acknowledge cycle.

**INTERRUPT MASK REGISTER (IMR)**

The IMR stores the bits which mask the interrupt lines to be masked (disabled). The IMR operates on the IRR. Masking of a higher priority input will not affect the interrupt request lines of lower priority.

**VECTOR REGISTERS (VR)**

This block contains a set of Vector Registers, one for each interrupt request line, to store the pre-programmed interrupt vector number. The corresponding vector number will be driven onto the Data Bus of the 82370 during the Interrupt Acknowledge cycle.

**CONTROL LOGIC**

The Control Logic coordinates the overall operations of the other internal blocks within the same bank. This logic will drive the Interrupt Output signal (INT) HIGH when one or more unmasked interrupt inputs are active (LOW). The INT output signal goes directly to the 80376 (in bank A) or to another bank to which this bank is cascaded (see Figure 4-1). Also,
this logic will recognize an Interrupt Acknowledge cycle (via M/IO#, D/C# and W/R# signals). During this bus cycle, the Control Logic will enable the corresponding Vector Register to drive the interrupt vector onto the Data Bus.

In bank A, the Control Logic is also responsible for handling the special ICW2 interrupt request input (IRQ1.5).

### 4.2 Interface Signals

#### 4.2.1 INTERRUPT INPUTS

There are 15 external Interrupt Request inputs and 5 internal Interrupt Requests. The external request inputs are: IRQ3#, IRQ9#, IRQ11# to IRQ23#. They are shown in bold arrows in Figure 4-1. All IRQ inputs are active LOW and they can be programmed (via a control bit in the Initialization Command Word 1 (ICW1)) to be either edge-triggered or level-triggered. In order to be recognized as a valid interrupt request, the interrupt input must be active (LOW) until the first INTA cycle (see Bus Functional Description). Note that all 15 external Interrupt Request inputs have weak internal pull-up resistors.

As mentioned earlier, an 82C59A can be cascaded to each external interrupt input to expand the interrupt capacity to a maximum of 120 levels. Also, two of the interrupt inputs are dual functions: IRQ3# can be used as Timer 2 output (TOUT2#) and IRQ9# can be used as DREQ4 input. IRQ3# is a bidirectional dual function pin. This interrupt request input is wired-OR with the output of Timer 2 (TOUT2#). If only IRQ3# function is to be used, Timer 2 should be programmed so that OUT2 is LOW. Note that TOUT2# can also be used to generate an interrupt request to IRQ3# input.

The five internal interrupt requests serve special system functions. They are shown in Table 4-1. The following paragraphs describe these interrupts.

<table>
<thead>
<tr>
<th>Table 4-1. 82370 Internal Interrupt Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interrupt Request</strong></td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>IRQ0#</td>
</tr>
<tr>
<td>IRQ8#</td>
</tr>
<tr>
<td>IRQ1#</td>
</tr>
<tr>
<td>IRQ4#</td>
</tr>
<tr>
<td>IRQ1.5#</td>
</tr>
</tbody>
</table>

#### TIMER 0 AND TIMER 3 INTERRUPT REQUESTS

IRQ8# and IRQ0# interrupt requests are initiated by the output of Timers 0 and 3, respectively. Each of these requests is generated by an edge-detector flip-flop.

The flip-flops are activated by the following conditions:

Set — Rising edge of timer output (TOUT);
Clear — Interrupt acknowledge for this request; OR Request is masked (disabled); OR Hardware Reset.

#### CHAINING AND TERMINAL COUNT INTERRUPTS

These interrupt requests are generated by the 82370 DMA Controller. The chaining request (IRQ1#) indicates that the DMA Base Register is not loaded. The Terminal Count request (IRQ4#) indicates that a software DMA request was cleared.

#### DEFAULT INTERRUPT (IRQ7#)

During an Interrupt Acknowledge cycle, if there is no active pending request, the PIC will automatically generate a default vector. This vector corresponds to the IRQ7# vector in bank A.

#### 4.2.2 INTERRUPT OUTPUT (INT)

The INT output pin is taken directly from bank A. This signal should be tied to the Maskable Interrupt Request (INTR) of the 80376. When this signal is active (HIGH), it indicates that one or more internal/external interrupt requests are pending. The 80376 is expected to respond with an interrupt acknowledge cycle.

#### 4.3 Bus Functional Description

The INT output of bank A will be activated as a result of any unmasked interrupt request. This may be a non-cascaded or cascaded request. After the PIC has driven the INT signal HIGH, the 80376 will respond by performing two interrupt acknowledge cycles. The timing diagram in Figure 4-3 shows a typical interrupt acknowledge process between the 82370 and the 80376 CPU.
After activating the INT signal, the 82370 monitors the status lines (M/IO#, D/C#, W/R#) and waits for the 80376 to initiate the first interrupt acknowledge cycle. In the 80376 environment, two successive interrupt acknowledge cycles (INTA) marked by M/IO# = LOW, D/C# = LOW, and W/R# = LOW are performed. During the first INTA cycle, the PIC will determine the highest priority request. Assuming this interrupt input has no external Slave Controller cascaded to it, the 82370 will drive the Data Bus with 00H in the first INTA cycle. During the second INTA cycle, the 82370 PIC will drive the Data Bus with the corresponding pre-programmed interrupt vector.

If the PIC determines (from the ICW3) that this interrupt input has an external Slave Controller cascaded to it, it will drive the Data Bus with the specific Slave Cascade Address (instead of 00H) during the first INTA cycle. This Slave Cascade Address is the pre-programmed content in the corresponding Vector Register. This means that no Slave Address should be chosen to be 00H. Note that the Slave Address and Interrupt Vector are different interpretations of the same thing. They are both the contents of the programmable Vector Register. During the second INTA cycle, the Data Bus will be floated so that the external Slave Controller can drive its interrupt vector on the bus. Since the Slave Interrupt Controller resides on the system bus, bus transceiver enable and direction control logic must take this into consideration.

In order to have a successful interrupt service, the interrupt request input must be held valid (LOW) until the beginning of the first interrupt acknowledge cycle. If there is no pending interrupt request when the first INTA cycle is generated, the PIC will generate a default vector, which is the IRQ7 vector (Bank A, level 7).

According to the Bus Cycle definition of the 80376, there will be four Bus Idle States between the two interrupt acknowledge cycles. These idle bus cycles will be initiated by the 80376. Also, during each interrupt acknowledge cycle, the internal Wait State Generator of the 82370 will automatically generate the required number of wait states for internal delays.

### 4.4 Modes of Operation

A variety of modes and commands are available for controlling the 82370 PIC. All of them are programable; that is, they may be changed dynamically under software control. In fact, each bank can be programmed individually to operate in different modes. With these modes and commands, many possible configurations are conceivable, giving the user enough versatility for almost any interrupt controlled application.

This section is not intended to show how the 82370 PIC can be programmed. Rather, it describes the operation in different modes.
4.4.1 END-OF-INTERRUPT

Upon completion of an interrupt service routine, the interrupted bank needs to be notified so its ISR can be updated. This allows the PIC to keep track of which interrupt levels are in the process of being serviced and their relative priorities. Three different End-Of-Interrupt (EOI) formats are available. They are: Non-Specific EOI Command, Specific EOI Command, and Automatic EOI Mode. Selection of which EOI to use is dependent upon the interrupt operations the user wishes to perform.

If the 82370 is NOT programmed in the Automatic EOI Mode, an EOI command must be issued by the 80376 to the specific 82370 PIC Controller Bank. Also, if this controller bank is cascaded to another internal bank, an EOI command must also be sent to the bank to which this bank is cascaded. For example, if an interrupt request of Bank C in the 82370 PIC is serviced, an EOI should be written into Bank C, Bank B and Bank A. If the request comes from an external interrupt controller cascaded to Bank C, then an EOI should be written into the external controller as well.

NON-SPECIFIC EOI COMMAND

A Non-Specific EOI command sent from the 80376 lets the 82370 PIC bank know when a service routine has been completed, without specification of its exact interrupt level. The respective interrupt bank automatically determines the interrupt level and resets the correct bit in the ISR.

To take advantage of the Non-Specific EOI, the interrupt bank must be in a mode of operation in which it can predetermine its in-service routine levels. For this reason, the Non-Specific EOI command should only be used when the most recent level acknowledged and serviced is always the highest priority level (i.e. in the Fully Nested Mode structure to be described below). When the interrupt bank receives a Non-Specific EOI command, it simply resets the highest priority ISR bit to indicate that the highest priority routine in service is finished.

Special consideration should be taken when deciding to use the Non-Specific EOI command. Here are two operating conditions in which it is best NOT used since the Fully Nested Mode structure will be destroyed:

— Using the Set Priority command within an interrupt service routine.
— Using a Special Mask Mode.

These conditions are covered in more detail in their own sections, but are listed here for reference.

SPECIFIC EOI COMMAND

Unlike a Non-Specific EOI command which automatically resets the highest priority ISR bit, a Specific EOI command specifies an exact ISR bit to be reset. Any one of the IRQ levels of an interrupt bank can be specified in the command.

The Specific EOI command is needed to reset the ISR bit of a completed service routine whenever the interrupt bank is not able to automatically determine it. The Specific EOI command can be used in all conditions of operation, including those that prohibit Non-Specific EOI command usage mentioned above.

AUTOMATIC EOI MODE

When programmed in the Automatic EOI Mode, the 80376 no longer needs to issue a command to notify the interrupt bank it has completed an interrupt routine. The interrupt bank accomplishes this by performing a Non-Specific EOI automatically at the end of the second INTA cycle.

Special consideration should be taken when deciding to use the Automatic EOI Mode because it may disturb the Fully Nested Mode structure. In the Automatic EOI Mode, the ISR bit of a routine in service is reset right after it is acknowledged, thus leaving no designation in the ISR that a service routine is being executed. If any interrupt request within the same bank occurs during this time and interrupts are enabled, it will get serviced regardless of its priority. Therefore, when using this mode, the 80376 should keep its interrupt request input disabled during execution of a service routine. By doing this, higher priority interrupt levels will be serviced only after the completion of a routine in service. This guideline restores the Fully Nested Mode structure. However, in this scheme, a routine in service cannot be interrupted since the host's interrupt request input is disabled.

4.4.2 INTERRUPT PRIORITIES

The 82370 PIC provides various methods for arranging the interrupt priorities of the interrupt request inputs to suit different applications. The following subsections explain these methods in detail.

4.4.2.1 Fully Nested Mode

The Fully Nested Mode of operation is a general purpose priority mode. This mode supports a multi-level interrupt structure in which all of the Interrupt Request (IRQ) inputs within one bank are arranged from highest to lowest.
Unless otherwise programmed, the Fully Nested Mode is entered by default upon initialization. At this time, IRQ0# is assigned the highest priority (priority = 0) and IRQ7# the lowest (priority = 7). This default priority can be changed, as will be explained later in the Rotating Priority Mode.

When an interrupt is acknowledged, the highest priority request is determined from the Interrupt Request Register (IRR) and its vector is placed on the bus. In addition, the corresponding bit in the In-Service Register (ISR) is set to designate the routine in service. This ISR bit will remain set until the 80376 issues an End Of Interrupt (EOI) command immediately before returning from the service routine; or alternately, if the Automatic End Of Interrupt (AEOI) bit is set, the ISR bit will be reset at the end of the second INTA cycle.

While the ISR bit is set, all further interrupts of the same or lower priority are inhibited. Higher level interrupts can still generate an interrupt, which will be acknowledged only if the 80376 internal interrupt enable flip-flop has been reenabled (through software inside the current service routine).

### 4.4.2.2 Automatic Rotation—Equal Priority Devices

Automatic rotation of priorities serves in applications where the interrupting devices are of equal priority within an interrupt bank. In this kind of environment, once a device is serviced, all other equal priority peripherals should be given a chance to be serviced before the original device is serviced again. This is accomplished by automatically assigning a device the lowest priority after being serviced. Thus, in the worst case, the device would have to wait until all other peripherals connected to the same bank are serviced before it is serviced again.

There are two methods of accomplishing automatic rotation. One is used in conjunction with the Non-Specific EOI command and the other is used with the Automatic EOI mode. These two methods are discussed below.

#### ROTATE ON NON-SPECIFIC EOI COMMAND

When the Rotate On Non-Specific EOI command is issued, the highest ISR bit is reset as in a normal Non-Specific EOI command. However, after it is reset, the corresponding Interrupt Request (IRQ) level is assigned the lowest priority. Other IRQ priorities rotate to conform to the Fully Nested Mode based on the newly assigned low priority.

Figure 4-4 shows how the Rotate On Non-Specific EOI command affects the interrupt priorities. Assume the IRQ priorities were assigned with IRQ0 the highest and IRQ7 the lowest. IRQ6 and IRQ4 are
already in service but neither is completed. Being the higher priority routine, IRQ4 is necessarily the routine being executed. During the IRQ4 routine, a rotate on Non-Specific EOI command is executed. When this happens, Bit 4 in the ISR is reset. IRQ4 then becomes the lowest priority and IRQ5 becomes the highest.

**ROTATE ON AUTOMATIC EOI MODE**

The Rotate On Automatic EOI Mode works much like the Rotate On Non-Specific EOI Command. The main difference is that priority rotation is done automatically after the second INTA cycle of an interrupt request. To enter or exit this mode, a Rotate-On-Automatic-EOI Set Command and Rotate-On-Automatic-EOI Clear Command is provided. After this mode is entered, no other commands are needed as in the normal Automatic EOI Mode. However, it must be noted again that when using any form of the Automatic EOI Mode, special consideration should be taken. The guideline presented in the Automatic EOI Mode also applies here.

### 4.4.2.3 Specific Rotation—Specific Priority

Specific rotation gives the user versatile capabilities in interrupt controlled operations. It serves in those applications in which a specific device's interrupt priority must be altered. As opposed to Automatic Rotation which will automatically set priorities after each interrupt request is serviced, specific rotation is completely user controlled. That is, the user selects which interrupt level is to receive the lowest or the highest priority. This can be done during the main program or within interrupt routines. Two specific rotation commands are available to the user: Set Priority Command and Rotate On Specific EOI Command.

**SET PRIORITY COMMAND**

The Set Priority Command allows the programmer to assign an IRQ level the lowest priority. All other interrupt levels will conform to the Fully Nested Mode based on the newly assigned low priority.

**ROTATE ON SPECIFIC EOI COMMAND**

The Rotate On Specific EOI Command is literally a combination of the Set Priority Command and the Specific EOI Command. Like the Set Priority Command, a specified IRQ level is assigned lowest priority. Like the Specific EOI Command, a specified level will be reset in the ISR. Thus, this command accomplishes both tasks in one single command.

#### 4.4.2.4 Interrupt Priority Mode Summary

In order to simplify understanding the many modes of interrupt priority, Table 4-2 is provided to bring out their summary of operations.

### 4.4.3 INTERRUPT MASKING

**VIA INTERRUPT MASK REGISTER**

Each bank in the 82370 PIC has an Interrupt Mask Register (IMR) which enhances interrupt control ca-

<table>
<thead>
<tr>
<th>Interrupt Priority Mode</th>
<th>Operation Summary</th>
<th>Effect On Priority After EOI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Non-Specific/Automatic</td>
</tr>
<tr>
<td>Fully-Nested Mode</td>
<td>IRQ0# - Highest Priority</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IRQ7# - Lowest Priority</td>
<td>No change in priority.</td>
</tr>
<tr>
<td></td>
<td>Highest ISR bit is reset.</td>
<td></td>
</tr>
<tr>
<td>Automatic Rotation</td>
<td>Interrupt level just serviced is the lowest priority.</td>
<td></td>
</tr>
<tr>
<td>(Equal Priority Devices)</td>
<td>Other priorities rotate to conform to Fully-Nested Mode.</td>
<td>Highest ISR bit is reset and the corresponding level becomes the lowest priority.</td>
</tr>
<tr>
<td>Specific Rotation</td>
<td>User specifies the lowest priority level.</td>
<td></td>
</tr>
<tr>
<td>(Specific Priority Devices)</td>
<td>Other priorities rotate to conform to Fully-Nested Mode.</td>
<td>Not Applicable.</td>
</tr>
</tbody>
</table>

Table 4-2. Interrupt Priority Mode Summary
pabilities. This IMR allows individual IRQ masking. When an IRQ is masked, its interrupt request is disabled until it is unmasked. Each bit in the 8-bit IMR disables one interrupt channel if it is set (HIGH). Bit 0 masks IRQ0, Bit 1 masks IRQ1 and so forth. Masking an IRQ channel will only disable the corresponding channel and does not affect the others' operations.

The IMR acts only on the output of the IRR. That is, if an interrupt occurs while its IMR bit is set, this request is not "forgotten". Even with an IRQ input masked, it is still possible to set the IRR. Therefore, when the IMR bit is reset, an interrupt request to the 80376 will then be generated, providing that the IRQ request remains active. If the IRQ request is removed before the IMR is reset, the Default Interrupt Vector (Bank A, level 7) will be generated during the interrupt acknowledge cycle.

SPECIAL MASK MODE

In the Fully Nested Mode, all IRQ levels of lower priority than the routine in service are inhibited. However, in some applications, it may be desirable to let a lower priority interrupt request to interrupt the routine in service. One method to achieve this is by using the Special Mask Mode. Working in conjunction with the IMR, the Special Mask Mode enables interrupts from all levels except the level in service. This is usually done inside an interrupt service routine by masking the level that is in service and then issuing the Special Mask Mode Command. Once the Special Mask Mode is enabled, it remains in effect until it is disabled.

4.4.4 EDGE OR LEVEL INTERRUPT TRIGGERING

Each bank in the 82370 PIC can be programmed independently for either edge or level sensing for the interrupt request signals. Recall that all IRQ inputs are active LOW. Therefore, in the edge triggered mode, an active edge is defined as an input transition from an inactive (HIGH) to active (LOW) state. The interrupt input may remain active without generating another interrupt. During level triggered mode, an interrupt request will be recognized by an active (LOW) input, and there is no need for edge detection. However, the interrupt request must be removed before the EOI Command is issued, or the 80376 must be disabled to prevent a second false interrupt from occurring.

In either modes, the interrupt request input must be active (LOW) during the first INTA cycle in order to be recognized. Otherwise, the Default Interrupt Vector will be generated at level 7 of Bank A.

4.4.5 INTERRUPT CASCADING

As mentioned previously, the 82370 allows for external Slave interrupt controllers to be cascaded to any of its external interrupt request pins. The 82370 PIC indicates that an external Slave Controller is to be serviced by putting the contents of the Vector Register associated with the particular request on the 80376 Data Bus during the first INTA cycle (instead of 00H during a non-slave service). The external logic should latch the vector on the Data Bus using the INTA status signals and use it to select the external Slave Controller to be serviced (see Figure 4-5). The selected Slave will then respond to the second INTA cycle and place its vector on the Data Bus. This method requires that if external Slave Controllers are used in the system, no vector should be programmed to 00H.

Since the external Slave Cascade Address is provided on the Data Bus during INTA cycle 1, an external latch is required to capture this address for the Slave Controller. A simple scheme is depicted in Figure 4-5 below.
4.4.5.1 Special Fully Nested Mode

This mode will be used where cascading is employed and the priority is to be conserved within each Slave Controller. The Special Fully Nested Mode is similar to the "regular" Fully Nested Mode with the following exceptions:

- When an interrupt request from a Slave Controller is in service, this Slave Controller is not locked out from the Master's priority logic. Further interrupt requests from the higher priority logic within the Slave Controller will be recognized by the 82370 PIC and will initiate interrupts to the 80376. In comparing to the "regular" Fully Nested Mode, the Slave Controller is masked out when its request is in service and no higher requests from the same Slave Controller can be serviced.

- Before exiting the interrupt service routine, the software has to check whether the interrupt serviced was the only request from the Slave Controller. This is done by sending a Non-Specific EOI Command to the Slave Controller and then reading its In Service Register. If there are no requests in the Slave Controller, a Non-Specific EOI can be sent to the corresponding 82370 PIC bank also. Otherwise, no EOI should be sent.

4.4.6 READING INTERRUPT STATUS

The 82370 PIC provides several ways to read different status of each interrupt bank for more flexible interrupt control operations. These include polling the highest priority pending interrupt request and reading the contents of different interrupt status registers.

4.4.6.1 Poll Command

The 82370 PIC supports status polling operations with the Poll Command. In a Poll Command, the pending interrupt request with the highest priority can be determined. To use this command, the INT output is not used, or the 80376 interrupt is disabled. Service to devices is achieved by software using the Poll Command.

This mode is useful if there is a routine command common to several levels so that the INTA sequence is not needed. Another application is to use the Poll Command to expand the number of priority levels.

Notice that the ICW2 mechanism is not supported for the Poll Command. However, if the Poll Command is used, the programmable Vector Registers are of no concern since no INTA cycle will be generated.

4.4.6.2 Reading Interrupt Registers

The contents of each interrupt register (IRR, ISR, and IMR) can be read to update the user's program on the present status of the 82370 PIC. This can be a versatile tool in the decision making process of a service routine, giving the user more control over interrupt operations.

The reading of the IRR and ISR contents can be performed via the Operation Control Word 3 by using a Read Status Register Command and the content of IMR can be read via a simple read operation of the register itself.

4.5 Register Set Overview

Each bank of the 82370 PIC consists of a set of 8-bit registers to control its operations. The address map of all the registers is shown in Table 4-3 below. Since all three register sets are identical in functions, only one set will be described.

Functionally, each register set can be divided into five groups. They are: the four Initialization Command Words (ICW's), the three Operation Control Words (OCW's), the Poll/Interrupt Request/In-Service Register, the Interrupt Mask Register, and the Vector Registers. A description of each group follows.
### Table 4-3. Interrupt Controller Register Address Map

<table>
<thead>
<tr>
<th>Port Address</th>
<th>Access</th>
<th>Register Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20H</td>
<td>Write</td>
<td>Bank B ICW1, OCW2, or OCW3</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>Bank B Poll, Request or In-Service Status Register</td>
</tr>
<tr>
<td>21H</td>
<td>Write</td>
<td>Bank B ICW2, ICW3, ICW4, OCW1</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>Bank B Mask Register</td>
</tr>
<tr>
<td>22H</td>
<td>Read</td>
<td>Bank B ICW2</td>
</tr>
<tr>
<td>28H</td>
<td>Read/Write</td>
<td>IRQ8 Vector Register</td>
</tr>
<tr>
<td>29H</td>
<td>Read/Write</td>
<td>IRQ9 Vector Register</td>
</tr>
<tr>
<td>2AH</td>
<td>Read/Write</td>
<td>IRQ11 Vector Register</td>
</tr>
<tr>
<td>2BH</td>
<td>Read/Write</td>
<td>IRQ12 Vector Register</td>
</tr>
<tr>
<td>2CH</td>
<td>Read/Write</td>
<td>IRQ13 Vector Register</td>
</tr>
<tr>
<td>2DH</td>
<td>Read/Write</td>
<td>IRQ14 Vector Register</td>
</tr>
<tr>
<td>2EH</td>
<td>Read/Write</td>
<td>IRQ15 Vector Register</td>
</tr>
<tr>
<td>2FH</td>
<td>Read/Write</td>
<td>Bank C ICW1, OCW2, or OCW3</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>Bank C Poll, Request or In-Service Status Register</td>
</tr>
<tr>
<td>A0H</td>
<td>Write</td>
<td>Bank C ICW2, ICW3, ICW4, OCW1</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>Bank C Mask Register</td>
</tr>
<tr>
<td>A1H</td>
<td>Write</td>
<td>Bank C ICW2, ICW3, ICW4, OCW1</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>Bank C Mask Register</td>
</tr>
<tr>
<td>A2H</td>
<td>Read</td>
<td>Bank C ICW2</td>
</tr>
<tr>
<td>A8H</td>
<td>Read/Write</td>
<td>IRQ16 Vector Register</td>
</tr>
<tr>
<td>A9H</td>
<td>Read/Write</td>
<td>IRQ17 Vector Register</td>
</tr>
<tr>
<td>AAH</td>
<td>Read/Write</td>
<td>IRQ18 Vector Register</td>
</tr>
<tr>
<td>ABH</td>
<td>Read/Write</td>
<td>IRQ19 Vector Register</td>
</tr>
<tr>
<td>ACH</td>
<td>Read/Write</td>
<td>IRQ20 Vector Register</td>
</tr>
<tr>
<td>ADH</td>
<td>Read/Write</td>
<td>IRQ21 Vector Register</td>
</tr>
<tr>
<td>AEH</td>
<td>Read/Write</td>
<td>IRQ22 Vector Register</td>
</tr>
<tr>
<td>AFH</td>
<td>Read/Write</td>
<td>IRQ23 Vector Register</td>
</tr>
<tr>
<td>30H</td>
<td>Write</td>
<td>Bank A ICW1, OCW2, or OCW3</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>Bank A Poll, Request or In-Service Status Register</td>
</tr>
<tr>
<td>31H</td>
<td>Write</td>
<td>Bank A ICW2, ICW3, ICW4, OCW1</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>Bank A Mask Register</td>
</tr>
<tr>
<td>32H</td>
<td>Read</td>
<td>Bank ICW2</td>
</tr>
<tr>
<td>38H</td>
<td>Read/Write</td>
<td>IRQ0 Vector Register</td>
</tr>
<tr>
<td>39H</td>
<td>Read/Write</td>
<td>IRQ1 Vector Register</td>
</tr>
<tr>
<td>3AH</td>
<td>Read/Write</td>
<td>IRQ1.5 Vector Register</td>
</tr>
<tr>
<td>3BH</td>
<td>Read/Write</td>
<td>IRQ3 Vector Register</td>
</tr>
<tr>
<td>3CH</td>
<td>Read/Write</td>
<td>IRQ4 Vector Register</td>
</tr>
<tr>
<td>3DH</td>
<td>Read/Write</td>
<td>Reserved</td>
</tr>
<tr>
<td>3EH</td>
<td>Read/Write</td>
<td>Reserved</td>
</tr>
<tr>
<td>3FH</td>
<td>Read/Write</td>
<td>IRQ7 Vector Register</td>
</tr>
</tbody>
</table>
4.5.1 INITIALIZATION COMMAND WORDS (ICW)

Before normal operation can begin, the 82370 PIC must be brought to a known state. There are four 8-bit Initialization Command Words in each interrupt bank to setup the necessary conditions and modes for proper operation. Except for the second command word (ICW2) which is a read/write register, the other three are write-only registers. Without going into detail of the bit definitions of the command words, the following subsections give a brief description of what functions each command word controls.

ICW1

The ICW1 has three major functions. They are:
- To select between the two IRQ input triggering modes (edge- or level-triggered);
- To designate whether or not the interrupt bank is to be used alone or in the cascade mode. If the cascade mode is desired, the interrupt bank will accept ICW3 for further cascade mode programming. Otherwise, no ICW3 will be accepted;
- To determine whether or not ICW4 will be issued; that is, if any of the ICW4 operations are to be used.

ICW2

ICW2 is provided for compatibility with the 82C59A only. Its contents do not affect the operation of the interrupt bank in any way. Whenever the ICW2 of any of the three banks is written into, an interrupt is generated from bank A at level 1.5. The interrupt request will be cleared after the ICW2 register has been read by the 80376. The user is expected to program the corresponding vector register or to use it as an indicator that an attempt was made to alter the contents. Note that each ICW2 register has different addresses for read and write operations.

ICW3

The interrupt bank will only accept an ICW3 if programmed in the external cascade mode (as indicated in ICW1). ICW3 is used for specific programming within the cascade mode. The bits in ICW3 indicate which interrupt request inputs have a Slave cascaded to them. This will subsequently affect the interrupt vector generation during the interrupt acknowledge cycles as described previously.

ICW4

The ICW4 is accepted only if it was selected in ICW1. This command word register serves two functions:
- To select either the Automatic EOI mode or software EOI mode;
- To select if the Special Nested mode is to be used in conjunction with the cascade mode.

4.5.2 OPERATION CONTROL WORDS (OCW)

Once initialized by the ICW's, the interrupt banks will be operating in the Fully Nested Mode by default and they are ready to accept interrupt requests. However, the operations of each interrupt bank can be further controlled or modified by the use of OCW's. Three OCW's are available for programming various modes and commands. Note that all OCW's are 8-bit write-only registers.

The modes and operations controlled by the OCW's are:
- Fully Nested Mode;
- Rotating Priority Mode;
- Special Mask Mode;
- Poll Mode;
- EOI Commands;
- Read Status Commands.

OCW1

OCW1 is used solely for masking operations. It provides a direct link to the Internal Mask Register (IMR). The 80376 can write to this OCW register to enable or disable the interrupt inputs. Reading the pre-programmed mask can be done via the Interrupt Mask Register which will be discussed shortly.

OCW2

OCW2 is used to select End-Of-Interrupt, Automatic Priority Rotation, and Specific Priority Rotation operations. Associated commands and modes of these operations are selected using the different combinations of bits in OCW2.

Specifically, the OCW2 is used to:
- Designate an interrupt level (0–7) to be used to reset a specific ISR bit or to set a specific priority. This function can be enabled or disabled;
- Select which software EOI command (if any) is to be executed (i.e. Non-Specific or Specific EOI);
- Enable one of the priority rotation operations (i.e. Rotate On Non-Specific EOI, Rotate On Automatic EOI, or Rotate On Specific EOI).
OCW3

There are three main categories of operation that OCW3 controls. They are summarized as follows:

— To select and execute the Read Status Register Commands, either reading the Interrupt Request Register (IRR) or the In-Service Register (ISR);
— To issue the Poll Command. The Poll Command will override a Read Register Command if both functions are enabled simultaneously;
— To set or reset the Special Mask Mode.

4.5.3 POLL/INTERRUPT REQUEST/IN-SERVICE STATUS REGISTER

As the name implies, this 8-bit read-only register has multiple functions. Depending on the command issued in the OCW3, the content of this register reflects the result of the command executed. For a Poll Command, the register read contains the binary code of the highest priority level requesting service (if any). For a Read IRR Command, the register content will show the current pending interrupt request(s). Finally, for a Read ISR Command, this register will specify all interrupt levels which are being serviced.

4.5.4 INTERRUPT MASK REGISTER (IMR)

This is a read-only 8-bit register which, when read, will specify all interrupt levels within the same bank that are masked.

4.5.5 VECTOR REGISTERS (VR)

Each interrupt request input has an 8-bit read/write programmable vector register associated with it. The registers should be programmed to contain the interrupt vector for the corresponding request. The contents of the Vector Register will be placed on the Data Bus during the INTA cycles as described previously.

4.6 Programming

Programming the 82370 PIC is accomplished by using two types of command words: ICW's and OCW's. All modes and commands explained in the previous sections are programmable using the ICW's and OCW's. The ICW's are issued from the 80376 in a sequential format and are used to setup the banks in the 82370 PIC in an initial state of operation. The OCW's are issued as needed to vary and control the 82370 PIC's operations.

Both ICW's and OCW's are sent by the 80376 to the interrupt banks via the Data Bus. Each bank distinguishes between the different ICW's and OCW's by the I/O address map, the sequence they are issued (ICW's only), and by some dedicated bits among the ICW's and OCW's.

An example of programming the 82370 interrupt controllers is given in Appendix C (Programming the 82370 Interrupt Controllers).

All three interrupt banks are programmed in a similar way. Therefore, only a single bank will be described in the following sections.

4.6.1 INITIALIZATION (ICW)

Before normal operation can begin, each bank must be initialized by programming a sequence of two to four bytes written into the ICW's.

Figure 4-6 shows the initialization flow for an interrupt bank. Both ICW1 and ICW2 must be issued for any form of operation. However, ICW3 and ICW4 are used only if designated in ICW1. Once initialized, if any programming changes within the ICW's are to be made, the entire ICW sequence must be reprogrammed, not just an individual ICW.

Note that although the ICW2's in the 82370 PIC do not effect the Bank's operation, they still must be programmed in order to preserve the compatibility with the 82C59A. The contents programmed are not relevant to the overall operations of the interrupt banks. Also, whenever one of the three ICW2's is programmed, an interrupt level 1.5 in Bank A will be generated. This interrupt request will be cleared upon reading of the ICW2 registers. Since the three ICW2's share the same interrupt level and the system may not know the origin of the interrupt, all three ICW2's must be read.
Certain internal setup conditions occur automatically within the interrupt bank after the first ICW (ICW1) has been issued. These are:

- The edge sensitive circuit is reset, which means that following initialization, an interrupt request input must make a HIGH-to-LOW transition to generate an interrupt;
- The Interrupt Mask Register (IMR) is cleared; that is, all interrupt inputs are enabled;
- IRQ7 input of each bank is assigned priority 7 (lowest);
- Special Mask Mode is cleared and Status Read is set to IRR;
- If no ICW4 is needed, then no Automatic-EOI is selected.

**4.6.2 VECTOR REGISTERS (VR)**

Each interrupt request input has a separate Vector Register. These Vector Registers are used to store the pre-programmed vector number corresponding to their interrupt sources. In order to guarantee proper interrupt handling, all Vector Registers must be programmed with the predefined vector numbers. Since an interrupt request will be generated whenever an ICW2 is written during the initialization sequence, it is important that the Vector Register of IRQ1.5 in Bank A should be initialized and the interrupt service routine of this vector is set up before the ICW’s are written.
4.6.3 OPERATION CONTROL WORDS (OCW)

After the ICW's are programmed, the operations of each interrupt controller bank can be changed by writing into the OCW's as explained before. There is no special programming sequence required for the OCW's. Any OCW may be written at any time in order to change the mode of or to perform certain operations on the interrupt banks.

4.6.3.1 Read Status and Poll Commands (OCW3)

Since the reading of IRR and ISR status as well as the result of a Poll Command are available on the same read-only Status Register, a special Read Status/Poll Command must be issued before the Poll/Interrupt Request/In-Service Status Register is read. This command can be specified by writing the required control word into OCW3. As mentioned earlier, if both the Poll Command and the Status Read Command are enabled simultaneously, the Poll Command will override the Status Read. That is, after the command execution, the Status Register will contain the result of the Poll Command.

Note that for reading IRR and ISR, there is no need to issue a Read Status Command to the OCW3 every time the IRR or ISR is to be read. Once a Read Status Command is received by the interrupt bank, it "remembers" which register is selected. However, this is not true when the Poll Command is used.

In the Poll Command, after the OCW3 is written, the 82370 PIC treats the next read to the Status Register as an interrupt acknowledge. This will set the appropriate IS bit if there is a request and read the priority level. Interrupt Request input status remains unchanged from the Poll Command to the Status Read.

In addition to the above read commands, the Interrupt Mask Register (IMR) can also be read. When read, this register reflects the contents of the pre-programmed OCW1 which contains information on which interrupt request(s) is(are) currently disabled.

4.7 Register Bit Definition

INITIALIZATION COMMAND WORD 1 (ICW1)

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>LTIM</td>
<td>X</td>
<td>SNGL</td>
<td>IC4</td>
</tr>
</tbody>
</table>

0 = NO ICW3 NEEDED
1 = ICW4 NEEDED

0 = EXTERNAL CASCADE (ICW3 NEEDED)
1 = NO EXTERNAL CASCADE (ICW3 NOT NEEDED)

D7 D6 D5 D4 D3 D2 D1 D0

CONTENT IS NOT RELEVANT TO THE ACTUAL OPERATION OF THE BANK BUT CAN BE READ BY THE INTERRUPT SERVICE ROUTINE TO DETERMINE WHERE THE INTERRUPT VECTORS OF EACH BANK START.
INITIALIZATION COMMAND WORD 3 (ICW3)

ICW3 for Bank A:

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

0 = NO SLAVE CASCADED TO BANK A
1 = THERE IS A SLAVE CASCADED TO TOUT2#/IRQ3# PIN

ICW3 for Bank B:

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S15</td>
<td>S14</td>
<td>S13</td>
<td>S12</td>
<td>S11</td>
<td>X</td>
<td>S9</td>
<td>0</td>
</tr>
</tbody>
</table>

0 = NO CASCADED REQUEST TO IRQN
1 = THERE IS A CASCADED REQUEST CONNECTED TO IRQN (I.E. THE CORRESPONDING INTERRUPT REQUEST INPUTS)

ICW3 for Bank C:

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S23</td>
<td>S22</td>
<td>S21</td>
<td>S20</td>
<td>S19</td>
<td>S18</td>
<td>S17</td>
<td>S16</td>
</tr>
</tbody>
</table>

0 = NO CASCADED REQUEST TO IRQN
1 = THERE IS A CASCADED REQUEST CONNECTED TO IRQN

INITIALIZATION COMMAND WORD 4 (ICW4)

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S7FNM</td>
<td>X</td>
<td>X</td>
<td>AEOI</td>
<td>X</td>
</tr>
</tbody>
</table>

0 = NORMAL EOI
1 = AUTOMATIC EOI

0 = NOT SPECIAL FULLY NESTED MODE
1 = SPECIAL FULLY NESTED MODE
### OPERATION CONTROL WORD 1 (OCW1)

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>M7</td>
<td>M6</td>
<td>M5</td>
<td>M4</td>
<td>M3</td>
<td>M2</td>
<td>M1</td>
<td>M0</td>
</tr>
</tbody>
</table>

- **Mi = 1** Mask Set (Interrupt disabled)
- **Mi = 0** Mask Reset (Interrupt enabled)

### OPERATION CONTROL WORD 2 (OCW2)

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>S</td>
<td>E</td>
<td>O1</td>
<td>0</td>
<td>0</td>
<td>L2</td>
<td>L0</td>
</tr>
</tbody>
</table>

- **0 0 1** Non-Specific EOI Command
- **0 1 1** Specific EOI Command
- **1 0 1** Rotate on Non-Specific EOI
- **1 0 0** Rotate on Auto-EOI Mode (Set)
- **0 0 0** Rotate on Auto-EOI Mode (Clear)
- **1 1 1** Rotate on Specific EOI (L2-L0 Used)
- **1 1 0** Set Priority (L2-L0 Used)
- **0 1 0** No Operation

### OPERATION CONTROL WORD 3 (OCW3)

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ESMM</td>
<td>SMM</td>
<td>0</td>
<td>1</td>
<td>P</td>
<td>RR</td>
<td>RIS</td>
</tr>
</tbody>
</table>

- **ESMM** — Enable Special Mask Mode. When this bit is set to 1, it enables the SMM bit to set or reset the Special Mask Mode. When this bit is set to 0, SMM bit becomes don't care.

- **SMM** — Special Mask Mode. If ESMM = 1 and SMM = 1, the interrupt controller bank will enter Special Mask Mode. If ESMM = 1 and SMM = 0, the bank will revert to normal mask mode. When ESMM = 0, SMM has no effect.

**Notes:**
- **RR** — Read IR Reg.
- **RIS** — Read IS Reg.
POLL/INTERRUPT REQUEST/IN-SERVICE STATUS REGISTER

Poll Command Status

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>W2</td>
<td>W1</td>
<td>W0</td>
</tr>
</tbody>
</table>

Binary code of the highest priority level requesting

0 - No pending interrupt
1 - Pending interrupt

NOTE:
Although all Interrupt Request inputs are active LOW, the internal logical will invert the state of the pins so that when there is a pending interrupt request at the input, the corresponding IRQ bit will be set to HIGH in the Interrupt Request Status register.

Interrupt Request Status

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ7</td>
<td>IRQ6</td>
<td>IRQ5</td>
<td>IRQ4</td>
<td>IRQ3</td>
<td>IRQ2</td>
<td>IRQ1</td>
<td>IRQ0</td>
</tr>
</tbody>
</table>

If IRQ bit is:
0 - No request
1 - Request pending

In-Service Status

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS7</td>
<td>IS6</td>
<td>IS5</td>
<td>IS4</td>
<td>IS3</td>
<td>IS2</td>
<td>IS1</td>
<td>IS0</td>
</tr>
</tbody>
</table>

If IS bit is:
0 - Not in-service
1 - Request is in-service

VECTOR REGISTER (VR)

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
</table>

8-bit vector number

290164–63
290164–64
290164–65
290164–66

4-193
Table 4-4. Register Operational Summary

<table>
<thead>
<tr>
<th>Operational Description</th>
<th>Command Words</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully Nested Mode</td>
<td>OCW-Default</td>
<td></td>
</tr>
<tr>
<td>Non-specific EOI Command</td>
<td>OCW2</td>
<td>EOI</td>
</tr>
<tr>
<td>Specific EOI Command</td>
<td>OCW2</td>
<td>SL, EOI, L0–L2</td>
</tr>
<tr>
<td>Automatic EOI Mode</td>
<td>ICW1, ICW4</td>
<td>IC4, AEOI</td>
</tr>
<tr>
<td>Rotate On Non-Specific</td>
<td>OCW2</td>
<td>EOI</td>
</tr>
<tr>
<td>EOI Command</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotate On Automatic</td>
<td>OCW2</td>
<td>R, SL, EOI</td>
</tr>
<tr>
<td>EOI Mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set Priority Command</td>
<td>OCW2</td>
<td>L0–L2</td>
</tr>
<tr>
<td>Rotate On Specific</td>
<td>OCW2</td>
<td>R, SL, EOI</td>
</tr>
<tr>
<td>EOI Command</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt Mask Register</td>
<td>OCW1</td>
<td>M0–M7</td>
</tr>
<tr>
<td>Special Mask Mode</td>
<td>OCW3</td>
<td>ESMM, SMM</td>
</tr>
<tr>
<td>Level Triggered Mode</td>
<td>ICW1</td>
<td>LTIM</td>
</tr>
<tr>
<td>Edge Triggered Mode</td>
<td>ICW1</td>
<td>LTIM</td>
</tr>
<tr>
<td>Read Register Command, IRR</td>
<td>OCW3</td>
<td>RR, RIS</td>
</tr>
<tr>
<td>Read Register Command, ISR</td>
<td>OCW3</td>
<td>RR, RIS</td>
</tr>
<tr>
<td>Read IMR</td>
<td>IMR</td>
<td>M0–M7</td>
</tr>
<tr>
<td>Poll Command</td>
<td>OCW3</td>
<td>P</td>
</tr>
<tr>
<td>Special Fully Nested Mode</td>
<td>ICW1, ICW4</td>
<td>IC4, SFNM</td>
</tr>
</tbody>
</table>

4.8 Register Operational Summary

For ease of reference, Table 4-4 gives a summary of the different operating modes and commands with their corresponding registers.

5.0 PROGRAMMABLE INTERVAL TIMER

5.1 Functional Description

The 82370 contains four independently Programmable Interval Timers: Timer 0–3. All four timers are functionally compatible to the Intel 82C54. The first three timers (Timer 0–2) have specific functions. The fourth timer, Timer 3, is a general purpose timer. Table 5-1 depicts the functions of each timer. A brief description of each timer's function follows.

Table 5-1. Programmable Interval Timer Functions

<table>
<thead>
<tr>
<th>Timer</th>
<th>Output</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IRQ8</td>
<td>Event Based IRQ8 Generator</td>
</tr>
<tr>
<td>1</td>
<td>TOUT1/REF#</td>
<td>Gen. Purpose/DRAM Refresh Req.</td>
</tr>
<tr>
<td>2</td>
<td>TOUT2/IRQ3#</td>
<td>Gen. Purpose/Speaker Out/IRQ3#</td>
</tr>
<tr>
<td>3</td>
<td>TOUT3#</td>
<td>Gen. Purpose/IRQ0 Generator</td>
</tr>
</tbody>
</table>

TIMER 0—Event Based Interrupt Request 8 Generator

Timer 0 is intended to be used as an Event Counter. The output of this timer will generate an Interrupt Request 8 (IRQ8) upon a rising edge of the timer output (TOUT0). Normally, this timer is used to implement a time-of-day clock or system tick. The Timer 0 output is not available as an external signal.

TIMER 1—General Purpose/DRAM Refresh Request

The output of Timer 1, TOUT1, can be used as a general purpose timer or as a DRAM Refresh Request signal. The rising edge of this output creates a DRAM refresh request to the 82370 DRAM Refresh Controller. Upon reset, the Refresh Request function is disabled, and the output pin is the Timer 1 output.

TIMER 2—General Purpose/Speaker Out/IRQ3#

The Timer 2 output, TOUT2#, could be used to support tone generation to an external speaker. This pin is a bidirectional signal. When used as an input, a logic LOW asserted at this pin will generate an Interrupt Request 3 (IRQ3#) (see Programmable Interrupt Controller).
Figure 5-1. Block Diagram of Programmable Interval Timer

TIMER 3—General Purpose/Interrupt Request 0 Generator

The output of Timer 3 is fed to an edge detector and generates an Interrupt Request 0 (IRQ0) in the 82370. The inverted output of this timer (TOUT3#) is also available as an external signal for general purpose use.

5.1.1 INTERNAL ARCHITECTURE

The functional block diagram of the Programmable Interval Timer section is shown in Figure 5-1. Following is a description of each block.

DATA BUFFER & READ/WRITE LOGIC

This part of the Programmable Interval Timer is used to interface the four timers to the 82370 internal bus. The Data Buffer is for transferring commands and data between the 8-bit internal bus and the timers.

The Read/Write Logic accepts inputs from the internal bus and generates signals to control other functional blocks within the timer section.

CONTROL WORD REGISTERS I & II

The Control Word Registers are write-only registers. They are used to control the operating modes of the timers. Control Word Register I controls Timers 0, 1 and 2, and Control Word Register II controls Timer 3. Detailed description of the Control Word Registers will be included in the Register Set Overview section.

COUNTER 0, COUNTER 1, COUNTER 2, COUNTER 3

Counters 0, 1, 2, and 3 are the major parts of Timers 0, 1, 2, and 3, respectively. These four functional blocks are identical in operation, so only a single counter will be described. The internal block diagram of one counter is shown in Figure 5-2.
The four counters share a common clock input (CLKIN), but otherwise are fully independent. Each counter is programmable to operate in a different mode.

Although the Control Word Register is shown in the figure, it is not part of the counter itself. Its programmed contents are used to control the operations of the counters.

The Status Register, when latched, contains the current contents of the Control Word Register and status of the output and Null Count Flag (see Read Back Command).

The Counting Element (CE) is the actual counter. It is a 16-bit presettable synchronous down counter.

The Output Latches (OL) contain two 8-bit latches (OLM and OLL). Normally, these latches "follow" the content of the CE. OLM contains the most significant byte of the counter and OLL contains the least significant byte. If the Counter Latch Command is sent to the counter, OL will latch the present count until read by the 80376 and then return to follow the CE. One latch at a time is enabled by the timer's Control Logic to drive the internal bus. This is how the 16-bit Counter communicates over the 8-bit internal bus. Note that CE cannot be read. Whenever the count is read, it is one of the OL's that is being read.

When a new count is written into the counter, the value will be stored in the Count Registers (CR), and transferred to CE. The transferring of the contents from CR's to CE is defined as "loading" of the counter. The Count Register contains two 8-bit registers: CRM (which contains the most significant byte) and CRL (which contains the least significant byte). Similar to the OL's, the Control Logic allows one register at a time to be loaded from the 8-bit internal bus. However, both bytes are transferred from the CR's to the CE simultaneously. Both CR's are cleared when the Counter is programmed. This way, if the Counter has been programmed for one byte count (either the most significant or the least significant byte only), the other byte will be zero. Note that CE cannot be written into directly. Whenever a count is written, it is the CR that is being written.

As shown in the diagram, the Control Logic consists of three signals: CLKIN, GATE, and OUT. CLKIN and GATE will be discussed in detail in the section that follows. OUT is the internal output of the counter. The external outputs of some timers (TOUT) are the inverted version of OUT (see TOUT1, TOUT2#, TOUT3#). The state of OUT depends on the mode of operation of the timer.
5.2 Interface Signals

5.2.1 CLKIN

CLKIN is an input signal used by all four timers for internal timing reference. This signal can be independent of the 82370 system clock, CLK2. In the following discussion, each “CLK Pulse” is defined as the time period between a rising edge and a falling edge, in that order, of CLKIN.

During the rising edge of CLKIN, the state of GATE is sampled. All new counts are loaded and counters are decremented on the falling edge of CLKIN.

5.2.2 TOUT1, TOUT2#, TOUT3#

TOUT1, TOUT2# and TOUT3# are the external output signals of Timer 1, Timer 2 and Timer 3, respectively. TOUT2# and TOUT3# are the inverted signals of their respective counter outputs, OUT.

If Timer 2 is to be used as a tone generator of a speaker, external buffering must be used to provide sufficient drive capability.

The Outputs of Timer 2 and 3 are dual function pins. The output pin of Timer 2 (TOUT2#/IRQ3#), which is a bidirectional open-collector signal, can also be used as interrupt request input. When the interrupt function is enabled (through the Programmable Interrupt Controller), a LOW on this input will generate an Interrupt Request 3# to the 82370 Programmable Interrupt Controller. This pin has a weak internal pull-up resistor. To use the IRQ3# function, Timer 2 should be programmed so that OUT2 is LOW. Additionally, OUT3 of Timer 3 is connected to an edge detector which will generate an Interrupt Request 0 (IRQ0) to the 82370 after the rising edge of OUT3 (see Figure 5-1).

5.2.3 GATE

GATE is not an externally controllable signal. Rather, it can be software controlled with the Internal Control Port. The state of GATE is always sampled on the rising edge of CLKIN. Depending on the mode of operation, GATE is used to enable/disable counting or trigger the start of an operation.

For Timer 0 and 1, GATE is always enabled (HIGH). For Timer 2 and 3, GATE is connected to Bit 0 and 6, respectively, of an Internal Control Port (at address 61H) of the 82370. After a hardware reset, the state of GATE of Timer 2 and 3 is disabled (LOW).

5.3 Modes of Operation

Each timer can be independently programmed to operate in one of six different modes. Timers are programmed by writing a Control Word into the Control Word Register followed by an Initial Count (see Programming).

The following are defined for use in describing the different modes of operation.

- CLK Pulse—A rising edge, then a falling edge, in that order, of CLKIN.
- Trigger—A rising edge of a timer’s GATE input.
- Timer/Counter loading—The transfer of a count from Count Register (CR) to Count Element (CE).

5.3.1 MODE 0-INTERRUPT ON TERMINAL COUNT

Mode 0 is typically used for event counting. After the Control Word is written, OUT is initially LOW, and will remain LOW until the counter reaches zero. OUT then goes HIGH and remains HIGH until a new count or a new Mode 0 Control Word is written into the counter.

In this mode, GATE=HIGH enables counting; GATE=LOW disables counting. However, GATE has no effect on OUT.

After the Control Word and initial count are written to a timer, the initial count will be loaded on the next CLK pulse. This CLK pulse does not decrement the count, so for an initial count of N, OUT does not go HIGH until N+1 CLK pulses after the initial count is written.

If a new count is written to the timer, it will be loaded on the next CLK pulse and counting will continue from the new count. If a two-byte count is written, the following happens:

1. Writing the first byte disables counting, OUT is set LOW immediately (i.e. no CLK pulse required).
2. Writing the second byte allows the new count to be loaded on the next CLK pulse.

This allows the counting sequence to be synchronized by software. Again, OUT does not go HIGH until N+1 CLK pulses after the new count of N is written.
NOTES:
The following conventions apply to all mode timing diagrams.
1. Counters are programmed for binary (not BCD) counting and for reading/writing least significant byte (LSB) only.
2. The counter is always selected (CS# always low).
3. CW stands for "Control Word"; CW = 10 means a control word of 10, Hex is written to the counter.
4. LSB stands for "Least significant byte" of count.
5. Numbers below diagrams are count values. The lower number is the least significant byte. The upper number is the most significant byte. Since the counter is programmed to read/write LSB only, the most significant byte cannot be read.
N stands for an undefined count.
Vertical lines show transitions between count values.

Figure 5-3. Mode 0

If an initial count is written while GATE is LOW, the counter will be loaded on the next CLK pulse. When GATE goes HIGH, OUT will go HIGH N CLK pulses later; no CLK pulse is needed to load the counter as this has already been done.

5.3.2 MODE 1—GATE RETRIGGERABLE ONE-SHOT

In this mode, OUT will be initially HIGH. OUT will go LOW on the CLK pulse following a trigger to start the one-shot operation. The OUT signal will then remain LOW until the timer reaches zero. At this point, OUT will stay HIGH until the next trigger comes in. Since the state of GATE signals of Timer 0 and 1 are internally set to HIGH.

After writing the Control Word and initial count, the timer is considered “armed”. A trigger results in loading the timer and setting OUT LOW on the next CLK pulse. Therefore, an initial count of N will result in a one-shot pulse width of N CLK cycles. Note
that this one-shot operation is retriggerable; i.e. OUT will remain LOW for N CLK pulses after every trigger. The one-shot operation can be repeated without rewriting the same count into the timer.

If a new count is written to the timer during a one-shot operation, the current one-shot pulse width will not be affected until the timer is retriggered. This is because loading of the new count to CE will occur only when the one-shot is triggered.

**5.3.3 MODE 2—RATE GENERATOR**

This mode is a divide-by-N counter. It is typically used to generate a Real Time Clock interrupt. OUT will initially be HIGH. When the initial count has decremented to 1, OUT goes LOW for one CLK pulse, then OUT goes HIGH again. Then the timer reloads the initial count and the process is repeated. In other words, this mode is periodic since the same sequence is repeated indefinitely. For an initial count of N, the sequence repeats every N CLK cycles.

Similar to Mode 0, GATE = HIGH enables counting, where GATE = LOW disables counting. If GATE goes LOW during an output pulse (LOW), OUT is set HIGH immediately. A trigger (rising edge on GATE) will reload the timer with the initial count on the next CLK pulse. Then, OUT will go LOW (for one CLK pulse) N CLK pulses after the new trigger. Thus, GATE can be used to synchronize the timer.
After writing a Control Word and initial count, the timer will be loaded on the next CLK pulse. OUT goes LOW (for one CLK pulse) N CLK pulses after the initial count is written. This is another way the timer may be synchronized by software.

Writing a new count while counting does not affect the current counting sequence because the new count will not be loaded until the end of the current counting cycle. If a trigger is received after writing a new count but before the end of the current period, the timer will be loaded with the new count on the next CLK pulse after the trigger, and counting will continue with the new count.

5.3.4 MODE 3—SQUARE WAVE GENERATOR

Mode 3 is typically used for Baud Rate generation. Functionally, this mode is similar to Mode 2 except
for the duty cycle of OUT. In this mode, OUT will be initially HIGH. When half of the initial count has expired, OUT goes low for the remainder of the count. The counting sequence will be repeated, thus this mode is also periodic. Note that an initial count of N results in a square wave with a period of N CLK pulses.

The GATE input can be used to synchronize the timer. GATE=HIGH enables counting; GATE=LOW disables counting. If GATE goes LOW while OUT is LOW, OUT is set HIGH immediately (i.e. no CLK pulse is required). A trigger reloads the timer with the initial count on the next CLK pulse.

After writing a Control Word and initial count, the timer will be loaded on the next CLK pulse. This allows the timer to be synchronized by software.

Writing a new count while counting does not affect the current counting sequence. If a trigger is received after writing a new count but before the end of the current half-cycle of the square wave, the timer will be loaded with the new count on the next CLK pulse and counting will continue from the new count. Otherwise, the new count will be loaded at the end of the current half-cycle.

There is a slight difference in operation depending on whether the initial count is EVEN or ODD. The following description is to show exactly how this mode is implemented.

**EVEN COUNTS:**

OUT is initially HIGH. The initial count is loaded on one CLK pulse and is decremented by two on succeeding CLK pulses. When the count expires (decremented to 2), OUT changes to LOW and the timer is loaded with the initial count. The above process is repeated indefinitely.

**ODD COUNTS:**

OUT is initially HIGH. The initial count minus one (which is an even number) is loaded on one CLK pulse and is decremented by two on succeeding CLK pulses. One CLK pulse after the count expires (decremented to 2), OUT changes to LOW and the timer is loaded with the initial count minus one again. Succeeding CLK pulses decrement the count by two. When the count expires, OUT goes HIGH immediately and the timer is loaded with the initial count minus one. The above process is repeated indefinitely. So for ODD counts, OUT will HIGH or \((N+1)/2\) counts and LOW for \((N-1)/2\) counts.
5.3.5 MODE 4–INITIAL COUNT TRIGGERED STROBE

This mode allows a strobe pulse to be generated by writing an initial count to the timer. Initially, OUT will be HIGH. When a new initial count is written into the timer, the counting sequence will begin. When the initial count expires (decremented to 1), OUT will go LOW for one CLK pulse and then go HIGH again.

After writing the Control Word and initial count, the timer will be loaded on the next CLK pulse. This CLK pulse does not decrement the count, so for an initial count of N, OUT does not strobe LOW until N + 1 CLK pulses after initial count is written.

If a new count is written during counting, it will be loaded in the next CLK pulse and counting will continue from the new count.

Again, GATE = HIGH enables counting while GATE = LOW disables counting. GATE has no effect on OUT.

NOTE:
A GATE transition should not occur one clock prior to terminal count.

Figure 5-6. Mode 3
If a two-byte count is written, the following will occur:

1. Writing the first byte has no effect on counting.

2. Writing the second byte allows the new count to be loaded on the next CLK pulse.

OUT will strobe LOW $N+1$ CLK pulses after the new count of $N$ is written. Therefore, when the strobe pulse will occur after a trigger depends on the value of the initial count loaded.

5.3.6 MODE 5—GATE RETRIGGERABLE STROBE

Mode 5 is very similar to Mode 4 except the count sequence is triggered by the gate signal instead of by writing an initial count. Initially, OUT will be HIGH. Counting is triggered by a rising edge of GATE. When the initial count has expired (decremented to 1), OUT will go LOW for one CLK pulse and then go HIGH again.

After loading the Control Word and initial count, the Count Element will not be loaded until the CLK pulse after a trigger. This CLK pulse does not decrement the count. Therefore, for an initial count of $N$, OUT does not strobe LOW until $N+1$ CLK pulses after a trigger.
The counting sequence is retriggerable. Every trigger will result in the timer being loaded with the initial count on the next CLK pulse.

If the new count is written during counting, the current counting sequence will not be affected. If a trigger occurs after the new count is written but before the current count expires, the timer will be loaded with the new count on the next CLK pulse and a new count sequence will start from there.

5.3.7 OPERATION COMMON TO ALL MODES

5.3.7.1 GATE

The GATE input is always sampled on the rising edge of CLkin. In Modes 0, 2, 3 and 4, the GATE input is level sensitive. The logic level is sampled on the rising edge of CLkin. In Modes 1, 2, 3 and 5, the GATE input is rising edge sensitive. In these modes,
Summary of Gate Operations

<table>
<thead>
<tr>
<th>Mode</th>
<th>GATE LOW or Going LOW</th>
<th>GATE Rising</th>
<th>HIGH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable count</td>
<td>No Effect</td>
<td>Enable count</td>
</tr>
<tr>
<td>1</td>
<td>No Effect</td>
<td></td>
<td>No Effect</td>
</tr>
<tr>
<td>2</td>
<td>1. Disable count</td>
<td>1. Initiate count</td>
<td>Enable count</td>
</tr>
<tr>
<td></td>
<td>2. Sets output HIGH</td>
<td>2. Reset output</td>
<td>Enable count</td>
</tr>
<tr>
<td></td>
<td>immediately</td>
<td>after next clock</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1. Disable count</td>
<td>1. Initiate count</td>
<td>Enable count</td>
</tr>
<tr>
<td></td>
<td>2. Sets output HIGH</td>
<td>2. Reset output</td>
<td>Enable count</td>
</tr>
<tr>
<td></td>
<td>immediately</td>
<td>after next clock</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Disable count</td>
<td>No Effect</td>
<td>Enable count</td>
</tr>
<tr>
<td>5</td>
<td>No Effect</td>
<td></td>
<td>No Effect</td>
</tr>
</tbody>
</table>

a rising edge of GATE (trigger) sets an edge sensitive flip-flop in the timer. The flip-flop is reset immediately after it is sampled. This way, a trigger will be detected no matter when it occurs; i.e. a HIGH logic level does not have to be maintained until the next rising edge of CLKin. Note that in Modes 2 and 3, the GATE input is both edge and level sensitive.

5.3.7.2 Counter

New counts are loaded and counters are decremented on the falling edge of CLKin. The largest possible initial count is 0. This is equivalent to $2^{16}$ for binary counting and $10^4$ for BCD counting.

Note that the counter does not stop when it reaches zero. In Modes 0, 1, 4 and 5, the counter 'wraps around' to the highest count: either FFFF Hex for binary counting or 9999 for BCD counting, and continues counting. Modes 2 and 3 are periodic. The counter reloads itself with the initial count and continues counting from there.

The minimum and maximum initial count in each counter depends on the mode of operation. They are summarized below.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

5.4 Register Set Overview

The Programmable Interval Timer module of the 82370 contains a set of six registers. The port address map of these registers is shown in Table 5-2.

Table 5-2. Timer Register Port Address Map

<table>
<thead>
<tr>
<th>Port Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>40H</td>
<td>Counter 0 Register (read/write)</td>
</tr>
<tr>
<td>41H</td>
<td>Counter 1 Register (read/write)</td>
</tr>
<tr>
<td>42H</td>
<td>Counter 2 Register (read/write)</td>
</tr>
<tr>
<td>43H</td>
<td>Control Word Register I</td>
</tr>
<tr>
<td></td>
<td>(Counter 0, 1 &amp; 2) (write-only)</td>
</tr>
<tr>
<td>44H</td>
<td>Counter 3 Register (read/write)</td>
</tr>
<tr>
<td>45H</td>
<td>Reserved</td>
</tr>
<tr>
<td>46H</td>
<td>Reserved</td>
</tr>
<tr>
<td>47H</td>
<td>Control Word Register II</td>
</tr>
<tr>
<td></td>
<td>(Counter 3) (write-only)</td>
</tr>
</tbody>
</table>

5.4.1 COUNTER 0, 1, 2, 3 REGISTER

These four 8-bit registers are functionally identical. They are used to write the initial count value into the respective timer. Also, they can be used to read the latched count value of a timer. Since they are 8-bit registers, reading and writing of the 16-bit initial count must follow the count format specified in the Control Word Registers; i.e. least significant byte only, most significant byte only, or least significant byte then most significant byte (see Programming).

5.4.2 CONTROL WORD REGISTER I & II

There are two Control Word Registers associated with the Timer section. One of the two registers (Control Word Register I) is used to control the operations of Counters 0, 1 and 2 and the other (Control Word Register II) is for Counter 3. The major functions of both Control Word Registers are listed below:
— Select the timer to be programmed.
— Define which mode the selected timer is to operate in.
— Define the count sequence; i.e. if the selected timer is to count as a Binary Counter or a Binary Coded Decimal (BCD) Counter.
— Select the byte access sequence during timer read/write operations; i.e. least significant byte only, most significant only, or least significant byte first, then most significant byte.

Also, the Control Word Registers can be programmed to perform a Counter Latch Command or a Read Back Command which will be described later.

5.5 Programming

5.5.1 Initialization

Upon power-up or reset, the state of all timers is undefined. The mode, count value, and output of all timers are random. From this point on, how each timer operates is determined solely by how it is programmed. Each timer must be programmed before it can be used. Since the outputs of some timers can generate interrupt signals to the 82370, all timers should be initialized to a known state.

Counters are programmed by writing a Control Word into their respective Control Word Registers. Then, an Initial Count can be written into the corresponding Count Register. In general, the programming procedure is very flexible. Only two conventions need to be remembered:

1. For each timer, the Control Word must be written before the initial count is written.
2. The 16-bit initial count must follow the count format specified in the Control Word (least significant byte only, most significant byte only, or least significant byte first, followed by most significant byte).

Since the two Control Word Registers and the four Counter Registers have separate addresses, and each timer can be individually selected by the appropriate Control Word Register, no special instruction sequence is required. Any programming sequence that follows the conventions above is acceptable.

A new initial count may be written to a timer at any time without affecting the timer's programmed mode in any way. Count sequence will be affected as described in the Modes of Operation section. Note that the new count must follow the programmed count format.

If a timer is previously programmed to read/write two-byte counts, the following precaution applies. A program must not transfer control between writing the first and second byte to another routine which also writes into the same timer. Otherwise, the read/write will result in incorrect count.

Whenever a Control Word is written to a timer, all control logic for that timer(s) is immediately reset (i.e. no CLK pulse is required). Also, the corresponding output in, TOUT#, goes to a known initial state.

5.5.2 Read Operation

Three methods are available to read the current count as well as the status of each timer. They are: Read Counter Registers, Counter Latch Command and Read Back Command. Below is a description of these methods.

Read Counter Registers

The current count of a timer can be read by performing a read operation on the corresponding Counter Register. The only restriction of this read operation is that the CLKIN of the timers must be inhibited by using external logic. Otherwise, the count may be in the process of changing when it is read, giving an undefined result. Note that since all four timers are sharing the same CLKIN signal, inhibiting CLKIN to read a timer will unavoidably disable the other timers also. This may prove to be impractical. Therefore, it is suggested that either the Counter Latch Command or the Read Back Command can be used to read the current count of a timer.

Another alternative is to temporarily disable a timer before reading its Counter Register by using the GATE input. Depending on the mode of operation, GATE = LOW will disable the counting operation. However, this option is available on Timer 2 and 3 only, since the GATE signals of the other two timers are internally enabled all the time.

Counter Latch Command

A Counter Latch Command will be executed whenever a special Control Word is written into a Control Word Register. Two bits written into the Control Word Register distinguish this command from a 'regular' Control Word (see Register Bit Definition). Also, two other bits in the Control Word will select which counter is to be latched.

Upon execution of this command, the selected counter's Output Latch (OL) latches the count at the time the Counter Latch Command is received. This
count is held in the latch until it is read by the 80376, or until the timer is reprogrammed. The count is then unlatched automatically and the OL returns to "following" the Counting Element (CE). This allows reading the contents of the counters "on the fly" without affecting counting in progress. Multiple Counter Latch Commands may be used to latch more than one counter. Each latched count is held until it is read. Counter Latch Commands do not affect the programmed mode of the timer in any way.

If a counter is latched, and at some time later, it is latched again before the prior latched count is read, the second Counter Latch Command is ignored. The count read will then be the count at the time the first command was issued.

In any event, the latched count must be read according to the programmed format. Specifically, if the timer is programmed for two-byte counts, two bytes must be read. However, the two bytes do not have to be read right after the other. Read/write or programming operations of other timers may be performed between them.

Another feature of this Counter Latch Command is that read and write operations of the same timer may be interleaved. For example, if the timer is programmed for two-byte counts, the following sequence is valid:

1. Read least significant byte.
2. Write new least significant byte.
3. Read most significant byte.
4. Write new most significant byte.

If a timer is programmed to read/write two-byte counts, the following precaution applies. A program must not transfer control between reading the first and second byte to another routine which also reads from that same timer. Otherwise, an incorrect count will be read.

READ BACK COMMAND

The Read Back Command is another special Command Word operation which allows the user to read the current count value and/or the status of the selected timer(s). Like the Counter Latch Command, two bits in the Command Word identify this as a Read Back Command (see Register Bit Definition).

The Read Back Command may be used to latch multiple counter Output Latches (OL's) by selecting more than one timer within a Command Word. This single command is functionally equivalent to several Counter Latch Commands, one for each counter to be latched. Each counter's latched count will be held until it is read by the 80376 or until the timer is reprogrammed. The counter is automatically unlatched when read, but other counters remain latched until they are read. If multiple Read Back commands are issued to the same timer without reading the count, all but the first are ignored; i.e. the count read will correspond to the very first Read Back Command issued.

As mentioned previously, the Read Back Command may also be used to latch status information of the selected timer(s). When this function is enabled, the status of a timer can be read from the Counter Register after the Read Back Command is issued. The status information of a timer includes the following:

1. Mode of timer:
   This allows the user to check the mode of operation of the timer last programmed.
2. State of TOUT pin of the timer:
   This allows the user to monitor the counter's output pin via software, possibly eliminating some hardware from a system.
3. Null Count/Count available:
   The Null Count Bit in the status byte indicates if the last count written to the Count Register (CR) has been loaded into the Counting Element (CE). The exact time this happens depends on the mode of the timer and is described in the Programming section. Until the count is loaded into the Counting Element (CE), it cannot be read from the timer. If the count is latched or read before this occurs, the count value will not reflect the new count just written.

If multiple status latch operations of the timer(s) are performed without reading the status, all but the first command are ignored; i.e. the status read in will correspond to the first Read Back Command issued.

Both the current count and status of the selected timer(s) may be latched simultaneously by enabling both functions in a single Read Back Command. This is functionally the same as issuing two separate Read Back Commands at once. Once again, if multiple read commands are issued to latch both the count and status of a timer, all but the first command will be ignored.

If both count and status of a timer are latched, the first read operation of that timer will return the latched status, regardless of which was latched first. The next one or two (if two count bytes are to be read) read operations return the latched count. Note that subsequent read operations on the Counter Register will return the unlatched count (like the first read method discussed).
5.6 Register Bit Definitions

**COUNTER 0, 1, 2, 3 REGISTER (READ/WRITE)**

<table>
<thead>
<tr>
<th>Port Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>40H</td>
<td>Counter 0 Register (read/write)</td>
</tr>
<tr>
<td>41H</td>
<td>Counter 1 Register (read/write)</td>
</tr>
<tr>
<td>42H</td>
<td>Counter 2 Register (read/write)</td>
</tr>
<tr>
<td>44H</td>
<td>Counter 3 Register (read/write)</td>
</tr>
<tr>
<td>45H</td>
<td>Reserved</td>
</tr>
<tr>
<td>46H</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**Control Word Register I & II (WRITE-ONLY)**

<table>
<thead>
<tr>
<th>Port Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>43H</td>
<td>Control Word Register I</td>
</tr>
<tr>
<td></td>
<td>(Counter 0, 1, 2 (write-only))</td>
</tr>
<tr>
<td>47H</td>
<td>Control Word Register II</td>
</tr>
<tr>
<td></td>
<td>(Counter 3) (write-only)</td>
</tr>
</tbody>
</table>

Note that these 8-bit registers are for writing and reading of one byte of the 16-bit count value, either the most significant or the least significant byte.
**COUNTER LATCH COMMAND FORMAT**

(Write to Control Word Register)

```
D7 D6 D5 D4 D3 D2 D1 D0
SC1 SC0 0 0 X X X X

00 COUNTER 0 (OR 3)
01 COUNTER 1
10 COUNTER 2
11 READ BACK COMMAND
```

**READ BACK COMMAND FORMAT**

(Write to Control Word Register)

```
D7 D6 D5 D4 D3 D2 D1 D0
1 1 COUNT# STATUS CNT2 CNT1 CNT0/3 0

0 = LATCH COUNT
1 = DO NOT LATCH COUNT

0 = COUNTER NOT SELECTED
1 = COUNTER IS SELECTED

0 = LATCH STATUS
1 = DO NOT LATCH STATUS
```

**STATUS FORMAT**

(Returned from Read Back Command)

```
D7 D6 D5 D4 D3 D2 D1 D0
OUT NULL COUNT RW1 RW0 M2 M1 MO BCD

0 = OUTPUT PIN = 0
1 = OUTPUT PIN = 1

0 = COUNT AVAILABLE FOR READING
1 = NULL COUNT
```

---

290164-78
290164-79
290164-80
6.0 WAIT STATE GENERATOR

6.1 Functional Description

The 82370 contains a programmable Wait State Generator which can generate a pre-programmed number of wait states during both CPU and DMA initiated bus cycles. This Wait State Generator is capable of generating 1 to 16 wait states in non-pipelined mode, and 0 to 15 wait states in pipelined mode. Depending on the bus cycle type and the two Wait State Control inputs (WSC 0-1), a pre-programmed number of wait states in the selected Wait State Register will be generated.

The Wait State Generator can also be disabled to allow the use of devices capable of generating their own READY# signals. Figure 6-1 is a block diagram of the Wait State Generator.

6.2 Interface Signals

The following describes the interface signals which affect the operation of the Wait State Generator. The READY#, WSC0 and WSC1 signals are inputs. READYO# is the ready output signal to the host processor.

6.2.1 READY#

READY# is an active LOW input signal which indicates to the 82370 the completion of a bus cycle. In the Master mode (e.g. 82370 initiated DMA transfer), this signal is monitored to determine whether a peripheral or memory needs wait states inserted in the current bus cycle. In the Slave mode, it is used (together with the ADS# signal) to trace CPU bus cycles to determine if the current cycle is pipelined.

6.2.2 READYO#

READYO# (Ready Out#) is an active LOW output signal and is the output of the Wait State Generator. The number of wait states generated depends on the WSC(0–1) inputs. Note that special cases are handled for access to the 82370 internal registers and for the Refresh cycles. For 82370 internal register access, READYO# will be delayed to take into the command recovery time of the register. One or more wait states will be generated in a pipelined cycle. During refresh, the number of wait states will be determined by the preprogrammed value in the Refresh Wait State Register.

In the simplest configuration, READYO# can be connected to the READY# input of the 82370 and the 80376 CPU. This is, however, not always the case. If external circuitry is to control the READY# inputs as well, additional logic will be required (see Application Issues).

6.2.3 WSC(0–1)

These two Wait State Control inputs, together with the M/IO# input, select one of the three pre-programmed 8-bit Wait State Registers which determines the number of wait states to be generated. The most significant half of the three Wait State Registers corresponds to memory accesses, the least significant half to I/O accesses. The combination WSC(0–1) = 11 disables the Wait State Generator.

![Wait State Generator Block Diagram](image-url)
6.3 Bus Function

6.3.1 WAIT STATES IN NON-PIPELINED CYCLE

The timing diagram of two typical non-pipelined cycles with 82370 generated wait states is shown in Figure 6-2. In this diagram, it is assumed that the internal registers of the 82370 are not addressed. During the first T2 state of each bus cycle, the Wait State Control and the M/IO# inputs are sampled to determine which Wait State Register (if any) is selected. If the WSC inputs are active (i.e. not both are driven HIGH), the pre-programmed number of wait states corresponding to the selected Wait State Register will be requested. This is done by driving the READYO# output HIGH during the end of each T2 state.

The WSC (0–1) inputs need only be valid during the very first T2 state of each non-pipelined cycle. As a general rule, the WSC inputs are sampled on the rising edge of the next clock (82384 CLK) after the last state when ADS# (Address Status) is asserted.

The number of wait states generated depends on the type of bus cycle, and the number of wait states requested. The various combinations are discussed below.

1. Access the 82370 internal registers: 2 to 5 wait states, depending upon the specific register addressed. Some back-to-back sequences to the Interrupt Controller will require 7 wait states.

2. Interrupt Acknowledge to the 82370: 5 wait states.

3. Refresh: As programmed in the Refresh Wait State Register (see Register Set Overview). Note that if WCS (0–1) = 11, READYO# will stay inactive.

4. Other bus cycles: Depending on WCS (0–1) and M/IO# inputs, these inputs select a Wait State Register in which the number of wait states will be equal to the pre-programmed wait state count in the register plus 1. The Wait State Register selection is defined as follows (Table 6-1).

<table>
<thead>
<tr>
<th>M/IO#</th>
<th>WSC(0–1)</th>
<th>Register Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>WAIT REG 0 (I/O half)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>WAIT REG 1 (I/O half)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>WAIT REG 2 (I/O half)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>WAIT REG 0 (MEM half)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>WAIT REG 1 (MEM half)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>WAIT REG 2 (MEM half)</td>
</tr>
<tr>
<td>X</td>
<td>11</td>
<td>Wait State Gen. Disabled</td>
</tr>
</tbody>
</table>

The Wait State Control signals, WSC (0–1), can be generated with the address decode and the Read/Write control signals as shown in Figure 6-3.

Figure 6-2. Wait States in Non-Pipelined Cycles
Address Decode

W/R# ➔ Logic ➔ WSC(0-1)

Figure 6-3. WSC (0–1) Generation

Note that during HALT and SHUTDOWN, the number of wait states will depend on the WSC (0–1) inputs, which will select the memory half of one of the Wait State Registers (see CPU Reset and Shutdown Detect).

6.3.2 WAIT STATES IN PIPELINED CYCLES

The timing diagram of two typical pipelined cycles with 82370 generated wait states is shown in Figure 6-4. Again, in this diagram, it is assumed that the 82370 internal registers are not addressed. As defined in the timing of the 80376 processor, the Address (A1–23), Byte Enable (BHE#, BLE#), and other control signals (M/IO#, ADS#) are asserted one T-state earlier than in a non-pipelined cycle; i.e. they are asserted at T2P. Similar to the non-pipelined case, the Wait State Control (WSC) inputs are sampled in the middle of the state after the last state the ADS# signal is asserted. Therefore, the WSC inputs should be asserted during the T1P state of each pipelined cycle (which is one T-state earlier than in the non-pipelined cycle).

The number of wait states generated in a pipelined cycle is selected in a similar manner as in the non-pipelined case discussed in the previous section. The only difference here is that the actual number of wait states generated will be one less than that of the non-pipelined cycle. This is done automatically by the Wait State Generator.

6.3.3 EXTENDING AND EARLY TERMINATING BUS CYCLE

The 82370 allows external logic to either add wait states or cause early termination of a bus cycle by controlling the READY# input to the 82370 and the host processor. A possible configuration is shown in Figure 6-5.

Figure 6-4. Wait States in Pipelined Cycles
The EXT. ROY # (External Ready) signal of Figure 6-5 allows external devices to cause early termination of a bus cycle. When this signal is asserted LOW, the output of the circuit will also go LOW (even though the READYO# of the 82370 may still be HIGH). This output is fed to the READY # input of the 80376 and the 82370 to indicate the completion of the current bus cycle.

Similarly, the EXT. NOT READY (External Not Ready) signal is used to delay the READY # input of the processor and the 82370. As long as this signal is driven HIGH, the output of the circuit will drive the READY # input HIGH. This will effectively extend the duration of a bus cycle. However, it is important to note that if the two-level logic is not fast enough to satisfy the READY # setup time, the OR gate should be eliminated. Instead, the 82370 Wait State Generator can be disabled by driving both WSC (0–1) HIGH. In this case, the addressed memory or I/O device should activate the external READY # input whenever it is ready to terminate the current bus cycle.

Figures 6-6 and 6-7 show the timing relationships of the ready signals for the early termination and extension of the bus cycles. Section 6-7, Application Issues, contains a detailed timing analysis of the external circuit.
Due to the following implications, it should be noted that early termination of bus cycles in which 82370 internal registers are accessed is not recommended.

1. Erroneous data may be read from or written into the addressed register.

2. The 82370 must be allowed to recover either before HLDA (Hold Acknowledge) is asserted or before another bus cycle into an 82370 internal register is initiated.

The recovery time, in clock periods, equals the remaining wait states that were avoided plus 4.

6.4 Register Set Overview

Altogether, there are four 8-bit internal registers associated with the Wait State Generator. The port address map of these registers is shown below in Table 6-2. A detailed description of each follows.

<table>
<thead>
<tr>
<th>Port Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>72H</td>
<td>Wait State Register 0 (read/write)</td>
</tr>
<tr>
<td>73H</td>
<td>Wait State Register 1 (read/write)</td>
</tr>
<tr>
<td>74H</td>
<td>Wait State Register 2 (read/write)</td>
</tr>
<tr>
<td>75H</td>
<td>Ref. Wait State Register (read/write)</td>
</tr>
</tbody>
</table>

WAIT STATE REGISTER 0, 1, 2

These three 8-bit read/write registers are functionally identical. They are used to store the pre-programmed wait state count. One half of each register contains the wait state count for I/O accesses while the other half contains the count for memory accesses. The total number of wait states generated will depend on the type of bus cycle. For a non-pipelined cycle, the actual number of wait states requested is equal to the wait state count plus 1. For a pipelined cycle, the number of wait states will be equal to the wait state count in the selected register. Therefore, the Wait State Generator is capable of generating 1 to 16 wait states in non-pipelined mode, and 0 to 15 wait states in pipelined mode.

Note that the minimum wait state count in each register is 0. This is equivalent to 0 wait states for a pipelined cycle and 1 wait state for a non-pipelined cycle.

REFRESH WAIT STATE REGISTER

Similar to the Wait State Registers discussed above, this 4-bit register is used to store the number of wait states to be generated during a DRAM refresh cycle.

Note that the Refresh Wait State Register is not selected by the WSC inputs. It will automatically be chosen whenever a DRAM refresh cycle occurs. If the Wait State Generator is disabled during the refresh cycle (WSC (0-1) = 11), READYO# will stay inactive and the Refresh Wait State Register is ignored.

6.5 Programming

Using the Wait State Generator is relatively straightforward. No special programming sequence is required. In order to ensure the expected number of wait states will be generated when a register is selected, the registers to be used must be programmed after power-up by writing the appropriate wait state count into each register. Note that upon hardware reset, all Wait State Registers are initialized with the value FFH, giving the maximum number of wait states possible. Also, each register can be read to check the wait state count previously stored in the register.

6.6 Register Bit Definition

WAIT STATE REGISTER 0, 1, 2

<table>
<thead>
<tr>
<th>Port Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>72H</td>
<td>Wait State Register 0 (read/write)</td>
</tr>
<tr>
<td>73H</td>
<td>Wait State Register 1 (read/write)</td>
</tr>
<tr>
<td>74H</td>
<td>Wait State Register 2 (read/write)</td>
</tr>
</tbody>
</table>

REFRESH WAIT STATE REGISTER

Port Address: 75H (Read/Write)
6.7 Application Issues

6.7.1 EXTERNAL ‘READY’ CONTROL LOGIC

As mentioned in section 6.3.3, wait state cycles generated by the 82370 can be terminated early or extended longer by means of additional external logic (see Figure 6-5). In order to ensure that the READY# input timing requirement of the 80376 and the 82370 is satisfied, special care must be taken when designing this external control logic. This section addresses the design requirements.

A simplified block diagram of the external logic along with the READY# timing diagram is shown in Figure 6-8. The purpose is to determine the maximum delay time allowed in the external control logic in order to satisfy the READY# setup time.

First, it will be assumed that the 80376 is running at 16 MHz (i.e. CLK2 is 32 MHz). Therefore, one bus state (two CLK2 periods) will be equivalent to 62.5 ns. According to the AC specifications of the 82370, the maximum delay time for valid READYO# signal is 31 ns after the rising edge of CLK2 in the beginning of T2 (for non-pipelined cycle) or T2P (for pipelined cycle). Also, the minimum READY# setup time of the 80376 and the 82370 should be 19 ns before the rising edge of CLK2 at the beginning of the next bus state. This limits the total delay time for the external READY# control logic to be 12.5 ns (62.5−31−19) in order to meet the READY# setup timing requirement.

![Figure 6-8. 'READY' Timing Consideration](image-url)
7.0 DRAM REFRESH CONTROLLER

7.1 Functional Description

The 82370 DRAM Refresh Controller consists of a 24-bit Refresh Address Counter and Refresh Request logic for DRAM refresh operations (see Figure 7-1). TIMER 1 can be used as a trigger signal to the DRAM Refresh Request logic. The Refresh Bus Size can be programmed to be 8- or 16-bit wide. Depending on the Refresh Bus Size, the Refresh Address Counter will be incremented with the appropriate value after every refresh cycle. The internal logic of the 82370 will give the Refresh operation the highest priority in the bus control arbitration process. Bus control is not released and re-requested if the 82370 is already a bus master.

7.2 Interface Signals

7.2.1 TOUT1/REF#

The dual function output pin of TIMER 1 (TOUT1/REF#) can be programmed to generate DRAM Refresh signal. If this feature is enabled, the rising edge of TIMER 1 output (TOUT1 #) will trigger the DRAM Refresh Request logic. After some delay for gaining access of the bus, the 82370 DRAM Controller will generate a DRAM Refresh signal by driving REF# output LOW. This signal is cleared after the refresh cycle has taken place, or by a hardware reset.

If the DRAM Refresh feature is disabled, the TOUT1/REF# output pin is simply the TIMER 1 output. Detailed information of how TIMER 1 operates is discussed in section 6—Programmable Interval Timer, and will not be repeated here.

7.3 Bus Function

7.3.1 ARBITRATION

In order to ensure data integrity of the DRAMs, the 82370 gives the DRAM Refresh signal the highest priority in the arbitration logic. It allows DRAM Refresh to interrupt DMA in progress in order to perform the DRAM Refresh cycle. The DMA service will be resumed after the refresh is done.

In case of a DRAM Refresh during a DMA process, the cascaded device will be requested to get off the bus. This is done by de-asserting the EDACK signal. Once DREQn goes inactive, the 82370 will perform the refresh operation. Note that the DMA controller does not completely relinquish the system bus during refresh. The Refresh Generator simply "steals" a bus cycle between DMA accesses.

Figure 7-2 shows the timing diagram of a Refresh Cycle. Upon expiration of TIMER 1, the 82370 will try to take control of the system bus by asserting HOLD. As soon as the 82370 see HLDA go active, the DRAM Refresh Cycle will be carried out by activating the REF# signal as well as the address and control signals on the system bus (Note that REF# will not be active until two CLK periods HLDA is asserted). The address bus will contain the 24-bit ad-

![Diagram of DRAM Refresh Controller](image-url)
dress currently in the Refresh Address Counter. The control signals are driven the same way as in a Memory Read cycle. This “read” operation is complete when the READY# signal is driven LOW. Then, the 82370 will relinquish the bus by de-asserting HOLD. Typically, a Refresh Cycle without wait states will take five bus states to execute. If “n” wait states are added, the Refresh Cycle will last for five plus “n” bus states.

How often the Refresh Generator will initiate a refresh cycle depends on the frequency of CLkin as well as Timer 1’s programmed mode of operation. For this specific application, Timer 1 should be programmed to operate in Mode 2 to generate a constant clock rate. See section 6—Programmable Interval Timer for more information on programming the timer. One DRAM Refresh Cycle will be generated each time Timer 1 expires (when TOUT1 changes from LOW to HIGH).

The Wait State Generator can be used to insert wait states during a refresh cycle. The 82370 will automatically insert the desired number of wait states as programmed in the Refresh Wait State Register (see Wait State Generator).

### 7.4 Modes of Operation

#### 7.4.1 WORD SIZE AND REFRESH ADDRESS COUNTER

The 82370 supports 8- and 16-bit refresh cycle. The bus width during a refresh cycle is programmable (see Programming). The bus size can be programmed via the Refresh Control Register (see Register Overview). If the DRAM bus size is 8- or 16-bits, the Refresh Address Counter will be incremented by 1 or 2, respectively.

The Refresh Address Counter is cleared by a hardware reset.

#### 7.5 Register Set Overview

The Refresh Generator has two internal registers to control its operation. They are the Refresh Control Register and the Refresh Wait State Register. Their port address map is shown in Table 7-1 below.

![Figure 7-2. 82370 Refresh Cycle](image-url)
Table 7-1. Register Address Map

<table>
<thead>
<tr>
<th>Port Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1CH</td>
<td>Refresh Control Reg. (read/write)</td>
</tr>
<tr>
<td>75H</td>
<td>Ref. Wait State Reg. (read/write)</td>
</tr>
</tbody>
</table>

The Refresh Wait State Register is not part of the Refresh Generator. It is only used to program the number of wait states to be inserted during a refresh cycle. This register is discussed in detail in section 7 (Wait State Generator) and will not be repeated here.

**REFRESH CONTROL REGISTER**

This 2-bit register serves two functions. First, it is used to enable/disable the DRAM Refresh function output. If disabled, the output of TIMER 1 is simply used as a general purpose timer. The second function of this register is to program the DRAM bus size for the refresh operation. The programmed bus size also determines how the Refresh Address Counter will be incremented after each refresh operation.

**7.6 Programming**

Upon hardware reset, the DRAM Refresh function is disabled (the Refresh Control Register is cleared). The following programming steps are needed before the Refresh Generator can be used. Since the rate of refresh cycles depends on how TIMER 1 is programmed, this timer must be initialized with the desired mode of operation as well as the correct refresh interval (see Programming Interval Timer). Whether or not wait states are to be generated during a refresh cycle, the Refresh Wait State Register must also be programmed with the appropriate value. Then, the DRAM Refresh feature must be enabled and the DRAM bus width should be defined. These can be done in one step by writing the appropriate control word into the Refresh Control Register (see Register Bit Definition). After these steps are done, the refresh operation will automatically be invoked by the Refresh Generator upon expiration of Timer 1.

In addition to the above programming steps, it should be noted that after reset, although the TOUT1/REF# becomes the Time 1 output, the state of this pin in undefined. This is because the Timer module has not been initialized yet. Therefore, if this output is used as a DRAM Refresh signal, this pin should be disqualified by external logic until the Refresh function is enabled. One simple solution is to logically AND this output with HLDA, since HLDA should not be active after reset.

**7.7 Register Bit Definition**

**REFRESH CONTROL REGISTER**

Port Address: 1CH (Read/Write)

**8.0 RELOCATION REGISTER AND ADDRESS DECODE**

**8.1 Relocation Register**

All the integrated peripheral devices in the 82370 are controlled by a set of internal registers. These registers span a total of 256 consecutive address locations (although not all the 256 locations are used). The 82370 provides a Relocation Register which allows the user to map this set of internal registers into either the memory or I/O address space. The function of the Relocation Register is to define the base address of the internal register set of the 82370 as well as if the registers are to be memory- or I/O-mapped. The format of the Relocation Register is depicted in Figure 9-1.

```

D7  D6  D5  D4  D3  D2  D1  D0

MUST BE ZERO

00 REF. DISABLED
01 INTEL RESERVED
10 BUS SIZE = 16
11 BUS SIZE = 8

290164-93
```
Note that the Relocation Register is part of the internal register set of the 82370. It has a port address of 7FH. Therefore, any time the content of the Relocation Register is changed, the physical location of this register will also be moved. Upon reset of the 82370, the content of the Relocation Register will be cleared. This implies that the 82370 will respond to its I/O addresses in the range of 0000H to 00FFH.

### 8.1.1 I/O-MAPPED 82370

As shown in the figure, Bit 0 of the Relocation Register determines whether the 82370 registers are to be memory-mapped or I/O mapped. When Bit 0 is set to '0', the 82370 will respond to I/O Addresses. Address signals BHE#, BLE#, A1–A7 will be used to select one of the internal registers to be accessed. Bit 1 to Bit 7 of the Relocation Register will correspond to A9 to A15 of the Address bus, respectively. Together with A8 implied to be '0', A15 to A8 will be fully decoded by the 82370. The following shows how the 82370 is mapped into the I/O address space.

**Example**

Relocation Register = 11001110 (0CEH)

82370 will respond to I/O address range from 0CE00H to 0CEFFH.

Therefore, this I/O mapping mechanism allows the 82370 internal registers to be located on any even, contiguous, 256 byte boundary of the system I/O space.

### 8.1.2 MEMORY-MAPPED 82370

When Bit 0 of the Relocation Register is set to '1', the 82370 will respond to memory addresses. Again, Address signals BHE#, BLE#, A1–A7 will be used to select one of the internal registers to be accessed. Bit 1 to Bit 7 of the Relocation Register will correspond to A17–A23, respectively. A16 is assumed to be '0', and A8–A15 are ignored. Consider the following example.

**Example**

Relocation Register = 10100111 (0A7H)

The 82370 will respond to memory addresses in the range of A6XX00H to A60XXFFH (where 'X' is don't care).

This scheme implies that the internal registers can be located in any even, contiguous, 2**16 byte page of the memory space.

### 8.2 Address Decoding

As mentioned previously, the 82370 internal registers do not occupy the entire contiguous 256 address locations. Some of the locations are 'unoccupied'. The 82370 always decodes the lower 8 address signals (BHE#, BLE#, A1–A7) to determine if any one of its registers is being accessed. If the address does not correspond to any of its registers, the 82370 will not respond. This allows external devices to be located within the 'holes' in the 82370 address space. Note that there are several unused addresses reserved for future Intel peripheral devices.

### 8.3 Chip-Select (CHPSEL #)

The Chip-Select signal (CHPSEL #) will go active when the 82370 is addressed in a Slave bus.
cycle (either read or write), or in an interrupt acknowledge cycle in which the 82370 will drive the Data Bus. For a given bus cycle, CHPSEL# becomes active and valid in the first T2 (in a non-pipelined cycle) or in T1P (in a pipelined cycle). It will stay valid until the cycle is terminated by READY# driven active. As CHPSEL# becomes valid well before the 82370 drives the Data Bus, it can be used to control the transceivers that connect the local CPU bus to the system bus. The timing diagram of CHPSEL# is shown in Figure 8-2.

9.0 CPU RESET AND SHUTDOWN DETECT

The 82370 will activate the CPURST signal to reset the host processor when one of the following conditions occurs:
- 82370 RESET is active;
- 82370 detects a 80376 Shutdown cycle (this feature can be disabled);
- CPURST software command is issued to 80376.

Whenever the CPURST signal is activated, the 82370 will reset its own internal Slave-Bus state machine.

9.1 Hardware Reset

Following a hardware reset, the 82370 will assert its CPURST output to reset the host processor. This output will stay active as long as the RESET input is active. During a hardware reset, the 82370 internal registers will be initialized as defined in the corresponding functional descriptions.

9.2 Software Reset

CPURST can be generated by writing the following bit pattern into 82370 register location 64H.

\[
\begin{array}{cccccc}
D7 & \ldots & D0 \\
1 & 1 & 1 & X & X & X & 0
\end{array}
\]

The Write operation into this port is considered as an 82370 access and the internal Wait State Generator will automatically determine the required number of wait states. The CPURST will be active following the completion of the Write cycle to this port. This signal will last for 62 CLK2 periods. The 82370 should not be accessed until the CPURST is deactivated.

This internal port is Write-Only and the 82370 will not respond to a Read operation to this location. Also, during a software reset command, the 82370 will reset its Slave-Bus state machine. However, its internal registers remain unchanged. This allows the operating system to distinguish a 'warm' reset by reading any 82370 internal register previously programmed for a non-default value. The Diagnostic registers can be used for this purpose (see Internal Control and Diagnostic Ports).

9.3 Shutdown Detect

The 82370 is constantly monitoring the Bus Cycle Definition signals (M/IO#, D/C#, W/R#) and is able to detect when the 80376 is in a Shutdown bus cycle. Upon detection of a processor shutdown, the 82370 will activate the CPURST output for 62 CLK2 periods to reset the host processor. This signal is generated after the Shutdown cycle is terminated by the READY# signal.
Although the 82370 Wait State Generator will not automatically respond to a Shutdown (or Halt) cycle, the Wait State Control inputs (WSC0, WSC1) can be used to determine the number of wait states in the same manner as other non-82370 bus cycles.

This Shutdown Detect feature can be enabled or disabled by writing a control bit in the Internal Control Port at address 61H (see Internal Control and Diagnostic Ports). This feature is disabled upon a hardware reset of the 82370. As in the case of Software Reset, the 82370 will reset its Slave-Bus state machine but will not change any of its internal register contents.

10.0 INTERNAL CONTROL AND DIAGNOSTIC PORTS

10.1 Internal Control Port

The format of the Internal Control Port of the 82370 is shown in Figure 10-1. This Control Port is used to enable/disable the Processor Shutdown Detect mechanism as well as controlling the Gate inputs of the Timer 2 and 3. Note that this is a Write-Only port. Therefore, the 82370 will not respond to a read operation to this port. Upon hardware reset, this port will be cleared; i.e., the Shutdown Detect feature and the Gate inputs of Timer 2 and 3 are disabled.

Port Address: 61H (Write only)

10.2 Diagnostic Ports

Two 8-bit read/write Diagnostic Ports are provided in the 82370. These are two storage registers and have no effect on the operation of the 82370. They can be used to store checkpoint data or error codes in the power-on sequence and in the diagnostic service routines. As mentioned in the CPU RESET AND SHUTDOWN DETECT section, these Diagnostic Ports can be used to distinguish between 'cold' and 'warm' reset. Upon hardware reset, both Diagnostic Ports are cleared. The address map of these Diagnostic Ports is shown in Figure 10-2.

<table>
<thead>
<tr>
<th>Port Address</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagnostic Port 1 (Read/Write)</td>
<td>80H</td>
</tr>
<tr>
<td>Diagnostic Port 2 (Read/Write)</td>
<td>88H</td>
</tr>
</tbody>
</table>

Figure 10-2. Address Map of Diagnostic Ports

11.0 INTEL RESERVED I/O PORTS

There are nineteen I/O ports in the 82370 address space which are reserved for Intel future peripheral device use only. Their address locations are: 10H, 12H, 14H, 16H, 2AH, 3DH, 3EH, 45H, 46H, 76H, 77H, 7DH, 7EH, CCH, CDH, D0H, D2H, D4H, and D6H. These addresses should not be used in the system since the 82370 will respond to read/write operations to these locations and bus contention may occur if any peripheral is assigned to the same address location.

Figure 10-1. Internal Control Port
12.0 PACKAGE THERMAL SPECIFICATIONS

The Intel 82370 Integrated System Peripheral is specified for operation when case temperature is within the range of 0°C to 78°C for the ceramic 132-pin PGA package, and 68°C for the 100-pin plastic package. The case temperature may be measured in any environment, to determine whether the 82370 is within specified operating range. The case temperature should be measured at the center of the top surface opposite the pins.

The ambient temperature is guaranteed as long as \( T_c \) is not violated. The ambient temperature can be calculated from the \( \theta_{jc} \) and \( \theta_{ja} \) from the following equations:

\[
T_J = T_c + P \cdot \theta_{jc} \\
T_A = T_j - P \cdot \theta_{ja} \\
T_C = T_a + P \cdot (\theta_{ja} - \theta_{jc})
\]

Values for \( \theta_{ja} \) and \( \theta_{jc} \) are given in Table 12.1 for the 100-lead fine pitch. \( \theta_{ja} \) is given at various airflows. Table 12.2 shows the maximum \( T_a \) allowable (without exceeding \( T_c \)) at various airflows. Note that \( T_a \) can be improved further by attaching "fins" or a "heat sink" to the package. \( P \) is calculated using the maximum hot \( I_{cc} \).

---

**Table 12.1 82370 Package Thermal Characteristics**

<table>
<thead>
<tr>
<th>Package</th>
<th>( \theta_{jc} )</th>
<th>( \theta_{ja} ) Versus Airflow-ft³/min (m³/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 (0)</td>
<td>200 (1.01) 400 (2.03) 600 (3.04) 800 (4.06) 1000 (5.07)</td>
</tr>
<tr>
<td>100L Fine Pitch</td>
<td>7</td>
<td>33 27 24 21 18 17</td>
</tr>
<tr>
<td>132L PGA</td>
<td>2</td>
<td>21 17 14 12 11 10</td>
</tr>
</tbody>
</table>

**Table 12.2 82370 Maximum Allowable Ambient Temperature at Various Airflows**

<table>
<thead>
<tr>
<th>Package</th>
<th>( \theta_{jc} )</th>
<th>( T_a(c) ) Versus Airflow-ft³/min (m³/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 (0)</td>
<td>200 (1.01) 400 (2.03) 600 (3.04) 800 (4.06) 1000 (5.07)</td>
</tr>
<tr>
<td>100L Fine Pitch</td>
<td>7</td>
<td>63 74 79 85 91 92</td>
</tr>
<tr>
<td>132L PGA</td>
<td>2</td>
<td>74 83 88 93 97 99</td>
</tr>
</tbody>
</table>

100L PQFP Pkg:
\[
\begin{align*}
T_c &= T_a + P \cdot (\theta_{ja} - \theta_{jc}) \\
T_c &= 63 + 220 \text{ mA}(33 - 7) \\
T_c &= 63 + 220 \text{ mA}(26) \\
T_c &= 63 + 5.72 \\
T_c &= 68.7
\end{align*}
\]

132L PGA Pkg:
\[
\begin{align*}
T_c &= T_a + P \cdot (\theta_{ja} - \theta_{jc}) \\
T_c &= 74 + 220 \text{ mA}(21 - 2) \\
T_c &= 74 + 220 \text{ mA}(19) \\
T_c &= 74 + 4.18 \\
T_c &= 78.2
\end{align*}
\]
13.0 ELECTRICAL SPECIFICATIONS

82370 D.C. Specifications Functional Operating Range:

\[ V_{CC} = 5.0V \pm 10\%; T_{CASE} = 0^\circ C \text{ to } 78^\circ C \text{ for } 132\text{-pin PGA, } 0^\circ C \text{ to } 68^\circ C \text{ for } 100\text{-pin plastic} \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter Description</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_{IL} )</td>
<td>Input Low Voltage</td>
<td>-0.3</td>
<td>0.8</td>
<td>V</td>
<td>(Note 1)</td>
</tr>
<tr>
<td>( V_{IH} )</td>
<td>Input High Voltage</td>
<td>2.0</td>
<td>( V_{CC} + 0.3 )</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>( V_{ILC} )</td>
<td>CLK2 Input Low Voltage</td>
<td>-0.3</td>
<td>0.8</td>
<td>V</td>
<td>(Note 1)</td>
</tr>
<tr>
<td>( V_{IHc} )</td>
<td>CLK2 Input High Voltage</td>
<td>( V_{CC} - 0.8 )</td>
<td>( V_{CC} + 0.3 )</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>( V_{OL} )</td>
<td>Output Low Voltage</td>
<td>( I_{OL} = 4 \text{ mA} ): ( A_{1-23}, D_{0-15}, \text{ BHE#}, \text{ BLE#} )</td>
<td>0.45</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( I_{OL} = 5 \text{ mA} ): All Others</td>
<td>0.45</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>( V_{OH} )</td>
<td>Output High Voltage</td>
<td>( I_{OH} = -1 \text{ mA} ): ( A_{23-A_{1}}, D_{15}-D_{0}, \text{ BHE#}, \text{ BLE#} )</td>
<td>2.4</td>
<td>V</td>
<td>(Note 5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( I_{OH} = -0.2 \text{ mA} ): ( A_{23-A_{1}}, D_{15}-D_{0}, \text{ BHE#}, \text{ BLE#} )</td>
<td>( V_{CC} - 0.5 )</td>
<td>V</td>
<td>(Note 5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( I_{OH} = -0.9 \text{ mA} ): All Others</td>
<td>2.4</td>
<td>V</td>
<td>(Note 5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( I_{OH} = -0.18 \text{ mA} ): All Others</td>
<td>( V_{CC} - 0.5 )</td>
<td>V</td>
<td>(Note 5)</td>
</tr>
<tr>
<td>( I_{LI} )</td>
<td>Input Leakage Current</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All Inputs Except:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRQ11# - IRQ23#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOP#, TOUT2/IRQ3#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DREQ4/IRQ9#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( I_{LL1} )</td>
<td>Input Leakage Current</td>
<td>Inputs:</td>
<td>0 &lt; ( V_{IN} &lt; V_{CC} )</td>
<td>(Note 3)</td>
</tr>
<tr>
<td>IRQ11# - IRQ23#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOP#, TOUT2/IRQ3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DREQ4/IRQ9#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( I_{LO} )</td>
<td>Output Leakage Current</td>
<td>0 &lt; ( V_{IN} &lt; V_{CC} )</td>
<td>(Note 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( \pm 15 \mu A )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( I_{CC} )</td>
<td>Supply Current (CLK2 = 32 MHz)</td>
<td>220</td>
<td>mA</td>
<td>(Note 4)</td>
<td></td>
</tr>
<tr>
<td>( C_{1} )</td>
<td>Input Capacitance</td>
<td>12</td>
<td>pF</td>
<td>(Note 2)</td>
<td></td>
</tr>
<tr>
<td>( C_{CLK} )</td>
<td>CLK2 Input Capacitance</td>
<td>20</td>
<td>pF</td>
<td>(Note 2)</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. Minimum value is not 100% tested.
2. \( f_{C} = 1 \text{ MHz} \); sampled only.
3. These pins have weak internal pullups. They should not be left floating.
4. \( I_{CC} \) is specified with inputs driven to CMOS levels, and outputs driving CMOS loads. \( I_{CC} \) may be higher if inputs are driven to TTL levels, or if outputs are driving TTL loads.
5. Tested at the minimum operating frequency of the part.
**82370 A.C. Specifications** These A.C. timings are tested at 1.5V thresholds, except as noted. Functional Operating Range: \( V_{CC} = 5.0V \pm 10\% \); \( T_{CASE} = 0^\circ C \) to 76\(^\circ C\) for 132-pin PGA, 0\(^\circ C\) to 68\(^\circ C\) for 100-pin plastic

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter Description</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Operating Frequency (1/(t1a \times 2))</td>
<td>4</td>
<td>16</td>
<td>MHz</td>
<td></td>
</tr>
<tr>
<td>t2a</td>
<td>CLK2 High Time</td>
<td>31</td>
<td>125</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t2b</td>
<td>CLK2 High Time</td>
<td>5</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t3a</td>
<td>CLK2 Low Time</td>
<td>5</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t3b</td>
<td>CLK2 Low Time</td>
<td>7</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>CLK2 Fall Time</td>
<td>7</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td>CLK2 Rise Time</td>
<td>7</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t6</td>
<td>A1-A23, BHE#, BLE#</td>
<td>4</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDACK0-EDACK2 Valid Delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t7</td>
<td>A1-A23, BHE#, BLE#</td>
<td>4</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDACK0-EDACK3 Float Delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t8</td>
<td>A1-A23, BHE#, BLE#</td>
<td>6</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t9</td>
<td>A1-A23, BHE#, BLE#</td>
<td>4</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t10</td>
<td>W/R#, M/IO#, D/C#</td>
<td>4</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t11</td>
<td>W/R#, M/IO#, D/C#</td>
<td>4</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 13-1. Drive Levels and Measurement Points for A.C. Specification**
## 82370 A.C. Specifications

These A.C. timings are tested at 1.5V thresholds, except as noted.

**Functional Operating Range:** $V_{CC} = 5.0V \pm 10\%$; $T_{CASE} = 0^\circ C$ to $78^\circ C$ for 132-pin PGA, $0^\circ C$ to $68^\circ C$ for 100-pin plastic (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter Description</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>t12</td>
<td>W/R#, M/IO#, D/C# Setup Time</td>
<td>6</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t13</td>
<td>W/R#, M/IO#, D/C# Hold Time</td>
<td>4</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t14</td>
<td>ADS# Valid Delay</td>
<td>6</td>
<td></td>
<td>ns</td>
<td>CL = 50 pF (Note 1)</td>
</tr>
<tr>
<td>t15</td>
<td>ADS# Floating Delay</td>
<td>4</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t16</td>
<td>ADS# Setup Time</td>
<td>21</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t17</td>
<td>ADS# Hold Time</td>
<td>4</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t18</td>
<td>Slave Mode D0–D15 Read Valid</td>
<td>3</td>
<td></td>
<td>ns</td>
<td>$C_L = 120$ pF</td>
</tr>
<tr>
<td>t19</td>
<td>Slave Mode D0–D15 Read Floating</td>
<td>6</td>
<td></td>
<td>ns</td>
<td>(Note 1)</td>
</tr>
<tr>
<td>t20</td>
<td>Slave Mode D0–D15 Write Setup</td>
<td>31</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t21</td>
<td>Slave Mode D0–D15 Write Hold</td>
<td>26</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t22</td>
<td>Master Mode D0–D15 Write Valid</td>
<td>4</td>
<td></td>
<td>ns</td>
<td>$C_L = 120$ pF</td>
</tr>
<tr>
<td>t23</td>
<td>Master Mode D0–D15 Write Floating</td>
<td>4</td>
<td></td>
<td>ns</td>
<td>(Note 1)</td>
</tr>
<tr>
<td>t24</td>
<td>Master Mode D0–D15 Read Setup</td>
<td>8</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t25</td>
<td>Master Mode D0–D15 Read Hold</td>
<td>6</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t26</td>
<td>READY# Setup Time</td>
<td>19</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t27</td>
<td>READY# Hold Time</td>
<td>4</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t28</td>
<td>WSC0–WSC1 Setup Time</td>
<td>6</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t29</td>
<td>WSC0–WSC1 Hold Time</td>
<td>21</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t30</td>
<td>RESET Setup Time</td>
<td>13</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t31</td>
<td>RESET Hold Time</td>
<td>4</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t32</td>
<td>READY#O Valid Delay</td>
<td>4</td>
<td></td>
<td>ns</td>
<td>$C_L = 25$ pF</td>
</tr>
<tr>
<td>t33</td>
<td>CPU#RST Valid Delay (Falling Edge Only)</td>
<td>2</td>
<td></td>
<td>ns</td>
<td>$C_L = 50$ pF</td>
</tr>
<tr>
<td>t34</td>
<td>HOLD Valid Delay</td>
<td>5</td>
<td></td>
<td>ns</td>
<td>$C_L = 100$ pF</td>
</tr>
<tr>
<td>t35</td>
<td>HLDA Setup Time</td>
<td>21</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t36</td>
<td>HLDA Hold Time</td>
<td>6</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t37a</td>
<td>EOP# Setup (Synchronous)</td>
<td>21</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t38a</td>
<td>EOP# Hold (Synchronous)</td>
<td>6</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t37b</td>
<td>EOP# Setup (Asynchronous)</td>
<td>11</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t38b</td>
<td>EOP# Hold (Asynchronous)</td>
<td>11</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t39</td>
<td>EOP# Valid Delay (Falling Edge Only)</td>
<td>5</td>
<td></td>
<td>ns</td>
<td>$C_L = 100$ pF</td>
</tr>
<tr>
<td>t40</td>
<td>EOP# Floating Delay</td>
<td>5</td>
<td></td>
<td>ns</td>
<td>(Note 1)</td>
</tr>
<tr>
<td>t41a</td>
<td>DREQ Setup (Synchronous)</td>
<td>21</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t42a</td>
<td>DREQ Hold (Synchronous)</td>
<td>4</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t41b</td>
<td>DREQ Setup (Asynchronous)</td>
<td>11</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t42b</td>
<td>DREQ Hold (Asynchronous)</td>
<td>11</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t43</td>
<td>INT Valid Delay from IRQn</td>
<td>500</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t44</td>
<td>NA# Setup Time</td>
<td>5</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>t45</td>
<td>NA# Hold Time</td>
<td>15</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
</tbody>
</table>
**82370 A.C. Specifications**

These A.C. timings are tested at 1.5V thresholds, except as noted.

Functional Operating Range: \( V_{CC} = 5.0V \pm 10\% \); \( T_{CASE} = 0°C \) to 78°C for 132-pin PGA, 0°C to 68°C for 100-pin plastic (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter Description</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>t46</td>
<td>CLKIN Frequency</td>
<td>DC</td>
<td>10</td>
<td>MHz</td>
<td></td>
</tr>
<tr>
<td>t47</td>
<td>CLKIN High Time</td>
<td>30</td>
<td>ns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t48</td>
<td>CLKIN Low Time</td>
<td>50</td>
<td>ns</td>
<td>0.8V</td>
<td></td>
</tr>
<tr>
<td>t49</td>
<td>CLKIN Rise Time</td>
<td>10</td>
<td>ns</td>
<td>0.8V to 3.7V</td>
<td></td>
</tr>
<tr>
<td>t50</td>
<td>CLKIN Fall Time</td>
<td>10</td>
<td>ns</td>
<td>3.7V to 0.8V</td>
<td></td>
</tr>
<tr>
<td>t51</td>
<td>TOUT1 / REF Valid Delay from CLK2 (Refresh)</td>
<td>4</td>
<td>36</td>
<td>ns</td>
<td>( C_L = 120 \text{ pF} )</td>
</tr>
<tr>
<td>t52</td>
<td>from CLKIN (Timer)</td>
<td>3</td>
<td>93</td>
<td>ns</td>
<td>( C_L = 120 \text{ pF} )</td>
</tr>
<tr>
<td>t53</td>
<td>TOUT2 Valid Delay (from CLKIN, Falling Edge Only)</td>
<td>3</td>
<td>93</td>
<td>ns</td>
<td>( C_L = 120 \text{ pF} )</td>
</tr>
<tr>
<td>t54</td>
<td>TOUT2 Float Delay</td>
<td>3</td>
<td>36</td>
<td>ns</td>
<td>(Note 1)</td>
</tr>
<tr>
<td>t55</td>
<td>TOUT3 Valid Delay</td>
<td>3</td>
<td>93</td>
<td>ns</td>
<td>( C_L = 120 \text{ pF} )</td>
</tr>
<tr>
<td>t56</td>
<td>CHPSEL Valid Delay</td>
<td>1</td>
<td>35</td>
<td>ns</td>
<td>( C_L = 25 \text{ pF} )</td>
</tr>
</tbody>
</table>

**NOTE:**

1. Float condition occurs when the maximum output current becomes less than \( I_{LO} \) in magnitude. Float delay is not tested.

For testing purposes, the float condition occurs when the dynamic output driven voltage changes with current loads.

---

**Figure 13-2. A.C. Test Load**

**Figure 13-3**
Figure 13-4. Input Setup and Hold Timing
Figure 13-5. Reset Timing

Figure 13-6. Address Output Delays
Figure 13-7. Data Bus Output Delays

Figure 13-8. Control Output Delays
**14.0 REVISION HISTORY**

This 82370 data sheet, version -002, contains updates and improvements to previous versions. A revision summary is listed here for your convenience.

**The sections significantly revised since version -001 are:**
- Section 12.0 Electrical Characteristics renumbered Section 13.0.
- Section 12.0 Package Thermal Specifications added.
- Section 13.0 Electrical Specifications updated $T_{case}$, $V_{OH}$, $I_{CC}$, $T_{33}$, $T_{39}$, Figure 13.6.
- Appendix C, Programming the 82370 Interrupt Controllers, added.
- Appendix D, System Notes, added.
- Section 14.0 Revision History added.
## APPENDIX A
PORTS LISTED BY ADDRESS

<table>
<thead>
<tr>
<th>Port Address (HEX)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Read/Write DMA Channel 0 Target Address, A0–A15</td>
</tr>
<tr>
<td>01</td>
<td>Read/Write DMA Channel 0 Byte Count, B0–B15</td>
</tr>
<tr>
<td>02</td>
<td>Read/Write DMA Channel 1 Target Address, A0–A15</td>
</tr>
<tr>
<td>03</td>
<td>Read/Write DMA Channel 1 Byte Count, B0–B15</td>
</tr>
<tr>
<td>04</td>
<td>Read/Write DMA Channel 2 Target Address, A0–A15</td>
</tr>
<tr>
<td>05</td>
<td>Read/Write DMA Channel 2 Byte Count, B0–B15</td>
</tr>
<tr>
<td>06</td>
<td>Read/Write DMA Channel 3 Target Address, A0–A15</td>
</tr>
<tr>
<td>07</td>
<td>Read/Write DMA Channel 3 Byte Count, B0–B15</td>
</tr>
<tr>
<td>08</td>
<td>Read/Write DMA Channel 0–3 Status/Command I Register</td>
</tr>
<tr>
<td>09</td>
<td>Read/Write DMA Channel 0–3 Software Request Register</td>
</tr>
<tr>
<td>0A</td>
<td>Write DMA Channel 0–3 Set-Reset Mask Register</td>
</tr>
<tr>
<td>0B</td>
<td>Write DMA Channel 0–3 Mode Register I</td>
</tr>
<tr>
<td>0C</td>
<td>Write Clear Byte-Pointer FF</td>
</tr>
<tr>
<td>0D</td>
<td>Write DMA Master-Clear</td>
</tr>
<tr>
<td>0E</td>
<td>Write DMA Channel 0–3 Clear Mask Register</td>
</tr>
<tr>
<td>0F</td>
<td>Read/Write DMA Channel 0–3 Mask Register</td>
</tr>
<tr>
<td>10</td>
<td>Intel Reserved</td>
</tr>
<tr>
<td>11</td>
<td>Read/Write DMA Channel 0 Byte Count, B16–B23</td>
</tr>
<tr>
<td>12</td>
<td>Intel Reserved</td>
</tr>
<tr>
<td>13</td>
<td>Read/Write DMA Channel 1 Byte Count, B16–B23</td>
</tr>
<tr>
<td>14</td>
<td>Intel Reserved</td>
</tr>
<tr>
<td>15</td>
<td>Read/Write DMA Channel 2 Byte Count, B16–B23</td>
</tr>
<tr>
<td>16</td>
<td>Intel Reserved</td>
</tr>
<tr>
<td>17</td>
<td>Read/Write DMA Channel 3 Byte Count, B16–B23</td>
</tr>
<tr>
<td>18</td>
<td>Write DMA Channel 0–3 Bus Size Register</td>
</tr>
<tr>
<td>19</td>
<td>Read/Write DMA Channel 0–3 Chaining Register</td>
</tr>
<tr>
<td>1A</td>
<td>Write DMA Channel 0–3 Command Register II</td>
</tr>
<tr>
<td>1B</td>
<td>Write DMA Channel 0–3 Mode Register II</td>
</tr>
<tr>
<td>1C</td>
<td>Read/Write Refresh Control Register</td>
</tr>
<tr>
<td>1E</td>
<td>Reset Software Request Interrupt</td>
</tr>
<tr>
<td>20</td>
<td>Write Bank B ICW1, OCW2 or OCW3</td>
</tr>
<tr>
<td></td>
<td>Read Bank B Poll, Interrupt Request or In-Service Status Register</td>
</tr>
<tr>
<td>21</td>
<td>Write Bank B ICW2, ICW3, ICW4 or OCW1</td>
</tr>
<tr>
<td>22</td>
<td>Read Bank B Interrupt Mask Register</td>
</tr>
<tr>
<td>28</td>
<td>Read/Write IRQ8 Vector Register</td>
</tr>
<tr>
<td>29</td>
<td>Read/Write IRQ9 Vector Register</td>
</tr>
<tr>
<td>2A</td>
<td>Reserved</td>
</tr>
<tr>
<td>Port Address (HEX)</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>2B</td>
<td>Read/Write IRQ11 Vector Register</td>
</tr>
<tr>
<td>2C</td>
<td>Read/Write IRQ12 Vector Register</td>
</tr>
<tr>
<td>2D</td>
<td>Read/Write IRQ13 Vector Register</td>
</tr>
<tr>
<td>2E</td>
<td>Read/Write IRQ14 Vector Register</td>
</tr>
<tr>
<td>2F</td>
<td>Read/Write IRQ15 Vector Register</td>
</tr>
<tr>
<td>30</td>
<td>Write Bank A ICW1, OCW2 or OCW3 Read Bank A Poll, Interrupt Request or In-Service Status Register</td>
</tr>
<tr>
<td>31</td>
<td>Write Bank A ICW2, ICW3, IOW4 or OCW1 Read Bank A Interrupt Mask Register</td>
</tr>
<tr>
<td>32</td>
<td>Read Bank A ICW2</td>
</tr>
<tr>
<td>38</td>
<td>Read/Write IRQ0 Vector Register</td>
</tr>
<tr>
<td>39</td>
<td>Read/Write IRQ1 Vector Register</td>
</tr>
<tr>
<td>3A</td>
<td>Read/Write IRQ1.5 Vector Register</td>
</tr>
<tr>
<td>3B</td>
<td>Read/Write IRQ3 Vector Register</td>
</tr>
<tr>
<td>3C</td>
<td>Read/Write IRQ4 Vector Register</td>
</tr>
<tr>
<td>3D</td>
<td>Reserved</td>
</tr>
<tr>
<td>3E</td>
<td>Reserved</td>
</tr>
<tr>
<td>3F</td>
<td>Read/Write IRQ7 Vector Register</td>
</tr>
<tr>
<td>40</td>
<td>Read/Write Counter 0 Register</td>
</tr>
<tr>
<td>41</td>
<td>Read/Write Counter 1 Register</td>
</tr>
<tr>
<td>42</td>
<td>Read/Write Counter 2 Register</td>
</tr>
<tr>
<td>43</td>
<td>Write Control Word Register I—Counter 0, 1, 2</td>
</tr>
<tr>
<td>44</td>
<td>Read/Write Counter 3 Register</td>
</tr>
<tr>
<td>45</td>
<td>Reserved</td>
</tr>
<tr>
<td>46</td>
<td>Reserved</td>
</tr>
<tr>
<td>47</td>
<td>Write Word Register II—Counter 3</td>
</tr>
<tr>
<td>61</td>
<td>Write Internal Control Port</td>
</tr>
<tr>
<td>64</td>
<td>Write CPU Reset Register (Data—1111XXX0H)</td>
</tr>
<tr>
<td>72</td>
<td>Read/Write Wait State Register 0</td>
</tr>
<tr>
<td>73</td>
<td>Read/Write Wait State Register 1</td>
</tr>
<tr>
<td>74</td>
<td>Read/Write Wait State Register 2</td>
</tr>
<tr>
<td>75</td>
<td>Read/Write Refresh Wait State Register</td>
</tr>
<tr>
<td>76</td>
<td>Reserved</td>
</tr>
<tr>
<td>77</td>
<td>Reserved</td>
</tr>
<tr>
<td>7D</td>
<td>Reserved</td>
</tr>
<tr>
<td>7E</td>
<td>Reserved</td>
</tr>
<tr>
<td>7F</td>
<td>Read/Write Relocation Register</td>
</tr>
<tr>
<td>80</td>
<td>Read/Write Internal Diagnostic Port 0</td>
</tr>
<tr>
<td>81</td>
<td>Read/Write DMA Channel 2 Target Address, A16–A23</td>
</tr>
<tr>
<td>82</td>
<td>Read/Write DMA Channel 3 Target Address, A16–A23</td>
</tr>
<tr>
<td>83</td>
<td>Read/Write DMA Channel 1 Target Address, A16–A23</td>
</tr>
<tr>
<td>87</td>
<td>Read/Write DMA Channel 0 Target Address, A16–A23</td>
</tr>
<tr>
<td>88</td>
<td>Read/Write Internal Diagnostic Port 1</td>
</tr>
<tr>
<td>89</td>
<td>Read/Write DMA Channel 6 Target Address, A16–A23</td>
</tr>
<tr>
<td>8A</td>
<td>Read/Write DMA Channel 7 Target Address, A16–A23</td>
</tr>
<tr>
<td>8B</td>
<td>Read/Write DMA Channel 5 Target Address, A16–A23</td>
</tr>
<tr>
<td>8F</td>
<td>Read/Write DMA Channel 4 Target Address, A16–A23</td>
</tr>
<tr>
<td>Port Address (HEX)</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>90</td>
<td>Read/Write DMA Channel 0 Requester Address, A0–A15</td>
</tr>
<tr>
<td>91</td>
<td>Read/Write DMA Channel 0 Requester Address, A16–A23</td>
</tr>
<tr>
<td>92</td>
<td>Read/Write DMA Channel 1 Requester Address, A0–A15</td>
</tr>
<tr>
<td>93</td>
<td>Read/Write DMA Channel 1 Requester Address, A16–A23</td>
</tr>
<tr>
<td>94</td>
<td>Read/Write DMA Channel 2 Requester Address, A0–A15</td>
</tr>
<tr>
<td>95</td>
<td>Read/Write DMA Channel 2 Requester Address, A16–A23</td>
</tr>
<tr>
<td>96</td>
<td>Read/Write DMA Channel 3 Requester Address, A0–A15</td>
</tr>
<tr>
<td>97</td>
<td>Read/Write DMA Channel 3 Requester Address, A16–A23</td>
</tr>
<tr>
<td>98</td>
<td>Read/Write DMA Channel 4 Requester Address, A0–A15</td>
</tr>
<tr>
<td>99</td>
<td>Read/Write DMA Channel 4 Requester Address, A16–A23</td>
</tr>
<tr>
<td>9A</td>
<td>Read/Write DMA Channel 5 Requester Address, A0–A15</td>
</tr>
<tr>
<td>9B</td>
<td>Read/Write DMA Channel 5 Requester Address, A16–A23</td>
</tr>
<tr>
<td>9C</td>
<td>Read/Write DMA Channel 6 Requester Address, A0–A15</td>
</tr>
<tr>
<td>9D</td>
<td>Read/Write DMA Channel 6 Requester Address, A16–A23</td>
</tr>
<tr>
<td>9E</td>
<td>Read/Write DMA Channel 7 Requester Address, A0–A15</td>
</tr>
<tr>
<td>9F</td>
<td>Read/Write DMA Channel 7 Requester Address, A16–A23</td>
</tr>
<tr>
<td>A0</td>
<td>Write Bank C ICW1, OCW2 or OCW3</td>
</tr>
<tr>
<td></td>
<td>Read Bank C Poll, Interrupt Request or In-Service Status Register</td>
</tr>
<tr>
<td>A1</td>
<td>Write Bank C ICW2, ICW3, ICW4 or OCW1</td>
</tr>
<tr>
<td></td>
<td>Read Bank C Interrupt Mask Register</td>
</tr>
<tr>
<td>A2</td>
<td>Read Bank C ICW2</td>
</tr>
<tr>
<td>A8</td>
<td>Read/Write IRQ16 Vector Register</td>
</tr>
<tr>
<td>A9</td>
<td>Read/Write IRQ17 Vector Register</td>
</tr>
<tr>
<td>AA</td>
<td>Read/Write IRQ18 Vector Register</td>
</tr>
<tr>
<td>AB</td>
<td>Read/Write IRQ19 Vector Register</td>
</tr>
<tr>
<td>AC</td>
<td>Read/Write IRQ20 Vector Register</td>
</tr>
<tr>
<td>AD</td>
<td>Read/Write IRQ21 Vector Register</td>
</tr>
<tr>
<td>AE</td>
<td>Read/Write IRQ22 Vector Register</td>
</tr>
<tr>
<td>AF</td>
<td>Read/Write IRQ23 Vector Register</td>
</tr>
<tr>
<td>C0</td>
<td>Read/Write DMA Channel 4 Target Address, A0–A15</td>
</tr>
<tr>
<td>C1</td>
<td>Read/Write DMA Channel 4 Byte Count, B0–B15</td>
</tr>
<tr>
<td>C2</td>
<td>Read/Write DMA Channel 5 Target Address, A0–A15</td>
</tr>
<tr>
<td>C3</td>
<td>Read/Write DMA Channel 5 Byte Count, B0–B15</td>
</tr>
<tr>
<td>C4</td>
<td>Read/Write DMA Channel 6 Target Address, A0–A15</td>
</tr>
<tr>
<td>C5</td>
<td>Read/Write DMA Channel 6 Byte Count, B0–B15</td>
</tr>
<tr>
<td>C6</td>
<td>Read/Write DMA Channel 7 Target Address, A0–A15</td>
</tr>
<tr>
<td>C7</td>
<td>Read/Write DMA Channel 7 Byte Count, B0–B15</td>
</tr>
<tr>
<td>C8</td>
<td>Read DMA Channel 4–7 Status/Command I Register</td>
</tr>
<tr>
<td>C9</td>
<td>Read DMA Channel 4–7 Software Request Register</td>
</tr>
<tr>
<td>CA</td>
<td>Write DMA Channel 4–7 Set-Reset Mask Register</td>
</tr>
<tr>
<td>CB</td>
<td>Write DMA Channel 4–7 Mode Register I</td>
</tr>
<tr>
<td>CC</td>
<td>Reserved</td>
</tr>
<tr>
<td>CD</td>
<td>Reserved</td>
</tr>
<tr>
<td>CE</td>
<td>Write DMA Channel 4–7 Clear Mask Register</td>
</tr>
<tr>
<td>CF</td>
<td>Read/Write DMA Channel 4–7 Mask Register</td>
</tr>
<tr>
<td>D0</td>
<td>Intel Reserved</td>
</tr>
<tr>
<td>D1</td>
<td>Read/Write DMA Channel 4 Byte Count, B16–B23</td>
</tr>
<tr>
<td>D2</td>
<td>Intel Reserved</td>
</tr>
<tr>
<td>D3</td>
<td>Read/Write DMA Channel 5 Byte Count, B16–B23</td>
</tr>
<tr>
<td>Port Address (HEX)</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>D4</td>
<td>Intel Reserved</td>
</tr>
<tr>
<td>D5</td>
<td>Read/Write DMA Channel 6 Byte Count, B16–B23</td>
</tr>
<tr>
<td>D6</td>
<td>Intel Reserved</td>
</tr>
<tr>
<td>D7</td>
<td>Read/Write DMA Channel 7 Byte Count, B16–B23</td>
</tr>
<tr>
<td>D8</td>
<td>Write DMA Channel 4–7 Bus Size Register</td>
</tr>
<tr>
<td>D9</td>
<td>Read/Write DMA Channel 4–7 Chaining Register</td>
</tr>
<tr>
<td>DA</td>
<td>Write DMA Channel 4–7 Command Register II</td>
</tr>
<tr>
<td>DB</td>
<td>Write DMA Channel 4–7 Mode Register II</td>
</tr>
</tbody>
</table>
## APPENDIX B
### PORTS LISTED BY FUNCTION

<table>
<thead>
<tr>
<th>Port Address (HEX)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DMA CONTROLLER</strong></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>Write DMA Master-Clear</td>
</tr>
<tr>
<td>0C</td>
<td>Write DMA Clear Byte-Pointer FF</td>
</tr>
<tr>
<td>08</td>
<td>Read/Write DMA Channel 0–3 Status/Command I Register</td>
</tr>
<tr>
<td>C8</td>
<td>Read/Write DMA Channel 4–7 Status/Command I Register</td>
</tr>
<tr>
<td>1A</td>
<td>Write DMA Channel 0–3 Command Register II</td>
</tr>
<tr>
<td>DA</td>
<td>Write DMA Channel 4–7 Command Register II</td>
</tr>
<tr>
<td>0B</td>
<td>Write DMA Channel 0–3 Mode Register I</td>
</tr>
<tr>
<td>CB</td>
<td>Write DMA Channel 4–7 Mode Register I</td>
</tr>
<tr>
<td>1B</td>
<td>Write DMA Channel 0–3 Mode Register II</td>
</tr>
<tr>
<td>DB</td>
<td>Write DMA Channel 4–7 Mode Register II</td>
</tr>
<tr>
<td>09</td>
<td>Read/Write DMA Channel 0–3 Software Request Register</td>
</tr>
<tr>
<td>C9</td>
<td>Read/Write DMA Channel 4–7 Software Request Register</td>
</tr>
<tr>
<td>1E</td>
<td>Reset Software Request Interrupt</td>
</tr>
<tr>
<td>0E</td>
<td>Write DMA Channel 0–3 Clear Mask Register</td>
</tr>
<tr>
<td>CE</td>
<td>Write DMA Channel 4–7 Clear Mask Register</td>
</tr>
<tr>
<td>0F</td>
<td>Read/Write DMA Channel 0–3 Mask Register</td>
</tr>
<tr>
<td>CF</td>
<td>Read/Write DMA Channel 4–7 Mask Register</td>
</tr>
<tr>
<td>0A</td>
<td>Write DMA Channel 0–3 Set-Reset Mask Register</td>
</tr>
<tr>
<td>CA</td>
<td>Write DMA Channel 4–7 Set-Reset Mask Register</td>
</tr>
<tr>
<td>18</td>
<td>Write DMA Channel 0–3 Bus Size Register</td>
</tr>
<tr>
<td>D8</td>
<td>Write DMA Channel 4–7 Bus Size Register</td>
</tr>
<tr>
<td>19</td>
<td>Read/Write DMA Channel 0–3 Chaining Register</td>
</tr>
<tr>
<td>D9</td>
<td>Read/Write DMA Channel 4–7 Chaining Register</td>
</tr>
<tr>
<td>00</td>
<td>Read/Write DMA Channel 0 Target Address, A0–A15</td>
</tr>
<tr>
<td>87</td>
<td>Read/Write DMA Channel 0 Target Address, A16–A23</td>
</tr>
<tr>
<td>01</td>
<td>Read/Write DMA Channel 0 Byte Count, B0–B15</td>
</tr>
<tr>
<td>11</td>
<td>Read/Write DMA Channel 0 Byte Count, B16–B23</td>
</tr>
<tr>
<td>90</td>
<td>Read/Write DMA Channel 0 Requester Address, A0–A15</td>
</tr>
<tr>
<td>91</td>
<td>Read/Write DMA Channel 0 Requester Address, A16–A23</td>
</tr>
<tr>
<td>Port Address (HEX)</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>02</td>
<td>Read/Write DMA Channel 1 Target Address, A0–A15</td>
</tr>
<tr>
<td>83</td>
<td>Read/Write DMA Channel 1 Target Address, A16–A23</td>
</tr>
<tr>
<td>03</td>
<td>Read/Write DMA Channel 1 Byte Count, B0–B15</td>
</tr>
<tr>
<td>13</td>
<td>Read/Write DMA Channel 1 Byte Count, B16–B23</td>
</tr>
<tr>
<td>92</td>
<td>Read/Write DMA Channel 1 Requester Address, A0–A15</td>
</tr>
<tr>
<td>93</td>
<td>Read/Write DMA Channel 1 Requester Address, A16–A23</td>
</tr>
<tr>
<td>04</td>
<td>Read/Write DMA Channel 2 Target Address, A0–A15</td>
</tr>
<tr>
<td>81</td>
<td>Read/Write DMA Channel 2 Target Address, A16–A23</td>
</tr>
<tr>
<td>05</td>
<td>Read/Write DMA Channel 2 Byte Count, B0–B15</td>
</tr>
<tr>
<td>15</td>
<td>Read/Write DMA Channel 2 Byte Count, B16–B23</td>
</tr>
<tr>
<td>94</td>
<td>Read/Write DMA Channel 2 Requester Address, A0–A15</td>
</tr>
<tr>
<td>95</td>
<td>Read/Write DMA Channel 2 Requester Address, A16–A23</td>
</tr>
<tr>
<td>06</td>
<td>Read/Write DMA Channel 3 Target Address, A0–A15</td>
</tr>
<tr>
<td>82</td>
<td>Read/Write DMA Channel 3 Target Address, A16–A23</td>
</tr>
<tr>
<td>07</td>
<td>Read/Write DMA Channel 3 Byte Count, B0–B15</td>
</tr>
<tr>
<td>17</td>
<td>Read/Write DMA Channel 3 Byte Count, B16–B23</td>
</tr>
<tr>
<td>96</td>
<td>Read/Write DMA Channel 3 Requester Address, A0–A15</td>
</tr>
<tr>
<td>97</td>
<td>Read/Write DMA Channel 3 Requester Address, A16–A23</td>
</tr>
<tr>
<td>C0</td>
<td>Read/Write DMA Channel 4 Target Address, A0–A15</td>
</tr>
<tr>
<td>8F</td>
<td>Read/Write DMA Channel 4 Target Address, A16–A23</td>
</tr>
<tr>
<td>C1</td>
<td>Read/Write DMA Channel 4 Byte Count, B0–B15</td>
</tr>
<tr>
<td>D1</td>
<td>Read/Write DMA Channel 4 Byte Count, B16–B23</td>
</tr>
<tr>
<td>98</td>
<td>Read/Write DMA Channel 4 Requester Address, A0–A15</td>
</tr>
<tr>
<td>99</td>
<td>Read/Write DMA Channel 4 Requester Address, A16–A23</td>
</tr>
<tr>
<td>C2</td>
<td>Read/Write DMA Channel 5 Target Address, A0–A15</td>
</tr>
<tr>
<td>8B</td>
<td>Read/Write DMA Channel 5 Target Address, A16–A23</td>
</tr>
<tr>
<td>C3</td>
<td>Read/Write DMA Channel 5 Byte Count, B0–B15</td>
</tr>
<tr>
<td>D3</td>
<td>Read/Write DMA Channel 5 Byte Count, B16–B23</td>
</tr>
<tr>
<td>9A</td>
<td>Read/Write DMA Channel 5 Requester Address, A0–A15</td>
</tr>
<tr>
<td>9B</td>
<td>Read/Write DMA Channel 5 Requester Address, A16–A23</td>
</tr>
<tr>
<td>C4</td>
<td>Read/Write DMA Channel 6 Target Address, A0–A15</td>
</tr>
<tr>
<td>89</td>
<td>Read/Write DMA Channel 6 Target Address, A16–A23</td>
</tr>
<tr>
<td>C5</td>
<td>Read/Write DMA Channel 6 Byte Count, B0–B15</td>
</tr>
<tr>
<td>D5</td>
<td>Read/Write DMA Channel 6 Byte Count, B16–B23</td>
</tr>
<tr>
<td>9C</td>
<td>Read/Write DMA Channel 6 Requester Address, A0–A15</td>
</tr>
<tr>
<td>9D</td>
<td>Read/Write DMA Channel 6 Requester Address, A16–A23</td>
</tr>
<tr>
<td>C6</td>
<td>Read/Write DMA Channel 7 Target Address, A0–A15</td>
</tr>
<tr>
<td>8A</td>
<td>Read/Write DMA Channel 7 Target Address, A16–A23</td>
</tr>
<tr>
<td>C7</td>
<td>Read/Write DMA Channel 7 Byte Count, B0–B15</td>
</tr>
<tr>
<td>D7</td>
<td>Read/Write DMA Channel 7 Byte Count, B16–B23</td>
</tr>
<tr>
<td>9E</td>
<td>Read/Write DMA Channel 7 Requester Address, A0–A15</td>
</tr>
<tr>
<td>9F</td>
<td>Read/Write DMA Channel 7 Requester Address, A16–A23</td>
</tr>
<tr>
<td>Port Address (HEX)</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>INTERRUPT CONTROLLER</strong></td>
<td></td>
</tr>
</tbody>
</table>
| 20 | Write Bank B ICW1, OCW2 or OCW3  
Read Bank B Poll, Interrupt Request or In-Service  
Status Register |
| 21 | Write Bank B ICW2, ICW3, ICW4 or OCW1  
Read Bank B Interrupt Mask Register |
| 22 | Read Bank B ICW2 |
| 28 | Read/Write IRQ8 Vector Register |
| 29 | Read/Write IRQ9 Vector Register |
| 2A | Reserved |
| 2B | Read/Write IRQ11 Vector Register |
| 2C | Read/Write IRQ12 Vector Register |
| 2D | Read/Write IRQ13 Vector Register |
| 2E | Read/Write IRQ14 Vector Register |
| 2F | Read/Write IRQ15 Vector Register |
| A0 | Write Bank C ICW1, OCW2 or OCW3  
Read Bank C Poll, Interrupt Request or In-Service  
Status Register |
| A1 | Write Bank C ICW2, ICW3, ICW4 or OCW1  
Read Bank C Interrupt Mask Register |
| A2 | Read Bank C ICW2 |
| A8 | Read/Write IRQ16 Vector Register |
| A9 | Read/Write IRQ17 Vector Register |
| AA | Read/Write IRQ18 Vector Register |
| AB | Read/Write IRQ19 Vector Register |
| AC | Read/Write IRQ20 Vector Register |
| AD | Read/Write IRQ21 Vector Register |
| AE | Read/Write IRQ22 Vector Register |
| AF | Read/Write IRQ23 Vector Register |
| 30 | Write Bank A ICW1, OCW2 or OCW3  
Read Bank A Poll, Interrupt Request or In-Service  
Status Register |
| 31 | Write Bank A ICW2, ICW3, ICW4 or OCW1  
Read Bank A Interrupt Mask Register |
<p>| 32 | Read Bank A ICW2 |
| 38 | Read/Write IRQ0 Vector Register |
| 39 | Read/Write IRQ1 Vector Register |
| 3A | Read/Write IRQ1.5 Vector Register |
| 3B | Read/Write IRQ3 Vector Register |
| 3C | Read/Write IRQ4 Vector Register |
| 3D | Reserved |
| 3E | Reserved |
| 3F | Read/Write IRQ7 Vector Register |</p>
<table>
<thead>
<tr>
<th>Port Address (HEX)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROGRAMMABLE INTERVAL TIMER</strong></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>Read/Write Counter 0 Register</td>
</tr>
<tr>
<td>41</td>
<td>Read/Write Counter 1 Register</td>
</tr>
<tr>
<td>42</td>
<td>Read/Write Counter 2 Register</td>
</tr>
<tr>
<td>43</td>
<td>Write Control Word Register I—Counter 0, 1, 2</td>
</tr>
<tr>
<td>44</td>
<td>Read/Write Counter 3 Register</td>
</tr>
<tr>
<td>47</td>
<td>Write Word Register II—Counter 3</td>
</tr>
<tr>
<td><strong>CPU RESET</strong></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>Write CPU Reset Register (Data—1111XXX0H)</td>
</tr>
<tr>
<td><strong>WAIT STATE GENERATOR</strong></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>Read/Write Wait State Register 0</td>
</tr>
<tr>
<td>73</td>
<td>Read/Write Wait State Register 1</td>
</tr>
<tr>
<td>74</td>
<td>Read/Write Wait State Register 2</td>
</tr>
<tr>
<td>75</td>
<td>Read/Write Refresh Wait State Register</td>
</tr>
<tr>
<td><strong>DRAM REFRESH CONTROLLER</strong></td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td>Read/Write Refresh Control Register</td>
</tr>
<tr>
<td><strong>INTERNAL CONTROL AND DIAGNOSTIC PORTS</strong></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Write Internal Control Port</td>
</tr>
<tr>
<td>80</td>
<td>Read/Write Internal Diagnostic Port 0</td>
</tr>
<tr>
<td>88</td>
<td>Read/Write Internal Diagnostic Port 1</td>
</tr>
<tr>
<td><strong>RELOCATION REGISTER</strong></td>
<td></td>
</tr>
<tr>
<td>7F</td>
<td>Read/Write Relocation Register</td>
</tr>
<tr>
<td><strong>INTEL RESERVED PORTS</strong></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
</tr>
<tr>
<td>12</td>
<td>Reserved</td>
</tr>
<tr>
<td>14</td>
<td>Reserved</td>
</tr>
<tr>
<td>16</td>
<td>Reserved</td>
</tr>
<tr>
<td>2A</td>
<td>Reserved</td>
</tr>
<tr>
<td>3D</td>
<td>Reserved</td>
</tr>
<tr>
<td>3E</td>
<td>Reserved</td>
</tr>
<tr>
<td>45</td>
<td>Reserved</td>
</tr>
<tr>
<td>46</td>
<td>Reserved</td>
</tr>
<tr>
<td>76</td>
<td>Reserved</td>
</tr>
<tr>
<td>77</td>
<td>Reserved</td>
</tr>
<tr>
<td>7D</td>
<td>Reserved</td>
</tr>
<tr>
<td>7E</td>
<td>Reserved</td>
</tr>
<tr>
<td>CC</td>
<td>Reserved</td>
</tr>
<tr>
<td>CD</td>
<td>Reserved</td>
</tr>
<tr>
<td>D0</td>
<td>Reserved</td>
</tr>
<tr>
<td>D2</td>
<td>Reserved</td>
</tr>
<tr>
<td>D4</td>
<td>Reserved</td>
</tr>
<tr>
<td>D6</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
APPENDIX C
PROGRAMMING THE 82370 INTERRUPT CONTROLLERS

This Appendix describes two methods of programming and initializing the Interrupt Controllers of the 82370. A simple interrupt service routine is also shown which provides compatibility with the 82C59 Interrupt Controller.

The two methods of programming the 82370 Interrupt Controllers are needed to provide simple initialization procedures in different software environments. For new applications, a simple initialization and programming sequence can be used. For PC-DOS or other applications which expect 8259s, an interrupt handler for initialization traps must be provided. Once the handler is in place, all three 82370 Interrupt Controller banks can be programmed or initialized in the same manner as an 8259.

The ICW2 interrupt is generated by the 82370 when writing the ICW2 command to any of the interrupt controller banks. This interrupt is supplied to provide compatibility to existing code that expects to be programming 82C59s. The ICW2 value is stored in the ICW2 register of the associated bank, but is ignored by the controller. It is the responsibility of the ICW2 interrupt handler to read the ICW2 register and use its value to program the individual vector registers accordingly.

NEW APPLICATIONS

New applications do not generally require compatibility with previous code, or at least the code is usually easily modifiable. If the application fits this description, then the ICW2 interrupt can be ignored. This is done by initializing the interrupt controller as necessary, and before enabling CPU interrupts, removing the ICW2 interrupt request by reading the ICW2 register. Listing 1 shows the code for doing this for bank A. The same procedure can be used for the other banks.
Listing 1.
Initialization of an 82370 Interrupt Controller Bank
Without ICW2 Interrupts

INIT_BANK_A proc near
    cli ;disable all interrupts

;initialize controller logic
    mov al,ICW1 ;begin sequence
    out 30h,al
    mov al,ICW2 ;send dummy ICW2
    out 31h,al
    mov al,ICW3 ;send ICW3 if necessary
    out 31h,al
    mov al,ICW4 ;send ICW4
    out 31h,al
    mov al,BANK_A_MASK ;write to mask register (OCW1)
    out 31h,al

;program vector registers
    mov al,ICW2 ;IRQ0
    out 38h,al
    mov al,ICW2+1 ;IRQ1
    out 39h,al
    mov al,ICW2_VECTOR ;IRQ1.5 (probably never used in
    out 3Ah,al ;this system)
    mov al,ICW2+3 ;IRQ3
    out 3Bh,al
    mov al,ICW2+4 ;IRQ4
    out 3Ch,al
    mov al,ICW2+7 ;IRQ7
    out 3Fh,al

;remove ICW2 interrupt request
    in al,31h ;read mask register to work around
               ;A-step errata
    in al,32h ;read ICW2 register to clear
               ;interrupt request

;return to calling program
    sti ;re-enable interrupts
    ret

INIT_BANK_A endp
OLD APPLICATIONS

In applications where 8259 compatibility is required, the ICW2 interrupt handler must be invoked whenever an interrupt controller is initialized (ICW1–ICW2–ICWn sequence). The handler's purpose is to read the ICW2 value from the ICW2 read register and write the appropriate sequence of vectors to the vector registers. Listing 2 shows the typical initialization sequence (this is not changed from the 8259), and the required initialization for operation of the ICW2 interrupt handler. Listing 2 shows the ICW2 interrupt handler.

```
cli ;disable all interrupts

;initialize controller logic

mov al,ICW1 ;begin sequence
out 30h,al
mov al,ICW2 ;send dummy ICW2
out 31h,al

;******
mov al,ICW3 ;send ICW3 if necessary
out 31h,al  ; note that using ICW3 for
            ; cascading bank B is not required
            ; and will affect the way E0Is are
            ; required for nesting. It is
            ; advised that ICW3 not be used.

;******
mov al,ICW4 ;send ICW4
out ;31h,al

mov al,Bank_A.Mask ;write to mask register (OCW1=7Bh)
out 31h,al  ;don't mask off IRQ1.5 or Default
            ; interrupt (IRQ7)

;program necessary vector registers

mov al,ICW2_VECTOR ;IRQ1.5
out 3Ah,al

mov al,IRQ7_DEFAULT VECTOR
out 3Fh,al

;remove ICW2 interrupt request for bank A

in al,31h ;read mask register to work around
           ; A-step errata

in al,32h ;read ICW2 register to clear
           ; interrupt request

;at this point install interrupt call vector for ICW2, if
 ;not already done somewhere else in the code

sti ;re-enable interrupts
```
Listing 3.
ICW2 Interrupt Service Routine

ICW2_INT_HANDLER    proc near
                    push ax
                    push cx
                    push dx

; service bank B
                    in al,21h ;read mask register for A-step errata
                    in al,22h ;read ICW2
                    mov cx,8 ;count vectors
                    mov dx,28h ;point to vectors

BANK_B_LOOP:
                    out dx,al ;write vector
                    inc al ;next vector
                    inc dx ;next vector I/O address
                    loop BANK_B_LOOP

; service bank C
                    in al,0A1h ;read mask register for A-step errata
                    in al,0A2h ;read ICW2
                    mov cx,8 ;count vectors
                    mov dx,0A8h ;point to vectors

BANK_C_LOOP:
                    out dx,al ;write vector
                    inc al ;next vector
                    inc dx ;next vector I/O address
                    loop BANK_C_LOOP

                    pop dx ;restore registers
                    pop cx
                    pop ax
                    iret ;return

ICW2_INT_HANDLER    endp
Table 1. Interrupt Controller Registers

<table>
<thead>
<tr>
<th>Bank A</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30H write</td>
<td>ICW1, OCW2, OCW3</td>
<td>Poll, IRR, ISR</td>
</tr>
<tr>
<td>30H read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31H write</td>
<td>ICW2, ICW3, ICW4, OCW1</td>
<td>IMR</td>
</tr>
<tr>
<td>31H read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32H read</td>
<td>ICW2 read register</td>
<td></td>
</tr>
<tr>
<td>32H write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38H read/write</td>
<td>IRQ0 vector</td>
<td></td>
</tr>
<tr>
<td>39H read/write</td>
<td>IRQ1 vector</td>
<td></td>
</tr>
<tr>
<td>3AH read/write</td>
<td>IRQ1.5 vector</td>
<td></td>
</tr>
<tr>
<td>3BH read/write</td>
<td>IRQ3 vector</td>
<td></td>
</tr>
<tr>
<td>3CH read/write</td>
<td>IRQ4 vector</td>
<td></td>
</tr>
<tr>
<td>3DH</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>3EH</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>3FH read/write</td>
<td>IRQ7 vector</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bank B</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20H write</td>
<td>ICW1, OCW2, OCW3</td>
<td>Poll, IRR, ISR</td>
</tr>
<tr>
<td>20H read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21H write</td>
<td>ICW2, ICW3, ICW4, OCW1</td>
<td>IMR</td>
</tr>
<tr>
<td>22H read</td>
<td>ICW2 read register</td>
<td></td>
</tr>
<tr>
<td>28H read/write</td>
<td>IRQ8 vector</td>
<td></td>
</tr>
<tr>
<td>29H read/write</td>
<td>IRQ9 vector</td>
<td></td>
</tr>
<tr>
<td>2AH</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>2BH read/write</td>
<td>IRQ11 vector</td>
<td></td>
</tr>
<tr>
<td>2CH read/write</td>
<td>IRQ12 vector</td>
<td></td>
</tr>
<tr>
<td>2DH read/write</td>
<td>IRQ13 vector</td>
<td></td>
</tr>
<tr>
<td>2EH read/write</td>
<td>IRQ14 vector</td>
<td></td>
</tr>
<tr>
<td>2FH read/write</td>
<td>IRQ15 vector</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bank C</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A0H write</td>
<td>ICW1, OCW2, OCW3</td>
<td>Poll, IRR, ISR</td>
</tr>
<tr>
<td>A0H read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1H write</td>
<td>ICW2, ICW3, ICW4, OCW1</td>
<td>IMR</td>
</tr>
<tr>
<td>A1H read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2H read</td>
<td>ICW2 read register</td>
<td></td>
</tr>
<tr>
<td>A8H read/write</td>
<td>IRQ16 vector</td>
<td></td>
</tr>
<tr>
<td>A9H read/write</td>
<td>IRQ17 vector</td>
<td></td>
</tr>
<tr>
<td>AAH read/write</td>
<td>IRQ18 vector</td>
<td></td>
</tr>
<tr>
<td>ABH read/write</td>
<td>IRQ19 vector</td>
<td></td>
</tr>
<tr>
<td>ACH read/write</td>
<td>IRQ20 vector</td>
<td></td>
</tr>
<tr>
<td>ADH read/write</td>
<td>IRQ21 vector</td>
<td></td>
</tr>
<tr>
<td>AEH read/write</td>
<td>IRQ22 vector</td>
<td></td>
</tr>
<tr>
<td>AFH read/write</td>
<td>IRQ23 vector</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX D
SYSTEM NOTES

1. BHE # IN MASTER MODE.

In Master Mode, BHE # will be activated during DMA to/from 8-bit devices residing at even locations when the remaining byte count is greater than 1.

For example, if an 8-bit device is located at 00000000 Hex and the number of bytes to be transferred is > 1, the first address/BHE # combination will be 00000000/0. In some systems this will cause the bus controller to perform two 8-bit accesses, the first to 00000000 Hex and the second to 00000001 Hex. However, the 82370's DMA will only read/write one byte. This may or may not cause a problem in the system depending on what is located at 00000001 Hex.

Solution:
There are two solutions if BHE active is unacceptable. Of the two, number 2 is the cleanest and most recommended.

1. If there is an 8-bit device that uses DMA located at an even address, do not use that address + 1. The limitation of this solution is that the user must have complete control over what addresses will be used in the end system.

2. Do not allow the Bus Controller to split cycles for the DMA.

2. RESET OUTPUT OF 82370:

The 80376 requires its RESET line to be active for 80 clock cycles. The 82370 generates holds the RESET line active for 62 clock cycles.

The following design example shows how the user can extend the active high of the RESET line to 80 clock cycles.

Extending the RESET Output of the 82370

This section describes a hardware solution for using the 82370’s CPURST output and the software reset command to cause the 80376 to enter into a self-test.

The 80376 requires two simultaneous events in order to initiate the self-test sequence. The RESET input of the processor must be held active for at least 80 CLK2 periods and the BUSY # input must be low 8 CLK2 periods prior to and 8 CLK2 periods subsequent to RESET going inactive.

A system which does not have an 80387SX will simply have the BUSY # input to the 80376 tied low. A system which contains the 80387SX will require extra logic between the BUSY # output of the 80387SX and the BUSY # input of the 80376 in order to force self-test on reset. The extra BUSY # logic required will not be described here.

The 82370 CPURST output is intended to be retimed with faster TTL components in order to meet the RESET input setup time requirements of the 80376 and 80387SX. This requires a 74F379 (quad flip-flop with enable) or equivalent. The flip-flops required are described in TECHBIT (Ed Grochowski, April 10, 1987).

The 82370 does not meet the RESET pulse duration requirements for causing self-test of the 80376 when a software reset command is issued to the 82370. The 82370 provides a RESET pulse width of 62 CLK2 periods, the 80376 requires 80 CLK2 periods as mentioned earlier.

In order to cause the 80376 to do a self-test after a software reset, the CPURST output pulse of the 82370 must be lengthened. Figure 1 shows a circuit which will do this.
Note that the CPURST output is the OR of the 82370 RESET input and the output of the software reset command logic, and thus will have the same duration as the RESET input during power-on.

The additional circuitry required consists of an OR gate, a one-shot, a capacitor, and a resistor more than is found in a system without the 82370. The one-shot (74121) is inserted between the CPURST output of the 82370 and the input of the retiming flip-flops (74F379). The period of the one-shot should be long enough to guarantee the 80 CLK2 periods that the 80376 requires.

The OR gate (74F32) is required to guarantee that the 80376 is held in a RESET state while the 82370 is being reset. This is done to be sure that BE3# is held low when the RESET input to the 82370 goes inactive. BE3# is used during the reset to determine whether it is necessary to enter a special factory test mode. It must be low when the RESET input goes inactive, and the 80376 drives it low during reset.

Figure D-1. Extending 82370 Reset Output
Development Support Tools
COMPREHENSIVE ARCHITECTURE DEVELOPMENT SUPPORT FOR 80960 EMBEDDED APPLICATIONS

Intel's 80960 Development Starter Kits provide a quick, easy and economic way to evaluate Intel's 80960 architecture, benchmark 80960 performance, and begin initial application code development and debug. Tools were designed specifically for the 80960 microprocessor, allowing developers to take full advantage of the performance and ease-of-programming features built into the 80960's RISC-based design. The 80960 Development Starter Kits are conveniently hosted on the IBM PC/AT, meaning developers of 32-bit embedded microprocessor applications can get started with minimal hardware investment.

FEATURES

- ASM-960 macro assembler for developing and tuning speed-critical code.
- IC-960 highly optimizing C language compiler for high-level language software development.
- EVA-960KB plug-in software execution board for benchmarking performance, evaluating architecture, and developing and debugging application code.
- Many starter kit configurations for supporting a wide range of development needs.
- DOS-hosted on IBM PC/AT, and compatibles.
- VAX/VMS*-hosted ASM-960 and IC-960 on VAX/VMS and MicroVAX/VMS* in Q4, 1988

* VAX/VMS and MicroVAX/VMS are trademarks of Digital Equipment Corp.
** Sun 3 is a trademark of Sun Microsystems.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patents licenses are implied. Information contained herein supersedes previously published specifications on these devices from Intel and is subject to change without notice.

© Intel Corporation 1988

September 1988
Order Number: 389794-302

5-1
ASM-960 MACRO ASSEMBLER

The ASM-960 macro assembler is used to fine-tune sections of code for top program execution speed on the 80960KA, 80960KB, and 80960MC. ASM-960 does this by giving programmers absolute control over program instructions. In addition to the assembler and macro preprocessor, ASM-960 includes several utilities for application program maintenance and debug:

- LINKER/LOADER allows multiple and incremental program file links.
- ARCHIVER allows developers to build applications function libraries.
- DISASSEMBLER provides assembler mnemonics.
- SYMBOL DUMPER provides symbolic information from a program file for facilitating debug.
- PROM BUILDER produces a hex file suitable for PROM programmers.

IC-960 C LANGUAGE COMPILER

IC-960 is a highly optimizing C language compiler for the 80960KB and 80960MC microprocessors. IC-960 supports the full C language as described in the Kernighan and Ritchie book, *The C Programming Language* (Prentice-Hall, 1978). IC-960 is used in conjunction with ASM-960 for outputting object code files and includes standard ANSI extensions to the C language and the following enhancements for embedded application development:

- **Constants** allow high-level language definitions and ease of program maintenance.
- **Memory-mapped I/O** allows high-level language access to application specific input and output.
- **Inline assembly** simplifies the integration of convenient C language and speed-critical functions.
- **Floating point support** produces in-line code to take full advantage of the floating point capability of the 80960KB and 80960MC.

The DOS-hosted version requires a 2MB Aboveboard®. The execution vehicle hosted version requires the 4MB board (EVA960KB4MB) and provides a 5X compile-time speed improvement over the DOS-hosted version.

EVA-960KB SOFTWARE EXECUTION VEHICLE

The EVA-960KB is a software execution vehicle for the 80960KB microprocessor. It is a single PC AT plug-in board which provides easy and convenient architecture evaluation and benchmarking as well as software development. The EVA-960KB contains the following:

- 1M byte or 4M byte of one wait-state program memory (SRAM)
- 64K bytes of zero wait-state program memory (SRAM)
- Three application program accessible timers
- Hosted debug monitor which supports 2 program breakpoints, single step program execution, register and memory access, program download and upload
- DOS access libraries that allow: screen display, keyboard input, read and write disk files, ability to spawn a DOS process which could communicate to serial or parallel I/O
- 20MHz operation

ARCHITECTURE EVALUATION: STARTER KIT 1

The 80960 Development Starter Kit #1 is designed for immediate architecture evaluation and code development for 80960KA and 80960KB. It includes the EVA-960KB execution vehicle and ASM-960 Assembler for developing and debugging speed-critical software and performing basic performance benchmarking.

COMPLETE APPLICATION DEVELOPMENT: STARTER KIT 2

The 80960 Development Starter Kit #2 is a complete application development toolkit for developing and debugging both speed-critical and high-level software, as well as performing full benchmarking on the 80960KB. The kit includes the EVA-960KB Software Execution Vehicle, the ASM-960 Assembler, and the IC-960 C Language Compiler.

SOFTWARE DEVELOPMENT: STARTER KIT 3

The 80960 Development Starter Kit #3 provides all the software needed to get started on all phases, both speed critical and high-level, of software development. The kit includes the ASM-960 Assembler and the IC-960 C Language Compiler.

SOFTWARE DEVELOPMENT: STARTER KIT 4

The Software Development Starter Kit #4 includes ASMB960M Assembler and C960C C Compiler hosted on a MicroVAX/VMS. It provides all of the software needed to get started developing an 80960 application.

SOFTWARE DEVELOPMENT: STARTER KIT 5

The Software Development Starter Kit #5 includes ASMB960V Assembler and C960V C Compiler hosted on VAX/VMS. It provides all of the software needed to get started developing an 80960 application.

FAST APPLICATION DEVELOPMENT: STARTER KIT 6

The Fast Development Starter Kit, Starter Kit #6, provides a development time speed improvement for customers that already own ASMB960D. The kit includes the 4M byte execution vehicle, EVA960KB1MB, and the execution vehicle hosted C Compiler. C960EP.

COMPLETE FAST APPLICATION DEVELOPMENT: STARTER KIT 7

The Complete Fast Development Starter Kit, Starter Kit #7, provides a complete and fast development toolkit. The kit includes the 4M byte execution vehicle, EVA960KB1MB, ASMB960D Assembler and the execution vehicle hosted C Compiler. C960EP.
SOFTWARE DEVELOPMENT:
STARTER KIT 8

The Software Development Starter Kit #8 includes ASM960U Assembler and C960U C Compiler hosted on a Sun 3 workstation. It provides all of the software needed to get started developing an 80960 application.

SERVICE, SUPPORT AND TRAINING

Intel augments its 80960 development tools with a full array of seminars, classes and workshops; field application engineering expertise; and telephone and on-site support at all stages of development.

ORDERING INFORMATION

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASM·960D</td>
<td>ASM·960 Assembler contains the assembler, linker/loader, macro preprocessor, archiver, PROM builder, and other object module utilities. DOS hosted. Requires a class I software license agreement plus addendum.</td>
</tr>
<tr>
<td>ASM960M</td>
<td>Same as above, MicroVAX/VMS hosted. Requires a class I software license agreement plus addendum.</td>
</tr>
<tr>
<td>ASM960V</td>
<td>Same as above, VAX/VMS hosted. Requires a class I software license agreement plus addendum.</td>
</tr>
<tr>
<td>ASM960U</td>
<td>Same as above, Sun 3 hosted. Requires a class I software license agreement plus addendum.</td>
</tr>
<tr>
<td>C960DP</td>
<td>IC960 optimizing C compiler, with ANSI extensions for the embedded applications, contains standard STDIO libraries and inline assembly capability. DOS hosted. Requires a 2M byte Above™ Board.</td>
</tr>
<tr>
<td>C960EP</td>
<td>Same as above, execution vehicle (EVA960KB4MB) hosted.</td>
</tr>
<tr>
<td>C960M</td>
<td>Same as above, MicroVAX/VMS hosted.</td>
</tr>
<tr>
<td>C960V</td>
<td>Same as above, VAX/VMS hosted.</td>
</tr>
<tr>
<td>C960U</td>
<td>Same as above, Sun 3 hosted.</td>
</tr>
<tr>
<td>EVA960KB</td>
<td>Software development and execution vehicle for the 80960KA and 80960KB microprocessors, contains 1M byte DRAM program memory.</td>
</tr>
<tr>
<td>EVA960KB4MB</td>
<td>Same as above plus EVA960KB4MB execution vehicle and C960EP execution vehicle hosted C Compiler.</td>
</tr>
</tbody>
</table>

960SKIT1 Architecture Evaluation Kit includes EVA960KB execution vehicle plus ASM·960D Compiler.
960SKIT2 Same as above plus IC960DP Compiler (requires Above™ Board)
960SKIT2AB Same as SKIT2 plus Intel Above™ Board with 2M byte memory
960SKIT3 Contains ASM·960D and IC960DP Compiler
960SKIT3AB Same as above plus Intel Above™ Board with 2M byte memory
960SKIT4 Contains ASM960M Assembler and C960M C Compiler, hosted on MicroVAX/VMS.
960SKIT5 Contains ASM960V Assembler and C960V C Compiler, hosted on VAX/VMS.
960SKIT6 Fast development kit contains EVA960KB4MB execution vehicle and C960EP execution vehicle hosted C Compiler.
960SKIT7 Same as above plus ASM960D Assembler.
960SKIT8 Same as above plus ASM960U Assembler and C960U C Compiler, hosted on Sun 3.
A COMPLETE ADA SOLUTION FOR REAL-TIME EMBEDDED APPLICATIONS

Ada-960 from Intel is a complete Ada development environment for 80960MC based real-time, embedded applications.

The 80960MC is a high performance, 32-bit military embedded processor especially designed to support Ada in fault-tolerant, shared-memory multiprocessor applications.

Ada-960 is hosted on VAXVMS.* The cross-development environment includes a highly optimizing Ada cross-compiler, a linker, a librarian, a source-level symbolic debugger, a target monitor, predefined packages and subprograms, the Ada run-time system, a user guide, and a detailed run-time system implementor’s guide. The run-time system makes optimal use of the Ada support built into the 80960MC processor and is carefully designed for real-time embedded applications.

FEATURES
• Complete VAXVMS® hosted Ada cross development environment
• Makes optimal use of the Ada support offered by the 80960 MC
• Run-time system is small, fast and predictable for real-time applications
• Designed for embedded applications with a highly optimizing compiler, selective linking, highly modular reconfigurable run-time, and source level symbolic debugger interfacing to a target monitor, or an emulator
THE 80960MC AND ADA: A GOOD MATCH

The Intel 80960MC embedded processor is designed to support applications written in Ada. Ada-960 from Intel is implemented to make optimal use of the Ada support built into the 80960MC.

- Ada-960 maps Ada tasks directly to 80960MC processes.
- Ada-960 uses the 80960MC embedded processor hardware to dispatch and manage Ada tasks.
- Ada-960 maps Ada task priorities directly to 80960MC process priorities.
- Ada-960 uses the 80960MC memory management unit to provide inter-task protection.
- Ada-960 uses 80960MC semaphores to implement run-time system critical sections.
- Ada-960 uses the 80960MC on-chip floating point unit to perform floating point operations.

The unique architecture of the 80960MC allows Ada-960 to use the processor hardware to provide functionality normally implemented in software on other architectures. This includes automatic dispatching and preemptive priority scheduling of Ada tasks.

Ada-960's use of the 80960MC makes the run-time easily extendable to support fault-tolerant shared-memory multiprocessor configurations supported by the 80960MC embedded processor.

ADA-960 FOR REAL-TIME APPLICATIONS

Ada-960 is carefully designed for use in the development of real-time applications. Some of the real-time features of Ada-960 include:

Minimum "Interrupts Off" Time: The Ada-960 run-time system disables interrupts for a minimal amount of time. The "interrupts off" time does not vary with the size of the application.

Pre-emptive Priority Scheduling: Ada-960 provides a fully pre-emptive, priority-driven tasking run-time. The 80960MC hardware is used to ensure that the highest priority task is always the one that is running. The run-time system uses the 80960MC hardware to switch to a higher priority task (than the one currently executing) whenever such a task becomes ready to run.

Predictable Performance: Ada-960 provides predictable performance that is insensitive to target system load. System response time remains constant and fully deterministic as the number of tasks grows.

- Ada-960 ensures that scheduling latency is independent of system load.
- Ada-960 guarantees response-time to interrupts.
- Ada-960 provides predictable memory allocation times. Memory allocation is implemented through efficient algorithms that ensure a constant upper bound on the time taken to allocate memory.

Run-time Extension: Ada-960 provides a run-time system extension package that gives applications dynamic control over tasking, scheduling, critical sections and other run-time functions.

ADA-960 FOR EMBEDDED APPLICATIONS

Ada-960 is designed for embedded applications and provides:

Small, Fast run-time System: Ada-960 provides a compact, high performance run-time system. The run-time system is very modular to support selective linking by the Ada-960 linker. The modularity of the run-time and the selective linking features of the linker ensure that all unused Ada language features are automatically omitted from the application's final executable image.

Retargetable Run-time System: Ada-960 provides an easily retargetable run-time system. The run-time system is designed to be easily retargeted to custom 80960MC boards and comes with all the necessary source files and documentation for on-site customization to specific interrupt and IO requirements.

VAX/VMS Hosted Remote Debug: Ada-960 provides a VAX/VMS-hosted source-level, symbolic Ada debugger. The debugger allows users to debug applications on a remote 80960MC target via its interface to either the standard Intel 80860MC monitor or the standard Intel 80960MC in-circuit-emulator.

Retargetable Target Monitor: Ada-960 provides an easily retargetable target monitor. The target monitor resides on the target board. The monitor communicates with and supports the Ada debugger hosted on VAX/VMS. The target monitor is easily retargetable to custom boards. This allows the Ada debugger to be used in debugging applications on non-standard 80960MC hardware configurations.

ROMable Code and 86HEX: The Ada-960 compiler produces ROMable code. The Ada linker can produce an application's executable image in 86Hex so that the application may be easily burnt into PROM's.

Combining Ada and non-Ada Code: Ada-960 provides implementation-defined pragma FOREIGN-BODY and pragma LINKAGE-NAME to support the combination of Ada and non-Ada code.

Chapter 13 Support: Ada-960 provides chapter 13 support including representation specifications, machine-code insertion, interrupt entries, and so on.
The Ada-960 cross development environment includes tools for compiling, linking, and debugging, along with libraries and a complete set of documentation. The Ada-960 cross development environment from Intel makes the following support, documentation, tools, and software available:

**Compiler:** A fast, highly optimizing Ada-960 compiler that generates efficient, compact 80960MC code.

The compiler performs virtually all optimizations that are "traditional", as well as several that are Ada-specific. Each optimization is carefully tuned to the 80960MC architecture.

Optimizations performed include transformations that affect value and variable handling, code motion and elimination, tail recursion elimination, and loop strength reduction. Ada-specific data packing and code transformations such as constraint check elimination, overflow check elimination, and parameter binding are also performed. The Ada-960 code generator schedules generated machine instructions to make optimal use of parallel execution opportunities available on the 80960MC embedded processor.

**Librarian:** An Ada-960 librarian to manage the Ada program library. The librarian controls the interaction of compilation units and the linking of executable images. The Ada-960 librarian supports the Ada separate compilation and dependency control requirement.

**Linker:** An advanced Ada-960 linker to support enhanced selective linking at the subprogram level. Subprograms that are not used in an Ada application are not linked unless specifically requested. Full control over memory layout and mapping is supported with a rich command language.

**Debugger:** A very powerful Ada-960 source-level, symbolic Ada debugger that supports the debugging of both pure Ada and combined Ada code. The Ada-960 debugger is hosted on VAX/VMS and interfaces with target 80960MC boards through either the standard Intel 80960MC target monitor or the standard 80960MC in-circuit-emulator.

The Ada-960 debugger allows users to examine and modify their applications using the same names that appear in the source program. Users can evaluate Ada expressions, set breakpoints and tracepoints, and debug multi-tasking Ada programs.

Program breakpoints can be made conditional on arbitrary conditions, and debugger commands can be executed automatically at breakpoints.

The Ada-960 debugger can call functions and procedures in an Ada application. This feature can be used to extend the set of debugger facilities or to test parts of the application interactively.

The Ada-960 debugger allows users to display Ada variables in formats appropriate to each of their types. Users can also specify formats appropriate to the current application. Special browsing features within the debugger eliminate the need for paper listings during debugging sessions.

The Ada-960 debugger provides a flexible display of the state of Ada tasks and can display the current callstack within any task. The debugger can list tasks on various run-time queues and can suspend or change the priority of tasks.

The source level features of the Ada-960 debugger are complemented by a complete set of machine-level commands.

Script files containing debugger commands may be created in and executed by the debugger. The Ada-960 debugger can record a log of all debugging actions for later analysis or replay.

For better programmer productivity, the debugger has a multiwindow interface with separate windows for debugger commands, Ada source and program output. The Ada-960 debugger provides "scoreboard windows" for real-time display of user-selected program information.

The debugger is usable from all types of terminals and has special features to support bit-mapped displays.

Use of the debugger is possible at all optimization levels without recompilation of the Ada program.

**Target Monitor:** A standard Intel 80960MC target monitor that is easily re-targetable to custom 80960MC boards. The target monitor comes with all the necessary source files and documentation for on-site customization to specific interrupt and I/O requirements.

**ICE:** A standard Intel 80960MC in-circuit-emulator. The in-circuit-emulator delivers real-time emulation at processor speed, and allows non-intrusive debugging of applications under development.

**Predefined Packages and Subprograms:** An Ada-960 library containing precompiled predefined Ada packages and subprograms.

**Run-time System:** An Ada-960 library containing the Ada run-time system. The run-time system is small, fast, and predictable. The Ada-960 run-time is re-targetable to different 80960MC boards and is especially designed for real-time embedded applications. The run-time system is carefully designed to make optimal use of the Ada support provided by the 80960MC embedded processor.
The run-time system comes with all the necessary source files and documentation for on-site customization to specific interrupt and I/O conventions. Most of the run-time system is written in Intel Ada-960 with a small portion written in Intel ASM-960 assembler.

**Run-time System Extension**: A run-time system extension package to allow Ada applications dynamic control over the tasking, scheduling, critical sections, and other run-time functions.

**Source Code**: Source code for both the run-time system and the run-time system extension.

**Documentation**: High quality documentation including a User Guide and a detailed Run-time System Implementor’s Guide. The Run-time System Implementor’s Guide provides full documentation of the interface between the Ada-960 compiler and the run-time system. It also documents the design of, and provides guidance to, modifying the run-time system.

**WORLDWIDE SERVICE AND SUPPORT**

Intel augments its 68060 architecture family development tools with a full array of seminars, classes, and workshops; on-site consulting services; field application engineering expertise; telephone hot-line support; and software and hardware maintenance contracts. This full line of services will ensure your design success.

This is preliminary information. Changes to the product can be made at any time.

*VAX/VMS is a trademark of Digital Equipment Corp.*
IN-CIRCUIT EMULATOR FOR THE 80960KA AND 80960KB MICROPROCESSORS

The ICE™-960KB In-Circuit Emulator delivers real-time hardware and software debugging capabilities for 80960KA/KB-based designs. The capabilities include emulation of the 80960KA/KB microprocessor, hardware and software breakpoint specification, fastbreaks, two types of trace capability, large trace buffering and sophisticated human interface. The ICE-960KB In-Circuit Emulator gives you unmatched control over all phases of hardware/software debug, including developing, integrating and testing, which improves the developers productivity and speeds time to market.

FEATURES

- Real-Time Emulation of the 80960KA/KB microprocessors up to 20MHz
- 256K bytes of memory in Standalone Self Test Unit
- Zero wait-state operation from user system memory
- Examine and modify memory and the 80960 Registers
- 2 hardware and 32 software Breakpoints settable on any Instruction Address, and break on Trace Buffer Full
- Hosted on IBM PC-AT® running DOS (version 3.3)
- Assembly and Disassembly of code in 80960 instruction mnemonics
- Dynamically monitor or update program variables or memory with Fastbreaks

- Real-time Bus Trace with Time-Tags for tracking code execution times
- Execution Trace for tracking instruction execution inside on-chip Instruction Cache
- Stores 1024 frames of program execution history or bus cycles or both
- Versatile software featuring Color, Pulldown Menus, Forms, Command Line with Syntax Guidance and Editing, Control Constructs, Debug Procedures and DOS Command Input (shell)
REAL-TIME EMULATION

The ICE-960KB In-Circuit Emulator provides emulation of the 80960KA/KB at speeds up to 20 MHz, thus providing early detection of subtle timing problems that may arise at full speed. Intel's intimate knowledge of the component makes possible the tightest conceivable conformance between timing parameters of the emulator and the target microprocessor.

PROCESSOR/MEMORY EXAMINATION AND MODIFICATION

The 80960KA/KB registers can be accessed mnemonically (e.g. gl2, r5, fp3) with the ICE-960KB emulator software. Data can be displayed or modified in one of four bases (hexadecimal, decimal, octal, or binary). Program memory contents can be disassembled and displayed as 80960 assembly instruction mnemonics. Additionally, 80960 assembly instruction mnemonics can be assembled and stored into program memory.

PROGRAM TRACING

The ICE-960KB emulator can store 1024 frames of program execution history or 5120 cycles of the 80960KA/KB address/data bus activity in the trace buffer. Each frame of program execution contains a discontinuity address (branch, call, return, etc.), and a time-tag. This information can be used to reconstruct a history of the program execution. With the execution trace option enabled, the ICE-960KB will run at less than full speed; typically 70-90% of full-speed. Each trace frame of bus cycles contains one complete bus burst access, the address cycle followed by the four data cycles, and a time-tag. While using bus trace, the ICE-960KB runs at the full-speed of the 80960KA/KB microprocessor.

EVENT RECOGNITION (BREAKPOINT CONTROL) AND EMULATION CONTROL

Two hardware and thirty-two software breakpoints can be active at any time. The ICE-960KB emulator allows any number of breakpoints to be defined and then activated when needed. The breakpoints can be set on any instruction address. Additionally, emulation can be automatically stopped when the trace buffer is full. Besides the ability to execute program code at full speed between specified points, the ICE-960KB emulator provides the capability to single-step through program code. Fastbreaks are short pauses in program execution to examine or modify memory or 80960 registers.

STANDALONE OPERATION

Product software can be developed and debugged prior to and independent of hardware availability with the Standalone Self Test unit (SAST), which contains 256K bytes of two wait-state program memory. The SAST also provides diagnostic testing to assure full functionality of the ICE-960KB emulator.

VERSATILE AND POWERFUL HOST SOFTWARE

The easy to use ICE-960KB emulator software takes advantage of color and pull-down menus to complement its already powerful command set. The software includes: an on-line help facility, a dynamic command entry and syntax guide, screen oriented editor, assembler and disassembler, input/output redirection, command piping, DOS command entry, and the ability to customize the command set via debug procedures and literal definitions.

DEBUG PROCEDURES AND LITERALS

Debug procedures (PROCs) are user-defined groups of ICE-960KB emulator commands. They can be stored on disk and recalled during later debugging sessions. PROCs can be used to simplify the process of debugging by grouping repetitive or a required ordering of emulator commands, which can then be accessed by typing the name of the PROC. Literals are user-defined abbreviations for whole or partial ICE-960KB emulator commands. Literals are a shorthand method of customizing the emulator commands to fit your needs and preferences.

SPECIFICATIONS

HOST REQUIREMENTS

IBM PC-AT* (minimum requirements) with 640KB of conventional memory
1MB of RAM (Lotus, Intel, Microsoft expanded memory specification)
(Intel's Above™board with 1.0MB RAM is required)
20 MB Fixed Disk
At least one 3½" Floppy Disk drive
A serial interface
80287 Numerics Coprocessor
DOS Operating System (version 3.3)

REQUIRED SYSTEM RESOURCES

The ICE-960KB emulator requires the following: a) exclusive use of the 80960KA/KB's on-chip debug registers and b) 304 bytes of target system RAM in the register save area of the stack, 256 bytes for flushing the 80960 local registers, and 48 bytes for saving the processor control block (PRCB).
**MECHANICAL SPECIFICATIONS**

### TABLE 1. ICE-960KB Emulator Physical Characteristics

<table>
<thead>
<tr>
<th>Unit</th>
<th>Width</th>
<th>Height</th>
<th>Length</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control unit</td>
<td>10.5</td>
<td>26.7</td>
<td>1.5</td>
<td>3.8</td>
</tr>
<tr>
<td>Processor module*</td>
<td>3.8</td>
<td>9.6</td>
<td>1.5</td>
<td>3.8</td>
</tr>
<tr>
<td>SAST</td>
<td>6.0</td>
<td>15.2</td>
<td>2.0</td>
<td>5.1</td>
</tr>
<tr>
<td>OIB</td>
<td>3.8</td>
<td>9.6</td>
<td>0.9</td>
<td>2.3</td>
</tr>
<tr>
<td>Power supply</td>
<td>2.8</td>
<td>7.1</td>
<td>4.2</td>
<td>10.7</td>
</tr>
<tr>
<td>User cable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serial cable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*measurement includes target adaptor

---

**ELECTRICAL SPECIFICATIONS**

### SYNC Line Specification

The SYNCIN line must be valid for at least one instruction cycle because it is only sampled on instruction boundaries. The SYNCIN line is a standard TTL input. The SYNCOUT line is driven by a TTL open collector with a 4.75K-ohm pull-up resistor.

### AD/DC Specifications

The following tables describe the DC specification differences between the ICE-960KB emulator and the 80960KA/KB microprocessor; for more details refer to the User Guide.

---

### TABLE 2. AC Specifications With The OIB Installed

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>t12</td>
<td>clock low time</td>
<td>t12 + 1ns</td>
<td></td>
</tr>
<tr>
<td>t13</td>
<td>clock high time</td>
<td>t13 + 1ns</td>
<td></td>
</tr>
<tr>
<td>t16</td>
<td>output valid delay</td>
<td>t16 + 8ns</td>
<td>t16 + 16ns</td>
</tr>
<tr>
<td>A/D 0:31</td>
<td>DT/R*,DEN*,BEO-3*,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADS*,W/R*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HILDA,CACHE,LOCK*,INTA*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALE*</td>
<td></td>
<td>t16 + 10ns</td>
<td>t16 + 20ns</td>
</tr>
<tr>
<td>t17</td>
<td>ALE* width</td>
<td>t17-6.5ns</td>
<td></td>
</tr>
<tr>
<td>t18</td>
<td>ALE* disable delay</td>
<td>t18 + 8ns</td>
<td>t18 + 14ns</td>
</tr>
<tr>
<td>t19</td>
<td>output float delay</td>
<td>t19 + 5ns</td>
<td>t19 + 22ns</td>
</tr>
<tr>
<td>A/D 0:31</td>
<td>DT/R*,DEN*,BEO-3*,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADS*,W/R*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HILDA,CACHE,LOCK*,INTA*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t10</td>
<td>input setup 1</td>
<td>t10 + 2ns</td>
<td></td>
</tr>
<tr>
<td>t11</td>
<td>BADAC*,INT0-3* deassertion</td>
<td>t10 + 14ns</td>
<td></td>
</tr>
<tr>
<td>t16</td>
<td>reset setup time</td>
<td>t16 + 6</td>
<td></td>
</tr>
</tbody>
</table>

*symbol refers to 80960KB specification
TABLE 3. ICE-960KB Emulator DC Specifications

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM-Icc</td>
<td>Supply current with 80960KB-20</td>
<td>1400mA</td>
</tr>
<tr>
<td>OIB-Icc</td>
<td>Supply current</td>
<td>PM-Icc + 1100mA</td>
</tr>
</tbody>
</table>

TABLE 4. Additional DC Loading

<table>
<thead>
<tr>
<th>Signal</th>
<th>(without OIB installed)</th>
<th>(with OIB installed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IffI</td>
<td>Maximum</td>
<td>III Maximum</td>
</tr>
<tr>
<td>IffI</td>
<td>Maximum</td>
<td>III Maximum</td>
</tr>
<tr>
<td>AD(0:31)</td>
<td>100 µA 0.6 mA</td>
<td>20 µA - 1 mA</td>
</tr>
<tr>
<td>DEX#</td>
<td>40 µA 1.0 mA</td>
<td>20 µA - 1 mA</td>
</tr>
<tr>
<td>W/R#</td>
<td>140 µA 1.6 mA</td>
<td>20 µA - 1 mA</td>
</tr>
<tr>
<td>ADS#</td>
<td>140 µA 1.6 mA</td>
<td>20 µA - 1 mA</td>
</tr>
<tr>
<td>CLK2</td>
<td>80 µA 2.2 mA</td>
<td>50 µA - 2 mA</td>
</tr>
<tr>
<td>RESET</td>
<td></td>
<td>50 µA - 2 mA</td>
</tr>
<tr>
<td>BE(0:3)#</td>
<td>20 µA - 1 mA</td>
<td></td>
</tr>
<tr>
<td>DT/R#</td>
<td>20 µA - 1 mA</td>
<td></td>
</tr>
<tr>
<td>INTO#,INT3#</td>
<td>20 µA - 1 mA</td>
<td></td>
</tr>
<tr>
<td>INT1,INT2</td>
<td>20 µA - 1 mA</td>
<td></td>
</tr>
<tr>
<td>BADAC#</td>
<td>20 µA - 1 mA</td>
<td></td>
</tr>
<tr>
<td>ALE#</td>
<td>20 µA - 1 mA</td>
<td></td>
</tr>
<tr>
<td>LOCK#</td>
<td>20 µA - 1 mA</td>
<td></td>
</tr>
<tr>
<td>READY#</td>
<td>20 µA - 1 mA</td>
<td></td>
</tr>
<tr>
<td>HOLD</td>
<td>20 µA - 1 mA</td>
<td></td>
</tr>
<tr>
<td>FAILURE#</td>
<td>20 µA - 1 mA</td>
<td></td>
</tr>
</tbody>
</table>

**Power Supply**

- 100-120V or 220-240V (Selectable)
- 50-60 Hz
- 2 amps (AC Max) @ 120V
- 1 amp (AC Max) @ 240V

**Environmental Characteristics**

- Operating Temperature: 10 C to 40 C (50 F to 104 F)
- Operating Humidity: Maximum 85% Relative Humidity, non-condensing

**Ordering Information**

- **Order Code**
  - ICE960KB: The complete ICE-960KB emulator system including control unit, processor module, power supply, SAST, OIB, SAB, serial communications cable (SCOM4), IEDIT, software version 1.x, and upgrade certificate for version 2.0 software. (Requires software license, Class I)

- **Description**
  - ASM960D: DOS hosted assembler, linker/loader, macro preprocessor, archiver (librarian), PROM builder, and other object module utilities. (Requires software license, Class I plus addendum I)
  - C960D: DOS hosted optimizing C compiler, with ANSI extensions for embedded applications, contains standard STUDIO libraries and has inline assembly capability. Requires a 2M byte Above*board. (Requires software license, Class I)

For direct information on Intel's Development Tools, or for the number of your nearest sales office or distributor, call 800-874-6835 (U.S.). For information or literature on additional Intel products, call 800-548-4725 (U.S. and Canada).

**United States**, Intel Corporation 3065 Bowers Ave., Santa Clara, CA 95051 Tel: (408) 765-8080

**Japan**, Intel Japan K.K. 5-6 Tokodai, Tsukuba-shi, Ibaraki, 300-26 Tel: 029747-8511

**United Kingdom**, Intel Corporation (U.K.) Ltd. Pipers Way, Swindon, Wiltshire, England SN3 1RJ Tel: (0793) 696000

*IBM PC/AT is a trademark of IBM*
<table>
<thead>
<tr>
<th>STATE</th>
<th>CITY</th>
<th>ADDRESS</th>
<th>PHONE</th>
<th>FAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALABAMA</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>ARIZONA</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>CALIFORNIA</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>COLORADO</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>CONNECTICUT</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>FLORIDA</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>GEORGIA</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>NEW MEXICO</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>TEXAS</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>VIRGINIA</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>WISCONSIN</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>CANADA</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>BRITISH COLUMBIA</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>ONTARIO</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
<tr>
<td>QUEBEC</td>
<td>Intel Corp.</td>
<td>3900 Corporate Parkway, 1000 Toyota Central, Bldg. A</td>
<td>(205) 530-1900</td>
<td></td>
</tr>
</tbody>
</table>
# DOMESTIC DISTRIBUTORS

## ALABAMA
- **Arrow Electronics, Inc.**
  - 1913 Henderson Dr. North
  - Huntsville, 35805
  - Tel: (256) 877-1955

## CALIFORNIA
- **Hamilton/Avnet Electronics, Inc.**
  - 6064 University Plaza
  - Winter Park, 32793
  - Tel: (407) 641-4100
  - TWX: 910-395-2538

## COLORADO
- **Hamilton/Avnet Electronics, Inc.**
  - 3935 South Wadsworth Blvd., Suite 1
  - Littleton, 80123
  - Tel: (303) 794-8500

## CONNECTICUT
- **Vieweg Distribution Group**
  - 1115 Sen. Center Rd.
  - Ronco, 06380
  - Tel: (203) 838-3062

## FLORIDA
- **Hamilton/Avnet Electronics, Inc.**
  - 6002 Oak Hill Lane
  - Coral Gables, 33146
  - Tel: (305) 833-0050
  - TWX: 910-833-0234

## GEORGIA
- **Wynn/Avnet Electronics, Inc.**
  - 2003 Garden Park Dr.
  - Norcross, 30071
  - Tel: (770) 944-1300

## IOWA
- **Hamilton/Avnet Electronics, Inc.**
  - 3520 Third Street
  - Davenport, 52806
  - Tel: (563) 392-0550

## ILLINOIS
- **Hamilton/Avnet Electronics, Inc.**
  - 5425 Pheasant Run
  - Aurora 60507
  - Tel: (630) 892-2618

## KANSAS
- **Wynn/Avnet Electronics, Inc.**
  - 4420 E. 2nd St.
  - Kansas City, 66111
  - Tel: (913) 384-9090

## KENTUCKY
- **Wynn/Avnet Electronics, Inc.**
  - 3062 South Third St.
  - Louisville, 40209
  - Tel: (502) 586-0060

## MASSACHUSETTS
- **Pioneer/Avnet Electronics, Inc.**
  - 5100 Galbraith Rd.
  - Danvers, 01923
  - Tel: (978) 795-1811

## MICHIGAN
- **Pioneer/Avnet Electronics, Inc.**
  - 1255 N. Sycamore Avenue
  - Kalamazoo, 49001
  - Tel: (269) 388-8600

## MARYLAND
- **Hamilton/Avnet Electronics, Inc.**
  - 6002 Oak Hill Lane
  - Columbia, 21046
  - Tel: (301) 995-0003
  - TWX: 710-236-9000

## MONTANA
- **Pioneer/Avnet Electronics, Inc.**
  - 1400 Grand Avenue
  - Missoula, 59801
  - Tel: (406) 721-8800

## MARYLAND
- **Hamilton/Avnet Electronics, Inc.**
  - 6002 Oak Hill Lane
  - P.O. Box 4014
  - Columbia, 21046
  - Tel: (301) 995-0004

## MINNESOTA
- **Pioneer/Avnet Electronics, Inc.**
  - 135 W. 7th St.
  - Minneapolis, 55402
  - Tel: (612) 798-0600

## MISSOURI
- **Pioneer/Avnet Electronics, Inc.**
  - 135 W. 7th St.
  - St. Louis, 63101
  - Tel: (314) 944-8856

## MISSISSIPPI
- **Pioneer/Avnet Electronics, Inc.**
  - 3500 Medical Center Dr.
  - Jackson, 39216
  - Tel: (601) 359-9000

## MONTANA
- **Pioneer/Avnet Electronics, Inc.**
  - 1400 Grand Avenue
  - Missoula, 59801
  - Tel: (406) 721-8800

## NEW HAMPSHIRE
- **Pioneer/Avnet Electronics, Inc.**
  - 135 W. 7th St.
  - Manchester, 03102
  - Tel: (603) 627-9490

## NEW JERSEY
- **Pioneer/Avnet Electronics, Inc.**
  - 40 East Street
  - Morristown, 07960
  - Tel: (973) 585-6080

## NEW YORK
- **Pioneer/Avnet Electronics, Inc.**
  - 135 W. 7th St.
  - Rochester, 14623
  - Tel: (716) 631-5000

## PENNSYLVANIA
- **Pioneer/Avnet Electronics, Inc.**
  - 2029 Wickham Ave.
  - Allentown, 18102
  - Tel: (610) 584-9500

## SOUTH CAROLINA
- **Pioneer/Avnet Electronics, Inc.**
  - 135 W. 7th St.
  - Charleston, 29401
  - Tel: (803) 740-2426

## TENNESSEE
- **Pioneer/Avnet Electronics, Inc.**
  - 2029 Wickham Ave.
  - Nashville, 37205
  - Tel: (615) 234-5800

## TEXAS
- **Pioneer/Avnet Electronics, Inc.**
  - 135 W. 7th St.
  - Dallas, 75208
  - Tel: (214) 969-3900

## VIRGINIA
- **Pioneer/Avnet Electronics, Inc.**
  - 135 W. 7th St.
  - Richmond, 23220
  - Tel: (804) 643-0000

## WASHINGTON
- **Pioneer/Avnet Electronics, Inc.**
  - 135 W. 7th St.
  - Seattle, 98102
  - Tel: (206) 282-5000

## WISCONSIN
- **Pioneer/Avnet Electronics, Inc.**
  - 135 W. 7th St.
  - Milwaukee, 53201
  - Tel: (414) 271-1500

## WYOMING
- **Pioneer/Avnet Electronics, Inc.**
  - 135 W. 7th St.
  - Cheyenne, 82001
  - Tel: (307) 632-5000

---

*Microcomputer System Technical Distributor Center*
DOMESTIC DISTRIBUTORS (Cont’d.)

NEW YORK (Cont’d.)

1Pioneer Electronics
60 Chesterfield Parkway West
Gansevoort, NY 12195
Tel: (518) 657-4131
TWX: 510-234-6399

1Arrow Electronics
640 Fairview Park
Fairport 14450
Tel: (716) 885-8500
TWX: 510-535-7001

NORTH CAROLINA
1Arrow Electronics, Inc.
6240 Greenway Road
Raleigh 27604
Tel: (919) 878-0169
TWX: 510-928-1656

1Hampton/Avnet Electronics
3510 Spring Forest Drive
High Point 27262
Tel: (910) 978-1971

1Pioneer/Technologies Group, Inc.
5001 A Southern Pines Blvd.
Charlotte 28210
Tel: (704) 527-1188
TWX: 810-521-0366

OHIO
1Arrow Electronics, Inc.
7602 McEwen Road
Canonsville 45459
Tel: (513) 652-5843
TWX: 510-819-1111

1Arrow Electronics, Inc.
8338 Cooper Road
Sycamore 44056
Tel: (216) 896-4471
TWX: 510-427-9409

1Pioneer Electronics
504 Sycamore Drive
Dayton 45429
Tel: (513) 439-4133
TWX: 510-503-3331

1Hamiton/Avnet Electronics
45172 Mayflower Blvd
Warrensville Heights 44128
Tel: (216) 245-0386
TWX: 810-927-9452

1Hamiton/Avnet Electronics
77 Brookridge Blvd
Westlake 44145
Tel: (440) 816-2004

1Pioneer Electronics
443 Fieldway Boulevard
Dayton 45424
Tel: (513) 233-2000
TWX: 510-459-1622

1Pioneer Electronics
4661 13th Street
Cleveland 44116
Tel: (216) 877-5000
TWX: 810-422-2211

OKLAHOMA
1Arrow Electronics, Inc.
1217 E. 51st St., Suite 102A
Tulsa 74148
Tel: (918) 525-1737

1Hamiton/Avnet Electronics
12171 E. 51st St., Suite 102A
Tulsa 74148
Tel: (918) 525-1737

1Pioneer Electronics
18060 Kramer
Austin 78758
Tel: (512) 365-4000
TWX: 510-974-1323

1Pioneer Electronics
5853 Post West Drive
Houston 77026
Tel: (713) 988-5555
TWX: 510-881-1066

1Wyle Distribution Group
5250 N.E. 82nd Young Park
Suite 600
Hillsboro 97124
Tel: (503) 660-4000
TWX: 810-480-2033

OREGON
1Arrow Electronics Corp.
1865 N.W. 198th Place
Beaverton 97005
Tel: (503) 823-9500
TWX: 510-407-6746

1Hamiton/Avnet Electronics
6224 S.W. Jean Road
Beaverton 97005
Tel: (503) 683-7134
TWX: 510-407-6746

1Pioneer Electronics
299 Kappa Drive
Pittsburgh 15203
Tel: (412) 421-4150

1Pioneer Electronics
8800 Liberty Ave
Pittsburgh 15203
Tel: (412) 421-4150

1Pioneer/Technologies Group, Inc.
Industrial Park
Warren 60844
Tel: (815) 377-7000
TWX: 810-565-1066

Pennsylvania
1Arrow Electronics
650 Seco Road
Monroeville 15166
Tel: (412) 897-0000

1Hamiton/Avnet Electronics
2800 Liberty Ave
Pittsburgh 15222
Tel: (412) 421-4150

1Pioneer Electronics
289 Kappa Drive
Pittsburgh 15222
Tel: (412) 882-3200
TWX: 712-795-3122

1Pioneer/Technologies Group, Inc.
Industrial Park
Warren 60844
Tel: (815) 377-7000

1Right/Wyle Electronics
501 Herson Road
Heronwood 75081
Tel: (214) 235-9953

UTAH
1Arrow Electronics
1946 Parkway Blvd.
Salt Lake City 84119
Tel: (801) 973-8913

1Hamiton/Avnet Electronics
885 West 1000 West
Salt Lake City 84111
Tel: (801) 572-2000

1Pioneer Electronics
27525 N.E. 200th South
Issaquah 98077
Tel: (425) 392-5000

1Wyle Distribution Group
1525 West 2000 South
Suite 100
West Valley 84119
Tel: (801) 974-9653

WISCONSIN
1Hamiton/Avnet Electronics
2111 W. Wight Hill Lane
Milwaukee 53214
Tel: (414) 560-5111
TWX: 510-850-1110

1Hamiton/Avnet Electronics
4850 Wright Rd., Suite 190
Staunton 51747
Tel: (713) 340-7700
TWX: 510-681-9323

1Pioneer Electronics
18300 Kramer
Austin 78758
Tel: (512) 835-4000
TWX: 510-974-1323

1Pioneer Electronics
5853 Post West Drive
Houston 77026
Tel: (713) 988-5555
TWX: 510-881-1066

1Wyle Distribution Group
1810 Greenwood Avenue
Richmond 75081
Tel: (214) 235-9953

1Zentronics
62411 Mission Viejo
California 92692
Tel: (714) 273-1021

1LEFT/COLUMBIA
1Hamiton/Avnet Electronics
10055 South Boundary
Burnsville V8A 3H3
Tel: (604) 877-0231

1Zentronics
155 Notepad Road
New Berlin 53151
Tel: (608) 647-7057

WASHINGTON
1Hamiton/Avnet Electronics
13422 S.E. Eastgate Way
Bellevue 98007
Tel: (425) 644-0267

1Arrow Electronics, Inc.
13422 S.E. Eastgate Way
Bellevue 98007
Tel: (425) 644-0267

1Hamiton/Avnet Electronics
14212 N.E. 211th Street
Bellevue 98005
Tel: (425) 426-8075
TWX: 510-443-2489

1Wyle Distribution Group
4550 N.E. 180th Street
Renton 98059
Tel: (206) 881-1150

WISCONSIN
1Hamiton/Avnet Electronics
200 N. Plymouth Blvd., Ste. 100
Milwaukee 53213
Tel: (414) 767-8000
TWX: 510-850-1110

1Hamiton/Avnet Electronics
2975 Moorland Road
New Berlin 53151
Tel: (414) 744-5100
TWX: 510-260-1160

1Zentronics
155 Notepad Road
Unit C
New Berlin 53151
Tel: (608) 647-7057

1Zentronics
9010 W. InEffect Road
Milwaukee 53223
Tel: (414) 767-8000
TWX: 510-260-1160

1Wyle Distribution Group
155 Notepad Road
Unit C
New Berlin 53151
Tel: (608) 647-7057

ONTARIO (Cont’d.)

1Hamiton/Avnet Electronics
10031 McDermott
Mississauga, L5T 1M4
Tel: (905) 873-7789
TWX: 06-212813

1Hamiton/Avnet Electronics
6845 Reesor Road
Units 3-4
Mississauga, L4S 1R2
Tel: (905) 492-8607

1Hamiltron/Avnet Electronics
6845 Reesor Road
Unit 6
Mississauga, L4S 1R2
Tel: (905) 492-8607

1Microcomputer System Technical Distributor Center
CG/SALE/111088
INTERNATIONAL SALES OFFICES

AUSTRALIA
Intel Australia Pty. Ltd.
330 Pacific Hwy.
Level 8
Crows Nest, NSW, 2065
Tel: (02) 947-3174
TLX: A4 20097
FAX: (02) 253-3352

BRAZIL
Intel Semicondutores do Brasil LTDA
Av. Paulista, 1159-C1344045
01311 - São Paulo - S.P.
Tel: 55-11-3097-8599
TLX: 115314 SAPI BR
FAX: 55-11-310-7581

CHINA/HONG KONG
Intel Semiconductor Ltd.
10/F East Tower
Bond Center
Tel: 852-2971-7011
FAX: 852-2971-7010

JAPAN
Intel Japan K.K.
5-6 Yoshida, Taito-ku
Tokyo, 100-25
Tel: 03-3741-8111
FAX: 03-3741-8450

JAPAN (Cont'd.)
Intel Japan K.K.
Nikon Semel Atugi Bldg.
1-2-1 Akihabara
Chuo-ku, Tokyo 121
Tel: 03-6206-9379
FAX: 03-6206-9381

JAPAN (Cont'd.)
Intel Japan K.K.
Shanghai Bldg.
1-1-11 Marunouchi
Chiyoda-ku, Tokyo 100
Tel: 03-3201-0682
FAX: 03-3200-9702

KOREA
Intel Technology Asia, Ltd.
Business Center 18th Floor
61, Yoido-Dong, Young Deung Po-Ku
Seoul 100
Tel: (02) 794-5186, 8286, 9386
TLX: K90210 INTELKO
FAX: 2794-0096

SINGAPORE
Intel Singapore Technology, Ltd.
101 Thomson Road #27-06
Goshin Square
Singapore 1100
Tel: 65-726-7911
TLX: 29697 INTEL
FAX: 65-725-9256

TWAIR
Intel Technology (Far East) Ltd.
Taiwan Branch
10F, No. 200, Tun Hwa N. Road
Taipei, R.O.C.
Tel: 886-2-715-9560
TLX: 13115 INTELWAN
FAX: 886-2-717-2455

INTERNATIONAL DISTRIBUTORS/REPRESENTATIVES

ARGENTINA
Dafysys S.R.L.
Characuao, 94-4 Piso
1065 Buenos Aires
Tel: 54-1-334-1871
FAX: 54-1-334-2267

AUSTRIA
Total Electronics
15-17 Hume Street
Huntinggates, 3156
Victoria, Australia
Tel: 03-544-5244
FAX: 03-544-5244

BRAZIL
Elta Microelectronics
R. Geraldo Pires Gomes, 78
9 Andar
22470 - Sao Paulo - S.P.
Tel: 55-11-3924-8417
TLX: 9911131111 ELBR BR
FAX: 55-11-324-9242

CHILE
Dini Instruments
Sucre 2533
Casilla 6055, Correo 22
Santiago
Tel: 56-2-259-8139
TLX: 44042549 CUCZ

CHINA/HONG KONG
Novel Precision Machinery Co., Ltd.
flat 29, Kingsgate Ind. Bldg.
Phase 1, 29 Kwai Hei Street
N.T., Kwun Tong
Hong Kong
Tel: 852-2532-2422
TLX: 50114 JINDO HK
FAX: 852-2841-601

CO/SALE/111088

INDIA
Micron Devices
Arum Complex
No. 61, D.V.O. Road
Blessing Avenue
Bangalore, 560 004
Tel: 91-80-5272
01-91-80-5271
01-91-80-621-425
FAX: 91-80-5272

MICRONIC DEVICES
Flat 500, Gagan Deep
12, Rajendra Place
New Delhi 110 008
Tel: 91-11-227-7373
FAX: 91-11-227-7373

MICRONIC DEVICES
No. 516 5th Floor
Swati Chambers
Son, Trinamoy Road
Chandd
Bombay 400 071
Tel: 91-22-5555
FAX: 91-22-5555

S&S Corporation
Caman Business Center
Suite 6
6160 Buxton Rd.
San Jose, CA 95124
U.S.A.
Tel: (408) 497-2616
TLX: 802381 DDRY C

JAPAN
Asahi Electronics Co. Ltd.
KanM Bldg. 2-14 Asano
Kakunodate-kita
3-11-10-2, Minato-ku
Tokyo, 115-0001
Tel: 91-3-3541-9991
FAX: 91-3-3541-9991

JAPAN (Cont'd.)
Asahi Electronics Co. Ltd.
KanM Bldg. 2-14 Asano
Kakunodate-kita
3-11-10-2, Minato-ku
Tokyo, 115-0001
Tel: 91-3-3541-9991
FAX: 91-3-3541-9991

MEXICO
Dopexel S.A.
Tocnil 368 Frace.
Ind. San Antonio
Atzcapotzalco
Mexico, D.F.
Tel: 52-5-561-0311
FAX: 52-5-561-0311

VENEZUELA
P. Becerra S.A.
Anteloes & R.R.
Residencia Yanmarita
Locales 4 411
La Campiña, Caracas
Tel: 502-5-561-0311
TLX: 28404 PRIEN VC
FAX: 502-5-561-0311

NEW ZEALAND
Switch Enterprises
36 Olive Road
Penthouse, Auckland
ATTN: Dean Danford
Tel: 64-9-591-255
FAX: 64-9-592-868

SOUTH AFRICA
Electronic Building Elements, Pty. Ltd.
P.O. Box 4609
Pine Square, 1th Street
Hartbeespoort, Pretoria 0001
Tel: 27-27-489-9211
031-677-7810
FAX: 27-27-489-9211

TAIWAN
Micro Electronics Corporation
No. 385, Ming Shen East Rd.
Taichung, R.O.C.
Tel: 886-4-251-8233
FAX: 886-4-251-8233

VENEZUELA
P. Becerra S.A.
Anteloes & R.R.
Residencia Yanmarita
Locales 4 411
La Campiña, Caracas
Tel: 502-5-561-0311
TLX: 28404 PRIEN VC
FAX: 502-5-561-0311

*Field Application Location