Prime

4150 Processor Functional Specification

Steve Small, Editor

PE-T-1440

August 16, 1988

4150 Processor Functional Specification

Steve Small, Editor

PE-T-1440

August 16, 1988

Date: August 16, 1988

To: Prime Engineering

From: Steve Small, Editor

Subject: 4150 Processor Functional Spec

Reference: PE-TI-999 Fox Architecture Document

Abstract

This document defines the functional operation and hardware architecture for the 4150 and 4050 processors. The 4150 is the top of the line office environment machine as well as the entry level computer room machine. Both processors are TTL processors that support Prime's S, R, V, and I mode instruction sets.

Except where specifically noted in the text, all statements made in this document apply to both the 4150 and the 4050 processors.

I'd would like to take this opportunity to thank the writers of this document. Alphabetically:

Denise Chiacchia

Tony Dorohov

Tom Kinahan

Mark Laird

Brian Lefsky

Tom O'Brien

Stu Rae

Sherri Root

Steve Small

Table of Contents

List of Figures

List of Tables

1. System Overview

1.1 Introduction

The 4050 and 4150 are TTL super-minicomputers based on the 9755 pipeline. They are targeted as the high-end of Prime's office product line and as entry level computer room machines, replacing the 2755 and 9755 in the current product line.

These three board CPUs support S, R, V, and I mode addressing, including the latest enhancements such as ASCII8, "C" instructions, and 32IX mode. Additional major features include soft error recovery in the cache and STLB, a fully associative write buffer, a variable speed cooling system, and optional battery backed-up memory. The 4150 is designed to operate with at least Rev 20.2.5 of the PRIMOS operating system, although some features are not available until Rev 21. The 4050 requires at least Rev 20.2.6 or 21.0.2B.

UNIX support is being implemented and will be phased into newly built machines during 1988. The IS and CMI boards initially shipped will not provide the necessary hardware support for UNIX.

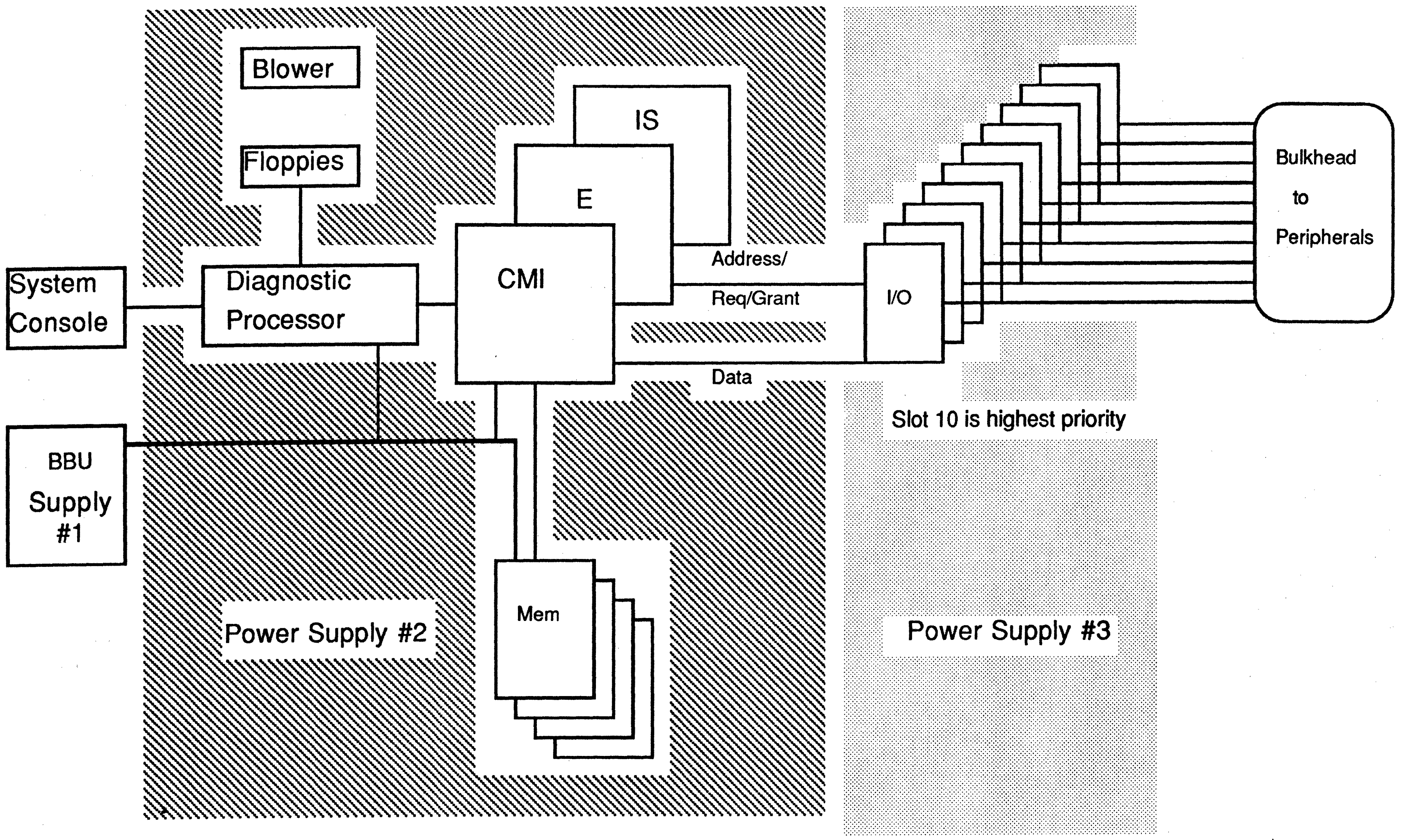
The 4050 and 4150 systems are very similar. They share hardware architecture, much microcode, use the same memory boards and diagnostic processor, and have identical mechanical features. They even share one CPU board. When only one system is mentioned throughout this document, it may be assumed that any comments are relevant for both systems, unless the two are being compared.

The 4050 system has an average performance of 2.7 MIPs, can also support up to 64 MB of main memory, supports up to 10 I/O controllers, and has a sustained I/O bandwidth of 5.8 MB/s.

The 4150 system is has an average performance of 3.6 MIPs, can support up to 64 MB of main memory, supports up to 10 I/O controllers, and has a sustained I/O bandwidth of 6.24 MB/s.

Major new features added to the 9755 design include:

- A 2-way set associative, 128KB cache with soft error recovery
- A 2-way set associative, 1K entry STLB with soft error recovery
- A fully associative write buffer
- New floating point algorithms
- A new diagnostic processor



When the power shunt board is used in place of the BBU, power is shunted from P.S.#2 to the BBU power bus.

Figure 1-1 System Diagram of 4150

- A diagnostic history floppy disk
- Up to 32 MB memory with Abel memory boards (64 MB with Cain boards)
- Battery backed-up main memory
- Lower power
- Over 15 dBA quieter in most environments
- Smaller footprint
- Uses standard 208VAC power (not 3 phase)

1.2 System Packaging

The system is packaged in a minimum of two cabinets. The first (main) cabinet contains the processor, memory, diagnostic processor, and I/O controllers. The main cabinet may optionally contain a battery backed-up power supply for memory protection and/or one Roadrunner (Sr.) for up to 64 asynchronous communications ports. The second and subsequent (peripheral) cabinets contain peripheral devices.

The main cabinet contains a single 22-slot backplane that integrates the diagnostic processor, power supplies, CPU, memory, and I/O. All boards and power supplies are mounted vertically and accessible from the front of the cabinet.

The main cabinet is the same height and depth as a 2755 (low-boy) cabinet, but is wider (24.5"). The cooling system draws air in from the bottom of the cabinet, across the boards, into the blower and out the back of the cabinet at the top. Besides providing a very efficient and relatively quiet cooling system, this design allows cabinets to be abutted at their sides, resulting in an overall floor space requirement equal to or less than that of the 2755 CPU cabinet. In fact, in typical configurations requiring no more than 64 asynchronous communications lines, no more than 1.5 GB of disk storage, and one tape drive, the 4050 or 4150 requires less space than a 2755.

Cables are routed from the front of the cabinet to the back across the top of the cabinet. They are guided by cable troughs and are easily accessible by removing the top panel and dropping the hinged rear panel.

The status panel is located at the top of the front panel. Below the status panel is a pop-open panel which hides the floppy disks (2) and the key switch.

All external connectors are bulkheaded in the rear. All rear bulkheads, a portion of the exhaust duct, and the PDU hinge outward for service. In some configurations, a Roadrunner will be installed on the rear panel. It derives its cooling from exhaust air. Therefore, it is located just below the exhaust port and becomes a part of the exhaust port.

The peripheral cabinet(s) may contain disk drives, tape drives, communications ports, or some combination thereof. Each office peripheral cabinet may be considered to have an upper section and a lower section. Each section can house either 2 FSD disk drives, one streamer or quad density tape drive, or 64 asynchronous communication ports (Roadrunner Sr.). This cabinet is the same as the new 2755 peripheral cabinet. The standard 53" peripheral cabinet is used in the computer room, which has four sections.

Total main bay power dissipation and heat output are higher than those of the 2755. The 4050 and 4150 processors dissipate approximately 70% more power than the 2755 processor, resulting in approximately 135 Watts higher system power dissipation. The remaining power dissipation is configuration dependent and could also be higher due to new controllers and more slots.

Although greater cooling capacity is required, in a typical environment the main bay noise level is lower than that of the 2755. A new, higher efficiency, speed-controlled cooling system is employed to achieve much higher air flow without increasing noise levels.

1.3 Diagnostic Processor

The Mink diagnostic processor provides a superset of the functionality of its 9755 equivalent, the Weasel. New DP functions implemented on the Mink include a high speed parallel interface to reduce control store loading time by at least 5:1, BBU support, analog voltage sensing of 5V supplies, voltage and frequency margining (using additional capabilities in the power supplies and in the clock generator in the processor), and support for the RAS bus. New higher density half-height floppy disks are used, with one for microcode storage and one available as a system log to maintain a record of major system events to support field service.

1.4 System Power

Two Aphrodite power supplies are used. One powers the Mink, CPU, and memory, while the other powers the I/O. Power for the floppy disk and blower is taken from the processor's Aphrodite. Both supplies are required in all system configurations. The Roadrunner, if present, contains its own power supply.

A Daphne power supply with integral BBU can be installed directly into the system backplane for a low cost BBU solution. The supply blocks one memory slot and four I/O slots. In addition, the system can still support 24 MB of main memory with Abel memory boards or the full system maximum of 64 MB with Cain boards. The Daphne is capable of providing extensive ride-through of main memory, the Mink, and the dynamic memory refresh logic in the CPU. When the Daphne is in the system, the last displaced I/O slot must have a shield installed to prevent noise coupling from the power supply into adjacent logic. When the Daphne is not present, a shunt board must be plugged into the Daphne slot to pass power and control signals from the CPU Aphrodite to the logic that would have been powered by the Daphne. All memory and I/O slots are usable with the shunt board in place.

Note that all 10 I/O slots are powered from one Aphrodite power supply, which must be considered in configuring the system in case there are a number of new controllers with excessive power consumption.

The system is configured for 208V 60Hz input power in domestic applications and 240V 50 Hz in international configurations. The internal power supplies and peripherals in domestic systems will be configured for 120V, which the PDU derives from the 208 V input. International systems will have all 240V power supplies and peripheral devices. The maximum operational configuration of the main bay will dissipate about 1800 W.

1.5 Configuration Rules

1.5.1 Memory

1. Only 8 MB Abel, 16 MB Cain, and 32 MB Cain boards may be used.

2. Always begin in slot #1 and do not skip slots unless the 32 MB Cain boards are used.

3. Always skip the next slot when using a 32 MB Cain board. 32 MB Cain boards may be placed only in slots 1 and 3. If there is one 32 MB Cain board, slot 2 may not be used. If there are two 32 MB Cain boards, no other boards may be used.

4. Always put the highest density remaining board in the next legal slot. Thus, exhaust the supply of 32 MB boards before inserting 16 MB boards and exhaust the supply of 16 MB boards before inserting 8 MB boards.

Table 1-1 shows the legal memory configurations.

and control signals from the CPU Aphrodite to the logic that would have been powered by the Daphne. All memory and I/O slots are usable with the shunt board in place.

Note that all 10 I/O slots are powered from one Aphrodite power supply, which must be considered in configuring the system in case there are a number of new controllers with excessive power consumption.

The system is configured for 208V 60Hz input power in domestic applications and 240V 50 Hz in international configurations. The internal power supplies and peripherals in domestic systems will be configured for 120V, which the PDU derives from the 208 V input. International systems will have all 240V power supplies and peripheral devices. The maximum operational configuration of the main bay will dissipate about 1800 W.

1.5 Configuration Rules

1.5.1 Memory

1. Only 8 MB Abel, 16 MB Cain, and 32 MB Cain boards may be used.

2. Always begin in slot #1 and do not skip slots unless the 32 MB Cain boards are used.

3. Always skip the next slot when using a 32 MB Cain board. 32 MB Cain boards may be placed only in slots 1 and 3. If there is one 32 MB Cain board, slot 2 may not be used. If there are two 32 MB Cain boards, no other boards may be used.

4. Always put the highest density remaining board in the next legal slot. Thus, exhaust the supply of 32 MB boards before inserting 16 MB boards and exhaust the supply of 16 MB boards before inserting 8 MB boards.

Table 1-1 shows the legal memory configurations.

TABLE 1-1. Memory Configurations

64 MB Configurations

Slot	1	2	3	4
	32		32	
	32		16	16
	16	16	16	16

56 MB Configurations

Slot	1	2	3	4
	32		16	8
	16	16	16	8

48 MB Configurations

Slot	1	2	3	4	
	32		16		
	16	16	16		
	16	16	8	* 8	(See Note 1)

40 MB Configurations

Slot	1	2	3	4	
	32		8		
	16	16	8		
	16	8	* 8	* 8	(See Note 1)

32 MB Configurations

Slot	1	2	3	4	
	32				
	16	16			
	16	8	* 8		(See Note 1)
	8	8	8	8	

24 MB Configurations

Slot	1	2	3	4
	16	8		
	8	8	8	

16 MB Configurations

Slot	1	2	3	4
	16			
	8	8		

8 MB Configurations

Slot	1	2	3	4
	8			

NOTE 1: 8 MB holes are created in memory where indicated by the asterisks in the configuration tables. These produce some additional Primos memory management overhead and should be avoided when possible.

1.5.2 I/O

The highest priority slot is slot #10, the most distant slot from the processor. The other slots descend in priority from slot 10 down to slot 1.

Power available for I/O controllers is limited to that available from one Aphrodite, 130A +5V, and 7A each from +12V and -12V.

If a Daphne BBU is present in the system, the number of I/O controllers is limited to six. The amount of power available for I/O controllers is not reduced.

1.5.2.1 Required I/O Controller Revision Levels

Certain I/O controllers must be at a specified revision level to work in a 4150 or 4050 system. Table 1-2 shows these requirements.

TABLE 1-2. I/O Controller Required Revision Levels

Board Name	Slang Name	Model Number	Part Number	Revision
IDC3	Koala	6580	TLA10019-001	R
MSTC	Minnow	2382-003	TLA10234-001	R
MPC4		7010T	SPL91521-91	H
STSC	Streamer		2301-901	AA
ASYNC LAC	ICS3	CLAC304	ESA10063-001	C
BMTC	Marlin		2023-001	N
PNC II			2384-001	G

1.5.3 Power

Two Aphrodite power supplies are always required.

Either a shunt board or a Daphne power supply is required.

1.6 9755 and 4050/4150 Comparisons

The 4050 and 4150 differ from the 9755 in technology, partitioning, packaging, and performance characteristics. The six 9755 boards have been compressed into three boards. The three backplanes of the 9755 have been replaced by one backplane that integrates the diagnostic processor, power supplies, CPU, memory, and I/O. Performance variations occur due to the 4050 and 4150's slower cycle times, VLSI enhancements (especially for floating point operations), new I/O microcode, larger set associative cache, larger set associative STLB, larger branch cache, associative write buffer, more efficient memory interface and increased main memory capability.

The 4050 and 4150 boards are all mounted vertically. All 9755 boards were mounted horizontally.

New higher density memory boards (Abel and Cain) support a new processor interface. New slot selection logic suppresses "holes" in memory for certain mixed memory configurations.

Although most 4050 and 4150 microcode is derived from 9755 microcode, substantial editing was performed to at least half of the code. The remaining code was rewritten to take advantage of special new hardware or machine dependent characteristics such as the new VLSI, I/O, and the DP interface. Some microcode fields have been redefined. New microdiagnostics have been written to enhance the coverage afforded by the 9755 diagnostics and cover new hardware.

The Mink DP provides the functionality of its 9755 equivalent, the Weasel, with the addition of a high speed parallel interface to reduce control store loading time by at least 5:1, BBU support, analog voltage sensing of 5V supplies, voltage and frequency margining (using additional capabilities in the power supplies and in the clock generator in the processor), and hardware support for the RAS bus. New higher density half-height floppy disks are used, with one for microcode storage and one available as a system log to maintain a record of major system events to support field service. Note that RAS bus support is being phased into the processor during 1988, requiring a different backplane than that originally shipped in the early systems.

I/O priority net arbitration has been moved into the processor.

The 4050 and 4150 have a larger set associative cache and STLB. Soft error recovery is implemented on both.

The major processor busses were converted to tristate buses. Therefore, special controls to eliminate clashes had to be implemented. Bus timing had to be altered to accommodate these changes.

The write buffer has been made fully associative in the 4050/4150 processors, so that it is no longer necessary to empty the write buffer before performing a main memory read operation. Writes can be merged more effectively so that fewer (and larger) main memory write operations are performed.

The Daphne power supply with integral BBU is available to provide a guaranteed five minutes of memory ride-through on the 4050 and 4150, with fully charged batteries. Due to the very low load placed on the DAPHNE, the actual ride-through capability observed will be much more than five minutes.

The PDA is similar to the 9755 FEP with enhancements such as a deeper stack, higher density displays (including support for the PT200 132 character display), and special support for triggering halts and stack traces from any combination of up to four random signals collected from any CPU board(s).

2. Processor Overview

2.1 General Description

The 4150 and 4050 processors are each implemented on three 16" x 17" boards. They both use "AS" and "ALS" SSI and MSI components, CMOS static RAMs with TTL compatible I/O, Motorola MCA2800ALS Macrocell arrays (ECL internal, TTL external VLSI), and high speed PALs and PROMs.

The "AS" and "ALS" logic families were chosen to allow a power/performance tradeoff with pin-compatible devices, while retaining the highest performance commercial TTL logic family. This has led to an office-installable configuration of a machine(4150) with performance similar to that of a 9755. Separate I/O RAMs (AMD9150) used in the register file and Registered Dual Bus Interface (74AS646) parts helped in dealing with the problem of building machines with a technology usually oriented toward shared buses (TTL), but based on a pipeline and hardware organization designed to take advantage of separate input and output buses common in ECL, the technology in which the 9755 was implemented.

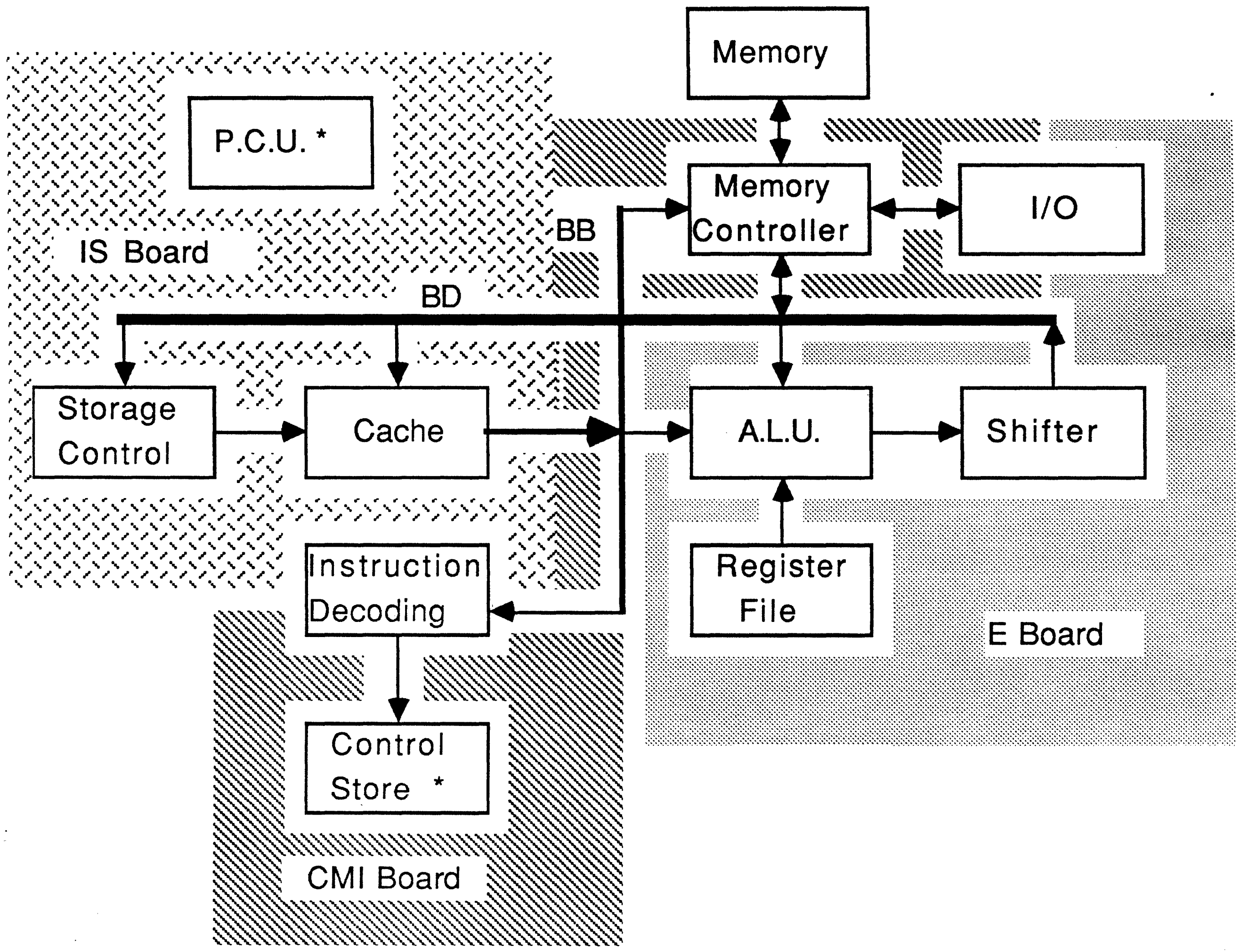
The 4050 processor cycle time is 133.33 nsec. This means that a new single microstep instruction may be executed every 133.33 nsec. The 4150 can do the same thing in 125 nsec.

General performance enhancements include a larger, better organized cache and STLB, an associative write buffer, a faster memory cycle, and new I/O microcode. Standard burst mode performance is 5.8 MB/s for the 4050 and 6.2 MB/s for the 4150. The general mix performance, measured via the GASP benchmark in 32IX mode, is 2.3 MIPS for the 4050 and 3.0 MIPS for the 4150.

Floating point performance has been emphasized, with special hardware and new algorithms added to speed up floating point execution. In 32IX mode, the 4050 execute 3000 single precision Whetstones per second and 2300 double precision Whetstones per second. The 4150 executes 4300 single precision Whetstones per second and 3100 double precision Whetstones per second. The single precision Whetstone number divided by 1000 is often quoted as a floating point performance in MIPS, which would indicate 3.0 MIPS and 4.1 MIPS, respectively, in the environment modeled by Whetstones.

The often quoted average of GASP and single precision Whetstones performance is 2.7 MIPS for the 4050 and 3.6 MIPS for the 4150.

Because of the delays produced by the technology translation at the inputs and outputs of the VLSI parts, it was determined that they would only help with performance in cases when a significant amount of serial logic could be implemented within the chip. This led to fully parallel look-ahead carry logic in the PEALU (ALU VLSI), as well as parallel distribution of all ALU outputs separately to every PBDI (barrel shifter VLSI) part. Most other parts were



* - The C.S. and P.C.U. generate controls to almost all units. Several units return control to the C.S. and P.C.U. to affect future controls.

Figure 2-1 Block Diagram of 4150 Processor

primarily intended to reduce component count, typically replacing from 40 to 85 20-pin DIPs with one VLSI part. Each VLSI part consumes the area of about eight 20-pin DIPs. Much of that area is reserved space around the part to enhance routing, as well as cooling of other nearby parts.

The functional microcode is derived from 9755 microcode about 80% of the time. The remainder of the code has been rewritten to take advantage of special new hardware or machine dependent characteristics such as I/O and the Diagnostic Processor (DP) interface. Increased hardware capabilities have required some changes in the size of the microcode fields. The parity fields have been redefined to help make more control bits available.

The 4050, 4150, and 6350 are highly microcode compatible. Some areas, especially I/O, are very different. In other areas in which differences exist, the code is written to adapt to the target machine. Most code runs identically on all of the machines. The 4050 and 4150 have control stores with 16K words of 80 bits each. All functional code will fit within 8K words. However, the available technology in TTL supported a 16K word RAM more easily.

The microdiagnostics were also derived from those of the 9755. However, significant enhancements have been made to improve testing and to test new functionality. Approximately 35K words of microdiagnostics are in existence today.

The diagnostic processor interface has been redesigned to work with the Mink diagnostic processor.

Standard Prime I/O is supported with a sustained burst mode transfer rate of about 5.8 MB/s on the 4050 and 6.24 MB/s on the 4150. Ten I/O controllers are supported in the fully configured systems.

The priority net arbitration in the I/O bus has been moved into the processor. This allows for faster settling of the priority net as well as the ability to determine which controller was granted access. Slot 10 is the highest priority slot.

The two-way set associative cache with 128K bytes of total data storage and the two-way set associative STLB with 1024 total entries provide improved general mix performance, as well as improved consistency in performance across environments by eliminating susceptibility to two-way thrashing. Three-way thrashing is very uncommon, especially with system software support to help prevent it. Since thrashing cannot be prevented by static allocation when using EPFs, performance variation due to thrashing was a major concern.

A fully associative write buffer reduces the memory access latency, thus reducing the average memory access time. It also improves the memory bus utilization and, therefore, the effective bandwidth.

Major busses are implemented with tri-state interfaces. Special control circuitry is provided to ensure that there is a complete turn-off of one driver before turn-on of the next driver.

Although there are some performance implications, reliability would be negatively impacted if this were not carefully implemented.

A Processor Diagnostic Aid (PDA) is supported to assist with complicated debugging. This device is used in development and manufacturing, and in some cases for engineering assisted field support. It is similar to the 9755 FEP with enhancements such as a deeper stack, higher density displays (including support for the PT200 132 character display), common source code for the 4050, 4150 and 6350, and, in the 4050 and 4150, the ability to halt or trigger the stack on any combination of up to four random signals collected from any CPU board(s).

2.2 Board Descriptions

The E board is a combination of the E1 and E2 boards of the 9755, with some variations. For example, the I/O addressing and priority resolution logic reside on the 4050 and 4150 E boards.

The CMI board includes the control store, the memory controller, and the I/O support (other than that mentioned under the E board). It also contains the DP interface support and master clock generation circuitry. This is roughly equivalent to the CS, MC, and clock boards on the 9755. The decode net logic and other instruction decoding hardware used for microcode sequencing are included as well.

The IS board contains the cache, STLB, PCU, branch cache, and effective addressing hardware. It is roughly equivalent to the I and S boards in the 9755.

The PDA is the rough equivalent of the 9755 FEP.

2.3 VLSI Requirements

The E board uses three Macrocell options and a total of eleven parts. There are seven PEALU parts which implement the major elements of the arithmetic and logical data paths. There are three PBDI barrel shifter parts. These parts provide many special data rearrangement functions, including the ability to determine how to shift data for floating point adjust and normalize operations directly from the data, or from the difference of the exponents. It also supports guard digits and rounding. One PRFADR register file addressing part is used. It contains the E unit copy of RP (program counter), REC (event counter), and skip net support.

The CMI board uses two Macrocell options and a total of four parts. There are two PUSEQ parts which generate microcode addresses, support microcode subroutines, decode instructions to produce microcode entry points, and implement the last stage of the condition testing (jump net) for microcode sequencing. There are two PADBUF parts which implement the associative functions for the write buffer. These parts generate the address for the write buffer, detect read/write collisions, merge data, and independently select and initiate write operations.

The IS board uses three Macrocell options and a total of six parts. There are two PCADR cache addressing parts which include RP, EAS, EAD, and increment and decrement hardware. These chips generate memory addresses. Some support for effective address formation is also included. The PSSS is the STLB set selection part, which detects associative matches, hits and misses on STLB accesses, IOTLB addressing, parity checking, soft parity error support, and selects the correct STLB data. The PCSS is the cache set selection part. It performs similar functions to those of the PSSS, except that they apply to the cache instead of the STLB. It also provides special support for branch instructions.

2.4 Major and Critical Paths

The 4050 master clock period is nominally 33.33 nsec, which results in a "beat rate" (the elementary unit of useful time) of 66.67 nsec. A microcode step takes two beats, or 133.33 nsec. The 4150 master clock period is nominally 31.25 nsec, which results in a "beat rate" of 62.5 nsec and a microstep time of 125 nsec. These compare to 20, 40 and 80 nS, respectively in the 9755.

All path delay calculations have been done to ensure worst case operation of the system at nominal master clock frequency. The worst case times for all logic parts are those quoted in the data books for 50 pF loading. These are specified for 0 to 70 degrees C and 5.0V +/- 10%, except for a few parts that specify 5.0V +/- 5%. Line propagation delays are estimated, usually adding a penalty of 0.5 nS for each input in the net. Clock skews are based on published min/max data for parts in the path and approximately 25% of worst case delay as skew between outputs of the same device. VLSI delays are based on information generated by the vendor's simulation software.

Minimum delays are used only for hold times. In these cases, clock skews are calculated as negative times and line propagation delays are considered to be zero. In these cases, the minimum times specified in data books or 25% of VLSI maximum times are used as minimum delays.

Margin testing is done to ensure that the design provides adequate safety margins to allow for some degradation in component performance over time. Testing at high frequency (34 MHz) and at both low (4.75) and high (5.25) voltage are performed to ensure stability at nominal frequency. The high margin crystal is included in the system and may be selected via the Diagnostic Processor. The power supplies support Diagnostic Processor controllable voltages to perform voltage margining. All systems are tested at margins in manufacturing.

2.5 Processor Physical Description

The shrinking of the 9755 into three board systems required extensive use of high density components such as PALs, octal (and greater) density parts, and the Macrocell arrays. The partitioning was driven by two factors: real estate to implement the required functions, and interconnect limitations between the boards. The insertion of the CPU, DP, memory, and I/O into one backplane and within one chassis required that the processor boards be of substantially similar size. The width of the CPU boards (vertical dimension when inserted in the system) is the same as all of the other boards in the backplane. The depth is slightly different to allow for the difference in the way that the new connectors seat compared to the way the edge connectors seat. Each CPU board has one 488-pin, 4-row high density connector. Power and ground have been distributed through the connectors to provide signal shielding and minimize ground inductance.

Each board is capable of housing the equivalent of approximately 375 20-pin DIPs or equivalent. Spacing is very tight, but 200 mil spacing has been enforced between component holes on any two adjacent components in both directions, except for resistor SIPs. This is done to ensure auto-insertion capability of all unsocketed components, except SIPs, and the sockets for all socketed components, with the possible exception of the VLSI sockets.

Each CPU backplane slot is a dedicated slot and cannot accept any other board (except that the IS and PDA may be exchanged). The power supply pins are identical to prevent disastrous results if the wrong board is plugged into a slot, but some damage could still occur from totem pole drivers that could be placed in contention. The connectors are keyed using keying pins that are installed into the connector before assembly of the board or backplane to help prevent this.

If a CPU board is inserted backwards, the connector will not mate properly with the backplane connector. Labels have been included on the CPU board silk screens indicating which edge should be up when properly inserted. The boards with edge connectors have no such protection, and can easily be inserted incorrectly. Such boards can be damaged if installed incorrectly. This is particularly true of the Mink, since it is oriented opposite to the rest of the boards with edge connectors. Labels on the chassis above and below each slot indicate "component side" of each slot and should be used to properly orient the boards.

All boards and power supplies are inserted vertically. All boards except the E and Mink have their components on the side of the board nearer the I/O power supply. The IS and Mink boards face each other with the Mink in "backwards", and the E and CMI boards face each other with the E board in "backwards".

The cooling towers on the VLSI parts and the modem on the Mink would require a board pitch (spacing) of over 1", totaling about 4.5" for the Mink and CPU boards. With the pairings described above, the total backplane requirement is about 2.75" for the Mink and

CPU. The excessively high components on these boards are arranged in a configuration that allows them to pass each other without conflicts. This permits insertion or withdrawal of any board without requiring any other board to be moved.

When inserting a PDA in the system, the PDA should be inserted before the IS board because of the very close spacing of those boards and the length of the connector pins on the IS boards. These pins frequently contact the housing of the PDA connector if the IS is inserted first.

3. Pipeline Control Unit Functional Overview

3.1 Introduction

The 4150 CPU is implemented as a ten stage synchronous pipeline, capable of handling up to five machine level instructions simultaneously. The time for each stage to complete its operation is termed a "beat". The 4150 beat is 62.5 nsec in duration, while the 4050 beat is 66.67 nsec. Optimally, machine level instructions can enter into and depart from the pipeline every two beats (125 nsec on the 4150, 133.33 nsec on the 4050).

The primary functions of each stage are:

STAGE 1 -	Program counter (IRP) loaded into registers addressing cache, STLB, and branch cache.
STAGE 2 -	Instruction is fetched and loaded into cache data registers (RCD). STLB is accessed for possible use. Branch cache is accessed for possible use.
STAGE 3 -	Instruction opcode is transferred to the control store (CS) and decoding of the instruction begins.
STAGE 4 -	The base and index registers are accessed in the Instruction (I) unit for required effective address information. Decode net address point is calculated in CS.
STAGE 5 -	Effective address calculation in I unit. Control store address (BCY) is generated.
STAGE 6 -	Effective address loaded into STLB and cache address registers. Microcode word loaded into RCM registers on all boards.
STAGE 7 -	Operand data is accessed and loaded into cache data registers. STLB accessed for possible use. Register file source address loaded in execution (E) unit.
STAGE 8 -	Begin execution of operation in E unit.
STAGE 9 -	Complete execution of operation. Register file destination address loaded in E unit.
STAGE 10 -	Write results to register file.

The PCU resides entirely on the IS board, which was designed by Tony Dorohov.

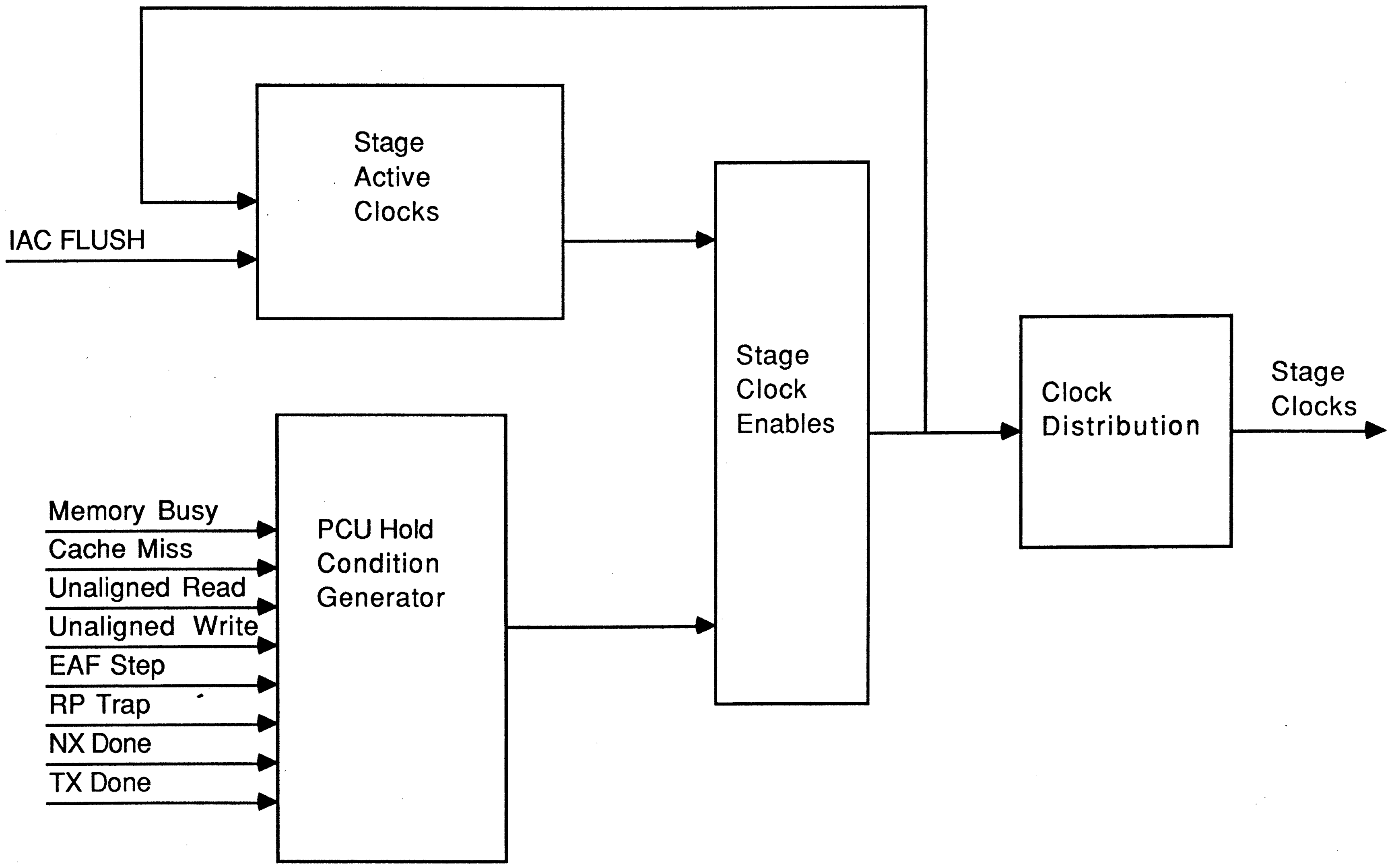


Figure 3-1 Block Diagram of Pipeline Control Unit

3.1.1 Stage Clock Generation

The Pipeline Control Unit (PCU) controls which of the 10 stages are clocked in any given beat. During each beat, the PCU distributes clock enables (ENCS01+ .....ENCS10+) to clock generation circuits on all the processor boards, which in turn generate the associated clock pulse (CS01+...CS10+) on the subsequent beat.

The PCU consists of:

- Registers containing bits denoting which states are currently active (i.e. ST1A+ denoting stage 1 is currently active)
- Combinatorial logic which collects information from all boards which may affect stage clocking.
- Drivers which distribute the stage clock enables to all the CPU boards.

If a stage is active and there are no external conditions received by the PCU indicating that the stage should be delayed, the enable line will be driven active to all the boards activated during that stage.

If a stage is clocked, its active bit in the register is cleared and the active bit for the next stage is set. For example, when stage 1 is clocked at CS01+ the active bit for stage 2 ST2A+ will be set. Refer to Figure 3-2.

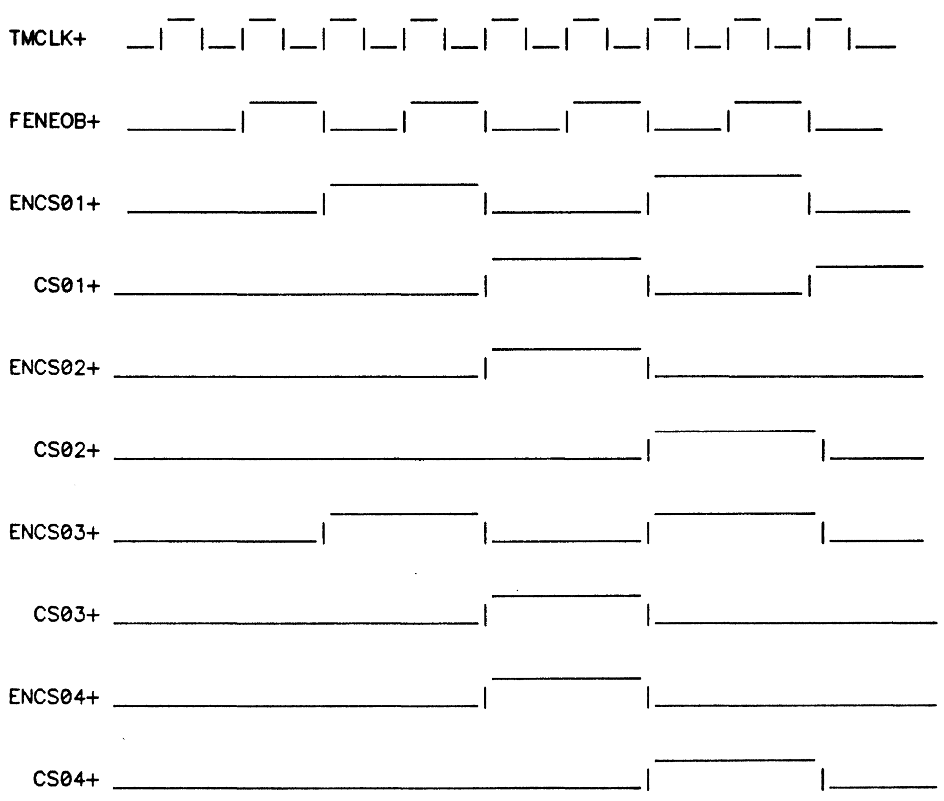
The external conditions that the PCU receives include:

- Cache miss
- Register file collisions
- Fetch cycle traps
- E unit exception conditions

Some of the capabilities of the PCU include:

1. Holding the front end of the pipeline (stages 1-6) while cycling multi-microcode through the E unit.
2. Extending all even stages of the pipeline if extra time is required for a particular stage to complete its operation.
3. Flushing the pipeline following execution of a conditional instruction.
4. Force NOPs into the E unit while instruction refill occurs in the front of the pipeline.

FIG. 3-2. Stage Clocking Example



5. Suspend all pipeline operations during non-overlappable operations such as cache miss.

3.2 9755 and 4150 Comparisons

There are no functional differences between the 9755 and 4150 PCUs.

3.3 VLSI Requirements

No VLSI parts are required in the PCU implementation.

3.4 Major and Critical Paths

The critical path through the PCU starts when a pipeline hold condition is detected, continues through the PCU by shutting off all the ENCSxx signals, and ends when the setup times of these signals are met throughout the machine.

4. Control Store Unit Functional Overview

4.1 Introduction

The major element of the Control Store (CS) is a random access memory in which the system microcode image is stored. The CS allows the overlapping of the execution of the current microinstruction with the fetching of the next microinstruction from the CS RAM.

The CS logic was designed by John Strusienski. It resides on the CMI board, which was designed by Tom O'Brien.

The control store logic may be broken into the following hardware elements:

- Microsequencer
- Control store RAM
- Maintenance & initialization logic
- Parity checkers on the CS RAMs
- BCYPDA bus

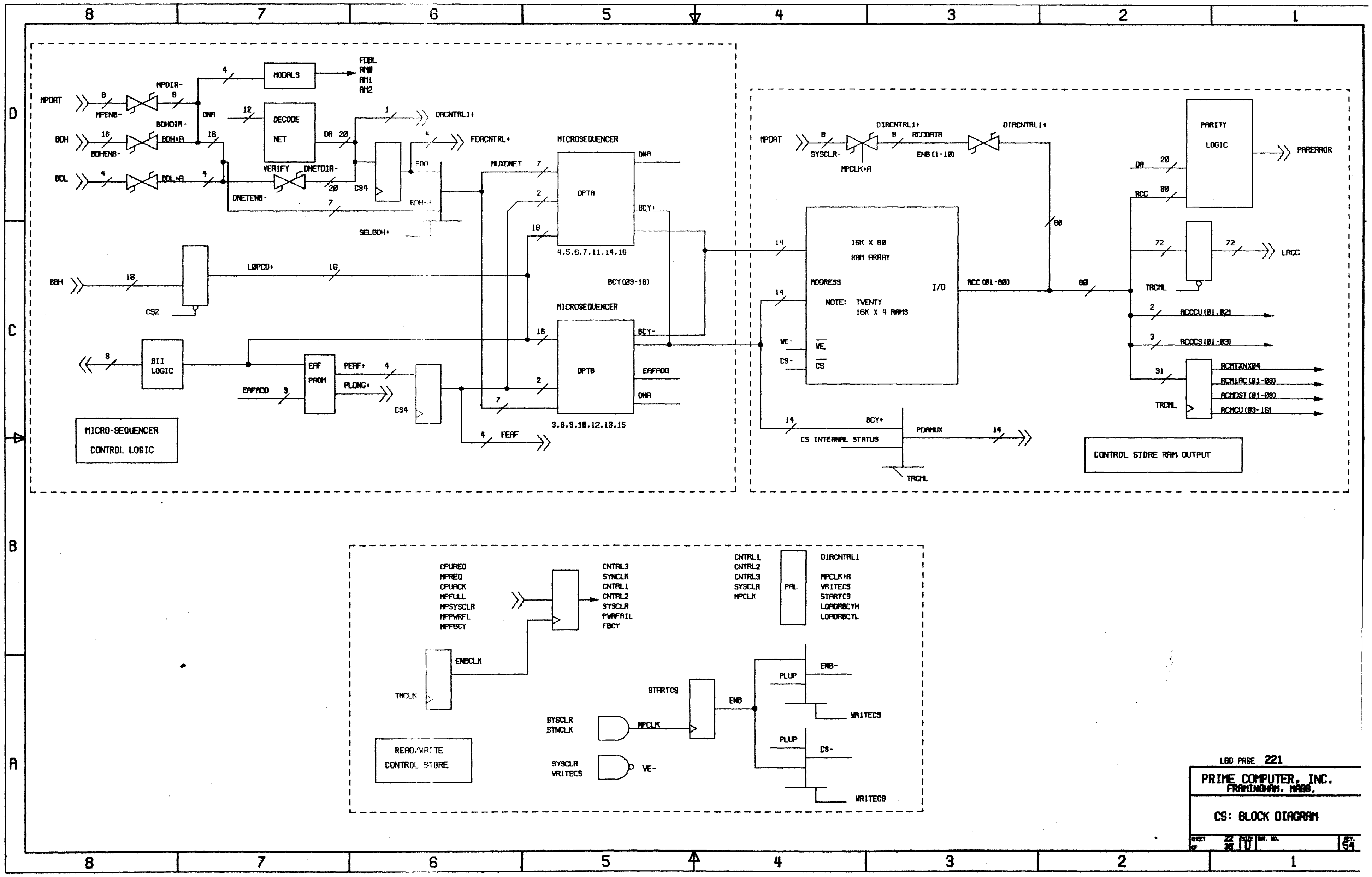


Figure 4-1 Block Diagram of Control Store

4.1.1 Microsequencer

The microsequencer consists of two VLSI chips which produce the control store address. The address bits are referred to as BCY bits. (On an ancient Prime machine, addresses were referred to as Y, leading to the name Bus Control Y. The name has, unfortunately, stuck.)

4.1.2 Control Store RAM

The control store consists of a 16K x 80-bit static RAM array. The output of the array is buffered, latched, or clocked as appropriate, and then sent to all microcode controlled hardware in the machine.

4.1.3 Maintenance & Initialization Logic

The maintenance & initialization logic consists of the interface between the Diagnostic Processor (DP) or Processor Diagnostic Aid (PDA) and the CPU. There are two times when this interface is used:

- When the CPU is halted, either the DP or the PDA may write or read the CS RAMs.
- When the CPU is running, either the DP or the PDA may force the execution of a specific microinstruction. This is known as FORCEBCY.

4.1.4 Parity Checker

The outputs of the control store RAMs (80 bits) include one parity bit for each nine data bits.

4.1.5 BCYPDA

The control store sends a private bus containing BCY information to the PDA. This bus allows the PDA to record the microinstruction stream for display at a later time.

4.2 9755 and 4150 Comparisons

There are no functional differences between the 9755 and 4050 CS units.

4.3 VLSI Requirements

Two microSEQuencer (PUSEQ) VLSI chips are needed to implement the control store logic.

4.4 Major and Critical Paths

The major path through the CS involves receiving a jump condition from another unit, changing the BCYs, and accessing the CS RAMs. This is a two beat path from CS8+ to TRCML+.

Conditional ReTurNs (CRTNs) are critical, but TXNX logic is used to make this path work because of the I unit intervention needed. This is a two beat path from CS8+ to TRCML+.

The internal PUSEQ push/pop stack path is critical, due to the time required from TSTACK+ to TWRITE-.

5. Instruction Decode/Effective Address Formation Functional Overview

5.1 Introduction

The Instruction (I) unit operates during stages 3, 4, 5, and 6 in the pipeline, performing instruction decode and effective address formation.

The I unit resides partially on the IS board, which was designed by Tony Dorohov, and partially on the CMI board, which was designed by Tom O'Brien.

5.1.1 Instruction Decoding

The instruction information is read from cache and clocked into the Cache Set Select (PCSS) VLSI chip at the end of stage 2. The PCSS VLSI transfers the instruction information over Bus B (BB) to the I unit and Control Store (CS) unit for instruction decode during stages 3 and 4.

The PCSS VLSI provides the necessary staging and steering logic to properly format instruction opcodes and displacements for decoding and effective address formation. The cache data registers (RCD) located on the PCSS VLSI are clocked at the end of stage 2 with instruction information and at the end of stage 7 with data information. Under full pipeline operation the PCSS part sources BB every beat with an alternating stream of 32-bit instruction data composed of opcode and displacement, or 32-bit operand data targeted for the Execution (E) unit. Table 5-1 illustrates these actions. It is important to note that the instruction stream data is read during during stage 2 and driven onto BB during stage 3, while operand data is read during stage 7 and driven onto BB during stage 8.

Note that since the displacement is not needed until stage 5 for effective address calculation, the combination of opcode and displacement placed on BB at any odd beat is comprised of the currently prefetched opcode and the last prefetched displacement. The reason for this displacement staging is because the opcode must first be decoded to determine the instruction length and location of the displacement within the instruction.

The decode net is an 8K x 20-bit lookup table addressed by the instruction decoding logic. The output is used by the microsequencer to address the CS RAM. The addresses produced in this manner are the addresses of the microcode entry points for all PMA instructions. The lookup table also contains control information used internally on the I unit.

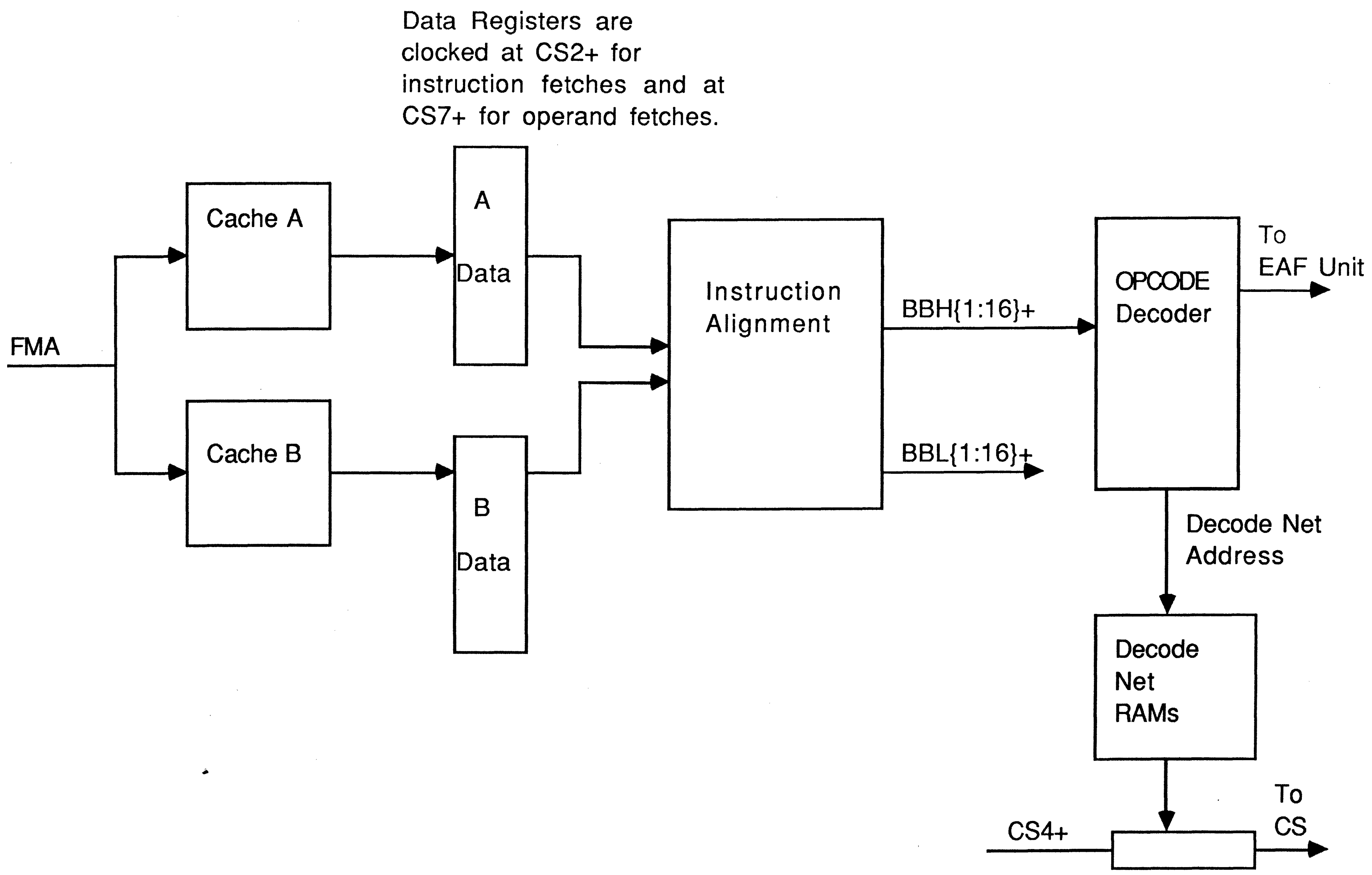
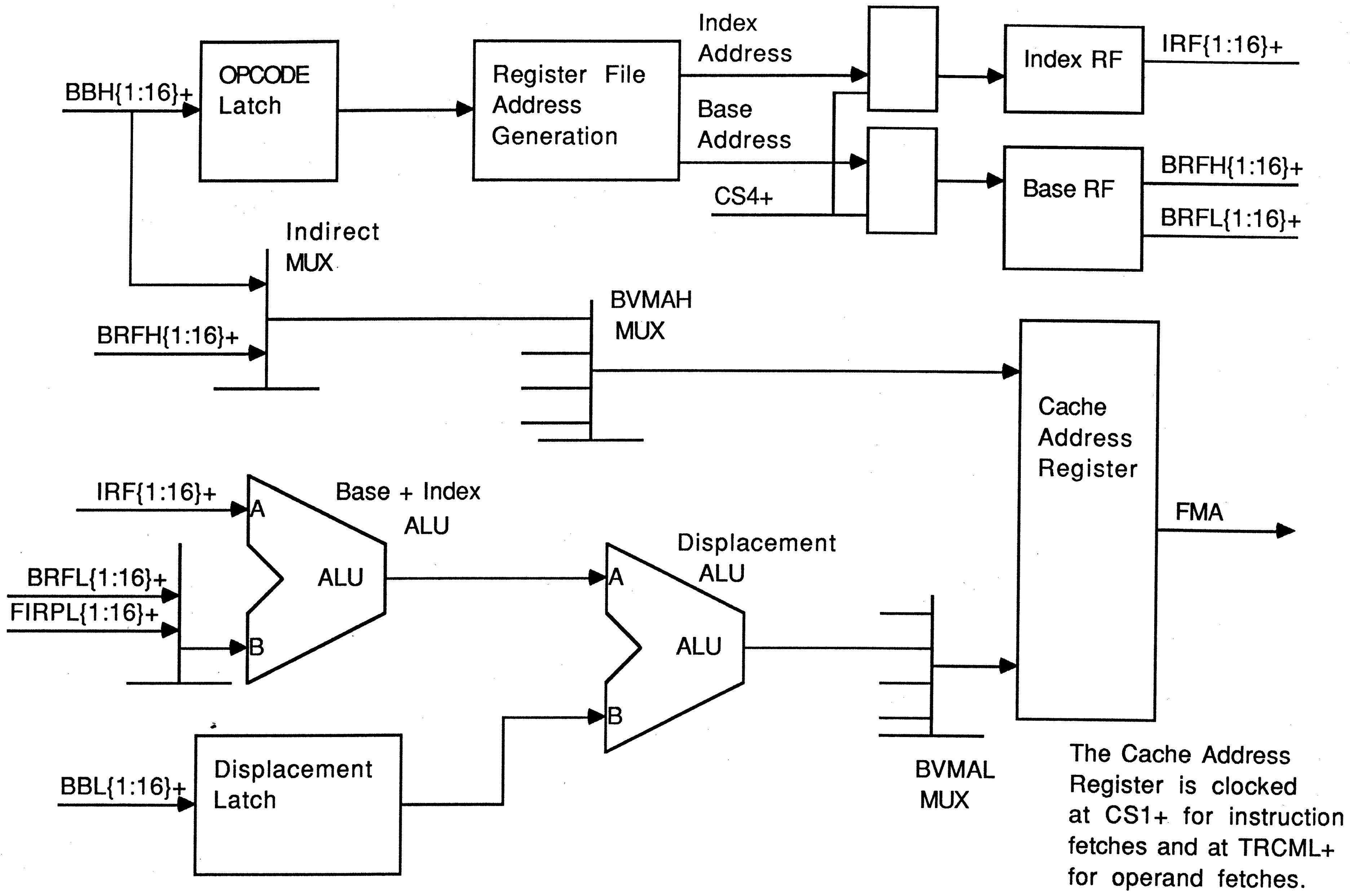


Figure 5-1 Block Diagram of Instruction Decode Unit



The Cache Address Register is clocked at CS1+ for instruction fetches and at TRCML+ for operand fetches.

Figure 5-2 EAF Unit Block Diagram

TABLE 5-1. BB Data During Different Pipeline Stages

INSTRUCTION #	0	1	2	3	4	BBH / BBL
STAGE	5	3	1			OPCODE 1 / DISP 0
	6	4	2			—— ——
	7	5	3	1		OPCODE 2 / DISP 1
	8	6	4	2		OPERAND 0
		7	5	3	1	OPCODE 3 / DISP 2
		8	6	4	2	OPERAND 1

5.1.2 Effective Address Formation

The I unit effective address formation takes place between stage 4 and stage 6. The opcode bits pass through combinational logic to form the base and index register files' read addresses. These addresses are latched during stage 4. The upper 16 bits of the effective address are formed by the contents of the high side of the base register or by the program counter for RP relative operations. The lower 16 bits of the address are formed by taking the contents of the base register or program counter and feeding it thru an index ALU and displacement ALU. The index ALU is used for Base Register (BR) + Index (X) and Register Program counter (RP) + X operations. The displacement ALU is used for BR + DISP, RP + DISP, BR + X + DISP, and RP + X + DISP operations. The calculation takes place during stages 4, 5 and 6, with the final full virtual address being loaded into the registers addressing cache and the STLB at the end of stage 6.

5.1.3 Register File Collisions

A pipelined processor depends on keeping the pipeline full to achieve its performance goals. This means effective addresses for the next instruction must be calculated while the current instruction is still executing. Unfortunately, if the next instruction uses the result of the current one in its effective address calculation, the address will be calculated incorrectly. This condition is called a register file collision. Special logic is implemented as part of the EAF unit to detect and generate a pipeline hold condition, if necessary, when register file collisions occur.

5.2 9755 and 4150 Comparisons

An extra bit has been added to the decode net control bit field to help with the tracking of floating point registers.

An extra address bit has been added to the decode net to double the usuable size to 8K locations. Since the decode net RAMs are 16K deep, no extra logic is needed to utilize the extra 4K except for the one address bit. This address bit is bit 2 of the KEYS register, the Double Precision bit. This extra address bit allows extra microcode entry points for PMA instructions in native UNIX mode.

5.3 VLSI Requirements

The I unit VLSI requirements are:

- 2 PCSS chips, which format the cache data for the I and E units
- 2 PCADR chips, which implement the displacement ALU

5.4 Major and Critical Paths

- Clocking cache data to opcode valid on BBH. The data must be valid before the end of the beat.
- Return to fetch generation to RMA clock with instruction address. This path must be less than 1 beat.
- Register collision detection to holding off EAF address latch enable. This path must be less than 1 beat.

6. Branch Cache Functional Overview

6.1 Introduction

One major goal of a pipelined architecture is to maintain a continuous stream of instructions flowing through the pipe. This insures that the maximum possible performance is achieved. An obvious pitfall of pipelined instruction flow is the branch or jump class of instructions, which alter the normal instruction sequence. A jump or branch will force the machine to "flush", or remove from the pipe, all of the partially processed instructions which follow the branch. Since this pipe flushing degrades performance, the 4150 is equipped with a branch cache, which tries to take advantage of certain known properties of branches and jumps.

Examination of the pipelined execution of a DO loop shows that all but one of the iterations through the loop branch to the top of the loop, and thus flush the pipe. The branch cache logic is intended to avoid this pipe flush by substituting the target address of the branch (in this case, the address of the top of the DO loop) for the address of the next sequential instruction. Since many branch instructions are conditional in nature, the branch cache operations can sometimes be a detriment rather than an aid to machine performance. This problem has led to the development of an algorithm for loading target addresses into the branch cache. In simple English, this algorithm is, "branch the way you branched last time you executed this instruction". This allows the branch cache to maintain a high performance level during DO loop execution as well as during those conditional branches which are rarely taken.

The branch cache functionality is distributed among the entire machine and the microcode. The Instruction (I) unit holds the branch cache itself and controls the variations in instruction sequencing. The Execution (E) unit maintains the true program counter. The Control Store (CS) unit provides an alternate microcode entry point in the case of unaligned branches, and the microcode controls the updating and invalidation of the branch cache.

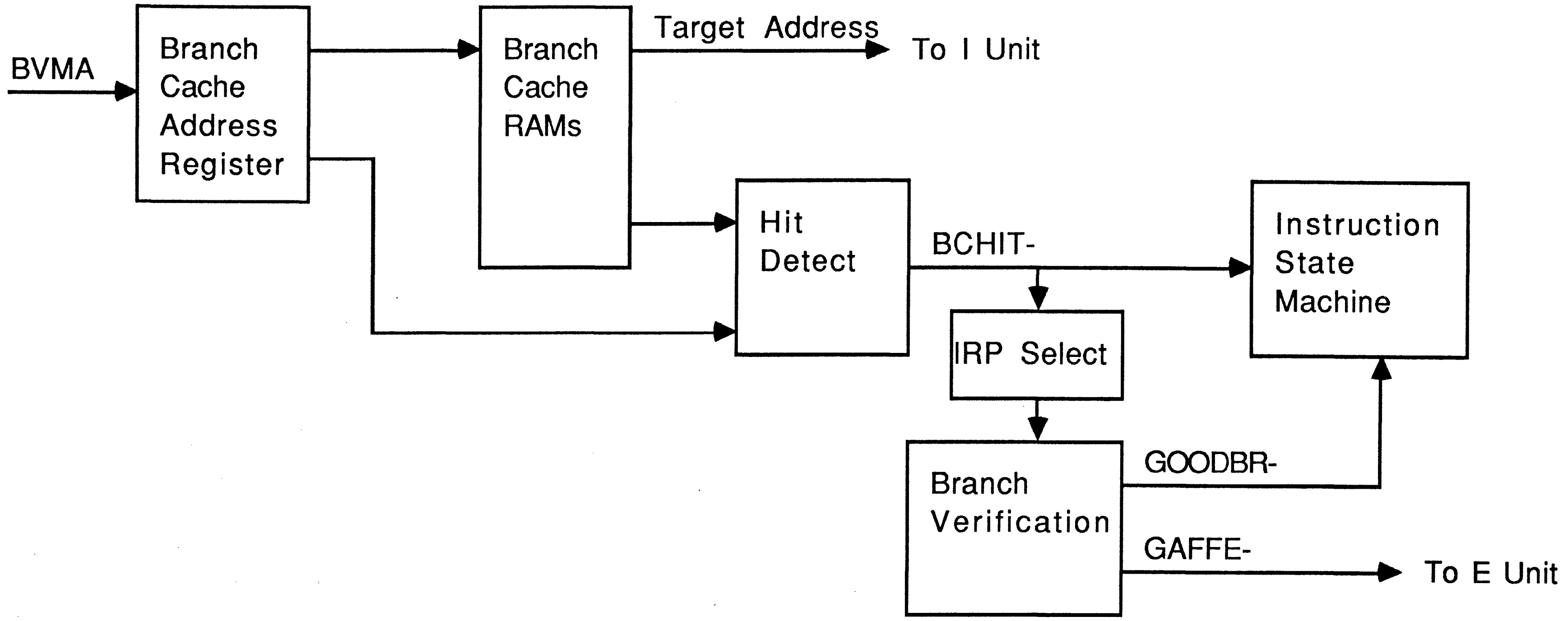


Figure 6-1 Block Diagram of Branch Cache

6.1.1 Overview of Operation

The branch cache logic consists of three main parts:

1. The branch cache array
2. The detection logic
3. The IRP/RP control logic

One simple way to understand the operation of these various parts would be to describe the normal operation of the branch cache.

The branch cache is initially set to be invalid. At some point in the program execution a conditional branch instruction is executed. For the sake of this discussion, we will assume the condition is true. The branch instruction microcode will proceed to update the branch cache with the target address, mark the entry as valid, reload the I unit program counter (IRPL) with the target address, and flush the pipe. This is the slow and painful way to branch, but it only needs to be done once for each DO loop construct. Further encounters with this branch instruction will cause branch cache hits to occur, resulting in IRPL being loaded with the target address rather than IRPL+2, which would occur for a sequential operation.

Now the problem is the case in which the branch condition is false and the next sequential instruction should be executed. In this case the I unit has assumed that the condition is true and loaded the target address into IRPL before the E unit has actually looked at the condition. The front end of the pipeline has been partially filled with bad instructions. Here the hardware and microcode determine that the branch was not valid. The branch cache is invalidated, the correct program counter value obtained from the E unit is loaded into IRPL, and the pipe is flushed. Further executions of this instruction will assume that the condition will not be met.

The actual hit detection is accomplished by comparing the portion of the current IRPL value not used to address the branch cache to the corresponding value that was stored during the branch cache update. If these values match and the valid bit is set, there is a hit. The hit condition will cause the instruction flow to be modified by taking the target address from the branch cache and storing it in IRPL.

Before the branch instruction is permitted to enter the E unit, the branch target must be validated. This is done during the effective address formation of the branch instruction. A full 32-bit comparison is done between the a staged version of the program counter and the target address. The results of this comparison is clocked and sent to the E unit for program counter (RP) control and to be used by the Conditional ReTurN (CRTN) logic.

There are a variety of problems or "Gaffes" which could cause the branch cache to do the wrong thing. For example, a specific location in one segment could contain a valid branch instruction and thus force a branch cache update with good data. A second segment could contain a non-branch instruction in the same relative word location. Execution of this second instruction would cause the first instruction's target address to find its way into IRPL since the branch cache does not check segment numbers or the user ID. Since non-branch microcode is not equipped to deal with changes in instruction flow, a microcode trap handler is used to reload IRPL, invalidate the branch cache, and flush the pipe.

6.2 9755 and 4150 Comparisons

The 9755 branch cache was 256 locations deep. The 4150 branch cache is 1024 locations deep.

The 9755 didn't have the ability to do a full 32-bit branch cache validation. This meant that any branch that could possibly leave the segment was never allowed to have a validated branch cache entry. The 4150 has no such restriction.

A third difference involves a deficiency in the 4150 hardware. The value of IRPL16 inside the Cache Address VLSI chip is not always accurate. This results in improper validation of branches. To overcome this, the branch cache validation signal can only be trusted if a branch actually occurred. The effect of this difference is that the 9755 had the ability to forward branch by one, while the 4150 does not. (This is not a very useful function.)

6.3 VLSI Requirements

The branch cache logic partially controls the pre-IRPL mux selects in the Cache Address VLSI chip (PCADR). One mux select line is controlled by the branch cache hit detection logic, which will force the next update of IRPL to come from the branch cache array rather than the PCADR increment logic used to advance IRPL under normal operation. Thus the substitution of the branch target address for the next sequential IRPL value is done through a mux select line.

The PCADR part also produces the signal which lets the rest of the CPU know that the addresses involved in the current branch operation are valid.

The microsequencer (PUSEQ) VLSI chip on the CS unit has the ability to force decode net address bit 12 to a logic zero on unaligned branches.

6.4 Major and Critical Paths

For the most part there are no critical timing paths in the branch cache logic. The major path from the clocking of the branch cache address register, through branch cache hit detection, and switching the mux to meet the IRPL setup time has 1 1/2 beats to do the job. The branch cache validation path through the PCADR is somewhat critical. Here, the effective address calculation must be compared to the staged IRPL value in two beats.

7. Cache Functional Overview

7.1 Introduction

The 4150 contains a two way, set associative, virtually addressed instruction and data cache. Each set has a data capacity of 16K elements, organized in a 2 x 32-bit arrangement. The total cache data capacity is 128KB. The cache access time is designed to be one beat, so accesses may be made every beat. The cache hit rate is projected at 99%.

The cache resides entirely on the IS board, which was designed by Tony Dorohov.

7.1.1 Basic Cache Operation

There are two sections in each cache entry, a data section and an index section. The index section assists in cache hit/miss decisions. It contains excess address bits (bits which would specify addresses greater than 16K), status bits which relate to the validity of the data in the corresponding location of the data section, and history bits which relate to how old that data is.

The cache hit or miss determination is done in parallel with the data access. The low order address bits are used to address the cache RAMs directly. The excess address bits are compared with the bits in the index section. If all these bits match and if the validity bits are set, the data in the corresponding location of the data section is valid, and a cache hit has been detected. This process goes on simultaneously in both cache sets, and either may detect a hit.

If neither set detects a hit, a cache miss sequence is started. The address which caused the miss undergoes a virtual to physical translation and is sent to the Memory Controller (MC) over BB, along with a cache miss read request. The pipeline is stopped until the MC fetches the data from memory. Any cache miss causes the MC to fetch 64 bits, all of which are written into the cache.

The decision of which set of the cache to write into is made with the help of the history bits in the index sections of the cache. A Least Recently Used (LRU) algorithm is used.

When the data is finally fetched, it is written into registers and placed on BB for 1 beat.

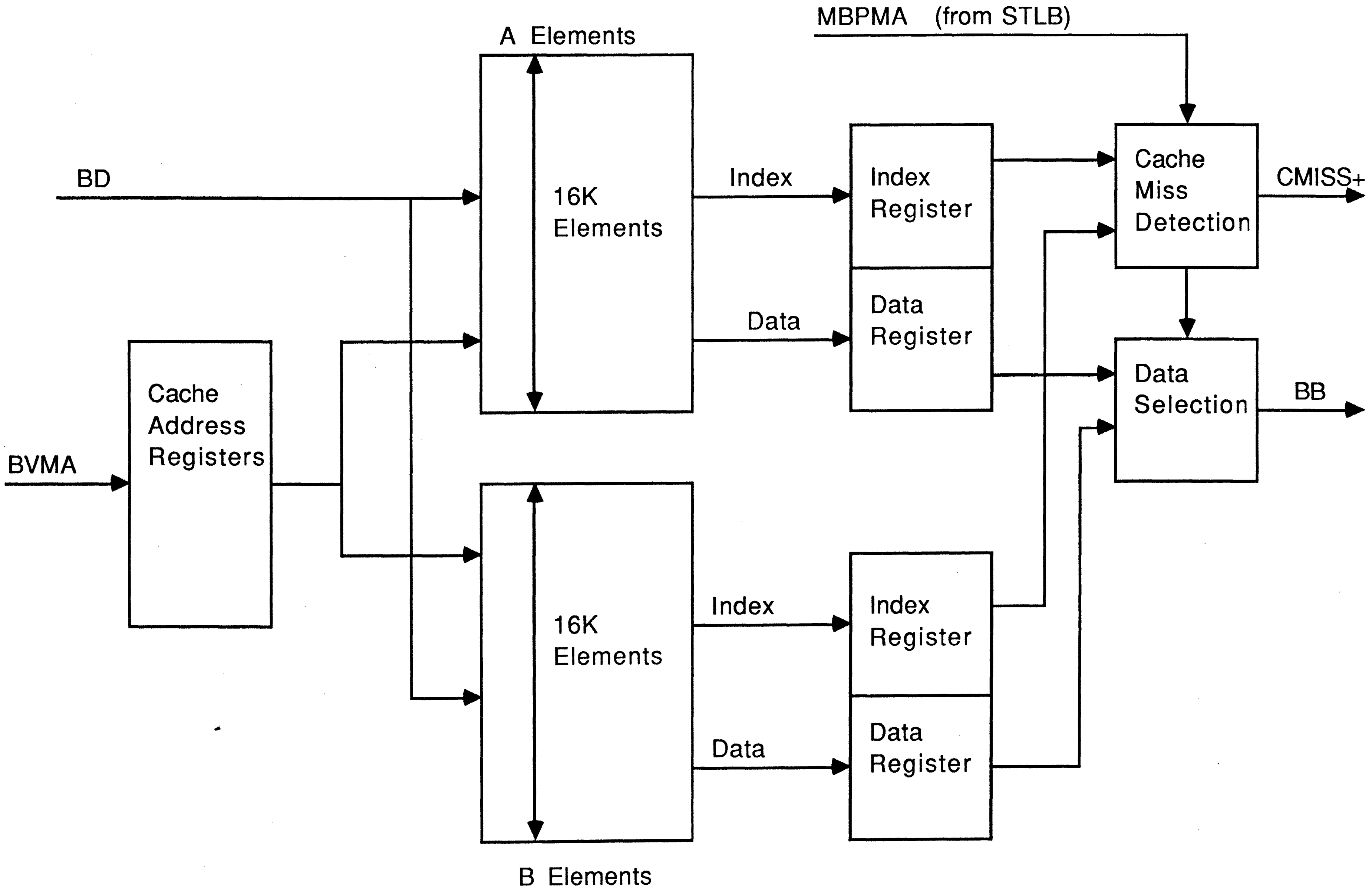


Figure 7-1 Block Diagram of Cache

7.1.2 Cache Writes and Operand Reads

The cache reads the operand data and index during stage 7 of the pipeline and stores the data into registers at the end of the beat. The cache data registers are partitioned into even and odd quantities. Data from cache can be accessed in either 16 or 32-bit quantities. A 32-bit cache read from an odd address is called an unaligned read. Unaligned reads require an extra beat to read the second data word.

Control circuitry writes the cache with data driven on BD by the E unit 1 and 1/2 beats after CS7+. Cache writes can be 16 or 32 bits long. A 32-bit write to an odd address is called an unaligned write. Unaligned writes are handled by a combination of hardware and microcode. Unaligned writes require four extra beats over the aligned case, one to reload the address and three to write the second word. They are handled differently depending on whether the microstep is reloading RMA or not.

7.2 9755 and 4150 Comparisons

The 9755 had a direct mapped, 4K x 32-bit cache. Except for the size, each set of the the 4150 cache is implemented the same as was the 9755's cache. Additional logic controls the interaction between the two sets.

The set associative cache implementation allows two virtual addresses which have identical displacements (low 16-bits) to reside in cache simultaneously. Consider the case where two segments are being used frequently, perhaps by two different subroutines. The likelihood that references will be made to each segment with identical displacements is high. Thrashing occurs when one reference causes another one to be overwritten, only to be overwritten by the original one a short time later. In a direct mapped cache each of these references causes a cache miss sequence. In a set associative cache, the data for each reference can be in opposite cache sets, and no thrashing results. Although 3-way thrashing can occur, it is extremely rare.

7.3 VLSI Requirements

Two Cache Set Select (PCSS) chips are used extensively in the cache implementation. Each chip is responsible for making the hit/miss decision on one of the cache sets. These chips also drive BB with the address during a miss sequence, as well as driving the data onto BB at the end of the access. Finally, the chips also perform parity checking of the cache data.

7.4 Major and Critical Paths

The ability to read cache every beat is the major path through the cache.

Detecting a RP cache miss and subsequently holding off the next odd stage clocks from occurring at the end of the beat.

Detecting an EA cache miss and subsequently holding off the next even stage clocks from occurring at the end of the beat.

Getting data from the E unit to the cache in one beat during memory writes.

Generating the memory trap signal on the S unit and inhibiting any cache misses that may be pending at the end of the beat.

Generating the signals TEBBLCH+ to control the BB latch on the E unit and TCLA+ to control the opcode latch in the decode net. The signal TEBBLCH+ is generated at CS7+ and must be pulled away after the the data is stable on BB and before the end of the beat. The signal TCLA+ is generated at CS2+ and must be pulled away after the opcode is stable on BB but before the end of the beat.

8. Storage Management Unit Functional Overview

The Storage Management (S) unit contains the cache, Segment Translation Lookaside Buffer (STLB), and branch cache functions. The cache is discussed in chapter 7, and the branch cache is discussed in chapter 6. The rest of the S unit contains logic involved in addressing these storage devices. The majority of this logic sits in the Cache Address (PCADR) VLSI option, which sources the S unit internal memory address bus (BVMA). BVMA is the source for all memory related address pointers in the I and S units. These include the cache address registers, the STLB address registers, Register Memory Address (RMA), the backup effective address register (ERMA), and the backup program counter (PRMA). ERMA contains a copy of RMA from the last time it contained an operand address. PRMA contains a copy of RMA from the last time it contained the value of the program counter. The S unit also contains the memory trap logic.

The S unit resides entirely on the IS board, which was designed by Tony Dorohov.

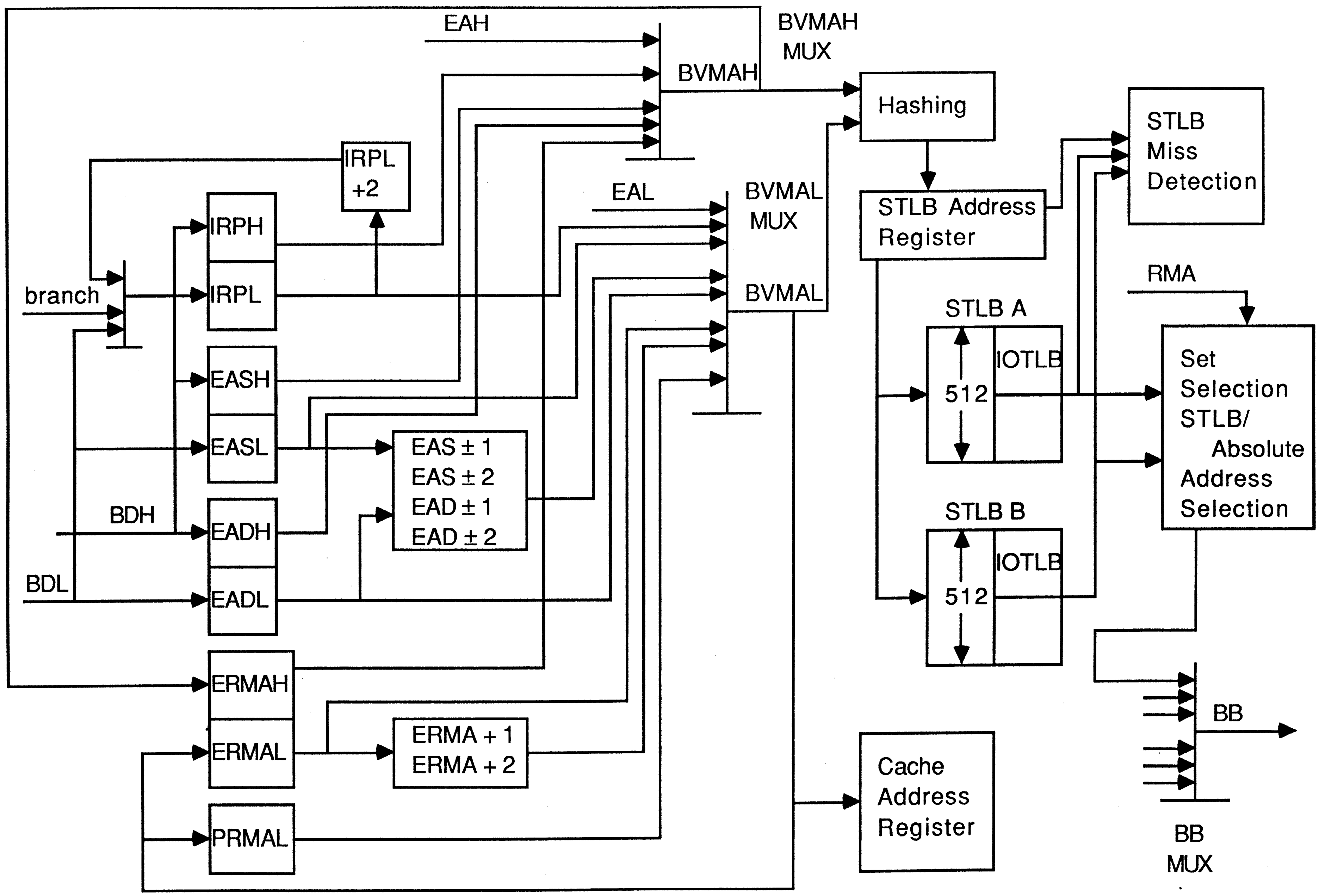


Figure 8-1 Block Diagram of Storage Management Unit

8.1 Virtual Memory Concept

8.1.1 Introduction

The 4150, like all 50 series processors, has a virtual memory system. Virtual memory is a concept where the amount of storage addressable by the programmer exceeds the amount of real memory connected to the CPU. All the virtual storage resides in disk units or other fast-access mass-storage devices and is brought into the real memory only as needed.

The components of a virtual memory system are disks, main memory, and cache.

Cache	The cache is a high speed data buffer between main memory and the pipeline. The cache stores copies of the information contained in the most recently referenced memory locations. During program execution the buffer is used to speed up memory references.
Main memory	The main memory is packaged on printed circuit boards. The 4150 supports up to 64 MB of memory.
Disks	Disks provide storage for all virtual memory. The size of the disk storage will vary from system to system. For example, one peripheral bay with four disks installed could easily provide 3.2 GB of disk storage.

Note, the virtual address specifies a location in the virtual address space. This address may or may not correspond to a location currently loaded in physical memory.

This concept is very similar to a higher order cache. The virtual memory equivalent of a line is a page, with the real memory filling the role of the cache storage. Whenever an address is generated, a check must be run to see whether the particular page is in the real memory or in the mass storage device. The latter case corresponds to a page fault, which is analogous to a miss in cache terminology. In the case of a hit the hardware must determine which real page corresponds to the desired one. This means the virtual memory hardware must convert the virtual address into something that can address a physical memory location (a physical address), and must then search physical memory for that location. PRIMOS employs a series of tables to translate between a virtual address and a physical address (See the System Architecture guide for more details). By manipulating these tables, the relationship between virtual and real addresses can change arbitrarily.

A high speed buffer called the Segment Translation Lookaside Buffer (STLB) contains the virtual-to-physical address mapping for the most recently accessed virtual addresses. The system uses the STLB with the cache to reduce the time needed to access information. If these buffers contain valid information for the process making a reference to a piece of data, the processor can access them in very little time instead of having to make a long memory access.

8.1.2 Segmentation

The virtual memory is divided into units called segments that contain up to 128 KB each. A program and its data sets can be viewed as a collection of linked segments. The links arise from the fact that a program segment may use or "call" another program or data segment Segments are virtual units, not physical ones, that aid the user and the system in organizing their virtual address spaces. The virtual address space of each user contains 4096 segments. These are subdivided into four groups of 1024 each. The segments are subdivided to make address translation and segment sharing easier.

8.1.2.1 Shared and Unshared Segments

In the Prime virtual memory scheme each user address space of 4096 segments is divided into shared and unshared space. The first 2048 segments are shared with all other users. This allows the operating system, shared libraries, and shared subsystems to be seen by all users. The second 2048 segments are private, containing information unique to each user. This means if two users reference segment '4000, they are specifying completely different locations.

This arrangement of shared and unshared segments means that there is no possibility of one user's private space conflicting with that of another user. It also means that one copy of PRIMOS and the shared system software need be maintained, and thus reduces memory use.

8.1.3 Protection Rings

Designating shared and unshared segments is not the only form of protection available to the 50 Series virtual memory. Three hardware implemented rings provide a security system that checks each memory reference for its right to access the specified part of memory.

8.1.3.1 Ring 0

Ring 0 represents the highest level of protection and grants the greatest number of privileges. The kernel of PRIMOS runs under Ring 0 protection, which means that its segments cannot be accessed by the users except through protected entry points, and that it has read, write, and execute privileges to all segments. PRIMOS can access any information in the system.

8.1.3.2 Ring 3

Users run under Ring 3 protection, which means that they cannot arbitrarily access Ring 0 routines or items contained in the private segments of other users' address spaces. Each segment under ring 3 protection may have a different combination of read, write, and execute access rights.

8.1.3.3 Ring 1

Ring 1 provides privileges less powerful than those of Ring 0 but more powerful than those of Ring 3.

8.2 Addressing Cache

8.2.1 Cache Address Sources

8.2.1.1 IRP Register Logic

The Instruction unit Register Program counter (IRP) provides addresses for fetching instructions of the executing program. The contents of IRP are loaded into the cache and STLB address registers at the end of stage 1. IRP is then incremented by two at the next CS2.5+.

8.2.1.2 ERMA and PRMA Logic

ERMA and PRMA are registers used primarily for restoring cache addresses in the event of a cache miss or unaligned read. The PRMA register is used to store instruction addresses while ERMA is used to store operand addresses. When an instruction is fetched from cache the address is simultaneously clocked into RMA, PRMA, and the cache and STLB address registers at the end of stage 1. The copy stored in PRMA is used to restore the miss address during Register Program counter (RP) (instruction stream) cache misses. When an operand is fetched from cache, the address is simultaneously clocked into ERMA and the cache and STLB address registers at the end of stage 6. The address stored in ERMA is used to restore the miss address during EA (data stream) cache misses. ERMA is also used to provide the correct address during unaligned reads and writes.

8.2.1.3 EAS and EAD Registers

The EAS/EAD registers are microcode visible 32-bit registers used primarily in string instruction handling. The EAS/EAD registers are used together for executing string instructions, procedure call, process exchange, and other microcode algorithms where consecutive memory locations are referenced.

8.2.2 Virtual to Physical Address Translation

A two set associative Segment Translation Lookaside Buffer (STLB) is implemented in the 4150. For a discussion of two set associative memories, refer to chapter 7. The STLB has a total of 1024 entries, 512 per set. The STLB provides the most recent virtual-to-physical address translations.

The steps the 4050 takes to convert the virtual address into a physical address are:

1. Check the STLB and the cache. If both contain the correct information, the reference can be completed. If the STLB contains the wrong information go on to the next step (STLB miss routine).

2. Translate the virtual address into a physical address. During the translation, identify if the virtual page containing the information is currently loaded into memory. If it is, load the physical page address (the result of the translation) into the STLB and retry the access (go back to step 1). If main memory does not contain the page, go to the next step (page fault handling routine).

3. Find the correct virtual page on the disk and move it into main memory. The reference is retried after the virtual page is loaded into a physical page.

8.2.3 Memory Traps

Memory traps are breaks in the microcode execution. These breaks may occur due to any one of the following reasons:

- The STLB entry contains the wrong virtual to physical translation (STLB miss)
- A procedure tries to reference a memory location for which it has insufficient access rights (access violation)
- The memory address is in the read address trap range during V, S, and R modes. This signifies that the addressed location is in the current user's register file and not in memory. (address trap)

When a trap occurs the processor saves the current microinstruction address on the microcode stack and goes to the predetermined microinstruction address that handles traps. The processor handles the trap, then returns to the microinstruction address where the trap originally occurred. Detection of memory related traps is performed by the S unit each time the cache is accessed for instruction or operand data.

Traps can be detected for both instruction and operand accesses. For operand accesses, the trap logic is enabled during stage 7 and a trap can be detected by the end of the beat. Should one occur, the PCU is notified and a special pipeline trap sequence is entered. At the end of the trap handler, all addresses will be restored in the S unit and the pipeline will be started up in the same state as when the trap occurred. The S unit trap logic also handles E unit traps which are clocked at the end of stage 7. All E unit traps take precedence over those in the S unit. If no E unit trap is detected, the S unit will send a trap address to the control store and the microcode will handle the trap. Instruction related memory traps are detected during CS2, but are not serviced until all instructions ahead in the pipeline have been executed. Upon detecting an instruction related memory trap, the S unit informs the PCU, which then allows the pipeline to completely empty. Once this occurs, the PCU allows the S unit trap to be processed.

8.2.4 UNIX Support

The 50 series architecture stipulates that 32-bit accesses to the last 16-bit address in a segment wrap around to the beginning of the segment for the second half of the data. UNIX requires that the last 16 bits of such a reference be the first 16 bits of the next segment. New S unit hardware is added in the 4150 to support this functionality, the flat address space hardware.

The flat address space hardware allows the microcode and the hardware to interact in such a way that data that spans a segment boundary can be referenced correctly in UNIX mode. Two new S unit traps have been created, flat trap and wrap trap. Flat trap and the associated microcode trap handler allows reads of 32 bits of cache data across a segment boundary. Wrap trap and the associated microcode trap handler makes EAS and EAD behave like 32-bit up/down counters instead of 16-bit up/down counters on the low side. Both of these traps are enabled only when the machine is in mapped mode, I mode, and bit 2 of the keys is set.

8.3 9755 and 4150 Comparisons

The 4150 does not support direct microcode reads of EAS and EAD registers as the 9755 did. Reads of EAS and EAD are performed by having the microcode first transfer EAS or EAD to RMA and then reading RMA.

The 4150 does not support the operation EAD+1>MA.

The 4150 STLB entries contain extra control bits which the 9755 didn't implement. There are 3 bits for the purge count and 2 bits to help control the 2 set STLB.

The 4150 has 1024 total entries organized as 2 sets with 512 elements each. The 9755 has 1 STLB with 512 entries.

There are no longer separate clocks on the low side of the various address registers (EAS, EAD, RMA, etc.). This is because of pin restrictions on the PCADR VLSI chip. The high side of these registers contain loopback paths, so that when the low side is clocked with a new value, the high side is clocked with itself.

The 4150 includes UNIX support hardware.

8.4 VLSI Requirements

Four VLSI chips of two types are used in the S unit.

- 2 STLB Set Select (PSSS) chips, which perform STLB miss detection, STLB set selection, and memory trap generation. The PSSS chips also contain a copy of RMA. This feature is used to support direct microcode reads of RMA and generation of memory addresses during cache misses.

- 2 Cache ADdRess (PCADR) chips, which implement IRP, EAS, EAD, the ERMA and PRMA registers, ERMAL increment logic, EAS and EAD increment/decrement logic, and the BVMA mux.

8.5 Major and Critical Paths

1. From memory trap detect at CS2+ to holding off increment of IRP at CS2.5+. 1/2 beat path.
2. From clocking IRPL at CS2.5+ to loading cache and STLB address registers 1/2 beat later at CS1+. 1/2 beat path.
3. Clocking cache address register with ERMAL+1 one beat after CS7+ during unaligned reads. 1 beat path.
4. From clocking cache address register at CS1+ (and TRCML+) to reading physical address (physical page number) from STLB and loading into registers on PCSS part at CS2+ (and CS7+). 1 beat.
5. From MPMA mux control generation at TRCML+ to selecting proper data from PSSS chips to clocking that data on PCSS part at CS7+. 1 beat.

9. Execution Unit Functional Overview

9.1 Introduction

The Execution (E) Unit consists of the Arithmetic Logic Units (ALUs), the barrel shifter, and the register file and its addressing logic. Other registers which facilitate data manipulation are also present. The function of the E unit is to read one or more of the sources of data available to it, manipulate this data in the ALU, transport or manipulate the data in the barrel shifter, and load the data into one of the registers on the E, I, or S units, into memory, into the branch cache, or into the register file. The ALUs, barrel shifter, and the register file addressing logic are implemented in VLSI.

The E unit functions are implemented entirely on the E board, which was designed by John Strusienski.

The manipulation of data occurs in two stages, first through the ALU and then through the barrel shifter. The ALU has two busses sent to it, Bus A and Bus B. The main 48-bit ALU is divided into three 16-bit sections. These sections are designated (in decreasing significance) High (H), Low (L), and Extended (E). There is an additional 8-bit ALU for use in multiplies which is designated as Extra (X).

The sources for busses A and B are shown in Table 9-1.

Data manipulation is accomplished by a combination of ALU and barrel shifter modes. The barrel shifter is connected to the ALU output. The ALU may be used in arithmetic mode, logical mode, transport mode, or in combinations of these modes. The barrel shifter can be used to do shifts/rotates on 16, 32, or 48-bit boundaries. The barrel shifter is also used for floating point adjust and normalize operations, greatly enhancing floating point performance over previous machines. Both of these data manipulation entities are entirely under microcode control.

A new microcode instruction (hereafter referred to as I(n)) is read at stage 6 (even). The next odd stage (stage 7) marks the beginning of the execution of this instruction. The register file and/or cache are read during this stage to fetch operands. Execution continues for the next two beats, stages 8 and 9. The pipe alternates between odd and even stages. Because of the odd-even action of the pipe, stage 8 of I(n) and stage 6 of I(n+1) are executed at the same time, where I(n+1) is the next instruction in the microinstruction stream. Similarly, when the execution of I(n) is finished at the end of stage 9, stage 7 of I(n+1) has been executed, and I(n+1) can be processed on the next beat. Results of I(n) are then written to the register file (or other destination) during stage 10 of the the current microinstruction.

From this description, some general observations about reads/writes of the register file can be made. The register file is read during stage 7, an odd pipe stage. Writes to the register file

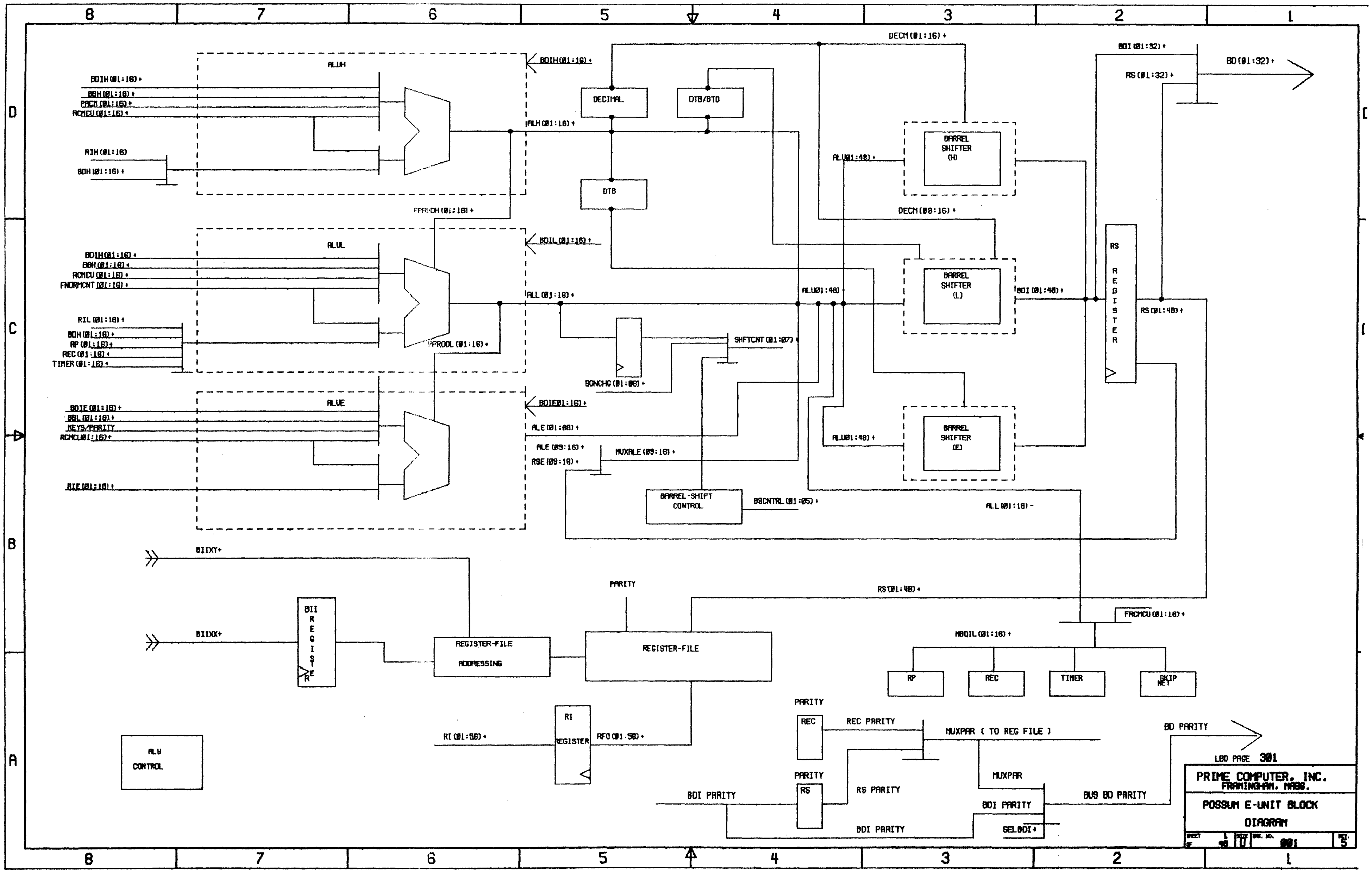


Figure 9-1 Block Diagram of Execution Unit

TABLE 9-1. ALU Data Sources

BUS	SOURCE	COMMENT
A(H)	RIH(1:16)	Register file high side
	BDH(1:16)	Bus D high side
B(H)	BDIH(1:16)	Bus D internal high side
	BBH(1:16)	Bus B high side
	PACK(1:16)	Packer prom output
	RCMCU(1:16)	Microcode word emit field
A(L)	RIL(1:16)	Register file low side
	BDL(1:16)	Bus D low side
	RPREC(1:16)	RP or REC
B(L)	BDIL(1:16)	Bus D internal low side
	BBL(1:16)	Bus B low side
	RCMCU(1:16)	Ucode word emit field
	FNRMCNT(1:16)	Normalize Count
A(E)	RIE(1:16)	Register file extended
B(E)	BDIE(1:16)	Bus D internal extended
	BBL(1:16)	Bus B low side data
	KEYS/PARITY (1:16)	Keys register and Parity information data
	RCMCU(1:16)	Ucode word emit field
A(X)	RI(0:7)	ZEROS
	EMIT(0:7)	FRNDBIT(01:03) Round bits || ZEROS
B(X)	JUNK(0:7)	FGRDBIT(01:03) Guard bits
	BB(0:7)	ZEROS
	EMIT(0:7)	FRNDBIT(01:03) Round bits || ZEROS

occur during stage 10, an even pipe stage. Thus, on alternate stages of the pipe the register file is being either read (odd stages) or written (even stages).

9.2 9755 and 4150 Comparisons

The 4150 E unit contains the functionality of the 9755 E1 and E2 boards combined. It was designed to run at a 62.5 nsec beat rate. The implementation is different in the areas where the VLSI parts are used. Floating point and shift instructions see considerable performance improvement due to the availability of the barrel shifter. Multiply and divide instructions have also been re-implemented with considerable improvement in performance. A 56-bit data path has been implemented through the ALU to support multiply, boosting performance over the old 48-bit path.

9.3 VLSI Requirements

The VLSI requirements for the E unit are:

- 7 Execution ALU (PEALU) chips, each of which contains an 8-bit slice ALU and some of the associated registers and multiply logic.
- 3 Bus Data Interface (PBDI) chips, each of which contains a 16-bit barrel shifter and the associated floating point logic.
- 1 Register File Address (PRFADR) chip, which includes the register file address logic, microsecond timer, as the RP and REC counters.

9.4 Major and Critical Paths

The critical paths on the E unit include:

- Partial product generation during multiply operations
- Partial quotient generation during divide operations
- Generation of jump conditions for the CS
- Rounding operations
- Register file write cycle

10. Memory Controller Unit Functional Overview

10.1 Introduction

The purpose of the 4150's Memory Controller (MC) logic is to act as an interface between the memory and the rest of the CPU. The MC acts as a buffer which accepts data destined for memory, stores the data internally, and then either writes to memory or reads memory and merges new data with existing data and writes this to memory while the CPU continues processing. These writes happen in minimum time from the CPU's viewpoint since it doesn't have to wait for the memory write cycle to complete.

The MC logic resides entirely on the CMI board, which was designed by Tom O'Brien.

10.1.1 Overview of Operation

The MC board controls reading, writing, and refreshing of main memory. It generates Error Correction Code (ECC) check bits on data going to memory and checks the entire data and check word during memory reads. It also checks parity on BD during CPU write operations.

The MC accepts 16 or 32-bit writes from the CPU, stores and/or merges them into a write buffer, and then writes the data to main memory sometime later.

Read operations can be started by microcode request or by cache miss. In either case, the MC always fetches 64 bits from main memory and sends the appropriate word(s) back to the CPU. The requested data is usually available 22 beats after the request was issued.

The Write Buffer (WB) consists of four 64-bit locations addressed on a MOD4 boundary. Each location is further divided into four 16-bit sections. Any combination of 16 or 32-bit writes from the CPU within a MOD4 boundary may be merged into the same WB location. The WB notifies the Memory Timer (MT) of pending memory cycles. If a 64-bit write or an aligned 32-bit write is pending in the WB, the MT does the appropriate write to memory. If only 16, 48, and unaligned 32-bit writes are pending, the MT reads main memory and merges the appropriate data into the WB. This action forces an aligned write to be pending in the WB, and a memory write will follow.

The MT contains a sequencer which executes refresh, memory write, and memory read routines. It starts a particular routine when it sees the request for that routine. For example, it starts a refresh routine when it gets a request from the refresh counter, and starts a memory write routine when it gets a request from the WB. Signals produced by the MT control the WB, the ECC logic, and main memory. ECC bits are generated on every write to memory, and checked on every read. The code used is a single error correcting, double error detecting code. Corrected data is written back into the WB, from which it is written back to memory again, correcting main memory.

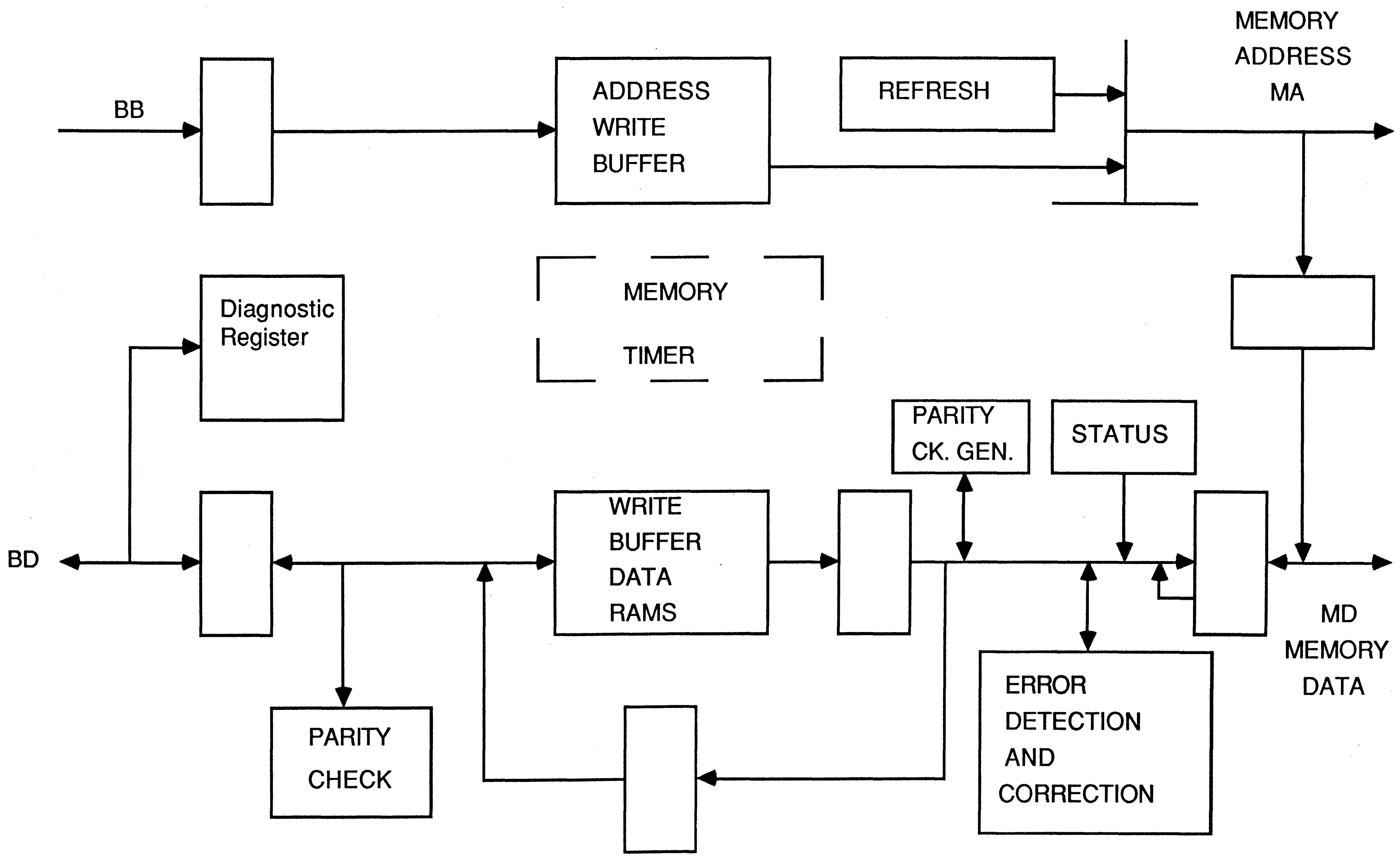


Figure 10-1 Block Diagram of Memory Controller

Main memory is divided into 2 arrays of 39-bits each, 32 data and 7 ECC bits. Memory configurations on the 4150 can vary from 8 to 64 MB. The MC automatically adjusts its memory addressing scheme to interface to any legal memory configuration during system initialization. This adjustment leaves no holes in the physical address space provided all the memory arrays used are the same and that the configuration rules are followed. These rules are discussed in chapter 26.

10.2 9755 and 4150 Comparisons

The MC logic for the 4150 is completely different from that of the 9755. The major differences are:

- The 4150 has one MT, as opposed to the two MTs on the 9755. Each memory word is 64 bits wide (plus check bits) and is broken down into two 32-bit half words. Each half goes to a separate memory board on the 9755, while all bits in the word go to the same board on the 4150. Since each 9755 memory board only contained 32 bits (plus check bits) a separate write or read was required for each 32 bits required, thus a separate MT was required for each half word. Each 4150 memory board contains 64 data bits. Separate MTs are, therefore, no longer needed since all data bits are written to the one board.

- In the 9755 a memory array read could only occur when the WB was empty. This insured that the CPU was not getting stale data from memory. This is not necessary on the 4150. The MT can replace stale data from memory with the fresh data from the WB "on the fly". This results in much faster throughput, since there is no time wasted in emptying the WB whenever a read is requested.

- The 9755 data bus consists of two 32-bit (plus check bits) data buses. The 4150 data bus consists of one 32 bit (plus check bits) data bus. The data is sent over to the memory board 32 bits at a time, latched on the memory array, and written into the array either 32 bits or 64 bits at a time depending on the operation specified.

- A completely new memory array board is used with the 4150, employing 256K (8 MB boards) or 1 MB (32 MB boards) DRAMs and supporting the new memory bus design.

- The ECC logic is completely different between the two processors. The 4150 utilizes an ECC chip to check/correct data, while discrete logic with a different code was used on the 9755.

- The MC logic on the 4150 contains DMx support since the I/O logic is tied very closely to the Memory Controller. This allows I/O data transfers to take place between the I/O interface and the MC without the use of BD.

- There are 12 refresh address bits on the 4150 while there are 8 bits on the 9755. This is to accommodate the larger RAM size on the memory array boards and to provide extra bits for future expansion if needed.

- The 4150 has Battery Back Up (BBU) capability. The MC provides refresh to the memory array boards in the case of a short term power failure to maintain the integrity of data in memory.

10.3 VLSI Requirements

The MC uses two ADdress BUFfer VLSI chips (PADBUF) in a bit-slice fashion. Each slice processes half of the memory address bits.

10.4 Major and Critical Paths

The critical paths are as follows:

- The path from the time the memory address is latched into the MC until the WB pointer is latched during a CPU memory access must be less than 2.5 beats.
- Setting MBSY- on the MC unit on a READ or WRITE and sending it to the S unit is a one beat path.

The major paths are as follows:

- Bringing data back from memory into the MC and going through an error correction routine if a correctable error (ECCC) is detected adds 3 beats to the memory read routine.

11. I/O Interface Functional Overview

11.1 Introduction

The main function of the I/O logic is to transfer data between memory and a peripheral controller as rapidly and efficiently as possible.

The I/O data interface resides on the CMI board, which was designed by Tom O'Brien. The I/O address interface resides on the E board, which was designed by John Strusienski. The control interface is split between the two boards.

11.1.1 Overview of Operation

Direct Memory transfer (DMx) operations are started by a peripheral controller asserting a request after having been set up by PIO instructions. This causes the microcode to trap out of its current operation to service the I/O request. Arbitration is done among multiple requesting controllers, and one controller is granted the bus to do its transfers.

A typical I/O transfer consists of an address phase and a data phase. In the address phase, the I/O controller which has won control sends information to the CPU about what kind of transfer is required and where the data for the transfer is to be written (or read from). During the data phase the data is actually transferred. Another address phase is performed, overlapping the data phase, to see if any other controllers need service. When all controllers have been serviced the microcode returns to what it was doing.

The controllers' request lines and other control signals make up Bus Peripheral Control (BPC). Address information sent by the controller during an address phase is received on Bus Peripheral Address (BPA). Data is actually exchanged over Bus Peripheral Data (BPD) during the data phase.

The main data path is very straightforward. On an output transfer, data is read from memory and clocked into the BPD data register under control of the Memory Timer (MT). Data is then driven onto BPD under microcode control. There is no need to use the system BD bus. On input transfers data is clocked into the BPD data register from BPD under microcode control, with no need to use the system BD bus. From there, the MT writes it directly to the Write Buffer (WB).

In burst mode transfers, a single address phase is followed by four consecutive data phases without any additional address phases. These burst mode transfers may be requested at any time, but will only be honored by the CPU if the address and the amount of data to be transferred meet the proper requirements.

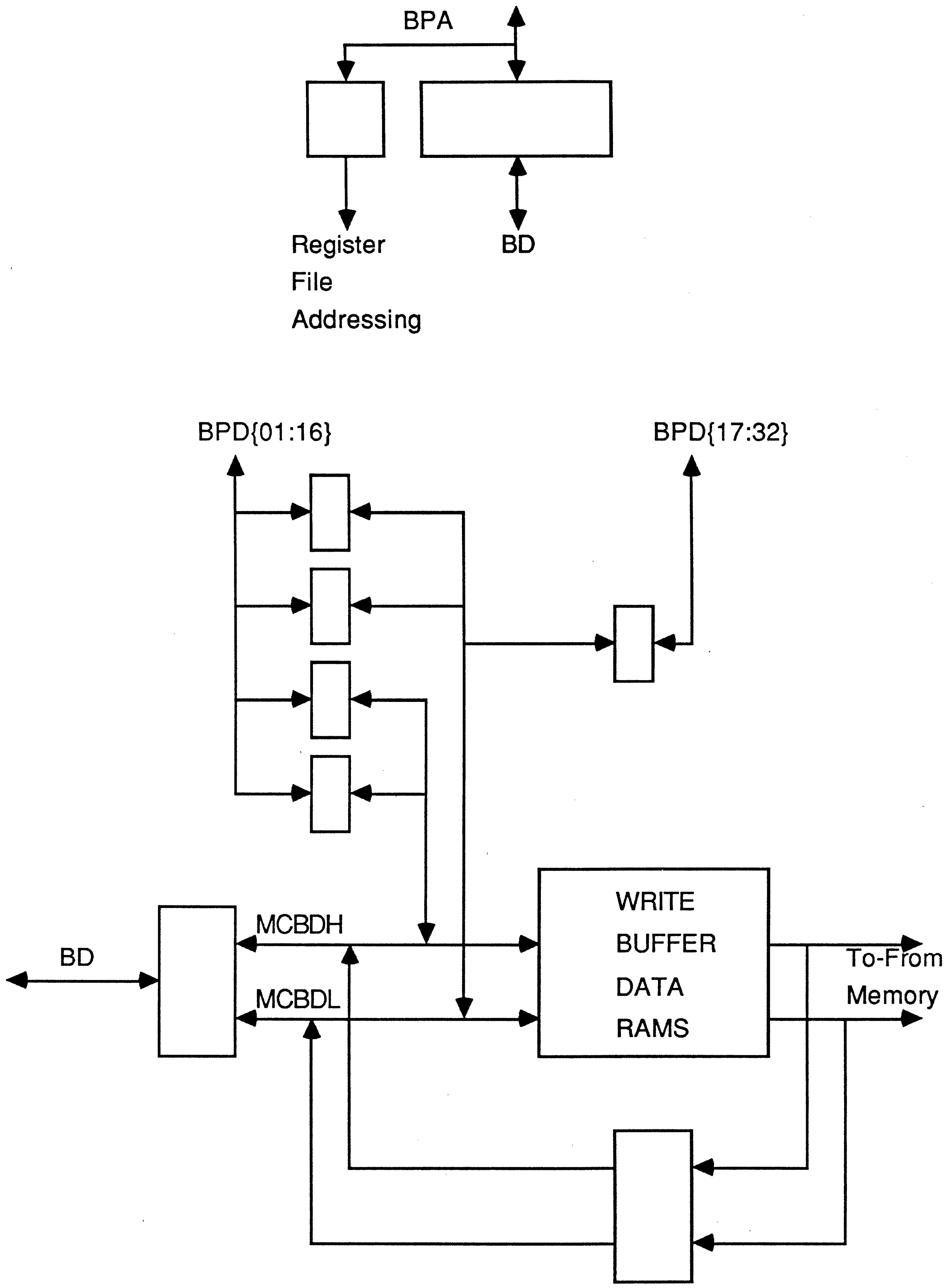


Figure 11-1 Block Diagram of I/O Interface

11.2 9755 and 4150 Comparisons

The control and data paths for the 4150 will be closely tied into the memory controller logic, and so will be unlike that of the 9755 or any other existing machines. Data goes to (or from) memory on an internal MC bus, so it is not necessary to place I/O data on the system bus for I/O transfers.

11.2.1 Burst DMT Mode

This new mode transfers four words for each address phase instead of the normal one word in regular DMT operations. This mode is to DMT as burst DMA is to normal DMA.

11.3 VLSI Requirements

There are no VLSI chips required for any I/O processing. However, memory will be addressed in the normal manner, which means that the address will be stored in the PADBUF VLSIs in the MC.

11.4 Major and Critical Paths

Data going to (or from) the BPD drivers utilizes the internal MC data bus, so the I/O data timing is dependent on the MC timing.

During a burst transfer, 64 bits are written into the WB data RAMs in the time it normally takes to write 32 bits. There is critical timing in this operation which involves turning off one set of BPD drivers after the first 32 bits have been written and getting the next set of drivers turned on to allow the second 32 bits to be written 3 TMCLKs (93.75 ns) later.

12. System Busses Functional Overview

12.1 Introduction

The major system busses are Bus B (BB), Bus D (BD), Bus Virtual Memory Address (BVMA), Bus Peripheral Address (BPA), Bus Peripheral Data (BPD), the Memory Address bus (MA), and the Memory Data bus (MD). They are briefly described below.

Busses connect different boards together, and are therefore implemented in the backplane. The 4150 backplane was designed by John Strusienski.

12.1.1 BB

BB can only be driven by the S unit. It is most frequently used for transferring cache data to the E unit (operands) or CS (instruction stream). It also drives the physical address to the MC on memory references. BB consists of 32 data bits plus 4 parity bits. It is visible to the PDA.

12.1.2 BD

BD can be driven by the following units:

- E
- MC
- CS
- S
- PDA
- I/O Interface

It is most frequently used to transport E unit results to I or S unit destinations or memory. It is also used for transporting data from memory to cache during cache misses, as well from the Diagnostic Processor (DP) to the CPU. BD consists of 32 data bits plus 4 parity bits. It is visible to the PDA.

12.1.3 BVMA

BVMA is driven only by the S unit. It is used to send the virtual memory address for a current memory access to the cache, STLB, and branch cache. BVMA consists of 32 address bits. A copy of it is visible to the PDA.

12.1.4 BPA, BPD

BPA and BPD are used for transferring address and data information, respectively, between the CPU and the I/O controllers. BPA consists of 18 address bits plus 2 parity bits. BPD consists of 32 data bits plus 4 parity bits. These busses are not visible to the PDA.

12.1.5 MA, MD

Busses Memory Address (MA) and Memory Data (MD) are used to address main memory and transfer data to and from memory, respectively. MA consists of 25 address bits plus four parity bits. MD consists of 32 data bits plus 7 check bits. These busses are not visible to the PDA.

12.1.6 BD Arbitration

Bus BD is the only bus in the system that can be driven by multiple sources on different processor boards. Skew problems and the need to avoid delays in bus arbitration to avoid tristate clashes made it necessary to keep the bus arbitration hardware on one board. The ALEG microcode field is central to the bus arbitration decision since it can specify BD as an ALU source during any microcode step. The E unit is the only unit which looks at this field, and therefore has the additional task of performing BD arbitration.

The E unit normally has control of BD. Control of the bus has to be relinquished in case of cache misses, PDA Force BCYs, I/O transfer address phases, and explicit microcode controlled memory reads. When a change of BD control is about to happen, the following sequence occurs:

1. The current unit in control is disabled.
2. One half beat later, the new unit to be in control is enabled.

This scheme provides a half beat period between any two units driving BD to avoid tristate clashes.

12.2 9755 and 4150 Comparisons

- 9755 busses used wired-OR technology. 4150 busses use tristate technology.
- BPA, BPD, MA, MD are completely contained in backplane on the 4150. These busses were connected from one backplane to another with cabling on the 9755.
- BA (the E unit ALU A leg input bus) is no longer an inter-board bus in the 4150.

12.3 VLSI Requirements

BB is always driven directly by two PCSS chips. MA is driven by the buffered outputs of two PADBUF chips. BD is usually driven by the buffered outputs of PBDI chips.

12.4 Major and Critical Paths

The major path in this section of the machine is the BD arbitration path. This path includes detecting a need to pass control of BD to a new source, as well as actually performing the change in time to allow the new source to get the data to its destination without causing a tristate clash.)

13. Processor Diagnostic Aid Functional Overview

13.1 Introduction

13.1.1 General

The 4150 Processor Diagnostic Aid (PDA) is functionally equivalent to a Field Engineering Panel (FEP) on 9755 based machines. The functions and operation of the PDA as implemented on the 4150 processor will be described in this chapter.

All of the PDA's functions are implemented on the PDA board, which was designed by Tom Kinahan.

The purpose of the PDA is to aid in the debug of the 4150 processor by sampling and displaying the operation of the CPU for the operator. It samples the CPU's major busses and clock signals and displays them. The operator can control data sampling with a number of commands similar to those implemented on commercial logic analyzers. The PDA incorporates a Z80 microprocessor to implement many of these functions. All of these capabilities make the PDA an important debugging tool.

The majority of the PDA is dedicated to a large (1K by 160 bits) stack. The stack samples CPU busses and stores them for an operator to examine later. The 4150 PDA samples all the stage clocks, control store addresses (BCYs), hold conditions, BDL, BDH, BBL, BBH, BVMA, PCU status, I/O priority, write buffer pointer bits, 4 logic analyzer bits, memory data valid, memory busy, return to fetch level, INIT+, and machine check.

The PDA can load, read, and modify the 16K x 80-bit control store by using a bus that is common to the Diagnostic Processor, PDA, and control store. Current software restrictions limit access to the lower 8K of control store.

The PDA can halt the CPU on various conditions such as Machine Check or sense register matches.

The PDA can FORCEBCY, which means that it can make the CPU begin executing microcode at a specified address. Using this FORCEBCY functionality, the PDA can work with the system's microcode to load main memory and/or the decode net.

The PDA works in a basic configuration called host mode. Host mode requires a host CPU running PRIMOS. The operator logs onto the host and uses an assignable AMLC line to give commands to the PDA installed in the test system. A self diagnostic mode is available to debug the PDA. It allows access to the Z80 memory and to diagnostic routines via a Mink Diagnostic Processor.

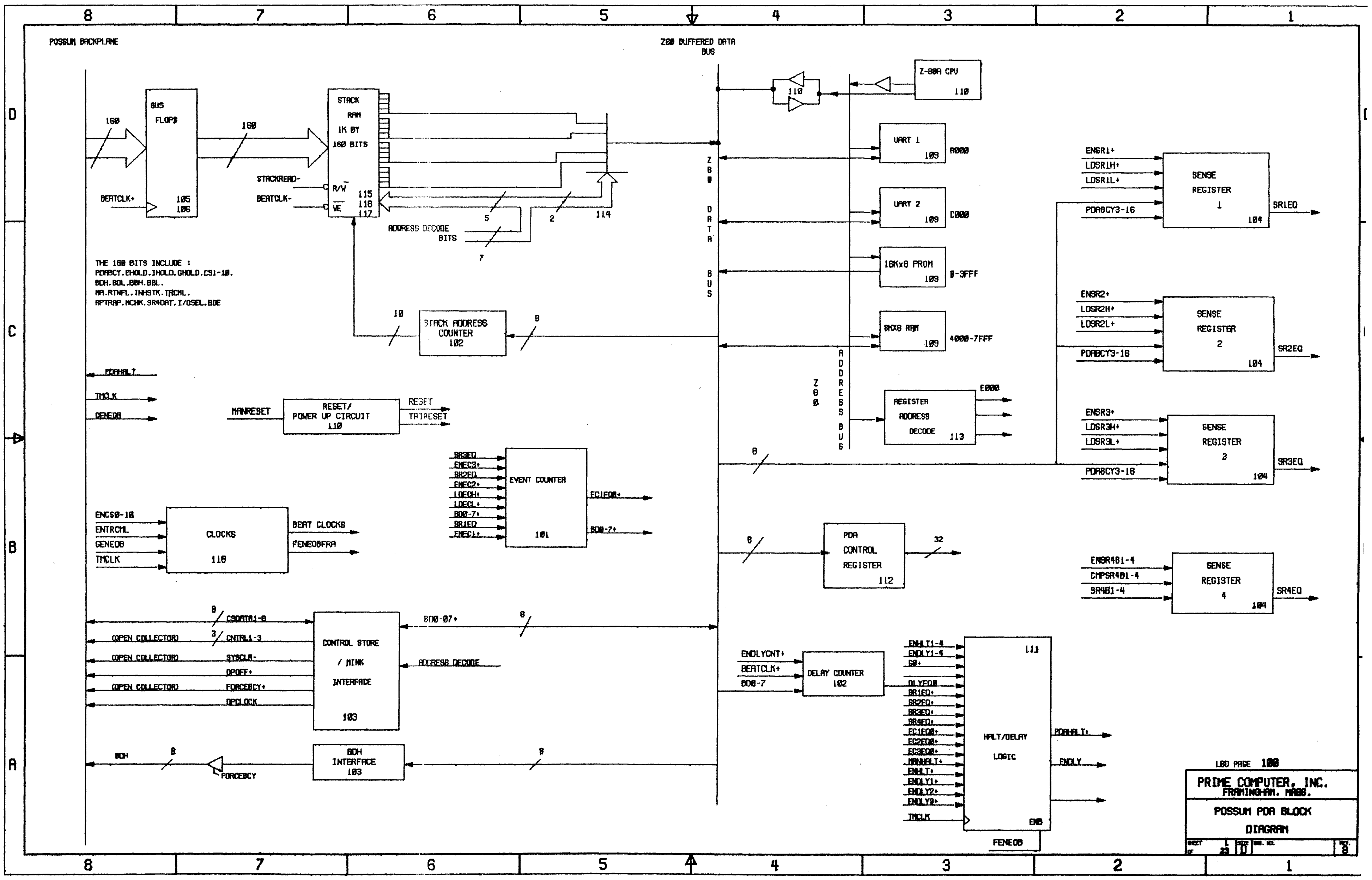


Figure 13-1 Block Diagram of PDA

Host mode allows access to the microcode database, is able to load control store, and can display the microcode labels in a stack dump. Host mode allows access to microcode address labels, which can be used with the sense register commands, update control store (UC) command, and the GO command. The host CPU also has access to source listings which can be displayed at the user's terminal without leaving the PDA command environment. Stack dumps can be saved and spooled for future reference, and abbrevs can be used.

The PDA does NOT detract from the performance of the CPU while installed. It does **NOT** steal cycles from the CPU. Its major function is stack storage, and as such cannot impair the behavior of the processor. Having a PDA installed in a system will not affect system behavior. It **CAN** do evil things to the CPU when the operator is unskilled in its use. Examples of this would be modifying control store while the machine is running, or causing the machine to halt accidently.

13.1.2 Z80 Microprocessor

A Z80 microprocessor is used in this implementation. It was chosen due to its ease of use and because there were large amounts of code written for it that could be easily adapted to the 4150 PDA.

13.1.3 Stack

The stack on the 4150 PDA is 1K x 160 bits. This allows the operator to see almost anything in the CPU s/he might want.

13.1.4 Halts / Delays

The operator has the option of halting or triggering on sense register matches. Sense registers have parameters that allow the operator to halt the CPU or trigger the delay counter depending on what s/he wishes.

13.1.5 Delay Counter

The delay counter can be set to a value such that the stack will not trigger until that number of beats after the sense register match. There are various methods of using the delay counter such as pre-triggering (when you want to see the trigger label at the top of the stack), post-triggering (when you want to see the trigger label at the bottom of the stack) and center triggering (when you want the trigger label to be somewhere between the top and bottom). The type of triggering can be controlled by changing the value of the delay counter. A delay of 1 gives post-triggering while a delay of 1024 gives pre-triggering. The delay counter is a 16-bit counter and can have any value between 1 and 65535 inclusive. A value larger than 1024 results in the trigger point not appearing in the stack.

13.1.6 Sense Registers

Sense registers are multi-bit comparators that trigger the delay counter. When the delay counter reaches zero, the stack triggers (stops the storing of additional data) or PDAHALT- is asserted. As an example, the operator could set a sense register match on SR1 to a microcode abel of START (octal 200). If the microsequencer were to issue the address of START to the control store, the sense register would become active and the proper action would be taken. In this case, it would stop the stack from storing additional data, also known as triggering. The action of triggering the stack allows the operator to capture the events leading up to and/or some known number of events after the sense register match.

13.1.7 Forcing Microcode Address

The operator can FORCEBCY. This is the ability to force microcode execution to begin at any desired address. The microcode assist package must be loaded into the control store for this to operate properly.

13.1.8 Reading/Writing Control Store

The 4150 PDA can read and write control store. This will only work when the PDA is in host mode. This is due to the fact that the code running on the host system will tell the PDA what to load into the control store under test.

13.2 9755 and 4150 Comparisons

The differences between the 9755 and 4150 PDA are detailed in the following subsections.

13.2.1 Stand Alone / Self Diagnostic Mode

The 4150 PDA does not support the full stand alone mode that the 9755 FEP supports. Instead, the 4150 PDA supports only a self diagnostic mode. This mode is accessible from the Mink Diagnostic Processor by issuing the 'MO PDA' command.

13.2.2 Power Up Circuit

The 4150 PDA has a circuit that disables all internal tristate busses on power up and manual reset. It does NOT disable the Z80 memory. Internal busses are re-enabled with a single Z80 write to a known address.

13.2.3 Sense Register 4

Sense Register 4 will have the same functionality as in the 9755. However, SR4 has evolved into having each CPU board support it. Each CPU board has four testpoints connected to open collector drivers which are wire-ANDed onto the backplane and connected as the inputs to SR4. SR4 will function just as the other sense registers except that it cannot be used as an event counter input. In addition, the contents of these 4 bits will be on the stack for operator perusal.

13.2.4 Stack

The 4150 PDA will use a 1K deep stack in contrast with the 9755 which has a 256 location stack. See the PDA User's Guide for the stack format.

13.2.5 BCY Match Counter

One function that was often mentioned as an improvement to the PDA is a match counter, and this suggestion has been implemented in the 4150 PDA. This counter will, when enabled, count the number of sense register matches rather than halting or triggering. As a result, when the operator specifies the contents of a sense register, s/he can make the processor halt after a specified number of iterations of that address. Due to software dependencies the counter could only be 31-bits wide, making the maximum value of the event counter equal to 2,147,483,647 and the minimum equal to -2,147,483,648.

The match counter has an 8 to 1 mux as its enable. Connected to this Z80 programmable mux are SR1, SR2, SR3L, SR3H, SR4, GHOLD+, DMx, and 1 input that is programmable by the operator by adding a wire to the signal to be counted.

The match counter can be displayed. This means that there is now a means of metering how many times an event (e.g. a microcode step) has occurred since it was armed.

13.2.6 Microlooper

The 9755 FEP had a microlooper. The 4150 PDA does not have a microlooper. To get this functionality, the operator can update the control store. In addition to this, the method for forcing BCY address by microcode assist precludes the need for a microlooper.

13.2.7 New Halt/Trigger Condition (ARM2)

This new halt condition is a combination of both an SR2 match AND an SR3 match. This supports halting and/or triggering on a match of both Effective Address and BCY Address. This means that it can halt on any specific instruction that references a predefined effective address.

Warning: Due to the fact that there could be considerable time between the SR3 and the SR2 match, this feature may not do what you expect.

13.2.8 New Halt/Trigger Condition (ARM1)

This feature allows the operator to use SR1 as an enable and SR2 as the trigger for the stack, or to halt the CPU. It uses a registered SR1 match signal, which cannot be cleared until a RESET- becomes active. This is a new feature in the 4150 PDA.

Warning: Due to the fact that there could be considerable time between the SR1 and the SR2 match, this feature may not do what you expect.

13.2.9 Bus Virtual Memory Address (BVMA) Interface

The BEMA bus on a 9755 and the BVMA bus on a 4150 are basically the same thing. BVMA contains the Effective Address (EA) during TRCML and the Program Counter (RP) during CS1. The 4150 PDA has BVMA in the stack and as an input to sense register 3 as in the 9755. BVMA comes to the PDA active low. Please note that PDA software adjusts for this inversion.

13.3 VLSI Requirements

No VLSI parts are used to implement PDA functionality.

13.4 Major and Critical Paths

13.4.1 Stack Write Cycle

The stack write cycle has proven to be the most critical path in the PDA. The path involves sampling data from the backplane and the rest of the CPU and storing it into registers while the address for the stack becomes stable. At this point, the write pulse, which is shaped by the use of delay lines, stores the data into the stack RAM.

13.4.2 Halt Path

The timing from the sense register match to finally result in the PDAHALT- signal going active in 2 beats is a critical path.

13.4.3 Trigger Path

This path involves getting a sense register match to result in a stack trigger in two beats.

14. Microcode Functional Overview

14.1 Introduction

The 4150 microword contains 13 fields and a total word length of 80 bits. Table 14-1 briefly describes these fields.

The lion's share of functional microcode was written by Denise Chiacchia, Paul Curtis, and Andy Ray. Tom O'Brien wrote the DMx microcode. Stu Rae coordinated the microdiagnostic effort. Microdiagnostic writers were Denise Chiacchia, Tony Dorohov, Mark Mason, Stu Rae, Sherri Root, and Steve Small.

14.1.1 Microcode Fields

TABLE 14-1. 4150 Microcode Fields

RCM Bits	Field Name	# Bits	Labels
41-45	BLEG	5	RCMBB1-5
65-68	TXNX	4	RCMTXNX1-4
1,2,32-36,69,			
71-79	CU	17	RCMCU 00-16
23-29,31	DST	8	RCMDST1-8
46-49,51-55	RFS	7	RCMRF01-7
	RFD	2	RCMRF08-09
3	TR	1	RCMTR1
37-39	CS	3	RCMCS1-3
14-19,21,22	ALEG-ALU	8	RCMALU1-8
4-6	EAE	3	RCMEA1-3
7-9,11-13	BDL	6	RCMBDL1-6
56-59,61-64	IAC	8	RCMIAC1-8
10,20,30,40,			
50,60,70,80	PARITY	8	RCMPAR1-8

The EAE field is a 3-bit field which controls increment and decrement operations on the low 16-bits of the two effective address registers (EAS and EAD) .

The ALEG-ALU field controls the 56-bit ALU and the muxes on the ALEG input of the ALUs. The register file is usually the input to the A leg. Other sources include the multiplier registers, CU field, BD, RPL, and the normalize count register.

The 5-bit BLEG field specifies one of 32 possible sources to the B input of the ALUs. There are 16 sources for the 48 MSBs of the B input, and 2 sources for the 8 LSBs. Cache data is usually the input to the B leg.

The 6-bit BDL field controls the barrel shifter's operation.

The 8-bit IAC field refers to independent action codes. Hardware units which don't have their own microcode fields are used infrequently, and their functions tend to be peripheral to the main data flow through the machine. As such, they are said to function independently. The IAC field controls these independent hardware functions.

The RF fields (RFS and RFD) specify which register is to be used in the current operation. The RFS field is 7 bits wide, while the RFD field is 2 bits wide.

The CS and CU fields are used to determine the address of the next microcode instruction. The CS field is 3 bits wide, while the CU field is 17 bits wide.

The CU field contains one of ⁺wo types of information:

- An emit value, which is a number whose value is known at assembly time for use as data

- Information used to calculate the address of the next microinstruction to be executed.

The value of the CS field determines how to interpret the CU field. When the CS field is 010, bits 1-2 of the CU field further define the CS operation. Table 14-2 lists the values of the CS field and CU bits 1-2, and shows the operation specified by each pair of values.

TABLE 14-2. CS and CU Field Definitions

CS Field (RCM Bits 49-51)	CU Bits 1-2 (RCM Bits 32-33)	CU Operation
000	——	RTN
001	——	JUMP
010	00	DECODE
010	01	BDH branch
010	10	LDA
010	11	GOTO
011	——	EMIT
100	——	CRTN
101	——	CALL w/ JUMP
110	——	CALL w/ GOTO
111	——	PUSH w/ EMIT

The 8-bit DST field controls which destinations are to be written at the end of the microstep.

The 4-bit TXNX field controls how long a microstep will take to execute. There are two possible ways to extend a step:

- A TX increases the time between CS7 and CS8 by 31.25 nsec for every TX specified. For example, TX= 2 would increase the time interval by 62.5 nsec.
- An NX increases the time between CS8 and TRCML by 31.25 nsec for every NX specified. For example, NX= 2 would increase the time interval by 62.5 nsec.

The 1-bit TR field controls whether DMx traps can occur during that microstep. A TR value of 1 will disallow such traps.

There are 8 parity bits generated for the microword.

14.2 9755 and 4150 Comparisons

The microcode for the 4150 is substantially the same as that for the 9755. The most obvious exceptions include some changes in timing, changes in shift instruction implementation due to the barrel shifter, significant changes in I/O microcode, and changes in the arithmetic algorithms, especially multiply and divide. The VLSI speed and the barrel shifter functionality contribute to a number of these changes.

14.2.1 Timing Changes

The 4050 beat rate is longer than that of the 9755. As expected, some instructions take longer, on average, to execute in the 4050. However, due to data path optimization and new microcode algorithms, some instructions actually execute faster on the 4050. Exact timing comparisons are not available at this time, but will be included here in future revisions of this document.

14.2.2 Shift Instruction Implementation

The shift instructions now provide a shift amount to the barrel shifter, which implements up to a 48-bit shift in one step.

14.2.3 Multiply Implementation

Multiplication is done with a 3-bit Booth's algorithm instead of the 2-bit Booth's algorithm used in the 9755. A three operand adder is provided inside the PEALU chip.

14.2.4 Divide Implementation

The non-performing divide implemented on the 9755 has been replaced by a non-restoring divide in the 4150.

14.2.5 I/O

The microcode will have less involvement in the I/O in the 4150 than in the 9755. The primary reasons for this are:

- The 4150 beat rate is slower than that of the 9755. Consequently, an I/O transfer taking a given amount of time will execute more microcode steps on a 9755 than it will on a 4150. Therefore, the 4150 microcode can't do as many things as the 9755 microcode did. More I/O functionality must be implemented in hardware.

- Data can be routed directly from BPD to memory in the MC, so E unit data paths aren't needed.

14.3 Microdiagnostics

The microdiagnostics for the 4150 are structured much like the 9755 microdiagnostics. There is a Kernel which exercises the minimum core of hardware required to allow Mother, the microdiagnostic executive, to run reliably. The CS and E units are tested first, including the floating point logic. The I and S units are exercised next, with many of the tests devoted to parity, and cache and STLB array testing. These tests use the array test procedure established for the 9755. The memory subsystem is the last part of the microdiagnostic effort. These diagnostics test the cache miss logic, the write buffer operations, and the main memory arrays.

There are a total of 9_diagnostic overlays, known as UDIAG1-UDIAG9. Most of the diagnostic overlays map directly to SYSV overlays. UDIAG1 thru UDIAG4 are exact copies of SYSV1 thru SYSV4, UDIAG6 is a copy of SYSV5, and SYSV6 was split into two diagnostic overlays UDIAG8 and UDIAG9. This leaves two microdiagnostic overlays unaccounted for, both of which contain array tests deemed to time consuming for initialization testing. UDIAG5 contains the I Unit array tests. The following is an index of the tests contained in UDIAG5.

```
*
*   INDEX OF TESTS IN UDIAG5
*   ------------------------
*
******************************************************************************
*
*****     BEGINNING OF CACHE ARRAY TESTS     *****
*
******************************************************************************
* TEST3001 - VALID BIT TEST OF ONE CACHE CELL (A SIDE)
* TEST3002 - VALID BIT TEST OF ONE CACHE CELL (B SIDE)
* TEST3003 - VALID BIT TEST OF ALL CACHE CELLS (A SIDE)
* TEST3004 - VALID BIT TEST OF ALL CACHE CELLS (B SIDE)
* TEST3005 - VALID BIT WALKING ONES ON ADDRESS (A SIDE)
* TEST3006 - VALID BIT WALKING ONES ON ADDRESS (B SIDE)
* TEST3007 - VALID BIT WALKING ZEROES ON ADDRESS (A SIDE)
* TEST3008 - VALID BIT WALKING ZEROES ON ADDRESS (B SIDE)
* TEST3009 - VALID BIT WRITE RECOVERY (A SIDE)
* TEST300A - VALID BIT WRITE RECOVERY (B SIDE)
* TEST300B - VALID BIT READ ACCESS (A SIDE)
* TEST300C - VALID BIT READ ACCESS (B SIDE)
* TEST300D - DATA TEST OF ONE CACHE CELL (A SIDE - HI WORD)
* TEST300E - DATA TEST OF ONE CACHE CELL (A SIDE - LO WORD)
* TEST300F - DATA TEST OF ONE CACHE CELL (B SIDE - HI WORD)
* TEST3010 - DATA TEST OF ONE CACHE CELL (B SIDE - LO WORD)
* TEST3011 - DATA TEST OF ALL CACHE CELLS (A SIDE)
* TEST3012 - TEST_TNUM
* TEST3013 - DATA TEST OF ALL CACHE CELLS (B SIDE)
* TEST3014 - CACHE DATA, WALK A ONE THRU ADDRESS LINES (A SIDE)
* TEST3015 - CACHE DATA, WALK A ONE THRU ADDRESS LINES (B SIDE)
* TEST3016 - CACHE DATA, WALK A ZERO THRU ADDRESS LINES (A SIDE)
* TEST3017 - CACHE DATA, WALK A ZERO THRU ADDRESS LINES (B SIDE)
* TEST3018 - CACHE DATA WRITE RECOVERY (A SIDE)
* TEST3019 - CACHE DATA WRITE RECOVERY (B SIDE)
* TEST301A - TEST_TNUM
* TEST301B - CACHE DATA READ ACCESS (A SIDE)
* TEST301C - CACHE DATA READ ACCESS (B SIDE)
* TEST301D - CACHE DATA CONTENTS EQUAL ADDRESS (A SIDE)
* TEST301E - CACHE DATA CONTENTS EQUAL ADDRESS (B SIDE)
* TEST301F - TEST_TNUM
* TEST3020 - FORCE BITS TEST (A SIDE - FORCE BIT SET)
* TEST3021 - FORCE BITS TEST (A SIDE - FORCE BIT CLEARED)
* TEST3022 - FORCE BITS TEST (B SIDE - FORCE BIT SET)
* TEST3023 - FORCE BITS TEST (B SIDE - FORCE BIT CLEARED)
* TEST3024 - TEST_TNUM
* TEST3030 - INDEX TEST ONE CELL (A SIDE)
* TEST3031 - INDEX TEST ONE CELL (B SIDE)
* TEST3032 - INDEX TEST ALL CELLS (A SIDE)
* TEST3033 - INDEX TEST ALL CELLS (B SIDE)
* TEST3034 - INDEX WALKING ONES ON ADDRESS (A SIDE)
* TEST3035 - TEST_TNUM
* TEST3036 - INDEX WALKING ONES ON ADDRESS (B SIDE)
* TEST3037 - INDEX WALKING ZEROES ON ADDRESS (A SIDE)
* TEST3038 - INDEX WALKING ZEROES ON ADDRESS (B SIDE)
* TEST3039 - INDEX WRITE RECOVERY (A SIDE)
* TEST303A - INDEX WRITE RECOVERY (B SIDE)
* TEST303B - TEST_TNUM
* TEST303C - INDEX READ ACCESS (A SIDE)
* TEST303D - INDEX READ ACCESS (B SIDE)
* TEST303E - TEST_TNUM
*
*
******************************************************************************
*
*****     BEGINNING OF STLB ARRAY TESTS     *****
*
******************************************************************************
* TEST3200 - PART 2 OF THE LBVAA ARRAY TESTS
* TEST3201 - PART 2 OF THE LBVAB ARRAY TESTS
* TEST3202 - PART 2 OF THE LBPAA ARRAY TESTS
* TEST3203 - PART 2 OF THE LBPAB ARRAY TESTS
* TEST3204 - PART 2 OF THE LPIDA ARRAY TESTS
* TEST3205 - PART 2 OF THE LPIDB ARRAY TESTS
* TEST3206 - PART 2 OF THE PCNTA ARRAY TESTS
* TEST3207 - PART 2 OF THE PCNTB ARRAY TESTS
* TEST3208 - PART 2 OF THE ARRA ARRAY TESTS
* TEST3209 - PART 2 OF THE ARRB ARRAY TESTS
* TEST320A - PART 2 OF THE PMODA ARRAY TESTS
* TEST320B - PART 2 OF THE PMODB ARRAY TESTS
* TEST320C - PART 2 OF THE FRCA ARRAY TESTS
* TEST320D - PART 2 OF THE FRCB ARRAY TESTS
* TEST320E - PART 2 OF THE LBPVLDA ARRAY TESTS
* TEST320F - PART 2 OF THE LBPVLDB ARRAY TESTS
* TEST3210 - PART 2 OF THE PARITY BIT ARRAY TESTS, STLBA
* TEST3211 - PART 2 OF THE PARITY BIT ARRAY TESTS, STLBB
* TEST3212 - PART 2 OF THE PARITY BIT ARRAY TESTS, IOTLBA
* TEST3213 - PART 2 OF THE PARITY BIT ARRAY TESTS, IOTLBB
* TEST3214 - PART 2 OF THE LBPOWNA ARRAY TESTS
* TEST3215 - PART 2 OF THE LBPOWNB ARRAY TESTS
* TEST3216 - TEST_TNUM
****** SAVE SOME ROOM FOR FUTURE LBVAAW AND LBVABW WALKING 1'S AND 0'S TESTS
* TEST3250 - LBPAA, FLOAT 1 THROUGH ADDRESS OF STLBA, CALL LBPAAW
* TEST3251 - LBPAA, FLOAT 1 THROUGH ADDRESS OF STLBA, CALL LBPAAW
* TEST3252 - LBPAA, FLOAT 1 THROUGH ADDRESS OF STLBA, CALL LBPAAW
* TEST3260 - LBPAA, FLOAT 0 THROUGH ADDRESS OF STLBA, CALL LBPAAW
* TEST3261 - LBPAA, FLOAT 0 THROUGH ADDRESS OF STLBA, CALL LBPAAW
* TEST3262 - LBPAA, FLOAT 0 THROUGH ADDRESS OF STLBA, CALL LBPAAW
* LBPAAW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE LBPAA ARRAY TESTS.
* TEST3270 - LBPAB, FLOAT 1 THROUGH ADDRESS OF STLBB, CALL LBPABW
* TEST3271 - LBPAB, FLOAT 1 THROUGH ADDRESS OF STLBB, CALL LBPABW
* TEST3272 - LBPAB, FLOAT 1 THROUGH ADDRESS OF STLBB, CALL LBPABW
* TEST3280 - LBPAB, FLOAT 0 THROUGH ADDRESS OF STLBB, CALL LBPABW
* TEST3281 - LBPAB, FLOAT 0 THROUGH ADDRESS OF STLBB, CALL LBPABW
* TEST3282 - LBPAB, FLOAT 0 THROUGH ADDRESS OF STLBB, CALL LBPABW
* LBPABW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE LBPAB ARRAY TESTS.
* TEST3290 - PART 5 OF THE STLBA LBPAA ARRAY TESTWRITE RECOVERY CHECK
* TEST3291 - PART 5 OF THE STLBB LBPAB ARRAY TESTWRITE RECOVERY CHECK
* TEST3292 - PART 6 OF THE STLBA ARRAY TESTREAD ACCESS CHECK OF LBPAA
* TEST3293 - PART 6 OF THE STLBB ARRAY TESTREAD ACCESS CHECK OF LBPAB
* TEST32A0 - LPIDA, FLOAT 1 THROUGH ADDRESS STLBA, CALL LPIDAW
* TEST32A1 - LPIDA, FLOAT 1 THROUGH ADDRESS STLBA, CALL LPIDAW
* TEST32A2 - LPIDA, FLOAT 1 THROUGH ADDRESS STLBA, CALL LPIDAW
* TEST32B0 - LPIDA, FLOAT 0 THROUGH ADDRESS STLBA, CALL LPIDAW
* TEST32B1 - LPIDA, FLOAT 0 THROUGH ADDRESS STLBA, CALL LPIDAW
* TEST32B2 - LPIDA, FLOAT 0 THROUGH ADDRESS STLBA, CALL LPIDAW
* LPIDAW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE LPIDA ARRAY TESTS.
* TEST32C0 - LPIDB, FLOAT 1 THROUGH ADDRESS STLBB, CALL LPIDBW
* TEST32C1 - LPIDB, FLOAT 1 THROUGH ADDRESS STLBB, CALL LPIDBW
* TEST32C2 - LPIDB, FLOAT 1 THROUGH ADDRESS STLBB, CALL LPIDBW
* TEST32D0 - LPIDB, FLOAT 0 THROUGH ADDRESS STLBB, CALL LPIDBW
* TEST32D1 - LPIDB, FLOAT 0 THROUGH ADDRESS STLBB, CALL LPIDBW
* TEST32D2 - LPIDB, FLOAT 0 THROUGH ADDRESS STLBB, CALL LPIDBW
* LPIDBW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE LPIDB ARRAY TESTS.
* TEST32E0 - PART 5 OF THE STLBA LPIDA ARRAY TEST WRITE RECOVERY CHECK
* TEST32E1 - PART 5 OF THE STLBB LPIDB ARRAY TEST WRITE RECOVERY CHECK
* TEST32E2 - PART 6 OF THE STLBA ARRAY TEST READ ACCESS CHECK OF LPIDA
* TEST32E3 - PART 6 OF THE STLBB ARRAY TEST READ ACCESS CHECK OF LPIDB
* TEST32F0 - LPCNTA, FLOAT 1 THROUGH ADDRESS STLBA, CALL LPCAW
* TEST32F1 - LPCNTA, FLOAT 1 THROUGH ADDRESS STLBA, CALL LPCAW
* TEST32F2 - LPCNTA, FLOAT 1 THROUGH ADDRESS STLBA, CALL LPCAW
* TEST3300 - LPCNTA, FLOAT 0 THROUGH ADDRESS STLBA, CALL LPCAW
* TEST3301 - LPCNTA, FLOAT 0 THROUGH ADDRESS STLBA, CALL LPCAW
```

* TEST3302 - LPCNTA, FLOAT 0 THROUGH ADDRESS STLBA, CALL LPCAW
* LPCAW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE LPCNTA ARRAY TESTS.
* TEST3310 - LPCNTB, FLOAT 1 THROUGH ADDRESS STLBB, CALL LPCBW
* TEST3311 - LPCNTB, FLOAT 1 THROUGH ADDRESS STLBB, CALL LPCBW
* TEST3312 - LPCNTB, FLOAT 1 THROUGH ADDRESS STLBB, CALL LPCBW
* TEST3320 - LPCNTB, FLOAT 0 THROUGH ADDRESS STLBB, CALL LPCBW
* TEST3321 - LPCNTB, FLOAT 0 THROUGH ADDRESS STLBB, CALL LPCBW
* TEST3322 - LPCNTB, FLOAT 0 THROUGH ADDRESS STLBB, CALL LPCBW
* LPCBW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE LPCNTB ARRAY TESTS.
* TEST3330 - PART 5 OF THE STLBA LPCNTA ARRAY TEST WRITE RECOVERY CHECK
* TEST3331 - PART 5 OF THE STLBB LPCNTB ARRAY TEST WRITE RECOVERY CHECK
* TEST3332 - PART 6 OF THE STLBA ARRAY TEST READ ACCESS CHECK OF LPCNTA
* TEST3333 - PART 6 OF THE STLBB ARRAY TEST READ ACCESS CHECK OF LPCNTB
* TEST3334 - TEST_TNUM
* TEST3340 - ARRA, FLOAT 1 THROUGH ADDRESS STLBA, CALL ARRAW
* TEST3341 - ARRA, FLOAT 1 THROUGH ADDRESS STLBA, CALL ARRAW
* TEST3342 - ARRA, FLOAT 1 THROUGH ADDRESS STLBA, CALL ARRAW
* TEST3350 - ARRA, FLOAT 0 THROUGH ADDRESS STLBA, CALL ARRAW
* TEST3351 - ARRA, FLOAT 0 THROUGH ADDRESS STLBA, CALL ARRAW
* TEST3352 - ARRA, FLOAT 0 THROUGH ADDRESS STLBA, CALL ARRAW
* ARRAW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE ACCESS RIGHTS ARRAY TESTS.
* TEST3360 - ARRB, FLOAT 1 THROUGH ADDRESS STLBB, CALL ARRBW
* TEST3361 - ARRB, FLOAT 1 THROUGH ADDRESS STLBB, CALL ARRBW
* TEST3362 - ARRB, FLOAT 1 THROUGH ADDRESS STLBB, CALL ARRBW
* TEST3370 - ARRB, FLOAT 0 THROUGH ADDRESS STLBB, CALL ARRBW
* TEST3371 - ARRB, FLOAT 0 THROUGH ADDRESS STLBB, CALL ARRBW
* TEST3372 - ARRB, FLOAT 0 THROUGH ADDRESS STLBB, CALL ARRBW
* ARRBW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE ACCESS RIGHTS ARRAY TESTS.
* TEST3380 - PART 5 OF THE STLBA ARRA ARRAY TEST WRITE RECOVERY CHECK
* TEST3381 - PART 5 OF THE STLBB ARRB ARRAY TEST WRITE RECOVERY CHECK
* TEST3382 - PART 6 OF THE STLBA ARRAY TEST READ ACCESS CHECK OF ARRA
* TEST3383 - PART 6 OF THE STLBB ARRAY TEST READ ACCESS CHECK OF ARRB
* TEST3390 - PMODA, FLOAT 1 THROUGH ADDRESS STLBA, CALL PMODAW
* TEST3391 - PMODA, FLOAT 1 THROUGH ADDRESS STLBA, CALL PMODAW
* TEST3392 - PMODA, FLOAT 1 THROUGH ADDRESS STLBA, CALL PMODAW
* TEST33A0 - PMODA, FLOAT 0 THROUGH ADDRESS STLBA, CALL PMODAW
* TEST33A1 - PMODA, FLOAT 0 THROUGH ADDRESS STLBA, CALL PMODAW
* TEST33A2 - PMODA, FLOAT 0 THROUGH ADDRESS STLBA, CALL PMODAW
* PMODAW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE PMODA ARRAY TESTS.
* TEST33B0 - PMODB, FLOAT 1 THROUGH ADDRESS STLBB, CALL PMODBW
* TEST33B1 - PMODB, FLOAT 1 THROUGH ADDRESS STLBB, CALL PMODBW
* TEST33B2 - PMODB, FLOAT 1 THROUGH ADDRESS STLBB, CALL PMODBW
* TEST33C0 - PMODB, FLOAT 0 THROUGH ADDRESS STLBB, CALL PMODBW
* TEST33C1 - PMODB, FLOAT 0 THROUGH ADDRESS STLBB, CALL PMODBW
* TEST33C2 - PMODB, FLOAT 0 THROUGH ADDRESS STLBB, CALL PMODBW
* PMODBW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE PMODB ARRAY TESTS.
* TEST33D0 - PART 5 OF THE STLBA PMODA ARRAY TEST WRITE RECOVERY CHECK
* TEST33D1 - PART 5 OF THE STLBB PMODB ARRAY TEST WRITE RECOVERY CHECK
* TEST33D2 - PART 6 OF THE STLBA ARRAY TEST READ ACCESS CHECK OF PMODA
* TEST33D3 - PART 6 OF THE STLBB ARRAY TEST READ ACCESS CHECK OF PMODB
* TEST33E0 - FRC, FLOAT 1 THROUGH ADDRESS STLBA, CALL FRCAW
* TEST33E1 - FRC, FLOAT 1 THROUGH ADDRESS STLBA, CALL FRCAW
* TEST33E2 - FRC, FLOAT 1 THROUGH ADDRESS STLBA, CALL FRCAW
* TEST33F0 - FRC, FLOAT 0 THROUGH ADDRESS STLBA, CALL FRCAW
* TEST33F2 - FRC, FLOAT 0 THROUGH ADDRESS STLBA, CALL FRCAW
* TEST33F3 - FRC, FLOAT 0 THROUGH ADDRESS STLBA, CALL FRCAW
* FRCAW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE FRCA ARRAY TESTS.
* TEST3400 - FRC, FLOAT 1 THROUGH ADDRESS STLBB, CALL FRCBW
* TEST3401 - FRC, FLOAT 1 THROUGH ADDRESS STLBB, CALL FRCBW
* TEST3402 - FRC, FLOAT 1 THROUGH ADDRESS STLBB, CALL FRCBW
* TEST3410 - FRC, FLOAT 0 THROUGH ADDRESS STLBB, CALL FRCBW

```
* TEST3411 - FRC, FLOAT 0 THROUGH ADDRESS STLBB, CALL FRCBW
* TEST3412 - FRC, FLOAT 0 THROUGH ADDRESS STLBB, CALL FRCBW
* FRCBW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE FRCB ARRAY TESTS.
* TEST3420 - PART 5 OF THE STLBA FRCA ARRAY TEST WRITE RECOVERY CHECK
* TEST3421 - PART 5 OF THE STLBA FRCA ARRAY TEST WRITE RECOVERY CHECK
* TEST3422 - PART 5 OF THE STLBB FRCB ARRAY TEST WRITE RECOVERY CHECK
* TEST3423 - PART 5 OF THE STLBB FRCB ARRAY TEST WRITE RECOVERY CHECK
* TEST3424 - PART 6 OF THE STLBA FRCA ARRAY TEST READ ACCESS CHECK
* TEST3425 - PART 6 OF THE STLBA FRCA ARRAY TESTREAD ACCESS CHECK
* TEST3426 - PART 6 OF THE STLBB FRCB ARRAY TEST READ ACCESS CHECK
* TEST3427 - PART 6 OF THE STLBB FRCB ARRAY TESTREAD ACCESS CHECK
* TEST3430 - VLDA, FLOAT 1 THROUGH ADDRESS STLBA, CALL VLDAW
* TEST3431 - VLDA, FLOAT 1 THROUGH ADDRESS STLBA, CALL VLDAW
* TEST3432 - VLDA, FLOAT 1 THROUGH ADDRESS STLBA, CALL VLDAW
* TEST3440 - VLDA, FLOAT 0 THROUGH ADDRESS STLBA, CALL VLDAW
* TEST3441 - VLDA, FLOAT 0 THROUGH ADDRESS STLBA, CALL VLDAW
* TEST3442 - VLDA, FLOAT 0 THROUGH ADDRESS STLBA, CALL VLDAW
* VLDAW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE VLDA ARRAY TESTS.
* TEST3450 - VLDB, FLOAT 1 THROUGH ADDRESS STLBB, CALL VLDBW
* TEST3451 - VLDB, FLOAT 1 THROUGH ADDRESS STLBB, CALL VLDBW
* TEST3452 - VLDB, FLOAT 1 THROUGH ADDRESS STLBB, CALL VLDBW
* TEST3460 - VLDB, FLOAT 0 THROUGH ADDRESS STLBB, CALL VLDBW
* TEST3461 - VLDB, FLOAT 0 THROUGH ADDRESS STLBB, CALL VLDBW
* TEST3462 - VLDB, FLOAT 0 THROUGH ADDRESS STLBB, CALL VLDBW
* VLDBW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE VLDB ARRAY TESTS.
* TEST3470 - PART 5 OF THE STLBA VLDA ARRAY TEST WRITE RECOVERY CHECK
* TEST3471 - PART 5 OF THE STLBA VLDA ARRAY TEST WRITE RECOVERY CHECK
* TEST3472 - PART 5 OF THE STLBB VLDB ARRAY TEST WRITE RECOVERY CHECK
* TEST3473 - PART 5 OF THE STLBB VLDB ARRAY TEST WRITE RECOVERY CHECK
* TEST3474 - PART 6 OF THE STLBA VLDA ARRAY TEST READ ACCESS CHECK
* TEST3475 - PART 6 OF THE STLBB VLDA ARRAY TESTREAD ACCESS CHECK
* TEST3476 - PART 6 OF THE STLBB VLDB ARRAY TEST READ ACCESS CHECK
* TEST3477 - PART 6 OF THE STLBB VLDB ARRAY TESTREAD ACCESS CHECK
* TEST3480 - PARITY, FLOAT 1 THROUGH ADDRESS STLBA, CALL PARAW
* TEST3481 - PARITY, FLOAT 1 THROUGH ADDRESS STLBA, CALL PARAW
* TEST3482 - PARITY, FLOAT 1 THROUGH ADDRESS STLBA, CALL PARAW
* TEST3490 - PARITY, FLOAT 0 THROUGH ADDRESS STLBA, CALL PARAW
* TEST3491 - PARITY, FLOAT 0 THROUGH ADDRESS STLBA, CALL PARAW
* TEST3492 - PARITY, FLOAT 0 THROUGH ADDRESS STLBA, CALL PARAW
* PARAW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE PARITY BIT ARRAY TESTS.
* TEST34A0 - PARITY, FLOAT 1 THROUGH ADDRESS STLBB, CALL PARBW
* TEST34A1 - PARITY, FLOAT 1 THROUGH ADDRESS STLBB, CALL PARBW
* TEST34A2 - PARITY, FLOAT 1 THROUGH ADDRESS STLBB, CALL PARBW
* TEST34B0 - PARITY, FLOAT 0 THROUGH ADDRESS STLBB, CALL PARBW
* TEST34B1 - PARITY, FLOAT 0 THROUGH ADDRESS STLBB, CALL PARBW
* TEST34B2 - PARITY, FLOAT 0 THROUGH ADDRESS STLBB, CALL PARBW
* PARBW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE PARITY BIT ARRAY TESTS.
* TEST34C0 - PART 5 OF THE STLBA PARITY ARRAY TEST WRITE RECOVERY CHECK
* TEST34C1 - PART 5 OF THE STLBA PARITY ARRAY TEST WRITE RECOVERY CHECK
* TEST34C2 - PART 5 OF THE STLBB PARITY ARRAY TEST WRITE RECOVERY CHECK
* TEST34C3 - PART 5 OF THE STLBB PARITY ARRAY TEST WRITE RECOVERY CHECK
* TEST34C4 - PART 6 OF THE STLBA PARITY ARRAY TEST READ ACCESS CHECK
* TEST34C5 - PART 6 OF THE STLBA PARITY ARRAY TESTREAD ACCESS CHECK
* TEST34C6 - PART 6 OF THE STLBB PARITY ARRAY TEST READ ACCESS CHECK
* TEST34C7 - PART 6 OF THE STLBB PARITY ARRAY TESTREAD ACCESS CHECK
****** SAVE SOME ROOM FOR FUTURE IOTLB WALKING 1'S AND 0'S TESTS
****** SAVE SOME ROOM FOR FUTURE IOTLB WRITE RECOVERY TESTS
****** SAVE SOME ROOM FOR FUTURE IOTLB READ ACCESS TESTS
* TEST3520 - LBPOWN, FLOAT 1 THROUGH ADDRESS STLBA, CALL POWNAW
* TEST3521 - LBPOWN, FLOAT 1 THROUGH ADDRESS STLBA, CALL POWNAW
* TEST3522 - LBPOWN, FLOAT 1 THROUGH ADDRESS STLBA, CALL POWNAW
* TEST3530 - LBPOWN, FLOAT 0 THROUGH ADDRESS STLBA, CALL POWNAW
* TEST3531 - LBPOWN, FLOAT 0 THROUGH ADDRESS STLBA, CALL POWNAW
* TEST3532 - LBPOWN, FLOAT 0 THROUGH ADDRESS STLBA, CALL POWNAW
* POWNAW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE POWNA ARRAY TESTS
* TEST3540 - LBPOWN, FLOAT 1 THROUGH ADDRESS STLBB, CALL POWNBW
* TEST3541 - LBPOWN, FLOAT 1 THROUGH ADDRESS STLBB, CALL POWNBW
* TEST3542 - LBPOWN, FLOAT 1 THROUGH ADDRESS STLBB, CALL POWNBW
* TEST3550 - LBPOWN, FLOAT 0 THROUGH ADDRESS STLBB, CALL POWNBW
* TEST3551 - LBPOWN, FLOAT 0 THROUGH ADDRESS STLBB, CALL POWNBW
* TEST3552 - LBPOWN, FLOAT 0 THROUGH ADDRESS STLBB, CALL POWNBW
* POWNBW - COMMON ROUTINE USED FOR PARTS 3 & 4 OF THE POWNB ARRAY TESTS
* TEST3560 - PART 5 OF THE STLBA LBPOWNA ARRAY TEST WRITE RECOVERY CHECK
* TEST3561 - PART 5 OF THE STLBA LBPOWNA ARRAY TEST WRITE RECOVERY CHECK
* TEST3562 - PART 5 OF THE STLBB LBPOWNB ARRAY TEST WRITE RECOVERY CHECK
* TEST3563 - PART 5 OF THE STLBB LBPOWNB ARRAY TEST WRITE RECOVERY CHECK
* TEST3564 - PART 6 OF THE STLBA LBPOWNA ARRAY TEST READ ACCESS CHECK
* TEST3565 - PART 6 OF THE STLBA LBPOWNA ARRAY TEST READ ACCESS CHECK
* TEST3566 - PART 6 OF THE STLBB LBPOWNB ARRAY TEST READ ACCESS CHECK
* TEST3567 - PART 6 OF THE STLBB LBPOWNB ARRAY TEST READ ACCESS CHECK
* TEST3568 - TEST_TNUM
*
```

UDIAG7 contains the Memory array tests. The following is an index of the tests contained in UDIAG7.

```
*
*   INDEX OF TESTS IN UDIAG7
*   ________________________
*
* TEST2200 - MEMORY SCAN DATA TEST 0 - 32 MB (CALLS TEST220X)
* TEST2201 - TEST_TNUM
* TEST2202 - MEMORY SCAN DATA TEST 32 - 64 MB (CALLS TEST220X)
* TEST2203 - TEST_TNUM
* TEST2210 - MEMORY MARCH DATA TEST 0 - 16 MB (CALLS TEST221X)
* TEST2211 - TEST_TNUM
* TEST2212 - MEMORY MARCH DATA TEST 16 - 32 MB (CALLS TEST221X)
* TEST2213 - TEST_TNUM
* TEST2214 - MEMORY MARCH DATA TEST 32 - 48 MB (CALLS TEST221X)
* TEST2215 - TEST_TNUM
* TEST2216 - MEMORY MARCH DATA TEST 48 - 64 MB (CALLS TEST221X)
* TEST2217 - TEST_TNUM
* TEST2220 - ADDRESS AS DATA TEST 0 - 32 MB (CALLS TEST222X)
* TEST2221 - TEST_TNUM
* TEST2222 - ADDRESS AS DATA TEST 32 - 64 MB (CALLS TEST222X)
* TEST2223 - TEST_TNUM
* TEST2230 - MEMORY MARCH CHECK BITS TEST 0 - 16 MB (CALLS TEST221X)
* TEST2231 - TEST_TNUM
* TEST2232 - MEMORY MARCH CHECK BITS TEST 16 - 32 MB (CALLS TEST221X)
* TEST2233 - TEST_TNUM
* TEST2234 - MEMORY MARCH CHECK BITS TEST 32 - 48 MB (CALLS TEST221X)
* TEST2235 - TEST_TNUM
* TEST2236 - MEMORY MARCH CHECK BITS TEST 48 - 64 MB (CALLS TEST221X)
* TEST2237 - TEST_TNUM
* TEST2240 - WALKING 1'S MEMORY DATA TEST, BITS 1-4, 0-8 MB (CALLS TEST224X)
* TEST2241 - TEST_TNUM
* TEST2242 - WALKING 1'S MEMORY DATA TEST, BITS 5-8, 0-8 MB (CALLS TEST224X)
* TEST2243 - TEST_TNUM
* TEST2244 - WALKING 1'S MEMORY DATA TEST, BITS 9-12, 0-8 MB (CALLS TEST224X)
* TEST2245 - TEST_TNUM
* TEST2246 - WALKING 1'S MEMORY DATA TEST, BITS 13-16, 0-8 MB (CALLS TEST224X)
* TEST2247 - TEST_TNUM
```

* TEST2248 – WALKING 1'S MEMORY DATA TEST, BITS 17–20, 0–8 MB (CALLS TEST224X)
* TEST2249 – TEST_TNUM
* TEST224A – WALKING 1'S MEMORY DATA TEST, BITS 21–24, 0–8 MB (CALLS TEST224X)
* TEST224B – TEST_TNUM
* TEST224C – WALKING 1'S MEMORY DATA TEST, BITS 25–28, 0–8 MB (CALLS TEST224X)
* TEST224D – TEST_TNUM
* TEST224E – WALKING 1'S MEMORY DATA TEST, BITS 29–32, 0–8 MB (CALLS TEST224X)
* TEST224F – TEST_TNUM
* TEST2250 – WALKING 1'S MEMORY DATA TEST, BITS 1–4, 8–16 MB (CALLS TEST224X)
* TEST2251 – TEST_TNUM
* TEST2252 – WALKING 1'S MEMORY DATA TEST, BITS 5–8, 8–16 MB (CALLS TEST224X)
* TEST2253 – TEST_TNUM
* TEST2254 – WALKING 1'S MEMORY DATA TEST, BITS 9–12, 8–16 MB (CALLS TEST224X)
* TEST2255 – TEST_TNUM
* TEST2256 – WALKING 1'S MEMORY DATA TEST, BITS 13–16, 8–16 MB (CALLS TEST224X)
* TEST2257 – TEST_TNUM
* TEST2258 – WALKING 1'S MEMORY DATA TEST, BITS 17–20, 8–16 MB (CALLS TEST224X)
* TEST2259 – TEST_TNUM
* TEST225A – WALKING 1'S MEMORY DATA TEST, BITS 21–24, 8–16 MB (CALLS TEST224X)
* TEST225B – TEST_TNUM
* TEST225C – WALKING 1'S MEMORY DATA TEST, BITS 25–28, 8–16 MB (CALLS TEST224X)
* TEST225D – TEST_TNUM
* TEST225E – WALKING 1'S MEMORY DATA TEST, BITS 29–32, 8–16 MB (CALLS TEST224X)
* TEST225F – TEST_TNUM
* TEST2260 – WALKING 1'S MEMORY DATA TEST, BITS 1–4, 16–24 MB (CALLS TEST224X)
* TEST2261 – TEST_TNUM
* TEST2262 – WALKING 1'S MEMORY DATA TEST, BITS 5–8, 16–24 MB (CALLS TEST224X)
* TEST2263 – TEST_TNUM
* TEST2264 – WALKING 1'S MEMORY DATA TEST, BITS 9–12, 16–24 MB (CALLS TEST224X)
* TEST2265 – TEST_TNUM
* TEST2266 – WALKING 1'S MEMORY DATA TEST, BITS 13–16, 16–24 MB (CALLS TEST224X)
* TEST2267 – TEST_TNUM
* TEST2268 – WALKING 1'S MEMORY DATA TEST, BITS 17–20, 16–24 MB (CALLS TEST224X)
* TEST2269 – TEST_TNUM
* TEST226A – WALKING 1'S MEMORY DATA TEST, BITS 21–24, 16–24 MB (CALLS TEST224X)
* TEST226B – TEST_TNUM
* TEST226C – WALKING 1'S MEMORY DATA TEST, BITS 25–28, 16–24 MB (CALLS TEST224X)
* TEST226D – TEST_TNUM
* TEST226E – WALKING 1'S MEMORY DATA TEST, BITS 29–32, 16–24 MB (CALLS TEST224X)
* TEST226F – TEST_TNUM
* TEST2270 – WALKING 1'S MEMORY DATA TEST, BITS 1–4, 24–32 MB (CALLS TEST224X)
* TEST2271 – TEST_TNUM
* TEST2272 – WALKING 1'S MEMORY DATA TEST, BITS 5–8, 24–32 MB (CALLS TEST224X)
* TEST2273 – TEST_TNUM
* TEST2274 – WALKING 1'S MEMORY DATA TEST, BITS 9–12, 24–32 MB (CALLS TEST224X)
* TEST2275 – TEST_TNUM
* TEST2276 – WALKING 1'S MEMORY DATA TEST, BITS 13–16, 24–32 MB (CALLS TEST224X)
* TEST2277 – TEST_TNUM
* TEST2278 – WALKING 1'S MEMORY DATA TEST, BITS 17–20, 24–32 MB (CALLS TEST224X)
* TEST2279 – TEST_TNUM
* TEST227A – WALKING 1'S MEMORY DATA TEST, BITS 21–24, 24–32 MB (CALLS TEST224X)
* TEST227B – TEST_TNUM
* TEST227C – WALKING 1'S MEMORY DATA TEST, BITS 25–28, 24–32 MB (CALLS TEST224X)
* TEST227D – TEST_TNUM
* TEST227E – WALKING 1'S MEMORY DATA TEST, BITS 29–32, 24–32 MB (CALLS TEST224X)
* TEST227F – TEST_TNUM
* TEST2280 – WALKING 1'S MEMORY DATA TEST, BITS 1–4, 32–40 MB (CALLS TEST224X)
* TEST2281 – TEST_TNUM
* TEST2282 – WALKING 1'S MEMORY DATA TEST, BITS 5–8, 32–40 MB (CALLS TEST224X)
* TEST2283 – TEST_TNUM
* TEST2284 – WALKING 1'S MEMORY DATA TEST, BITS 9–12, 32–40 MB (CALLS TEST224X)

* TEST2285 – TEST_TNUM
* TEST2286 – WALKING 1'S MEMORY DATA TEST, BITS 13–16, 32–40 MB (CALLS TEST224X)
* TEST2287 – TEST_TNUM
* TEST2288 – WALKING 1'S MEMORY DATA TEST, BITS 17–20, 32–40 MB (CALLS TEST224X)
* TEST2289 – TEST_TNUM
* TEST228A – WALKING 1'S MEMORY DATA TEST, BITS 21–24, 32–40 MB (CALLS TEST224X)
* TEST228B – TEST_TNUM
* TEST228C – WALKING 1'S MEMORY DATA TEST, BITS 25–28, 32–40 MB (CALLS TEST224X)
* TEST228D – TEST_TNUM
* TEST228E – WALKING 1'S MEMORY DATA TEST, BITS 29–32, 32–40 MB (CALLS TEST224X)
* TEST228F – TEST_TNUM
* TEST2290 – WALKING 1'S MEMORY DATA TEST, BITS 1–4, 40–48 MB (CALLS TEST224X)
* TEST2291 – TEST_TNUM
* TEST2292 – WALKING 1'S MEMORY DATA TEST, BITS 5–8, 40–48 MB (CALLS TEST224X)
* TEST2293 – TEST_TNUM
* TEST2294 – WALKING 1'S MEMORY DATA TEST, BITS 9–12, 40–48 MB (CALLS TEST224X)
* TEST2295 – TEST_TNUM
* TEST2296 – WALKING 1'S MEMORY DATA TEST, BITS 13–16, 40–48 MB (CALLS TEST224X)
* TEST2297 – TEST_TNUM
* TEST2298 – WALKING 1'S MEMORY DATA TEST, BITS 17–20, 40–48 MB (CALLS TEST224X)
* TEST2299 – TEST_TNUM
* TEST229A – WALKING 1'S MEMORY DATA TEST, BITS 21–24, 40–48 MB (CALLS TEST224X)
* TEST229B – TEST_TNUM
* TEST229C – WALKING 1'S MEMORY DATA TEST, BITS 25–28, 40–48 MB (CALLS TEST224X)
* TEST229D – TEST_TNUM
* TEST229E – WALKING 1'S MEMORY DATA TEST, BITS 29–32, 40–48 MB (CALLS TEST224X)
* TEST229F – TEST_TNUM
* TEST22A0 – WALKING 1'S MEMORY DATA TEST, BITS 1–4, 48–56 MB (CALLS TEST224X)
* TEST22A1 – TEST_TNUM
* TEST22A2 – WALKING 1'S MEMORY DATA TEST, BITS 5–8, 48–56 MB (CALLS TEST224X)
* TEST22A3 – TEST_TNUM
* TEST22A4 – WALKING 1'S MEMORY DATA TEST, BITS 9–12, 48–56 MB (CALLS TEST224X)
* TEST22A5 – TEST_TNUM
* TEST22A6 – WALKING 1'S MEMORY DATA TEST, BITS 13–16, 48–56 MB (CALLS TEST224X)
* TEST22A7 – TEST_TNUM
* TEST22A8 – WALKING 1'S MEMORY DATA TEST, BITS 17–20, 48–56 MB (CALLS TEST224X)
* TEST22A9 – TEST_TNUM
* TEST22AA – WALKING 1'S MEMORY DATA TEST, BITS 21–24, 48–56 MB (CALLS TEST224X)
* TEST22AB – TEST_TNUM
* TEST22AC – WALKING 1'S MEMORY DATA TEST, BITS 25–28, 48–56 MB (CALLS TEST224X)
* TEST22AD – TEST_TNUM
* TEST22AE – WALKING 1'S MEMORY DATA TEST, BITS 29–32, 48–56 MB (CALLS TEST224X)
* TEST22AF – TEST_TNUM
* TEST22B0 – WALKING 1'S MEMORY DATA TEST, BITS 1–4, 56–64 MB (CALLS TEST224X)
* TEST22B1 – TEST_TNUM
* TEST22B2 – WALKING 1'S MEMORY DATA TEST, BITS 5–8, 56–64 MB (CALLS TEST224X)
* TEST22B3 – TEST_TNUM
* TEST22B4 – WALKING 1'S MEMORY DATA TEST, BITS 9–12, 56–64 MB (CALLS TEST224X)
* TEST22B5 – TEST_TNUM
* TEST22B6 – WALKING 1'S MEMORY DATA TEST, BITS 13–16, 56–64 MB (CALLS TEST224X)
* TEST22B7 – TEST_TNUM
* TEST22B8 – WALKING 1'S MEMORY DATA TEST, BITS 17–20, 56–64 MB (CALLS TEST224X)
* TEST22B9 – TEST_TNUM
* TEST22BA – WALKING 1'S MEMORY DATA TEST, BITS 21–24, 56–64 MB (CALLS TEST224X)
* TEST22BB – TEST_TNUM
* TEST22BC – WALKING 1'S MEMORY DATA TEST, BITS 25–28, 56–64 MB (CALLS TEST224X)
* TEST22BD – TEST_TNUM
* TEST22BE – WALKING 1'S MEMORY DATA TEST, BITS 29–32, 56–64 MB (CALLS TEST224X)
* TEST22BF – TEST_TNUM
* TEST22C0 – WALKING 1'S MEMORY ADDRESS TEST, MA14–13, 0–8 MB (CALLS TEST22CX)
* TEST22C1 – TEST_TNUM

```
* TEST22C2 - WALKING 1'S MEMORY ADDRESS TEST, MA12-11, 0-8 MB (CALLS TEST22CX)
* TEST22C3 - TEST_TNUM
* TEST22C4 - WALKING 1'S MEMORY ADDRESS TEST, MA10-9, 0-8 MB (CALLS TEST22CX)
* TEST22C5 - TEST_TNUM
* TEST22C6 - WALKING 1'S MEMORY ADDRESS TEST, MA8-7, 0-8 MB (CALLS TEST22CX)
* TEST22C7 - TEST_TNUM
* TEST22C8 - WALKING 1'S MEMORY ADDRESS TEST, MA6-5, 0-8 MB (CALLS TEST22CX)
* TEST22C9 - TEST_TNUM
* TEST22CA - WALKING 1'S MEMORY ADDRESS TEST, MA4-3, 0-8 MB (CALLS TEST22CX)
* TEST22CB - TEST_TNUM
* TEST22CC - WALKING 1'S MEMORY ADDRESS TEST, MA2-1, 0-8 MB (CALLS TEST22CX)
* TEST22CD - TEST_TNUM
* TEST22CE - WALKING 1'S MEMORY ADDRESS TEST, MA0-99, 0-8 MB (CALLS TEST22CX)
* TEST22CF - TEST_TNUM
* TEST22D0 - WALKING 1'S MEMORY ADDRESS TEST, MA98-97, 0-8 MB (CALLS TEST22CX)
* TEST22D1 - TEST_TNUM
* TEST22E0 - WALKING 1'S MEMORY ADDRESS TEST, MA14-13, 8-16 MB (CALLS TEST22CX)
* TEST22E1 - TEST_TNUM
* TEST22E2 - WALKING 1'S MEMORY ADDRESS TEST, MA12-11, 8-16 MB (CALLS TEST22CX)
* TEST22E3 - TEST_TNUM
* TEST22E4 - WALKING 1'S MEMORY ADDRESS TEST, MA10-9, 8-16 MB (CALLS TEST22CX)
* TEST22E5 - TEST_TNUM
* TEST22E6 - WALKING 1'S MEMORY ADDRESS TEST, MA8-7, 8-16 MB (CALLS TEST22CX)
* TEST22E7 - TEST_TNUM
* TEST22E8 - WALKING 1'S MEMORY ADDRESS TEST, MA6-5, 8-16 MB (CALLS TEST22CX)
* TEST22E9 - TEST_TNUM
* TEST22EA - WALKING 1'S MEMORY ADDRESS TEST, MA4-3, 8-16 MB (CALLS TEST22CX)
* TEST22EB - TEST_TNUM
* TEST22EC - WALKING 1'S MEMORY ADDRESS TEST, MA2-1, 8-16 MB (CALLS TEST22CX)
* TEST22ED - TEST_TNUM
* TEST22EE - WALKING 1'S MEMORY ADDRESS TEST, MA0-99, 8-16 MB (CALLS TEST22CX)
* TEST22EF - TEST_TNUM
* TEST22F0 - WALKING 1'S MEMORY ADDRESS TEST, MA98-97, 8-16 MB (CALLS TEST22CX)
* TEST22F1 - TEST_TNUM
* TEST2300 - WALKING 1'S MEMORY ADDRESS TEST, MA14-13, 16-24 MB (CALLS TEST22CX)
* TEST2301 - TEST_TNUM
* TEST2302 - WALKING 1'S MEMORY ADDRESS TEST, MA12-11, 16-24 MB (CALLS TEST22CX)
* TEST2303 - TEST_TNUM
* TEST2304 - WALKING 1'S MEMORY ADDRESS TEST, MA10-9, 16-24 MB (CALLS TEST22CX)
* TEST2305 - TEST_TNUM
* TEST2306 - WALKING 1'S MEMORY ADDRESS TEST, MA8-7, 16-24 MB (CALLS TEST22CX)
* TEST2307 - TEST_TNUM
* TEST2308 - WALKING 1'S MEMORY ADDRESS TEST, MA6-5, 16-24 MB (CALLS TEST22CX)
* TEST2309 - TEST_TNUM
* TEST230A - WALKING 1'S MEMORY ADDRESS TEST, MA4-3, 16-24 MB (CALLS TEST22CX)
* TEST230B - TEST_TNUM
* TEST230C - WALKING 1'S MEMORY ADDRESS TEST, MA2-1, 16-24 MB (CALLS TEST22CX)
* TEST230D - TEST_TNUM
* TEST230E - WALKING 1'S MEMORY ADDRESS TEST, MA0-99, 16-24 MB (CALLS TEST22CX)
* TEST230F - TEST_TNUM
* TEST2310 - WALKING 1'S MEMORY ADDRESS TEST, MA98-97, 16-24 MB (CALLS TEST22CX)
* TEST2311 - TEST_TNUM
* TEST2320 - WALKING 1'S MEMORY ADDRESS TEST, MA14-13, 24-32 MB (CALLS TEST22CX)
* TEST2321 - TEST_TNUM
* TEST2322 - WALKING 1'S MEMORY ADDRESS TEST, MA12-11, 24-32 MB (CALLS TEST22CX)
* TEST2323 - TEST_TNUM
* TEST2324 - WALKING 1'S MEMORY ADDRESS TEST, MA10-9, 24-32 MB (CALLS TEST22CX)
* TEST2325 - TEST_TNUM
* TEST2326 - WALKING 1'S MEMORY ADDRESS TEST, MA8-7, 24-32 MB (CALLS TEST22CX)
* TEST2327 - TEST_TNUM
* TEST2328 - WALKING 1'S MEMORY ADDRESS TEST, MA6-5, 24-32 MB (CALLS TEST22CX)
```

* TEST2329 – TEST_TNUM
* TEST232A – WALKING 1'S MEMORY ADDRESS TEST, MA4-3, 24-32 MB (CALLS TEST22CX)
* TEST232B – TEST_TNUM
* TEST232C – WALKING 1'S MEMORY ADDRESS TEST, MA2-1, 24-32 MB (CALLS TEST22CX)
* TEST232D – TEST_TNUM
* TEST232E – WALKING 1'S MEMORY ADDRESS TEST, MA0-99, 24-32 MB (CALLS TEST22CX)
* TEST232F – TEST_TNUM
* TEST2330 – WALKING 1'S MEMORY ADDRESS TEST, MA98-97, 24-32 MB (CALLS TEST22CX)
* TEST2331 – TEST_TNUM
* TEST2340 – WALKING 1'S MEMORY ADDRESS TEST, MA14-13, 32-40 MB (CALLS TEST22CX)
* TEST2341 – TEST_TNUM
* TEST2342 – WALKING 1'S MEMORY ADDRESS TEST, MA12-11, 32-40 MB (CALLS TEST22CX)
* TEST2343 – TEST_TNUM
* TEST2344 – WALKING 1'S MEMORY ADDRESS TEST, MA10-9, 32-40 MB (CALLS TEST22CX)
* TEST2345 – TEST_TNUM
* TEST2346 – WALKING 1'S MEMORY ADDRESS TEST, MA8-7, 32-40 MB (CALLS TEST22CX)
* TEST2347 – TEST_TNUM
* TEST2348 – WALKING 1'S MEMORY ADDRESS TEST, MA6-5, 32-40 MB (CALLS TEST22CX)
* TEST2349 – TEST_TNUM
* TEST234A – WALKING 1'S MEMORY ADDRESS TEST, MA4-3, 32-40 MB (CALLS TEST22CX)
* TEST234B – TEST_TNUM
* TEST234C – WALKING 1'S MEMORY ADDRESS TEST, MA2-1, 32-40 MB (CALLS TEST22CX)
* TEST234D – TEST_TNUM
* TEST234E – WALKING 1'S MEMORY ADDRESS TEST, MA0-99, 32-40 MB (CALLS TEST22CX)
* TEST234F – TEST_TNUM
* TEST2350 – WALKING 1'S MEMORY ADDRESS TEST, MA98-97, 32-40 MB (CALLS TEST22CX)
* TEST2351 – TEST_TNUM
* TEST2360 – WALKING 1'S MEMORY ADDRESS TEST, MA14-13, 40-48 MB (CALLS TEST22CX)
* TEST2361 – TEST_TNUM
* TEST2362 – WALKING 1'S MEMORY ADDRESS TEST, MA12-11, 40-48 MB (CALLS TEST22CX)
* TEST2363 – TEST_TNUM
* TEST2364 – WALKING 1'S MEMORY ADDRESS TEST, MA10-9, 40-48 MB (CALLS TEST22CX)
* TEST2365 – TEST_TNUM
* TEST2366 – WALKING 1'S MEMORY ADDRESS TEST, MA8-7, 40-48 MB (CALLS TEST22CX)
* TEST2367 – TEST_TNUM
* TEST2368 – WALKING 1'S MEMORY ADDRESS TEST, MA6-5, 40-48 MB (CALLS TEST22CX)
* TEST2369 – TEST_TNUM
* TEST236A – WALKING 1'S MEMORY ADDRESS TEST, MA4-3, 40-48 MB (CALLS TEST22CX)
* TEST236B – TEST_TNUM
* TEST236C – WALKING 1'S MEMORY ADDRESS TEST, MA2-1, 40-48 MB (CALLS TEST22CX)
* TEST236D – TEST_TNUM
* TEST236E – WALKING 1'S MEMORY ADDRESS TEST, MA0-99, 40-48 MB (CALLS TEST22CX)
* TEST236F – TEST_TNUM
* TEST2370 – WALKING 1'S MEMORY ADDRESS TEST, MA98-97, 40-48 MB (CALLS TEST22CX)
* TEST2371 – TEST_TNUM
* TEST2380 – WALKING 1'S MEMORY ADDRESS TEST, MA14-13, 48-56 MB (CALLS TEST22CX)
* TEST2381 – TEST_TNUM
* TEST2382 – WALKING 1'S MEMORY ADDRESS TEST, MA12-11, 48-56 MB (CALLS TEST22CX)
* TEST2383 – TEST_TNUM
* TEST2384 – WALKING 1'S MEMORY ADDRESS TEST, MA10-9, 48-56 MB (CALLS TEST22CX)
* TEST2385 – TEST_TNUM
* TEST2386 – WALKING 1'S MEMORY ADDRESS TEST, MA8-7, 48-56 MB (CALLS TEST22CX)
* TEST2387 – TEST_TNUM
* TEST2388 – WALKING 1'S MEMORY ADDRESS TEST, MA6-5, 48-56 MB (CALLS TEST22CX)
* TEST2389 – TEST_TNUM
* TEST238A – WALKING 1'S MEMORY ADDRESS TEST, MA4-3, 48-56 MB (CALLS TEST22CX)
* TEST238B – TEST_TNUM
* TEST238C – WALKING 1'S MEMORY ADDRESS TEST, MA2-1, 48-56 MB (CALLS TEST22CX)
* TEST238D – TEST_TNUM
* TEST238E – WALKING 1'S MEMORY ADDRESS TEST, MA0-99, 48-56 MB (CALLS TEST22CX)
* TEST238F – TEST_TNUM

```
* TEST2390 - WALKING 1'S MEMORY ADDRESS TEST, MA98-97, 48-56 MB (CALLS TEST22CX)
* TEST2391 - TEST_TNUM
* TEST23A0 - WALKING 1'S MEMORY ADDRESS TEST, MA14-13, 56-64 MB (CALLS TEST22CX)
* TEST23A1 - TEST_TNUM
* TEST23A2 - WALKING 1'S MEMORY ADDRESS TEST, MA12-11, 56-64 MB (CALLS TEST22CX)
* TEST23A3 - TEST_TNUM
* TEST23A4 - WALKING 1'S MEMORY ADDRESS TEST, MA10-9, 56-64 MB (CALLS TEST22CX)
* TEST23A5 - TEST_TNUM
* TEST23A6 - WALKING 1'S MEMORY ADDRESS TEST, MA8-7, 56-64 MB (CALLS TEST22CX)
* TEST23A7 - TEST_TNUM
* TEST23A8 - WALKING 1'S MEMORY ADDRESS TEST, MA6-5, 56-64 MB (CALLS TEST22CX)
* TEST23A9 - TEST_TNUM
* TEST23AA - WALKING 1'S MEMORY ADDRESS TEST, MA4-3, 56-64 MB (CALLS TEST22CX)
* TEST23AB - TEST_TNUM
* TEST23AC - WALKING 1'S MEMORY ADDRESS TEST, MA2-1, 56-64 MB (CALLS TEST22CX)
* TEST23AD - TEST_TNUM
* TEST23AE - WALKING 1'S MEMORY ADDRESS TEST, MA0-99, 56-64 MB (CALLS TEST22CX)
* TEST23AF - TEST_TNUM
* TEST23B0 - WALKING 1'S MEMORY ADDRESS TEST, MA98-97, 56-64 MB (CALLS TEST22CX)
* TEST23B1 - TEST_TNUM
* TEST23C0 - WRITE ABORT ON MA PARITY ERROR TEST
* TEST23C1 - BACK TO BACK 32 BIT WRITE TEST
* TEST23C2 - BACK TO BACK 16 BIT WRITE TEST
* TEST23C3 - RANDOM 16 BIT WRITE TEST
* TEST23C4 - TEST_TNUM
```

Note that the section on system initialization detatils the tests covered in each SYSV overlay. Refer to this section for the index of tests for the remaining UDIAGS. The SYSVs are organized as follows: Two overlays are dedicated primarily to the E unit and 2 overlays are mainly IS unit tests. A fifth overlay tests the MC unit and memory. The sixth overlay tests all the leftovers, including I/O. Each overlay is self contained, and executes the Kernel routines before starting the tests. All tests are capable of running by themselves (no previous tests need be run to set up conditions). Optimal Replaceable Units (ORUs) are defined to the board level. A variety of tests are included to validate the operations of individual VLSI chips.

The 6 SYSV overlays in the 4150 are an improvement over the 9755's 2 SYSV overlays. Areas of the machine which have significantly better coverage include the E unit, the MC, and overall error reporting.

15. VLSI Requirements

15.1 Introduction

The following VLSI parts are used in the 4150:

1. Microsequencer
2. Cache Address
3. STLB Set Select
4. Cache Set Select
5. Execution ALU
6. Barrel Shifter
7. Write Buffer Address
8. Register File Address

The mnemonic for each part has a "P" prepended to distinguish it from its cousin on the 6350. Each is briefly described in turn in the sections below. For more detailed information, refer to the individual part specifications.

15.2 Microsequencer

The microsequencer (PUSEQ) is used to control the sequencing of the control store (CS). The PUSEQ's BCY (Bus Control Address) outputs are used to address the CS RAMs.

The PUSEQ is partitioned into two identical slices. Each slice produces 7 of the 14 bits necessary to address the CS. Each slice also produces decode net address bits. Both slices reside on the CMI board.

The PUSEQ was designed by Ed Karaian.

15.3 Cache Address

The Cache Address Chip (PCADR) contains logic to maintain and source the I, branch cache, cache, and S units' internal memory address bus. This bus is used to produce the address for all memory related accesses on the IS board. These include cache address, STLB address, branch cache address, and backup copies of the effective address (ERMA) and program counter (PRMA) registers. The PCADR also supports Effective Address Formation.

PCADR is partitioned into two identical slices. One slice produces the low order bytes of both BVMAH and BVMAL, while the other slice produces the high order bytes of those same busses. Both PCADR slices reside on the IS board.

The PCADR was designed by Tony Dorohov.

15.4 STLB Set Select

The Segment Translation Lookaside Buffer (STLB) contains previously translated physical memory addresses. The 4150 uses a two set STLB. The STLB Set Select (PSSS) chip is responsible for determining which set, if either, has a valid address translation for the current memory reference. If neither set has such a valid translation a memory trap is raised, and the processor's microcode has to determine what to do about the memory access.

The PSSS is partitioned into two identical slices. Each slice watches one set of the STLB for valid translations and signals its findings on its outputs. The IS board on which the PSSS chips sit is responsible for monitoring these outputs and taking the translation from the valid set. Both slices reside on the IS board.

The PSSS was designed by Mark Mason.

15.5 Cache Set Select

The 4150 uses a two set data cache similar in principle to the two set STLB. The Cache Set Select (PCSS) chip watches the cache to determine which set, if either, has valid data, and to signal a cache miss if neither set is valid. The PCSS also drives BB with the required source as specified by the microcode, aligning data properly on the fly.

The PCSS is partitioned into two identical slices. Each slice watches one set of the cache and signals its findings on its outputs. The IS board on which the PCSS chips sit is responsible for making the final hit/miss decision, and starting a cache miss routine on the MC if necessary. Both slices reside on the IS board.

15.6 PEALU

The 4150 E unit contains a 48-bit data path which can be extended to 56-bits for multiply and divide operations. The E unit ALU (PEALU) chip provides the full function 2-input ALU which performs the number crunching functions of the data path. Look ahead carry logic and carry-save adder logic is also provided by these chips.

The 56-bit data path is made up of 7 identical PEALU slices, each 8-bits wide. All 7 slices reside on the E board.

The PEALU was designed by Stu Rae.

15.7 Barrel Shifter

The Bus D Internal (PBDI) chip provides full shifting (arithmetic, logical, and barrel), rotating, normalizing of floating point numbers, floating point adjusting, and decimal packing and unpacking, all over a full 48-bits. It also provides guard bit support for multiplication.

The PBDI is implemented in three identical slices. Each slice produces 16 outputs which are driven to the rest of the machine. All three slices reside on the E board.

15.8 Write Buffer Address

The write buffer attempts to make better use of the memory bandwidth by saving 16-bit and 32-bit writes from the CPU and trying to concatenate them into 64-bit writes before actually doing the write to memory. The ADdress BUFfer (PADBUF) chip provides the address and valid bit memories necessary for such a buffer.

The PADBUF is implemented in two identical slices, each of which drives half of the memory address bus. Status of the write buffer as a whole can be determined from either slice. Both slices reside on the CMI board.

The PADBUF was designed by Steve Small.

15.9 Register File Address

The Register File Address (PRFADR) chip provides an 8-bit address to the E unit's register file. The addresses are determined from RCM bits, BPA bits, DTAR bits, BBH bits, base register select bits, or BOPS bits. The PRFADR chip can handle all of these cases. The skip net and the RP and REC counters are also implemented by the PRFADR.

Only one PRFADR slice is necessary to perform all these functions. The slice resides on the E board.

The PRFADR was designed by John Strusienski and John Wishart.

16. Diagnostic Processor Interface Detailed Description

The interface between the Diagnostic Processor (DP) and the CPU has been changed from a serial port, as it existed on prior CPUs, to a parallel port. The parallel port will allow 8 bits of information plus parity to be transmitted simultaneously. The data and control lines involved in the communication between the DP and the CPU are shown in Figure 16-1.

FIG. 16-1. Diagnostic Processor Interface Signals

```
DPFULL+A        \
DPREQ+           \
                  > CONTROL LINES
CPUACK+A         /
CPUREQ+A        /

DPDAT{00:07}+A   \
                  > DATA LINES
DPPAR+A          /
```

The interface has two modes of operation:

1. When the CPU is running two way communication is used.
2. When SYSCLR- is asserted the DP has direct control over the interface. One way communication is used in this case.

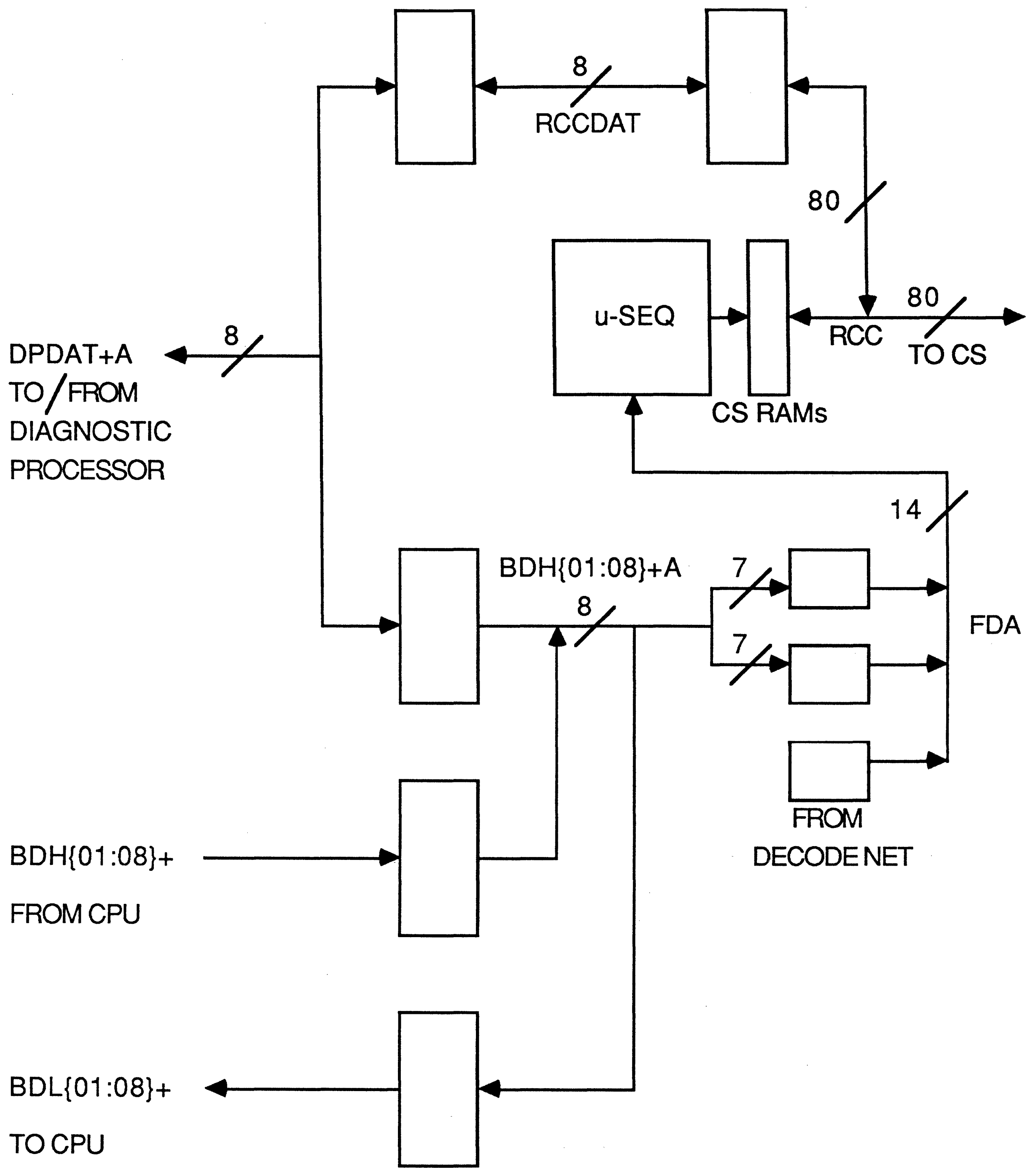


Figure 16-2 Diagnostic Processor Interface Block Diagram

16.1 DP Two Way Communication

When sending data from the DP to the CPU, the DP data is placed on BDL{01:08}+. When sending data from the CPU to the DP, the data is placed on BDH{01:08}+.

To send data to the CPU from the DP, the following handshaking takes place (illustrated in Figure 16-3):

1. The DP raises DPREQ+ which causes a fetch cycle trap on the CPU.
2. After taking a fetch cycle trap and detecting DPREQ+, the CPU raises CPUACK+A.
3. The DP resets DPREQ+ and places the data on DPDAT{00:07}+A. The DP keeps the data on the bus until CPUACK+ is reset by the CPU.
4. The DP data is buffered and placed on BDH{01:08}+A. It is then driven onto BDL{01:08}+ where it proceeds to its destination.
5. The CPU resets CPUACK+.

The CPU may send data to the DP for display on the system console. To send data from the CPU to the DP, the following handshaking takes place (illustrated in Figure 16-4):

1. The program wishing to send data to the system console executes a PIO instruction. This instruction causes the DP to assert DPREQ+, producing a fetch cycle trap in the CPU as before.
2. The CPU examines DPFULL+A sent out by the DP which tells the CPU if the DP's input buffer is empty or full. If this signal is low the CPU raises CPUREQ+A and places its data on BDH{01:08}+. The data is buffered BDH{01:08}+A and driven onto the DP bus DPDAT{00:07}+A.
3. After receiving CPUREQ+A, the DP sets DPFULL+ active. After an appropriate amount of time, the CPU resets CPUREQ+A and takes the data off the bus. The falling edge of CPUREQ+A latches the data onto the DP, so the CPU must guarantee hold time.
4. After reading the data out of its buffer the DP resets DPFULL+.

In both transmitting modes mentioned above the CPU controlled when the data was put on the bus. If the DP and the CPU both have data to send to each other it is up to the CPU to determine the order of operations. Parity is checked on BDH{01:08}+A along with the transmitted parity bits. The result of the parity checker is sent directly to the jump net. This signal will only be valid when data is being held on the bus, and at other times will not be guaranteed.

16.2 DP One Way Communication

This mode of operation takes place when SYSCLR- is asserted. In this case, the DP drives all four control lines. The CPU decodes three of these lines and uses the fourth as a clock signal. Table 16-1 shows the encodings.

TABLE 16-1. DP Interface One-Way Communication Commands

Operation	CPUACK+A CNT01	DPFULL+A CNT02	CPUREQ+A CNT03	DPREQ+ CLOCK
No-op	1	1	1	↑
Load RBCYL	1	0	1	↑
Load RBCYH	1	0	0	↑
Read CS	0	1	1	↑
Write CS	1	1	0	↑

The purpose of the three control lines is to set up the data path so the data on DPDAT{00:07}+A can get clocked into the correct destination when the fourth control line goes active.

To write control store the following sequence of operations takes place. NOTE: The clock line is held in the high state when the control lines are changed. Also, the clock line must be valid when SYSCLR- is asserted so that miscellaneous clocks do not occur.

1. The control lines are set to load RBCY high or low.
2. The data to load RBCY with is put on the data bus.
3. The fourth control line is toggled.
4. The control lines are changed to load the other half of RBCY, and steps 2 and 3 are repeated.
5. RBCY contains the address of the current control store location that is going to be written. The control lines can now be set to Write CS.
6. The clock line is toggled. This step starts off the write pulse timing chain and does not need to have correct data on the DP bus.
7. The correct data for the first 8 bits of the 80 bit microcode word are be placed on the bus and the clock line toggled.
8. Repeat step 7 for the second 8 bits. Continue until all 80 bits are written.
9. At this point one microcode word has been written and it is time to change the RBCY address. Repeat steps 1-8 until the entire control store has been written.

Note that the entire 80 bit microcode word must be written each time. There is no support for writing selective fields in the microcode.

The loading of RBCY and the writing of the control store is illustrated in Figures 16-5 and 16-6.

It is important to understand that when the control lines are changing the data paths are also changing. This means the CPU could be driving the DP bus at the same time the DP is driving the bus. Therefore, when the DP changes the control lines the DP should not be driving the DP bus.

Reading control store is very similar to writing it. After loading RBCY, the control lines in step 5 are set to Read CS. The CPU is now driving the DP bus with the first 8 bits of the current control store location. When the fourth control line is toggled (in step 7) the second 8 bits of the control store are presented to the DP. This operation continues until the last 8 bits of the 80 bit microcode word are sent back to the DP. Now RBCY can be loaded with the next microcode word address and the above process repeated.

16.2.1 PDA Acting as the DP

The above operations are done exactly the same way for the PDA to load and verify the control store. The only difference is that the PDA asserts GDPOFF+, which causes the DP to get off the DP bus so that the PDA can drive it. The CPU does not have any knowledge of who it is getting the data from, so the PDA and DP must perform the operations the same way. Also, note that the PDA only needs this interface to read and write control store. Once the CPU is running the PDA does not need the diagnostic processor interface.

The above control signals are all TTL open collector lines. The DP bus itself is a tristate bus.

16.3 VLSI Usage

There are no VLSI chips used in the DP interface.

16.4 Timing Diagrams

FIG. 16-3. Read Data from DP to CPU

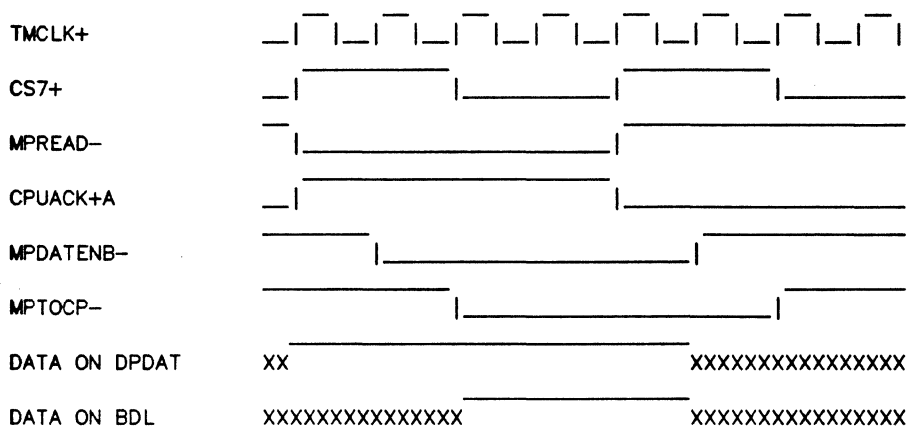


FIG. 16-4. Write Data from CPU to DP

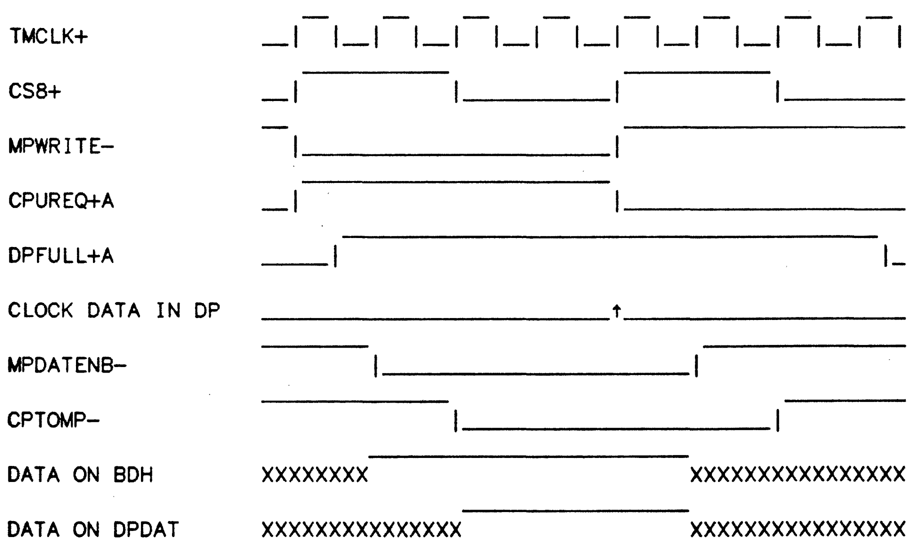


FIG. 16-5. Load RBCYH & RBCYL

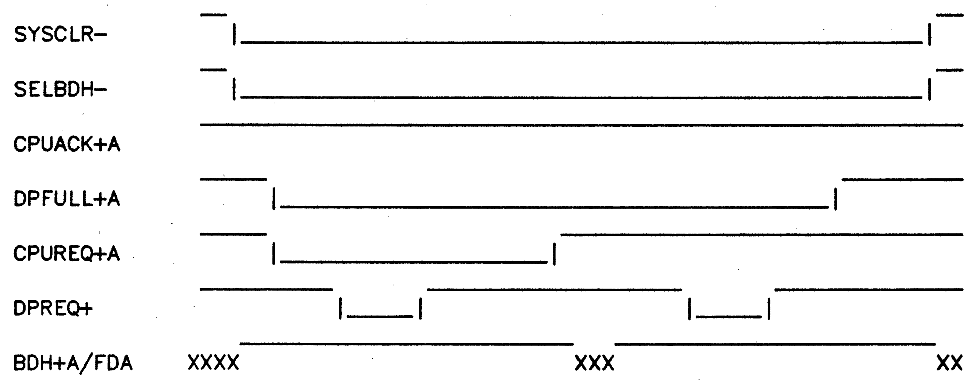
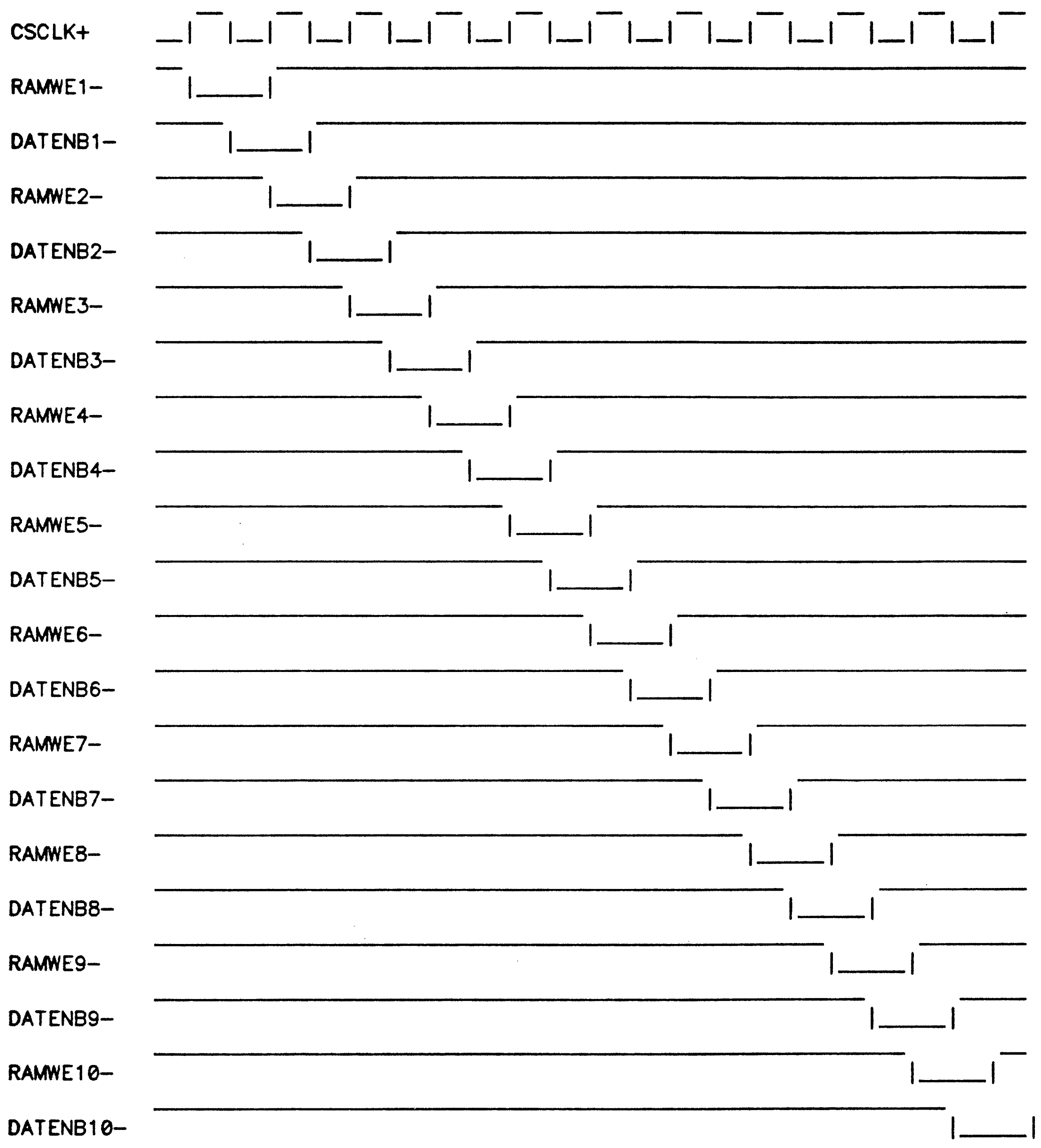


FIG. 16-6. Write Control Store



16.5 9755 Comparisons

The DP interface is implemented as a parallel bus in the 4150 as opposed to the serial bus of the 9755. All control logic is, therefore, completely different.

The 9755 DP is a Weasel board, while the 4150 uses a Mink board.

16.6 Critical Paths

There are no critical paths in the DP interface. This rather unusual claim can be made because the entire interface is under microcode control during two way communication, allowing timing to be fixed with TXs and/or microcode control during two way communication and each step has TXs and/or NXs. During one way communication, SYSCLR- is active, and the timing is based on information received from the DP, which is under a Z80 microprocessor control. The Z80's timing is much slower than the 4150's.

16.7 Partitioning

The DP interface logic resides entirely on the CMI board, which is discussed in chapter 30.

17. System Initialization Detailed Description

17.1 SYSVERIFY

The purpose of Sysverify is to guarantee the integrity of the system at coldstart prior to booting. In the event some failure is detected, the Sysverify package allows intervention through the Diagnostic Processor (DP) in order to more easily determine what hardware failure has occurred.

The Sysverify diagnostics are configured into six overlays. Normal execution sequences through all the overlays with no intervention. At power on, the DP loads in and starts SYSV1 executing. When this has completed, the overlay sends the DP a 45 hex to inform it that the overlay has completed. The DP acknowledges and then loads and begins execution of the next Sysverify overlay.

Completion of the final overlay will cause the DP to load functional microcode and the decode net, and then begin the Boot procedure.

17.1.1 Sysverify Overlays

The SYSV1 overlay is mainly responsible for testing the register file, the Pipeline Control Unit (PCU), Register Event Counter (REC) and Register Program counter Low (RPL), register file addressing and address traps. Most of the Execution (E) unit ALU modes and many of the E unit Jump Conditions (JCs) are verified on a slice by slice basis. The following is an index of the tests contained in SYSV1.

```
*
*    INDEX OF TESTS IN SYSV1
*    -----------------------
*
*    TEST0114 - TEST OF RDL, ALL, AND BD SWAP MODE
*    TEST0115 - TEST OF RDE, ALE, AND BD SWAP MODE.
*    TEST0116 - TEST OF RSL
*    TEST0117 - TEST OF RSE, ALE
*    TEST0118 - TEST OF RFL.
*    TEST0119 - TEST OF RFE.
*    TEST011A - NORSALU9 CLOCK
*    TEST011B - RSUM(IBB) - HIGH SIDE
*    TEST011C - RSUM(IBB) - LOW SIDE
*    TEST011D - RSUM(IBB) - EXTENDED SIDE
*    TEST011E - NORSCLK (BDI PATH)
*    TEST011F - TEST OF RDH LEFT SHIFT LOW BIT = 0
*    TEST0120 - TEST OF RDH LEFT SHIFT LOW BIT = 1
*    TEST0121 - RDE LOADED FROM ALU WITH LEFT SHIFT
*    TEST0122 - RDL LOADED FROM ALU WITH LEFT SHIFT
*    TEST0123 - TEST OF RDL LEFT SHIFT LOW BIT = 0
*    TEST0124 - TEST OF RDL LEFT SHIFT LOW BIT = 1
*    TEST0125 - TEST OF RDE LEFT SHIFT LOW BIT = 0
*    TEST0126 - TEST_TNUM
*    TEST0129 - TESTS THAT THE U_TIMER CAN BE LOADED WITH A ZERO
*    TEST012A - TESTS THAT THE U_TIMER CAN BE LOADED WITH ALL ONES
*   TEST012B - TESTS THE U_TIMER WITH WALKING ONES
*   TEST012C - TEST OF TX OF 1
*   TEST012D - TEST OF TX OF 2
*   TEST012E - TEST OF TX OF 3
*   TEST012F - TEST OF TX OF 4
*   TEST0130 - TEST OF NX OF 2
*   TEST0131 - TEST OF NX OF 4
*   TEST0132 - TEST OF TX 0 NX 0
*   TEST0133 - TEST OF TX = 1, AND NX = 2
*   TEST0134 - TEST OF TX = 1, AND NX = 4
*   TEST0135 - TEST OF TX = 1, AND NX = 6
*   TEST0136 - TEST OF TX = 2, NX = 2
*   TEST0137 - TEST OF TX = 2, NX = 4
*   TEST0138 - TEST OF TX = 5, NX = 2
*   TEST0139 - TEST OF TX = 3, NX = 2
*   TEST013A - TEST_TNUM
*   TEST0150 - TEST OF REC.
*   TEST0151 - TEST EMIT TO REC
*   TEST0152 - TEST EMIT ZERO TO REC
*   TEST0153 - TEST EMIT $5555 TO REC
*   TEST0154 - TEST EMIT $AAAA TO REC
*   TEST0155 - TEST FOR REC INCREMENT
*   TEST0156 - TEST REC DECREMENT
*   TEST0157 - TEST JUMP ON FRECC13 TRUE
*   TEST0158 - FRECC13 NOT
*   TEST0159 - FRECOUT TRUE
*   TEST015A - FRECOUT NOT
*   TEST015B - FNRECOUT FALSE
*   TEST015C - FNRECOUT TRUE
*   TEST015D - REC INC/DEC COMBINATIONS
*   TEST015E - REC06 TRUE
*   TEST015F - REC06 FALSE
*   TEST0160 - NREC13 FALSE
*   TEST0161 - NREC13 TRUE
*   TEST0162 - TEST_TNUM
*   TEST0163 - LOAD RPL
*   TEST0164 - INCREMENT RPL BY 1
*   TEST0165 - INCREMENT RPL BY 2
*   TEST0166 - TEST OF REC INC BY '200
*   TEST0167 - TEST OF REC DEC BY '200
*   TEST0168 - TEST OF FRECOV = 1, INC BY 1
*   TEST0169 - TEST OF FRECOV = 1, DEC BY 1
*   TEST016A - TEST OF FRECOV = 1, INC BY '200
*   TEST016B - TEST OF FRECOV = 1, DEC BY '200
*   TEST016C - TEST OF FRECOV = 0, INC
*   TEST016D - TEST OF FRECOV = 0, DEC
*   TEST016E - TEST OF FRECOV = 0, INC BY '200
*   TEST016F - TEST OF FRECOV = 0, DEC BY '200
*   TEST0170 - TEST OF RBRP1 AND RBRP2 = 0
*   TEST0171 - TEST OF RBRP1 AND RBRP2 = 01
*   TEST0172 - TEST OF RBRP1 AND RBRP2 = 10, INCRPBY1,INCRPBY1
*   TEST0173 - TEST OF RBRP1 AND RBRP2 = 10, INCRPBY1,INCRPBY1
*   TEST0174 - TEST OF RBRP1 AND RBRP2 = 11, INCRPBY1 3 TIMES
*   TEST0175 - TEST OF RBRP1 AND RBRP2 = 11, INCRPBY1,INCRPBY1, INCRPBY1
*   TEST0176 - TEST OF RBRP1 AND RBRP2 = 11, INCRPBY1, INRPBY2
*   TEST0177 - TEST OF OLDRPBU1 AND OLDRPBU2 = 0
*   TEST0178 - TEST OF OLDRPBU1 = 1
*   TEST0179 - TEST OF OLDRPBU2 = 1
*   TEST017A - TEST_TNUM
*   TEST0180 - WALK ONES THROUGH RFH
*   TEST0181 - WALK ONES THROUGH RFL
```

* TEST0182 – WALK ONES THROUGH RFE
* TEST0183 – WALK ZEROES THROUGH RFH
* TEST0184 – WALK ZEROES THROUGH RFL
* TEST0185 – WALK ZEROES THROUGH RFE
* TEST0186 – WALKING ONES THROUGH THE ADDRESS (RFH)
* TEST0187 – WALKING ONES THROUGH THE ADDRESS (RFL)
* TEST0188 – WALKING ONES THROUGH THE ADDRESS (RFE)
* TEST0189 – WALKING ZEROES THROUGH THE ADDRESS (RFH)
* TEST018A – WALKING ZEROES THROUGH THE ADDRESS (RFL)
* TEST018B – WALKING ZEROES THROUGH THE ADDRESS (RFE)
* TEST018C – WRITE RECOVERY (RFH)
* TEST018D – WRITE RECOVERY (RFL)
* TEST018E – WRITE RECOVERY (RFE)
* TEST018F – READ ACCESS (RFH)
* TEST0190 – READ ACCESS (RFL)
* TEST0191 – READ ACCESS (RFE)
* TEST0192 – CONTENTS EQUALS ADDRESS (RFH)
* TEST0193 – CONTENTS EQUALS ADDRESS (RFL)
* TEST0194 – CONTENTS EQUALS ADDRESS (RFE)
* TEST0195 – RFH BYPASS
* TEST0196 – RFL BYPASS
* TEST0197 – RFE BYPASS
* TEST0198 – INVOKE A BYPASS DECISION WHICH DOES NOT BYPASS (RFL)
* TEST0199 – TEST_TNUM
* TEST01A0 – TR0 – LOGICAL TO PHYSICAL MAPPING
* TEST01A1 – TR1 – LOGICAL TO PHYSICAL MAPPING
* TEST01A2 – TR2 – LOGICAL TO PHYSICAL MAPPING
* TEST01A3 – TR3 – LOGICAL TO PHYSICAL MAPPING
* TEST01A4 – TR4 – LOGICAL TO PHYSICAL MAPPING
* TEST01A5 – TR5 – LOGICAL TO PHYSICAL MAPPING
* TEST01A6 – TR6 – LOGICAL TO PHYSICAL MAPPING
* TEST01A7 – TR7 – LOGICAL TO PHYSICAL MAPPING
* TEST01A8 – TR8 – LOGICAL TO PHYSICAL MAPPING
* TEST01A9 – TR9 – LOGICAL TO PHYSICAL MAPPING
* TEST01AA – TR10 – LOGICAL TO PHYSICAL MAPPING
* TEST01AB – TR11 – LOGICAL TO PHYSICAL MAPPING
* TEST01AC – REOIV – LOGICAL TO PHYSICAL MAPPING
* TEST01AD – RDSAVE – LOGICAL TO PHYSICAL MAPPING
* TEST01AE – CFF00 – LOGICAL TO PHYSICAL MAPPING
* TEST01AF – RATMP – LOGICAL TO PHYSICAL MAPPING
* TEST01B0 – RMASAVE – LOGICAL TO PHYSICAL MAPPING
* TEST01B1 – PARREG0 – LOGICAL TO PHYSICAL MAPPING
* TEST01B2 – PARREG1 – LOGICAL TO PHYSICAL MAPPING
* TEST01B3 – PARREG2 – LOGICAL TO PHYSICAL MAPPING
* TEST01B4 – PARREG3 – LOGICAL TO PHYSICAL MAPPING
* TEST01B5 – PBSAVE – LOGICAL TO PHYSICAL MAPPING
* TEST01B6 – SYSREG1 – LOGICAL TO PHYSICAL MAPPING
* TEST01B7 – DSWPARITY – LOGICAL TO PHYSICAL MAPPING
* TEST01B8 – PSWPB – LOGICAL TO PHYSICAL MAPPING
* TEST01B9 – PSWKEYS – LOGICAL TO PHYSICAL MAPPING
* TEST01BA – PLA – LOGICAL TO PHYSICAL MAPPING
* TEST01BB – PLB – LOGICAL TO PHYSICAL MAPPING
* TEST01BC – DSWRMA – LOGICAL TO PHYSICAL MAPPING
* TEST01BD – DSWSTAT – LOGICAL TO PHYSICAL MAPPING
* TEST01BE – DSWPB – LOGICAL TO PHYSICAL MAPPING
* TEST01BF – RSAVPTR – LOGICAL TO PHYSICAL MAPPING
* TEST01C0 – FERRET1 – LOGICAL TO PHYSICAL MAPPING
* TEST01C1 – DSWITCH – LOGICAL TO PHYSICAL MAPPING
* TEST01C2 – TODBUF – LOGICAL TO PHYSICAL MAPPING
* TEST01C3 – LIGHTS – LOGICAL TO PHYSICAL MAPPING
* TEST01C4 – ACK1 – LOGICAL TO PHYSICAL MAPPING

```
*   TEST01C5 - INTVEC - LOGICAL TO PHYSICAL MAPPING
*   TEST01C6 - ACK2 - LOGICAL TO PHYSICAL MAPPING
*   TEST01C7 - C6666 - LOGICAL TO PHYSICAL MAPPING
*   TEST01C8 - PICSTAT - LOGICAL TO PHYSICAL MAPPING
*   TEST01C9 - C0080 - LOGICAL TO PHYSICAL MAPPING
*   TEST01CA - CB0B0 - LOGICAL TO PHYSICAL MAPPING
*   TEST01CB - C8000 - LOGICAL TO PHYSICAL MAPPING
*   TEST01CC - C3F - LOGICAL TO PHYSICAL MAPPING
*   TEST01CD - IUART - LOGICAL TO PHYSICAL MAPPING
*   TEST01CE - ONE - LOGICAL TO PHYSICAL MAPPING
*   TEST01CF - MINUS1 - LOGICAL TO PHYSICAL MAPPING
*   TEST01D0 - DGR17 - LOGICAL TO PHYSICAL MAPPING
*   TEST01D1 - DGR16 - LOGICAL TO PHYSICAL MAPPING
*   TEST01D2 - DGR15 - LOGICAL TO PHYSICAL MAPPING
*   TEST01D3 - DGR14 - LOGICAL TO PHYSICAL MAPPING
*   TEST01D4 - DGR13 - LOGICAL TO PHYSICAL MAPPING
*   TEST01D5 - DGR12 - LOGICAL TO PHYSICAL MAPPING
*   TEST01D6 - DGR11 - LOGICAL TO PHYSICAL MAPPING
*   TEST01D7 - DGR10 - LOGICAL TO PHYSICAL MAPPING
*   TEST01D8 - DGR7 - LOGICAL TO PHYSICAL MAPPING
*   TEST01D9 - DGR6 - LOGICAL TO PHYSICAL MAPPING
*   TEST01DA - DGR5 - LOGICAL TO PHYSICAL MAPPING
*   TEST01DB - DGR4 - LOGICAL TO PHYSICAL MAPPING
*   TEST01DC - DGR3 - LOGICAL TO PHYSICAL MAPPING
*   TEST01DD - DGR2 - LOGICAL TO PHYSICAL MAPPING
*   TEST01DE - DGR1 - LOGICAL TO PHYSICAL MAPPING
*   TEST01DF - DGR0 - LOGICAL TO PHYSICAL MAPPING
*   TEST01E0 - CURRENT REGISTER SET MODALS (CRS = 0)
*   TEST01E1 - CURRENT REGISTER SET MODALS (CRS = 1)
*   TEST01E2 - CURRENT REGISTER SET MODALS (CRS = 2)
*   TEST01E3 - CURRENT REGISTER SET MODALS (CRS = 3)
*   TEST01E4 - RF32 READ & WRITE
*   TEST01E5 - RF48 READ & WRITE (CHECKING LOW SIDE)
*   TEST01E6 - RF48 READ & WRITE (CHECKING EXTENDED SIDE)
*   TEST01E7 - TEST_TNUM
*   TEST01F0 - GR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01F1 - GR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01F2 - GR2 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01F3 - GR3 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01F4 - GR4 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01F5 - GR5 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01F6 - GR6 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01F7 - GR7 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01F8 - FAR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01F9 - FLR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01FA - FAR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01FB - FLR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01FC - PB - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01FD - SB - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01FE - LB - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST01FF - XB - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST0200 - DTAR3 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST0201 - DTAR2 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST0202 - DTAR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST0203 - DTAR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST0204 - KEYS - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST0205 - OWNER - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST0206 - FCODE - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST0207 - FADDR - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST0208 - RTIMER - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST0209 - CR31 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST020A - CR32 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST020B - CR33 - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST020C - TPB - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST020D - TSB - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST020E - TLB - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST020F - XEQPB - LOGICAL TO PHYSICAL MAPPING (CRS = 0)
*   TEST0210 - GR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0211 - GR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0212 - GR2 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0213 - GR3 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0214 - GR4 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0215 - GR5 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0216 - GR6 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0217 - GR7 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0218 - FAR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0219 - FLR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST021A - FAR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST021B - FLR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST021C - PB - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST021D - SB - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST021E - LB - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST021F - XB - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0220 - DTAR3 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0221 - DTAR2 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0222 - DTAR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0223 - DTAR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0224 - KEYS - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0225 - OWNER - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0226 - FCODE - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0227 - FADDR - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0228 - RTIMER - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0229 - CR31 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST022A - CR32 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST022B - CR33 - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST022C - TPB - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST022D - TSB - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST022E - TLB - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST022F - XEQPB - LOGICAL TO PHYSICAL MAPPING (CRS = 1)
*   TEST0230 - GR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0231 - GR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0232 - GR2 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0233 - GR3 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0234 - GR4 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0235 - GR5 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0236 - GR6 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0237 - GR7 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0238 - FAR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0239 - FLR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST023A - FAR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST023B - FLR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST023C - PB - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST023D - SB - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST023E - LB - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST023F - XB - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0240 - DTAR3 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0241 - DTAR2 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0242 - DTAR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0243 - DTAR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0244 - KEYS - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0245 - OWNER - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0246 - FCODE - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0247 - FADDR - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0248 - RTIMER - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0249 - CR31 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST024A - CR32 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST024B - CR33 - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST024C - TPB - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST024D - TSB - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST024E - TLB - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST024F - XEQPB - LOGICAL TO PHYSICAL MAPPING (CRS = 2)
*   TEST0250 - GR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0251 - GR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0252 - GR2 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0253 - GR3 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0254 - GR4 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0255 - GR5 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0256 - GR6 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0257 - GR7 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0258 - FAR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0259 - FLR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST025A - FAR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST025B - FLR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST025C - PB - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST025D - SB - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST025E - LB - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST025F - XB - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0260 - DTAR3 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0261 - DTAR2 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0262 - DTAR1 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0263 - DTAR0 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0264 - KEYS - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0265 - OWNER - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0266 - FCODE - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0267 - FADDR - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0268 - RTIMER - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0269 - CR31 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST026A - CR32 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST026B - CR33 - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST026C - TPB - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST026D - TSB - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST026E - TLB - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST026F - XEQPB - LOGICAL TO PHYSICAL MAPPING (CRS = 3)
*   TEST0270 - TEST_TNUM
*   TEST0276 - TEST OF ADDRESS TRAP, X REGISTER
*   TEST0277 - TEST OF ADDRESS TRAP, GR2H REGISTER
*   TEST0278 - TEST OF ADDRESS TRAP, GR2L
*   TEST0279 - TEST OF ADDRESS TRAP, Y REGISTER
*   TEST027A - TEST OF ADDRESS TRAP, FAR1H REGISTER
*   TEST027B - TEST OF ADDRESS TRAP, FAR1L REGISTER
*   TEST027C - TEST OF ADDRESS TRAP, FLR1L REGISTER
*   TEST027D - TEST_TNUM
*   TEST027E - TEST OF ADDRESS TRAP, DTAR3H REGISTER
*   TEST027F - TEST OF ADDRESS TRAP, FCODEH REGISTER
*   TEST0280 - TEST OF ADDRESS TRAP, FADDRL REGISTER
*   TEST0281 - TEST OF ADDRESS TRAP, FAR0H REGISTER
*   TEST0282 - TEST OF ADDRESS TRAP, SBH REGISTER
*   TEST0283 - TEST OF ADDRESS TRAP, SBL REGISTER
*   TEST0284 - TEST OF ADDRESS TRAP, LBH REGISTER
*   TEST0285 - TEST OF ADDRESS TRAP, LBL REGISTER
*   TEST0286 - TEST OF ADDRESS TRAP, DMX ADDR '60 HIGH
*   TEST0287 - TEST OF ADDRESS TRAP, DMX ADDR '60 LOW
*   TEST0288 - TEST_TNUM
*    TEST028A - TEST OF ADDRESS TRAP, DMX ADDR '62 HIGH
*    TEST028B - TEST OF ADDRESS TRAP, DMX ADDR '62 LOW
*    TEST028C - TEST OF ADDRESS TRAP, DMX ADDR '64 HIGH
*    TEST028D - TEST OF ADDRESS TRAP, DMX ADDR '64 LOW
*    TEST028E - TEST OF ADDRESS TRAP, DMX ADDR '66 HIGH
*    TEST028F - TEST OF ADDRESS TRAP, DMX ADDR '66 LOW
*    TEST0290 - TEST OF ADDRESS TRAP, DMX ADDR '70 HIGH
*    TEST0291 - TEST OF ADDRESS TRAP, DMX ADDR '70 LOW
*    TEST0292 - TEST OF ADDRESS TRAP, DMX ADDR '72 HIGH
*    TEST0293 - TEST OF ADDRESS TRAP, DMX ADDR '72 LOW
*    TEST0294 - TEST OF ADDRESS TRAP, DMX ADDR '74 HIGH
*    TEST0295 - TEST OF ADDRESS TRAP, DMX ADDR '74 LOW
*    TEST0296 - TEST OF ADDRESS TRAP, DMX ADDR '76 HIGH
*    TEST0297 - TEST OF ADDRESS TRAP, DMX ADDR '76 LOW
*    TEST0298 - TEST_TNUM
*    TEST02A0 - TEST OF RF SOURCE vs. RF DEST FIELDS RFS = '013
*    TEST02A1 - TEST OF RF SOURCE vs. RF DEST FIELDS RFS = '164
*    TEST02A2 - TEST_TNUM
*    TEST0300 - JUMP ON FALHEQ TRUE
*    TEST0301 - JUMP ON FALHEQ FALSE
*    TEST0302 - JUMP ON FALHNE TRUE
*    TEST0303 - JUMP ON FALHNE FALSE
*    TEST0304 - JUMP ON FALLENE TRUE
*    TEST0305 - JUMP ON FALLENE FALSE
*    TEST0306 - JUMP ON FALLNE TRUE
*    TEST0307 - JUMP ON FALLNE FALSE
*    TEST0308 - JUMP ON FAL32EQ TRUE
*    TEST0309 - JUMP ON FAL32EQ FALSE
*    TEST030A - JUMP ON FAL32NE TRUE
*    TEST030B - JUMP ON FAL32NE FALSE
*    TEST030C - JUMP ON FAL48EQ TRUE
*    TEST030D - JUMP ON FAL48EQ FALSE
*    TEST030E - TEST_TNUM
*    TEST0310 - TA (HIGH SIDE)
*    TEST0311 - TA (LOW SIDE)
*    TEST0312 - TA (EXTENDED SIDE)
*    TEST0313 - TB (HIGH SIDE)
*    TEST0314 - TB (LOW SIDE)
*    TEST0315 - TB (EXTENDED SIDE)
*    TEST0316 - AND (HIGH SIDE)
*    TEST0317 - AND (LOW SIDE)
*    TEST0318 - AND (EXTENDED SIDE)
*    TEST0319 - OR (HIGH SIDE)
*    TEST031A - OR (LOW SIDE)
*    TEST031B - OR (EXTENDED SIDE)
*    TEST031C - XOR (HIGH SIDE)
*    TEST031D - XOR (LOW SIDE)
*    TEST031E - XOR (EXTENDED SIDE)
*    TEST031F - NOTA (HIGH SIDE)
*    TEST0320 - NOTA (LOW SIDE)
*    TEST0321 - NOTA (EXTENDED SIDE)
*    TEST0322 - NOTB (HIGH SIDE)
*    TEST0323 - NOTB (LOW SIDE)
*    TEST0324 - NOTB (EXTENDED SIDE)
*    TEST0325 - ZEROES (HIGH SIDE)
*    TEST0326 - ZEROES (LOW SIDE)
*    TEST0327 - ZEROES (EXTENDED SIDE)
*    TEST0328 - MINUS1 (HIGH SIDE)
*    TEST0329 - MINUS1 (LOW SIDE)
*    TEST032A - MINUS1 (EXTENDED SIDE)
*    TEST032B - A.AND./B (HIGH SIDE)
*   TEST032C - A.AND./B (LOW SIDE)
*   TEST032D - A.AND./B (EXTENDED SIDE)
*   TEST032E - /A.AND.B (HIGH SIDE)
*   TEST032F - /A.AND.B (LOW SIDE)
*   TEST0330 - /A.AND.B (EXTENDED SIDE)
*   TEST0331 - TEST ALE (LOW SLICE) PROPAGATE TRUE
*   TEST0332 - TEST ALE (LOW SLICE) PROPAGATE FALSE
*   TEST0333 - TEST ALE (LOW SLICE) GENERATE TRUE
*   TEST0334 - TEST ALE (LOW SLICE) GENERATE FALSE
*   TEST0335 - TEST ALL (LOW SLICE) PROPAGATE TRUE
*   TEST0336 - TEST ALL (LOW SLICE) PROPAGATE FALSE
*   TEST0337 - TEST ALL (LOW SLICE) GENERATE TRUE
*   TEST0338 - TEST ALL (LOW SLICE) GENERATE FALSE
*   TEST0339 - TEST ALH (LOW SLICE) PROPAGATE TRUE
*   TEST033A - TEST ALH (LOW SLICE) PROPAGATE FALSE
*   TEST033B - TEST ALH (LOW SLICE) GENERATE TRUE
*   TEST033C - TEST ALH (LOW SLICE) GENERATE FALSE
*   TEST033D - TEST ALE (HIGH SLICE) PROPAGATE TRUE
*   TEST033E - TEST ALE (HIGH SLICE) PROPAGATE FALSE
*   TEST033F - TEST ALE (HIGH SLICE) GENERATE TRUE
*   TEST0340 - TEST ALE (HIGH SLICE) GENERATE FALSE
*   TEST0341 - TEST ALL (HIGH SLICE) PROPAGATE TRUE
*   TEST0342 - TEST ALL (HIGH SLICE) PROPAGATE FALSE
*   TEST0343 - TEST ALL (HIGH SLICE) GENERATE TRUE
*   TEST0344 - TEST ALL (HIGH SLICE) GENERATE FALSE
*   TEST0345 - TEST ALH (HIGH SLICE) PROPAGATE TRUE
*   TEST0346 - TEST ALH (HIGH SLICE) PROPAGATE FALSE
*   TEST0347 - TEST ALH (HIGH SLICE) GENERATE TRUE
*   TEST0348 - TEST ALH (HIGH SLICE) GENERATE FALSE
*   TEST0349 - TEST_TNUM
*   TEST0350 - ADD (HIGH SIDE)
*   TEST0351 - ADD (LOW SIDE)
*   TEST0352 - ADD (EXTENDED SIDE)
*   TEST0353 - SUBB (HIGH SIDE)
*   TEST0354 - SUBB (LOW SIDE)
*   TEST0355 - SUBB (EXTENDED SIDE)
*   TEST0356 - SUBA (LOW SIDE ONLY)
*   TEST0357 - INCA (HIGH SIDE)
*   TEST0358 - INCA (LOW SIDE)
*   TEST0359 - INCA (EXTENDED SIDE)
*   TEST035A - INCB (HIGH SIDE)
*   TEST035B - INCB (LOW SIDE)
*   TEST035C - INCB (EXTENDED SIDE)
*   TEST035D - DECA (HIGH SIDE)
*   TEST035E - DECA (LOW SIDE)
*   TEST035F - DECA (EXTENDED SIDE)
*   TEST0360 - TEST_TNUM
*   TEST0370 - JUMP ON FALH00 (TA MODE)
*   TEST0371 - JUMP ON FALH00 (TB MODE)
*   TEST0372 - JUMP ON FALH00 (XOR MODE)
*   TEST0373 - JUMP ON FALH00 (INCA MODE)
*   TEST0374 - JUMP ON FALH00 (DECA MODE)
*   TEST0375 - JUMP ON FALH00 (ADD MODE)
*   TEST0376 - JUMP ON FALH00 (SUBB MODE)
*   TEST0377 - TEST_TNUM
*   TEST0378 - JUMP ON ALHCOUT TRUE
*   TEST0379 - JUMP ON ALHCOUT FALSE
*   TEST037A - JUMP ON FALHOV TRUE
*   TEST037B - JUMP ON FALHOV FALSE
*   TEST037C - JUMP ON ALH16 TRUE
*   TEST037D - JUMP ON ALH16 FALSE
*   TEST037E - JUMP ON ALL16 TRUE
*   TEST037F - JUMP ON ALL16 FALSE
*   TEST0380 - TEST C=0 TRUE
*   TEST0381 - TEST C=ALH01 TRUE
*   TEST0382 - TEST C=ALH01 FALSE
*   TEST0383 - JUMP ON NCBIT (USING C=ALH01)
*   TEST0384 - JUMP ON NCBIT (USING ALH01 FALSE)
*   TEST0385 - TEST C=COV TRUE
*   TEST0386 - TEST C=COV FALSE
*   TEST0387 - TEST C=ALHCOUT TRUE
*   TEST0388 - TEST C=ALHCOUT FALSE
*   TEST0389 - TEST C=ALHOV TRUE
*   TEST038A - TEST C=ALHOV FALSE
*   TEST038B - TEST C=SOVFLAC TRUE (ALH 10)
*   TEST038C - TEST C=SOVFLAC TRUE (ALH 01)
*   TEST038D - TEST C=SOVFLAC TRUE (CBIT SET)
*   TEST038E - TEST C=SOVFLAC FALSE (UNNORMALIZED)
*   TEST038F - TEST C=SOVFLAC FALSE (CBIT RESET)
*   TEST0390 - TEST C=SOVFLAC TRUE (CBIT SET - NORMALIZED)
*   TEST0391 - JUMP ON LINK (USING L=0)
*   TEST0392 - JUMP ON NLINK (USING L=0)
*   TEST0393 - TEST C=BDICBIT TRUE
*   TEST0394 - TEST C=BDICBIT FALSE
*   TEST0395 - TEST C=PLINK TRUE
*   TEST0396 - TEST C=PLINK FALSE
*   TEST0397 - TEST L=ALH01 TRUE
*   TEST0398 - TEST L=ALH01 FALSE
*   TEST0399 - TEST L=ALH03 TRUE
*   TEST039A - TEST L=ALH03 FALSE
*   TEST039B - TEST L=ALHCOUT TRUE
*   TEST039C - TEST L=ALHCOUT FALSE
*   TEST039D - TEST L=ALHOV TRUE
*   TEST039E - TEST L=ALHOV FALSE
*   TEST039F - TEST L=ALLOV TRUE
*   TEST03A0 - TEST L=ALLOV FALSE
*   TEST03A1 - TEST L= BDICBIT TRUE
*   TEST03A2 - TEST L= BDICBIT FALSE
*   TEST03A3 - TEST_TNUM
*   TEST03B0 - TEST ALL CIN = 0
*   TEST03B1 - TEST ALL CIN = 1
*   TEST03B2 - TEST ALE CIN = 0
*   TEST03B3 - TEST ALE CIN = 1
*   TEST03B4 - TEST ALL CIN = COE(0)
*   TEST03B5 - TEST ALL CIN = COE(1)
*   TEST03B6 - TEST ALH CIN = 0
*   TEST03B7 - TEST ALH CIN = 1
*   TEST03B8 - TEST ALH CIN = COL(0)
*   TEST03B9 - TEST ALH CIN = COL(1)
*   TEST03BA - TEST ALH CIN = CBIT(1)
*   TEST03BB - TEST ALH CIN = CBIT(0)
*   TEST03BC - TEST ALL CIN = ALHCOUT(0)
*   TEST03BD - TEST ALL CIN = ALHCOUT(1)
*   TEST03BE - TEST_TNUM
*   TEST03C0 - TB:ZERO (HIGH SIDE)
*   TEST03C1 - TA/TB (HIGH SIDE)
*   TEST03C2 - TB/ZERO (HIGH SIDE)
*   TEST03C3 - TB/TA (HIGH SIDE)
*   TEST03C4 - ZERO/TB (HIGH SIDE)
*   TEST03C5 - ZERO/TA (HIGH SIDE)
*   TEST03C6 - TCB:ZERO (HIGH SIDE)
*   TEST03C7 - TB:TA (LOW SIDE)
*    TEST03C8 - TA:TB (LOW SIDE)
*    TEST03C9 - TA:ZERO (LOW SIDE)
*    TEST03CA - TB:ZERO (LOW SIDE)
*    TEST03CB - ZERO:TB (LOW SIDE)
*    TEST03CC - ZERO:TA (LOW SIDE)
*    TEST03CD - TCB:ZERO (LOW SIDE)
*    TEST03CE - INCA:ADD (LOW SIDE)
*    TEST03CF - ADD:INCA (LOW SIDE)
*    TEST03D0 - DECA:SUBB (LOW SIDE)
*    TEST03D1 - TB/TA (EXTENDED SIDE)
*    TEST03D2 - TA/ZERO (EXTENDED SIDE)
*    TEST03D3 - TA/TB (EXTENDED SIDE)
*    TEST03D4 - TB:ZERO (EXTENDED SIDE)
*    TEST03D5 - TCB:ZERO (EXTENDED SIDE)
*    TEST03D6 - TEST_TNUM
*    TEST03E0 - JUMP ON ADL00 TRUE
*    TEST03E1 - JUMP ON ADL00 FALSE
*    TEST03E2 - JUMP ON ADL01 TRUE
*    TEST03E3 - JUMP ON ADL01 FALSE
*    TEST03E4 - JUMP ON AM0 TRUE
*    TEST03E5 - JUMP ON AM0 FALSE
*    TEST03E6 - JUMP ON AM1 TRUE
*    TEST03E7 - JUMP ON AM1 FALSE
*    TEST03E8 - JUMP ON AM2 TRUE
*    TEST03E9 - JUMP ON AM2 FALSE
*    TEST03EA - JUMP ON NASCII8 TRUE
*    TEST03EB - JUMP ON NASCII8 FALSE
*    TEST03EC - JUMP ON CRS1 TRUE
*    TEST03ED - JUMP ON CRS1 FALSE
*    TEST03EE - JUMP ON CRS2 TRUE
*    TEST03EF - JUMP ON CRS2 FALSE
*    TEST03F0 - JUMP ON FVIM TRUE
*    TEST03F1 - JUMP ON FVIM FALSE
*    TEST03F2 - JUMP ON FNRND TRUE
*    TEST03F3 - JUMP ON FNRND FALSE
*    TEST03F4 - JUMP ON KEYSH11 TRUE
*    TEST03F5 - JUMP ON KEYSH11 FALSE
*    TEST03F6 - JUMP ON RDE01 TRUE
*    TEST03F7 - JUMP ON RDE01 FALSE
*    TEST03F8 - JUMP ON RDE05 TRUE
*    TEST03F9 - JUMP ON RDE05 FALSE
*    TEST03FA - JUMP ON CC48EQ TRUE
*    TEST03FB - JUMP ON CC48EQ FALSE
*    TEST03FC - JUMP ON CCEQ TRUE (SETCC)
*    TEST03FD - JUMP ON CCEQ TRUE (SETCC48)
*    TEST03FE - JUMP ON CCEQ FALSE (SETCC)
*    TEST03FF - JUMP ON CCEQ FALSE (SETCC48)
```

The SYSV2 overlay runs through the remaining JCs before thoroughly testing out all the Bus D Internal (BDI) barrel shifter modes. Additionally, a number of tests are executed to verify some of the more random E unit functions such as the link bit. The following is an index of the tests contained in SYSV2.

```
*
*    INDEX OF TESTS IN SYSV2
*    ----------------------
*
*  TEST0400 - JUMP ON CCNE14 TRUE (SETCC)
*  TEST0401 - JUMP ON CCNE14 TRUE (SETCC48)
*  TEST0402 - JUMP ON CCNE14 FALSE (SETCC)
```

* TEST0403 – JUMP ON CCNE14 FALSE (SETCC48)
* TEST0404 – JUMP ON CCLT TRUE
* TEST0405 – JUMP ON CCLT FALSE
* TEST0406 – JUMP ON CCGE TRUE
* TEST0407 – JUMP ON CCGE FALSE
* TEST0408 – JUMP ON CCGT TRUE
* TEST0409 – JUMP ON CCGT FALSE (LESS THAN)
* TEST040A – JUMP ON CCGT FALSE (EQUAL)
* TEST040B – JUMP ON CCLE TRUE (LESS THAN)
* TEST040C – JUMP ON CCLE TRUE (EQUAL)
* TEST040D – JUMP ON CCLE FALSE
* TEST040E – CRTN ON CCNE TRUE
* TEST040F – CRTN ON CCNE FALSE
* TEST0410 – JUMP ON FLEX TRUE
* TEST0411 – JUMP ON FLEX FALSE
* TEST0412 – CRTN ON FLEXRTN TRUE
* TEST0413 – CRTN ON FLEXCRTN FALSE
* TEST0414 – JUMP ON FNIEX TRUE (NO IAC PRESENT)
* TEST0415 – JUMP ON FNIEX TRUE (NO KEYS BIT PRESENT)
* TEST0416 – JUMP ON FNIEX TRUE (NO OVERFLOW PRESENT)
* TEST0417 – JUMP ON FNIEX FALSE
* TEST0418 – JUMP ON CCNE15 TRUE (SETCC)
* TEST0419 – JUMP ON CCNE15 FALSE (SETCC)
* TEST041A – TEST_TNUM
* TEST0420 – JUMP ON FALH04 TRUE
* TEST0421 – JUMP ON FALH04 FALSE
* TEST0422 – JUMP ON FALH07 TRUE
* TEST0423 – JUMP ON FALH07 FALSE
* TEST0424 – JUMP ON FALH08 TRUE
* TEST0425 – JUMP ON FALH08 FALSE
* TEST0426 – JUMP ON FALH13 TRUE
* TEST0427 – JUMP ON FALH13 FALSE
* TEST0428 – JUMP ON FALH14 TRUE
* TEST0429 – JUMP ON FALH14 FALSE
* TEST042A – JUMP ON FALH15 TRUE
* TEST042B – JUMP ON FALH15 FALSE
* TEST042C – JUMP ON FALL02 TRUE
* TEST042D – JUMP ON FALL02 FALSE
* TEST042E – JUMP ON FBA48EQ TRUE
* TEST042F – JUMP ON FBA48EQ FALSE
* TEST0430 – JUMP ON FBA48EQ14 TRUE
* TEST0431 – JUMP ON FBA48EQ14 FALSE
* TEST0432 – JUMP ON FIBB48EQ TRUE
* TEST0433 – JUMP ON FIBB48EQ FALSE
* TEST0434 – TEST_TNUM
* TEST0436 – JUMP ON FNALH02 TRUE
* TEST0437 – JUMP ON FNALH02 FALSE
* TEST0438 – JUMP ON FNALH05 TRUE
* TEST0439 – JUMP ON FNALH05 FALSE
* TEST043A – JUMP ON FALLOV TRUE
* TEST043B – JUMP ON FALLOV FALSE
* TEST043C – JUMP ON FALHLT TRUE
* TEST043D – JUMP ON FALHLT FALSE
* TEST043E – JUMP ON FALLLT TRUE
* TEST043F – JUMP ON FALLLT FALSE
* TEST0440 – JUMP ON FALHGE TRUE
* TEST0441 – JUMP ON FALHGE FALSE
* TEST0442 – JUMP ON FALHGT TRUE
* TEST0443 – JUMP ON FALHGT FALSE (LESS THAN)
* TEST0444 – JUMP ON FALHGT FALSE (EQUAL)
* TEST0445 – JUMP ON FALHLE TRUE (LESS THAN)

* **TEST0446** - JUMP ON FALHLE TRUE (EQUAL)
* **TEST0447** - JUMP ON FALHLE FALSE
* **TEST0448** - JUMP ON FALL00 FALSE
* **TEST0449** - JUMP ON FALL00 TRUE
* **TEST044A** - TEST_TNUM
* **TEST044D** - JUMP ON FNALLHBZ TRUE
* **TEST044E** - JUMP ON FNALLHBZ FALSE
* **TEST044F** - JUMP ON FAL51NE TRUE
* **TEST0450** - JUMP ON FAL51NE FALSE
* **TEST0451** - JUMP ON ALUMIN1(CS9) SINGLE PRECISION - TRUE
* **TEST0452** - JUMP ON ALUMIN1(CS9) SINGLE PRECISION - FALSE
* **TEST0453** - JUMP ON ALUMIN1(CS9) DOUBLE PRECISION - FALSE ADL00
* **TEST0454** - JUMP ON ALUMIN1(CS9) DOUBLE PRECISION - TRUE
* **TEST0455** - JUMP ON ALUMIN1(CS9) DOUBLE PRECISION - FALSE
* **TEST0456** - CRTN ON BITSFEEQ
* **TEST0457** - JUMP ON FALLLT14 TRUE
* **TEST0458** - JUMP ON FALLLT14 FALSE
* **TEST0459** - JUMP ON FALLLE TRUE (LESS THAN)
* **TEST045A** - JUMP ON FALLLE TRUE (EQUAL)
* **TEST045B** - JUMP ON FALLLE FALSE
* **TEST045C** - JUMP ON FSPLUS FALSE
* **TEST045D** - JUMP ON FSPLUS TRUE
* **TEST045E** - JUMP ON FSMINUS FALSE
* **TEST045F** - JUMP ON FSMINUS TRUE
* **TEST0460** - JUMP ON ADL02 TRUE
* **TEST0461** - JUMP ON ADL02 FALSE
* **TEST0462** - JUMP ON FEOI TRUE
* **TEST0463** - JUMP ON FEOI FALSE
* **TEST0464** - JUMP ON PXM TRUE
* **TEST0465** - JUMP ON PXM FALSE
* **TEST0466** - JUMP ON FALHNE TRUE
* **TEST0467** - JUMP ON FALHNE FALSE
* **TEST0468** - JUMP ON JCAP1 TRUE
* **TEST0469** - JUMP ON JCAP1 FALSE
* **TEST046A** - JUMP ON JCAP2 TRUE
* **TEST046B** - JUMP ON JCAP2 FALSE
* **TEST046C** - JUMP ON FTRAP_ALH00 TRUE
* **TEST046D** - JUMP ON FTRAP_ALH00 FALSE
* **TEST046E** - JUMP ON FTRAP_ALH02 FALSE
* **TEST046F** - JUMP ON FTRAP_ALH02 TRUE
* **TEST0470** - JUMP ON FTRAP_ALHCOUT TRUE
* **TEST0471** - JUMP ON FTRAP_ALHCOUT FALSE
* **TEST0472** - JUMP ON FTRAP_ALHNE TRUE
* **TEST0473** - JUMP ON FTRAP_ALHNE FALSE
* **TEST0474** - JUMP ON TRAP_ALL00 TRUE
* **TEST0475** - JUMP ON TRAP_ALL00 FALSE
* **TEST0476** - JUMP ON ALHEQ TRUE (LIVE TX=2)
* **TEST0477** - JUMP ON ALHEQ FALSE (LIVE TX=2)
* **TEST0478** - JUMP ON ALHNE TRUE (LIVE TX=2)
* **TEST0479** - JUMP ON ALHNE FALSE (LIVE TX=2)
* **TEST047A** - JUMP ON ALHEQ TRUE (LIVE TX=1)
* **TEST047B** - JUMP ON ALHEQ FALSE (LIVE TX=1)
* **TEST047C** - JUMP ON ALHNE TRUE (LIVE TX=1)
* **TEST047D** - JUMP ON ALHNE FALSE (LIVE TX=1)
* **TEST047E** - RESET & READ THE LIVE KEYS
* **TEST047F** - SET & READ THE LIVE KEYS
* **TEST0480** - JUMP ON FFPNZA = 10:B NO BYPASS
* **TEST0481** - JUMP ON FFPNZA = 01:B NO BYPASS
* **TEST0482** - JUMP ON FFPNZA FALSE NO BYPASS
* **TEST0483** - JUMP ON FFPNZA = 10:B (RS BYPASS)
* **TEST0484** - JUMP ON FFPNZA = 01:B (RS BYPASS)

* TEST0485 - JUMP ON FFPNZA FALSE (RS BYPASS)
* TEST0486 - JUMP ON FFPNZA15 = 10:B NO BYPASS
* TEST0487 - JUMP ON FFPNZA15 = 01:B NO BYPASS
* TEST0488 - JUMP ON FFPNZA15 FALSE NO BYPASS
* TEST0489 - JUMP ON FFPNZA15 = 10:B (RS BYPASS)
* TEST048A - JUMP ON FFPNZA15 = 01:B (RS BYPASS)
* TEST048B - JUMP ON FFPNZA15 FALSE (RS BYPASS)
* TEST048C - TEST_TNUM
* TEST0490 - JUMP ON FF48POW2 FALSE
* TEST0491 - JUMP ON FF48POW2 TRUE
* TEST0492 - JUMP ON FF24NPOW2 TRUE (ie. IT IS NOT A POWER OF 2)
* TEST0493 - JUMP ON FF24NPOW2 FALSE (ie. IT IS A POWER OF 2)
* TEST0494 - JUMP ON FMIN1 TRUE
* TEST0495 - JUMP ON FMIN1 FALSE
* TEST0496 - JUMP ON DPUNNORM FALSE
* TEST0497 - JUMP ON DPUNNORM TRUE (FAC0 UNNORMALIZED)
* TEST0498 - JUMP ON DPUNNORM TRUE (FAC0 UNNORMALIZED)
* TEST0499 - JUMP ON DPUNNORM TRUE (FAC1 UNNORMALIZED)
* TEST049A - JUMP ON DPUNNORM TRUE (FAC1 UNNORMALIZED)
* TEST049B - JUMP ON DPUNNORM TRUE (TR048 UNNORMALIZED)
* TEST049C - JUMP ON DPUNNORM TRUE (TR048 UNNORMALIZED)
* TEST049D - JUMP ON FNORM1,FNORM2 - UNNORMALIZED RS (00)
* TEST049E - JUMP ON FNORM1,FNORM2 - UNNORMALIZED FAC (00)
* TEST049F - JUMP ON FNORM1,FNORM2 - A NO SHIFT CASE (01)
* TEST04A0 - JUMP ON FNORM1,FNORM2 - A LEFT SHIFT BY 1 CASE (10)
* TEST04A1 - JUMP ON FNORM1,FNORM2 - AN UNPREDICTABLE CASE (11)
* TEST04A2 - TEST_TNUM
* TEST04B0 - JUMP ON GNFCYTR TRUE (ie NO TRAP PENDING)
* TEST04B1 - JUMP ON GNFCYTR FALSE - END-OF-INSTRUCTION CASE
* TEST04B2 - JUMP ON GNFCYTR FALSE - PIC CASE
* TEST04B3 - JUMP ON GNFCYTR FALSE - CP TIMER CASE
* TEST04B4 - TEST_TNUM
* TEST0500 - S$LEFT32 E=0 (HIGH SIDE)
* TEST0501 - S$LEFT32 E=0 (LOW SIDE)
* TEST0502 - S$LEFT48
* TEST0503 - S$LEFT_H E=0
* TEST0504 - S$LEFT_L E=0
* TEST0505 - TEST_TNUM
* TEST0507 - R$LEFT32 (LOW SIDE)
* TEST0508 - R$LEFT32 (HIGH SIDE)
* TEST0509 - S$LEFT_H E=0,ALL (HIGH SIDE)
* TEST050A - S$LEFT_H E=0,ALL (LOW SIDE)
* TEST050B - R$LEFT_H,ALL (HIGH SIDE)
* TEST050C - R$LEFT_H,ALL (LOW SIDE)
* TEST050D - ALH,S$LEFT_L E=0 (LOW SIDE)
* TEST050E - TEST_TNUM
* TEST0510 - S$RIGHT_H E=0 (HIGH SIDE)
* TEST0511 - S$RIGHT32 E=0
* TEST0512 - S$RIGHT32 E=ALH00 (HIGH SIDE)
* TEST0513 - S$RIGHT32 E=ALH00 (LOW SIDE)
* TEST0514 - S$RIGHT32 E=0 (LOW SIDE)
* TEST0515 - S$RIGHT32 E=0 (LOW SIDE)
* TEST0516 - R$RIGHT_H,L (HIGH SIDE)
* TEST0517 - R$RIGHT_H,L (LOW SIDE)
* TEST0518 - R$8LEFT32 (HIGH SIDE)
* TEST0519 - R$LEFT32 (LOW SIDE)
* TEST051A - S$5RIGHT 48 BITS
* TEST051B - S$9RIGHT 48 BITS
* TEST051C - R$8LEFT32 (LOW SIDE)
* TEST051D - TEST_TNUM
* TEST0551 -> TEST0590 => CALL TEST055X - TEST [S$NLEFT_H E= 0] BDI MODE

```
*  TEST0591 -> TEST05D0 => CALL TEST059X - TEST [ S$NLEFT32 E= 0 ] BDI MODE
*  TEST05D1 -> TEST05DF => CALL TEST05DX - TEST [ S$NRIGHT_H ] BD MODE
*                          SHFTCNT = 1 : 15
*  TEST05E0 -> TEST060F => CALL TEST05EX - TEST [ S$NRIGHT_H ] BDI MODE
*                          SHFTCNT = 16:64
*  TEST0610 - TEST_TNUM
*  TEST0621 -> TEST063F => CALL TEST062X - TEST [ S$NRIGHT32 E= 0 ] BD MODE
*                          SHFTCNT = 1 : 31
*  TEST0640 -> TEST065F => CALL TEST064X - TEST [ S$NRIGHT32 E= 0 ] BD MODE
*                          SHFTCNT = 32 : 64
*  TEST0660 - TEST_TNUM
*  TEST0661 -> TEST06A0 => CALL TEST066X - TEST [ R$NLEFT32 ,E] (1:64 ROTATES )
*  TEST06A1 -> TEST06E0 => CALL TEST06AX - TEST [ R$NLEFT_H, L ,E] HIGH SIDE
*                          TEST 1:64
*  TEST06E1 -> TEST06EF => CALL TEST06EX - TEST [ R$NRIGHT_H,L,E ] BDI MODE
*                          SHFTCNT = 01:15
*  TEST06F0 -> TEST06FE => CALL TEST06EX - TEST [ R$NRIGHT_H,L,E ] BDI MODE
*                          SHFTCNT = 16:31
*  TEST06FF - TEST_TNUM
*  TEST0700 -> TEST070E => CALL TEST06EX - TEST [ R$NRIGHT_H,L,E ] BDI MODE
*                          SHFTCNT = 32:47
*  TEST070F - TEST_TNUM
*  TEST0710 -> TEST071E => CALL TEST06EX - TEST [ R$NRIGHT_H,L,E ] BDI MODE
*                          SHFTCNT = 48:63
*  TEST071F - TEST_TNUM
*  TEST0721 -> TEST073F => CALL TEST072X - TEST [ R$NRIGHT32,E  ] BD MODE
*                          SHFTCNT = 1 : 31
*  TEST0740 -> TEST075F => CALL TEST074X - TEST [ R$NRIGHT32,E  ] BD MODE
*                          SHFTCNT = 32 : 63
*  TEST0760 - TEST_TNUM
*  TEST07C1 -> TEST07EF => CALL TEST07CX - TEST [ ADJUST ] BD MODE POSTIVE
*                          ADJUST = 1 : 47
*  TEST07F0 -> TEST07FF => CALL TEST07FX - TEST [ ADJUST ] BD MODE POSTIVE
*                          ADJUST = 48 : 64
*  TEST0800 - TEST_TNUM
*  TEST0811 -> TEST083F => CALL TEST081X - TEST [ ADJUST ] BD MODE NEGITIVE
*                          ADJUST = -1 :-47
*  TEST0840 -> TEST084E => CALL TEST084X - TEST [ ADJUST ] BD MODE NEGITIVE
*                          ADJUST =-48 :-64
*  TEST084F - TEST_TNUM
*  TEST0860 -> TEST086E => CALL TEST086X - NORMALIZE (ALH) CHECK FNCNT= 14 to 0
*  TEST086F - TEST_TNUM
*  TEST0870 -> TEST087F => CALL TEST087X - NORMALIZE (ALL) CHECK FNCNT= 30 to 15
*  TEST0880 -> TEST088F => CALL TEST088X - NORMALIZE (ALE) CHECK FNCNT= 46 to 31
*  TEST0890 - TEST_TNUM
*  TEST08A0 -> TEST08AE => CALL TEST08AX - NORMALIZE (ALH) CHECK FNCNT= 14 to 0
*  TEST08AF - TEST_TNUM
*  TEST08B0 -> TEST08BF => CALL TEST08BX - NORMALIZE (ALL) CHECK FNCNT= 30 to 15
*  TEST08C0 -> TEST08CF => CALL TEST08CX - NORMALIZE (ALE) CHECK FNCNT= 46 to 31
*  TEST08D0 - NORMALIZE (ALH)BACK SHIFT CHECK FNCNT= -1
*  TEST08D1 - NORMALIZE (ALH)BACK SHIFT CHECK FNCNT= 0
*  TEST08D2 - NORMALIZE (ALH) ZERO CASE. CHECKS FNCNT= 47 DEC
*  TEST08D3 - NORMALIZE (ALH) $FFFF . CHECKS FNCNT= 0
*  TEST08D4 - NORMALIZE (ALH) $8000 CASE. CHECKS FNCNT= 0
*  TEST08D5 - NORMALIZE (ALH) $4000 CASE. CHECKS FNCNT= 0
*  TEST08D6 - NORMALIZE (ALH)CHECK ALH00=0 FOR BACK SHIFT
*  TEST08D7 - NORMALIZE (ALH)FIX00 CHECK FNCNT= 47
*  TEST08D8 - NORMALIZE (ALH)FIX00 CHECK FNCNT= 47
*  TEST08D9 - TEST_TNUM
*  TEST08E0 -> TEST08EE => CALL TEST08EX - NORMALIZE (ALH) CHECK  RDH= $4000,
*                          FNCNT= 14 to 0
*  TEST08F0 -> TEST08FE => CALL TEST08FX - NORMALIZE (ALL) CHECK RDL = $4000,
*                          FNCNT= 30 to 15
*  TEST0900 -> TEST090E => CALL TEST090X - NORMALIZE (ALE) CHECK RD = $4000 ,
*                          FNCNT= 46 to 31
*  TEST0910 -> TEST0913 => CALL TEST091X - GUARD1 GUARD2 TEST
*  TEST0914 - TEST 16 BIT SHIFT RIGHT  OF RS48 FROM ALH TO ALL
*  TEST0915 - TEST TO SEE LOWER 8 BITS OF RSE IS COPIED TO EACH 8 BIT SECTION
*  TEST0916 - TEST TO SEE LOWER 8 BITS OF RSE IS COPIED TO EACH 8 BIT SECTION
*  TEST0917 - TEST TO SEE LOWER 8 BITS OF RSE IS COPIED TO EACH 8 BIT SECTION
*  TEST0918 - TEST RIE/LINK LEFT SHIFT MODE (LINK = 1)
*  TEST0919 - TEST RIE/LINK LEFT SHIFT MODE (LINK = 0)
*  TEST091A - TEST RIE/LINK LEFT SHIFT MODE (LINK = 1 RI8CLK)
*  TEST091B - TEST RIE/LINK LEFT SHIFT MODE (LINK = 0 RI8CLK)
*  TEST091C - TEST RIL LEFT SHIFT MODE (SHIFT IN 1)
*  TEST091D - TEST RIL LEFT SHIFT MODE (SHIFT IN 0)
*  TEST091E - TEST RIL LEFT SHIFT MODE (SHIFT IN 1/RI8CLK)
*  TEST091F - TEST RIL LEFT SHIFT MODE (SHIFT IN 0/RI8CLK)
*  TEST0920 - TEST RIH LEFT SHIFT MODE (SHIFT IN 1)
*  TEST0921 - TEST RIH LEFT SHIFT MODE (SHIFT IN 0)
*  TEST0922 - TEST RIH LEFT SHIFT MODE (SHIFT IN 1/RI8CLK)
*  TEST0923 - TEST RIH LEFT SHIFT MODE (SHIFT IN 0/RI8CLK)
*  TEST0924 - TEST RDE LEFT SHIFT MODE (SHIFT IN 1/RI8CLK)
*  TEST0925 - TEST RDE LEFT SHIFT MODE (SHIFT IN 0/RI8CLK)
*  TEST0926 - OUT OF RANGE OF GREATER THAN 64.
*  TEST0927 - ADJUST MODE ALU SELECT (PASS BLEG)
*  TEST0928 - ADJUST MODE ALU SELECT (PASS ALEG)
*  TEST0929 - BDI CONDITIONAL TRANSPORT FAST MULTIPLY FEATURE (TRUE)
*  TEST092A - BDI CONDITIONAL TRANSPORT FAST MULTIPLY FEATURE (FALSE)
*  TEST092B - REC -> FRDX HARDWARE
*  TEST092C - BDI CONDITIONAL TRANSPORT (ALH -> BDIL IF AL51=0)
*  TEST092D - DIVIDE MODE ALTERNATE ALU SELECT INPUT (ADD)
*  TEST092E - DIVIDE MODE ALTERNATE ALU SELECT INPUT (SUBA)
*  TEST092F - TEST_TNUM
*  TEST0950 - TEST OF THE [ QNORM ] BD MODE WITH ADJCNTS OF -1:-47
*  TEST0951 - TEST_TNUM
*  TEST0960 - TEST OF THE [ QADJUST ] BD MODE WITH ADJCNTS OF -1:-47
*  TEST0961 - TEST OF THE [ ADJUST ] BD MODE POSTIVE ADJUST
*  TEST0962 - TEST OF THE [ ADJUST ] BD MODE POSTIVE ADJUST
*  TEST0963 - TEST OF THE [ ADJUST ] BD MODE POSTIVE ADJUST
*  TEST0964 - TEST_TNUM
*  TEST0970 - TEST0977 => CALL TEST097X - TEST BDI NORMALIZE OF 31 AND 32 BITS,
*                         TEST DOES NORMALIZE FROM 48 TO 1
```

SYSV3 begins the Instruction (I) unit verification. The first thing verified are the basic data paths. Then cache addressing is tested, followed by each cache set by itself, and then both sets together. Cache invalidate and force other set functions are also tested. The decode net is read and written before being tested for specific addressing mode entry points. Finally, I unit JCs are tested and PMA addressing modes (EAF) testing begins. The following is an index of the tests contained in SYSV3.

```
*
*   INDEX OF TESTS IN SYSV3
*   ------------------------
*
*  TEST2000 - BD -> RS -> MA -> BB  HIGH SIDE
*  TEST2001 - BD -> RS -> MA -> BB  LOW SIDE
*  TEST2002 - RF -> BD -> MA -> BB  BDI BYPASS PATH   HIGH SIDE
*  TEST2003 - RF -> BD -> MA -> BB  BDI BYPASS PATH   LOW SIDE
*  TEST2004 - BD -> ERMA -> MA -> BB
*  TEST2005 - BD -> ERMA -> MA -> BB
*  TEST2006 - BD -> ERMA -> MA -> ERMA+1 -> MA -> BB CHECK ERMAH FOR NO INCR
*  TEST2007 - BD -> ERMA -> MA -> ERMA+2 -> MA -> BB CHECK ERMAH FOR NO INCR
*  TEST2008 - BD -> ERMA -> MA -> ERMA+1 -> MA -> BB CHECK CARRY PROPAGATE
*  TEST2009 - BD -> ERMA -> MA -> ERMA+1 -> MA -> BB CHECK WITH EXPPAT
*  TEST200A - BD -> ERMA -> MA -> ERMA+2 -> MA -> BB CHECK CARRY PROPAGATE
*  TEST200B - BD -> ERMA -> MA -> ERMA+2 -> MA -> BB CHECK WITH EXPPAT
*  TEST200C - BD -> RMA  ~BD->RMAL  CHECK RMAH
*  TEST200D - ~BD -> RMA BD -> RMAL CHECK RMAL
*  TEST200E - BD -> EAS -> MA -> BB   HIGH   SIDE
*  TEST200F - BD -> EAS -> MA -> BB  LOW SIDE
*  TEST2010 - BD -> EAS -> MA -> EAS+1 -> MA -> BB CHECK EASH FOR NO INCR
*  TEST2011 - BD -> EAS -> MA -> EAS+2 -> MA -> BB CHECK EASH FOR NO INCR
*  TEST2012 - BD -> EAS -> MA -> EAS-1 -> MA -> BB CHECK EASH FOR NO DECR
*  TEST2013 - BD -> EAS -> MA -> EAS-2 -> MA -> BB CHECK EASH FOR NO DECR
*  TEST2014 - BD -> EAS -> MA -> EAS+1 -> MA -> BB CHECK CARRY PROPAGATE
*  TEST2015 - BD -> EAS -> MA -> EAS+1 -> MA -> BB CHECK WITH EXPPAT
*  TEST2016 - BD -> EAS -> MA -> EAS+2 -> MA -> BB
*  TEST2017 - BD -> EAS -> MA -> EAS-1 -> MA -> BB
*  TEST2018 - BD -> EAS -> MA -> EAS-2 -> MA -> BB
*  TEST2019 - BD -> EAS  ~BD-> EASL CHECK EASH
*  TEST201A - ~BD -> EAS BD-> EASL CHECK EASL
*  TEST201B - BD -> EAD -> MA -> BB  HIGH SIDE
*  TEST201C - BD -> EAD -> MA -> BB  LOW SIDE
*  TEST201D - BD -> EAD -> MA -> EAD+1 -> MA -> BB  CHECK EADH FOR NO INCR
*  TEST201E - BD -> EAD -> MA -> EAD+2 -> MA -> BB  CHECK EADH FOR NO INCR
*  TEST201F - BD -> EAD -> MA -> EAD-1 -> MA -> BB  CHECK EADH FOR NO DECR
*  TEST2020 - BD -> EAD -> MA -> EAD-2 -> MA -> BB  CHECK EADH FOR NO DECR
*  TEST2021 - BD -> EAD -> MA -> EAD+1 -> MA -> BB
*  TEST2022 - BD -> EAD -> MA -> EAD+2 -> MA -> BB
*  TEST2023 - BD -> EAD -> MA -> EAD-1 -> MA -> BB
*  TEST2024 - BD -> EAD -> MA -> EAD-2 -> MA -> BB
*  TEST2025 - BD-> IRP -> MA -> BB
*  TEST2026 - BD -> IRP -> MA -> BB  CHECK IRPL
*  TEST2027 - TEST_TNUM
*  TEST2040 - CACHE ADDRESSING
*  TEST2041 - CACHE ADDRESSING, ELEMENT A HIGH DATA
*  TEST2042 - CACHE ADDRESSING, ELEMENT A LOW DATA
*  TEST2043 - CACHE ADDRESSING, ELEMENT B HIGH DATA
*  TEST2044 - CACHE ADDRESSING, ELEMENT B LOW DATA
*  TEST2045 - CACHE ADDRESSING, ELEMENT B E SIDE DATA
*  TEST2046 - CACHE ADDRESSING, ELEMENT A E SIDE DATA
*  TEST2047 - TEST_TNUM
*  TEST2060 - TEST INVALID BIT CACHE B, HIGH SIDE DATA
*  TEST2061 - TEST INVALID BIT CACHE B, LOW SIDE DATA
*  TEST2062 - TEST INVALID BIT CACHE A, HIGH SIDE DATA
*  TEST2063 - TEST INVALID BIT CACHE A, LOW SIDE DATA
*  TEST2064 - CACHE ELEMENT 'B' INVALID BIT, JUMP ON NCMISS
*  TEST2065 - CACHE ELEMENT 'A' INVALID BIT, JUMP ON NCMISS
*  TEST2066 - FORCES BOTH CACHES INVALID, JUMP ON NCMISS
*  TEST2067 - IAC FRCELE RCD32A  ON CSS CHIP
*  TEST2068 - IAC FRCELE RCD32B  ON CSS CHIP
*  TEST2069 - FRCA - ELEMENT 'A' HIGH SIDE DATA
*  TEST206A - FRCA - ELEMENT 'A' LOW SIDE DATA
*  TEST206B - FRCB - ELEMENT 'B' HIGH SIDE DATA
*  TEST206C - FRCB - ELEMENT 'B' LOW SIDE DATA
*  TEST206D - TEST_TNUM
*  TEST20B0 - Testing read and write of location 0 of decode net
*  TEST20B1 - Testing read and write of all locations of decode net.
*  TEST20B2 - Test decode net addressability, locn 0 vs 1
*  TEST20B3 - Test decode net addressability, locn 0 vs 2
```

* TEST20B4 – Test decode net addressability, locn 0 vs 4
* TEST20B5 – Test decode net addressability, locn 0 vs $8
* TEST20B6 – Test decode net addressability, locn 0 vs $10
* TEST20B7 – Test decode net addressability, locn 0 vs $20
* TEST20B8 – Test decode net addressability, locn 0 vs $40
* TEST20B9 – Test decode net addressability, locn 0 vs $80
* TEST20BA – Test decode net addressability, locn 0 vs $100
* TEST20BB – Test decode net addressability, locn 0 vs $200
* TEST20BC – Test decode net addressability, locn 0 vs $400
* TEST20BD – Test decode net addressability, locn 0 vs $800
* TEST20BE – Test decode net addressability, locn $FFF vs $FFE
* TEST20BF – Test decode net addressability, locn $FFF vs $FFD
* TEST20C0 – Test decode net addressability, locn $FFF vs $FFB
* TEST20C1 – Test decode net addressability, locn $FFF vs $FF7
* TEST20C2 – Test decode net addressability, locn $FFF vs $FEF
* TEST20C3 – Test decode net addressability, locn $FFF vs $FDF
* TEST20C4 – Test decode net addressability, locn $FFF vs $FBF
* TEST20C5 – Test decode net addressability, locn $FFF vs $F7F
* TEST20C6 – Test decode net addressability, locn $FFF vs $EFF
* TEST20C7 – Test decode net addressability, locn $FFF vs $DFF
* TEST20C8 – Test decode net addressability, locn $FFF vs $BFF
* TEST20C9 – Test decode net addressability, locn $FFF vs $7FF
* TEST20CA – TEST_TNUM
* TEST20CB – TEST THE OPCODE REGISTER FOR A ZERO
* TEST20CC – TEST OPCODE BIT 16 FOR 1
* TEST20CD – TEST OPCODE BIT 15 FOR 1
* TEST20CE – TEST OPCODE BIT 14 FOR 1
* TEST20CF – TEST OPCODE BIT 13 FOR 1
* TEST20D0 – TEST OPCODE BIT 12 FOR 1
* TEST20D1 – TEST OPCODE BIT 11 FOR 1
* TEST20D2 – TEST OPCODE BIT 10 FOR 1
* TEST20D3 – TEST OPCODE BIT 09 FOR 1
* TEST20D4 – TEST OPCODE BIT 08 FOR 1
* TEST20D5 – TEST OPCODE BIT 07 FOR 1
* TEST20D6 – TEST OPCODE BIT 06 FOR 1
* TEST20D7 – TEST OPCODE BIT 05 FOR 1
* TEST20D8 – TEST OPCODE BIT 04 FOR 1
* TEST20D9 – TEST OPCODE BIT 03 FOR 1
* TEST20DA – TEST OPCODE BIT 02 FOR 1
* TEST20DB – TEST OPCODE BIT 01 FOR 1
* TEST20DC – TEST OPCODE REGISTER FOR ALL ONES
* TEST20DD – TEST OPCODE BIT 16 FOR 0
* TEST20DE – TEST OPCODE BIT 15 FOR 0
* TEST20DF – TEST OPCODE BIT 14 FOR 0
* TEST20E0 – TEST OPCODE BIT 13 FOR 0
* TEST20E1 – TEST OPCODE BIT 12 FOR 0
* TEST20E2 – TEST OPCODE BIT 11 FOR 0
* TEST20E3 – TEST OPCODE BIT 10 FOR 0
* TEST20E4 – TEST OPCODE BIT 09 FOR 0
* TEST20E5 – TEST OPCODE BIT 08 FOR 0
* TEST20E6 – TEST OPCODE BIT 07 FOR 0
* TEST20E7 – TEST OPCODE BIT 06 FOR 0
* TEST20E8 – TEST OPCODE BIT 05 FOR 0
* TEST20E9 – TEST OPCODE BIT 04 FOR 0
* TEST20EA – TEST OPCODE BIT 03 FOR 0
* TEST20EB – TEST OPCODE BIT 02 FOR 0
* TEST20EC – TEST OPCODE BIT 01 FOR 0
* TEST20ED – TEST_TNUM
* TEST20EE – TEST SKIP DECODE
* TEST20EF – TEST SHIFT DECODE
* TEST20F0 – TEST SHIFT DECODE

```
*  TEST20F1 - TEST SHIFT DECODE
*  TEST20F2 - TEST_TNUM
*  TEST20F5 - TEST SKIP ON A REG BIT LSB SET (SLN), FNSKIP IS FALSE
*  TEST20F6 - TEST SKIP ON A REG BIT 16 SET (SAS), FNSKIP IS FALSE
*  TEST20F7 - TEST SKIP ON A REG BIT 15 SET, FNSKIP IS FALSE
*  TEST20F8 - TEST SKIP ON A REG BIT 14 SET, FNSKIP IS FALSE
*  TEST20F9 - TEST SKIP ON A REG BIT 13 SET, FNSKIP IS FALSE
*  TEST20FA - TEST SKIP ON A REG BIT 12 SET, FNSKIP IS FALSE
*  TEST20FB - TEST SKIP ON A REG BIT 11 SET, FNSKIP IS FALSE
*  TEST20FC - TEST SKIP ON A REG BIT 10 SET, FNSKIP IS FALSE
*  TEST20FD - TEST SKIP ON A REG BIT 09 SET, FNSKIP IS FALSE
*  TEST20FE - TEST SKIP ON A REG BIT 08 SET, FNSKIP IS FALSE
*  TEST20FF - TEST SKIP ON A REG BIT 07 SET, FNSKIP IS FALSE
*  TEST2100 - TEST SKIP ON A REG BIT 06 SET, FNSKIP IS FALSE
*  TEST2101 - TEST SKIP ON A REG BIT 05 SET, FNSKIP IS FALSE
*  TEST2102 - TEST SKIP ON A REG BIT 04 SET, FNSKIP IS FALSE
*  TEST2103 - TEST SKIP ON A REG BIT 03 SET, FNSKIP IS FALSE
*  TEST2104 - TEST SKIP ON A REG BIT 02 SET, FNSKIP IS FALSE
*  TEST2105 - TEST SKIP ON A REG BIT 01 SET, FNSKIP IS FALSE
*  TEST2106 - TEST SKIP ON A REG BIT 16 RESET, FNSKIP IS TRUE
*  TEST2107 - TEST SKIP ON A REG BIT 15 RESET, FNSKIP IS TRUE
*  TEST2108 - TEST SKIP ON A REG BIT 14 RESET, FNSKIP IS TRUE
*  TEST2109 - TEST SKIP ON A REG BIT 13 RESET, FNSKIP IS TRUE
*  TEST210A - TEST SKIP ON A REG BIT 12 RESET, FNSKIP IS TRUE
*  TEST210B - TEST SKIP ON A REG BIT 11 RESET, FNSKIP IS TRUE
*  TEST210C - TEST SKIP ON A REG BIT 10 RESET, FNSKIP IS TRUE
*  TEST210D - TEST SKIP ON A REG BIT 09 RESET, FNSKIP IS TRUE
*  TEST210E - TEST SKIP ON A REG BIT 08 RESET, FNSKIP IS TRUE
*  TEST210F - TEST SKIP ON A REG BIT 07 RESET, FNSKIP IS TRUE
*  TEST2110 - TEST SKIP ON A REG BIT 06 RESET, FNSKIP IS TRUE
*  TEST2111 - TEST SKIP ON A REG BIT 05 RESET, FNSKIP IS TRUE
*  TEST2112 - TEST SKIP ON A REG BIT 04 RESET, FNSKIP IS TRUE
*  TEST2113 - TEST SKIP ON A REG BIT 03 RESET, FNSKIP IS TRUE
*  TEST2114 - TEST SKIP ON A REG BIT 02 RESET, FNSKIP IS TRUE
*  TEST2115 - TEST SKIP ON A REG BIT 01 RESET, FNSKIP IS TRUE
*  TEST2116 - TEST SKIP ON SENSE SWITCH 1 SET TO 1, FNSKIP IS FALSE
*  TEST2117 - TEST SKIP ON SENSE SWITCH 2 SET TO 1, FNSKIP IS FALSE
*  TEST2118 - TEST SKIP ON SENSE SWITCH 3 SET TO 1, FNSKIP IS FALSE
*  TEST2119 - TEST SKIP ON SENSE SWITCH 4 SET TO 1, FNSKIP IS FALSE
*  TEST211A - TEST SKIP ON SENSE SWITCH 1 RESET TO 0, FNSKIP IS FALSE
*  TEST211B - TEST SKIP ON CBIT SET TO 1
*  TEST211C - TEST SKIP ON A REG LESS THAN OR EQUAL TO 0
*  TEST211D - JUMP ON OPCODE BIT 16 FOR 1
*  TEST211E - JUMP ON OPCODE BIT 15 FOR 1
*  TEST211F - JUMP ON OPCODE BIT 14 FOR 1
*  TEST2120 - JUMP ON OPCODE BIT 07 FOR 1
*  TEST2121 - JUMP ON OPCODE BIT 16 FOR 0
*  TEST2122 - JUMP ON OPCODE BIT 15 FOR 0
*  TEST2123 - JUMP ON OPCODE BIT 14 FOR 0
*  TEST2124 - JUMP ON OPCODE BIT 07 FOR 0
*  TEST2125 - TEST_TNUM
*  TEST2150 - TEST 16S INDIRECT EAF
*  TEST2151 - TEST 32R INDIRECT EAF
*  TEST2152 - TEST 32R INDIRECT, POST-INDEXING
*  TEST2153 - TEST 32R STACK RELATIVE
*  TEST2154 - TEST 32R STACK RELATIVE, INDIRECT
*  TEST2155 - TEST 32R STACK RELATIVE, INDIRECT, POST-INDEXING
*  TEST2156 - TEST 32R STACK POST-INCREMENT
*  TEST2157 - TEST 32R STACK POST-INCREMENT INDIRECT
*  TEST2158 - TEST 32R STACK POST-INCREMENT, INDIRECT, POST-INDEXING
*  TEST2159 - TEST 32R STACK PRE-DECREMENT
*  TEST215A - TEST 32R STACK PRE-DECREMENT, INDIRECT
*  TEST215B - TEST 32R STACK PRE-DECREMENT, INDIRECT, POST-INDEXING
*  TEST215C - TEST 64R INDIRECTION
*  TEST215D - TEST 64R INDIRECT POST-INDEXING
*  TEST215E - TEST_TNUM
*  TEST21A0 - TEST OF IRPL, INCREMENT
*  TEST21A1 - TEST OF IRPL, TWO SHORT ALIGNED INSTR. AND ONE LONG INSTR
*  TEST21A2 - TEST OF IRPL, 1 LONG INSTR, 1 SHORT INSTR, AND 1 LONG INSTR
*  TEST21A3 - TEST OF IRPL, 1 SHORT ALIGNED INSTR AND 2 LONG UNALIGNED
*  TEST21A4 - TEST OF RPL, INCREMENT, THREE LONG ALIGNED INSTRUCTIONS        *
*  TEST21A5 - TEST OF RPL, 1 LNG INSTR, 1 SHORT INSTR, 1 LNG, ODD START ADDR
*  TEST21A6 - TEST OF RPL, 2 SHORT ALIGNED INSTR AND 1 LONG INSTR.
*  TEST21A7 - TEST OF RPL, 1 SHORT ALIGNED INSTR AND 2 LONG UNALIGNED
*  TEST21A8 - TEST_TNUM
*  TEST21C0 - INSTRUCTION ALIGNMENT, 2 LONG ALIGNED
*  TEST21C1 - INSTRUCTION ALIGNMENT, 2 SHORT ALIGNED, 1 LONG ALIGNED
*  TEST21C2 - INSTRUCTION ALIGNMENT, 1 SHORT ALIGNED, 1 LONG UNALIGNED, 1 SHORT
*  TEST21C3 - INSTRUCTION ALIGNMENT, 2 SHORT ALIGNED (MR), 1 LONG ALIGNED (MR)
*  TEST21C4 - INSTRUCTION ALIGNMENT, 1 SHORT ALIGNED, 1 LONG UNALIGNED, 1 SHORT
*  TEST21C5 - INSTRUCTION ALIGNMENT, 2 LONG UNALIGNED, CHECK RPL
*  TEST21C6 - INSTRUCTION ALIGNMENT, 2 SHORT ALIGNED, 1 LONG ALIGNED, CHECK RPL
*  TEST21C7 - INSTR ALIGNMENT,1 SHORT UNALIGNED,1 LONG ALIGNED,1 SHORT,CHECK RPL
*  TEST21C8 - INSTR ALIGNMENT,2 SHORT ALIGNED (MR),1 LONG ALIGNED (MR),CHECK RPL
*  TEST21C9 - INSTR ALIGNMENT,1 SHORT UNALIGNED,1 LNG UNALIGNED,1 SHORT,CHK RPL
*  TEST21CA - TEST_TNUM
*  TEST2300 => TEST230W - IMMEDIATE TYPE 1, FIRST NIBBLE
*  TEST2301 => TEST230W - IMMEDIATE TYPE 1, SECOND NIBBLE
*  TEST2302 => TEST230W - IMMEDIATE TYPE 1, THIRD NIBBLE
*  TEST2303 => TEST230W - IMMEDIATE TYPE 1, FOURTH NIBBLE
*  TEST2304 => TEST230X - IMMEDIATE TYPE 2, FIRST NIBBLE
*  TEST2305 => TEST230X - IMMEDIATE TYPE 2, SECOND NIBBLE
*  TEST2306 => TEST230X - IMMEDIATE TYPE 2, THIRD NIBBLE
*  TEST2307 => TEST230X - IMMEDIATE TYPE 2, FOURTH NIBBLE
*  TEST2308 => TEST230Y - IMMEDIATE TYPE 3, FIRST NIBBLE
*  TEST2309 => TEST230Y - IMMEDIATE TYPE 3, SECOND NIBBLE
*  TEST230A => TEST230Y - IMMEDIATE TYPE 3, THIRD NIBBLE
*  TEST230B => TEST230Y - IMMEDIATE TYPE 3, FOURTH NIBBLE
*  TEST230C - TEST OF GOOD DECODE NET ENTRY POINT FOR GR0 REG-TO-REG
*  TEST230D => TEST230Z - TEST GR0 THE HIGH SIDE NIBBLES
*  TEST230E => TEST230Z - TEST GR0 THE HIGH SIDE NIBBLES
*  TEST230F => TEST230Z - TEST GR0 THE HIGH SIDE NIBBLES
*  TEST2310 => TEST230Z - TEST GR0 THE HIGH SIDE NIBBLES
*  TEST2311 => TEST231W - TEST GR0 THE LOW SIDE NIBBLES
*  TEST2312 => TEST231W - TEST GR0 THE LOW SIDE NIBBLES
*  TEST2313 => TEST231W - TEST GR0 THE LOW SIDE NIBBLES
*  TEST2314 => TEST231W - TEST GR0 THE LOW SIDE NIBBLES
*  TEST2315 - TEST OF GOOD DECODE NET ENTRY POINT FOR GR1 REG-TO-REG
*  TEST2316 => TEST231X - TEST GR1 THE HIGH SIDE NIBBLES
*  TEST2317 => TEST231X - TEST GR1 THE HIGH SIDE NIBBLES
*  TEST2318 => TEST231X - TEST GR1 THE HIGH SIDE NIBBLES
*  TEST2319 => TEST231X - TEST GR1 THE HIGH SIDE NIBBLES
*  TEST231A => TEST231Y - TEST GR1 THE LOW SIDE NIBBLES
*  TEST231B => TEST231Y - TEST GR1 THE LOW SIDE NIBBLES
*  TEST231C => TEST231Y - TEST GR1 THE LOW SIDE NIBBLES
*  TEST231D => TEST231Y - TEST GR1 THE LOW SIDE NIBBLES
*  TEST231E - TEST OF GOOD DECODE NET ENTRY POINT FOR GR2 REG-TO-REG
*  TEST231F => TEST231Z - TEST GR2 THE HIGH SIDE NIBBLES
*  TEST2320 => TEST231Z - TEST GR2 THE HIGH SIDE NIBBLES
*  TEST2321 => TEST231Z - TEST GR2 THE HIGH SIDE NIBBLES
*  TEST2322 => TEST231Z - TEST GR2 THE HIGH SIDE NIBBLES
*  TEST2323 => TEST232W - TEST GR2 THE LOW SIDE NIBBLES
*  TEST2324 => TEST232W - TEST GR2 THE LOW SIDE NIBBLES
*  TEST2325 => TEST232W - TEST GR2 THE LOW SIDE NIBBLES
*  TEST2326 => TEST232W - TEST GR2 THE LOW SIDE NIBBLES
*  TEST2327 - TEST OF GOOD DECODE NET ENTRY POINT FOR GR3 REG-TO-REG
*  TEST2328 => TEST232X - TEST GR3 THE HIGH SIDE NIBBLES
*  TEST2329 => TEST232X - TEST GR3 THE HIGH SIDE NIBBLES
*  TEST232A => TEST232X - TEST GR3 THE HIGH SIDE NIBBLES
*  TEST232B => TEST232X - TEST GR3 THE HIGH SIDE NIBBLES
*  TEST232C => TEST232Y - TEST GR3 THE LOW SIDE NIBBLES
*  TEST232D => TEST232Y - TEST GR3 THE LOW SIDE NIBBLES
*  TEST232E => TEST232Y - TEST GR3 THE LOW SIDE NIBBLES
*  TEST232F => TEST232Y - TEST GR3 THE LOW SIDE NIBBLES
*  TEST2330 - TEST OF GOOD DECODE NET ENTRY POINT FOR GR4 REG-TO-REG
*  TEST2331 => TEST233W - TEST GR4 THE HIGH SIDE NIBBLES
*  TEST2332 => TEST233W - TEST GR4 THE HIGH SIDE NIBBLES
*  TEST2333 => TEST233W - TEST GR4 THE HIGH SIDE NIBBLES
*  TEST2334 => TEST233W - TEST GR4 THE HIGH SIDE NIBBLES
*  TEST2335 => TEST233X - TEST GR4 THE LOW SIDE NIBBLES
*  TEST2336 => TEST233X - TEST GR4 THE LOW SIDE NIBBLES
*  TEST2337 => TEST233X - TEST GR4 THE LOW SIDE NIBBLES
*  TEST2338 => TEST233X - TEST GR4 THE LOW SIDE NIBBLES
*  TEST2339 - TEST OF GOOD DECODE NET ENTRY POINT FOR GR5 REG-TO-REG
*  TEST233A => TEST233Y - TEST GR5 THE HIGH SIDE NIBBLES
*  TEST233B => TEST233Y - TEST GR5 THE HIGH SIDE NIBBLES
*  TEST233C => TEST233Y - TEST GR5 THE HIGH SIDE NIBBLES
*  TEST233D => TEST233Y - TEST GR5 THE HIGH SIDE NIBBLES
*  TEST233E => TEST233Z - TEST GR5 THE LOW SIDE NIBBLES
*  TEST233F => TEST233Z - TEST GR5 THE LOW SIDE NIBBLES
*  TEST2340 => TEST233Z - TEST GR5 THE LOW SIDE NIBBLES
*  TEST2341 => TEST233Z - TEST GR5 THE LOW SIDE NIBBLES
*  TEST2342 - TEST OF GOOD DECODE NET ENTRY POINT FOR GR6 REG-TO-REG
*  TEST2343 => TEST234W - TEST GR6 THE HIGH SIDE NIBBLES
*  TEST2344 => TEST234W - TEST GR6 THE HIGH SIDE NIBBLES
*  TEST2345 => TEST234W - TEST GR6 THE HIGH SIDE NIBBLES
*  TEST2346 => TEST234W - TEST GR6 THE HIGH SIDE NIBBLES
*  TEST2347 => TEST234X - TEST GR6 THE LOW SIDE NIBBLES
*  TEST2348 => TEST234X - TEST GR6 THE LOW SIDE NIBBLES
*  TEST2349 => TEST234X - TEST GR6 THE LOW SIDE NIBBLES
*  TEST234A => TEST234X - TEST GR6 THE LOW SIDE NIBBLES
*  TEST234B - TEST OF GOOD DECODE NET ENTRY POINT FOR GR7 REG-TO-REG
*  TEST234C => TEST234Y - TEST GR7 THE HIGH SIDE NIBBLES
*  TEST234D => TEST234Y - TEST GR7 THE HIGH SIDE NIBBLES
*  TEST234E => TEST234Y - TEST GR7 THE HIGH SIDE NIBBLES
*  TEST234F => TEST234Y - TEST GR7 THE HIGH SIDE NIBBLES
*  TEST2350 => TEST235W - TEST GR7 THE LOW SIDE NIBBLES
*  TEST2351 => TEST235W - TEST GR7 THE LOW SIDE NIBBLES
*  TEST2352 => TEST235W - TEST GR7 THE LOW SIDE NIBBLES
*  TEST2353 => TEST235W - TEST GR7 THE LOW SIDE NIBBLES
*  TEST2354 - TEST OF GOOD DECODE NET ENTRY POINT FOR FLR0 REG-TO-REG
*  TEST2355 => TEST235X - TEST FLR0 THE LOW SIDE NIBBLES
*  TEST2356 => TEST235X - TEST FLR0 THE LOW SIDE NIBBLES
*  TEST2357 => TEST235X - TEST FLR0 THE LOW SIDE NIBBLES
*  TEST2358 => TEST235X - TEST FLR0 THE LOW SIDE NIBBLES
*  TEST2359 - TEST OF GOOD DECODE NET ENTRY POINT FOR FLR1 REG-TO-REG
*  TEST235A => TEST235Y - TEST FLR1 THE LOW SIDE NIBBLES
*  TEST235B => TEST235Y - TEST FLR1 THE LOW SIDE NIBBLES
*  TEST235C => TEST235Y - TEST FLR1 THE LOW SIDE NIBBLES
*  TEST235D => TEST235Y - TEST FLR1 THE LOW SIDE NIBBLES
*  TEST235E - TEST OF GOOD DECODE NET ENTRY POINT FOR L INSTR, PB AS BASE REG
*  TEST235F => TEST235Z - TEST PB THE HIGH SIDE NIBBLES
*  TEST2360 => TEST235Z - TEST PB THE HIGH SIDE NIBBLES
* TEST2361 => TEST235Z - TEST PB THE HIGH SIDE NIBBLES
* TEST2362 => TEST235Z - TEST PB THE HIGH SIDE NIBBLES
* TEST2363 - TEST OF GOOD DECODE NET ENTRY POINT FOR L INSTR, SB AS BASE REG
* TEST2364 => TEST236W - TEST SB THE HIGH SIDE NIBBLES
* TEST2365 => TEST236W - TEST SB THE HIGH SIDE NIBBLES
* TEST2366 => TEST236W - TEST SB THE HIGH SIDE NIBBLES
* TEST2367 => TEST236X - TEST SB THE LOW SIDE NIBBLES
* TEST2368 => TEST236X - TEST SB THE LOW SIDE NIBBLES
* TEST2369 => TEST236X - TEST SB THE LOW SIDE NIBBLES
* TEST236A => TEST236X - TEST SB THE LOW SIDE NIBBLES
* TEST236B - TEST OF GOOD DECODE NET ENTRY POINT FOR L INSTR, LB AS BASE REG
* TEST236C => TEST236Y - TEST LB THE HIGH SIDE NIBBLES
* TEST236D => TEST236Y - TEST LB THE HIGH SIDE NIBBLES
* TEST236E => TEST236Y - TEST LB THE HIGH SIDE NIBBLES
* TEST236F => TEST236Z - TEST LB THE LOW SIDE NIBBLES
* TEST2370 => TEST236Z - TEST LB THE LOW SIDE NIBBLES
* TEST2371 => TEST236Z - TEST LB THE LOW SIDE NIBBLES
* TEST2372 => TEST236Z - TEST LB THE LOW SIDE NIBBLES
* TEST2373 - TEST OF GOOD DECODE NET ENTRY POINT FOR L INSTR, XB AS BASE REG
* TEST2374 => TEST237W - TEST XB THE HIGH SIDE NIBBLES
* TEST2375 => TEST237W - TEST XB THE HIGH SIDE NIBBLES
* TEST2376 => TEST237W - TEST XB THE HIGH SIDE NIBBLES
* TEST2377 => TEST237X - TEST XB THE LOW SIDE NIBBLES
* TEST2378 => TEST237X - TEST XB THE LOW SIDE NIBBLES
* TEST2379 => TEST237X - TEST XB THE LOW SIDE NIBBLES
* TEST237A => TEST237X - TEST XB THE LOW SIDE NIBBLES
* TEST237B - TEST_TNUM
* TEST23AB => TEST23AY - 64R,DIRECT,LONG REACH,PB | D,CHECK THE VALUE OF EAH
* TEST23AC => TEST23AY - 64R,DIRECT,LONG REACH,PB | D,CHECK THE VALUE OF EAH
* TEST23AD => TEST23AY - 64R,DIRECT,LONG REACH,PB | D,CHECK THE VALUE OF EAH
* TEST23AE => TEST23AZ - 64R,DIRECT,LONG REACH,PB | D,CHECK THE VALUE OF EAL
* TEST23AF => TEST23AZ - 64R,DIRECT,LONG REACH,PB | D,CHECK THE VALUE OF EAL
* TEST23B0 => TEST23AZ - 64R,DIRECT,LONG REACH,PB | D,CHECK THE VALUE OF EAL
* TEST23B1 => TEST23AZ - 64R,DIRECT,LONG REACH,PB | D,CHECK THE VALUE OF EAL
* TEST23B2 => TEST23BW - 64R,INDEXED,LONG REACH,PB|(D+X),CHECK THE VALUE OF EAH
* TEST23B3 => TEST23BW - 64R,INDEXED,LONG REACH,PB|(D+X),CHECK THE VALUE OF EAH
* TEST23B4 => TEST23BW - 64R,INDEXED,LONG REACH,PB|(D+X),CHECK THE VALUE OF EAH
* TEST23B5 => TEST23BX - 64R,INDEXED,LONG REACH,PB|(D+X),CHECK THE VALUE OF EAL
* TEST23B6 => TEST23BX - 64R,INDEXED,LONG REACH,PB|(D+X),CHECK THE VALUE OF EAL
* TEST23B7 => TEST23BX - 64R,INDEXED,LONG REACH,PB|(D+X),CHECK THE VALUE OF EAL
* TEST23B8 => TEST23BX - 64R,INDEXED,LONG REACH,PB|(D+X),CHECK THE VALUE OF EAL
* TEST23B9 => TEST23BY - 64R,INDEXED,LONG REACH,D <> 0,CHECK THE VALUE OF EAH
* TEST23BA => TEST23BY - 64R,INDEXED,LONG REACH,D <> 0,CHECK THE VALUE OF EAH
* TEST23BB => TEST23BY - 64R,INDEXED,LONG REACH,D <> 0,CHECK THE VALUE OF EAH
* TEST23BC - 64R,INDEXED,LONG REACH,D <> 0,CHECK THE VALUE OF EAL
* TEST23BD - 64R,INDEXED,LONG REACH,D <> 0,CHECK THE VALUE OF EAL
* TEST23BE - 64R,INDEXED,LONG REACH,D <> 0,CHECK THE VALUE OF EAL
* TEST23BF - 64R,INDEXED,LONG REACH,D <> 0,CHECK THE VALUE OF EAL
* TEST23C0 - TEST_TNUM
* TEST23E0 => TEST23EW - 64V, INDEXED, SB + D + X, CHECK EAH
* TEST23E1 => TEST23EW - 64V, INDEXED, SB + D + X, CHECK EAH
* TEST23E2 => TEST23EW - 64V, INDEXED, SB + D + X, CHECK EAH
* TEST23E3 - 64V, INDEXED, SB + D + X, CHECK EAL
* TEST23E4 - 64V, INDEXED, SB + D + X, CHECK EAL
* TEST23E5 - 64V, INDEXED, SB + D + X, CHECK EAL
* TEST23E6 - 64V, INDEXED, SB + D + X, CHECK EAL
* TEST23E7 => TEST23EX - 64V, INDEXED, LB + D + Y, CHECK EAH
* TEST23E8 => TEST23EX - 64V, INDEXED, LB + D + Y, CHECK EAH
* TEST23E9 => TEST23EX - 64V, INDEXED, LB + D + Y, CHECK EAH
* TEST23EA => TEST23EY - 64V, INDEXED, LB + D + Y, CHECK EAL
* TEST23EB => TEST23EY - 64V, INDEXED, LB + D + Y, CHECK EAL
*  TEST23EC => TEST23EY - 64V, INDEXED, LB + D + Y, CHECK EAL
*  TEST23ED => TEST23EY - 64V, INDEXED, LB + D + Y, CHECK EAL
*  TEST23EE => TEST23EZ - 32I, DIRECT, XB + D, CHECK EAH
*  TEST23EF => TEST23EZ - 32I, DIRECT, XB + D, CHECK EAH
*  TEST23F0 => TEST23EZ - 32I, DIRECT, XB + D, CHECK EAH
*  TEST23F1 => TEST23FW - 32I, DIRECT, XB + D, CHECK EAL
*  TEST23F2 => TEST23FW - 32I, DIRECT, XB + D, CHECK EAL
*  TEST23F3 => TEST23FW - 32I, DIRECT, XB + D, CHECK EAL
*  TEST23F4 => TEST23FW - 32I, DIRECT, XB + D, CHECK EAL
*  TEST23F5 => TEST23FX - 32I, INDIRECT, I(PB + D), CHECK EAH
*  TEST23F6 => TEST23FX - 32I, INDIRECT, I(PB + D), CHECK EAH
*  TEST23F7 => TEST23FX - 32I, INDIRECT, I(PB + D), CHECK EAH
*  TEST23F8 => TEST23FY - 32I, INDIRECT, I(PB + D), CHECK EAL
*  TEST23F9 => TEST23FY - 32I, INDIRECT, I(PB + D), CHECK EAL
*  TEST23FA => TEST23FY - 32I, INDIRECT, I(PB + D), CHECK EAL
*  TEST23FB => TEST23FY - 32I, INDIRECT, I(PB + D), CHECK EAL
*  TEST23FC => TEST23FZ - 32I, GENERAL REGISTER RELATIVE, D + GR0L, CHECK EAH
*  TEST23FD => TEST23FZ - 32I, GENERAL REGISTER RELATIVE, D + GR0L, CHECK EAH
*  TEST23FE => TEST23FZ - 32I, GENERAL REGISTER RELATIVE, D + GR0L, CHECK EAH
*  TEST23FF => TEST240V - 32I, GENERAL REGISTER RELATIVE, D + GR0L, CHECK EAL
*  TEST2400 => TEST240V - 32I, GENERAL REGISTER RELATIVE, D + GR0L, CHECK EAL
*  TEST2401 => TEST240V - 32I, GENERAL REGISTER RELATIVE, D + GR0L, CHECK EAL
*  TEST2402 => TEST240V - 32I, GENERAL REGISTER RELATIVE, D + GR0L, CHECK EAL
*  TEST2403 => TEST240W - 32I, GENERAL REGISTER RELATIVE, D + GR1L, CHECK EAH
*  TEST2404 => TEST240W - 32I, GENERAL REGISTER RELATIVE, D + GR1L, CHECK EAH
*  TEST2405 => TEST240W - 32I, GENERAL REGISTER RELATIVE, D + GR1L, CHECK EAH
*  TEST2406 => TEST240X - 32I, GENERAL REGISTER RELATIVE, D + GR1L, CHECK EAL
*  TEST2407 => TEST240X - 32I, GENERAL REGISTER RELATIVE, D + GR1L, CHECK EAL
*  TEST2408 => TEST240X - 32I, GENERAL REGISTER RELATIVE, D + GR1L, CHECK EAL
*  TEST2409 => TEST240X - 32I, GENERAL REGISTER RELATIVE, D + GR1L, CHECK EAL
*  TEST240A => TEST240Y - 32I, GENERAL REGISTER RELATIVE, D + GR2L, CHECK EAH
*  TEST240B => TEST240Y - 32I, GENERAL REGISTER RELATIVE, D + GR2L, CHECK EAH
*  TEST240C => TEST240Y - 32I, GENERAL REGISTER RELATIVE, D + GR2L, CHECK EAH
*  TEST240D => TEST240Z - 32I, GENERAL REGISTER RELATIVE, D + GR2L, CHECK EAL
*  TEST240E => TEST240Z - 32I, GENERAL REGISTER RELATIVE, D + GR2L, CHECK EAL
*  TEST240F => TEST240Z - 32I, GENERAL REGISTER RELATIVE, D + GR2L, CHECK EAL
*  TEST2410 => TEST240Z - 32I, GENERAL REGISTER RELATIVE, D + GR2L, CHECK EAL
*
```

SYSV4 continues vigorously through the addressing modes before moving onto aligned and unaligned cache reads. Two other functional areas are tested out here, the branch cache and the Segment Translation Lookaside Buffer (STLB). The following is an index of the tests contained in SYSV4.

```
*
*   INDEX OF TESTS IN SYSV4
*   -----------------------
*
*  TEST2411 => TEST241W - 32I, GENERAL REGISTER RELATIVE, D + GR3L, CHECK EAH
*  TEST2412 => TEST241W - 32I, GENERAL REGISTER RELATIVE, D + GR3L, CHECK EAH
*  TEST2413 => TEST241W - 32I, GENERAL REGISTER RELATIVE, D + GR3L, CHECK EAH
*  TEST2414 => TEST241X - 32I, GENERAL REGISTER RELATIVE, D + GR3L, CHECK EAL
*  TEST2415 => TEST241X - 32I, GENERAL REGISTER RELATIVE, D + GR3L, CHECK EAL
*  TEST2416 => TEST241X - 32I, GENERAL REGISTER RELATIVE, D + GR3L, CHECK EAL
*  TEST2417 => TEST241X - 32I, GENERAL REGISTER RELATIVE, D + GR3L, CHECK EAL
*  TEST2418 => TEST241Y - 32I, GENERAL REGISTER RELATIVE, D + GR4L, CHECK EAH
*  TEST2419 => TEST241Y - 32I, GENERAL REGISTER RELATIVE, D + GR4L, CHECK EAH
*  TEST241A => TEST241Y - 32I, GENERAL REGISTER RELATIVE, D + GR4L, CHECK EAH
*  TEST241B => TEST241Z - 32I, GENERAL REGISTER RELATIVE, D + GR4L, CHECK EAL
*  TEST241C => TEST241Z - 32I, GENERAL REGISTER RELATIVE, D + GR4L, CHECK EAL
* TEST241D => TEST241Z - 32I, GENERAL REGISTER RELATIVE, D + GR4L, CHECK EAL
* TEST241E => TEST241Z - 32I, GENERAL REGISTER RELATIVE, D + GR4L, CHECK EAL
* TEST241F => TEST242V - 32I, GENERAL REGISTER RELATIVE, D + GR5L, CHECK EAH
* TEST2420 => TEST242V - 32I, GENERAL REGISTER RELATIVE, D + GR5L, CHECK EAH
* TEST2421 => TEST242V - 32I, GENERAL REGISTER RELATIVE, D + GR5L, CHECK EAH
* TEST2422 => TEST242W - 32I, GENERAL REGISTER RELATIVE, D + GR5L, CHECK EAL
* TEST2423 => TEST242W - 32I, GENERAL REGISTER RELATIVE, D + GR5L, CHECK EAL
* TEST2424 => TEST242W - 32I, GENERAL REGISTER RELATIVE, D + GR5L, CHECK EAL
* TEST2425 => TEST242W - 32I, GENERAL REGISTER RELATIVE, D + GR5L, CHECK EAL
* TEST2426 => TEST242X - 32I, GENERAL REGISTER RELATIVE, D + GR6L, CHECK EAH
* TEST2427 => TEST242X - 32I, GENERAL REGISTER RELATIVE, D + GR6L, CHECK EAH
* TEST2428 => TEST242X - 32I, GENERAL REGISTER RELATIVE, D + GR6L, CHECK EAH
* TEST2429 => TEST242Y - 32I, GENERAL REGISTER RELATIVE, D + GR6L, CHECK EAL
* TEST242A => TEST242Y - 32I, GENERAL REGISTER RELATIVE, D + GR6L, CHECK EAL
* TEST242B => TEST242Y - 32I, GENERAL REGISTER RELATIVE, D + GR6L, CHECK EAL
* TEST242C => TEST242Y - 32I, GENERAL REGISTER RELATIVE, D + GR6L, CHECK EAL
* TEST242D => TEST242Z - 32I, GENERAL REGISTER RELATIVE, D + GR7L, CHECK EAH
* TEST242E => TEST242Z - 32I, GENERAL REGISTER RELATIVE, D + GR7L, CHECK EAH
* TEST242F => TEST242Z - 32I, GENERAL REGISTER RELATIVE, D + GR7L, CHECK EAH
* TEST2430 => TEST243V - 32I, GENERAL REGISTER RELATIVE, D + GR7L, CHECK EAL
* TEST2431 => TEST243V - 32I, GENERAL REGISTER RELATIVE, D + GR7L, CHECK EAL
* TEST2432 => TEST243V - 32I, GENERAL REGISTER RELATIVE, D + GR7L, CHECK EAL
* TEST2433 => TEST243V - 32I, GENERAL REGISTER RELATIVE, D + GR7L, CHECK EAL
* TEST2434 -  TEST_TNUM
* TEST2454 => TEST245W - 32I, INDEXED, BR + D + GR1H, CHECK EAH
* TEST2455 => TEST245W - 32I, INDEXED, BR + D + GR1H, CHECK EAH
* TEST2456 => TEST245W - 32I, INDEXED, BR + D + GR1H, CHECK EAH
* TEST2457 => TEST245X - 32I, INDEXED, BR + D + GR1H, CHECK EAL
* TEST2458 => TEST245X - 32I, INDEXED, BR + D + GR1H, CHECK EAL
* TEST2459 => TEST245X - 32I, INDEXED, BR + D + GR1H, CHECK EAL
* TEST245A => TEST245X - 32I, INDEXED, BR + D + GR1H, CHECK EAL
* TEST245B => TEST245Y - 32I, INDEXED, BR + D + GR2H, CHECK EAH
* TEST245C => TEST245Y - 32I, INDEXED, BR + D + GR2H, CHECK EAH
* TEST245D => TEST245Y - 32I, INDEXED, BR + D + GR2H, CHECK EAH
* TEST245E => TEST245Z - 32I, INDEXED, BR + D + GR2H, CHECK EAL
* TEST245F => TEST245Z - 32I, INDEXED, BR + D + GR2H, CHECK EAL
* TEST2460 => TEST245Z - 32I, INDEXED, BR + D + GR2H, CHECK EAL
* TEST2461 => TEST245Z - 32I, INDEXED, BR + D + GR2H, CHECK EAL
* TEST2462 => TEST246W - 32I, INDEXED, BR + D + GR3H, CHECK EAH
* TEST2463 => TEST246W - 32I, INDEXED, BR + D + GR3H, CHECK EAH
* TEST2464 => TEST246W - 32I, INDEXED, BR + D + GR3H, CHECK EAH
* TEST2465 => TEST246X - 32I, INDEXED, BR + D + GR3H, CHECK EAL
* TEST2466 => TEST246X - 32I, INDEXED, BR + D + GR3H, CHECK EAL
* TEST2467 => TEST246X - 32I, INDEXED, BR + D + GR3H, CHECK EAL
* TEST2468 => TEST246X - 32I, INDEXED, BR + D + GR3H, CHECK EAL
* TEST2469 => TEST246Y - 32I, INDEXED, BR + D + GR4H, CHECK EAH
* TEST246A => TEST246Y - 32I, INDEXED, BR + D + GR4H, CHECK EAH
* TEST246B => TEST246Y - 32I, INDEXED, BR + D + GR4H, CHECK EAH
* TEST246C => TEST246Z - 32I, INDEXED, BR + D + GR4H, CHECK EAL
* TEST246D => TEST246Z - 32I, INDEXED, BR + D + GR4H, CHECK EAL
* TEST246E => TEST246Z - 32I, INDEXED, BR + D + GR4H, CHECK EAL
* TEST246F => TEST246Z - 32I, INDEXED, BR + D + GR4H, CHECK EAL
* TEST2470 => TEST247V - 32I, INDEXED, BR + D + GR5H, CHECK EAH
* TEST2471 => TEST247V - 32I, INDEXED, BR + D + GR5H, CHECK EAH
* TEST2472 => TEST247V - 32I, INDEXED, BR + D + GR5H, CHECK EAH
* TEST2473 => TEST247W - 32I, INDEXED, BR + D + GR5H, CHECK EAL
* TEST2474 => TEST247W - 32I, INDEXED, BR + D + GR5H, CHECK EAL
* TEST2475 => TEST247W - 32I, INDEXED, BR + D + GR5H, CHECK EAL
* TEST2476 => TEST247W - 32I, INDEXED, BR + D + GR5H, CHECK EAL
* TEST2477 => TEST247X - 32I, INDEXED, BR + D + GR6H, CHECK EAH
* TEST2478 => TEST247X - 32I, INDEXED, BR + D + GR6H, CHECK EAH
*  TEST2479 => TEST247X - 32I, INDEXED, BR + D + GR6H, CHECK EAH
*  TEST247A => TEST247Y - 32I, INDEXED, BR + D + GR6H, CHECK EAL
*  TEST247B => TEST247Y - 32I, INDEXED, BR + D + GR6H, CHECK EAL
*  TEST247C => TEST247Y - 32I, INDEXED, BR + D + GR6H, CHECK EAL
*  TEST247D => TEST247Y - 32I, INDEXED, BR + D + GR6H, CHECK EAL
*  TEST247E => TEST247Z - 32I, INDEXED, BR + D + GR7H, CHECK EAH
*  TEST247F => TEST247Z - 32I, INDEXED, BR + D + GR7H, CHECK EAH
*  TEST2480 => TEST247Z - 32I, INDEXED, BR + D + GR7H, CHECK EAH
*  TEST2481 => TEST248V - 32I, INDEXED, BR + D + GR7H, CHECK EAL
*  TEST2482 => TEST248V - 32I, INDEXED, BR + D + GR7H, CHECK EAL
*  TEST2483 => TEST248V - 32I, INDEXED, BR + D + GR7H, CHECK EAL
*  TEST2484 => TEST248V - 32I, INDEXED, BR + D + GR7H, CHECK EAL
*  TEST2485 => TEST247W - 32I, IND, POSTINDEXED, I(BR + D) + GR1H, CHECK EAH
*  TEST2486 => TEST247W - 32I, IND, POSTINDEXED, I(BR + D) + GR1H, CHECK EAH
*  TEST2487 => TEST247W - 32I, IND, POSTINDEXED, I(BR + D) + GR1H, CHECK EAH
*  TEST2488 => TEST248X - 32I, IND, POSTINDEXED, I(BR + D) + GR1H, CHECK EAL
*  TEST2489 => TEST248X - 32I, IND, POSTINDEXED, I(BR + D) + GR1H, CHECK EAL
*  TEST248A => TEST248X - 32I, IND, POSTINDEXED, I(BR + D) + GR1H, CHECK EAL
*  TEST248B => TEST248X - 32I, IND, POSTINDEXED, I(BR + D) + GR1H, CHECK EAL
*  TEST248C => TEST247Y - 32I, IND, POSTINDEXED, I(BR + D) + GR2H, CHECK EAH
*  TEST248D => TEST247Y - 32I, IND, POSTINDEXED, I(BR + D) + GR2H, CHECK EAH
*  TEST248E => TEST247Y - 32I, IND, POSTINDEXED, I(BR + D) + GR2H, CHECK EAH
*  TEST248F => TEST248Z - 32I, IND, POSTINDEXED, I(BR + D) + GR2H, CHECK EAL
*  TEST2490 => TEST248Z - 32I, IND, POSTINDEXED, I(BR + D) + GR2H, CHECK EAL
*  TEST2491 => TEST248Z - 32I, IND, POSTINDEXED, I(BR + D) + GR2H, CHECK EAL
*  TEST2492 => TEST248Z - 32I, IND, POSTINDEXED, I(BR + D) + GR2H, CHECK EAL
*  TEST2493 => TEST249W - 32I, IND, POSTINDEXED, I(BR + D) + GR3H, CHECK EAH
*  TEST2494 => TEST249W - 32I, IND, POSTINDEXED, I(BR + D) + GR3H, CHECK EAH
*  TEST2495 => TEST249W - 32I, IND, POSTINDEXED, I(BR + D) + GR3H, CHECK EAH
*  TEST2496 => TEST249X - 32I, IND, POSTINDEXED, I(BR + D) + GR3H, CHECK EAL
*  TEST2497 => TEST249X - 32I, IND, POSTINDEXED, I(BR + D) + GR3H, CHECK EAL
*  TEST2498 => TEST249X - 32I, IND, POSTINDEXED, I(BR + D) + GR3H, CHECK EAL
*  TEST2499 => TEST249X - 32I, IND, POSTINDEXED, I(BR + D) + GR3H, CHECK EAL
*  TEST249A => TEST249Y - 32I, IND, POSTINDEXED, I(BR + D) + GR4H, CHECK EAH
*  TEST249B => TEST249Y - 32I, IND, POSTINDEXED, I(BR + D) + GR4H, CHECK EAH
*  TEST249C => TEST249Y - 32I, IND, POSTINDEXED, I(BR + D) + GR4H, CHECK EAH
*  TEST249D => TEST249Z - 32I, IND, POSTINDEXED, I(BR + D) + GR4H, CHECK EAL
*  TEST249E => TEST249Z - 32I, IND, POSTINDEXED, I(BR + D) + GR4H, CHECK EAL
*  TEST249F => TEST249Z - 32I, IND, POSTINDEXED, I(BR + D) + GR4H, CHECK EAL
*  TEST24A0 => TEST249Z - 32I, IND, POSTINDEXED, I(BR + D) + GR4H, CHECK EAL
*  TEST24A1 => TEST24AV - 32I, IND, POSTINDEXED, I(BR + D) + GR5H, CHECK EAH
*  TEST24A2 => TEST24AV - 32I, IND, POSTINDEXED, I(BR + D) + GR5H, CHECK EAH
*  TEST24A3 => TEST24AV - 32I, IND, POSTINDEXED, I(BR + D) + GR5H, CHECK EAH
*  TEST24A4 => TEST24AW - 32I, IND, POSTINDEXED, I(BR + D) + GR5H, CHECK EAL
*  TEST24A5 => TEST24AW - 32I, IND, POSTINDEXED, I(BR + D) + GR5H, CHECK EAL
*  TEST24A6 => TEST24AW - 32I, IND, POSTINDEXED, I(BR + D) + GR5H, CHECK EAL
*  TEST24A7 => TEST24AW - 32I, IND, POSTINDEXED, I(BR + D) + GR5H, CHECK EAL
*  TEST24A8 => TEST24AX - 32I, IND, POSTINDEXED, I(BR + D) + GR6H, CHECK EAH
*  TEST24A9 => TEST24AX - 32I, IND, POSTINDEXED, I(BR + D) + GR6H, CHECK EAH
*  TEST24AA => TEST24AX - 32I, IND, POSTINDEXED, I(BR + D) + GR6H, CHECK EAH
*  TEST24AB => TEST24AY - 32I, IND, POSTINDEXED, I(BR + D) + GR6H, CHECK EAL
*  TEST24AC => TEST24AY - 32I, IND, POSTINDEXED, I(BR + D) + GR6H, CHECK EAL
*  TEST24AD => TEST24AY - 32I, IND, POSTINDEXED, I(BR + D) + GR6H, CHECK EAL
*  TEST24AE => TEST24AY - 32I, IND, POSTINDEXED, I(BR + D) + GR6H, CHECK EAL
*  TEST24AF => TEST24AZ - 32I, IND, POSTINDEXED, I(BR + D) + GR7H, CHECK EAH
*  TEST24B0 => TEST24AZ - 32I, IND, POSTINDEXED, I(BR + D) + GR7H, CHECK EAH
*  TEST24B1 => TEST24AZ - 32I, IND, POSTINDEXED, I(BR + D) + GR7H, CHECK EAH
*  TEST24B2 => TEST24BW - 32I, IND, POSTINDEXED, I(BR + D) + GR7H, CHECK EAL
*  TEST24B3 => TEST24BW - 32I, IND, POSTINDEXED, I(BR + D) + GR7H, CHECK EAL
*  TEST24B4 => TEST24BW - 32I, IND, POSTINDEXED, I(BR + D) + GR7H, CHECK EAL
*  TEST24B5 => TEST24BW - 32I, IND, POSTINDEXED, I(BR + D) + GR7H, CHECK EAL
```

* TEST24B6 – TEST_TNUM
* TEST2506 => TEST250X – 32I, IND, PREINDEXED, I(BR + D + GR1H), CHECK EAH
* TEST2507 => TEST250X – 32I, IND, PREINDEXED, I(BR + D + GR1H), CHECK EAH
* TEST2508 => TEST250X – 32I, IND, PREINDEXED, I(BR + D + GR1H), CHECK EAH
* TEST2509 => TEST250Y – 32I, IND, PREINDEXED, I(BR + D + GR1H), CHECK EAL
* TEST250A => TEST250Y – 32I, IND, PREINDEXED, I(BR + D + GR1H), CHECK EAL
* TEST250B => TEST250Y – 32I, IND, PREINDEXED, I(BR + D + GR1H), CHECK EAL
* TEST250C => TEST250Y – 32I, IND, PREINDEXED, I(BR + D + GR1H), CHECK EAL
* TEST250D => TEST250Z – 32I, IND, PREINDEXED, I(BR + D + GR2H), CHECK EAH
* TEST250E => TEST250Z – 32I, IND, PREINDEXED, I(BR + D + GR2H), CHECK EAH
* TEST250F => TEST250Z – 32I, IND, PREINDEXED, I(BR + D + GR2H), CHECK EAH
* TEST2510 => TEST251V – 32I, IND, PREINDEXED, I(BR + D + GR2H), CHECK EAL
* TEST2511 => TEST251V – 32I, IND, PREINDEXED, I(BR + D + GR2H), CHECK EAL
* TEST2512 => TEST251V – 32I, IND, PREINDEXED, I(BR + D + GR2H), CHECK EAL
* TEST2513 => TEST251V – 32I, IND, PREINDEXED, I(BR + D + GR2H), CHECK EAL
* TEST2514 => TEST251W – 32I, IND, PREINDEXED, I(BR + D + GR3H), CHECK EAH
* TEST2515 => TEST251W – 32I, IND, PREINDEXED, I(BR + D + GR3H), CHECK EAH
* TEST2516 => TEST251W – 32I, IND, PREINDEXED, I(BR + D + GR3H), CHECK EAH
* TEST2517 => TEST251X – 32I, IND, PREINDEXED, I(BR + D + GR3H), CHECK EAL
* TEST2518 => TEST251X – 32I, IND, PREINDEXED, I(BR + D + GR3H), CHECK EAL
* TEST2519 => TEST251X – 32I, IND, PREINDEXED, I(BR + D + GR3H), CHECK EAL
* TEST251A => TEST251X – 32I, IND, PREINDEXED, I(BR + D + GR3H), CHECK EAL
* TEST251B => TEST251Y – 32I, IND, PREINDEXED, I(BR + D + GR4H), CHECK EAH
* TEST251C => TEST251Y – 32I, IND, PREINDEXED, I(BR + D + GR4H), CHECK EAH
* TEST251D => TEST251Y – 32I, IND, PREINDEXED, I(BR + D + GR4H), CHECK EAH
* TEST251E => TEST251Z – 32I, IND, PREINDEXED, I(BR + D + GR4H), CHECK EAL
* TEST251F => TEST251Z – 32I, IND, PREINDEXED, I(BR + D + GR4H), CHECK EAL
* TEST2520 => TEST251Z – 32I, IND, PREINDEXED, I(BR + D + GR4H), CHECK EAL
* TEST2521 => TEST251Z – 32I, IND, PREINDEXED, I(BR + D + GR4H), CHECK EAL
* TEST2522 => TEST252W – 32I, IND, PREINDEXED, I(BR + D + GR5H), CHECK EAH
* TEST2523 => TEST252W – 32I, IND, PREINDEXED, I(BR + D + GR5H), CHECK EAH
* TEST2524 => TEST252W – 32I, IND, PREINDEXED, I(BR + D + GR5H), CHECK EAH
* TEST2525 => TEST252X – 32I, IND, PREINDEXED, I(BR + D + GR5H), CHECK EAL
* TEST2526 => TEST252X – 32I, IND, PREINDEXED, I(BR + D + GR5H), CHECK EAL
* TEST2527 => TEST252X – 32I, IND, PREINDEXED, I(BR + D + GR5H), CHECK EAL
* TEST2528 => TEST252X – 32I, IND, PREINDEXED, I(BR + D + GR5H), CHECK EAL
* TEST2529 => TEST252Y – 32I, IND, PREINDEXED, I(BR + D + GR6H), CHECK EAH
* TEST252A => TEST252Y – 32I, IND, PREINDEXED, I(BR + D + GR6H), CHECK EAH
* TEST252B => TEST252Y – 32I, IND, PREINDEXED, I(BR + D + GR6H), CHECK EAH
* TEST252C => TEST252Z – 32I, IND, PREINDEXED, I(BR + D + GR6H), CHECK EAL
* TEST252D => TEST252Z – 32I, IND, PREINDEXED, I(BR + D + GR6H), CHECK EAL
* TEST252E => TEST252Z – 32I, IND, PREINDEXED, I(BR + D + GR6H), CHECK EAL
* TEST252F => TEST252Z – 32I, IND, PREINDEXED, I(BR + D + GR6H), CHECK EAL
* TEST2530 => TEST253V – 32I, IND, PREINDEXED, I(BR + D + GR7H), CHECK EAH
* TEST2531 => TEST253V – 32I, IND, PREINDEXED, I(BR + D + GR7H), CHECK EAH
* TEST2532 => TEST253V – 32I, IND, PREINDEXED, I(BR + D + GR7H), CHECK EAH
* TEST2533 => TEST253W – 32I, IND, PREINDEXED, I(BR + D + GR7H), CHECK EAL
* TEST2534 => TEST253W – 32I, IND, PREINDEXED, I(BR + D + GR7H), CHECK EAL
* TEST2535 => TEST253W – 32I, IND, PREINDEXED, I(BR + D + GR7H), CHECK EAL
* TEST2536 => TEST253W – 32I, IND, PREINDEXED, I(BR + D + GR7H), CHECK EAL
* TEST2537 – TEST_TNUM
* TEST2577 – 32R, DIRECT, SECTOR 0 REL., CHECK DECODE ENTRY POINT
* TEST2578 => TEST257X – 32R, DIRECT, SECTOR 0 REL., PBH | 0 + D, CHECK EAH
* TEST2579 => TEST257X – 32R, DIRECT, SECTOR 0 REL., PBH | 0 + D, CHECK EAH
* TEST257A => TEST257X – 32R, DIRECT, SECTOR 0 REL., PBH | 0 + D, CHECK EAH
* TEST257B => TEST257Y – 32R, DIRECT, SECTOR 0 REL., PBH | 0 + D, CHECK EAL
* TEST257C => TEST257Y – 32R, DIRECT, SECTOR 0 REL., PBH | 0 + D, CHECK EAL
* TEST257D => TEST257Y – 32R, DIRECT, SECTOR 0 REL., PBH | 0 + D, CHECK EAL
* TEST257E => TEST257Y – 32R, DIRECT, SECTOR 0 REL., PBH | 0 + D, CHECK EAL
* TEST257F – 32R, DIRECT, PROCEDURE REL., CHECK DECODE ENTRY POINT
* TEST2580 => TEST258W – 32R, DIRECT, PROC. REL., PBH | RPL + D + 1, CHECK EAH

```
*  TEST2581 => TEST258W - 32R, DIRECT, PROC. REL., PBH | RPL + D + 1, CHECK EAH
*  TEST2582 => TEST258W - 32R, DIRECT, PROC. REL., PBH | RPL + D + 1, CHECK EAH
*  TEST2583 => TEST258X - 32R, DIRECT, PROC. REL., PBH | RPL + D + 1, CHECK EAL
*  TEST2584 => TEST258X - 32R, DIRECT, PROC. REL., PBH | RPL + D + 1, CHECK EAL
*  TEST2585 => TEST258X - 32R, DIRECT, PROC. REL., PBH | RPL + D + 1, CHECK EAL
*  TEST2586 => TEST258X - 32R, DIRECT, PROC. REL., PBH | RPL + D + 1, CHECK EAL
*  TEST2587 => TEST258Y - 32R, DIRECT, PROC. REL., PBH | RPL + D + 1, CHECK EAL
*  TEST2588 => TEST258Y - 32R, DIRECT, PROC. REL., PBH | RPL + D + 1, CHECK EAL
*  TEST2589 => TEST258Y - 32R, DIRECT, PROC. REL., PBH | RPL + D + 1, CHECK EAL
*  TEST258A => TEST258Y - 32R, DIRECT, PROC. REL., PBH | RPL + D + 1, CHECK EAL
*  TEST258B - 32R, INDEXED, SECTOR 0 REL., CHECK DECODE ENTRY POINT
*  TEST258C => TEST258Z - 32R, INDEXED, SECTOR 0 REL., PBH | D + X, CHECK EAH
*  TEST258D => TEST258Z - 32R, INDEXED, SECTOR 0 REL., PBH | D + X, CHECK EAH
*  TEST258E => TEST258Z - 32R, INDEXED, SECTOR 0 REL., PBH | D + X, CHECK EAH
*  TEST258F => TEST259V - 32R, INDEXED, SECTOR 0 REL., PBH | D + X, CHECK EAL
*  TEST2590 => TEST259V - 32R, INDEXED, SECTOR 0 REL., PBH | D + X, CHECK EAL
*  TEST2591 => TEST259V - 32R, INDEXED, SECTOR 0 REL., PBH | D + X, CHECK EAL
*  TEST2592 => TEST259V - 32R, INDEXED, SECTOR 0 REL., PBH | D + X, CHECK EAL
*  TEST2593 - 32R, INDEXED, PROCEDURE REL., CHECK DECODE ENTRY POINT
*  TEST2594 => TEST259W - 32R, INDEXED, PROC. REL., PBH | RPL+D+X+1, CHECK EAH
*  TEST2595 => TEST259W - 32R, INDEXED, PROC. REL., PBH | RPL+D+X+1, CHECK EAH
*  TEST2596 => TEST259W - 32R, INDEXED, PROC. REL., PBH | RPL+D+X+1, CHECK EAH
*  TEST2597 => TEST259X - 32R, INDEXED, PROC. REL., PBH | RPL+D+X+1, CHECK EAL
*  TEST2598 => TEST259X - 32R, INDEXED, PROC. REL., PBH | RPL+D+X+1, CHECK EAL
*  TEST2599 => TEST259X - 32R, INDEXED, PROC. REL., PBH | RPL+D+X+1, CHECK EAL
*  TEST259A => TEST259X - 32R, INDEXED, PROC. REL., PBH | RPL+D+X+1, CHECK EAL
*  TEST259B => TEST259Y - 32R, INDEXED, PROC. REL., PBH | RPL+D+X+1, CHECK EAL
*  TEST259C => TEST259Y - 32R, INDEXED, PROC. REL., PBH | RPL+D+X+1, CHECK EAL
*  TEST259D => TEST259Y - 32R, INDEXED, PROC. REL., PBH | RPL+D+X+1, CHECK EAL
*  TEST259E => TEST259Y - 32R, INDEXED, PROC. REL., PBH | RPL+D+X+1, CHECK EAL
*  TEST259F - TEST_TNUM
*  TEST25CF - 16S, SECTOR 0 REL., CHECK DECODE ENTRY POINT
*  TEST25D0 => TEST25DW - 16S, SECTOR 0 REL., PBH | 0 + D, CHECK EAH
*  TEST25D1 => TEST25DW - 16S, SECTOR 0 REL., PBH | 0 + D, CHECK EAH
*  TEST25D2 => TEST25DW - 16S, SECTOR 0 REL., PBH | 0 + D, CHECK EAH
*  TEST25D3 => TEST25DX - 16S, SECTOR 0 REL., PBH | 0 + D, CHECK EAL
*  TEST25D4 => TEST25DX - 16S, SECTOR 0 REL., PBH | 0 + D, CHECK EAL
*  TEST25D5 => TEST25DX - 16S, SECTOR 0 REL., PBH | 0 + D, CHECK EAL
*  TEST25D6 => TEST25DX - 16S, SECTOR 0 REL., PBH | 0 + D, CHECK EAL
*  TEST25D7 - 16S, CURRENT SECTOR REL., CHECK DECODE ENTRY POINT
*  TEST25D8 => TEST25DY - 16S, CUR. SECT. REL., PBH | RPL[01:07] | D, CHECK EAH
*  TEST25D9 => TEST25DY - 16S, CUR. SECT. REL., PBH | RPL[01:07] | D, CHECK EAH
*  TEST25DA => TEST25DY - 16S, CUR. SECT. REL., PBH | RPL[01:07] | D, CHECK EAH
*  TEST25DB => TEST25DZ - 16S, CUR. SECT. REL., PBH | RPL[01:07] | D, CHECK EAL
*  TEST25DC => TEST25DZ - 16S, CUR. SECT. REL., PBH | RPL[01:07] | D, CHECK EAL
*  TEST25DD => TEST25DZ - 16S, CUR. SECT. REL., PBH | RPL[01:07] | D, CHECK EAL
*  TEST25DE => TEST25DZ - 16S, CUR. SECT. REL., PBH | RPL[01:07] | D, CHECK EAL
*  TEST25DF - 16S, SECTOR 0 REL., INDEXED, CHECK DECODE ENTRY POINT
*  TEST25E0 => TEST25EW - 16S, SECT. 0 REL., INDEXED, PBH | 0 + D + X, CHECK EAH
*  TEST25E1 => TEST25EW - 16S, SECT. 0 REL., INDEXED, PBH | 0 + D + X, CHECK EAH
*  TEST25E2 => TEST25EW - 16S, SECT. 0 REL., INDEXED, PBH | 0 + D + X, CHECK EAH
*  TEST25E3 => TEST25EX - 16S, SECT. 0 REL., INDEXED, PBH | 0 + D + X, CHECK EAL
*  TEST25E4 => TEST25EX - 16S, SECT. 0 REL., INDEXED, PBH | 0 + D + X, CHECK EAL
*  TEST25E5 => TEST25EX - 16S, SECT. 0 REL., INDEXED, PBH | 0 + D + X, CHECK EAL
*  TEST25E6 => TEST25EX - 16S, SECT. 0 REL., INDEXED, PBH | 0 + D + X, CHECK EAL
*  TEST25E7 - 16S, CURRENT SECTOR REL., CHECK DECODE ENTRY POINT
*  TEST25E8 => TEST25EY - 16S, CUR. SECT. REL., PBH |(RPL[01:07]|D)+X, CHECK EAH
*  TEST25E9 => TEST25EY - 16S, CUR. SECT. REL., PBH |(RPL[01:07]|D)+X, CHECK EAH
*  TEST25EA => TEST25EY - 16S, CUR. SECT. REL., PBH |(RPL[01:07]|D)+X, CHECK EAH
*  TEST25EB => TEST25EZ - 16S, CUR. SECT. REL., PBH |(RPL[01:07]|D)+X, CHECK EAL
*  TEST25EC => TEST25EZ - 16S, CUR. SECT. REL., PBH |(RPL[01:07]|D)+X, CHECK EAL
*  TEST25ED => TEST25EZ - 16S, CUR. SECT. REL., PBH |(RPL[01:07]|D)+X, CHECK EAL
*  TEST25EE => TEST25EZ - 16S, CUR. SECT. REL., PBH |(RPL[01:07]|D)+X, CHECK EAL
*  TEST25EF - TEST_TNUM
*  TEST262F - 64V, DIRECT, CHECK DECODE ENTRY POINT, LDX SHORT
*  TEST2630 => TEST263W - 64V, PC + D + 1, CHECK EAL, POSITIVE DISPLACEMENT
*  TEST2631 => TEST263W - 64V, PC + D + 1, CHECK EAL, POSITIVE DISPLACEMENT
*  TEST2632 => TEST263W - 64V, PC + D + 1, CHECK EAL, POSITIVE DISPLACEMENT
*  TEST2633 => TEST263W - 64V, PC + D + 1, CHECK EAL, POSITIVE DISPLACEMENT
*  TEST2634 - 64V, DIRECT, CHECK DECODE ENTRY POINT, LDA SHORT
*  TEST2635 => TEST263X - 64V, PC + D + 1, CHECK EAL, NEGATIVE DISPLACEMENT
*  TEST2636 => TEST263X - 64V, PC + D + 1, CHECK EAL, NEGATIVE DISPLACEMENT
*  TEST2637 => TEST263X - 64V, PC + D + 1, CHECK EAL, NEGATIVE DISPLACEMENT
*  TEST2638 => TEST263X - 64V, PC + D + 1, CHECK EAL, NEGATIVE DISPLACEMENT
*  TEST2639 - 64V, DIRECT, CHECK DECODE ENTRY POINT, LDL
*  TEST263A => TEST263Y - 64V, PBH | D, CHECK EAL, ALIGNED
*  TEST263B => TEST263Y - 64V, PBH | D, CHECK EAL, ALIGNED
*  TEST263C => TEST263Y - 64V, PBH | D, CHECK EAL, ALIGNED
*  TEST263D => TEST263Y - 64V, PBH | D, CHECK EAL, ALIGNED
*  TEST263E => TEST263Z - 64V, PBH | D, CHECK EAL, UNALIGNED
*  TEST263F => TEST263Z - 64V, PBH | D, CHECK EAL, UNALIGNED
*  TEST2640 => TEST263Z - 64V, PBH | D, CHECK EAL, UNALIGNED
*  TEST2641 => TEST263Z - 64V, PBH | D, CHECK EAL, UNALIGNED
*  TEST2642 - 64V, INDIRECT, CHECK DECODE ENTRY POINT, LDL
*  TEST2643 => TEST264W - 64V, I(PBH | D), CHECK EAH
*  TEST2644 => TEST264W - 64V, I(PBH | D), CHECK EAH
*  TEST2645 => TEST264W - 64V, I(PBH | D), CHECK EAH
*  TEST2646 => TEST264X - 32I, I(PBH | D), CHECK EAL
*  TEST2647 => TEST264X - 32I, I(PBH | D), CHECK EAL
*  TEST2648 => TEST264X - 32I, I(PBH | D), CHECK EAL
*  TEST2649 => TEST264X - 32I, I(PBH | D), CHECK EAL
*  TEST264A - 64V, INDIRECT, POINTER FAULT, CHECK EASH
*  TEST264B - 64V, INDIRECT, POINTER FAULT, CHECK EASL
*  TEST264C - TEST_TNUM
*  TEST2800 - 32-BIT, ALIGNED CACHE READ, BOTH ELEMENTS OF CACHE VALID.
*  TEST2801 - 32-BIT, ALIGNED CACHE READ, BOTH ELEMENTS OF CACHE VALID.
*  TEST2802 - 32-BIT, ALIGNED CACHE READ, 'A' ELEMENT OF CACHE VALID ONLY.
*  TEST2803 - 32-BIT, ALIGNED CACHE READ, 'A' ELEMENT OF CACHE VALID ONLY.
*  TEST2804 - 32-BIT, ALIGNED CACHE READ, 'B' ELEMENT OF CACHE VALID ONLY.
*  TEST2805 - 32-BIT, ALIGNED CACHE READ, 'B' ELEMENT OF CACHE VALID ONLY.
*  TEST2806 - 32-BIT, ALIGNED CACHE READ, BOTH ELEMENTS OF CACHE VALID.
*  TEST2807 - 32-BIT, ALIGNED CACHE READ, BOTH ELEMENTS OF CACHE VALID.
*  TEST2808 - 32-BIT, ALIGNED CACHE READ, 'A' ELEMENT OF CACHE VALID ONLY.
*  TEST2809 - 32-BIT, ALIGNED CACHE READ, 'A' ELEMENT OF CACHE VALID ONLY.
*  TEST280A - 16-BIT, ALIGNED CACHE READ, BOTH ELEMENTS OF CACHE VALID.
*  TEST280B - 16-BIT, ALIGNED CACHE READ, BOTH ELEMENTS OF CACHE VALID.
*  TEST280C - 16-BIT, UNALIGNED CACHE READ, BOTH ELEMENTS OF CACHE VALID.
*  TEST280D - 16-BIT, UNALIGNED CACHE READ, BOTH ELEMENTS OF CACHE VALID.
*  TEST280E - 16-BIT, ALIGNED CACHE READ, 'A' ELEMENT OF CACHE VALID ONLY.
*  TEST280F - 16-BIT, ALIGNED CACHE READ, 'A' ELEMENT OF CACHE VALID ONLY.
*  TEST2810 - 16-BIT, UNALIGNED CACHE READ, 'B' ELEMENT OF CACHE VALID ONLY.
*  TEST2811 - 16-BIT, UNALIGNED CACHE READ, 'B' ELEMENTS OF CACHE VALID ONLY.
*  TEST2812 - 32-BIT, UNALIGNED CACHE READ, ACROSS MOD4 BOUNDARY, 'B' ELEMENT
*             OF CACHE VALID ONLY, FOR BOTH MOD4 WORDS
*  TEST2813 - 32-BIT, UNALIGNED CACHE READ, ACROSS MOD4 BOUNDARY, 'B' ELEMENT
*             OF CACHE VALID ONLY, FOR BOTH MOD4 WORDS
*  TEST2814 - 32-BIT, UNALIGNED CACHE READ, ACROSS MOD4 BOUNDARY, ACROSS CACHE
*             ELEMENTS, 'B' ELEMENT OF CACHE VALID FOR FIRST MOD4 WORD; 'A'
*             ELEMENT OF CACHE VALID FOR SECOND MOD4 WORD
*  TEST2815 - 32-BIT, UNALIGNED CACHE READ, ACROSS MOD4 BOUNDARY, ACROSS CACHE
*             ELEMENTS, 'B' ELEMENT OF CACHE VALID FOR FIRST MOD4 WORD; 'A'
*             ELEMENT OF CACHE VALID FOR SECOND MOD4 WORD
*  TEST2816 - TEST_TNUM
*  TEST2830 - CHECK JUMP CONDITION CACHE UPDT
*             WRITE TO CACHEA -> CACHEB -> CACHEA check CUPDT toggles
*  TEST2831 - CHECK JUMP CONDITION CACHE UPDT
*             WRITE TO CACHEB -> CACHEA -> CACHEB check CUPDT toggles
*  TEST2832 - CHECK JUMP CONDITION CACHE UPDT
*             WRITE TO CACHEA -> CACHEA  check CUPDT toggles
*  TEST2833 - CHECK JUMP CONDITION CACHE UPDT
*             WRITE TO CACHEB -> CACHEB  check CUPDT toggles
*  TEST2834 - TEST_TNUM
*  TEST2840 => BRNCIRP - BRANCH CACHE ADDRESS= $00000000,DATA= EXPPAT,
*                        CHECK IRPL
*  TEST2841 => BRNCIRP - BRANCH CACHE ADDRESS= $FFFFFFFF||SETADDR,
*                        DATA= EXPPAT, CHECK IRPL
*  TEST2842 => BRNCIRP - BRANCH CACHE ADDRESS= 0||SETADDR,DATA= EXPPAT,
*                        CHECK IRPL
*  TEST2843 => BRNCIRP - BRANCH CACHE ADDRESS= IRPH5SET||SETADDR,DATA= EXPPAT,
*                        CHECK IRPL
*  TEST2844 => BRNCIRP - BRANCH CACHE ADDRESS=WALKING ONES,DATA=EXPPAT,
*                        CHECK IRPL
*  TEST2845 => BRNCIRP - BRANCH CACHE ADDRESS=WALKING ZEROS,DATA=EXPPAT,
*                        CHECK IRPL
*  TEST2846 => BRNCHIT - BRANCH CACHE ADDRESS= $00000000,DATA= EXPPAT,
*                        JUMP ON NBCHIT
*  TEST2847 => BRNCHIT - BRANCH CACHE ADDRESS= $FFFFFFFF||SETADDR,DATA= EXPPAT,
*                        JUMP ON NBCHIT
*  TEST2848 => BRNCHIT - BRANCH CACHE ADDRESS=0||SETADDR,DATA= EXPPAT,
*                        JUMP ON NBCHIT
*  TEST2849 => BRNCHIT - BRANCH CACHE ADDRESS=IRPH5SET||SETADDR,DATA= EXPPAT,
*                        JUMP ON NBCHIT
*  TEST284A => BRNCHIT - BRANCH CACHE ADDRESS=WALKING ONES,DATA=EXPPAT,
*                        JUMP ON NBCHIT
*  TEST284B => BRNCHIT - BRANCH CACHE ADDRESS=WALKING ZEROS,DATA=EXPPAT,
*                        JUMP ON NBCHIT
*  TEST284C - TEST_TNUM
*  TEST2850 - TEST INCREMENT OF IRP WITH ONE GOOD BRANCH
*  TEST2851 - TEST CRTN : CONDITION TRUE AND GOODBRANCH FALSE, TARGET MISCOMPARE
*  TEST2852 - TEST CRTN : CONDITION TRUE AND GOODBRANCH FALSE, SEG MISCOMPARE
*  TEST2853 - TEST CRTN : CONDITION FALSE AND BCHIT FALSE, INVLD SET
*  TEST2854 - TEST CRTN : CONDITION FALSE AND BCHIT FALSE, INDEX MISCOMPARE
*  TEST2855 - TEST CRTN : CONDITION FALSE AND BCHIT TRUE
*  TEST2856 - TEST CRTN : CONDITION TRUE AND GOODBRANCH
*  TEST2857 - TEST_TNUM
*  TEST2880 - DIAGNOSTIC READ OF STLB ELEMENT 'A' LBPA
*  TEST2881 - DIAGNOSTIC READ OF STLB ELEMENT 'B' LBPA
*  TEST2882 - DIAGNOSTIC READ OF STLB ELEMENT 'A' LPID
*  TEST2883 - DIAGNOSTIC READ OF ELEMENT 'B' LPID
*  TEST2884 - DIAGNOSTIC READ OF ELEMENT 'A' ACCESS RIGHTS
*  TEST2885 - DIAGNOSTIC READ OF ELEMENT 'B' ACCESS RIGHTS
*  TEST2886 - DIAGNOSTIC READ OF ELEMENT 'A' LBVA
*  TEST2887 - DIAGNOSTIC READ OF ELEMENT 'B' LBVA
*  TEST2888 - DIAGNOSTIC READ OF ELEMENT 'A' PURGE COUNT
*  TEST2889 - DIAGNOSTIC READ OF ELEMENT 'B' PURGE COUNT
*  TEST288A - READ OF SRMAL
*  TEST288B - READ OF SRMAL, CHECKING FLOPPED LBPA FROM CSS ON BBH
*  TEST288C - CHECK OF STLB UPDATE ALGORITHM
*  TEST288D - CHECK OF STLB HARD PARITY BITS (FRCA, FRCB)
*  TEST288E - CHECK OF STLB HASH
*  TEST288F - CHECK OF IOTLB/STLB ADDRESSING
*  TEST2890 - READ OF STLB VALID BITS (BPVLDA-,BPVLDB-)
*  TEST2891 - READ OF STLB PAGE MODIFIED BITS (BPUMODA,BPUMODB)
*  TEST2892 - CHECK OF FAVIOL LOGIC
*  TEST2893 - CHECK OF SHARED BIT FOR STLB ELEMENT 'A'
*  TEST2894 - CHECK OF SHARED BIT FOR STLB ELEMENT 'B'
*  TEST2895 - TEST_TNUM
*
```

Memory diagnostics are contained in SYSV5. The overlay performs a thorough test of the MC logic, starting with the basic data paths and going through the MT, ECCC and ECCU detection, and verification of the refresh circuitry. The memory arrays are partially tested. Time constraints prevent a thorough test of the memory arrays. (Such tests would take approximately 15 min/8 MB of memory.) The following is an index of the tests contained in SYSV5.

```
*
*   INDEX OF TESTS IN SYSV5
*   -----------------------
*
*  TEST4000 - BDL -> DATA WRITE BUFFER -> BDL
*  TEST4001 - BDH -> DATA WRITE BUFFER -> BDH
*  TEST4002 - BDH -> DATA WRITE BUFFER -> MD -> BDH
*  TEST4003 - BDL -> DATA WRITE BUFFER -> MD -> BDL
*  TEST4004 - BBH -> ADDR WRITE BUFFER -> MA -> BDH
*  TEST4005 - BBL -> ADDR WRITE BUFFER -> MA -> BDL
*  TEST4006 - TESTNUM
*  TEST401X - DATA WRITE BUFFER HIGH SIDE
*  TEST401Y - DATA WRITE BUFFER LOW SIDE
*  TEST402X - ADDRESS WRITE BUFFER BITS{17:24}
*  TEST402Y - ADDRESS WRITE BUFFER BITS{1:16}
*  TEST4028 - TESTNUM
*  TEST403X - WRITE BUFFER LOCATION ? HIT DETECT
*  TEST4034 - TESTNUM
*  TEST404X - VALID WRITE BUFFER LOCATION ? BIT 0 OR 2
*  TEST404Y - VALID WRITE BUFFER LOCATION ? BIT 1 OR 3
*  TEST4050 - WRITE BUFFER LOCATION 3 FREE ADDRESS
*  TEST4051 - WRITE BUFFER LOCATION 2 FREE ADDRESS
*  TEST4052 - WRITE BUFFER LOCATION 1 FREE ADDRESS
*  TEST4053 - WRITE BUFFER LOCATION 0 FREE ADDRESS
*  TEST4054 - TESTNUM
*  TEST4060 - MEMORY TIMER DIAGNOSTIC ROUTINE
*  TEST4061 - NO VALID BIT BYPASSES
*  TEST4062 - BOTH VALID BIT BYPASSES
*  TEST4063 - ONE VALID BIT BYPASS
*  TEST4064 - DATAV BEFORE MRDY TEST
*  TEST4065 - TESTNUM
*  TEST4070 - MA SHIFTER 4 MBYTE UNI MODE
*  TEST4071 - MA SHIFTER 16 MBYTE UNI MODE
*  TEST4072 - TESTNUM
*  TEST4080 - REFRESH TEST
*  TEST408V - 64 BIT MEMORY WRITE TEST
*  TEST408W - 32 BIT EVEN MEMORY WRITE TEST
*  TEST408X - 32 BIT ODD MEMORY WRITE TEST
*  TEST408Y - 16 BIT MEMORY WRITE TEST
*  TEST408Z - UNALIGNED 32 BIT MEMORY WRITE TEST
*  TEST408B - TESTNUM
*  TEST40C6 - 32-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT
*  TEST40C7 - 32-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT
*  TEST40C8 - 32-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT
*  TEST40C9 - 32-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT
*  TEST40CA - 32-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT
*  TEST40CB - 32-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT
*  TEST40CC - 32-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT
*  TEST40CD - 32-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT
*  TEST40CE - 32-BIT, UNALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT
*  TEST40CF - 32-BIT, UNALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT
*  TEST40D0 - 32-BIT, UBALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT
*  TEST40D1 - 32-BIT, UNALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT
*  TEST40D2 - 16-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT OF CACHE
*  TEST40D3 - 16-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT OF CACHE
*  TEST40D4 - 16-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT OF CACHE
*  TEST40D5 - 16-BIT, ALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT OF CACHE
*  TEST40D6 - 16-BIT, UNALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT OF CACHE
*  TEST40D7 - 16-BIT, UNALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT OF CACHE
*  TEST40D8 - 16-BIT, UNALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT OF CACHE
*  TEST40D9 - 16-BIT, UNALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT OF CACHE
*  TEST40DA - 32-BIT, UNALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT ACROSS MOD4 BOUNDARY, MISS ON
*             FIRST WORD
*  TEST40DB - 32-BIT, UNALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT ACROSS MOD4 BOUNDARY, MISS ON
*             FIRST WORD
*  TEST40DC - 32-BIT, UNALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT ACROSS MOD4 BOUNDARY, MISS ON
*             SECOND WORD
*  TEST40DD - 32-BIT, UNALIGNED CACHE MISS, BOTH ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT ACROSS MOD4 BOUNDARY, MISS ON
*             SECOND WORD
*  TEST40DE - 32-BIT, UNALIGNED CACHE MISS, 'A' ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT ACROSS MOD4 BOUNDARY, MISS ON
*             FIRST WORD
*  TEST40DF - 32-BIT, UNALIGNED CACHE MISS, 'A' ELEMENTS OF CACHE INVALID,
*             FORCE UPDATE ON 'A' ELEMENT ACROSS MOD4 BOUNDARY, MISS ON
*             FIRST WORD
*  TEST40E0 - 32-BIT, UNALIGNED CACHE MISS, 'B' ELEMENT OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT ACROSS MOD4 BOUNDARY, MISS ON
*             SECOND WORD
*  TEST40E1 - 32-BIT, UNALIGNED CACHE MISS, 'B' ELEMENT OF CACHE INVALID,
*             FORCE UPDATE ON 'B' ELEMENT ACROSS MOD4 BOUNDARY, MISS ON
*             SECOND WORD
*  TEST40E2 - 16-BIT, ALIGNED WRITE, BOTH ELEMENTS OF CACHE INVALID CACHE
*             SHOULD NOT GET UPDATED, DUE TO CACHE MISS
*  TEST40E3 - 16-BIT, ALIGNED WRITE, 'A' ELEMENT OF CACHE VALID CACHE
*             SHOULD GET UPDATED
*  TEST40E4 - 16-BIT, ALIGNED WRITE, BOTH ELEMENTS OF CACHE VALID BOTH
*             ELEMENTS CACHE SHOULD GET UPDATED
*  TEST40E5 - 32-BIT, ALIGNED WRITE, BOTH ELEMENTS OF CACHE INVALID CACHE
*             SHOULD NOT GET UPDATED
*  TEST40E6 - 32-BIT, ALIGNED WRITE, BOTH ELEMENTS OF CACHE INVALID CACHE
*             SHOULD NOT GET UPDATED
*  TEST40E7 - 32-BIT, ALIGNED WRITE, 'B' ELEMENT OF CACHE VALID CACHE
*             SHOULD GET UPDATED
*  TEST40E8 - 32-BIT, ALIGNED WRITE, 'B' ELEMENT OF CACHE VALID CACHE
*             SHOULD GET UPDATED
*  TEST40E9 - 32-BIT, UNALIGNED WRITE, BOTH ELEMENTS OF CACHE INVALID CACHE
*             SHOULD NOT GET UPDATED
*  TEST40EA - 32-BIT, UNALIGNED WRITE, BOTH ELEMENTS OF CACHE INVALID CACHE
*             SHOULD NOT GET UPDATED
*  TEST40EB - 32-BIT, UNALIGNED WRITE, 'A' ELEMENT OF CACHE VALID FIRST
*             16-BIT WORD. FIRST 16-BIT WORD OF CACHE SHOULD GET UPDATED,
*             NOT THE SECOND WORD
*  TEST40EC - 32-BIT, UNALIGNED WRITE, 'A' ELEMENT OF CACHE VALID FIRST
*             16-BIT WORD. FIRST 16-BIT WORD OF CACHE SHOULD GET UPDATED,
*             NOT THE SECOND WORD
*  TEST40ED - 32-BIT, UNALIGNED WRITE, 'A' ELEMENT OF CACHE VALID SECOND
*             16-BIT WORD. SECOND 16-BIT WORD OF CACHE SHOULD GET UPDATED,
*             NOT THE FIRST WORD
*  TEST40EE - 32-BIT, UNALIGNED WRITE, 'A' ELEMENT OF CACHE VALID SECOND
*             16-BIT WORD. SECOND 16-BIT WORD OF CACHE SHOULD GET UPDATED,
*             NOT THE FIRST WORD
*  TEST40EF - 32-BIT, UNALIGNED WRITE, 'A' ELEMENT OF CACHE VALID BOTH 16_BIT
*             WORDS. BOTH 16-BIT WORDS OF CACHE SHOULD GET UPDATED.
*  TEST40F0 - 32-BIT, UNALIGNED WRITE, 'A' ELEMENT OF CACHE VALID BOTH 16-BIT
*             WORDS. BOTH 16-BIT WORDS OF CACHE SHOULD GET UPDATED.
*  TEST40F1 - 32-BIT, ALIGNED CACHE READ, 'A' ELEMENT OF CACHE HAS BAD PARITY.
*             BOTH SIDES OF CACHE SHOULD GET UPDATED.
*  TEST40F2 - 32-BIT, ALIGNED CACHE READ, 'B' ELEMENT OF CACHE HAS BAD PARITY.
*             BOTH SIDES OF CACHE SHOULD GET UPDATED.
*  TEST40F3 - TEST_TNUM
*  TEST40F4 - CACHE MISS CAUSED BY LBPOWN+.
*  TEST40F5 - CACHE MISS CAUSED BY LBPOWN+.
*  TEST40F6 - TESTNUM
*  TEST4160 - ECCC JUMP CONDITION, EVEN ACCESS
*  TEST4161 - ECCC JUMP CONDITION, ODD ACCESS
*  TEST4162 - ECCU TRAP TEST, EVEN ACCESS
*  TEST4163 - ECCU TRAP TEST, ODD ACCESS
*  TEST4165 - TESTNUM
*  TEST418X - ECCC DATA TEST HIGH SIDE
*  TEST419X - ECCC DATA TEST LOW SIDE
*  TEST41A0 - TESTNUM
*  TEST41CX - ECCC SYNDROME TEST
*  TEST41E0 - ECCC ERROR ADDRESS TEST
*  TEST41E1 - ECCC RECYCLE TEST
*  TEST41E2 - TESTNUM
```

SYSV6 tests the basic I/O data paths, character and decimal instruction hardware, and the system timers. It also tests parity checking throughout the whole machine. The following is an index of the tests contained in SYSV6.

```
*
*   INDEX OF TESTS IN SYSV6
*   -----------------------
```

```
*
* TEST6000 - PARTIAL PRODUCT BUS
* TEST6001 - MULTIPLY TEST
* TEST6002 - DIVIDE TEST, ALTSELECT - ALHCOUT - LINK - RIE16
* TEST6003 - DIVIDE TEST, ALTSELECT - ALHCOUT - LINK - RIE16
* TEST6004 - DIVIDE TEST, ALTSELECT - ALHCOUT - LINK - RIE16
* TEST6005 - DIVIDE TEST, ALTSELECT - ALHCOUT - LINK - RIE16
* TEST6006 - TEST_TNUM
* TEST6020 - TEST OF ARGTSBIT - S BIT SET
* TEST6021 - TEST OF ARGTSBIT - S BIT RESET
* TEST6022 - TEST OF ARGTLBIT - L BIT SET
* TEST6023 - TEST OF ARGTLBIT - L BIT RESET
* TEST6024 - TEST OF ARGTBR1=0 AND ARGTBR2=0
* TEST6025 - TEST OF ARGTBR1=0 AND ARGTBR2=1
* TEST6026 - TEST OF ARGTBR1=1 AND ARGTBR2=0
* TEST6027 - TEST OF ARGTBR1=1 AND ARGTBR2=1
* TEST6028 - TEST OF NARGEBIT TRUE - XB E BIT = 0, BIT FIELD = 0
* TEST6029 - TEST OF NARGEBIT FALSE - XB E BIT = 1, BIT FIELD = 0
* TEST602A - TEST OF NARGEBIT FALSE - XB E BIT = 0, BIT FIELD = 8
* TEST602B - TEST OF NARGEBIT FALSE - XB E BIT = 0, BIT FIELD = 4
* TEST602C - TEST OF NARGEBIT FALSE - XB E BIT = 0, BIT FIELD = 2
* TEST602D - TEST OF NARGEBIT FALSE - XB E BIT = 0, BIT FIELD = 1
* TEST602E - XB AS GR(TBR)
* TEST602F - TLB AS GR(TBR)
* TEST6030 - TPB AS GR(TBR)
* TEST6031 - TSB AS GR(TBR)
* TEST6032 - TEST_TNUM
* TEST6040 - LCC HARDWARE, NO BYTE SWAP ON BBE, IAC CBYTESWAP NOT ASSERTED
*            RMAH04+ = 1
* TEST6041 - LCC HARDWARE,NO BYTE SWAP ON BBE, CBYTESWAP ASSERTED, RMAH04+ = 1
* TEST6042 - LCC HARDWARE, BYTE SWAP ON BBE, CBYTESWAP ASSERTED, RMAH04+ = 0
* TEST6043 - SCC HARDWARE, BDI MODE [H,L], HIGH SIDE, CSTORE NOT ASSERTED
*            RMAH04+ = 1
* TEST6044 - SCC HARDWARE, BDI MODE [H,L], LOW SIDE, CSTORE NOT ASSERTED
*            RMAH04+ = 1
* TEST6045 - SCC HARDWARE, BDI MODE [H,H], HIGH SIDE, CSTORE ASSERTED
*            RMAH04+ = 0
* TEST6046 - SCC HARDWARE, BDI MODE [H,H], LOW SIDE, CSTORE ASSERTED
*            RMAH04+ = 0
* TEST6047 - SCC LOGIC, BDI MODE [E,E], HIGH SIDE, CSTORE ASSERTED, RMAH04+ = 1
* TEST6048 - SCC LOGIC, BDI MODE [E,E], LOW SIDE, CSTORE ASSERTED, RMAH04+ = 1
* TEST6049 - TEST_TNUM
* TEST6100 => TEST610W - TEST OF ZMSTRT, MODE= AAAA, ZFF0&1=00, ZFF2&3=00
* TEST6101 => TEST610W - TEST OF ZMSTRT, MODE= AAAA, ZFF0&1=00, ZFF2&3=01
* TEST6102 => TEST610W - TEST OF ZMSTRT, MODE= AAAA, ZFF0&1=00, ZFF2&3=10
* TEST6103 => TEST610W - TEST OF ZMSTRT, MODE= AAAA, ZFF0&1=00, ZFF2&3=11
* TEST6104 => TEST610X - TEST OF ZMSTRT, MODE= BAAA, ZFF0&1=01, ZFF2&3=00
* TEST6109 => TEST610Y - TEST OF ZMSTRT, MODE= BBAA, ZFF0&1=10, ZFF2&3=01
* TEST610E => TEST610Z - TEST OF ZMSTRT, MODE= BBBA, ZFF0&1=11, ZFF2&3=10
* TEST6110 - TEST OF ZMSTRT, FXSRCDST=1
* TEST6111 => TEST611X - TEST OF ZMSTRT, FXSRCDST=0
* TEST6112 => TEST611X - TEST OF ZMSTRT, FXSRCDST=0
* TEST6113 => TEST611X - TEST OF ZMSTRT, FXSRCDST=0
* TEST6114 - TEST OF ZMSTRT, FXBITS=00
* TEST6115 - TEST OF ZMSTRT, FXBITS=01
* TEST6116 - TEST OF ZMSTRT, FXBITS=10
* TEST6117 - TEST OF ZMSTRT, FXBITS=11
* TEST6118 - TEST OF OBTAIN, MODE= BBBB
* TEST6119 - TEST OF OBTAIN, MODE= ABBB
* TEST611A - TEST_TNUM
* TEST6120 - TEST OF OBTAIN, MODE= AABB
```

* **TEST6121** – TEST OF OBTAIN, MODE= AAAB
* **TEST6122** – TEST OF UNLPCK, MODE= BABB
* **TEST6123** – TEST OF UNLPCK, MODE= BBAB
* **TEST6124** – TEST OF UNLPCK, MODE= BAAB
* **TEST6125** – TEST OF UNLPCK, MODE= BBBA
* **TEST6126** – TEST OF UNLPCK, MODE= BBAA
* **TEST6127** – TEST OF UNLPCK, MODE= BAAA
* **TEST6128** – TEST OF UNLUNP, MODE= BABB
* **TEST6129** – TEST OF UNLUNP, MODE= BBAB
* **TEST612A** – TEST OF UNLUNP, MODE= BAAB
* **TEST612B** – TEST OF UNLUNP, MODE= BBBA
* **TEST612C** – TEST OF UNLUNP, MODE= BBAA
* **TEST612D** – TEST OF UNLUNP, MODE= BAAA
* **TEST612E** – TEST OF ZSTRD=0
* **TEST612F** – TEST OF ZSTRD=1
* **TEST6130** – TEST OF ZBITS (ZFF01 ALIGNMENT), FZBITS=00, ZMFAST
* **TEST6131** – TEST OF ZBITS (ZFF01 ALIGNMENT), FZBITS=01, ZMFAST
* **TEST6132** – TEST OF ZBITS (ZFF01 ALIGNMENT), FZBITS=10, ZMFAST
* **TEST6133** – TEST OF ZBITS (ZFF01 ALIGNMENT), FZBITS=11, ZMFAST
* **TEST6134** – TEST_TNUM
* **TEST6300** – ASCII8 TEST, SET/RESET
* **TEST6301** – PACKER PROM, PACKED FORMAT
* **TEST6302** – PACKER PROM, (SIGN) IAC FOR NEGATIVE EMBEDDED SIGN
* **TEST6303** – PACKER PROM, (SIGN) IAC FOR NEGATIVE UNEMBEDDED SIGN
* **TEST6304** – PACKER PROM, (SIGN) IAC FOR POSITIVE EMBEDDED SIGN
* **TEST6305** – PACKER PROM, (SIGN) IAC FOR POSITIVE UNEMBEDDED SIGN
* **TEST6306** – XADU TEST, DECIMAL ALU CORRECTORS AND UNPACK LOGIC
* **TEST6307** – XADP, DECIMAL ALU CORRECTORS AND PACK LOGIC
* **TEST6308** – TEST OF DECNE JUMP CONDITION
* **TEST6309** – TEST OF BTD CONVERSION
* **TEST630A** – TEST OF DTB CONVERSION
* **TEST630B** – TEST_TNUM
* **TEST6400** – TEST FOR DMX RF 0
* **TEST6401** – TEST FOR DMX RF 1
* **TEST6402** – TEST FOR DMX RF 2
* **TEST6404** – TEST FOR DMX RF 4
* **TEST6408** – TEST FOR DMX RF 8
* **TEST640F** – TEST FOR DMX RF 15
* **TEST6410** – TEST FOR DMX RF 16
* **TEST6417** – TEST FOR DMX RF 23
* **TEST641B** – TEST FOR DMX RF 27
* **TEST641D** – TEST FOR DMX RF 29
* **TEST641E** – TEST FOR DMX RF 30
* **TEST641F** – TEST FOR DMX RF 31
* **TEST6420** – TEST_TNUM
* **TEST6500** – JUMP ON GPICOV FALSE (USING INHIBIT EXTERNAL MODAL BIT)
* **TEST6501** – JUMP ON GPICOV FALSE (USING STOPPIC IAC)
* **TEST6502** – JUMP ON PIC FALSE (USING RESETPOV)
* **TEST6503** – JUMP ON PIC TRUE (PIC HAS TO OVERFLOW HERE)
* **TEST6504** – JUMP ON GPICOV TRUE
* **TEST6505** – JUMP ON GTIMEROV FALSE USING DSCPTMR
* **TEST6506** – JUMP ON TMR FALSE USING DSCPTMR
* **TEST6507** – JUMP ON TMR TRUE (TMR HAS TO OVERFLOW HERE)
* **TEST6508** – JUMP ON GTIMEROV TRUE
* **TEST6509** – TEST_TNUM
* **TEST6600** – EXPPAT -> BDH(01:16) -> BPD(01:16) -> BDH(01:16), PARITY OFF
* **TEST6601** – EXPPAT -> BDH(01:16) -> BPD(01:16) -> BDL(01:16), PARITY OFF
* **TEST6602** – EXPPAT -> BDL(01:16) -> BPD(01:16) -> BDL(01:16), PARITY OFF
* **TEST6603** – EXPPAT -> BDL(01:16) -> BPD(01:16) -> BDH(01:16), PARITY OFF
* **TEST6604** – USERF JUMP CONDITION TRUE
* **TEST6605** – USERF JUMP CONDITION FALSE

- TEST6606 – DEVICE 20 OR 4 JUMP CONDITION TEST (TRUE AND FALSE)
- TEST6607 – CMI Burst Mode 646s – 1st word OUT and IN
- TEST6608 – CMI Burst Mode 646s – 2nd word OUT
- TEST6609 – CMI Burst Mode 646s – 3rd word OUT
- TEST660A – CMI Burst Mode 646s – 4th word OUT
- TEST660B – TEST_TNUM
- TEST7000 – CLEAR OUT ALL PE'S, JUMP ON FMCHK
- TEST7001 – TEST_TNUM
- TEST7010 – JUMP ON RCCER FALSE (NO PARITY ERROR)
- TEST7011 – JUMP ON RCCER TRUE (RCCPER1)
- TEST7012 – JUMP ON RCCER TRUE (RCCPER2)
- TEST7013 – JUMP ON RCCER TRUE (RCCPER3)
- TEST7014 – JUMP ON RCCER TRUE (RCCPER4)
- TEST7015 – JUMP ON RCCER TRUE (RCCPER5)
- TEST7016 – JUMP ON RCCER TRUE (RCCPER6)
- TEST7017 – JUMP ON RCCER TRUE (RCCPER7)
- TEST7018 – JUMP ON RCCER TRUE (RCCPER8)
- TEST7019 – TEST_TNUM
- TEST7020 – TEST OF BAD PARITY, BAH, WALKING ONES
- TEST7021 – TEST OF BAD PARITY, BAH, WALKING ZEROS
- TEST7022 – TEST OF BAD PARITY, BAL, WALKING ONES
- TEST7023 – TEST OF BAD PARITY, BAL, WALKING ZEROS
- TEST7024 – TEST OF BAD PARITY, BAE, WALKING ONES
- TEST7025 – TEST OF BAD PARITY, BAE, WALKING ZEROS
- TEST7026 – TEST_TNUM
- TEST7030 – IAC FORCEBADP RCD32A DATA
- TEST7031 – IAC FORCEBADP RCD32B DATA
- TEST7032 – IAC FORCEBADP RCD32A DATA – JUMP (NCMISS)
- TEST7038 – IAC ISBADP RCD32A INDEX, WALKING ONES THROUGH INDEX
- TEST7039 – IAC ISBADP RCD32B INDEX, WALKING ONES THROUGH INDEX
- TEST703A – HARD CACHE PE, GOOD DATA,GOOD INDEX – NO PE
- TEST703B – HARD CACHE PE, RCD32A BYTE 1
- TEST703C – HARD CACHE PE, RCD32A BYTE 2
- TEST703D – HARD CACHE PE, RCD32A BYTE 3
- TEST703E – HARD CAHCE PE, RCD32A BYTE 4
- TEST703F – HARD CACHE PE, RCD32A INDEX
- TEST7040 – HARD CACHE PE, RCD32B BYTE 1
- TEST7041 – HARD CACHE PE, RCD32B BYTE 2
- TEST7042 – HARD CACHE PE, RCD32B BYTE 3
- TEST7043 – HARD CACHE PE, RCD32B BYTE 4
- TEST7044 – HARD CACHE PE, RCD32B INDEX
- TEST7045 – TEST_TNUM
- TEST7050 – NFATALCPE JUMP CONDITION TRUE
- TEST7051 – NFATALCPE JUMP CONDITION FALSE
- TEST7052 – TEST_TNUM
- TEST7060 – CHECK OF STLB PARITY ERROR REPORTING
- TEST7061 – HARD STLB PARITY ERRORS, STLBA AND STLBB
- TEST7062 – TEST_TNUM
- TEST7070 => TEST707X – BD BYTEWISE PARITY TRAP TEST
- TEST7071 => TEST707X – BD BYTEWISE PARITY TRAP TEST
- TEST7072 => TEST707X – BD BYTEWISE PARITY TRAP TEST
- TEST7073 => TEST707X – BD BYTEWISE PARITY TRAP TEST
- TEST7074 => TEST_TNUM
- TEST7078 => TEST707Z – MA 6-BIT-WISE PARITY TRAP TEST
- TEST7079 => TEST707Z – MA 6-BIT-WISE PARITY TRAP TEST
- TEST707A => TEST707Z – MA 6-BIT-WISE PARITY TRAP TEST
- TEST707B => TEST707Z – MA 6-BIT-WISE PARITY TRAP TEST
- TEST707C – ECCU Trap test
- TEST707D – MISSING MEMORY MODULE TRAP TEST
- TEST707E – ABORT WRITE ON MA PARITY TEST
- TEST707F – TEST_TNUM

* TEST7100 – CHECK OF FMTRP LOGIC
* TEST7101 – CHECK OF FSPE LOGIC
* TEST7102 – CHECK OF MEMTRAP ON ALIGNED ADDRESS
* TEST7103 – CHECK OF MEMTRAP, CAUSED BY PARITY ERROR ON ALIGNED ADDRESS
* TEST7104 – CHECK OF ACCESS VIOLATION ON ALIGNED ADDRESS
* TEST7105 – CHECK OF PAGE MODIFIED TRAP ON ALIGNED ADDRESS
* TEST7106 – CHECK OF READ ADDRESS TRAP ON ALIGNED ADDRESS
* TEST7107 – CHECK OF MEMTRAP ON UNALIGNED ADDRESS
* TEST7108 – CHECK OF MEMTRAP, CAUSED BY PARITY ERROR ON UNALIGNED ADDRESS
* TEST7109 – CHECK OF ACCESS VIOLATION ON UNALIGNED ADDRESS
* TEST710A – CHECK OF PAGE MODIFIED TRAP ON UNALIGNED ADDRESS
* TEST710B – CHECK OF READ ADDRESS TRAP ON UNALIGNED ADDRESS
* TEST710C – CHECK OF READ ADDRESS TRAP ON SEGMENT WRAP
* TEST7110 – POINTER FAULT, 64V, INDIRECT, I(PB + D)
* TEST7111 – POINTER FAULT, 32I, INDIRECT, I(BR + D)
* TEST7112 – TEST OF IRPL HOLD DUE TO MEMORY TRAP
* TEST7113 – TEST_TNUM
* TEST7200 – RING ZERO ADDRESS, NO RIGHTS, NO ACCESS VIOL.
* TEST7201 – RING 1, NO RIGHTS; CACHE READ; GET ACCESS VIOL.
* TEST7202 – RING 1, NO RIGHTS; MEMORY WRITE; GET ACCESS VIOL.
* TEST7203 – RING 1, NO RIGHTS; RTN TO FETCH; GET ACCESS VIOL.
* TEST7204 – RING 1, GATE ACCESS; ISSUE IACGATE; NO ACCESS VIOL.
* TEST7205 – RING 1, GATE ACCESS; DO NOT ISSUE IACGATE; GET ACCESS VIOL.
* TEST7206 – RING 1, READ ACCESS ONLY; CACHE READ; NO ACCESS VIOL.
* TEST7207 – RING 1, READ ACCESS ONLY; MEMORY WRITE; GET ACCESS VIOL.
* TEST7208 – RING 1, READ ACCESS ONLY; RTN TO FETCH; GET ACCESS VIOL.
* TEST7209 – RING 1, READ AND WRITE ACCESS; CACHE READ; NO ACCESS VIOL.
* TEST720A – RING 1, READ AND WRITE ACCESS; MEMORY WRITE; NO ACCESS VIOL.
* TEST720B – RING 1, READ AND WRITE ACCESS; RTN TO FETCH; GET ACCESS VIOL.
* TEST720C – RING 1, EXECUTE ACCESS ONLY; CACHE READ; GET ACCESS VIOL.
* TEST720D – RING 1, EXECUTE ACCESS ONLY; MEMORY WRITE; GET ACCESS VIOL.
* TEST720E – RING 1, EXECUTE ACCESS ONLY; RTN TO FETCH; GET ACCESS VIOL.
* TEST720F – RING 1,EXECUTE AND WRITE ACCESS; CACHE READ; GET ACCESS VIOL.
* TEST7210 – RING 1, EXECUTE AND WRITE ACCESS; MEMORY WRITE; GET ACCESS VIOL.
* TEST7211 – RING 1, EXECUTE AND WRITE ACCESS; RTN TO FETCH; GET ACCESS VIOL.
* TEST7212 – RING 1, EXECUTE AND READ ACCESS; CACHE READ; NO ACCESS VIOL.
* TEST7213 – RING 1, EXECUTE AND READ ACCESS; MEMORY WRITE; GET ACCESS VIOL.
* TEST7214 – RING 1, EXECUTE AND READ ACCESS; RTN TO FETCH; NO ACCESS VIOL.
* TEST7215 – RING 1, ALL ACCESS; READ CACHE; NO ACCESS VIOL.
* TEST7216 – RING 1, ALL ACCESS; MEMORY WRITE; NO ACCESS VIOL.
* TEST7217 – RING 1, ALL ACCESS; RTN TO FETCH; NO ACCESS VIOL.
* TEST7218 – RING 3, NO RIGHTS; CACHE READ; GET ACCESS VIOL.
* TEST7219 – RING 3, NO RIGHTS; MEMORY WRITE; GET ACCESS VIOL.
* TEST721A – RING 3, NO RIGHTS; RTN TO FETCH; GET ACCESS VIOL.
* TEST721B – RING 3, GATE ACCESS; ISSUE IACGATE; NO ACCESS VIOL.
* TEST721C – RING 3, GATE ACCESS ONLY; DO NOT ISSUE IACGATE; GET ACCESS VIOL.
* TEST721D – RING 3, READ ACCESS ONLY; CACHE READ; NO ACCESS VIOL.
* TEST721E – RING 3, READ ACCESS ONLY; MEMORY WRITE; GET ACCESS VIOL.
* TEST721F – RING 3, READ ACCESS ONLY; RTN TO FETCH; GET ACCESS VIOL.
* TEST7220 – RING 3, READ AND WRITE ACCESS; CACHE READ; NO ACCESS VIOL.
* TEST7221 – RING 3, READ AND WRITE ACCESS; MEMORY WRITE; NO ACCESS VIOL.
* TEST7222 – RING 3, READ AND WRITE ACCESS; RTN TO FETCH; GET ACCESS VIOL.
* TEST7223 – RING 3, EXECUTE ACCESS ONLY; CACHE READ; GET ACCESS VIOL.
* TEST7224 – RING 3, EXECUTE ACCESS ONLY; MEMORY WRITE; GET ACCESS VIOL.
* TEST7225 – RING 3, EXECUTE ACCESS ONLY; RTN TO FETCH; GET ACCESS VIOL.
* TEST7226 – RING 3, EXECUTE AND WRITE ACCESS; CACHE READ; GET ACCESS VIOL.
* TEST7227 – RING 3, EXECUTE AND WRITE ACCESS; MEMORY WRITE; GET ACCESS VIOL.
* TEST7228 – RING 3, EXECUTE AND WRITE ACCESS; RTN TO FETCH; GET ACCESS VIOL.
* TEST7229 – RING 3, EXECUTE AND READ ACCESS; CACHE READ; NO ACCESS VIOL.
* TEST722A – RING 3, EXECUTE AND READ ACCESS; MEMORY WRITE; GET ACCESS VIOL.
* TEST722B – RING 3, EXECUTE AND READ ACCESS; RTN TO FETCH; NO ACCESS VIOL.

```
* TEST722C - RING 3, ALL ACCESS; READ CACHE; NO ACCESS VIOL.
* TEST722D - RING 3, ALL ACCESS; MEMORY WRITE; NO ACCESS VIOL.
* TEST722E - RING 3, ALL ACCESS; RTN TO FETCH; NO ACCESS VIOL.
*
*
```

17.2 SYSCLR

After the functional microcode has been loaded, the DP releases SYSCLR-. This should cause microcode to start executing from address 200 octal as discussed in chapter 19. Sysclr microcode disables traps to prevent bogus traps from occurring while it accomplishes the following system initialization tasks:

1. Clear the microsequencer stack and RPA.
2. Clear 400 octal register file locations and the modals.
3. Initialize RP and IRP to 1000 octal.
4. Loads all the constant registers.
5. Initialize the PCU.
6. Invalidate the cache, branch cache, and STLB.
7. Reset all I/O signals, initialize the Write Buffer, memory controller diagnostic register, and register file copy of same.
8. Size memory.
9. Reset all addressable latches.
10. Clear out any parity errors.
11. Update keys with live keys.
12. Enable traps and send the DP a 47 hex to inform it that Sysclr is complete.

This same microcode is executed whenever the SYSOFF command is issued from the system console.

18. Pipeline Control Unit Detailed Description

18.1 Instruction Flow Overview

The Pipeline Control Unit (PCU) of the 4150 is almost the same as that of the 9755 system. The central processor is implemented as a ten stage synchronous pipeline, capable of handling up to 5 machine level instructions simultaneously. The time for each stage to complete its particular operation is termed a "beat". A beat is two ticks of the system clock. Optimally, machine-level instructions can enter into and depart from the pipeline every two beats. Since instructions enter the pipeline only once every other beat the industry refers to this machine as a 5 stage pipeline with a basic cycle of two beats. A beat in this type of machine is often referred to as a minor cycle.

Figure 18-1 diagrams the concept of instructions flowing through the pipeline. Across the top are numbered beats which mark time progressing toward the right; vertically are shown activities of the various stages. Within the diagram, capital letters denote individual PM. level instructions, while numerals indicate additional microcode instructions needed to complete a PMA level instruction. Figure 18-1 and the associated explanation are highly simplified, and are intended simply to give a flavor of how instructions proceed through the pipeline.

FIG. 18-1. Conceptual Flow of Pipelined Instructions

```
                                   TIME  ------>

                                   1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2
STAGE            1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
                ---------------------------------------------------
 1  P>MA        |E| |F| |G| |H| | | | | |I| |J| |K| | |L| | |M| |
 2  INST        | |E| |F| |G| |H| | | | | |I| |J| | |K| | |L| |M|
 3  DEC         |D| |E| |F| |G| | | | | |H| |I| |J| | |K| | |L| |M
 4  BR,X        | |D| |E| |F| |G| | | | | |H| |I| | |J| | |K| |L|
 5  BCY         |C| |D| |E| |F| |1| |2| |G| |H| |I| | |J| | |K| |M
 6  EA,RCM      | |C| |D| |E| |F| |1| |2| |G| |H| | |I| | |J| |K|
 7  OP,RF       |B| |C| |D| |E| |F| |1| |2| |G| |H|u| |I| | |J| |K
 8  EX1         | |B| |C| |D| |E| |F| |1| |2| |G| | |H| | |I| |J|
 9  EX2         |A| |B| |C| |D| |E| |F| |1| |2| |G| | |H| | |I| |J
10  WR          | |A| |B| |C| |D| |E| |F| |1| |2| | |G| | |H| |I|
```

At time 1, the pipeline has five instructions (E, D, C, B, and A) being simultaneously processed in the five "odd" stages; instruction "A" is in the final stage of execution at the end of the pipeline, while instruction "E" is just entering the pipeline. During time period 2, the same five instructions are being processed, this time in the "even" stages. Similarly during time periods 3 through 8, the pipeline is processing at optimum efficiency. At each point in time, 5 machine level instructions are being processed. Every two beats an instruction is retired and a new one enters the pipeline.

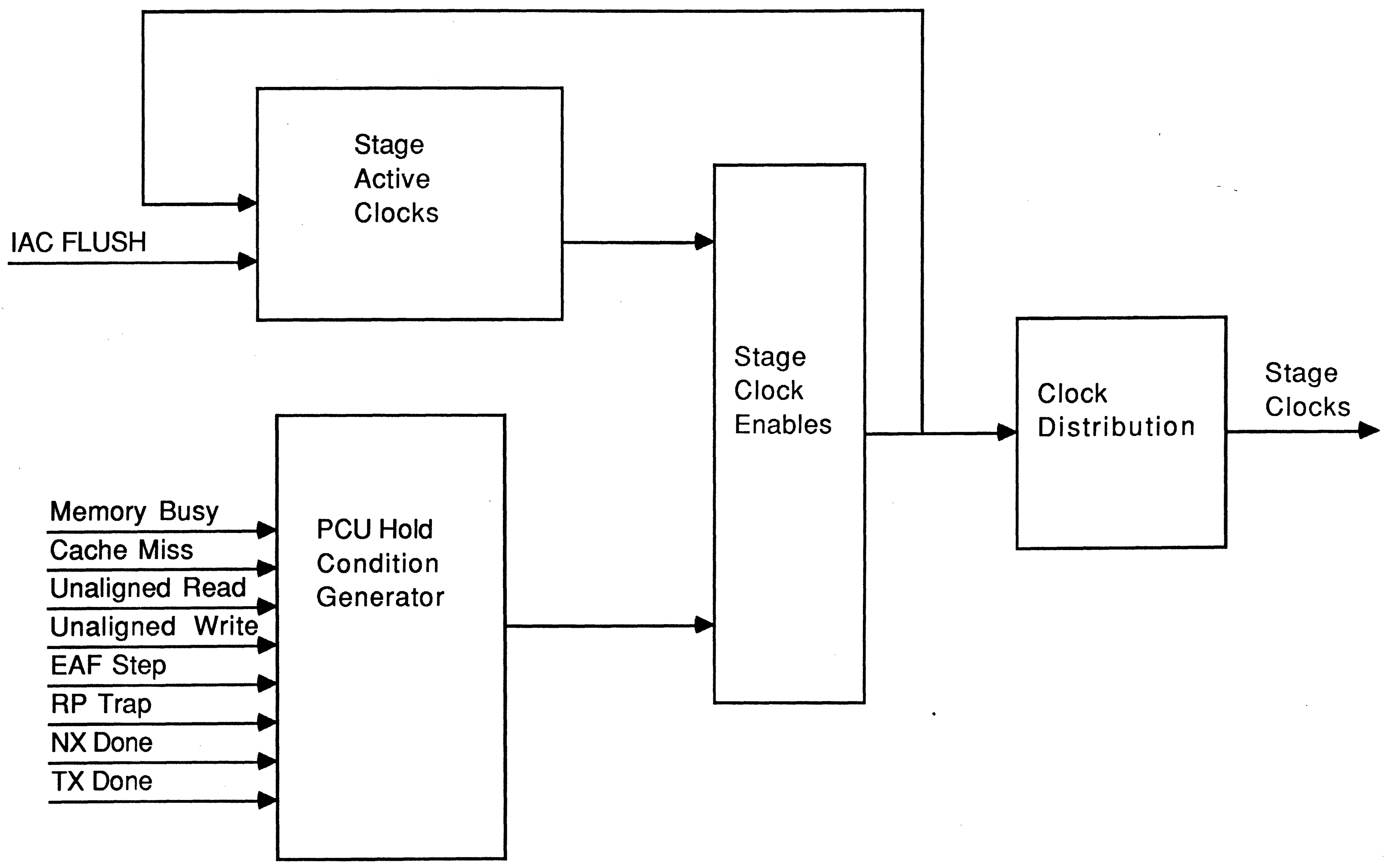


Figure 18-2 Block Diagram of Pipeline Control Unit

There are many instances in which a portion of the pipeline can become blocked. Examples of this are:

- Multiple microcode steps required for a PMA instruction
- Cache misses
- Unaligned operand read cycles
- Traps
- Incorrect pre-fetch sequences (which require refilling the pipeline)
- Register collisions
- Time extensions to perform certain arithmetic operations in the execution unit.

To illustrate one such example, assume PMA instruction "F" requires two extra microcode step to complete its function. In Figure 18-1, these two extra microcode steps are indicated by the 1's and 2's that appear in the pipeline from time periods T9 through T16. Since PMA instruction F is not a terminal microcode step (one which retires an instruction), the front of the pipeline must be stopped. No more pre-fetching can occur until the Execution (E) unit can complete the two extra microcode steps. At time period T13, the pipeline control unit "realizes" that microcode step 2 is terminal, and therefore instructions can continue to advance in the front end of the pipeline.

Another example of delaying the front end of the pipeline is evident in time period T18. Assume PMA instruction "H" is a memory reference instruction requiring 32 bits of data and further assume that the effective address is odd. The odd effective address requires cache to be cycled twice in order to access both words of data. This operation is performed automatically in hardware. During time period T17, stage 7 loads the autoincremented address into the cache address registers while reading the first portion of the data. During T18, stage 8 reads the second portion of the desired 32 bits of data. It is readily seen that an unaligned data read results in a beat penalty. Once the unaligned operation is complete the pipeline can again process with full parallelism (T19).

One final example of a pipeline delay is shown during time period T21. PMA instruction "I" requires extra time to complete an arithmetic operation. The execution phase is given an extra beat, which holds up the entire pipeline for 1 beat.

Delays and conflicts increase the ideal instruction processing time of 125 nsec.

18.2 PCU Operation

The PCU mechanism is a state machine that maintains the status information about all instructions in the pipeline. It consists of three basic elements:

- registers containing bits denoting which stages are currently active or processing data
- combinatorial logic which collects conditions from all boards which may delay stage clocking
- drivers which distribute the clock stage enables to all boards

If a stage is active and there are no external conditions received by the PCU indicating that the stage should be delayed, the enable line will be driven active to all boards handling that particular stage. The external conditions that the PCU receives include:

- Instruction information and exception and register collisions from the Instruction (I) unit.
- Memory exception, cache miss, and branch cache conditions from the Storage management (S) unit.
- Microcode specified timing information that controls the length of the execution and microcode control word formation stages.
- Fetch cycle traps and exception conditions from the E unit, such as integer exception, parity error, etc.

Some of the capabilities of the PCU include:

- Holding the front end of the pipeline (stages 1-6) while cycling multi-microcode through the E unit (fetching microcode and executing stages 7-10).
- Altering the flow of instructions based on control information provided by the microcode, An example of this is flushing the pipeline following execution of a conditional branch instruction when it is determined that the branch cache has predicted incorrect program flow.
- Extending all even stages if extra time is required for a particular stage to complete its operation.
- Alter the relative overlap of stages 6 and 8 to facilitate different types of microcode sequencing, including live conditional branches.
- Force NOPs (NOP_STEP) into the E unit while instruction refill occurs in the front of the pipeline. This makes the E unit do nothing while the next instruction is being fetched.

- Suspend all pipeline operations during non-overlappable operations such as cache miss access to main memory.
- Introduce separation between sequential instructions in the front end of the pipeline in order to handle "register-usage collisions" between instructions.
- Hold an instruction in the fetch stage if a memory trap occurs. This allows all instructions ahead in the pipeline to complete execution so that the exception can be processed in the correct order.

18.3 PCU Control

The pipeline can conceptually be thought of as consisting of a front end and a back end. The front end of the pipeline consists of stages 1 through 6, and is responsible for fetching and decoding the instruction stream. The signals ENCS01+...ENCS05+ and ENINIT+ are generated and distributed to the rest of the processor. The stage clocks CS1+....CS6+ indicate the successful completion of stage 1 ... stage 6. Note that the signal ENINIT+ is used to generate CS6+.

At this point, it is important to make the distinction between the signals CS05+ and CS05BCY+, and the signals CS06+ and TRCML+. The signal CS05+ and CS06+ are stage clocks in the front end of the pipeline indicating the successful completion of stage 5 and stage 6 respectively. These clocks, like all the front end pipeline clocks, occur only ONCE per instruction.

The signals CS05BCY+ and TRCML+ are used in the back end of the pipeline. The signal CS05BCY+ is used by the CS unit to clock the BCY address for the control store. This clock occurs at the end of almost every ODD beat. The exception to this would be during CRTN ODD-ODD sequences discussed below. The signal TRCML+ follows every CS05BCY+ on the EVEN beat. The signal TRCML+ occurs at every CS6+, which is a reason why many people confuse the two signals. Like all the clock signals in the back end of the pipeline, CS05BCY+ and TRCML+ may occur MANY times during an instruction if the instruction requires more than one microcode step to complete execution. These two microcode fetch stages are symbolized by F and T, respectively, throughout this document.

The rest of the pipeline consists of stages 7 through 10, and is responsible for executing program instructions. Unlike the front end of the pipe where each stage is executed only once per instruction, the stages in the back end of the pipeline are executed several times during multi-microcode instructions. The signals ENCS07+ .. ENCS10+, along with ENCS05BCY+ and ENTRCML+, are generated and distributed throughout the processor. The back end of the pipe is microcode driven, while the front end of the pipe is controlled entirely by the hardware.

At the beginning of each stage of the pipeline, that stage's active signal is asserted. If there are no external conditions inhibiting the successful completion of the stage, the stage clock is asserted at the end of the beat and the stage active signal is reset. The stage active signals are clocked by registers and remain active until the successful completion of that particular stage. If there does exist a condition that inhibits the successful completion of the stage, the stage active signal remains active until the hold condition goes away. For example, at the beginning of stage 1, the signal ST1A+ is asserted. If there are no external conditions inhibiting the successful completion of stage 1, the signal ENCS01+ is asserted and distributed to the rest of the processor. At the end of the beat this signal resets the ST1A+ signal, signaling the successful completion of the stage. The signal ENCS01+ also generates a CS1+ stage clock, and is used to generate all other clocks that are to be active at CS1+. CS1+ indicates the end of stage 1. In general, CSn+ indicates the end of stage n.

Figure 18-3 shows a simplified S-R flip-flop view of the conditions for setting and resetting the stage clock active signals. Figure 18-4 shows a simplified AND gate view for issuing stage clock enable signals. Refer to these figures during the following sections.

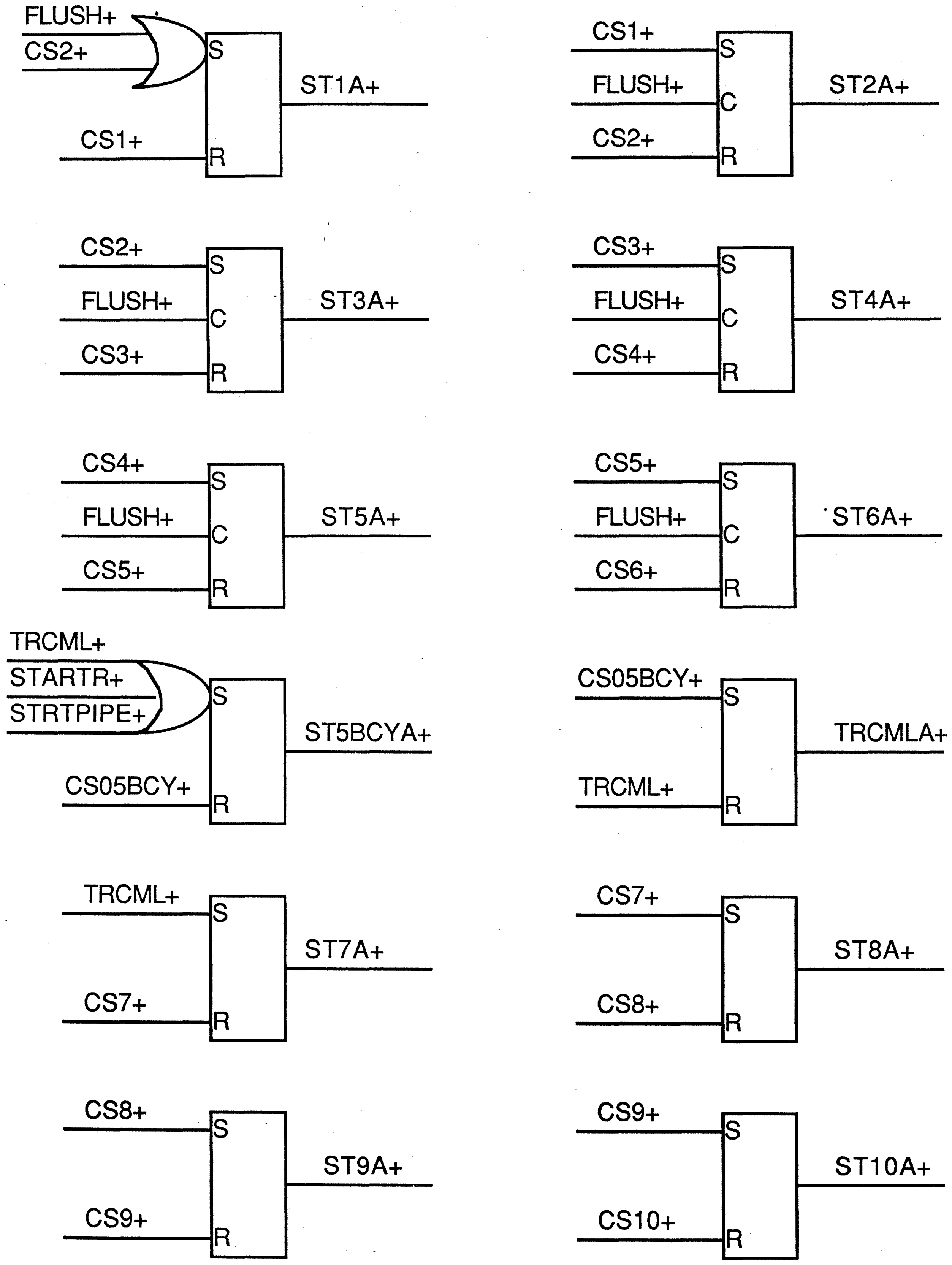


Figure 18-3 S-R Flip-Flop View of Stage Active Functionality

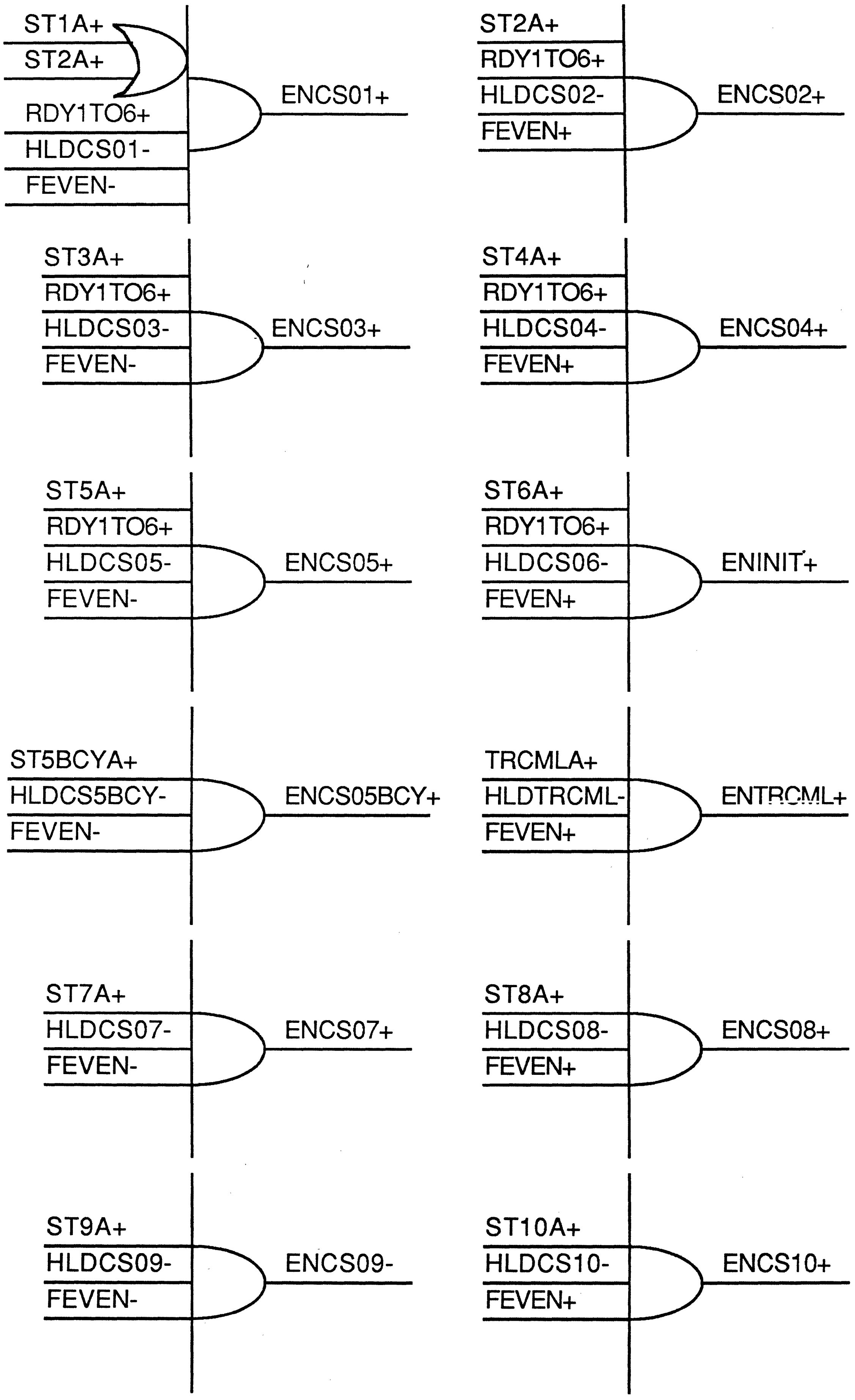


Figure 18-4 AND Gate View of Stage Clock Enables

18.3.1 Front End Control

The PCU is initialized on a master clear by forcing the stage active (STAx) signals to the inactive state. The front end (stages 1 to 6) of the pipe is started when the back end (stages F to 10) issues IAC FLUSH (via microcode). The signal FIACFLUSH+ is used to sequence PCU stages 1 to 6 when beginning program instruction execution or when flushing the pipeline is necessary because of a wrong guess during a conditional branch instruction.

The front end of the pipe is globally gated by the signal RDY1TO6+. This signal indicates that stages 1 through 6 should cycle if there is nothing else holding it off. This signal is asserted when the microcode does a return-to-fetch or a return from Effective Address Formation (EAF) microcode routine. It is also asserted when the pipe is refilling following a flush.

The signal FEVEN+ is fundamental to the generation of most of the ENCS signals. It acts to force all of the even stages to be segregated from all of the odd stages in the pipe. Whe FEVEN+ is active only the even ENCS signals (ENCS02...ENCS10) will be generated. In th normal case, FEVEN+ is merely the equivalent of ST5BCYA+. However, there are special cases in which we need to generate successive ODD beats. One example of this is when the microcode algorithm does a conditional return-to-fetch (CRTN). The condition code register (on the E unit) is clocked at CS09+. If the condition is true and the return taken, we need to generate a CS1+ one beat later. Thus in these cases the PCU will conditionally enable stages 1, 3 and 5 one beat after stage 7 and 9 of the CRTN, with the even stages happening one beat after that. This sequence is illustrated in Figure 18-5.

FIG. 18-5. Consecutive Odd Pipeline Stages Illustration

```
CRTN taken (ODD-ODD sequence)            CRTN not taken

        T-8-0   (CRTN step)                   T-8-0  (CRTN step)
       F-7-9                                 F-7-9
1-3-5 ———————                                ——————— <====dead beat
        T-8-0                                 T-8-0
```

18.3.2 Back End Control

The back end of the pipeline begins with the clocking of the microcode address (BCY) CS05BCY+. The signal FEVEN+ segregates the even stage clocks from the odd stage clocks. The TXNX hardware controls the clock enables in the back end of the pipeline. (See section 18.4.1.) The microcode uses it to specify timing information that controls the duration of the clock intervals in the back end of the pipeline.

The microcode can specify that only the back end of the pipeline be sequenced, as during multi-microcode execution, or sequence the front end of the pipeline when a return (RTN+ is active) is specified and the microcode stack is empty (NRCSFL+ is active).

18.4 External PCU Hold Conditions

The PCU can extend the length of and/or delay the occurrence of the various clock stages. These hold conditions are discussed in the following subsections. Figure 18-6 shows a block diagram of this functionality, while Table 18-1 defines the actions taken.

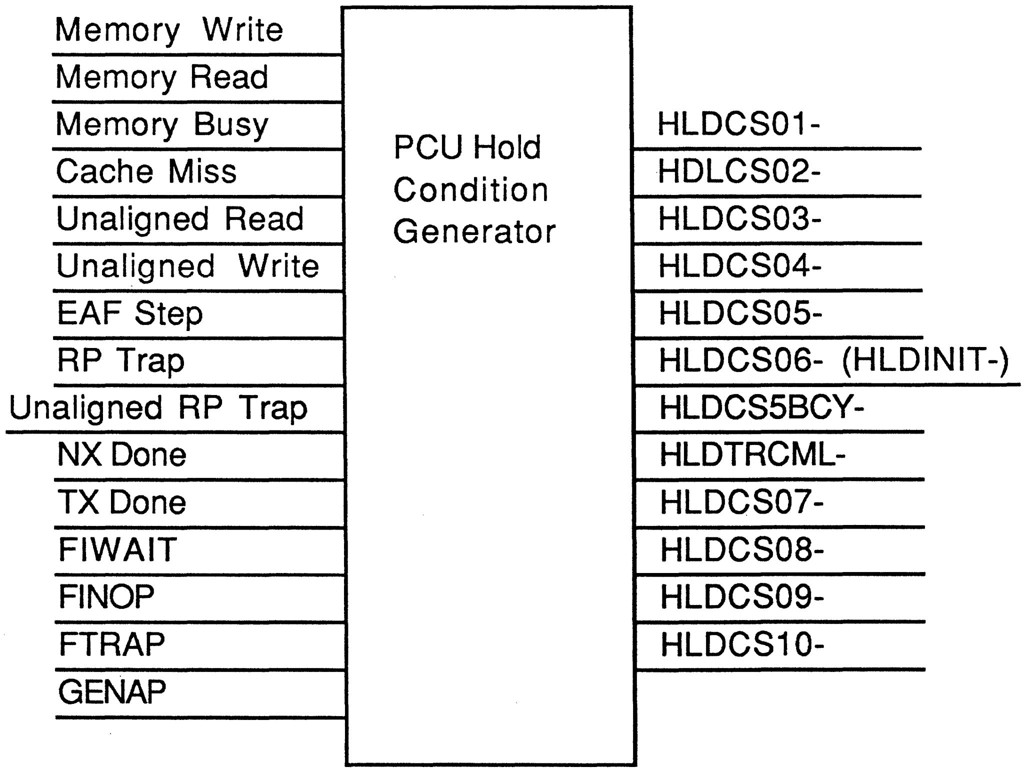


FIG. 18-6. PCU Hold Condition Block Diagram

TABLE 18-1. Pipeline Hold Conditions

Condition	CS1	CS2	CS3	CS4	CS5	CS5BCY	CS6	TRCML	CS7	CS8	CS9	CS10
Memory Busy on Write	X	X	X	X	X	X	X	X	X	X	X	X
Memory Busy on Read	X	X	X	X	X	X	X	X	X	X	X	X
Cache Miss	X	X	X	X	X	X	X	X	X	X	X	X
Unaligned Read	X	X	X	X	X	X	X	X	X	X	X	X
Unaligned Write	X	X	X	X	X	X	X	X	X	X	X	X
EAF Step	X		X									
RP Trap			X									
Unaligned RP Trap					X							
NX Done		X		X			X	X				
TX Done		X		X			X	X		X		
Register Collision FIWAIT		X		X			X	X		X		
FTRAP		X		X			X	X		X		
Register Collision FINOP	X		X		X							
GENAP				X								

18.4.1 Variable Length Microcode Step Timing

The microcode specifies timing information that controls the length of the execution and microcode control and formation stages. These microcode extensions are specified in the TME field by Time eXtension (TX) and Next microstep eXtension (NX) conditions.

The TX portion of the time extension causes CS08+ for the current step and TRCML+ for the next step to be delayed by the number of TMCLKs specified (one TMCLK is half of a beat). The NX portion of the extension delays the occurrence of TRCML+ for the next step from CS08+ of the current step by the number of TMCLKs specified. If both fields are zero, CS08+ for a microcode step occurs 4 TMCLKs (two beats) after TRCML+ for that step and the TRCML+ of the next step is coincident with CS08+ for the current step. Figure 18-7 illustrates these actions.

The TX specifier can be 0-6 TMCLKs, while the NX specifier can be 0-8 TMCLKs. The TX specifier can be an odd number (half beat granularity), but the NX specifier can only be an even number (full beat granularity).

FIG. 18-7. TX NX Illustrations

```
A. No microcode extension : TX=0,NX=0

      T-8-0     LDA
     F-7-9
      T-8-0     STA

B. TX microcode extension : TX=2,NX=0

      T-8-0    LDA
     F-7-9
     ------0
      T-8      STA

C. NX microcode extension: TX=0,NX=2

      T-8-0    LDA
     F-7-9-
      --8-0
      T----    STA
     F-7-9

D. TX,and NX microcode extension: TX=2,NX=2

      T-8-0    LDA
     F-7-9-
     ------0
     ---8--
      T----    STA
     F-7-9
```

The TXNX logic is implemented in the following manner: The TX specification can be an even or odd number. An even TX means that TRCML+ and CS08+ will be delayed on full beat boundaries. For example, a TX= 2 inserts a one beat (2 TMCLKs) delay, while a TX= 4 inserts a 2 beat delay. An odd TX means the delay will be on half beat boundaries. For example, a TX= 1 will insert a half beat delay (1 TMCLK), while a TX= 3 will insert a 1 and 1/2 beat delay (3 TMCLKs).

Even TXs and NXs are handled by down counters which are clocked at the end of every beat. The counter is loaded with the number of full beat delays specified by the TX or NX field at CS7+. These counters generate the signals TXDONE- and NXDONE- when the delay has finished. For example, a TX=2 would load the TX counter with 1 at CS7+. The counter would make TXDONE- inactive during stage 8, thereby holding off the TRCML+ of the next instruction and CS08+ for the current instruction for one beat. A beat after CS7+ the counter would count down, and the signal TXDONE- would be activated.

The odd portion of the TX specification is implemented by inserting a one TMCLK delay in the signal FENEOB+ (sent to all boards) during stage 8. This pushes the end of the beat after CS7+ out a half a beat. Therefore, for a TX=3, FENEOB+ would be delayed by a half beat creating a TX=1, and the TX counter would delay TRCML+ and CS08+ one beat for an additional TX of 2.

18.4.2 EHOLDs

When the S unit needs more time to complete an operation it uses the signal FEHOLD+ to insert extra beats in the pipeline. This signal inhibits all clock enables in the PCU. There are five instances in which holding the pipeline is required:

1. Writes to memory when the Memory Controller (MC) is busy: The signal FEHOLD+ is raised at CS7+ if a memory access is requested and the MC is asserting MBSY-. The PCU is held in the hold state until MBSY- is removed.

2. Cache misses: During cache misses the signal FEHOLD+ is asserted to suspend all pipeline clocks until the cache miss is completed.

3. Unaligned writes: If microcode is doing an unaligned write and no traps are pending and no RMA destination has been called out in the current step, FEHOLD+ is asserted for one beat starting at the rising edge of TRCML+. This extra beat is used to clock the cache address register with the address for the write of the second word. (If microcode was loading RMA in the write step, the delay associated with that load is used to get the incremented address and the extra beat is not necessary.)

4. Explicit memory reads: If a microstep is waiting on data from an explicit read (the signal IACMRDY- asserted) but the data is not available yet from the MC, FEHOLD+ is held active until the data becomes available (MDATAV+ is asserted).

5. Unaligned reads: If microcode is doing an unaligned read, FEHOLD+ is asserted at CS7+ and held (for two beats unless a cache miss occurs) until the end of the unaligned read operation. (See unaligned read discussion in chapter 24 for more details.)

18.4.3 Traps

The S Unit may detect a memory "trap" condition at CS2+ of an attempted instruction fetch. The possible conditions are:

- Address trap
- Access violation
- Any of several problems starting with an STLB miss.

The basic ground rules for handling these conditions (referred to as RP traps) are as follows:

1. An instruction must not be allowed to enter the E unit if the fetch of any portion of that instruction generated an RP trap. The data supplied by such a fetch is invalid and must not be used. (An exception is the Address Pointer (AP) part of a GENAP instruction. The E unit reads the AP as part of the execution phase, and can deal with any traps that arise at that point.)

2. All instructions which have already been successfully fetched must be executed to completion before attempting to deal with an RP trap. The flow of execution may change before the trapping instruction would have entered the E unit, and a trap must never be raised unless the trapping instruction is going to be executed. An example: Suppose a segment is executing in which some of the pages are defined. Suppose further that the last instruction in the last defined page is a non-branch-cached jump instruction back to some previous word in this page. In this case we will prefetch into the undefined page before executing the JMP that changes the flow of execution. If we went ahead and tried to deal with the STLB miss (and subsequent page fault), PRIMOS would halt due to an attempted page fault to an undefined page.

3. A branch cache gaffe trap must take precedence over an RP trap. Consider a long unaligned instruction with a branch cache hit on the first word that sends IRP off to an undefined page. (This is an error on the part of the branch cache). On the attempted fetch of the second word of the instruction, we get an RP trap condition. The branch cache gaffe mechanism will also be trying to deal with this situation. We must not allow any attempt to deal with the RP trap since we did not really intend to fetch this erroneous second word.

4. The RP trap mechanism must work for targets of XEC instructions.

5. The RP trap mechanism must be prepared for the trap condition to disappear before the E unit gets around to attempting to deal with the trap. An example is the case of prefetch into a new page causing an STLB miss, but one of the instructions already in the pipe happens to resolve the STLB miss as part of an operand fetch.

18.4.3.1 Implementation Details

It is necessary to provide some mechanism of getting to the correct trap handler and providing the faulting address. We chose to transfer the faulting address from IRP to RMA in the S unit. (RMA actually means ERMA, cache address, or STLB address in this context.) This has several advantages over alternative schemes:

1. The trap condition can be "re-created" when we are ready, and the normal operand trap mechanism exploited, including the microcode entry points.

2. The trapping address is accessible simply by reading "RMA". No special mechanism is required for providing the trapping address to the microcode.

3. We make one last check for existence of the trap condition, ensuring that we never take a trap that has already been serviced.

All RP traps are taken out of the NOP_STEP, since it is required that we finish all previous instructions and not initiate any new ones. This provides a clean mechanism for restarting the pipe after dealing with the trap without attempting to re-execute already completed instructions.

The cycling of the front end of the pipe is frozen on detection of an RP trap by inhibiting ENCS03+. The back end is allowed to continue to cycle in order to complete any instructions already fetched, except for the case of an unaligned long instruction with an RP trap on the second word. For these cases ENCS05+ is also frozen. Generation of ENCS01+ and ENCS02+ is not frozen, in order to continuously retry the fetch and generate the trap out of the NOP_STEP at the proper time. Note that CS1+ is needed to reload the cache address registers with IRP, and that CS2+ is needed to relatch the RPTRAP- signal. Figure 18-8 illustrates the pipeline flow when an RP trap is detected.

It is very important to note that while we do freeze the IRP address that caused the RP trap, it is possible to overwrite this frozen address via BD. For example, we can load IRP with a new address if we jump over the trapping address. In this case we would load IRP via BD with the new address and flush the front end of the pipeline. By continuing to generate CS1 and CS2 clocks the RP trap would be cleared when the jump address was loaded into IRP.

The S unit is obliged to inhibit any change to IRP at CS2.5 of any instruction which raises an RP trap. This enables dealing with the trap later since the trapping IRP is frozen. The S unit must also not allow any RP cache misses to occur on RP traps, since in the case of an RP STLB miss, the physical address may not be valid.

FIG. 18-8. Pipeline Flow For RP Trap

Instruction i+4 causes an RP trap. This trap is not taken until all instructions ahead of it have finished execution.

```
1                             instruction i      IRP=1000 (Octal)
 2
1 3                                   i+1           =1002
 2 4 6 T
1 3 5 F 7                             i+2           =1004
 2 4 6 T 8
1 3 5 F 7 9                           i+3           =1006
 2 4 6 T 8 0
1 3 5 F 7 9                           i+4           =1010
 2 4 6 T 8 0     *1       i FINISHED
1   5 F T 9                           i+4           =1010
 2   6 T 8 0            i+1 FINISHED
1     F 7 9      *2                   i+4           =1010
 2     T 8 0            i+2 FINISHED
1     F 7 9                           i+4           =1010
       T 8 0            i+3 FINISHED

      F                 Enter Trap Code
       T
      F 7

  Notes:
     *1 Instruction i+4 causes RP trap at next CS2+
        Next CS3+ is inhibited until we have resolved
        the potential trap.
        Signal RPTRAP- is generated at CS2+.

     *2 The signal FEXNOP+ goes active, signifying that
        there are no longer any instructions left in
        the front end of the pipeline and that the
        next microcode entry point will be the NOP_STEP.
```

18.4.4 Miscellaneous

Given an instruction i requiring microcode assistance during the EAF stage, CS3+ of instruction i+1 and CS1+ of instruction i+2 are delayed until the E unit goes from the EAF microcode to the actual execution microcode.

During GENAPs (3 word instructions) ENCS04+ of instruction i+1 is inhibited for two beats.

During a 5-9 register file collision, CS6+ of instruction i+2 and CS8+ of instruction i+1 are inhibited for one beat.

18.5 9755 and 4150 Comparisons

The 4150 and the 9755 PCUs are functionally identical.

18.6 Clock Distribution

The master clock lines in the 4150 are among the most important signals in the processor. They have very high fan-outs because they drive many registers. A synchronous system's reliability is based on the premise that a noise-free clock is available.

The 4150 has two oscillators, one at 32 MHZ to generate the nominal clock rate (TMCLK-) and one at 34 MHZ to generate the high frequency clock rate (TMCLKHF-). These oscillators feed a 2-to-1 multiplexer. Normally the multiplexer select line points to the 32 MHZ oscillator. If the Diagnostic Processor issues an high frequency command (used during frequency margining), the mux is switched to select the 34 MHZ oscillator. (The 4050's nominal clock rate is 30 MHz. It uses the same 34 MHz high frequency clock rate as the 4150.)

The multiplexer generates four outputs, one for each board in the system. The signals generated are TMCLK-E, TMCLK-IS, TMCLK-CMI, and TMCLK-PDA. Each board uses an 74AS1804 to buffer its version of the TMCLK signal and to generate positive versions of the clock with which to drive all the registers on that board.

The clock line lengths are controlled such that each board receives the signal at approximately the same time. The CMI board, which generates the master clock signals to all the boards, receives its TMCLK version off the backplane as well. The signal TMCLK-CMI is generated on the CMI board and driven onto the backplane. The CMI then receives this signal on another pin of the backplane. This is done so that each board on the system receives its version of the system clock at approximately the same time.

18.7 Critical Paths

1 beat paths (62.5 nsec)

Generating RPTRAP- (at TRCDIE+) at the end of stage 2 and preventing all stage 3 clocks from occurring at the end of the beat. The following stage 3 clocks need to be held:

- TRCDIE+ -> FMTRPA+ -> HLD3- -> AENCS03- -> ENST3A+ -> keeping ST3A+ from being reset at the end of the beat. Time = 57.0 ns

- TRCDIE+ -> FMPTRPA+ -> HLD3- -> AENCS03- -> ENTRPST+ -> keeping TRPST+ from going active at the end of the beat. Time = 57.0 ns

Generating a Return to Fetch at TRCML+, which in turn starts the front end of the pipe moving at the end of the beat. TRCML+ -> RCMSxx -> RTNTOFCH- -> RDY1TO6+ -> AENCS01- -> ENST1A+ -> reset of ST1A+ at the end of the beat. Time = 57.0 ns

Generating a Conditional Return to Fetch (CRTN+) on the E unit at CS9+, and then generating CS1+, CS3+, and CS5+ stage clocks in 1 and 1/2 beats. TSETCC+ -> CCxx -> JC09 -> PCRTN+ -> GCRTN+ -> ENST1A+ -> reset of ST1A+. Time = 79.5 ns

Memory traps inhibiting a potential cache miss at the end of the beat. TRCDIE+ -> FMTRPA+ -> DISMISSPE-IS(22H) -> GHOLD- -> ENFCMISS -> holding FCMISS+ from going active at the end of the beat. Time = 52.5 ns

Memory traps inhibiting GHOLD+ signal from holding off stage clocks at the end of the beat if a cache miss is pending.

- IS stage clock timing: TRCDIE+ -> FMTRPA+ -> DIMISSPE-IS -> CMISSNDIS- -> GHLDORNEOB+ and stopping the inhibit of all stage clocks at the end of the beat. Time= 54.5 ns

- E stage timing: (CMI timing almost identical) TRCDIE+ -> FMTRPA+ -> DISMISSPE-IS -> CMISSNDIS -> GHLDORNEOB+ -> stopping the inhibit of all stage clocks at the end of the beat. Time = 60.5 ns

Cache miss detected and stopping all the stage clocks at the end of the beat. Note no memory trap detected.

- IS board timing: TRCDIE+ -> CMISS-A1 -> CMISS+ -> CMISSNDIS- -> GHLDORNEOB+ -> inhibiting next stage clocks from occurring at the end of the beat. Time = 56.5 ns

- E board timing: (CMI timing almost identical) TRCDIE+ -> CMISS-A1 -> CMISS+ -> CMISSNDIS- -> GHOLDORNEOB+ -> inhibiting all stage clocks from occurring at the end of the beat. Time = 62.5 ns

FEHOLD+ preventing TCADR+ from occurring at the end of the beat. FEHOLD+ -> ENTRMCL-IS -> ENTCADRA- -> ENTCADR+ -> stopping TCADR+ from going active at the end of the beat. Time = 57.0 ns

TX and NX steps pushing out even stage clocks which would have occurred at the end of the beat.

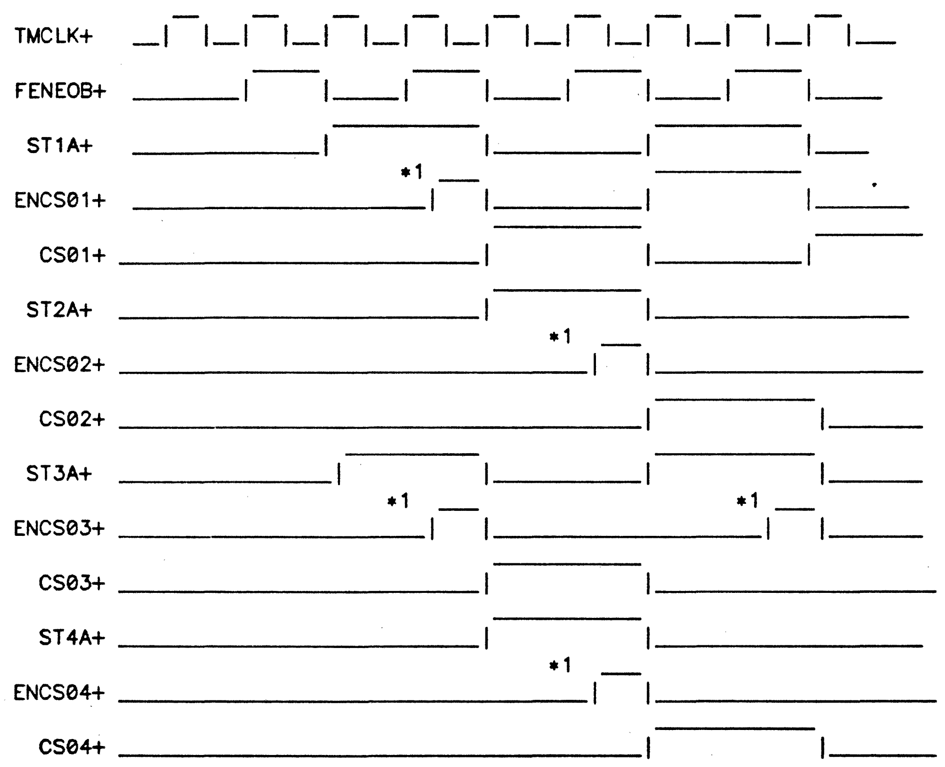
- Pushing out TCADR+: NXDONE- -> ENTRCML-IS -> ENTCADRA- -> ENTCADR+ -> stopping TCADR+ from occurring at the end of the beat. Time = 60.0 ns

- Pushing out FSELSRMA+: TXDONE- -> ENCS08-IS -> ENCS08+IS -> PENSELSRMA- -> ENSELSRMA+ -> pushing out FSELSRMA+. Time = 62.5 ns

Inhibit clocking FSELSRMA+ at CS7+ of memory write if an RP trap occurred during TRMCL+: TRCDIE+ -> FMTRPA+ -> GTRAP+ -> PENSELSRMA+ -> ENSELSRMA+ -> inhibiting FSELSRMA+ from going active at the end of the beat. Time = 61.5 ns

18.8 Timing Diagrams

FIG. 18-9. Stage Clocking Example



Notes:

*1 The signals ENCS01+, ENCS02+, ENCS03+, and ENCS04+ are not register outputs. When the signals become active in the beat are depended on the various hold conditions that gate them.

18.9 Partitioning

Clock distribution is implemented on every board as necessary and appropriate. The master system clock, TMCLK, is generated on the CMI board, which is also responsible for generating odd TXs. The microsequencer also resides on the CMI, which is discussed in detail in Chapter 30. Trap sequences are initiated by the E unit, which is discussed in Chapter 32. The remainder of the PCU functionality is implemented on the IS board, which is discussed in Chapter 31.

19. Control Store Unit Detailed Description

The Control Store (CS) unit is responsible for holding and retrieving microcode bits, generating the next microinstruction address from information in the current microinstruction, and distributing microcode bits to the other logic units as needed.

The microinstruction word is discussed in chapter 14.

The control store is loaded by the Diagnostic Processor (DP). The DP interface is discussed in chapter 16.

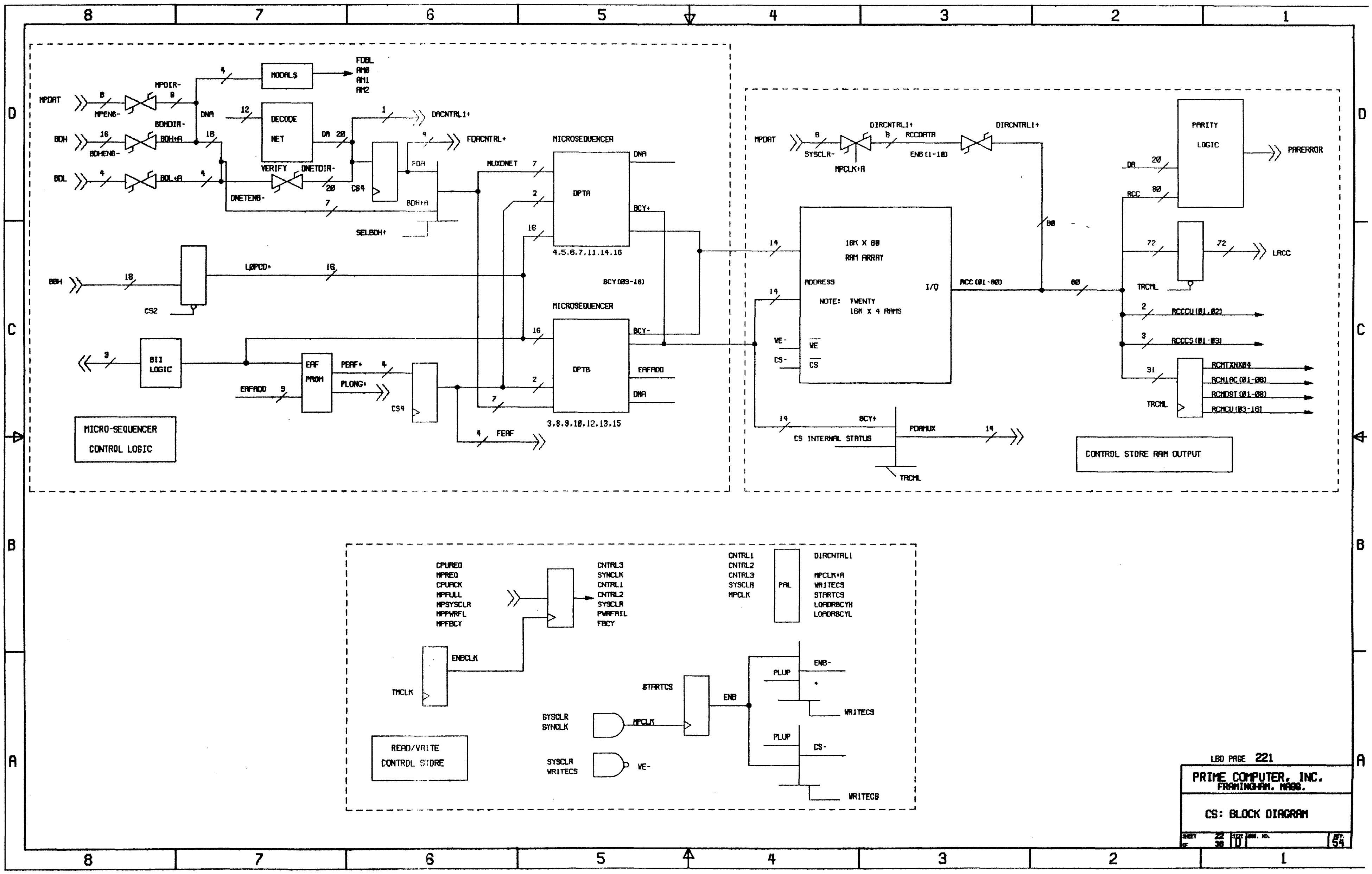


Figure 19-1 Block Diagram of Control Store

19.1 Control Store Memory

The CS memory is a 16K x 80-bit memory. The address to the CS memory is referred to as BCY. (On an ancient Prime machine, addresses were referred to as Y, leading to the name Bus Control Y. The name has, unfortunately, stuck.) The RAMs used are 16K x 4 RAMs with an access time of 25 nanoseconds.

The CS memory is loaded by the DP or PDA during system initialization. The following sequence of DP interface transactions is executed to load the CS memory:

1. Load RBCYH.
2. Load RBCYL.
3. Write CS. This writes the first 8 bits of the current CS memory address.
4. Repeat step 3 nine more times to write the rest of the current 80-bit word.
5. Repeat steps 1, 2, 3, and 4 until the entire CS memory has been written.

Reading the CS memory is very similar to writing it. The Write CS command in step 3 above would be replaced with a Read CS.

Note that the entire 80-bit microword must be written each time a microaddress is accessed. There is no provision for writing selected microword fields only.

The CS can be loaded by either the DP or the PDA. It can't tell the difference between the two.

19.2 Microinstruction Sequencing

The currently executing microinstruction contains information on how to generate the next microinstruction's address in the CS and CU fields. These fields, combined with information from other functional units, are input to the microsequencer, which produces the next microinstruction address. Some examples of information coming from other units which might affect microinstruction sequencing are jump conditions, traps, and effective address formation signals.

Table 14-2 shows the CS and CU field definitions. It is reprinted here as Table 19-1. The table shows the possible choices for next BCY generation under normal operation. Each will be discussed in turn, followed by a discussion of cases when abnormal things, such as traps, occur.

TABLE 19-1. CS and CU Field Definitions

CS Field (RCM Bits 49-51)	CU Bits 1-2 (RCM Bits 32-33)	CU Operation
000	——	RTN
001	——	JUMP
010	00	DECODE
010	01	BDH branch
010	10	LDA
010	11	GOTO
011	——	EMIT
100	——	CRTN
101	——	CALL w/ JUMP
110	——	CALL w/ GOTO
111	——	PUSH w/ EMIT

19.2.1 Normal Microinstruction Sequencing

A new BCY value is generated during each stage 5 BCY. (Recall the difference between stage 5 and stage 5 BCY as discussed in chapter 17.2.) The address is clocked into Register BCY (RBCY) at CS5BCY+. The CS memory is accessed during the next beat, and clocked or latched as necessary at Trigger Register Control Memory Late (TRCML+). (Why late ? Being early wouldn't work, and it's very difficult to be precisely punctual in these matters. In other words, the meaning of late in TRCML+ was lost with the meaning of using Y to indicate an address.)

Microinstruction sequencing always starts from the same CS memory address, 200 octal. This is set by the DP or PDA, and can be changed by either. The signal START- is a synchronized version of SYSCLR- sent by the DP. This signal causes the microsequencer to keep RBCY = 200 octal. When the first CS5BCY+ clock period starts after SYSCLR- is released, the first microinstruction's bits are fetched.

The first microinstruction is executed twice after SYSCLR- is released. This is to ensure traps get turned off in the CPU until the microcode gets a chance to clean the machine up. This happens only on the first release of SYSCLR- after power on.

19.2.1.1 RTN

The microsequencer includes a microinstruction stack for use in microcode subroutine calls and returns. When a RTN operation is done, the top microinstruction on the stack is fetched as the next microinstruction for execution, and the stack pointer is updated appropriately.

A RTN operation may be done when the stack is empty. This is called a Return to Fetch Level (RTNFL), and indicates that the next PMA instruction in the pipe should enter the E unit.

The signals listed below are associated with the RTN operation:

- NRCSFL-
  This signal tells the PCU that the stack is empty. If the microcode does a return, the PCU will start the front of the pipe.

- PUSH- and GIACPOP-
  These two signals define what type of stack operation is going to occur so that the TWRITE+ and TSTACK+ clock pulses can be generated.

- NOOP+
  This signal is generated by the I unit and causes the microsequencer to go to the NOP_STEP when it is doing a Return to Fetch level. This signal allows the I unit time to decode the next instruction while the E unit continues executing NOPs.

- FWRITE-
  This signal triggers the stack write pulse circuit inside the microsequencer which generates the write pulse for the microcode stack. The stack write pulse is generated when TSTACK+ or a trap occurs and is active for 1/2 beat after CS5+.

19.2.1.2 JUMP

The JUMP operation looks at the Jump Conditions (JCs) received from the other units. The microsequencer uses the JCs to generate Jump Address (JA) bits. These bits become the four least significant BCY bits, allowing the microsequencer to go to the address specified by the result of the jump operation. Figure 19-2 shows how the BCYs are formed during a jump.

FIG. 19-2. JUMP BCY Formation

```
JUMP

Microcode Syntax:   JUMP (n conds) TO (n**2 labels), n = 1,2,3,4.
                     can do up to a 16 way branch

  Interpretation of CU field:

   CU 1-2    CU 0,3,4 CU 5,6,7  CU 8,9,10 CU 11,12,13 CU 14,15,16
 ________________________________________________________________
| BCY11-12 | JA 13   | JA 14    | JA 15    |    JA 16    | CODE GRP |
 ________________________________________________________________

 Next address generated:

    BCY3-10       11-12       13    14    15    16
  ________________________________________________
 | RCH03-10 | CU 1,2     | JA | JA | JA | JA |
  ________________________________________________
  Note: RCH is the register which holds the current BCYs.
```

The jump net selects the specified condition(s) and sends them to the microsequencer, which uses them to generate BCY{13:16}+. The microcode assembler makes sure that unspecified jump conditions (the other three on a two-way jump, for example) are correct.

19.2.1.3 DECODE and LDA

The DECODE operation generates a microsequencer address which is determined by the result of the decode net and EAF unit output. Logical Decode net Address (LDA) does the same operation. Figure 19-3 shows the BCY formation for these operations.

FIG. 19-3. DECODE and LDA BCY Formation

```
LDA

Microcode Syntax:  CS = (2,LDA)

Next address generated:
                            BCY03-16
                   ________________________________
                  |    From decode net or EAF unit  |
                   ________________________________

DECODE

Microcode Syntax:   CS = (2,DECODE)

Next address generated:
                              BCY03-16
                   ________________________________
                  |   From decode net or EAF unit   |
                   ________________________________
```

19.2.1.4 BDH Branch

The BDH Branch operation generates a microsequencer address which is determined by the value placed on BDH{01:08}+, which is then clocked into the RBCY register. This register is clocked on 8-bit boundaries. Figure 19-4 shows the BCY formation for this operation.

FIG. 19-4. BDH Branch BCY Formation

```
BDH branch

Microcode Syntax: CS = (2,BDH)

Next address generated:
                              BCY03-16
                   __________________________________
                  |               RBCY                |
                   __________________________________
```

Some signals associated with BDH branch operations include:

- TRBCYH-
  This clock triggers the high side of the RBCY register inside the microsequencer during control store writes and BDH branches. The data is loaded 8 bits at a time over BDH. This clock goes active at TRCML+ and stays active for one beat.

- TRBCYL-
  This clock triggers the low side of the RBCY register inside the microsequencer during control store writes and BDH branches. The data is loaded 8 bits at a time over BDH. This clock goes active at TRCML+ and stays active for one beat.

19.2.1.5 GOTO

The GOTO operation generates a microsequencer address which is determined by the data contained in the CU field of the microcode word. Figure 19-5 shows the BCY formation for this operation.

FIG. 19-5. GOTO BCY Formation

```
GOTO

Microcode Syntax:  GOTO label

Next address generated:
                              BCY03-16
                  ________________________________
                 |            CU 3-16             |
                  ________________________________
```

19.2.1.6 EMIT

The EMIT operation generates a microsequencer address which is determined by the current microcode address with BCY bit 12 inverted. Figure 19-6 shows the BCY formation for this operation.

FIG. 19-6. EMIT BCY Formation

```
EMIT

Microcode Syntax:   =  EMIT constant given in BLEG or DEST field

Interpretation of CU field:

                          CU 1-16
         _________________________________________________
        |                 EMIT CONSTANT                   |
         _________________________________________________

Next address generated:

                BCY03-11        12        13-16
          ______________________________________
         |    RCH3-11    | ~RCH12 |  RCH13-16   |
          ______________________________________
```

19.2.1.7 CRTN

The CRTN operation generates a microsequencer address which is determined by the result of the CRTN condition, which comes from the jump net. If the condition is true, the address generated is the output of the Register Control Stack (RCS). If the condition is false, the address is generated in a similar fashion to the GOTO or JUMP cases. Figure 19-7 shows how the BCYs are formed for this operation.

CRTN operations may cause RTNFL sequences.

FIG. 19-7. CRTN BCY Formation

```
CRTN

 Microcode Syntax: CRTN (cond) ELSE GOTO label
                   CRTN (cond) ELSE JUMP (n conds) TO (n**2 labels)
                        where n = 1,2.

 Interpretation of CU field:

         CU 1-4      CU 5-7  CU 8-10 CU 11-13 CU 14-16
        ______________________________________________
       | BCY 11-14 | CRTN   | JA 15   | JA 16   | CODE GRP |
        ______________________________________________

 Next address generated, CRTN false:

                    BCY03-10       11-14      15      16
                   ___________________________________
                  |   RCH     |   CU 1-4   |  JA  |  JA  |
                   ___________________________________

 Next address generated, CRTN true:

                                BCY03-16
                   _________________________________
                  | Popped from microinstruction stack |
                   _________________________________
```

19.2.1.8 CALL w/ JUMP

The CALL [w/jump] operation works like a JUMP, but also pushes an address onto the microinstruction stack. Figure 19-8 illustrates this operation, which implements a conditional CALL.

FIG. 19-8. CALL w/ JUMP Operation

```
CALL w/ JUMP

Microcode Syntax:-CALL (n conds) TO (n**2 labels), n=1,2,3,4. [PUSH label]

Interpretation of CU field:

             CU 1,2    CU 0,3,4 CU 5-7 CU 8-10 CU 11-13  CU 14-16
             ----------------------------------------------------
            | BCY11-12 | JA 13 | JA 14 | JA 15 | JA 16  | CODE GRP |
             ----------------------------------------------------

Next address generated:

                           BCY03-10      11-12    13   14   15   16
                          ------------------------------------------
                         | RCH 03-10 | CU 1,2   | JA | JA | JA | JA |
                          ------------------------------------------

Address pushed onto RCS:

                              BCY03-10     11        12-16
                          -----------------------------------
                         |  RCH 03-10   | ~RCH11 |  RCH 12-16 |
                          -----------------------------------
```

19.2.1.9 CALL w/ GOTO

The CALL [w/GOTO] operation works like a GOTO, but also pushes an address on the microinstruction stack. Figure 19-9 illustrates this operation.

FIG. 19-9. CALL w/ GOTO Operation

```
CALL w/ GOTO

 Microcode Syntax:    CALL label [PUSH label]

 Interpretation of CU field:

                             CU 3-16
          ____________________________________________________
         |                   BCY03-16                         |
          ____________________________________________________

 Next address generated:

                                      BCY03-16
                      ________________________________________
                     |                CU 3-16                 |
                      ________________________________________

 Address pushed onto RCS:

                         BCY03-10       11        12-16
                      ___________________________________
                     |  RCH 03-10    | ~RCH11 |  RCH 12-16 |
                      ___________________________________
```

19.2.1.10 PUSH w/ EMIT

The PUSH [w/EMIT] operation generates a microsequencer address, which is determined by the current address with bit 12 inverted, and pushes the current address with bit 11 inverted onto the stack. Figure 19-10 illustrates this operation.

FIG. 19-10. PUSH w/ EMIT Operation

```
PUSH w/ EMIT

Microcode Syntax: PUSH [label] =  EMIT constant given in BLEG or DEST field

Interpretation of CU field:

                           CU 1-16
           ________________________________________________
          |                EMIT CONSTANT                   |
           ________________________________________________

Next address generated:

                    BCY03-11        12         13-16
                   ____________________________________
                  |   RCH 3-11    | ~RCH12 | RCH 13-16 |
                   ____________________________________

Address pushed onto RCS:

                    BCY03-10      11         12-16
                   _________________________________
                  | RCH 03-10   | ~RCH11 | RCH 12-16 |
                   _________________________________
```

19.2.2 Abnormal Microinstruction Sequencing

19.2.2.1 Traps

The microinstruction sequence can be altered by a trap. This causes the microcode to vector out of an instruction and handle some event, and then resume execution from where it was when the trap occurred.

When a trap occurs, BCY{13:16}+ are replaced with bits coming from either from E unit or the S unit, depending on which unit detected the trap. The E unit is the default case. The address of the microinstruction which would normally have been fetched next is pushed onto the microinstruction stack. E unit traps vector to octal locations 0-17, while S unit traps vector to octal locations 20-37.

The signals below are used in the trap logic:

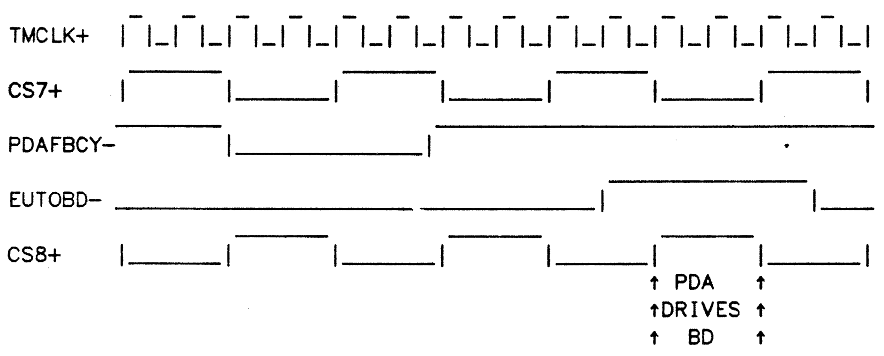
- FTRAP+
  This signal is generated by the I unit and causes the microsequencer to vector to an address specified by the trap address proms.

- FTRAPSEQA+
  This signal informs the CPU that the microcode currently executing is trap microcode and therefore hold onto any information pertaining to the instruction trapped out of.

19.2.2.2 FORCEBCY

With microcode assistance, the PDA can force execution to proceed from a particular address by asserting FORCEBCY+ at TRCML+. This signal causes the microsequencer to vector to CS memory location 72 octal and begin executing microcode. The microcode takes the data from BD and loads it into RBCY high and low. Now the microsequencer does a BDH branch and executes from that new BCY address. Figure 19-11 illustrates this operation.

FIG. 19-11. FORCEBCY Timing



19.2.2.3 Effective Address Formation

The signal FGEAF+ tells the microsequencer to get the next address (if at fetch level) from the EAF unit instead of the decode net. This signal goes active when the current instruction's operand can't be calculated without the use of the E unit's ALU. The EAF unit sends four bits to provide sixteen decode points into the microcode. These four bits effect BCY bits 13, 14, 15, and 16.

19.3 RCC Latches

The microcode bits, commonly called the RCCs, must be sent to all the CPU functional units to be clocked by TRCML+. Some units in the system need the current microcode word to be held for them until CS7+ of the current step, in which case they are latched with TRCML- and then clocked by the appropriate unit at CS7+. Other units need the RCCs clocked at TRCML+. Signals that are latched are called RCCxx, while those that are clocked are called RCMxx. Table 19-2 lists of all the RCCs and RCMs sent to other units on a board by board basis, and what is done to them before they leave the CS.

TABLE 19-2. RCC/RCM Distribution Summary

RCC field	RCC Latched at TRCML	RCM Clocked at TRCML	Board receiving RCC field
RCCBB	X		E IS
RCCTXNX	X	X	IS CMI PDA
RCCDST	X	X	E IS CMI
RCCRF	X		E
RCCCU3-13	X	X	E CMI
RCCCU1-2	X		E IS CMI
RCCCU14-16	X		E IS CMI
RCCCS	X		IS CMI
RCCALU	X		E
RCCEAE	X		IS
RCCBDL	X		E
RCCIAC	X	X	E IS CMI

19.4 VLSI Usage

The microsequencer is implemented using two PUSEQ VLSI chips in a bit-slice fashion. Each chip produces half the BCYs used to address the CS memory, as well as having internal logic to support the microinstruction stack, jump net, and trap address formation.

19.5 Critical Paths

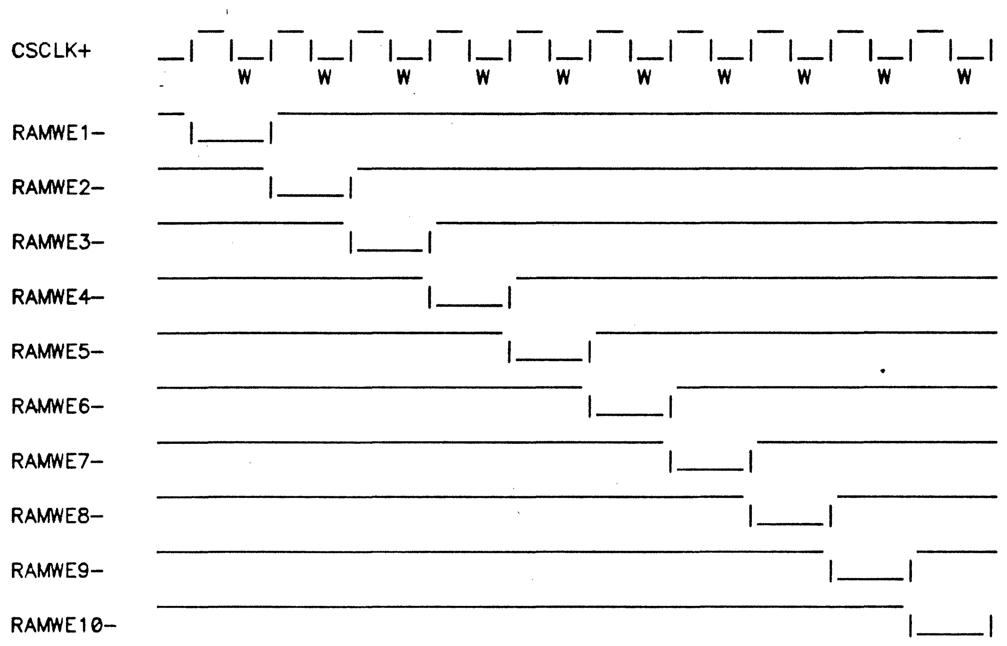
The major path through the CS involves receiving a jump condition from another unit, changing the BCYs, and accessing the CS RAMs. This is a two beat path from CS8+ to TRCML+.

Conditional ReTurNs (CRTNs) are critical, but TXNX logic is used to make this path work because of the I unit intervention needed. This is a two beat path from CS8+ to TRCML+.

The internal PUSEQ push/pop stack path is critical, due to the time required from TSTACK+ to TWRITE-.

19.6 Timing Diagrams

FIG. 19-12. Control Store Memory Write Cycle Timing



19.7 9755 Comparisons

The 4150 control store is 16K deep. The 9755 control store was 5K deep.

The 4150 control store includes 1 parity bit for every 9 data bits. The 9755 included 1 parity bit for every 8 data bits.

The CU, BDL, and BB fields have an extra bit added to them in the 4150 control store. Also, the EAE field was shortened by one bit.

19.8 Partitioning

All CS functionality is implemented on the CMI board, which is discussed in chapter 30.

20. Instruction Decode Detailed Description

20.1 Instruction Fetch

The Instruction (I) unit receives two words (32 bits) of instruction stream data fetched by the Storage management (S) unit during stage 2. This data is clocked into the Cache Set Select (PCSS) VLSI chip at the end of stage 2.

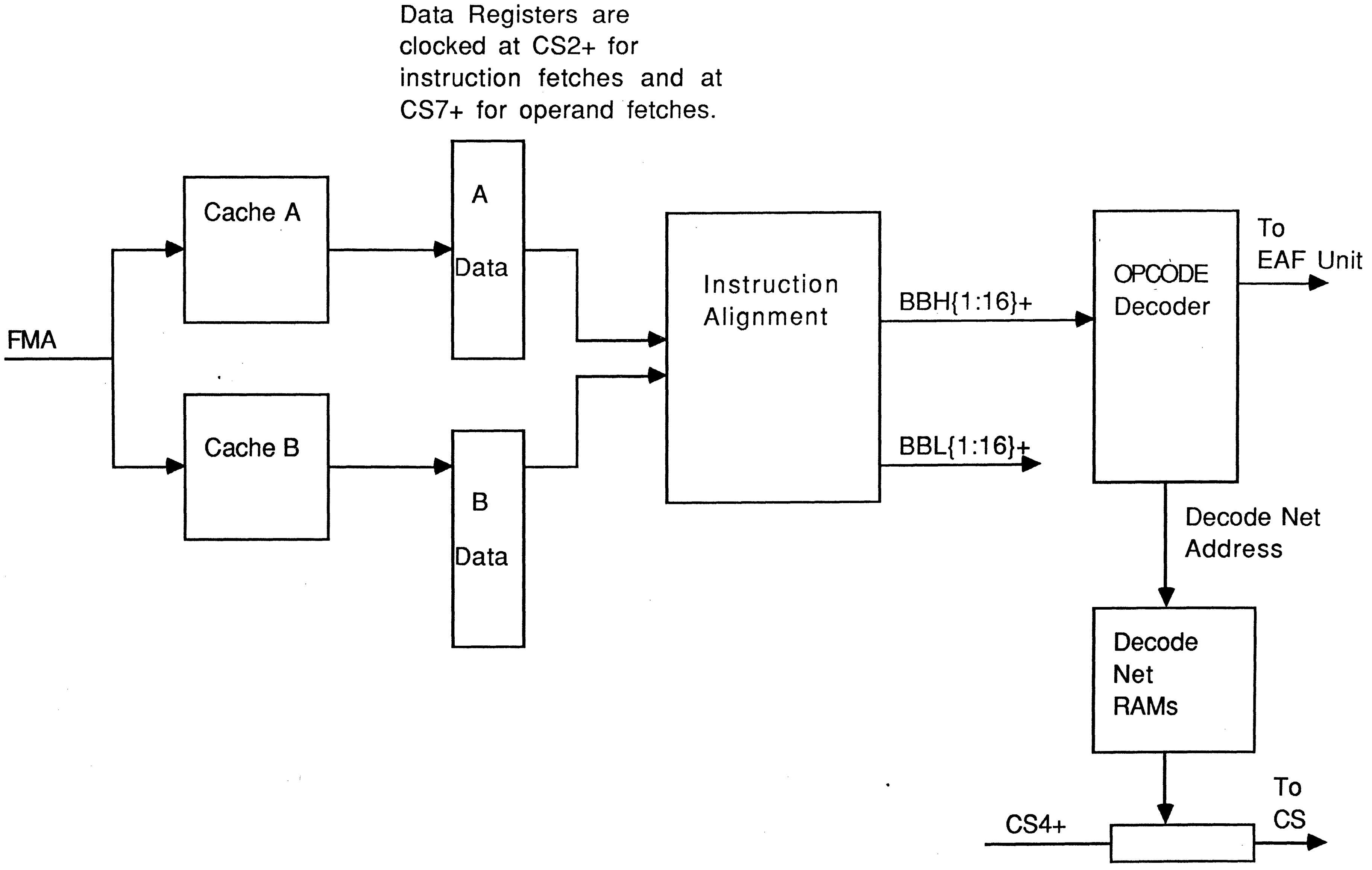


Figure 20-1 Block Diagram of Instruction Decode Unit

20.2 Instruction Stream Formats

Four instruction formats in the instruction stream are possible. An instruction may be long or short, unaligned or aligned. Alignment refers to whether the instruction starts on a Register Program counter Low (RPL) even (RPL16=0) or odd boundary (RPL16=1). Regardless of the composition of the instruction stream, the data just fetched is displayed on Bus B (BB) for other units to examine during stage 3 as illustrated in Table 20-1.

TABLE 20-1. BB Data During Different Pipeline Stages

INSTRUCTION #	0	1	2	3	4	BBH / BBL
STAGE	5	3	1			OPCODE 1 / DISP 0
	6	4	2			——— ———
	7	5	3	1		OPCODE 2 / DISP 1
	8	6	4	2		OPERAND 0
		7	5	3	1	OPCODE 3 / DISP 2
		8	6	4	2	OPERAND 1

Instruction alignment is needed because the decode net and effective address calculation hardware are dedicated to BBH and BBL respectively. The PCSS chip contains swap muxes for the instruction alignments. Instruction swaps are determined by the nature of the last instruction. By using the current RP and the result from the last instruction's decode indicating instruction length, the swap control for the current prefetched instruction can be determined. The instruction length of the current instruction is determined by the decode net control bit 1 (DNCNTRL1+) for generic instructions, or the outputs of the Effective Address Formation (EAF) unit for other instructions. Swaps are required if the last instruction was found to be aligned (RP=even) short or unaligned (RP=odd) long. The four possibilities are discussed in the following sections.

20.2.1 Aligned Short Instructions

On an aligned instruction fetch (RP=even), the first word contains opcode information. It is put on BBH and the second word is clocked in a register inside the PCSS chip and held until stage 5. By this time the first word has been decoded and determined to be a short instruction. The data word saved in the PCSS chip is therefore the next opcode. A swap occurs inside the PCSS chip, and this word is driven out on BBH.

20.2.2 Aligned Long Instructions

On an aligned instruction fetch (RP=even), the first word contains opcode information. It is put on BBH and the second word is clocked in a register inside the PCSS chip and held until stage 5. By this time the first word has been decoded and determined to be a long instruction. The data word saved in the PCSS chip is therefore the displacement portion of the instruction. It is driven out on BBL for the effective address calculation of the current instruction.

20.2.3 Unaligned Short Instructions

On an unaligned instruction fetch (RP=even) the second word contains the opcode information and is put on BBH.

20.2.4 Unaligned Long Instructions

On an unaligned instruction fetch (RP=even) the second word contains the opcode information and is put on BBH.

The displacement word is not available until the next stage 2 (stage 4 of the current instruction). The displacement word is then driven onto BBL during stage 5. Note that a swap is necessary inside the PCSS chip to accomplish this.

20.3 Instruction Decode

The properly aligned instruction information is put on BB during stage 3 and latched at the end of the beat. The instruction is decoded during stages 3 and 4 with pertinent information stored in registers clocked at the end of stage 4. The instruction decoding process consists of:

- Generating the microcode entry point from the decode net to be clocked at the end of stage 4. If the Execution (E) unit is not running multi-microcode and the current instruction does not require microcode EAF help, the decode net entry point is used to address the Control Store during stage 5.
- Determining the form of the instruction.
- Generating the base and index register file addresses to be clocked at the end of stage 4 for EAF calculation during stages 5 and 6.
- Determining the type of EAF to perform.

The microsequencer contains hardware to map the instruction opcode to a decode net address during stage 3. The decode net is accessed during stage 4 and the microcode entry point for the specific instruction is stored in registers at the end of the beat.

20.3.1 Decode Net

The decode net is an 8K x 20-bit lookup table. Twelve of the 20 bits comprise the decode net data sent to the microsequencer for decode net address calculation. Six other bits are control bits for the I unit to use, and the last 2 bits are for parity.

The address generation for the decode net is produced from the opcode bits by the microsequencer. The addressing is not straightforward due to the two pieces of information the decode net must supply. The first is an entry point in the control store for the microcode to start executing the instruction being decoded. The second is register file tracking information for the EAF unit. The encoding performed on the opcodes separates the instructions into different groups as follows:

- Generic B - non memory reference instructions, SRVI modes (e.g. IAB), addresses 0 to 1024 decimal
- Generic A - non memory reference instructions, SRVI modes (e.g. CRA), addresses 1025 to 2048 decimal
- I mode special memory reference (e.g. DFL), addresses 2049 to 2176 decimal
- I mode memory reference (e.g. LH), addresses 2177 to 2304 decimal
- S and R mode not double - single precision instructions (e.g. FA), addresses 2432 to 2560 decimal
- V mode memory reference - (e.g. LDA), addresses 2688 to 2816 decimal
- S and R mode double - double precision instructions (e.g. DFA), addresses 2944 to 3072 decimal
- Skips - instructions that skip, any mode (e.g. SS1), addresses 3073 to 3328 decimal
- Shifts - instructions that shift, any mode (e.g. LRL), addresses 3329 to 3584 decimal
- Register generics - instructions that operate on registers, any mode (e.g. LEQ), addresses 3585 to 4096 decimal

The different classifications of instructions are based on both the type of operation the instruction is trying to perform and on how the modals and keys are set. For more information on which instructions are in which categories see the System Architecture Reference Guide.

The decode net address consists of 13 bits, DNA{01:12}+ and DOUBLE-. Table 20-2 shows how the address bits are created.

TABLE 20-2. Decode Net Address Bit Generation

DECODE ADDR	GENA GENB	SHIFT	SKIP	REGISTER GENERICS	SR-MODE MEMORY	V-MODE MEMORY	I-MODE MEMORY	I-MODE SPECIAL
DNA01	0	1	1	1	1	1	1	1
DNA02	OPCD02+	1	0	OPCD02+	OPCD02+	OPCD02+	OPCD02+	OPCD02+
DNA03	OPCD07+	1	1	1	0	0	0	0
DNA04	OPCD08+	0	0	1	GOPCD13+	GOPCD13+	GTM0-	GTM0-
DNA05	OPCD09+	OPCD09+	OPCD09+	OPCD09+	GOPCD14+	GOPCD14+	OPCD04-	OPCD09+
DNA06	OPCD10+	OPCD10+	OPCD10+	OPCD10+	DBLMD-	1	0	0
DNA07	OPCD11+	OPCD11+	OPCD11+	OPCD11+	1	0	0	0
DNA08	OPCD12+	OPCD12+	OPCD12+	OPCD12+	0	0	0	1
DNA09	OPCD13+	OPCD07+	OPCD13+	OPCD13+	OPCD06+	OPCD06+	OPCD06+	OPCD07+
DNA10	OPCD14+	OPCD08+	OPCD14+	OPCD14+	OPCD05+	OPCD05+	OPCD05+	OPCD08+
DNA11	OPCD15+	OPCD15+	OPCD15+	OPCD15+	OPCD04-	OPCD04-	OPCD01-	OPCD01-
DNA12	OPCD16+	OPCD16+	OPCD16+	OPCD16+	OPCD03-	OPCD03-	OPCD03-	OPCD03-
DOUBLE	DBLMD-	DBLMD-	DBLMD-	DBLMD-	DBLMD-	DBLMD-	DBLMD-	DBLMD-

1. The terms GOPCD13+ and GOPCD14+ equal OPCODE13+ and OPCODE14+, respectively, except when they aren't really opcode bits but are part of a displacement, which is true for V-mode and R-mode short memory reference formats. (Note that in R-mode there are memory reference instructions which are one word but do NOT use the short memory FORMAT (e.g. stack pop)). In these cases GOPCD{13:14}+ are forced to zero.

2. The term GTM0- indicates that the Tag Modifier bits (OPCD10+, OPCD11+) are both zero if GTM0- = 0 and are not both zero if GTM0- = 1. GTM0- = 0 indicates a register-register or immediate format, GTM0- = 1 indicates a memory reference format.

3. The term DBLMD- indicates the state of the "double-precision" bit in the RMODE keys. DBLMD- = 0 implies the double-precision bit is set.

4. It should be noted that in some cases bits supplied for a particular class are not necessarily useful to that class, but are the most convenient for the hardware to supply.

There are six decode net bits (DNCNTRL{01:06}+) which are sent directly to other units to use as described below:

- DNCNTRL01+ is used to determine the length of generic instructions, including register generics.

  DNCNTRL01+ = 0 indicates a one word instruction (or GENAP, see below).

  DNCNTRL01+ = 1 indicates a two word instruction.

  DNCNTRL01+ serves no purpose at all for memory referencing instructions. It is set to 0 for all these cases.

- DNCNTRL02+ has two different meanings depending on the instruction type.

  DNCNTRL02+ = 1 for generic instructions indicates that this instruction is a GENAP.

  DNCNTRL02+ = 1 for memory referencing instructions indicates that the microcode desires that the effective address be incremented by two while it is being calculated. This is useful for DFAD and DFSB, which need to determine the exponent difference as quickly as possible. (In Prime's bizarre floating point format the exponent resides in the fourth word of a double precision memory argument.)

- DNCNTRL03+ is used to determine whether the instruction is permitted to attempt to use the branch cache. That is, the microcode for this instruction is prepared to deal with any branch cache hits that might occur. DNCNTRL03+ = 0 indicates that this is a branch or JMP instruction. DNCNTRL03+ = 1 indicates that it is not.

- DNCNTRL{04:05}+ are used as part of the register tracking mechanism. Since the architecture supports many different addressing modes, and thus many different ways to address/modify a register, the decode net specifies which register (if any) is modified by an instruction. The interpretation of these bits is shown in Table 20-3. They are used by the EAF unit during register file collision detection.

- DNCNTRL06+ is used for exponent tracking now that the EAF unit has a copy of both floating point registers (exponent only).

TABLE 20-3. DNCNTRL{04:05}+ Interpretation

ADDRESSING MODE	DNCNTRL04	DNCNTRL05	REGISTER MODIFIED
All non-32I	0	0	None
	0	1	XB
	1	0	Y
	1	1	X
32I	0	0	None
	0	1	XB
	1	0	GR(RDN)
	1	1	GR(RD)

Note: GR(RD) means that the destination register is one of the 8 32I mode general registers, specified by bits 7, 8, and 9 of the opcode. GR(RDN) means that the destination register is one of the four "odd" 32I general registers (GR1, GR3, GR5, GR7) selected in the same manner as GR(RD) but with the least significant bit forced to 1 instead of coming from opcode bit 9.

20.3.1.1 Decode Net Initialization

The Diagnostic Processor (DP) (or PDA) sends the decode net data and addresses to the CPU during system initialization. The DP interface is discussed in chapter 16.

The decode net is loaded by the microcode in the following manner:

1. SYSCLR microcode resets the double precision bit in the KEYS register, which allows access to the first 4K of the 8K decode net. The console command "LDNET" invokes the console microcode to assist in loading the decode net.

2. RMA is loaded with the address of the decode net location to be loaded. This is done by setting the bits in RMA to make the decode net addressing think the instruction is a Generic type B or A. This allows the first 2K of decode net to be accessed.

   The second 2K is loaded by setting a bit in the control store diagnostic register which overrides the decode net addressing logic and sets bit one of the address active. RMA is still loaded as if the first 2K of decode net is being addressed, but the signal from the diagnostic register called FBLKSEL+ causes the second 2K of decode net to be addressed.

   KEYS register bit 2, double .precision, is set by the console microcode, and the preceding pattern is repeated to load the other 4K of the decode net.

3. RMA is supplied to the microsequencer over BB during stage 8. The microsequencer takes the RMA address and produces a decode net address. The data for the decode net is held in RS and received on BDH and BDL. The write pulse is generated at CS8+, and all 20 bits are written at one time. Extensive use of TX and NX delays allows CS8+ to be positioned so that the address and data hold and setup times are correct.

20.3.2 Register File Address Generation

The opcode bits and addressing mode information are used to generate the base register file and index register file addresses during stage 4. The EAF unit needs this information to correctly calculate effective addresses. The term "base" register file address applies loosely since it may address a base, general, or floating point register. The general and floating point registers are used in some I mode instructions. The "base registers" are 32-bit registers while the index registers are 16-bit registers.

The base register file addresses are FBRFRDA{01:04}+. Table 20-4 shows how these bits are interpreted.

TABLE 20-4. Base Register File Address Selection

FBRFRDA{01:04}+	BRFH{1:16}+	BRFL{1:16}+	
0000	GR0H	GR0L	
0001	GR1H	GR1L	
0010	GR2H	GR2L	
0011	GR3H	GR3L	
0100	GR4H	GR4L	
0101	GR5H	GR5L	
0110	GR6H	GR6L	
0111	GR7H	GR7L	
1000	0	0	
1001	FLR0H	FLR0L	
1010	0	0	
1011	FLR1H	FLR1L	
1100	PBH	0	***
1101	SBH	SBL	
1110	LBH	LBL	
1111	XBH	XBL	

*** PBL is defined to be zero

For sector relative, procedure relative, and generic instructions IRPL{1:16}+ is selected instead of the low word from the base register file.

The hardware forces the selection of some base register (addresses 1100 to 1111 binary) under the following conditions:

- If the machine is in R, S, or V mode
- If the machine is in I mode and the instruction needs a base register. That is, the instruction is not a floating point or integer register-to-register instruction. This condition will make the signal GTMO+ go active.

The following conditions specify which base register to use:

- For memory reference instructions, refer to the System Architecture Guide. The register is determined by the opcode of the individual instruction.

- For generic instructions, PB is specifically selected to handle the possibility that the instruction might be a GENAP (which would require that the PBH be selected to form the address of the GENAP).

- During phase 1 of an indirect operation, PB is selected to obtain a zero to add to a possible post-index register.

- We must select PB on any instruction not explicitly needing some other base register. This avoids false register collision detection. (Refer to Chapter 21 for more detail.)

If we are not forcing the selection of some base register then we must be executing a 32I mode floating or integer register-to-register instruction. In this case, opcode bits 12 through 16 are used to select which register file location to address. Table 20-5 shows how these bits are interpreted.

TABLE 20-5. 32I Mode Extended Base Register Selection

OPCODE BITS 12	13	14	15	16	REGISTER
0	0	1	1	0	FLR0
0	1	1	1	0	FLR1
0	0	0	0	0	GR0
0	0	1	0	0	GR1
0	1	0	0	0	GR2
0	1	1	0	0	GR3
1	0	0	0	0	GR4
1	0	1	0	0	GR5
1	1	0	0	0	GR6
1	1	1	0	0	GR7

Tables 20-6 and 20-7 show the interpretation of the index register file address bits (FXRFDA{2:4}+). In I mode, these bits are a copy of OPCODE{12:14}+.

TABLE 20-6. I Mode Index Register Selection

FXRFRDA{2:4}+	XRF{1:16}+
000	GR0H
001	GR1H
010	GR2H
011	GR3H
100	GR4H
101	GR5H
110	GR6H
111	GR7H

TABLE 20-7. Non-I Mode Index Register Selection

OPCODE02+ (X field)	INDEX REGISTER
0	GR5H (Y register)
1	GR7H (X register)

20.4 VLSI Usage

Two PCSS VLSI chips are used for aligning the instruction stream.

Two PUSEQ VLSI chips are used in generating the decode net address.

20.5 9755 Comparisons

An extra bit has been added to the decode net control bit field to help with the tracking of floating point registers.

An extra address bit has been added to the decode net, which double sthe size to 8K locations. Since the decode net RAMs are 16K deep, no extra logic is needed to utilize the extra 4K except for the one address bit. This address bit is the KEYS register bit 2, the double precision bit. The extra address bit allows extra microcode entry points for PMA instructions in the native UNIX mode.

20.6 Critical Paths

Clocking the opcode at CS2+, forming the decode net address during stage 3, and clocking the contents of the decode net at CS4+. (2 beat path, needs to be done in 125 nsec) Worst case path involves using the DNCNTRL1+ signal to generate a unaligned condition to be clocked at CS4+. TRCDIE+ -> BBxx -> DNAxx -> DNCNTRL1+ -> LDNCTRL1+ -> LONG+ -> UNAL+ -> F4UNAL+. Time = 110.5 ns

TCLA+, used to latch the opcode information on the CMI board, must not be active at the end of the beat to avoid latching bad data. Using the signal FENEOB+IS3, which becomes active at midbeat, we need to close the latch in less then a 1/2 a beat. FENEOB+IS3 -> TCLA+. Time = 28 ns

20.7 Partitioning

The decode net is implemented on the CMI board, which is discussed in chapter 30. The rest of the I unit functionality is implemented on the IS board, which is discussed in chapter 31.

21. Effective Address Formation Detailed Description

When an instruction makes a memory reference, it provides information from which the virtual address can be calculated. This is referred to as calculating the effective address. Depending on the type of instruction, the information can be provided in several different formats, and the calculation done in various ways. This chapter gives a detailed description of how the Effective Address Formation (EAF) unit performs the effective address calculation.

21.1 Discussion of Basic Addressing Modes

The effective address hardware is implemented to handle without penalty the common cases of S, R, V, and I mode address formation such as BR + X + D, RP + X + D, etc. The EAF unit uses the contents of the fields in a memory reference instruction to select which of the four types of address formations to use:

- Direct
- Indexed
- Indirect
- Indirect indexed

The effective address calculation takes place during stages 5 and 6, with the final virtual (or absolute) address being loaded into the registers addressing cache and the STLB on the Storage management (S) unit at the end of stage 6.

The following is a brief description of the type of address formations supported on all 50 series machines. For a more detailed description see The System Architecture Guide.

- Direct Addressing - The effective address is calculated by adding the contents of the base register to the displacement (BR + D).
- Indexed Addressing - The effective address is calculated by adding the contents of the base register, index register, and the displacement (BR + X + D).
- Indirect Addressing - There are two forms of indirect addressing, short and long form:
    - Short Form Indirection (16-bit indirection) - Depending on the addressing mode, indirect addressing takes one of two forms. In the first, the displacement is treated as the address of a location in the procedure segment. The EAF unit then uses the contents of the addressed location as the effective address.

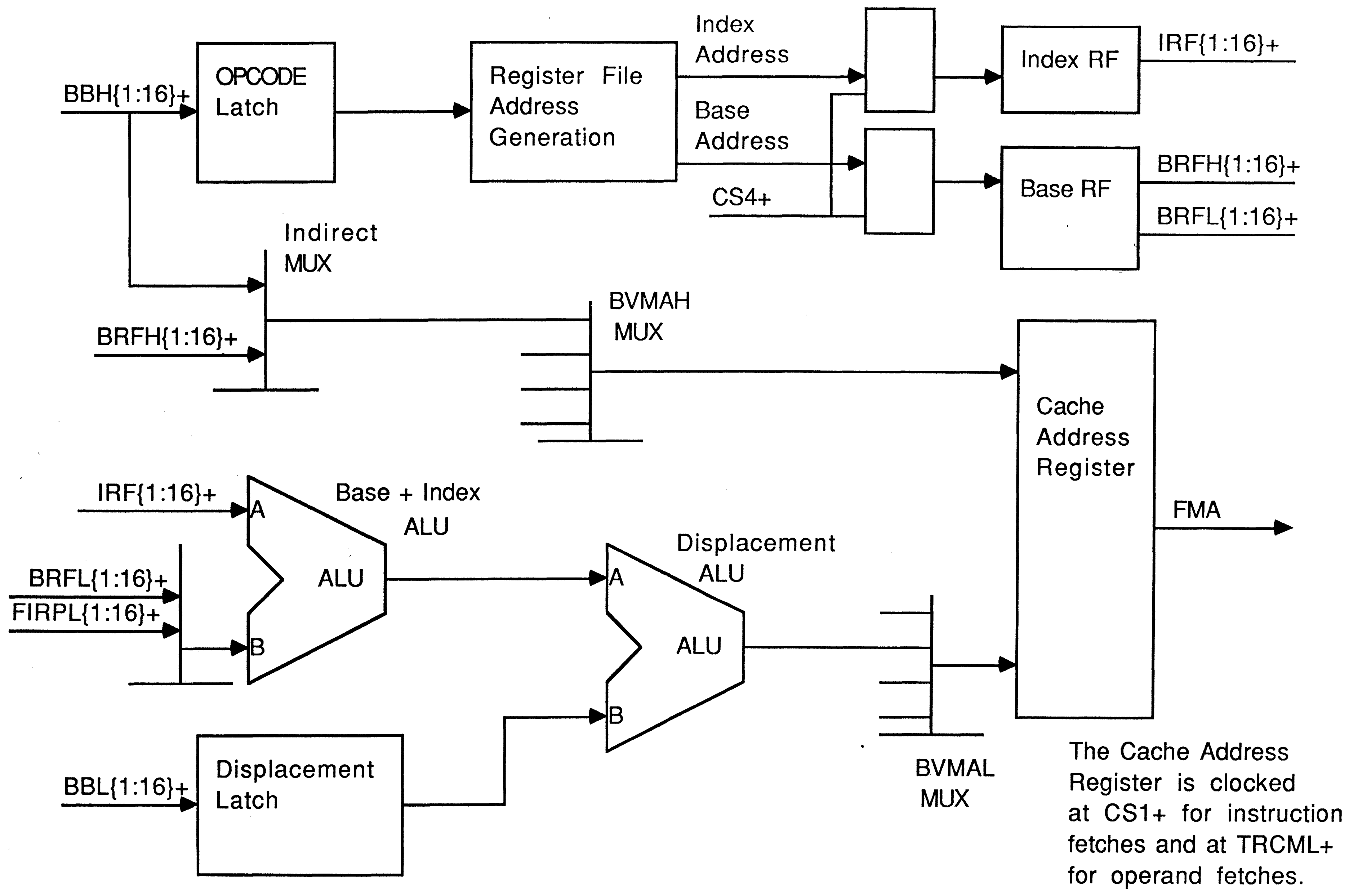


Figure 21-1 EAF Unit Block Diagram

Some addressing modes allow more than one level of indirection. In this second case, the displacement is treated as the address of some indirect address space. If this addressed location contains another indirect address, then the EAF unit uses these contents as the address of another location in memory. This indirection chain is followed until one addressed location does not contain an indirect address. The EAF unit uses the result of the chain as the effective address.

- o Long Form indirection (32 bit indirection) - In the long form of indirect addressing the displacement points to a location in memory that contains the 32-bit (or, occasionally, 48-bit) effective address.

- Indirect Indexed Addressing - This type of addressing takes one of two forms, indirect preindexed or indirect postindexed.

  - o When calculating a preindexed indirect address, the EAF unit adds the value of the index register to the contents of the base register and the displacement and uses the sum as an indirect address. It resolves any indirection chain and uses the result of the chain (or the indirect address itself, if there was no chain to follow) as the effective address.

  - o When calculating a postindexed indirect address, the EAF unit adds the contents of the base register and displacement and uses the result as an indirect address. It resolves any indirection chain, then adds the result of the chain (or the indirect address itself, if there was no chain to follow) to the contents of the specified index register to form the effective address.

21.2 Memory Reference Instruction Formats

Memory reference instructions can be short (one word) or long (two word). To help understand the detailed explanations of the EAF hardware in the various addressing modes, a brief summary of direct addressing is given below.

I mode	All memory reference instructions are two words long. The displacement is 16 bits, and is contained in the second word of the instruction. During EAF calculation this displacement is added to a base register. If no base register is specified in the instruction, PB is selected. PBL is defined as zero. Therefore, the displacement is added to zero to form the effective address.
V mode	Memory reference instructions can be one or two words long. For long memory reference instructions the displacement is contained in the second word. The effective address is calculated the same way as for I mode. Unlike I mode, V mode has short memory reference instructions. All short memory referencing instructions in V mode contain a sector bit. If the bit is active the instruction is using procedure relative addressing mode. The 9-bit displacement field is added to (or subtracted from, if it is negative) the Program Counter (IRPL) to form the effective address. It is important to

note that in procedure relative mode the displacement is added to IRPL and not to a base register.

If the sector bit is not active the instruction is using base register relative addressing mode. In this mode the 9-bit displacement field is used to encode both a base register and displacement. Depending on the address, the displacement is added to either the SB or LB base registers.

R mode

Like V mode, R mode contains both short and long memory referencing instructions. For long memory referencing instructions the displacement is contained in the second word. Opcode bits 15 and 16 are decoded to determine if the displacement is added to the SB register (stack relative) or to zero to form the effective address. For short memory referencing instructions, the type of operation is dependent on the sector bit. If the sector bit is active it is again procedure relative mode, and the sign extended displacement is added to IRPL to form the effective address. If the sector bit is zero, then the displacement is added to zero.

21.3 EAF Decoding

The EAF unit uses three PROMs to decode the necessary EAF operation. The microsequencer supplies the address for the EAF PROMs. This address is an encoding of the opcode bits, modals, and keys. Tables 21-1, 21-2, and 21-3 show the input and output signals used by the PROMs. Table 21-4 shows the opcode decoding necessary to determine the type of EAF necessary in I mode.

The outputs of the EAF decode PROMs supply the following information:

1. Four bits from the EAF PROMs are sent to the microsequencer to provide 16 decode points in the microcode.

2. Nine bits are used by the EAF unit to tell it what type of addressing needs to be performed. (INDIRECT+, LONG+, POSTINDEX+, PREINDEX+, 32ITA+, ENADRTR+, FEAF13+, FEAF14+, FGEAF+).

3. The EAF PROMs have a second job of generating ring weakening information. Two bits of the EAF decode address, FEAF13+ and FEAF14+, are used for this purpose. Ring weakening is discussed in Chapter 24.

The common outputs of the three EAF PROMs share a tristate bus. Depending on the addressing mode, one of the PROMs is enabled.

The EAF PROMs are accessed during stage 4 and all pertinent information is clocked at the end of stage 4.

TABLE 21-1. S or R Mode EAF Decoding

	Address definition
SRMODEEAF+	– SR mode instructions non generic non PIO
AM1+	– modal bit
AM2+	– modal bit
SRVLONG+	– SRV mode long displacements
DLT100+	– displacement less than 100 octal
OPCD01+	– indirection bit
OP2NLSX+	– Opcode 2 ORed into Lsx signal
OPCD15+	– CB bit *1
OPCD16+	– CB bit *1

	PROM output
PPREINDEX+	– EAF will be PREINDEXed
PLONG+	– instruction is Long
PEAF+	– EAF unit needs microcode EAF help
PPOSTINDEX+	– EAF will be POSTINDEXed
PEAF13+	– EAF decode address bit
PEAF14+	– EAF decode address bit
PEAF15+	– EAF decode address bit
PEAF16+	– EAF decode address bit

Note: The CB bits (bits 15 and 16) of R mode two word instructions distinguishes between long form memory and stack relative instruction types.

TABLE 21-2. V Mode EAF Decoding

	Address definition
VMODEEAF+	- V mode non generic
SRVLONG+	- SRV mode long displacements
DLT100+	- displacement less than 100 octal
LIVE+	- V mode displacement may be in register file range
LSX+	- instructions that load or store to X reg
OPCD07+	- sector bit for short form *1
OPCD01+	- indirection bit
OPCD02+	- using x register
OPCD12+	- using y register

	PROM outputs
PPREINDEX+	- EAF will be PREINDEXed
PLONG+	- instruction is long
PEAF+	- EAF unit needs microcode EAF help
POSTINDEX+	- EAF will be POSTINDEXed
PEAF13+	- EAF decode address bit
PEADRTR+	- Enables address traps
PEAF15+	- EAF decode address bit
PINDIRECT+	- EAF will be indirect

Note: If the instruction is short and the sector bit is set
then RP relative mode is used.

TABLE 21-3. I Mode EAF Decoding

Address definition

IMODEEAF+	- I mode non generic instructions
SKIPSPL+	- I mode special memory ref or V mode skips
OPCD10+	- TM bit defines type of EAF
OPCD11+	- TM bit defines type of EAF
OPCD12+	- SR bit
OPCD13+	- SR bit
OPCD14+	- SR bit
OPCD15+	- BR bit defines RR or Immediate
OPCD16+	- BR bit defines RR or Immediate

```
NOTE: TM = 3  indirect or indirect postindexed
      TM = 2  indirect or indirect preindexed
      TM = 1  direct or indexed
      TM = 0  Register-to-register(RR) or Immediate

      BR = 0  General RR
      BR = 1  Immediate Type 1 or Type 2
      BR = 2  Floating RR or Immediate Type 3
      BR = 3  Undefined


      SR = 0  GR0
      SR = 1  GR1 or FR1
      SR = 2  GR2
      SR = 3  GR3 or FR2
      SR = 4  GR4
      SR = 5  GR5
      SR = 6  GR6
      SR = 7  GR7
```

PROM outputs

PPREINDEX+	- EAF will be PREINDEXed
PLONG+	- instruction is long
PEAF+	- EAF unit needs microcode EAF help
POSTINDEX+	- EAF will be POSTINDEXed
PEAF13+	- EAF decode address bit
PEAF14+	- EAF decode address bit
P32ITA+	- I mode RR instruction/or Immediate
PINDIRECT+	- EAF wil be indirect

TABLE 21-4. I Mode EAF Function Table

	OPCODES											
	(4&5)	(10:16)			FEAF			INDEXING				
EAF type	SP	TM	SRC	BR	13:14	step	ita	pre	post	long	ind	weaken
Indirect	x	11	000	xx	00	IND_32I	0	0	0	1	1	1
IND Post-x	x	11	s>0	xx	00	IND_32I	0	0	1	1	1	1
Indirect	x	10	000	xx	00	IND_32I	0	0	0	1	1	1
IND Pre-x	x	10	s>0	xx	00	IND_32I	0	1	0	1	1	1
Direct	x	01	000	xx	—	–	0	0	0	1	0	1
Indexed	x	01	s>0	xx	—	–	0	1	1	1	0	1
Reg-Reg	0	00	xxx	00	01	–	1	0	0	0	0	0
Immediate1	0	00	000	01	11	–	1	0	0	1	0	0
Immediate2	0	00	s>0	01	10	–	0	0	0	1	0	0
Spec RR	1	00	xxx	00	10	UAF	0	0	0	0	0	–
Spec IM1or2	1	00	xxx	01	10	UAF	0	0	0	0	0	–
Immediate3	1	00	000	10	11	–	0	1	0	1	0	0
Floating-Reg	1	00	0x1	10	01	–	1	0	0	0	0	0
Non-Spec IM3	0	00	000	10	10	UAF	0	0	0	0	0	–
Non-Spec FR	0	00	s>0	10	10	UAF	0	0	0	0	0	–
Undefined	x	00	010	10	10	UAF	0	0	0	0	0	–
Undefined	x	00	1xx	10	10	UAF	0	0	0	0	0	–
Gen-Reg Rel	x	00	xxx	11	—	–	0	1	0	1	0	1

21.4 Microcode Assisted EAF

Instructions using the stack register or executing indirection or post-indexing need extra beats and microcode assistance to complete the effective address formation. If the instruction needs microcode assistance in the EAF calculation, the front end of the pipeline is held up and the microsequencer executes microcode to finish generating the effective address. The microcode entry point is determined by 4 control bits from the EAF PROM. The last step of the microcode executes a (CS= 2,DECODE), telling the PCU to return to fetch level and restart the front end of the pipeline.

21.5 Hardware EAF

The effective address is calculated by adding an offset to the first location within the segment the program is executing in. The segment number is generally provided in one of three ways:

1. If the instruction contains a base register, the segment number is found in the specified base register.

2. If the instruction does not contain a base register field, the segment number is found in the program counter (base register PB). In indirect addressing, the segment number field contains the segment number.

The offset portion of the effective address can be calculated in any of the following ways:

- Displacement (16 bit number given explicitly within the instruction)
- Displacement + offset from BR
- Displacement + index register
- Displacement + offset from BR + index register
- Indirect address
- Indirect address + index register

The EAF hardware includes two ALUs to do these calculations, the Register File ALU (ALUBX) and the Displacement ALU (ALUD). These ALUs calculate the 16-bit offset of the instruction.

ALUBX handles the Base Register + Index Register (BR + X) calculation. If no indexing is required, the ALU is put in Transport A mode, resulting in the contents of the selected Base Register, or IRPL (sector relative, procedure relative, generic instructions) appearing on ALUBX{1:16}+.

ALUD handles arithmetic operations between the output of ALUBX and the Displacement field. The term displacement is used rather loosely because the data on the displacement bus input can be one of several things, depending on the situation:

- Short memory reference instructions - 9-bit displacement extracted from the single instruction word plus 7 bits of sign extension (supplied by the displacement and sign extension logic in the PCSS VLSI chip)
- Long memory reference instructions - second word of the instruction
- 64V and 32I 32-bit indirects - The low order 16-bits of the 32-bit indirect pointer
- Two word generics - Second word of the instruction
- GENAPs - Same as one word generics, but irrelevant since ALUD will operate in TA mode (transport ALUBX) and ignore this data
- I mode integer and floating register to register - Again irrelevant since ALUD will operate in TA mode
- I mode immediate - Second word of the instruction (which has immediate data)

Tables 21-5 and 21-6 summarize the ALUBX and ALUD controls.

TABLE 21-5. Register File ALU (ALUBX) Control

	ALUBXADD+	ALUBXCIN-	OPERATION
TA	0	0	F= BR + 1
TA	0	1	F= BR
ADD	1	0	F= BR + X + 1
ADD	1	1	F= BR + X

TABLE 21-6. Displacement ALU (ALUD) Control

	TRNSPRTA+	TRNSPRTB+	EAFCIN-	OPERATION
	0	0	0	F = 1
ZERO	0	0	1	F = 0
	0	1	0	F = DISP + 1
TB	0	1	1	F = DISP
	1	0	0	F = BRX + 1
TA	1	0	1	F = BRX
	1	1	0	F = DISP + BRX + 1
ADD	1	1	1	F = DISP + BRX

21.5.1 I Mode Immediate and Register to Register Instructions

For all S, R, and V mode instructions, stages 5 and 6 are used to calculate effective addresses for operand reads during stage 7. For these instructions, the contents of the register file and displacement field are used to generate an effective address which is loaded into RMA at the end of stage 6. This address is used to access memory for the operand data during stage 7. The following instructions show some examples of this kind of operation:

```
LDA  MEM          EA = MEM        (DISP)
LDA  MEM,X        EA = MEM+X      (DISP+X)
LDA  MEM,LB0+7    EA = MEM+7+LB   (DISP+BR+X)
JMP  *-2          EA = IRP-2      (DISP+IRP)
```

The microcode for LDA, for example, simply does a cache read, passes the data through the Execution (E) unit's main ALU, and writes it into the A register in the register file. The microcode doesn't care how the effective address was formed. The address for the cache access was calculated by the EAF unit during stages 5 and 6.

For I mode register to register (RR) and immediate instructions, stages 5 and 6 are used to transfer operand information to RMA at the end of stage 6. No memory reference is required because the operand information is contained in the register file (for RR instructions) or in the displacement field (for immediate instructions).

There are three types of immediate instructions:

- Immediate Type 1 - (half word immediate) EAF unit transfers 16-bit immediate data located in second word of instruction into RMAH. Stores zeroes into RMAL.

- Immediate Type 2 - (full word immediate) EAF unit transfers 16-bit immediate data located in second word of instruction, sign-extended, into RMAH and RMAL.

- Immediate Type 3 - (Floating immediate) transfers high byte of immediate data located in second word of instruction to high byte of RMAH. Zeroes low byte of RMAH. Tranfers low byte of immediate data located in second word of instruction to low byte of RMAL. Zeroes high byte of RMAL.

Examples of these kinds of instructions are:

```
RR instructions:

        L GR1,GR2
        L GR3,GR4


Immediate instructions:

        L GR1,'123456     Immediate Type 1
        L GR1,'123456L    Immediate Type 2 (32 bit sign ext)
```

The operand information stored in RMA at the end of stage 6 is then accessed during stage 7 and transferred to the E unit. For example, the microcode for L GR1,GR2 transfers the contents of RMA into the destination register (GR1 in this case).

The following are examples of the EAF ALUs' control for some I mode instructions. Please refer to Tables 21-5 and 21-6.

```
Direct addressing    L GR1,MEM          ALUBX operation  FX = BR (BR=PBL=0)
                                        ALUD  operation  FD = DISP + FX

Direct indexed       L GR1,MEM,GR3      ALUBX operation  FX = BR + X (BR=PBL=0)
addressing                              ALUD  operation  FD = DISP + FX

Indirect             L GR1,MEM*         Phase 1:
addressing                              ALUBX operation  FX = BR (BR=PBL=0)
                                        ALUD  operation  FD = DISP + FX

                                        Phase 2:
                                        ALUBX operation  FX = BR (BR=PBL=0)
                                        ALUD  operation  FD = FX + DISP (DISP=IP)

Indirect             L GR1,MEM,GR3*     Phase 1:
addressing                              ALUBX operation FX = BR + X (BR=PBL=0)
preindexed                              ALUD  operation FD = FX + DISP

                                        Phase 2:
                                        ALUBX operation FX = BR (BRL=PBL=0)
                                        ALUD  operation FD = FX + DISP (DISP+IP)

Indirect             L GR1,MEM,*GR3     Phase 1:
addressing                              ALUBX operation FX = BR (BR=PBL=0)
postindexed                             ALUD  operation FD = FX + DISP

                                        Phase 2:
                                        ALUBX operation FX = BR + X (BR=PBL=0)
                                        ALUD  operation FD = FX + DISP (DISP+IP)
```

RR instruction	L GR1,GR3	ALUBX operation FX = BR (BR=GR3) ALUD operation FD = FX
Direct using base register	L GR1,LB0+4	ALUBX operation FX = BR (BR=XB) ALUD operation FD = FX + DISP (DISP=4)
Immediate Type 1 half word immediate	L GR1,'123456	ALUBX operation don't care ALUD operation FD = 0
Immediate Type 2 full word immediate	L GR1,'123456L	ALUBX operation don't care ALUD operation FD = DISP

21.5.2 Indirect Support

Instructions in which indirection is specified require two phases of execution. In the first phase, which is call Indirect Phase One (INDPHASE1-), the effective address is computed in the normal manner during stage 5 and 6, and the specified word is fetched. This address is called the indirect pointer (IP). The signal INDPHASE1- is active during stages 5 and 6. The signal FGEAF- becomes active at the end of stage 4. This signal then holds off stage 1 of instruction i+2 for 2 beats and stage 3 of instruction i for 2 beats. Refer to Figure 21-1. During this period, the EAF microcode step for indirection is executed. This microcode step reads the contents of cache and forms a second effective address and stores it into RMA.

FIG. 21-2. Pipeline Flow During Indirect

```
1             stage 1 of instruction i   (LDA TONY,*)
 2
1 3           stage 1 of instruction i+1
 2 4
   5 F    *
    6 T   +
1 3  F 7      stage 1 of instruction i+2
 2 4 6 T 8  -


* FGEAF- holds of stage 1 and 3
+ microcode assisted EAF step executed
  e.g I64RV microcode step for 64V indirection
- LDA microcode executed
```

In the second phase, which is called Indirect Phase Two (INDPHASE2-), the operation specified by the instruction is executed using the second address. The signal INDPHASE2- is active for the two beats.

The base register address is clocked at the end of stage 4 and at the end of the phase 1 (at INIT+) of an indirect operation. The appropriate base register is fetched during stage 5 for generating the address for the indirect pointer (IP). The additional clocking at INIT+ forces the fetching of PB. (PBL is all zeros, as discussed in Chapter 20.) The reason for this is that if post indexing is specified, we want to add the Index register to the IP. The Base + Index ALU (ALUBX) control is designed such that it will be in add mode at this time. Therefore by adding the index register to zero, the desired index register is effectively transported through ALUBX. It is then added to the IP via the Displacement ALU.

During the 1st beat of phase 2 of the indirect the low 16 bits of the IP are read out of cache and latched in the displacement latch. This value is then loaded into RMAL directly, or added to the index register (if Post-indexing) and then stored into RMAL.

The upper 16 bits of the IP (if 32 bit indirection) are read from cache and brought through a mux to RMAH during phase 2 of indirects. Note that if 16-bit indirection is specified, RMAH is not loaded at the end of phase 2.

21.6 Register File Tracking/Collisions

In a non-pipelined computer all the operations involved in executing a single instruction are completed before the next instruction is started. This is not true in a pipeline. Register collisions occur when two instructions in the pipeline need to use the same register at the same time. The register files on the EAF unit and E unit are written at stage 10 of the pipeline. The EAF unit reads its register file for EAF calculations and I mode RR operations during stages 5 and 6. Figure 21-2 shows that by the time that instruction i has finished with stage 10, instructions i+1 and i+2 would have BOTH finished with stage 6. That is, if instruction i was writing to a register file location (e.g. L GR1,TONY) that EITHER instruction i+1 (e.g. L GR3,GR1) or i+2 (e.g. L GR5,GR1) was using to calculate its effective address (or for an operand in I mode RR), there is a serious problem. Both instructions (i+1 and i+2) would have read the register file (during stage 5 and 6) BEFORE instruction i would have written it with the new data (during stage 10). Both of these cases create "register file collisions". The first case creates what is referred to as a 4-6 collision, while the latter case creates what is referred to as a 5-9 collision.

FIG. 21-3. Generalized Pipeline Flow

```
1               stage 1 of instruction i      (L GR1,TONY)
 2
1 3             stage 1 of instruction i+1    (L GR3,GR1)
 2 4
1 3 5 F         stage 1 of instruction 1+2    (L GR5,GR1)
 2 4 6 T
  3 5 F 7
   4 6 T 8
    5 F 7 9
     6 T 8 0   stage 10 of instruction i
               stage  8 of instruction i+1
               stage  6 of instruction i+2
```

21.6.1 5-9 Register Collision

A 5-9 register collision occurs when instruction i is modifying a register file location used by instruction i+2. The term "5-9" means that the register file collision is detected at stage 5 of instruction i+2 and stage 9 of instruction i. When this conflict occurs, stage 6 of instruction i+2 is held off for one beat until stage 10 for instruction i has finished executing (until instruction i has written the register file with the correct information). Figure 21-3 illustrates this action.

FIG. 21-4. 5-9 Register Collision Pipeline Flow

```
1               stage 1 of instruction i      (L GR1,TONY)
 2
1 3             stage 1 of instruction i+1    (NOP)
 2 4
1 3 5 F         stage 1 of instruction 1+2    (L GR3,D+GR1).
 2 4 6 T
  3 5 F 7
   4 6 T 8
    5 F 7 9 *
          0     stage 10 of instruction i
      6 T 8     stage 6 of instruction i+2


* 5-9 register collision detected, stage 6 of instruction i+2 is
  delayed 1 beat.
```

The register file detection logic for 5-9 collisions are implemented as follows: The register file read addresses (for both the base and index registers) are clocked at the end of stage 4. The register file write addresses are clocked at the end of stage 8. Stage 4 of instruction i+2 occurs at the same time as stage 8 of instruction i. The register file read address of instruction i+2 is compared to the register file write address of instruction i during the next beat. The signal BRC59+ refers to an address match of the base register and XRC59+ refers to an address match of the index register. These signals mean that a "potential" register file collision exists which needs to hold up the pipeline.

Special "bypass logic" is implemented that enables BVMA to be driven directly from BD. This bypass path is used during 5-9 collisions in I mode register-to-register instructions. For example in a sequence such as:

```
L R1,TONY
L R3,DOROHOV
A R2,R1
```

the EAF unit will fetch a stale value for R1. Rather than hold up the pipe for 1 beat and wait for R1 to be updated with the correct data and then read the register file, we take advantage of the fact that BD happens to have the right data (from the L R1,TONY instruction) coming over BD just as we are finishing stage 6 of the A R2,R1 instruction. Therefore we do not have to hold up the pipeline to wait for the register file to be updated.

This bypass path is only helpful for I mode RR instructions. Collisions involving the base or index registers cannot be bypassed, because the contents of those registers are used in the EAF calculation logic (e.g. L GR3,DISP+PB). Therefore, if BRC59+ is active or XRC59+ is active and we are doing indexing in the EAF phase (using the index register), then we have a register collision that requires holding up the pipeline. In these cases the signal FIWAIT- is asserted at the end of stage 5, causing the PCU to hold off all even clocks for one beat.

21.6.2 4-6 Register Collision

A 4-6 register collision occurs when instruction i is modifying a register file location used by instruction i+1. The term "4-6" means that the register file collision is detected at stage 4 of instruction i+1 and stage 6 of instruction i. When this conflict occurs, stage 5 of instruction i+1 is held off for two beats. This would make stage 5 of instruction i+1 happen at the same time as stage 9 of instruction i. At that point there is a 5-9 register collision, and the hardware discussed above (for 5-9 collisions) would come into play. Figure 21-4 illustrates this type of collision.

FIG. 21-5. 4-6 Register Collision Pipeline Flow

```
1                stage 1 of instruction i      (L GR1,TONY)
 2
1 3              stage 1 of instruction i+1    (L GR3,DISP+GR1)
 2 4
1 3 5 F          stage 1 of instruction 1+2
 2 4 6 T    +
     F 7
       T 8  $    NOP_STEP      (NOP)
  3 5 F 7 9 *
          0      stage 10 of instruction i
   4 6 T 8       stage 6 of instruction i+1

+ 4-6 register file collision detected, stage 5 of instruction i+1 is
      delayed 2 beats.

$ The E unit executes a NOP while waiting for stage 6
  of instruction i+1

* 5-9 register file collision detected, stage 6 of instruction i+1 is
      delayed 1 beat.
```

The register file detection logic for 4-6 collisions are implemented as follows: The register file write (or destination) address is generated on the E unit at the end of stage 8. This address is clocked on the EAF unit at the end of stage 9. Therefore, by the time the register file write address for instruction i is transferred to the EAF unit, instruction i+1 would have finished stage 7, and would have read stale data from the register file.

To handle register file collisions between consecutive instructions in the pipeline, we must know the register file destination address of instruction i in time to stop stage 6 of instruction i+1 from occurring. This critical knowledge is provided by the decode net.

Decode net control bits 4 thru 6 (DNCTRL{04:06}+) specify which register (if any) the impending instruction plans to modify (refer to Table 20-3). This information is clocked on the EAF unit at the end of stage 4 of the instruction.

Remember that stage 4 of instruction i corresponds to stage 2 of instruction i+1. During stages 3 and 4 of instruction i+1 (stages 5 and 6 of instruction i) the register file read address is being calculated. This read address is compared to the write address of instruction i. The signal BRC46+ means an address match of the base register, while XRC46+ means an address match of the base register. These signals mean that a register file collision exists that may need to hold up the pipeline. Just as the case for 5-9 collisions, special "bypass" logic exists which is used to alleviate 4-6 collisions during I mode RR instructions.

As discussed earlier, I mode RR instructions take information from the EAF unit's copy of the register file and put it into RMA at the end of stage 6. The E unit then copies RMA into the destination register. It also copies RMA into a special E unit register, RD, to assist the EAF unit in 4-6 register collisions. Now lets examine how this helps I mode RR instructions. Refer to Figure 21-5.

FIG. 21-6. 4-6 Register Collision Bypass

```
1               stage 1 of instruction i      (L GR1,GR2)
 2
1 3             stage 1 of instruction i+1    (L GR3,GR1)
 2 4
1 3 5 F         stage 1 of instruction 1+2
 2 4 6 T
  3 5 F 7
   4 6 T 8      *
    5 F 7 9
     6 7 8 0    stage 10 of instruction i

* A this point RD on the E unit has the contents of GR1
```

Notice that at the end of stage 6 of instruction i+1, RMA would have stale data from it's copy of GR1, but RD has the new version of GR1 being put there by instruction i. Therefore, by switching the BLEG selects on the E unit from selecting RMA (and stale data) to selecting RD (and new data), we can correctly execute instruction i+1 without holding up the pipeline. (RD and RMA are two different E unit BLEG sources.)

The signal USERD+ on the EAF unit does this switch. If BRC46+ is active and we are executing an I mode RR instruction, USERD+ becomes active at the end of stage 6. The E unit uses this signal to select RD rather than RMA.

Unfortunately this "bypass" logic works only for I mode RR instructions. Therefore, if BRC46+ is active AND we are using a base register (not I mode RR), or if XRC46+ is active AND we do need to do indexing during the EAF phase, then we must hold up the pipeline. Under these conditions, the signal FINOP+ is activated at the end of stage 4 of instruction i+1. This signal instructs the PCU to hold off stage 5 for two beats.

21.7 VLSI Usage

Two PCADR VLSI chips are used to implement the displacement ALU (ALUD).

21.8 9755 Comparisons

The EAF unit on the 4150 performs register tracking on the Floating Point registers. The 9755 did not. The reason the 9755 didn't have to is that all microcode transferring data to an floating point register also had an EAF step, which eliminates any potential register collisions.

The EAF unit has only one set of user registers in its register file. The 9755 had 4 user register sets in its register file.

21.9 Major and Critical Paths

One Beat Paths (62.5 nsec)

1. Register Collisions - 5-9 collision detection generating FIWAIT-. Path starts on E unit and ends on EAF unit. CS8+ -> RFDSTRAD5+ -> XR2FC9+ -> XRC59+ -> ENIWAIT- ->FIWAIT-(10E) at end of beat. Time = 58.0 ns.

2. FGEAF- generated at CS4+ to inhibiting CS1+ and CS3+ stage clocks at the end of the beat. Path starts on I unit and concludes on EAF unit. CS4+ -> FGEAF- -> HLD1- -> AENCS01- -> ENST1A+ -> inhibit the reset of ST1A+. Time = 61.0 ns. ST3A+ timing is identical.

Two Beat Paths (125 nsec)

EAF calculations. These paths start at CS4+ and end at TRCML+ two beats later. Indexing takes 117 ns. No indexing takes 115 ns.

21.10 Partitioning

The decode net and EAF PROMs are implemented on the CMI board, which is discussed in chapter 30. The rest of the EAF unit is implemented on the IS board, which is discussed in chapter 31.

22. Branch Cache Detailed Description

The branch cache logic consists of a 17-bit branch cache address register, a 1K x 24-bit branch cache array, some hit detection logic, some more logic used to validate the branch target, and some logic used to help manage IRP (I unit program counter) and detect branch cache gaffe traps.

The Execution (E) unit branch cache response logic influences the condition under test during branch instruction microcode, and controls the loading of the E unit RP.

The branch cache address register is loaded from BVMA at CS1+ with the current value of IRP. (In the event of an unaligned operand read and the loading of the address register happening simultaneously, BVMA will contain ERMA+1, not IRP. In this case, the branch cache registers will be loaded a beat after CS1+.) The branch cache address register drives the address lines of the branch cache array.

Each branch cache location contains a 16-bit branch target address (BRNA{01:16}+) and a 6-bit index which represents the high order branch cache address bits within a segment. An additional index bit, IRPH05+, is included in the index to help improve branch cache performance by separating branches in system code from user code. Each array entry also contains a valid bit (BCVLD+) and a bit used to track even vs. odd references.

The hit detect logic compares the current program counter to the index stored in the branch cache. A hit is prevented when the branch cache entry is invalid or if the branch cache is disabled by the addressable latch DISBCHITAL+. In addition, if the program counter was just loaded with an odd address and a branch cache hit is indicated for the even address the signal PASSEDHIT+ is activated, disabling the HIT+ signal. If a hit occurs, the branch target address BRNA{01:16}+ is loaded into IRP a half beat after CS2+. The BCHITORLD+ signal indicates whether the normal increment/hold of IRP should be performed or whether the branch cache target should be loaded into IRP.

The effective address of the branch instruction is compared against a staged version of the program counter to validate the branch. Since the branch cache hit logic doesn't have a full 32-bit tag and a notion of which user owns the entry, it is possible that the prediction logic can make a mistake. The GOODBR+ signal is a function of the partial validation signals. GOODBR+ cannot be trusted if an RP trap is pending. An erroneous validation may be indicated if the effective address was based on invalid cache data. In addition, the value of IRPL16+ is only valid if the IRP register was loaded. This implies that GOODBR+ cannot be trusted if a BCHIT+ and the corresponding IRP load didn't occur. The signal DISGOODBR- disables GOODBR+ from going active in these cases.

The Instruction (I) unit alignment logic looks at the state of the current IRP value, in addition to whether a branch cache hit has occurred, to determine whether the next opcode is

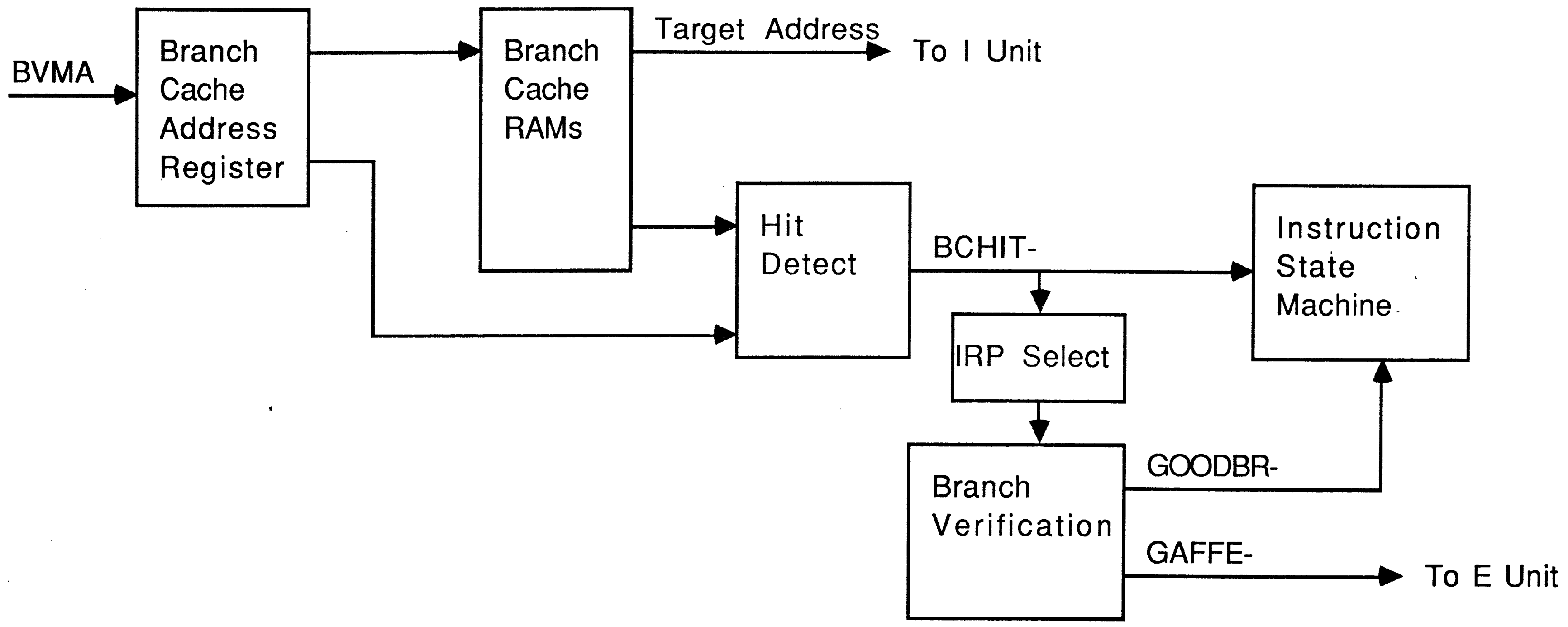


Figure 22-1 Block Diagram of Branch Cache

to be fetched from the high or low side of cache. Built into the RP clock is the knowledge of whether the branch occurred on the even or odd word and whether the instruction is long or short. If a branch cache hit occurred on the odd word and the even word is a short instruction, the IRP increment must be prevented, but the fact that a branch occurred must not be lost. If the above case occurs, TRPST+ will not occur at the normal time, CS3+, and the branch state will be held. FIRPL16+ will toggle, enabling the TRPST+ clock for the next CS3+.

The instruction state machine keeps track of instruction alignment in the instruction stream and issues the HIT+, GOODBR+, and BCGAFFE+ signals at the appropriate times. GOODBR+ and HIT+ are used by the E unit in calculating whether to CRTN-TO-FETCH or not. The BCGAFFE+ signal is issued when a BCHIT+ occurred on a non-branch instruction or on the opcode of a long unaligned branch. A trap occurs to get the instruction stream back in order.

Unaligned branches pose a difficult problem for the pipeline. The instruction fetch logic fetches 32 bits every other beat, and is expected to present the I unit with an opcode to be decoded every other beat. In addition, the I unit must provide the displacement to the EAF logic. For a normal unaligned instruction, the I unit provides for a bypass of the CS4 registers in the EAF unit. This allows the displacement of the previous instruction to be sent to the EAF logic at the same time as the opcode of the current instruction is being sent to the I unit.

In the case of an unaligned branch, the pipeline must wait for the displacement of the branch to be fetched before allowing the branch cache to vector to the target. This requirement is imposed so that the branch can be validated by the GOODBR logic. Unfortunately, by waiting for the displacement, the fetch of the target instruction was prevented. This condition starves the pipeline, and an additional fetch is required if a branch cache hit occurs.

A mixture of microcode and hardware is used to solve this problem. One can observe that most branch instruction require two microcode steps. The hardware, recognizing that an unaligned branch has occurred, can allow the pipeline to asynchronously fetch the target of the branch during execution of the first step of the algorithm, thereby filling the hole in the pipe. This is referred to as 'overlapped refill' of the pipeline. The signal that initiates this is F3GENAPST+. The pipeline enters the GENAP state during unaligned branches because the desired result, filling a hole in the pipeline, is the same function performed when skipping over the third word of three word GENAP instructions.

Since all branches are not multi-microcoded instructions, a method for dealing with the single microcode step branches is required. The microcode for the branches is carefully allocated such that single microcode step branches are allocated with BCY12+ active and multi-microcode step branches with BCY12+ inactive. In addition, a NOP microcode step is provided for each single microcode step branch that has the same microcode address as the branch, except that BCY12+ is inactive. Each NOP step then goes to its corresponding branch step. With this hardware and microcode in place, in the event of an unaligned branch, the hardware can simply force BCY12+ inactive by activating LDAENB12+. If the branch instruction requires more than one microcode step, the algorithm begins in the normal place. If the branch normally needs only one step the hardware forces execution to begin at the corresponding NOP step, and then execution proceeds as normal. The NOP step acts as a place holder for overlapped refill.

22.1 VLSI Usage

The branch cache logic partially controls the pre-IRPL mux selects in the Cache Address VLSI chip (PCADR). One mux select line is controlled by the branch cache hit detection logic, which will force the next update of IRPL to come from the branch cache array rather than the PCADR increment logic used to advance IRPL under normal operation. Thus the substitution of the branch target address for the next sequential IRPL value is done through a mux select line.

The PCADR part also produces the signal which lets the rest of the CPU know that the addresses involved in the current branch operation are valid.

The microsequencer (PUSEQ) VLSI chip on the CS unit has the ability to force decode net address bit 12 to a logic zero on unaligned branches.

22.2 9755 Comparisons

The 9755 branch cache was 256 locations deep. The 4150 branch cache is 1024 locations deep.

The 9755 didn't have the ability to do a full 32-bit branch cache validation. This meant that any branch that could possibly leave the segment was never allowed to have a validated branch cache entry. The 4150 has no such restriction.

A third difference involves a deficiency in the 4150 hardware. The value of IRPL16+ inside the Cache Address VLSI chip is not always accurate. This results in the branch cache validation not always being correct. To overcome this, the branch cache validation signal can only be trusted if a branch actually occurred. The effect of this difference is that the 9755 had the ability to forward branch by one, while the 4150 does not. (This is not a very useful function.)

Finally, system and user entries are isolated from each other in the 4150 branch cache. This was not done in the 9755.

22.3 Major and Critical Paths

For the most part there are no critical timing paths in the branch cache logic. The major path from the clocking of the branch cache address register, through branch cache hit detection, and switching the mux to meet the IRPL setup time has 1 1/2 beats to do the job. The branch cache validation path through the PCADR is somewhat critical. Here, the effective address calculation must be compared to the staged IRPL value in two beats.

22.4 Partitioning

The branch cache is implemented entirely on the IS board, which is discussed in chapter 31.

23. Cache Detailed Description

23.1 Associative Memory Introduction

Consider Table 23-0, which is to be stored in a computer's memory. It consists of a list of records for the 4150 baseball team. Each record contains 5 data fields:

1. The player's first name

2. The player's second name

3. The player's position

4. The player's batting average

5. The player's homerun total

Most information storage and retrieval problems involve accessing certain subfields within a set of records in answer to questions such as, "How many homeruns did Steve Small hit and what was his batting average?"

TABLE 23-1. 4150 Baseball Team Statistics

NAME	POSITION	BA	HR
Tony Dorohov	SS	.400	50
Steve Small	C	.275	10
Bob Parrow	LF	.350	30
Denise Chiacchia	P	.330	2
Tom Kinahan	1B	.225	15
Tom O'Brien	3B	.300	20
Carl Dimanno	CF	.100	1
Lynne Mulcahy	2B	.200	3
Mark Laird	RF	.250	5

If a conventional random access memory is being used, it is necessary to specify exactly the physical address of the Steve Small entry in the table, e.g., by the instruction

READ ROW 2

An alternative approach is to search the entire table using the NAME data field as an address. In such a system, the request for the data would be in the form of an instruction such as

READ NAME = Steve Small

The memory would be searched until Steve Small was found in the NAME field. This type of search is called an associative search.

An associative memory is one in which any stored item can be accessed directly by using the contents of the item in question as an address.

For example, to implement the above search algorithm we could have the direct mapped cache structure shown in Figure 23-1.

FIG. 23-1. Direct Mapped Cache (Analogy)

```
              MEMORY (CACHE)                        STLB
        _____________________________         ______________
        | Dimanno     CF .100  1    |         | Dimanno    |
        | Chiacchia   P  .330  2    |         | Chiacchia  |
        | Dorohov     SS .400 50    |         | Dorohov    |
   *    | O'Brien     3B .300 20    |         | O'Brien    |
        | Laird       RF .250  5    |         | Laird      |
        | Mulcahy     2B .200  3    |         | Mulcahy    |
        | Parrow      LF .350 30    |         | Parrow     |
        | Small       C  .275 10    |         | Small      |
        ____________________________          ______________
                      ↑                             ↑
                      |                             |
                      |                             |
           CACHE ADDRESS REGISTER          STLB ADDRESS REGISTER
         _____________________________     ____________________
         |      "First Name"         |     | Hash of 1st and  |
         _____________________________     | last name        |
                                           ____________________

 * Note that BOTH Kinahan and O'Brien map to the same location
   because their first names are both Tom. Therefore only one
   name can be stored in memory.
```

Each location in memory is stored according to the player's first name. No two player having the same first name can be in memory at the same time. Therefore, data about Tom Kinahan and Tom O'Brien cannot both be in cache simultaneously.

We address each location in the memory (analogous to 4150 cache) by using the FIRST NAME data field (analogous to the low 14 bits of the address field in the 4150). Each location in memory also contains a LAST NAME field (analogous to the INDEX in the 4150 cache).

Now assume we want the data related to "Tom O'Brien". We would load the memory address register with the FIRST NAME data field of "Tom". At the address specified by "Tom" we would read the following data out of memory:

O'Brien 3B .300 20

Now notice this very important point. The contents of cache may have contained the information for Tom Kinahan and not for Tom O'Brien. This is analogous to the cache miss cases for the 4150. To make sure we read the correct location, we have to compare the LAST NAME field located in the STLB to the LAST NAME field in the cache. If we were inquiring about Tom Kinahan, the contents of the STLB would have been the LAST NAME data field of "Kinahan", and we would be notified that the information we are looking for is not in cache. The information for the "correct" Tom would then be written into cache (via a cache miss operation) and we would then read the correct information.

The above is an example of a direct mapped cache. The problem with this type of organization is that we could not store both the data related to Tom Kinahan and the data related to Tom O'Brien in cache at the same time.

Now consider an associated memory organization, illustrated in Figure 23-2.

FIG. 23-2. Two Set Associative Memory (Analogy)

```
                 (SET A)                          (SET A)
              MEMORY (CACHE)                        STLB
          ________________________            ___________
         | Dimanno    CF .100  1  |          | Dimanno   |
         | Chiacchia  P  .330  2  |          | Chiacchia |
         | Dorohov    SS .400 50  |          | Dorohov   |
         | O'Brien    3B .300 20  |          | O'Brien   |
         | Laird      RF .250  5  |          | Laird     |
         | Mulcahy    2B .200  3  |          | Mulcahy   |
         | Parrow     LF .350 30  |          | Parrow    |
         | Small      C  .275 10  |          | Small     |
         ________________________             ___________
                     ↑                             ↑
                     |                             |
                     |                             |
                     |                             |
           CACHE ADDRESS REGISTER        STLB ADDRESS REGISTER
           ________________________         _______________
          |    "First Name"        |       | Hash of 1st and |
           ________________________        | last name       |
                   |                        _______________
                   |                             |
                   |                             |
                   |                             |
                   |                             |
     ______________|                             |_____________
    |                                                          |
    |            (SET B)                       (SET B)         |
    |         MEMORY (CACHE)                     STLB          |
    |         ________________________        ___________      |
    |        |                        |      |           |     |
    |        |                        |      |           |     |
    ------->|                        |      |           |<------
             | Kinahan 1b .225 15     |      | Kinahan   |
             |                        |      |           |
             |                        |      |           |
             |                        |      |           |
             |                        |      |           |
              ________________________        ___________
```

In this organization we have added another cache set and another STLB set. Now assume we were inquiring about Tom Kinahan. We would load the memory address registers with "Tom" and address BOTH caches and both STLBs. The cache in SET A would contain the information about Tom O'Brien and therefore would notify us that it does not contain the information we are looking for. However, the SET B cache would contain the correct information, and we would not take a cache miss.

This is an example of a two way set associative cache. The 4150 processor implements this type of memory structure in the cache.

23.2 Cache Organization

Cache memories are generally described in terms of total memory size. The 4150 'cache size' is 128 Kb.

The cache placement algorithm describes how the cache is organized. Ideally, the cache would be organized as a single fully associative memory, implying that any word in main memory could be assigned to any location in the cache. Each time the cache is accessed, all locations would be examined simultaneously, checking whether the physical address tag or 'index' matches the current physical address. However, since large associative memories are expensive and slow, caches are typically organized as a group of small, inexpensive, and relatively fast associative memories. Each associative memory is referred to as a 'set', and each set is made up of memory 'elements'. The number of elements in a set is considered the 'set size'. Each element contains a 'line' or 1/2 'block' of bytes that are transferred between memory and cache during an update. (A line is the amount of data transferred between cache and the E unit on a cache read. A block is the amount of data transferred between main memory and cache during a cache update.) Once a set is selected, the set is searched associatively for the appropriate element giving the desired line. From the line, a number of bytes, a 'word', are chosen and finally presented to the CPU.

Fully associative memories aren't used in the 4150. Instead, two caches are addressed simultaneously, giving the effect of an associative search. During a cache access, each cache array provides one of two elements of the selected set. The set selection is performed via a simple bit selection algorithm, whereby bits of the virtual address are used to directly address the cache array. The cache can be described as follows:

- cache size = 128k bytes
- sets = 16K
- set size = 2 elements
- block size = 8 bytes
- line size = 4 bytes

The 4150 cache employs a VLSI chip named the Cache Set Select option (PCSS). This part

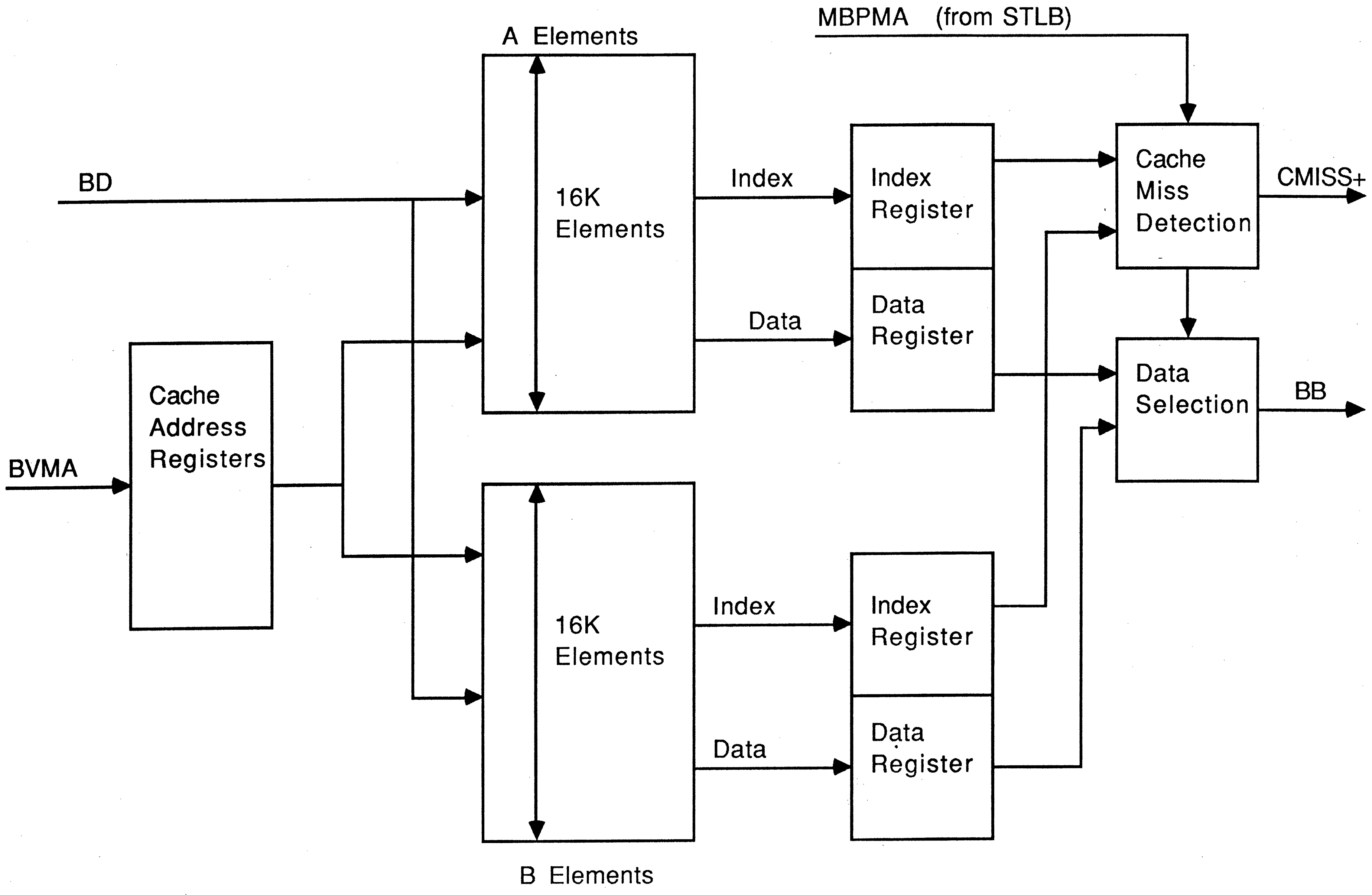


Figure 23-3 Block Diagram of Cache

chooses which element of the selected set is to be presented to the Execution (E) unit during operand fetches and to the Instruction (I) unit during instruction fetches. It's important to remember that the PCSS option doesn't select the set, it selects the desired element within a set !

23.3 Element Selection

The PCSS is the only source of BB. One important function is to provide 16-bit word swapped data to the E unit and instruction fetch logic. This function, in combination with the desire to minimize the loading of the cache outputs, not to mention pin limitations, motivated the interesting slicing of BB. Slice "A" of the PCSS drives the high bytes of BBH and BBL, while slice "B" drives the low bytes. This slicing facilitates 16x16 swaps by providing all the necessary data to a single slice without the need for a single cache data line driving more than one VLSI chip.

In addition, each PCSS option is dedicated to one of the two cache indices associated with a set. The basic algorithm of the PCSS is to compare the dedicated cache index to the physical page address presented to it by the STLB Set Select option (PSSS). If the comparison is true, the PCSS drives BB with data from the cache element it's dedicated to. Otherwise, it drives data from the other cache element.

It's very easy to speak of the dedicated cache element or other cache element from a single PCSS slice, but the situation becomes nearly hopeless when the cache is spoken of in its entirety. The dedicated cache element to one PCSS option is the "other" cache element to the second PCSS option. It is necessary to provide a means of describing the cache that is applicable on the chip as well as at the board level. During the development of the cache additional terminology was adopted to help characterize the cache.

The elements of all the sets are physically divided into two cache arrays. One cache, 'Cache A', contains all the element A's of all the sets, while the second cache, 'Cache B', contains all the element B's. Slice "A" of the PCSS is dedicated to 'Cache A', it considers the data it receives from cache A as the 'main' data and the data it receives from cache B as the 'auxiliary' data. Similarly, slice "B" of the PCSS is devoted to 'Cache B', and refers to the data from cache B as its 'main' data and the data from cache A as its 'auxiliary' data.

Cache data has been traditionally partitioned into an 'even' half and an 'odd' half. Even and odd refers to the 'high' or 'low' 16-bit words, and they get their names from the evenness or oddness of the address. Either or both sets of descriptors are acceptable and are riddled throughout the documentation.

Under normal operating conditions, each PCSS compares a copy of the physical page number to its dedicated index. If they are the same, the PCSS slice puts its main data on the BB outputs. Otherwise, the PCSS slice puts its auxiliary data out on BB. The result of the comparison is also presented at the output so that the external logic can determine whether a cache miss has occurred. A cache miss is detected when both PCSS options indicate that a miscompare has occurred, or when a parity error is detected.

23.4 Cache Replacement

The replacement algorithm of a cache refers to a method used in deciding which of the eligible cache elements should be replaced next. When both of the cache elements are used and a third data block needs to be brought in, the replacement algorithm dictates which element will be emptied to make room for the new data. Different replacement algorithms can be used to accomplish this task. Examples of replacement algorithms follow:

- LRU - update the Least Recently Used element
- FIFO - the First In is the First Out
- Random - randomly choose which to replace

The FIFO approach is used on the 4150 because of ease of implementation. The LRU provides the best performance, but requires status to be taken on every access. FIFO needs only to record status on updates. The random approach was dropped because of its unpredictable behavior.

The conventional means for implementing a FIFO algorithm is to maintain a status memory for the elements. The status memory is common for all elements and tracks information for a particular block for all the elements. Whenever an element needs to be updated, the status memory is queried for the appropriate block. This information indicates which element is to be modified. During the element update, the status memory is modified to indicate which element should be updated next time the block is to be modified.

The problem with the conventional method is that special control logic is required to update the status memory when any of the elements is modified.

The 4150 provides an efficient means of distributing the replacement algorithm between the elements without requiring a centralized status memory and unique control. Included with each element is an additional bit called the update bit. This bit, in conjunction with the corresponding bit in the other element, indicates which element is to be updated next. When the element is actually modified, this bit is toggled. Table 23-2 shows how the control logic interprets the update bits and decides which element to update next.

Included with each cache cell is a force bit. The force bit is used to mark a cache block permanently invalid. This bit is set by microcode when an unrecoverable error is detected in a cache cell. The idea here is that if a cell is stuck either high or low in the data or tag, then an additional error will be undetectable. To proceed executing after an error is detected

TABLE 23-2. FIFO Cache Replacement Algorithm

Current State		Element to modify	Next state	
a	b		a	b
0	0	a	1	0
1	0	b	1	1
1	1	a	0	1
0	1	b	0	0

is possible, by using one of the other eligible cache cells, but reliability is sacrificed. To avoid reducing the reliability in the name of enhanced performance and improved availability, the force bit is used to completely ignore the offending cache cell.

23.5 Cache Reads and Writes

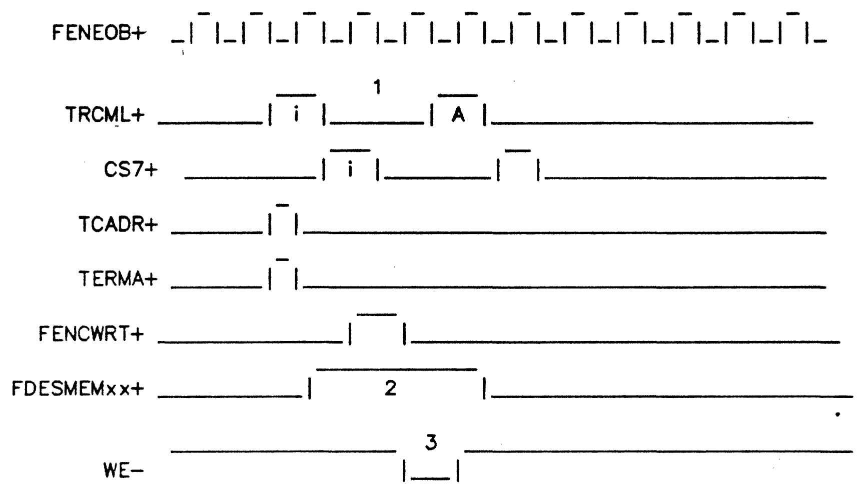
As discussed previously, each cache entry contains 32 bits of data, partitioned into 16-bit even and 16-bit odd data words. Cache reads or writes can be 16 bits or 32 bits. Which data location is read from or written to depends on the the least significant bit of the cache address. If the least significant bit of the cache address is a 1 the address is an odd address; if the least significant bit is a 0 the address is an even address.

Aligned writes are illustrated in Figure 23-4. The E unit sends the data to be written to the cache with the first 16 bits on BDH and the other 16 bits on BDL. If only 16 bits are being written, the E unit sends the 16 bits on both BDH and BDL.

Unaligned reads and writes occur if a 32-bit operation is made to an odd address.

FIG. 23-4. Aligned Cache Writes

Instruction "i" does a cache write.



```
Notes:
     1. All cache writes specify a TX=2
     2. FDESMEMxx+ is not a signal name.
        FDESMEMxx+ = FDESMEM32+ for 32 bit destinations
                   = FDESMEM16+ for 16 bit destinations
     3. Cache write would occurs at CS8.5 if TX=0
        WE- is not a signal name but is used to symbolize
        the cache write enables.
              TCAE- : write pulse to even side element A
              TCAO- : write pulse to odd side of element A
              TCBE- : write pulse to even side of element B
              TCBO- : write pulse to odd side of element B

            16 BIT EVEN WRITE    WE- = TCAE- or TCBE-
            16 BIT ODD WRITE     WE- = TCAO- or TCBO-
            32 BIT EVEN WRITE    WE- = TCAE- and TCAO-
                                       or TCBO- and TCBE-
```

23.5.1 Unaligned Cache Write

Unaligned writes are processed by a combination of hardware and microcode. Storage management (S) unit hardware handles the pointer manipulation, E unit hardware aligns the data correctly, and the microcode assists in writing it. Unaligned writes require four extra beats over the aligned case, one to reload the address and three to write the second word. They are handled differently depending on whether the microstep is reloading RMA or not. The two cases are discussed below.

23.5.1.1 Unaligned Write With No RMA Destination

The majority of unaligned writes are executed in store instructions which have no need to reload RMA. (Refer to Figure 23-5.)

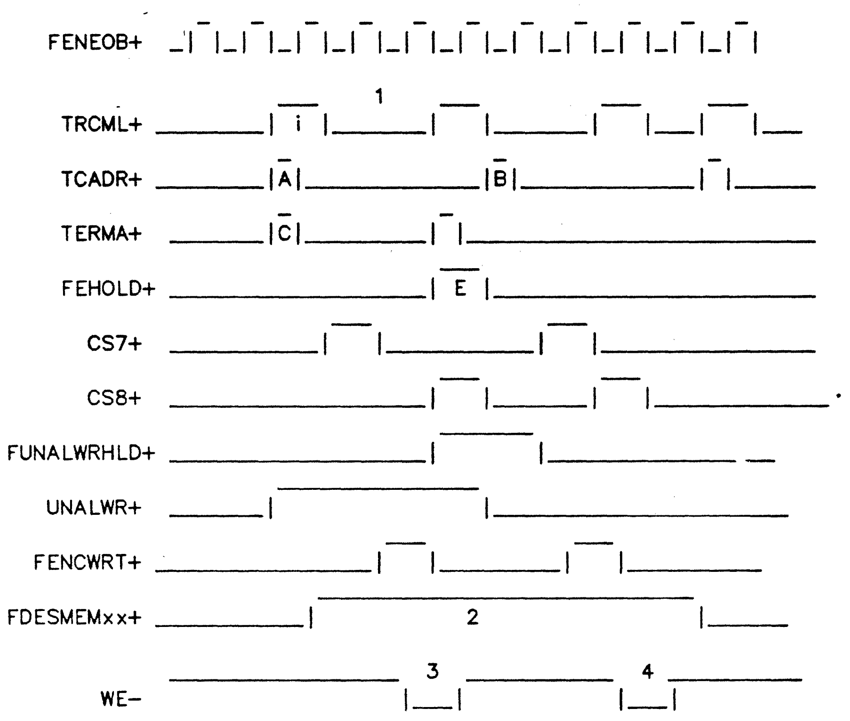
At TRCML+ of the unaligned write the cache address registers get loaded with the memory address of the first word. The E unit aligns the data so that the first 16 bits to be written appear on BDL. The cache detects the unaligned 32-bit write and automatically turns it into a 16-bit write to the odd side. During CS7+ of the write the BVMA selects switch to point to ERMA+1, the address of the second 16 bits. Therefore, by simply switching the BVMA selects we will load the address of the second word into the cache address registers at the next TRCML+.

Meanwhile, the microinstruction jumps on the LSB of the unincremented address to determine whether or not it was attempting an unaligned operation. If it was, it jumps to a cleanup step which knows that the address for writing the second word has already been loaded into RMA. It simply puts the second word onto BDH and does a 16-bit write. If we were writing three consecutive words (rather than 2), the cleanup step has the option of combining the second and third words and doing a 32-bit aligned write.

There must be a way of delaying the loading of cache address registers with the address of the second word until after the first write has completed. An NX= 2 and a RMA destination are conditionally inserted if a write turns out to be unaligned. This is accomplished by generating an FEHOLD+ for one beat starting at the next TRCML+.

FIG. 23-5. Unaligned Cache Writes: No RMA destination

Instruction "i" does a cache write.



Notes:

1. All cache writes with no RMA destination have TX=2 in the microcode step.
A. RMA and cache address register gets loaded with memory address of first word of write.
B. RMA and cache address registers get loaded with ERMA+1. BVMA selects are pointing to ERMA+1.
C. ERMA gets loaded with address of 1st word.
E. FEHOLD+ delays second write to cache until first write has finished.
2. FDESMEMxx+ is not a signal name.
   FDESMEMxx+ = FDESMEM32+ for 32 bit destinations
   = FDESMEM16+ for 16 bit destinations
3. Write first word to odd side of cache.
   WE- = TCAO- or TCAE-
4. Write second word to even side of cache.
   WE- = TCAE- or TCBE-

23.5.1.2 Unaligned Write With RMA Destination

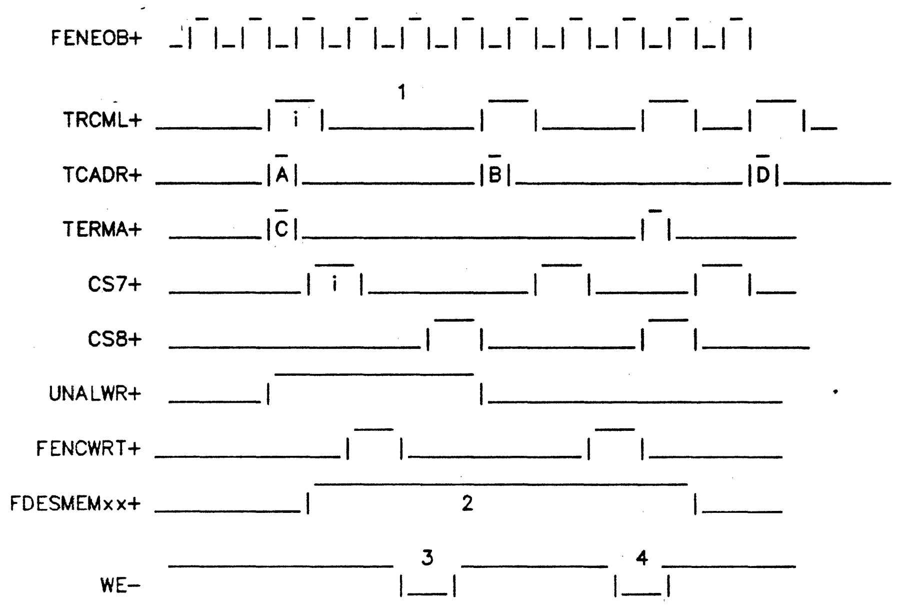
All 32-bit writes to cache with an RMA destination have a microcode restriction of TX= 2, NX= 2. This operation is similar to the unaligned write just discussed. Refer to Figure 23-6.

Note that because the write has a destination of RMA, a TCADR+ clock will occur at the next TRCML+. This TRCML+ is pushed out an additional two beats (with a NX=2) from CS8+ to allow the first data word to finish writing before reclocking the cache address registers. The EA>MA which the microstep was trying to do is lost and must be redone on the next step.

Meanwhile, the microstep jumps on the LSB of the unincremented address to determine whether or not it was attempting an unaligned operation. If it was, it jumps to a cleanup step which knows that the address for writing the second word has already been loaded into RMA. It simply puts the second word onto BDH and does a 16 bit write. If we were writing three consecutive words (rather than 2), the cleanup step has the option of combining the second and third words and doing a 32 bit aligned write. This step also re-executes the EA>MA which was overridden in the first step.

FIG. 23-6. Unaligned Cache Writes: RMA destination

Instruction "i" does a cache write.



Notes:

1. All cache writes with RMA destination specify a TX=2, NX=2 in the microcode.
A. RMA and cache address register gets loaded with memory address of first word of write.
B. RMA and cache address registers get loaded with ERMA+1. BVMA selects are pointing to ERMA+1
C. ERMA gets loaded with address of first word.
D. RMA and cache address register get loaded with EA>MA that ERMA+1>MA (note B) destroyed.
2. FDESMEMxx+ is not a signal name.
   FDESMEMxx+ = FDESMEM32+ for 32 bit destinations
   = FDESMEM16+ for 16 bit destinations
3. Write first word to odd side of cache.
   WE- = TCAO- or TCAE-
4. Write second word to even side of cache.
   WE- = TCAE- or TCBE-

23.5.2 Unaligned Cache Read

A 32-bit cache read from an odd address is referred to as an unaligned read. The first 16 bits of data are located at the odd address, while the second 16 bits are located at the next even address. Unaligned reads are handled by stopping the pipeline for 1 beat while the address of the second word is loaded in the cache address registers and a extra cache cycle is executed. Table 23-3 shows the starting memory image for Figure 23-7, which illustrates an unaligned read. The notes for Figure 23-7 are in Table 23-4.

TABLE 23-3. Memory Images for Figures 23-7, 23-8, 23-9, and 23-10

```
  500              L  GR1,LOC+1      instruction i
  502              L  GR2,LOC+4                  i+1
  504              L  GR3,LOC+6                  i+2
  506              L  GR4,LOC+8                  i+3
  510              L  GR5,LOC+10                 i+4
   .
   .
   .
 1000   LOC       DATA     0
 1001             DATA    10
 1002             DATA    20
 1003             DATA    30
 1004             DATA    40
 1005             DATA    50
 1006             DATA    60
 1007             DATA    70
 1010             DATA   100
 1011             DATA   110
```

FIG. 23-7. Unaligned Read Timing

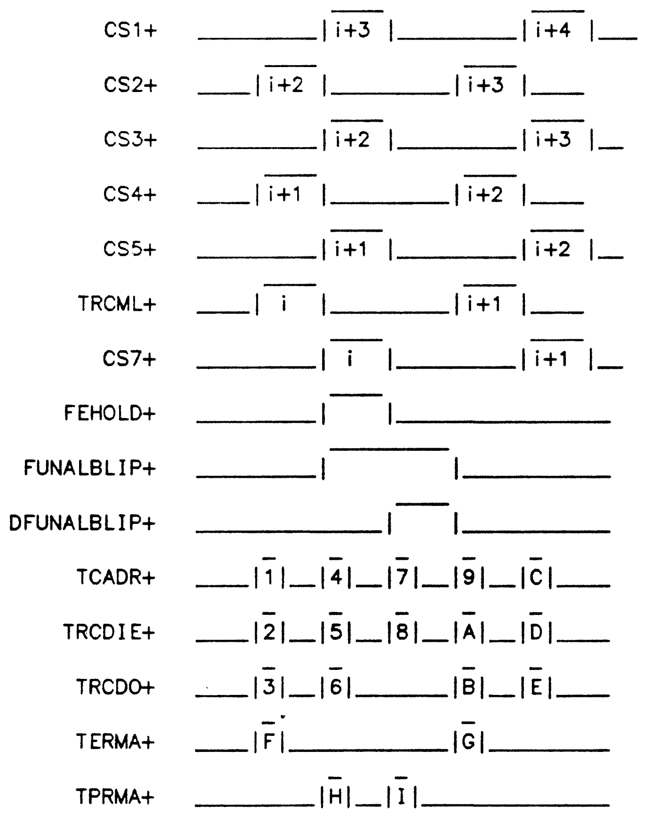


TABLE 23-4. Notes for Figure 23-7

1. Load RMA,and Cache Address Registers with '1001. The BVMA selects point to EA.
F. Load ERMA with '1001.
2 Load RCDE with Opcode (instruction i+2).
3. Load RCDO with Displacement (instruction i+2).
4. Load RMA and the Cache Address Registers with '1002. The BVMA selects, which normally would have pointed to IRP, now point to ERMAL+1. After the read of the second word of the unaligned, we will reload RMA with the RP address. See note 7.
H. Load PRMAL with '1002. Note that PRMAL should have been stored with the next RP address, but due to the unaligned miss, it gets loaded with ERMA+1. PRMAL will later get loaded with the correct address. See note I.
5. Load RCDE with '0 (contents of '1000). Note that because the 32-bit read is unaligned this data is no good.
6. Load RCDO with '10 (contents of '1001). Because the read is unaligned,this data is the first word, not the second word.
7. Load RMA with '506. We must now redo the CS1 that we lost due to the unaligned read. The BVMA selects points to IRP.
I. Load PRMAL with the correct RP address.
8. Load RCDE with '20 (contents of '1002). This is the second word of the data.
9. Load RMA and the cache address registers with '1004. Load ERMA with '1004. BVMA selects point to EA.
A. Load RCDE with Opcode (instruction i+3).
B. Load RCDO with Displacement (instruction i+3).
C. Load RMA and the cache address registers with '510. Load ERMA with '510. BVMA selects point to IRP.
D. Load RCDE with '40 (contents of '1004).
E. Load RCDO with '50 (contents of '1005).

23.6 Cache Miss

RP cache misses are enabled during stage 3 of the pipeline (the beat after CS2+) provided we are not retrying an RP address trap. The signal RPACC- specifies that RP cache misses are enabled.

EA cache misses are enabled if we are reading cache for the E unit (OPRD+) during the beat after CS7+ (or the beat after TRCDIE+ for the second word of an unaligned read) (FENEACMISS+), and we are not retrying an EA read address trap (FIACADRTR-). The signal EAACC- that EA misses are enabled.

Cache misses are enabled if RP misses or EA misses are enabled (GHOLDEN+), and no E unit trap (FEUTRAP-) or a memory trap (FMEMTRAP-) occurred during the last beat. Potential memory traps must take priority over cache misses because misses are invalid for certain type of memory traps. For example, read address traps override cache misses, while an STLB miss implies that the BPMA bits used in the cache miss detection logic for the index comparison are invalid. Deferring cache misses is never a problem, since if a trap occurs, the step getting the miss will be retried. If FMEMTRAP+ is raised without a trap occurring, the step will be retried as a NOP step.

If cache misses are enabled (DISMISSPE- not active), the signal GHOLD- becomes active if any of the following conditions are met:

- An index miscompare on both cache sets
- An parity error in either of the caches
- A shared bit being set in either

GHOLD- causes a cache miss to begin at the end of the beat. All register clocks which would normally occur at that time must be squashed so that invalid data is not sampled. Unfortunately, by the time a miss is detected it is to late to tell the Pipeline Control Unit (PCU) to inhibit all stage clocks in the machine. Instead, a special mechanism effectively freezes the entire pipeline for one beat. The timing involved in generating the GHOLD- signal is so tight that each board has to create its own version to save a gate delay. The cache unit sends over the following signals to enable each board to generate its own inhibit stage clock signal:

DISMISSPE–	disable cache misses
MISSPEA+	Index miscompare element A OR, Parity error on element A or B OR, Shared bit active in STLB element A or B
MISSPEB+	Index miscompare element B OR, Parity error on element A or B OR, Shared bit active in STLB element A or B
CMISSNDIS–	Cache miss enabled and index miscompare on both cache sets
CPEANDIS–	Cache miss enabled and cache parity error element A
CPEBNDIS–	Cache miss enabled and cache parity error element B
SHAREDNDIS–	Cache miss enabled and shared bit in STLB element A or B

The holding of the pipeline for the remainder of the miss is accomplished by the PCU.

Figure 23-8 illustrates the cache miss timing on the first word of a 32-bit read. The notes for this figure are in Table 23-5. Figure 23-9 shows the timing for a miss on the second word of the read, and its notes are in Table 23-6. Figure 23-10 shows the timing for an aligned miss on both words. Its notes are in Table 23-7. All three figures start with the memory image shown in Table 23-3.

FIG. 23-8. Unaligned Cache Miss: First Word only

Timing for EA cache miss at location 1001 for instruction i.

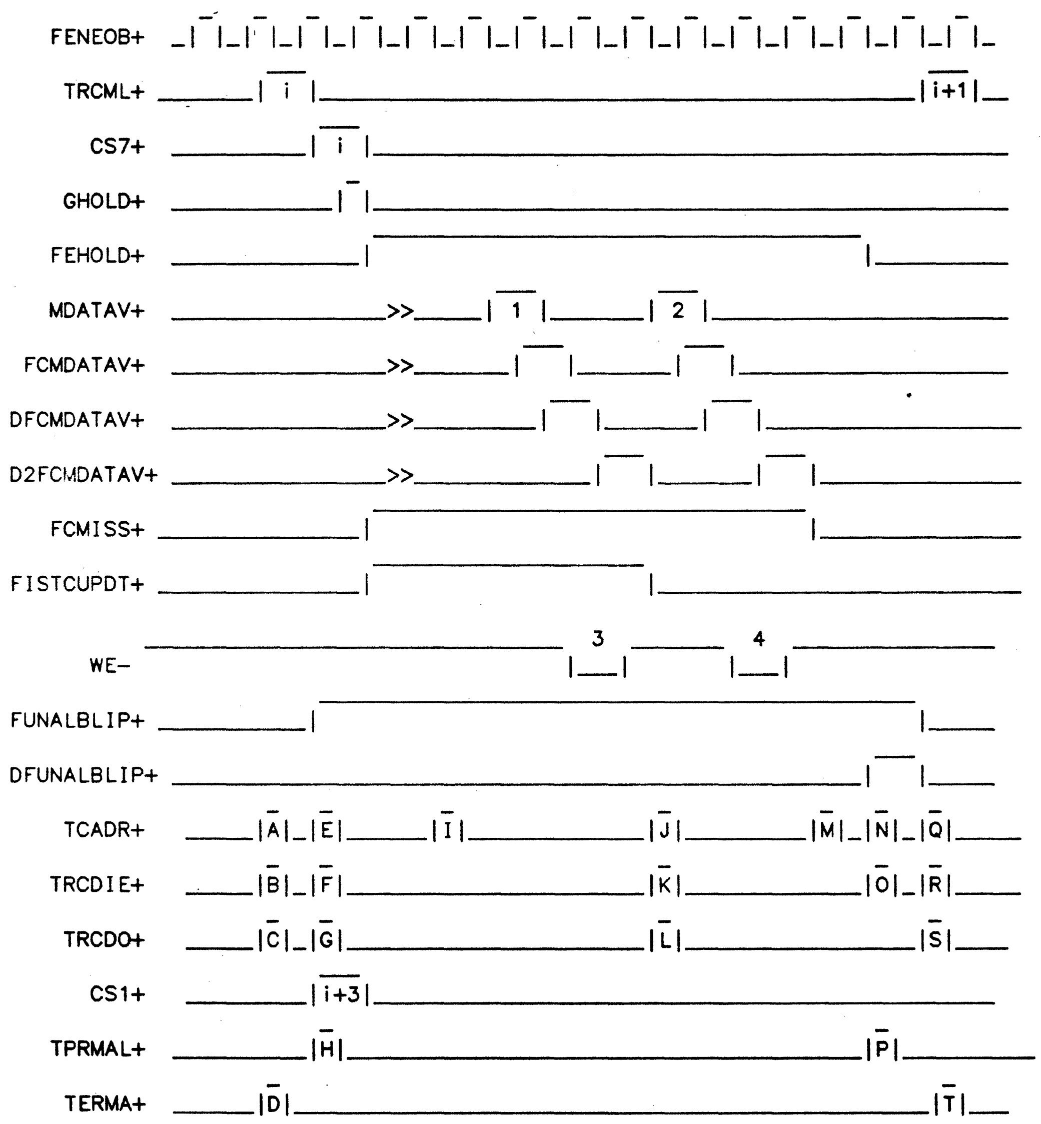


TABLE 23-5. Notes for Figure 23-8

1. MDATAV+ comes from the Memory Controller (MC), signifying that data is being sent over BD to the cache. The time between when FEHOLD+ becomes active to when the first MDATAV+ is sent depends on what the MC is doing when GHOLD+ is asserted.
2. The second MDATAV+ comes 2 beats after the first MDATAV+ if there is no ECC. If an ECC exists, the second MDATAV+ will follow the first by 4 beats.
A. Load RMA and Cache Address registers with '1001.
B. Load RCDE with Opcode (instruction i+2).
C. Load RCDO with Displacement (instruction i+2).
D. Load ERMA with '1001.
E. Load RMA and cache address registers with '1002.
   BVMA selects point to ERMA+1.
F. Load RCDE with xx (from location '1000).
   Cache miss so xx is bad data.
G. Load RCDO with xx (From location '1001).
   Cache miss so xx is bad data.
H. Load PRMAL with '1001.
I. Load cache address registers with '1001.
   BVMA selects point to ERMA.
   This address is sent to the MC over BB (after virtual to physical translation).
3. Write '0 to location '1000, '10 to location '1001 in cache data RAMs.
J. Load cache address registers with '1003.
   BVMA selects point to EA, bit 15 is inverted.
K. Load RCDE with '0 (Bad data due to unaligned).
L. Load RCDO with '10 (1st word).
4. Write '20 to location '1002, '30 to location '1003 in cache data RAMs.
M. Load RMA and cache address registers with '1002.
   BVMA selects point to ERMA+1.
N. Load RMA and cache address register with '506.
   BVMA selects point to IRP.
O. Load RCDE with '20 (correct second word).
P. Load PRMAL with '506.
Q. Load RMA and cache address registers with '1004.
R. Load RCDE with Opcode of instruction i+3.
S. Load RCDO with Displacement of instruction i+3.
T. Load ERMA with '1004.

FIG. 23-9. Unaligned Cache Miss: Second Word only

Timing for EA cache miss at location 1002 for instruction i.

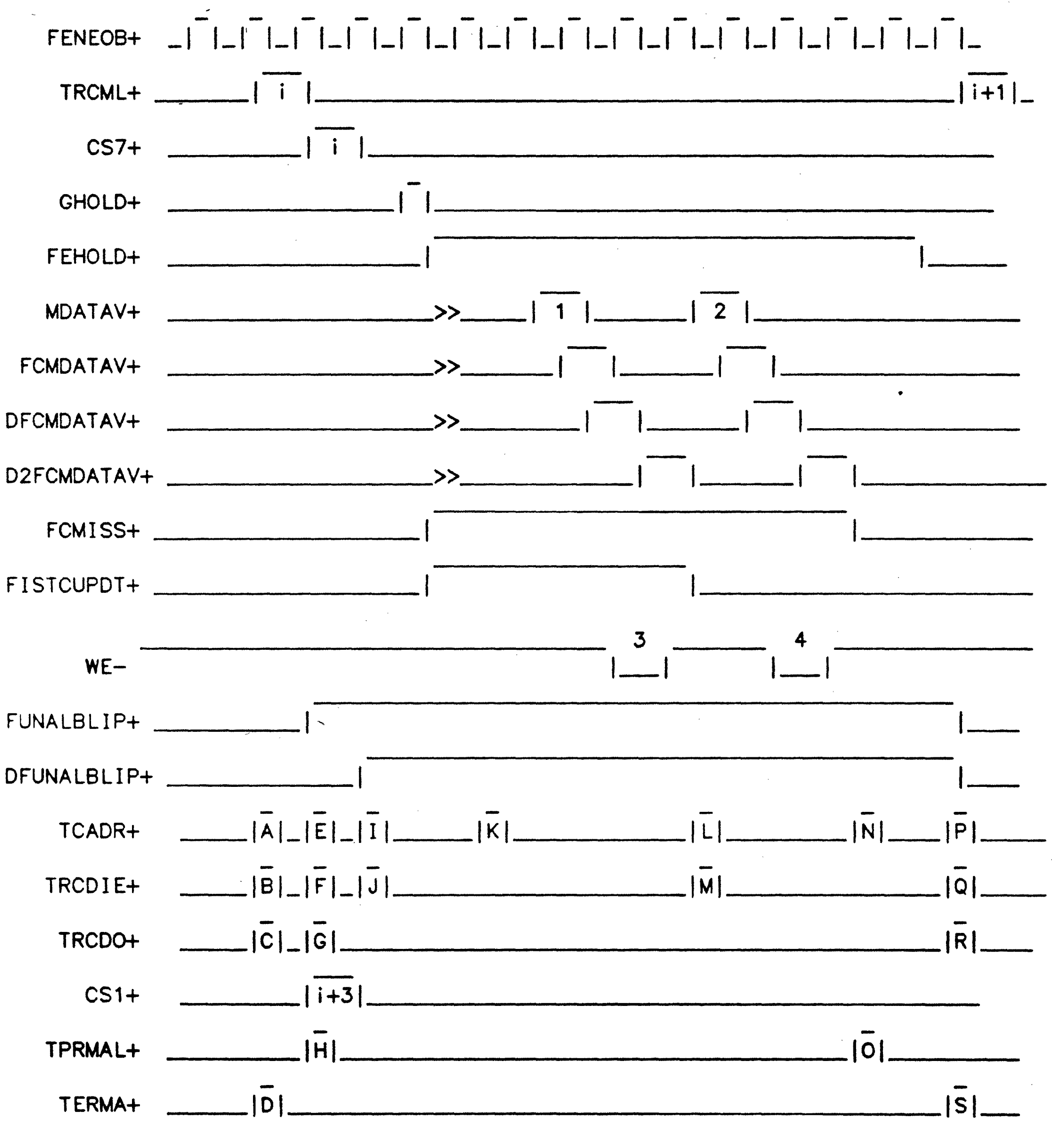


TABLE 23-6. Notes for Figure 23-9

1. MDATAV+ comes from the MC, signifying that data is being sent over BD to the cache. The time between when FEHOLD+ becomes active to when the first MDATAV+ is sent depends on what the MC is doing when GHOLD+ is asserted.
2. The second MDATAV+ comes 2 beats after the first MDATAV+ if there is no ECC. If an ECC exists, the second MDATAV+ will follow the first by 4 beats.

A. Load RMA and Cache Address registers with '1001.
B. Load RCDE with Opcode (instruction i+2).
C. Load RCDO with Displacement (instruction i+2).
D. Load ERMA with '1001.
E. Load RMA and cache address registers with '1002. BVMA selects point to ERMA+1.
F. Load RCDE with '0 (from location '1000).
G. Load RCDO with '10 (from location '1001).
H. Load PRMAL with '1001.
I. Load cache address registers with '510. BVMA selects point to IRP.
J. Load RCDE with xx (from '1002). Cache miss therefore xx is bad data.
K. Load cache address registers with '1002. BVMA selects point to ERMA+1. This address sent to MC over BB (after virtual to physical translation).

3. Write '20 to location '1002, '30 to location '1003. in cache data RAMs.

L. Load cache address registers with '1004. BVMA selects point to ERMA+I, bit 15 inverted.
M. Load RCDE with '20 (correct second word).

4. Write '0 to location '1000, '10 to location '1001 in cache data RAMs.

N. Load RMA and cache address registers with '506.
O. Load PRMAL with '506.
P. Load RMA and cache address registers with '1004.
Q. Load RCDE with Opcode of instruction i+3.
R. Load RCDO with Displacement of instruction i+3.
S. Load ERMA with '1004.

FIG. 23-10. Aligned Cache Miss

Timing for EA cache miss at location 1004 for instruction i+1.

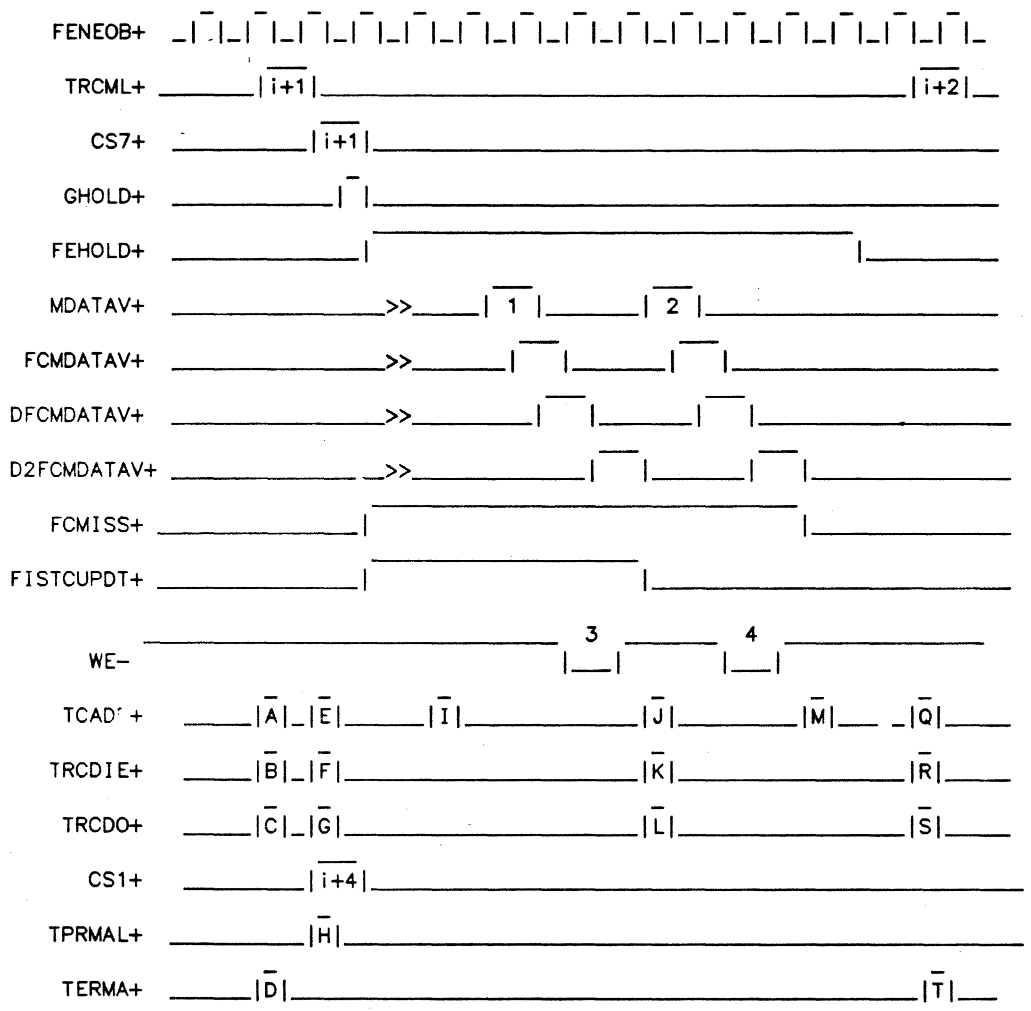


TABLE 23-7. Notes for Figure 23-10

```
1. MDATAV+ comes from the Memory Controller (MC),
   signifying that data is being sent over BD to the
   cache. The time between when FEHOLD+ becomes
   active to when the first MDATAV+ is sent depends on
   what the MC is doing when GHOLD+ is asserted.
2. The second MDATAV+ comes 2 beats after the first MDATAV+
   if there is no ECC. If an ECC exists, the second MDATAV+
   will follow the first by 4 beats.
A. Load RMA and cache address registers with '1004.
B. Load RCDE with Opcode (instruction i+3).
C. Load RCDO with Displacement (instruction i+3).
D. Load ERMA with '1004.
E. Load RMA and cache address registers with '510.
   BVMA selects point to IRP.
F. Load RCDE with xx (from location '1004).
   Cache miss so xx is bad data.
G. Load RCDO with xx (from location '1005).
   Cache miss so xx is bad data
H. Load PRMAL with '510.
I. Load cache address registers with '1004.
   BVMA selects point to ERMA.
   This address is sent to the MC over BB (after virtual
   to physical translation).
3. Write '40 to location '1004, '50 to location '1005
   in cache data RAMs.
J. Load cache address registers with '1006.
   BVMA selects point to EA, bit 15 is inverted.
K. Load RCDE with '40 (from address '1004).
L. Load RCDO with '50 (from address '1005).
4. Write '60 to location '1006, '70 to location '1007.
M. Load RMA and cache address registers with '510.
   BVMA selects point to PRMAL.
Q. Load RMA and cache address registers with '510.
R. Load RCDE with Opcode of instruction i+4.
S. Load RCDO with Displacement of instruction i+4.
T. Load ERMA with '510.
```

23.7 Error Detection and Reporting

On every cache access, read or store, parity is checked for the cache cell being used. If a parity error is detected the signal FISSOFTPE+ becomes active and causes a fetch-cycle trap on the E unit. A cache miss is also taken, and BOTH elements of the selected cache set are updated with the correct data. The reason both elements are updated is to make the control for fatal parity error detection more simple.

If both caches have a parity error after the update from memory, a fatal error is detected and a machine check is taken. If no fatal parity error is detected, there is a recoverable parity error. A recoverable parity error means that either a transient parity error occurred or there is a hard parity error in only one of the elements. The microcode must determine whether the error was a transient error or whether a cache cell is permanently damaged.

This error detection can be done in various ways requiring varying amounts of hardware. The 4150 implements a microcode algorithm that over a period of time executes a run time diagnostic. The diagnostic program would be initiated by a hardware timer that is generally used for maintaining wall clock time on the CPU. When invoked, the diagnostic wakes up and begins a cache test that searches for hard cache errors. After testing a portion of the cache, the diagnostic returns control to the CPU and waits until the next interrupt. When and if a hard failure is detected, the appropriate cache cell is forced invalid by setting its FRCINVLD bit. The diagnostic is also responsible for reporting the SOFT and HARD errors to the operating system's error log. The main reason this approach was taken was so that the hardware would not need to remember the address of the error.

23.8 VLSI Usage

The cache data registers are located on the PCSS chips. A part of the cache miss detection is done on the PCSS chips.

The cache address registers are in the PCADR chips.

23.9 9755 Comparisons

The following enhancements have been made to the cache organization:

- 2 way, set associative organization
- Recovery from single bit soft errors
- Recovery from single bit hard parity errors
- Fatal error detection
- Map out facility of single bit hard errors

23.10 Critical Paths

1 beat paths (62.5 nsec)

Reading data from cache and loading it into RCD registers in one beat. TCADR+ -> FMAxx -> cache RAM access -> setup at RCD registers. Time = 42.5 ns

Writing data to cache RAM during store operations. Data must be brought from E unit and written into cache RAMs in one beat. SELBDI- -> BDxx -> BDxx+IS -> BCDxx -> setup on cache data RAM. Time = 53.0 ns

23.11 Partitioning

All of the cache functionality is implemented on the IS board, which is discussed in chapter 31.

24. Storage Management Unit Detailed Description

24.1 Cache and STLB Addressing

There are a multitude of ways to address cache. Bus Virtual Memory Address (BVMA) is the source for the Cache and STLB address registers. The Cache ADdRess (PCADR) VLSI chip can be viewed as a 9:1 mux that selects the proper source for BVMA. External control is used to select one of the following BVMA sources:

- IRP - Incremented program counter.
- BD - Bus 'D' data.
- EA - Effective address data.
- EAS - Microcode pointer (source).
- EAD - Microcode pointer (destination).
- EAS/EAD incremented - Pre-incremented microcode pointers.
- ERMA - Backed up RMA for effective address.
- ERMA incremented - Pre-incremented ERMA.
- PRMA - Backed up RMA for program counter.

There are inherent priorities in the encoding of the BVMA selects which are exploited in the control logic.

The Storage management (S) unit maintains a copy of the cache address in the RMA register contained in the STLB Set Select (PSSS) VLSI chip. The reason for this register is to correct a side effect of the design goal of having only one source to BB. For timing considerations the Cache Set Select (PCSS) VLSI chip is the only source to BB. In order to read the RMA value in a similar manner as in the 9755, RMA data would have to make two VLSI chip crossings before reaching BB, (PCADR to PSSS to PCSS). In addition, special bypass logic would be required to provide the quick data path, not to mention the added complexity to the control logic. As an alternative, a copy of RMA is maintained in the PSSS chip and is read by sending the PCSS chip the data over Multiplexed Bus Physical Memory Address (MBPMA) during a RMA read. Support for direct reads of EAS and EAD has not been provided for the same reason. Reads of EAS and EAD are performed by having the microcode first transfer EAS or EAD to RMA and then reading RMA.

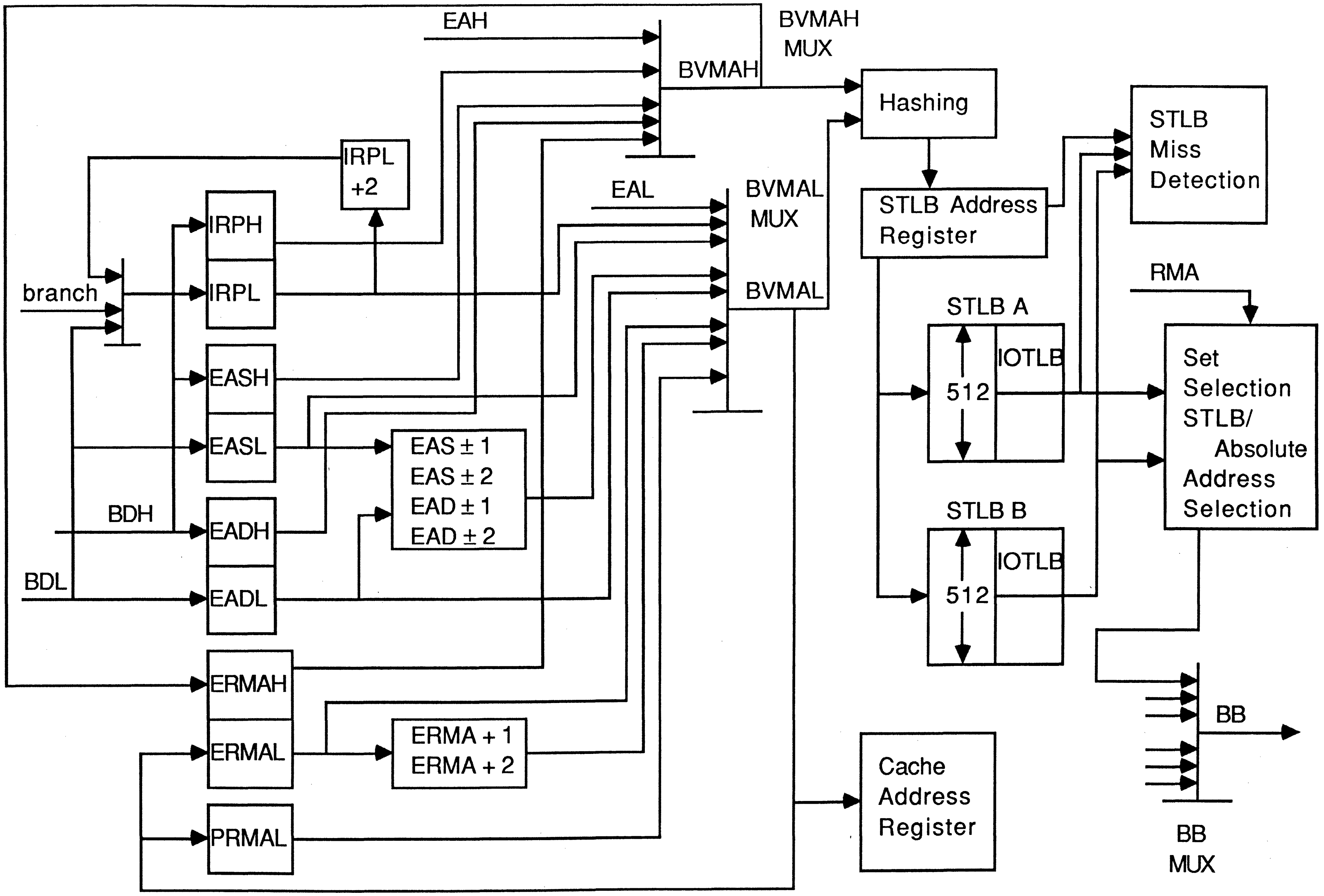


Figure 24-1 Block Diagram of Storage Management Unit

24.1.1 IRP

The IncRemented Program counter (IRP) provides a pointer to the cache location that is to be used to fetch the next instruction. IRP is selected on BVMA during odd pipeline stages unless there is an unaligned read in progress. The PCADR maintains IRP in two parts. The high side, or segment portion, is loaded from BDH and doesn't change until it's loaded again. The low side can be loaded from BDL, incremented by two, loaded from the branch cache target field, or held at its current value. The IRP value will be held at its current value under two conditions. The first occurs when a short instruction has been decoded. In this case the IRP must be held so that instruction fetching doesn't get too far ahead. The second case occurs during memory related traps. Here, IRP must be held so that fetching can be suspended until the trap condition is resolved.

The IRP selects are carefully chosen to provide priority on externally controlled select lines. Loading from BD has highest priority, followed by holding, loading from branch cache, and incrementing. The reason for this is that detecting a RP trap, program counter related trap, and holding the IRP value must take precedence over loads from the branch cache or increments in order to maintain the state of the machine during the duration of the trap.

24.1.2 EAS and EAD

To reduce pin count and the requirement for additional clocks, the PCADR chip has the ability to recycle high side data when only the low side of some registers are being modified. This scheme was used for EAS, EAD, and ERMA.

The EAS and EAD logic provides a shared incrementer/decrementer to manipulate these pointers under microcode control. The EAE microcode field dictates which hardware register is to be incremented or decremented, and by what value (one or two). Since the EAE field is only 3 bits, the EAD+1 state was sacrificed to support no operation.

24.1.3 ERMA and PRMA

The ERMA and PRMA registers are used to provide back up registers for RMA in the event of a cache miss. During a cache read the cache address registers are modified before the cache miss is detected. To properly modify the correct cache location a backup of the cache address register is used. Logic external to the PCADR knows whether the cache miss was on the instruction or operand and uses the appropriate backup register, PRMA or ERMA. A special input to the PCADR allows cache address bit 15 to be inverted so that the other half of the cache block can be modified.

24.1.4 Feedback Paths

Due to pin restrictions on the PCADR part the 4150 does not have separate clocks for the high sides of ERMA, EAS, and EAD registers. For example, an EAS or EASL destination will both create a TEAS+ clock that will go to both the EASH and EASL registers. To compensate for the lack of separate clocks, each of these registers have a feedback path. If this path is selected, the register is clocked with the data currently in that register, leaving it unchanged.

When either ERMAL, EASL, or EADL destination is called out, the signal SELFDBCK+ becomes active. This signal will activate the feedback paths.

Note for RMAL destinations (e.g. 16 bit indirects), ERMAH is clocked into RMAH.

24.2 STLB Organization

The Segmentation Table Lookaside Buffer (STLB) provides the most recent virtual-to-physical address translations. The STLB organization is much like the cache organization. There are 1024 STLB entries, organized as 512 sets with 2 elements per set. The address to the STLB is hashed in a manner similar to the 9755.

The format for each STLB entry is shown in Table 24-1.

TABLE 24-1. STLB Entry Format

BPMA01 - BPMA16	: upper 16 bits of physical address
BPMARP	: parity bit low byte
BPMALP	: parity bit high byte
BVMA05 - BVMA16	: segment number of each entry
PID01 - PID10	: process ID bits for entry
PIDP	: parity bit for PID field
ARR1X,W,R	: ring 1 access bits (time multiplexed with MBIO bits during DMX)
ARR3X,W,R	: ring 3 access bits (time multiplexed with MBIO bits during DMX)
PCNT01 - PCNT03	: purge count
STLBVLD	: valid entry bit
PMOD	: page modified bit
SHARED	: shared bit
BFRCS	: Hard parity bit
BUPDTS	: Update bit

The IOTLB contains 256 entries to support 4 segments of I/O windows. The IOTLB entries are stored redundantly in the two STLB elements to maintain availability in the event of a parity error in the IOTLB. IOTLB misses are not supported in the hardware since the size of the IOTLB is known to software, which is responsible for ensuring that no cell is used more than once.

Like the cache, the STLB employes a distributed FIFO replacement algorithm and has force bits to map out offending memory cells.

The STLB uses two slices of the PSSS to perform the element selection and memory trap detection. Like the PCSS, each PSSS is dedicated to one STLB set, and each slice is responsible for only half of the data. The 'A' slice deals with the high bytes of the physical page address (Bus Physical Memory Address or BPMA), and the 'B' slice deals with the low bytes. Under normal operating conditions, each PSSS compares a copy of virtual address and process identification bits (PID) to the tag of its dedicated element. If they are the same, the PSSS slice puts its main data, BPMA, on the MBPMA outputs. Otherwise, the PSSS slice puts its auxiliary data out on MBPMA. There are two copies of the MBPMA bus to minimize the loading on the signals between the PSSS and PCSS VLSI chips. The data on the two MBPMA busses are identical during normal operation, but differ during RMA and diagnostic reads. There is an external clocked copy of MBPMA that is used for providing the cache index RAMs with the physical page address during cache updates.

The result of the comparison is also presented to the outputs so that external logic can determine whether a memory trap has occurred. A memory trap is detected when both PSSS options indicate that a memory trap has occurred or when a parity error is detected. The PSSS VLSI chip provides information about what type of memory trap has occurred. There are separate output signals for access violation, page modified trap, and address trap. In the case of access violation or page modified trap, the outputs will only be activated if the tag comparison is true. Unfortunately, this rule is was not always followed, and if a FORCE bit is set, the access violation and page modified trap signals may be active even if the tag comparison is false. Because of this the microcode trap handlers for access violation and page modified must be cognizant of the fact that an erroneous page modified trap or access violation may have occurred, and check for a STLB miss before proceeding.

A quick purge mechanism is used to reduce some of the overhead associated with purges of the STLB. Included in each element of the STLB is a three bit purge count field. A three bit register is loaded whenever the PID register is loaded, and whenever the STLB is updated the purge count field is updated with the register contents. During a STLB read, the purge count bits are compared to the copy of the current purge count bits internal to the PSSS. The purge count bits can be thought of as three additional valid bits. Whenever a PTLB instruction is encountered, the microcode will increment a register that is used to load this field. By incrementing the purge count register, subsequent STLB references will indicate a mismatch. This quick purge mechanism allows the STLB to be quick purged seven times before a complete purge is required.

24.2.1 Shared Bit

The architecture provides the notion of a 'shared' bit. Shared here refers to a multiple number of virtual addresses sharing a single physical address. When two virtual addresses share the same physical address a potential cache consistency problem can occur. If both references don't share the same cache location, modifications through one cache leaf will not be reflected in the other cache leaves. Cache leaves are sections of the cache which correspond to physical pages in memory. Each cache set on the 4150 has 32 cache leaves.

The shared bit solves this problem by forcing references to a page with the shared bit set to take a cache miss. The shared bit is included in the page map table of the memory management data structures. This bit is loaded during STLB updates and is queried on every cache reference. After STLB data selection is performed, if the shared bit of the selected element is set, the subsequent cache reference will take a cache miss regardless of whether or not there is a tag match.

It should be noted that there is a subtle difference between the shared bit implementation on the 4150 and the 9755. On the 9755, the shared bit causes the cache to be loaded with the invalid bit set, while on the 4150 the shared bit just causes the cache miss to occur. The reason for this difference lies in one of the goals of the cache organizations. To maintain high availability, it is desirable to keep the cache valid during references. If the cache cell were to be invalidated, then in the event of a parity error, the data selection mechanism would be defeated and memory data would effectively be restricted to one set.

In addition, the 9755 required special attention when the software wanted to change the state of the shared bit. In the 9755, when the shared bit was first set the software had to guarantee that the data wasn't already resident in cache. If this were the case, since the shared bit only invalidated the cache on cache misses, a cache miss was required before the shared bit performed as the architecture dictates. In the 4150, once the shared bit is set, cache misses occur immediately. When the shared bit is released, the cache reverts back to normal operation.

24.2.2 IOTLB Support

Special support is given for cache addressing during I/O operations. Under IAC control, BDH{05:14}+ will be zeroed before RMA is loaded to help the microcode generate addresses in a certain range. This is necessary because I/O transfers are only allowed in the first four virtual memory segments.

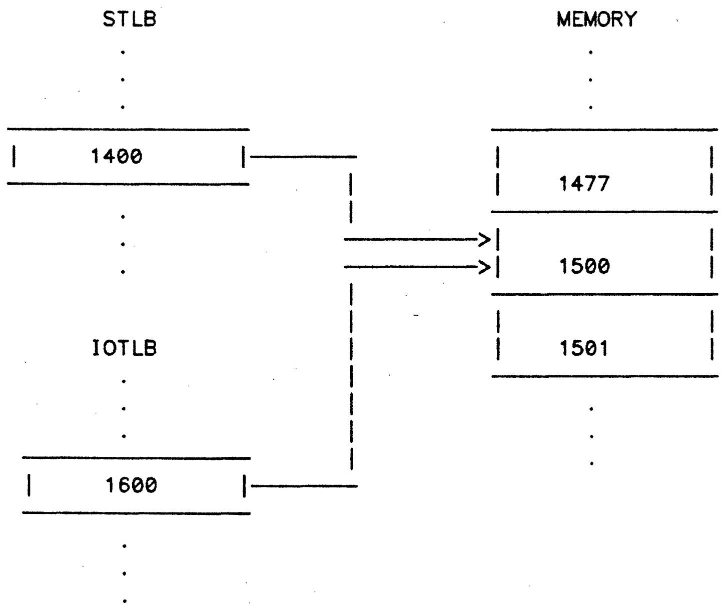
24.2.2.1 Cache Invalidation During DMx Input Operations

Whenever a Direct Memory transfer (DMx) to main memory is performed it is necessary to invalidate the cache location corresponding to the memory address where the input data is destined. The necessity for this is best explained by considering the alternative. Suppose there is some valid data in cache location 100 which corresponds to the same valid data in memory location 100. Further suppose that a DMx input transfer occurs to memory address 100. The data is transferred directly from the I/O controller to main memory. (Details of this are best left for chapter 27, which discusses the I/O interface.) The contents of location 100 in memory have been altered, but not the contents of cache location 100. Furthermore, the cache can't be updated because the data never appeared on BD. Continuing with the scenario, sometime later the CPU asks for the contents of location 100. The cache dutifully sends it the contents of cache location 100, the stale data. If the cache was invalidated during the DMx operation, the cache would initiate a cache miss sequence on the subsequent read request, fetching the proper data from main memory.

The cache is invalidated during DMx operations by the use of the IAC DMX. Whenever this IAC is used during a microinstruction which has a memory destination, the cache location pointed to by RMA is invalidated.

Unfortunately, that is not the end of the story. A further complication arises because the virtual to physical address translations for data accesses are accomplished via the STLB, while the same translations for DMx transfers are done via the IOTLB. Different virtual addresses may map to the same physical memory address. Figure 24-2 illustrates this. The STLB entry for virtual address 1400 and the IOTLB entry for virtual address 1600 both map to physical address 1500. Substituting address 1600 for address 100 in the scenario discussed earlier leads to an error. Cache location 1600 would be invalidated during the DMx operation, but when the CPU subsequently requested the data at address 1400, that data would still appear valid in the cache.

FIG. 24-2. STLB and IOTLB Mapping to the Same Location



The IOTLB is equipped with extra bits to help avoid this problem. Whenever a DMx input transfer is performed, the microcode instructs the S unit to get these Mapped Bus I/O (MBIO) bits from the IOTLB and merge them into RMA. On the PCADR chip ERMAL{02:06}+ are multiplexed with the MBIO bits. If a mapped I/O operation is performed, the mux selects point to the MBIO bits. Therefore an RMA>MAL operation (placing ERMA onto BVMA) will enable the MBIO bits to control what cache leaf is written to. This is done by issuing IAC DMX and destination [RMA>MAL] in the same microinstruction. In the example shown in Figure 24-2, this would change RMA from 1600 to 1400, and the correct cache location would be invalidated.

24.3 Traps

The following subsections describe S unit traps. The traps are listed in order of priority, from highest to lowest.

24.3.1 Read Address Trap

If a memory reference instruction forms an EA between '0 - '7 (V mode) or '0 - '37 (S and R modes), the addressed location is in the current register file, not memory. When such an address is calculated, this trap aborts the memory read and loads a cache entry with the contents of the addressed register. The cache is marked invalid and cache misses are disabled during the microstep retry. A cache miss occurs on the next reference to this cache entry.

24.3.2 STLB Miss

An STLB miss trap is taken if any of the following conditions occur:

1. The virtual address translation is not located in the STLB. This is detected by comparing the current virtual address' segment and page bits to those of the virtual address which caused the STLB to be written. Each STLB entry specifies the segment number of the virtual address used when the STLB entry was written.

2. The STLB contains invalid data. This is detected by checking the STLB entry's valid bit. If it contains a 1, the entry is valid; if it contains a 0, the entry is invalid.

3. The PID in the STLB entry must be identical to that of the process making the reference IF the segment number specified is in private address space. The PID is ONLY checked if the segment specified in the virtual address is greater than or equal to '4000 (private user address space).

The STLB miss trap microcode checks if the virtual page containing the information is currently in main memory. If it is, the microcode translates the virtual address to a physical one, and then puts the translation into the STLB. The reference is retried after the translation is loaded into STLB.

If the page is not in physical memory a page fault occurs. A fault occurs when the software tries to perform an action that cannot complete without special help. A software routine called the page fault handler finds the virtual page on the disk and moves it into main memory. The virtual-to-physical translation is loaded in the STLB and the reference is retried.

24.3.3 Access Violation

An access violation occurs if a procedure tries to reference a memory location for which it has insufficient access rights. This trap causes an access violation fault.

The access rights of a procedure referencing a memory location are stored in that memory location's STLB entry. Different access rights are given depending on whether the procedure making the memory reference is in Ring 1 or Ring 3. Procedures in Ring 0 have all access to any procedure that they reference.

The hardware checks the access rights of a particular memory reference by first isolating the ring number of the procedure making the reference. The referenced memory page's access rights for a procedure in that particular ring are then checked against the operation specified by the procedure making the memory reference.

For example, if the instruction specifies a read operation and the selected access field allows reads, then the read operation is valid. If, however, the instruction specifies a write and the access field allows only reads, then the operation is invalid and an access violation trap occurs.

Access rights are assigned on page boundaries. When a new page is loaded into cache its access rights are loaded into the STLB from the particular procedure's segment descriptor table. (See System Architecture Guide for more details.)

24.3.4 Page Modified Trap

This trap occurs during each step that writes into a physical page that is being modified for the first time. This is determined by examining the state of the STLB entry's page modified bit. If the bit is a 1, the page is being modified for the first time since this STLB entry was loaded. The trap routine loads a new translation into the STLB entry and resets the page modified bit.

24.3.5 Write Address Trap

This trap occurs during write operations specifying an address within the range '0 - '7 (V mode) or '0 - '37 (S and R modes). This trap aborts the write to memory and directs the write to the appropriate register file location.

24.3.6 Flat Trap

Flat trap is caused by a 32-bit cache read that crosses the segment boundary when the flat address space is being used (UNIX). It shares trap a vector with one of the address traps. The trap microcode determines which trap has occurred by checking if the machine is in I mode (address traps can not happen in I mode). Then it reads the first word from the next segment and writes it into cache, marking it invalid. The original address is reloaded into RMA, and cache misses are disabled for the retry. This operation is just like an address trap, except that the data is read from memory instead of the register file.

24.3.7 Wrap Trap

Wrap trap is caused by a carry or borrow out of the low side when incrementing or decrementing EAS or EAD . It shares a trap vector with one of the address traps. The trap microcode determines the type of trap by checking if the machine is in I mode. Then it jumps on condition EASOP to determine if it was an EAS or EAD operation that caused the trap. (Jump condition EASOP is high if EAS has been modified after EAD was last modified.) Next it determines if it was an increment or a decrement by looking at the low side of the modified register. If it is positive, the operation which caused the trap was a decrement. The high side of the modified register is incremented or decremented accordingly.

24.4 VLSI Usage

Four VLSI chips of two types are used in the S unit.

- 2 STLB Set Select (PSSS) chips, which perform STLB miss detection, STLB set selection, and memory trap generation. The PSSS chips also contain a copy of RMA. This feature is used to support direct microcode reads of RMA and generation of memory addresses during cache misses.

- 2 Cache ADdRess (PCADR) chips, which implement IRP, EAS, EAD, the ERMA and PRMA registers, ERMAL increment logic, EAS and EAD increment/decrement logic, and the BVMA mux.

24.5 Critical Paths

1. From memory trap detection at CS2+ to holding off incrementing of IRP at CS2.5+. 1/2 beat path.

2. Path starts at TRCDIE+ (at CS2+), generating a memory trap and then holding off IRPL load at CS2.5. This is a 1/2 beat path.

3. FMTRPA+ -> GFMTRAP+ -> holding off IRPL clock at CS2.5. 1/2 beat path. Time = 30.0 ns

4. From clocking IRPL at CS2.5+ to loading cache and STLB address registers 1/2 beat later at CS1+. 1/2 beat path.

5. TIRPL+ -> BVMA -> setup on the cache address registers. 1/2 beat path. Time = 25.5 ns

6. Clocking cache address register with ERMAL+1 one beat after CS7+ during unaligned reads. 1 beat path.

7. TRCML+ -> RCMBB02+ -> UNALRD+ -> MASEL3A+ -> MASEL3+ -> BVMAxx -> cache address register setup. 1 beat path. Time = 61.5 ns

8. From clocking cache address register at CS1+ (or TRCML+) to reading physical address (physical page number) from STLB and loading into registers on PCSS part at CS2+ (or CS7+). 1 beat.

9. TSADR+ -> FMAxx -> STLB RAMs -> MPBMAxx -> setup on index registers on PCSS. 1 beat path. Time = 55.0 ns

10. From MPMA mux control generation at TRCML+ to selecting proper data from PSSS chips to clocking that data on PCSS part at CS7+. 1 beat.

11. TRMCL+ -> RCMBBxx -> PMASELxx -> MBPMAxx -> setup on index registers on PCSS. 1 beat path. Time = 57.0 ns

24.6 9755 Comparisons

The 4150 does not support direct microcode reads of EAS and EAD registers as the 9755 did. Reads of EAS and EAD are performed by having the microcode first transfer EAS or EAD to RMA and then reading RMA.

The 4150 does not support the operation EAD+1>MA.

The 4150 STLB entries contain extra control bits which the 9755 didn't implement. There are 3 bits for the purge count and 2 bits to help control the 2 set STLB.

The 4150 STLB has 1024 total entries organized as 512 sets with 2 elements each. The 9755 has 1 STLB with 512 entries.

The shared bit in an STLB entry no longer requires that the data be invalidated in the cache before the bit is set.

The 4150 includes UNIX support hardware.

24.7 Partitioning

All S unit functionality is implemented on the IS board, which is discussed in chapter 31.

25. Execution Unit Detailed Description

The main function of the Execution (E) unit is to read one or more of the data sources available, manipulate the data in the ALUs, transport the resulting data through the barrel shifter logic, and load the data into the necessary register file locations. The E unit's essential sections are:

- ALU
- Barrel Shifter
- Register File

The first few sections of this chapter will describe these parts of the E unit without trying to explain how they all work together. That information appears in the latter sections of this chapter.

The general data flow through the E unit is illustrated in Figure 25-2.

The E unit also has exception detection logic which can cause processor traps. The system timers are also part of the E unit.

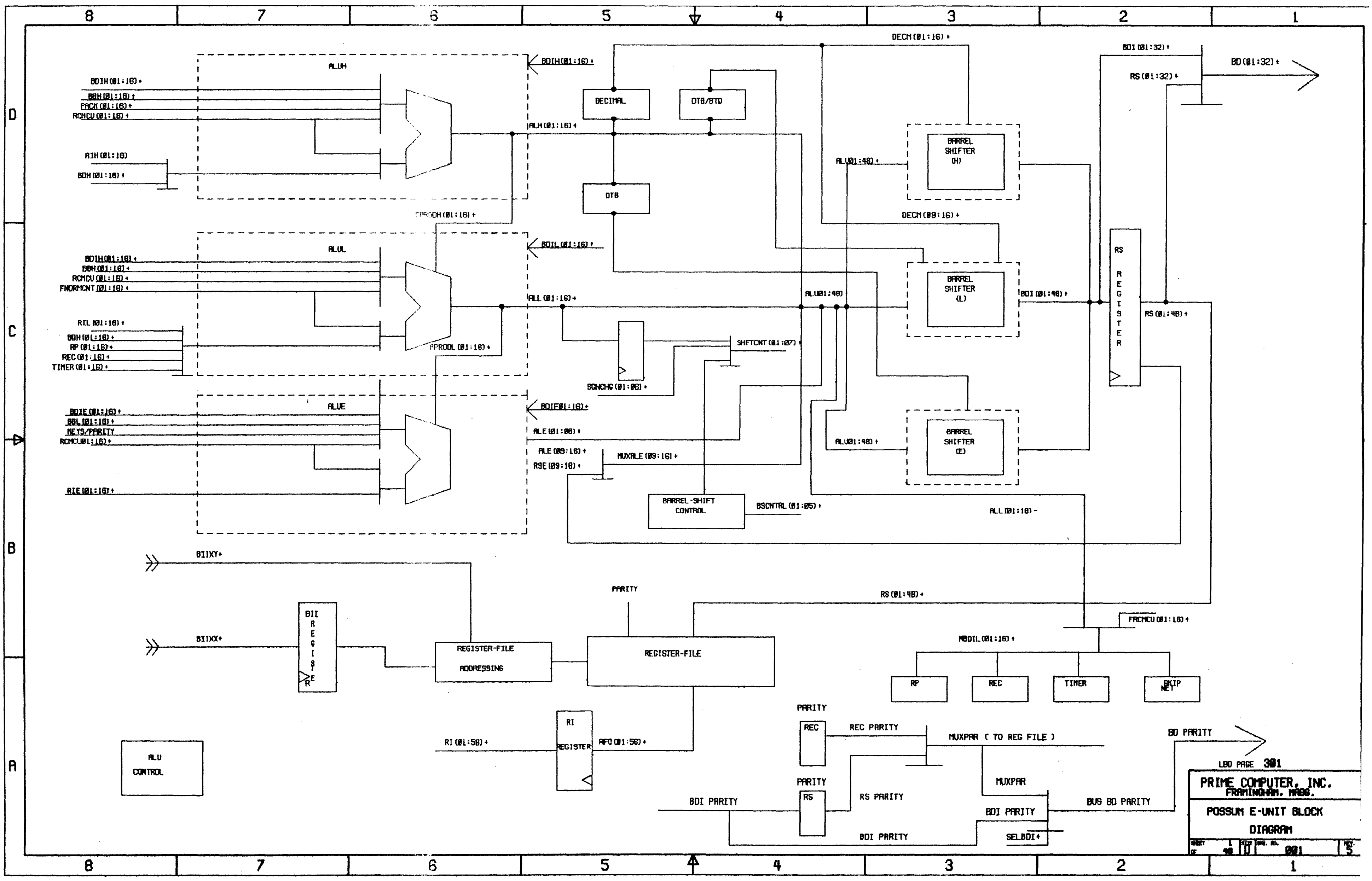


Figure 25-1 Block Diagram of Execution Unit

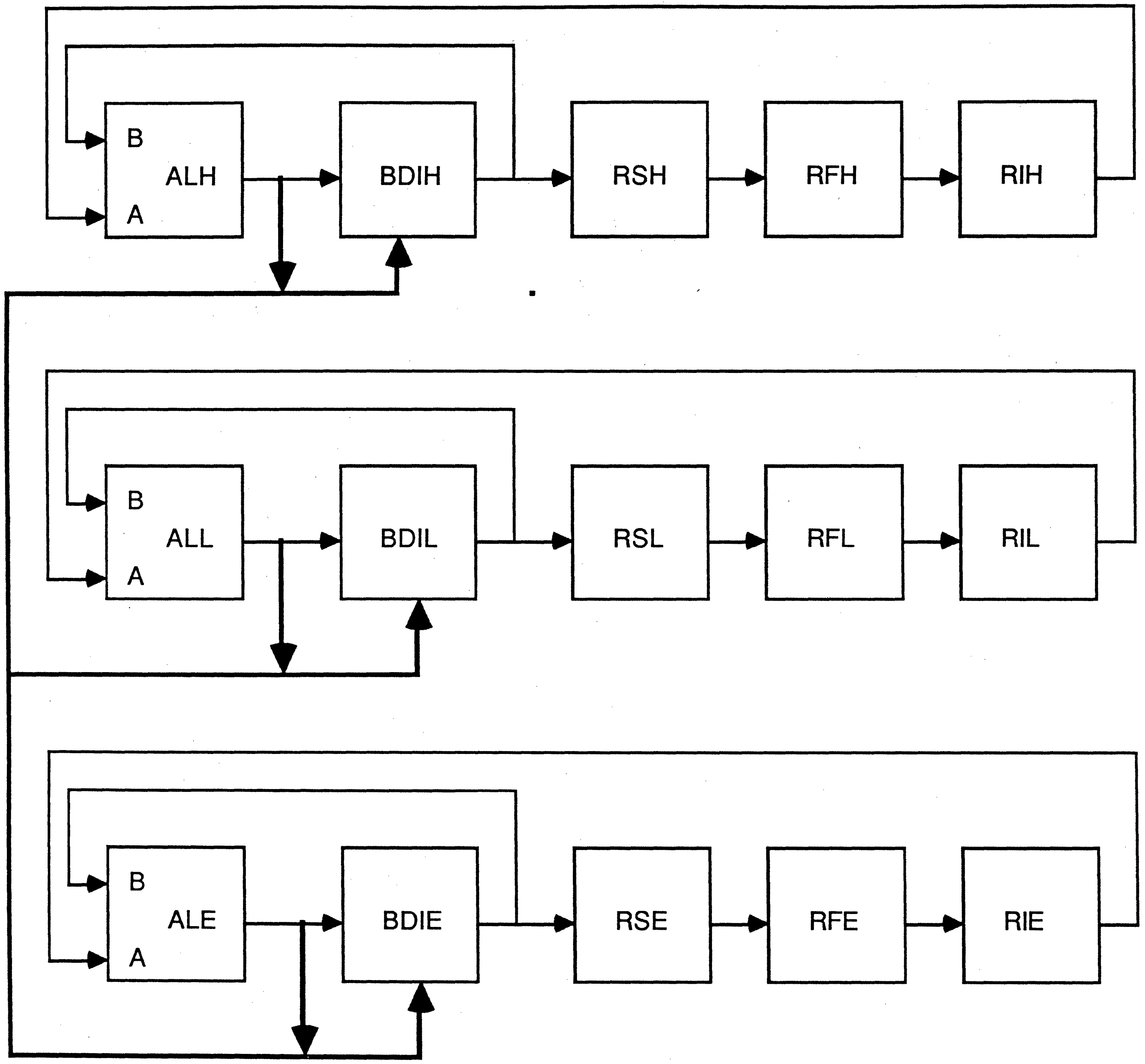


Figure 25-2 Execution Unit Data Flow Diagram

25.1 ALU Logic

The ALU is implemented using Execution Arithmetic Logic Unit (PEALU) VLSI chips. Each PEALU slice is 8-bits wide and has its own unique select inputs. Seven 8-bit slices are needed to create a 56-bit ALU. They are divided as follows:

- ALH - The ALH slices are called ALH1 and ALH2.
- ALL - The ALL slices are called ALL1 and ALL2.
- ALE - The ALE slices are called ALE1 and ALE2.
- ALX - The ALX slice is called ALX.

The ALX slice is only used during floating point operations.

25.1.1 ALU Modes

The main data path consists of the ALU manipulating data from two sources, the ALEG and the BLEG, each of which are fed by a number of sources. The sources for busses A and B are shown in Table 25-1.

The manipulation of data occurs in two stages, first through the ALU and then through the barrel shifter (BDI). This is accomplished by a combination of both ALU and BDI modes.

The ALU is capable of operating in a variety of different logical and arithmetic modes. There are 32 possible operating modes. These modes are selected by the microcode word ALU field. Most ALU modes are for byte (8-bit) wide operations, while some are for nibble (4-bit) wide operations. The nibble operations are denoted by a colon which separates the high and low nibble functions respectively. These modes are shown in Table 25-2.

The ALU selects are produced independently for each 16-bit ALU section (H, L, E). Each section can therefore act independently of or in tandem with its neighbors. The select signals are clocked internally in the ALU VLSI chips at CS7+.

There is one other ALU select input called DIV+ which is used to support a single bit (per beat) non-restoring divide algorithm. This select is used to switch the ALU modes between ADD and SUBA during divides based on the sign bit of the previous iteration. This select is enabled if and only if the other five ALU selects are in divide mode (ALU selects = '00).

TABLE 25-1. ALU Data Sources

BUS	SOURCE	COMMENT
A(H)	RIH{1:16}	Register file high side
	BDH{1:16}	Bus D high side
B(H)	BDIH{1:16}	Bus D internal high side
	BBH{1:16}	Bus B high side
	PACK{1:16}	Packer PROM output
	RCMCU{1:16}	Microcode word emit field
	RDH{1:16}	Register Data high side
A(L)	RIL{1:16}	Register file low side
	BDL{1:16}	Bus D low side
	RPREC{1:16}	RP or REC
B(L)	BDIL{1:16}	Bus D internal low side
	BBL{1:16}	Bus B low side
	RCMCU{1:16}	Microcode word emit field
	FNRMCNT{1:16}	Normalize Count
	RDL{1:16}	Register Data low side
A(E)	RIE{1:16}	Register file extended
B(E)	BDIE{1:16}	Bus D internal extended
	BBL{1:16}	Bus B low side data
	KEYS/PARITY {1:16}	Keys register and parity information data
	RCMCU{1:16}	Microcode word emit field
	RDE{1:16}	Register Data extended
A(X)	RI{0:7}	ZEROS
	EMIT{0:7}	FRNDBIT{01:03} Round bits || ZEROS
B(X)	JUNK{0:7}	FGRDBIT{01:03} Guard bits
	BB{0:7}	ZEROS
	EMIT{0:7}	FRNDBIT{01:03} Round bits || ZEROS

TABLE 25-2. ALU Modes

ALU SELECTS (octal)	ALU MODE
00	DIVIDE(SUBA)
01	INCA
02	ADD
03	SUBB:DECA
04	SUBA
05	ZERO:ADD
06	Undefined Mode
07	SUBB
10	NOTA
11	INCA
12	INCA:ADD
13	DECA
14	DECB
15	ZERO:INCB
16	INCB
17	NOTB
20	OR
21	TA:ZERO
22	AND
23	OR:TA
24	XOR
25	ZERO:TB
26	A.AND./B
27	/A.AND.B
30	TA
31	TA:TB
32	TB:TA
33	ZERO:TA
34	TB
35	ZERO
36	TB:ZERO
37	ONES

25.1.2 Carry

The ALUs provide full look-ahead carry across all 56 bits. This is accomplished by the carry select inputs. ALH and ALL both have two carry selects, ALE has only one carry select, and ALX has it's carry selects always tied low. These carry selects control the ALUs' Group Generates (GG) and Group Propagates (GP) for 16, 32, 48, or 56-bit add operations. ALE slices have only one carry select for either 48 or 56-bit operation, while ALX has none since it is only used in a 56-bit operation. ALH and ALL need two carry selects, since each may be involved in 16, 32, 48, or 56 bit operations.

25.1.3 ALU Output

The ALUs drive both positive and negative logic versions of their outputs. Along with current operation data the ALUs also provide result information. (In the following list, as well as throughout the remainder of the chapter, "x" can be "H", "L", or "E" to represent the ALU section from which it is coming.) This information consist of the three signals:

- ALx00- (external sign of result)
- ALxCO{1:2}+ (carry out on both byte and nibble boundaries)
- ALxEQ+ (zero detect)

25.1.4 BA Mux

The Bus A (BA) mux selects one of four possible sources which becomes the A leg input to the ALU. The mux has two selects, BAxSEL{2:3}+. These signals are generated by decoding the microcode ALU field, which is clocked at CS7+. The four possible sources are:

RI	Register file output data
RS	Register Stage BDI data clocked at CS9+ internal to the ALU
EMIT	microcode CU field bits
RCRY	Register CarRY bits from the carry save adder clocked by TRSUM+

25.1.5 IBB Mux

The Internal Bus B (IBB) mux selects one of five possible sources which becomes the B leg input to the ALU. The mux has three selects, IBBxSEL{1:3}+. These signals are generated by the decoded BB microcode field, which is clocked at CS7+. The five possible sources are:

EMIT	microcode CU field bits
RD	Register Data internal to the ALU clocked by TRDx+
RSUM	Register SUM bits from the carry save adder clocked by TRSUM+
BBH	main system Bus B High side data, latched on the rising edge of BBLATCH+. The latch enable signal opens the latch at CS7+ and closes it sometime before CS8+.
JUNK	data which is ALU slice dependent.

- ALH - pack PROM data
- ALL - microcode CU field bits clocked at CS7+
- ALE - condition code and parity information

- ALX - guard bits

25.1.6 RD and RS

The ALUs contain two special registers called Register Stage (RS) and Register Data (RD).

RS gets loaded with BDI output data clocked at CS9+, and is placed on the ALUs' A leg via BD. When a write to the register file is followed by a read from the same location, a copy of RS internal to the ALU is read instead of the actual register file. This is known as a register file bypass operation. These operations are necessary because of the pipelined nature of the machine. The data being produced by the current microinstruction is saved at the end of stage 9, and the next calculation starts at the beginning of the next stage 8. These times are identical; the end of stage 9 marks the beginning of the next stage 8. The register file is not written with the results of the current microinstruction until the end of stage 10 (the end of the next stage 8). Thus, the data being accessed from the register file at the beginning of stage 8 doesn't get into the register file until the end of stage 8.

There are actually three RS registers in the E unit, the one just described, which is internal to the PEALU slices, and two external. One of these is used to drive BD, while the other is used to drive the register file input.

RD gets loaded with either BDI output data or ALU output data depending on the BB select input, and is then placed on the ALUs B leg. RD is used for divide quotient collection in addition to its duties as a microcode destination, and that of acting as the multiplicand register.

25.2 Barrel Shifter Logic

The barrel shifter is implemented with three Bus D Internal (PBDI) VLSI chips. Each slice is 16 bits wide, creating a 48-bit data path. They are divided as follows:

- BDIH - Bus D Internal High side data
- BDIL - Bus D Internal Low side data
- BDIE - Bus D Internal Extended side data

Note that because the barrel shifter is capable of 48-bit shifts, all 48 bits of data from the ALUs are needed by each slice in order to generate its 16 bits of output. Personality pins tell the slices which part of the data path they belong to (High (H), Low (L), or Extended (E) sides).

25.2.1 BDI Encoding

The barrel shifters are controlled by the decoded microcode BDL field. The field is decoded with support hardware external to the VLSIs to provide each chip with five control lines BDIxSEL{1:5}+. The BDI functions are shown in Table 25-3.

TABLE 25-3. BDI Modes

BDISEL1+	BDISEL2+	BDISEL3+	BDISEL4+	BDISEL5+	RESULT
0	0	0	0	0	Rotate Left 32
0	0	0	0	1	Shift Left 32
0	0	0	1	0	Rotate Right 32
0	0	0	1	1	Shift Right 32
0	0	1	0	0	Reverse Mask Adjust
0	0	1	0	1	Adjust E Side
0	0	1	1	0	Adjust L Side
0	0	1	1	1	Adjust H Side
0	1	0	0	0	Reverse Mask Shift Left 48
0	1	0	0	1	Shift Left 48
0	1	0	1	0	Normalize
0	1	0	1	1	Shift Right 48
0	1	1	0	0	Rotate Left 16
0	1	1	0	1	Shift Left 16
0	1	1	1	0	Rotate Right 16
0	1	1	1	1	Shift Right 16
1	0	0	X	X	ZERO
1	0	1	0	0	Undefined
1	0	1	0	1	Transport E Side
1	0	1	1	0	Transport L Side
1	0	1	1	1	Transport H Side
1	1	0	X	X	Unpack
1	1	1	X	X	Multiply

E slice only operates in 48 bit mode for shifts and rotates.
E slice does not operate in Unpack mode.

The DECIMAL{01:16}+ and UNPACK{01:08}+ BDI inputs are used for transporting decimal data on all sides except the L side. The Register Event Counter (REC) is the input on the L side decimal data path. This is because in floating multiply algorithms, the step doing the last 48 by 8 multiply terminates by writing REC to the register file, and this is accomplished by inverting the mode of the barrel shifter from multiply to decimal transport mid-step.

25.2.2 Shift

The barrel shifter is capable of shifting 48 bits of data. The amount to shift during any operation is determined by six count bits called BDICNT{01:06}+. These count bits are the outputs of muxes which are controlled by the BDL microcode field.

25.2.3 BDI Output

The barrel shifter outputs 48 bits of data to three destinations:

- RS - needed for register file input data, clocked at CS9+
- BD - data is sent out onto Bus D
- ALUs - internal RS clocked at CS9+

25.3 Register File

The Register File ADdRess (PRFADR) VLSI generates the address for the register file read/write phases. A TRCML+ clock is used to distinguish between the read and write phases. Besides the main purpose of address generation, the PRFADR VLSI contains other information such as:

- Register File Bypass
- Register File Tracking
- Register Program counter (RP)
- Register Event Counter (REC)

Register file source and destination address information needed by the PRFADR is contained in the RFS and RFD microcode fields. The register file itself is implemented with 1K x 4-bit static RAMs. The register file is divided into three sections:

- Register File High (RFH)
- Register File Low (RFL)
- Register File Extended (RFE)

There are 256 register file locations in both RFH and RFL, while RFE has only 128 locations. One RFE locations is accessible through either of two RFH or RFL locations, as shown in Figure 25-3.

The register file is read during stage 7 and written during stage 10 as shown by Figure 25-4. Data read from the register file is placed into Register Input (RI) before being read into the ALU.

FIG. 25-3. Register File Location Layout

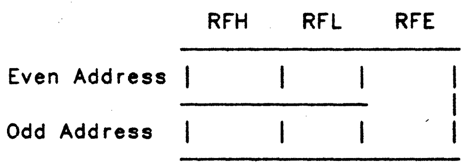
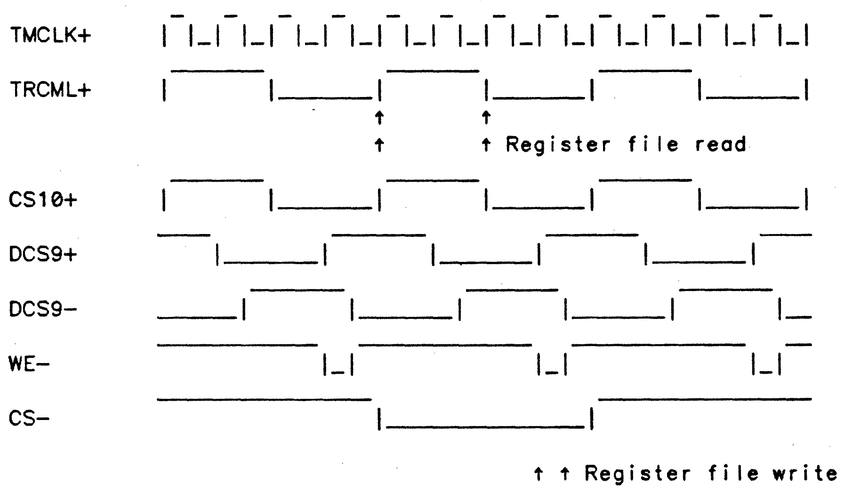


FIG. 25-4. Register File Read/Write Timing



25.3.1 Register File Bypass

The destination and source register file addresses are compared in the PRFADR VLSI chip at TRCML+. If a match is detected, HITON7+ and HITON8+ are generated, and the data for the source register can be found in RS. The signal HITON7+ goes active when bits 1-7 of the address match, while HITON8+ goes active when bits 1-8 match. HITON8+ is used for RFH and RFL tracking, while HITON7+ is used for RFE tracking. (Recall, RFE has half as many locations as RFH or RFL.)

25.3.2 Register File Tracking

The four LSBs of the destination register are used by the EAF unit for register file tracking. This information is only valid when a resident user register has been accessed.

25.3.3 RP/REC

The RP and REC registers are located inside the PRFADR slice. REC is a 16-bit register, while RP is a 15-bit register with bit 16 maintained external to the chip. Both of these registers are loaded from bus MBDIL{01:16}+, and only one of the registers can be read out of the slice at a time. (Bus MBDIL is a bus which is multiplexed between the L ALU's output and the microcode EMIT field.) Both registers share the increment and decrement logic inside the slice. REC is always in load mode during stage 9, and in INC/DEC mode during stage 8. RP is always loaded or incremented by TRCML+. In cases of error reporting or other conditions where the I unit's RP is inaccurate (because it is prefetching future instructions), the E unit's RP comes to the rescue and gets execution back on track. REC is also used as an exponent ALU during multiply. The default modes are RP in increment, and REC in load.

25.4 Multiply

25.4.1 Booth's Algorithm

The 4150 multiply is a 3-bit modified Booth's algorithm designed to operate on 8 bits of multiplier per beat. Thus, a 48 x 8-bit multiply producing a 56-bit partial product can be accomplished each beat. Six consecutive 48 x 8-bit multiplies (in 6 beats) will produce a 48-bit product (discarding unneeded low order product bits).

Booth's algorithm accomplishes a multiplication by strategically adding easily producible factors of the multiplicand. The factors to add are determined by scrutinizing the multiplier. All the factors used are shifted copies of the multiplicand or the two's complement of the multiplicand. Refer to Figure 25-5.

Restricting ourselves to 8-bit signed arithmetic for the moment, we see that the multiplier is broken up into four 3-bit factors. This is done by the addition of a "hidden" bit below the least significant bit. The hidden bit is always 0. For any value of any factor we can extract the appropriate coefficient from the factor table. This coefficient is "multiplied" by a copy of the multiplicand, and the four resulting partial products are added together. In reality, no multiplication is necessary since all the coefficients are products of 2. All the partial products can be formed by shifting and/or two's complementing a copy of the multiplicand. Figures 25-6, 25-7, and 25-8 illustrate the operation of this algorithm for multipliers of 3, 118, and -2 respectively.

FIG. 25-5. Booth's Algorithm

```
                                             Hidden Bit
                                                  |
                                                  |
               1                           8      v
             ____________________________________________
            |   |   |   |   |   |   |   |   ||   |
   RS       |   |   |   |   |   |   |   |   || 0 |
            |___|___|___|___|___|___|___|___||___|

                                    |___________|     Factor 1

                              |_________|             Factor 4

                        |_________|                   Factor 16

                  |________|                          Factor 64
```

Factor 1		Factor 4		Factor 16		Factor 64	
0 0 0	0	0 0 0	0	0 0 0	0	0 0 0	0
0 0 1	1	0 0 1	4	0 0 1	16	0 0 1	64
0 1 0	1	0 1 0	4	0 1 0	16	0 1 0	64
0 1 1	2	0 1 1	8	0 1 1	32	0 1 1	128
1 0 0	-2	1 0 0	-8	1 0 0	-32	1 0 0	-128
1 0 1	-1	1 0 1	-8	1 0 1	-16	1 0 1	-64
1 1 0	-1	1 1 0	-4	1 1 0	-16	1 1 0	-64
1 1 1	0	1 1 1	0	1 1 1	0	1 1 1	0

FIG. 25-6. Booth's Algorithm: Multiplier = 3 Example

```
                                         Hidden Bit
                                             |
                                             |
           1                           8     v
          _______________________________________
          |   |   |   |   |   |   |   |   ||   |
RS        | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 || 0 |
          |___|___|___|___|___|___|___|___||___|

                                  |____________|     Factor 1

                              |__________|           Factor 4

                          |__________|               Factor 16

                      |__________|                   Factor 64
```

Factor 1		Factor 4		Factor 16		Factor 64	
0 0 0	0	0 0 0	0	0 0 0	0	0 0 0	0
0 0 1	1	0 0 1	4	0 0 1	16	0 0 1	64
0 1 0	1	0 1 0	4	0 1 0	16	0 1 0	64
0 1 1	2	0 1 1	8	0 1 1	32	0 1 1	128
1 0 0	-2	1 0 0	-8	1 0 0	-32	1 0 0	-128
1 0 1	-1	1 0 1	-8	1 0 1	-16	1 0 1	-64
1 1 0	-1	1 1 0	-4	1 1 0	-16	1 1 0	-64
1 1 1	0	1 1 1	0	1 1 1	0	1 1 1	0

```
Result = 3 x multiplicand =  -1 x multiplicand     Factor 1
                          +   4 x multiplicand     Factor 4
                          +   0 x multiplicand     Factor 16
                          +   0 x multiplicand     Factor 64
                          _____________________
                              3 x multiplicand
```

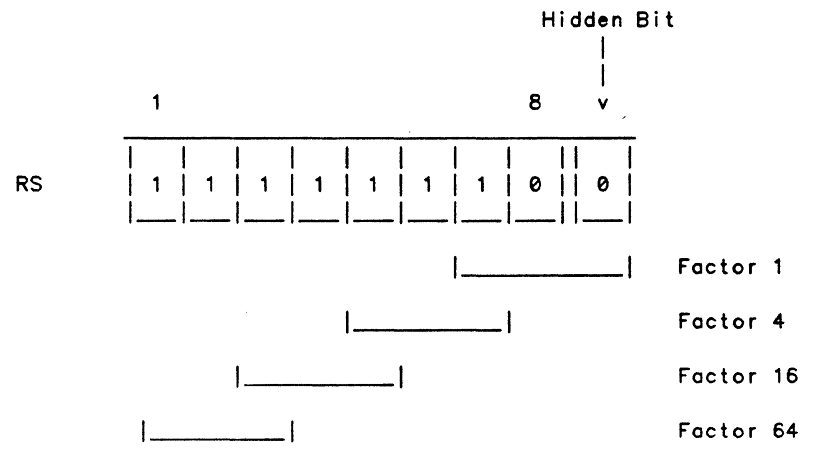
FIG. 25-7. Booth's Algorithm: Multiplier = 118 Example

```
                                    Hidden Bit
                                          |
                                          |
          1                        8      v
        _____________________________________
        |   |   |   |   |   |   |   |   ||   |
RS      | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 || 0 |
        |___|___|___|___|___|___|___|___||___|

                                |___________|      Factor 1

                        |___________|              Factor 4

                |___________|                      Factor 16

        |__________|                               Factor 64
```

Factor 1		Factor 4		Factor 16		Factor 64	
0 0 0	0	0 0 0	0	0 0 0	0	0 0 0	0
0 0 1	1	0 0 1	4	0 0 1	16	0 0 1	64
0 1 0	1	0 1 0	4	0 1 0	16	0 1 0	64
0 1 1	2	0 1 1	8	0 1 1	32	0 1 1	128
1 0 0	-2	1 0 0	-8	1 0 0	-32	1 0 0	-128
1 0 1	-1	1 0 1	-8	1 0 1	-16	1 0 1	-64
1 1 0	-1	1 1 0	-4	1 1 0	-16	1 1 0	-64
1 1 1	0	1 1 1	0	1 1 1	0	1 1 1	0

```
Result = 118 x multiplicand =  -2 x multiplicand     Factor 1
                            +   8 x multiplicand     Factor 4
                            + -16 x multiplicand     Factor 16
                            + 128 x multiplicand     Factor 64
                            ______________________
                              118 x multiplicand
```

FIG. 25-8. Booth's Algorithm: Multiplier = -2 Example



Factor 1		Factor 4		Factor 16		Factor 64	
0 0 0	0	0 0 0	0	0 0 0	0	0 0 0	0
0 0 1	1	0 0 1	4	0 0 1	16	0 0 1	64
0 1 0	1	0 1 0	4	0 1 0	16	0 1 0	64
0 1 1	2	0 1 1	8	0 1 1	32	0 1 1	128
1 0 0	-2	1 0 0	-8	1 0 0	-32	1 0 0	-128
1 0 1	-1	1 0 1	-8	1 0 1	-16	1 0 1	-64
1 1 0	-1	1 1 0	-4	1 1 0	-16	1 1 0	-64
1 1 1	0	1 1 1	0	1 1 1	0	1 1 1	0

```
Result = -2 x multiplicand =  -2 x multiplicand     Factor 1
                           +   0 x multiplicand     Factor 4
                           +   0 x multiplicand     Factor 16
                           +   0 x multiplicand     Factor 64
                           ---------------------
                              -2 x multiplicand
```

25.4.2 Multiply Implementation

25.4.2.1 Multiply Hardware

The multiply logic consists of a portion of each of the seven PEALU slices, the RS register, a 2:1 mux found on the output of ALE bits 9-16, and random logic. Some special clocking is also required to allow some clocks to operate on beat boundaries. There are a number of other little features inserted in the machine to solve special problems. These include the ability to reset the contents of RI, insert rounding bits into the original partial product using the RI path, and conditional decrement feature found in the REC controls.

The PEALU multiply logic consists of:

- Factor logic, used to generate intermediate factors from portions of the multiplier and multiplicand. This logic does the multiplication during each iteration of the loop.

- Carry-save adder, used to combine the several factors with the partial product to generate the next partial product in carry-save format. This logic combines the result of the current multiplication with the partial results from the previous iterations.

- Multiplicand register, used to hold the 48-bit multiplicand for the duration of the multiply operation. This logic also includes RD.

- Main ALU, used to convert the partial product, in carry-save format, to two's complement format.

- RS register, used to hold the multiplier, which is sent over the BDI bus from the barrel shifter.

Slicing

The 3-bit Booth's algorithm is implemented in each 8-bit PEALU slice. Our problems (multiply problems, anyway) would be over if all we ever wanted were 16-bit products. Since this isn't the case, we need some way to keep track of partial products produced by early iterations while we actually perform later iterations. This sounds like a job for the RSUM amd RCRY registers. Refer to Figure 25-9. This Figure shows two PEALU slices (if you imagine them as only one bit wide for the moment). Since we want to get a four bit product and we can only do two bit arithmetic we will need the services of these registers. In the example shown we are multiplying 2 x 3 unsigned. Three is the multiplicand, 2 is the multiplier. On the first iteration, each slice has a 0 in RS (the lsb of the multiplier). each slice multiplies 1 x 0 in the multiply logic (the multiplicand is also sliced) while adding RSUM + RCRY in the ALU. (Assume RCRY = RSUM = 0 on the first iteration.) The output of the ALU to the left (RSUM + RCRY = 0) is added to the partial product (0), and the result is stored in RSUM and RCRY. In the example, after the first iteration RSUM and RCRY on both slices are still equal to 0. On the second iteration, each slice has a 1 in RS. Each slice multiplies 1 x 1 while adding the ALU output of the slice to the left (RSUM + RCRY = 0) and storing the result (1) in RSUM and RCRY. The operation is finished by adding RSUM + RCRY on all the slices to obtain the final result.

Meanwhile, back in 48-bit arithmetic mode, the BDI VLSI logic is used to guarantee that each of the 7 PEALU slices is provided with the same 8 bits of the multiplier. This is called "broadcasting" the multiplier. The BDI takes the 8 LSBs of the ALEMUX input and duplicates this byte on every 8-bit BDI output bus.

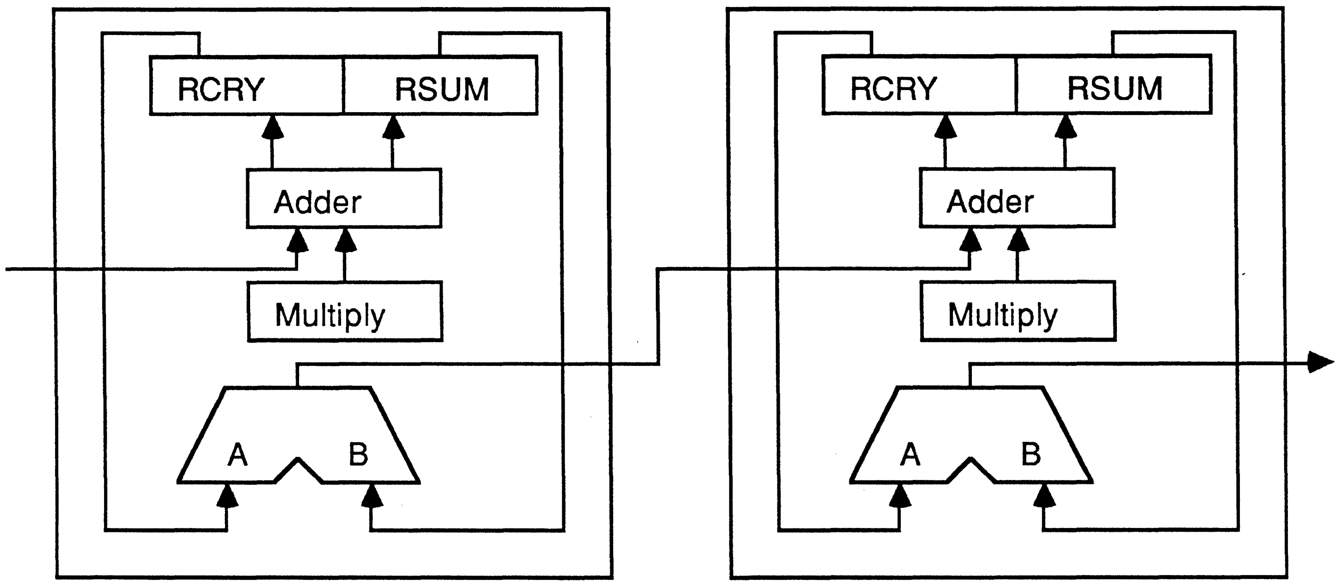
Another unusual section of logic is the RS register at the output of the BDI VLSI logic. This register is preceded by a 48-bit 2:1 mux, which provides a straight path to the register for normal operations and a second path which implements a 48 bit 'right shift by 8' algorithm. The purpose of this feature is to allow the multiplier, which is held in RS, to move a new byte of information into the low order 8 bits every iteration of the multiply loop.

Each slice performs the operation:

$$RS \times MCAND + RSUM + RCRY$$

The results are stored in RSUM and RCRY.

The slices are connected:



```
   11   MCAND
  x10   RS
 ----
 0110
```

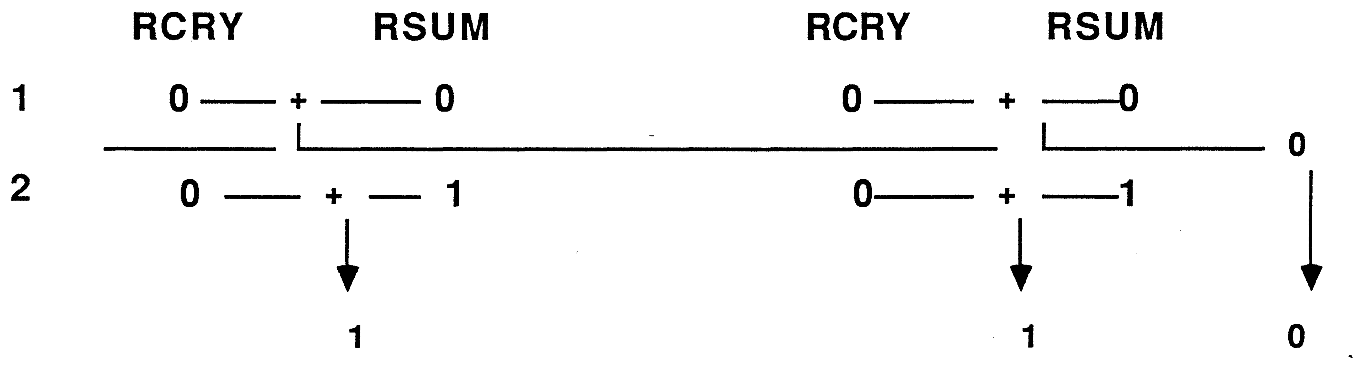


Figure 25-9 Multiply Hardware Slicing

25.4.2.2 Multiply Algorithm

A 16 x 16 bit integer multiply (MPY) is performed as follows:

1. Fetch the memory operand (multiplier) and pass it through the ALH slices. The low order 16 bits of the multiplier are forced to be zero through the ALL and ALE slices. This is detected in the BDI logic by testing the ALU outputs for this condition and altering the BDI shift controls if the condition is met. The altered controls will cause the BDI logic to perform a 32-bit right shift, putting the low byte of ALH into the coveted multiplier spot (e.g. right justifying the multiplier). The BDI also drives all zeroes during the first half of the step, which are clocked into RS, clearing the hidden bits.

2. The MultipliCAND (MCAND) is moved from the register file to the MCAND/RD registers. The first 8 bits of the multiplier are "broadcast" to the various PEALU chip RS registers, and some features are invoked.

   The first feature is a set of controls used to alter the normal RS, RSUM/RCRY clocks to operate on a beat boundary.

   The second feature is used to zero RI, thus providing an initial partial product of all zeroes. (Recall that each slice adds the output of the previous slice's ALU to its partial products, expecting them to be the sum RSUM + RCRY. By passing RI, RSUM + RCRY = 0 is assured for the first iteration.) It is important to note that the multiplicand passes through RI during this step, but the multiplicand is clocked into the MCAND register at CS8+, before RI gets cleared.

3. After the mutliplicand and multiplier are loaded (which occurs in parallel), one beat of the inner loop of the multiply is performed. This terminates the second step of the microcode algorithm. Then one more beat of the inner loop is executed, which occurs during the next microcode step, and the product is saved back into the register file.

   One beat of the inner loop performs the operation:

   (MCAND * 8 bits of multiplier) + RSUM + RCRY

   The parenthetical part of the expression is performed by the factor logic, while the 3 operand addition is accomplished by the carry-save adder.

4. MCAND gets loaded left shifted by one because the multiply hardware is optimized for floating point. The MPY product is right shifted by one, with the shift end being the true sign bit.

25.5 Divide

25.5.1 Non-Restoring Divide Algorithm

The 4150 implements a single bit non-restoring divide algorithm at the rate of a bit per beat. The concept behind a non-restoring divide algorithm is that successive iterations of the inner loop of the divide will reduce the partial remainder until it eventually reaches zero (or as close to zero as the precision of the divide will allow). If the last iteration left a positive remainder, subtract on this iteration. If the last iteration left a negative remainder, add on this iteration.

The sign bit produced by each iteration is a quotient bit, and can be saved in a shift register. If the dividend and divisor have the same sign these bits must be inverted before storage. Negative quotients require a increment to obtain a final result since the algorithm produces 1's complement numbers.

The algorithm always uses positive/positive numbers, and should therefore always produce a positive remainder. If the remainder isn't positive, adding the divisor to it will make it positive.

N quotient bits require n+1 iterations of the algorithm. The first one effectively divides the sign bits, which is meaningless.

Figures 25-10, 25-11, and 25-12 illustrate the operation of the algorithm. Each one shows the algorithm in operation as if it were doing a long division the way you were taught in elementary school (assuming you're young enough to have been taught the 'new' math). Quotient bits are collected down the left side of the page. The algorithm appears to be drifting to the right after each iteration. This is a graphical representation of the dividend being shifted one place left after each iteration as the sign bit is shifted off for quotient bit collection. The figures illustrate 8 bits divided by 4 bits, but the algorithm is easily expanded as many bits as necessary.

FIG. 25-10. Non-Restoring Divide Example: 6 / 2

```
                                               0 0 0 0 0 1 1 0
                                             - 0 0 1 0
                                             ---------------
Quotient Bits *            0 <-----------      1 1 1 0 0 1 1 0
                                             +   0 0 1 0
                                             ---------------
      |                    0 <-----------        1 1 1 0 1 1 0
      |                                        +   0 0 1 0
      |                                        -------------
      |                    0 <-----------          1 1 1 1 1 0
      |                                          +   0 0 1 0
      |                                          -----------
      |                    1 <-----------            0 0 0 1 0
      |                                            -   0 0 1 0
      |                                            ---------
      V                    1 <-----------              0 0 0 0  Remainder

                   6 / 2 = 3   remainder 0

                 * Quotient bits are inverted before collection
                   if the divisor and dividend have like signs.
```

FIG. 25-11. Non-Restoring Divide Example: 5 / -4

```
                                          0 0 0 0 0 1 0 1
                                        - 0 1 0 0              **
                                        ---------------
Quotient Bits *            1 <---------   1 1 0 0 0 1 0 1
                                        +   0 1 0 0
                                        ---------------
      |                    1 <---------     1 1 0 0 1 0 1
      |                                   +   0 1 0 0
      |                                   -------------
      |                    1 <---------       1 1 0 1 0 1
      |                                     +   0 1 0 0
      |                                     -----------
      |                    1 <---------         1 1 1 0 1
      |                                       +   0 1 0 0
      |                                       ---------
      V                *** 0 <---------           0 0 0 1   Remainder ****
```

5 / -4 = -1 remainder -1

* Quotient bits are not inverted before collection if the divisor and dividend have different signs.

** The algorithm requires positive/positive. The divisor is inverted before the divide starts.

*** Negative quotients require an increment to obtain the final answer because the algorithm produces 1's complement answers.

**** The algorithm always produces positive remainders. The microcode must explicitly make the remainder negative when necessary.

FIG. 25-12. Non-Restoring Divide Example: 14 / 3

```
                                            0 0 0 0 1 1 1 0
                                           - 0 0 1 1
                                           ---------------
Quotient Bits *          0 <-------------   1 1 0 1 1 1 1 0
                                           +  0 0 1 1
                                           ---------------
       |                 0 <-------------     1 1 1 0 1 1 0
       |                                     +  0 0 1 1
       |                                     -------------
       |                 1 <-------------       0 0 0 0 1 0
       |                                      -   0 0 1 1
       |                                       -----------
       |                 0 <-------------         1 1 1 0 0
       |                                        +   0 0 1 1
       |                                         ---------
       V                 0 <-------------           1 1 1 1
                                                  + 0 0 1 1   **
                                                   -------
                                                    0 0 1 0 Remainder

                  14 / 3 = 4  remainder 2

          * Quotient bits are inverted before collection
            if the divisor and dividend have like signs.

         ** The algorithm must always produce a positive
            remainder.  If it doesn't, add the divisor to
            it ot make it positive.
```

25.5.2 Divide Implementation

25.5.2.1 Divide Hardware

The divide logic consists of a section of each of the seven PEALU VLSI chips and some special clocking features. The 48-bit RI is implemented as a 48-bit shift register (left shift by 1) in order to capture the quotient through the link bit. Clocking, control, and parity checking during a divide operation are handled by microcode via the IAC field.

The hardware is set up to accept only positive numbers divided by positive numbers. Microcode is used to accept any other combination of numbers and do the appropriate action required by the hardware.

The PEALU VLSI divide logic consists of:

- A special ALU mode which uses the ALTernate SELect (ALTSEL) input of the chip to determine if the operation will be an add or a subtract.

- The RD register, which is used to hold the partial remainder. The RD register is fed by a 2:1 mux. One of the mux inputs is the 56-bit ALU output wired as a one bit left shift. This mux input allows us to left shift the partial remainder automatically each iteration.

- The RSUM_IBB register is used to hold some data temporarily during the setup to the inner loop.

The RI register usually operates as a 48-bit, parallel load, parallel unload register, and is generally clocked at CS7+. During a divide, RI operates as the quotient register. RI becomes a 48-bit shift register, set to perform a left shift by 1 at every clock. Since the inner loop of the divide is set to dispatch one bit per beat, the clock to RI must also perform at the rate of one bit per beat. The shift input to RI is the link bit. The link bit input is ALHCOUT+ (ALH Carry Out), which is selected by issuing IAC DIVIDE.

Creative clocking allows the algorithm to be performed in few microcode steps. Since the inner loop operates at a rate of one bit per beat, there should be two bits per microcode step. If the microcode step has a TX= 2 in it, there would be three beats during the step and thus three bits of quotient. This use of TXs allows the microcode to extend the usefulness of each step and reduce the length of the algorithm in terms of steps executed.

25.5.2.2 Divide Algorithm

A 32 / 16 bit integer divide (DIV) yielding 16 bits of quotient and 16 bits of remainder is performed as follows:

1. The algorithm starts out by checking the signs of the divisor and the dividend, 2's complementing these arguments as necessary so that the inner loop is entered with a positive divisor and a positive dividend. The dividend is loaded into RD.

2. The divisor is subtracted from the dividend, creating a partial remainder. The sign bit of this partial remainder is used to select the ALU operation to perform on the next iteration. The sign bit is also saved in RI as the quotient. This is one iteration. The result of the operation is loaded into RD left shifted by 1.

3. The ALU mode selection will be an add if the result of the last iteration produced a negative number.

   The ALU mode selection will be a subtract if the result of the last iteration produced a positive number.

4. Repeat step three as many times as necessary to produce the desired number of quotient bits. The value in RD at the end is the remainder. If it is negative, add the divisor to it to make it positive again.

5. Fix the sign of the quotient and remainder as necessary based on the signs of the original operands.

25.6 Floating Point

Floating point numbers allow non-integers and very large numbers to be represented inside a computer. The reader is strongly urged to familiarize him/herself with the floating point chapter of the System Architecture Reference Manual before attempting this section.

In general, the floating point instruction set performs operations on two floating point operands. One of the operands is in memory (cache), while the other is in a Floating ACcumulator (FAC). The FACs are in the register file of the E unit. The exponent part of any floating point number is kept in excess 128 notation.

All floating point numbers are stored in a format called normalized. This means that the first two bits of the mantissa are different. Storing numbers in this fashion assures that there is no duplication of sign bits at the beginning of the number, and provides as many bits as possible at the end of the number for significant digits (precision). The binary point is always assumed to be between bit 1 and bit 2.

25.6.1 Floating Add

A floating point add in binary is analogous to a scientific notation add in base ten. Just as you have to line up the decimal point before you can do the addition in base ten (remember the 'new' math?), you have to line up the binary points before you can do the addition in base two. Keeping that in mind, here's the algorithm for a floating point add.

1. Make the exponents of the two numbers equal. This is the same as lining up the binary points, and is called adjusting. The adjust function is explained in excruciating detail below.

2. Add the two mantissas.

3. Normalize the result.

25.6.1.1 Adjust

As stated above, adjusting is done to align the binary point before an addition. Since the binary points don't really exist they can't be moved. We have to settle for moving the data around. The data can't be shifted left, since that would destroy significant bits. Shifting data to the right (sign extending) while holding the (imaginary) binary point stationary implies that the mantissa is getting smaller. To keep the number the same, you have to add to the exponent. The exponent will consequently get larger. When the exponents are equal the binary points are aligned. Thus we have to adjust the operand with the smaller exponent (the smaller operand).

Adjusting is done in two steps:

1. Subtract the exponents. N = exp1 - exp2. ALU operation. Store N in the latch LADJCNT (Latch ADJust CouNT).

2. Shift the mantissa of the smaller operand right |N| places, sign extended, in the barrel shifter.

For example:

```
010E3
101E1      101E1 needs to be adjusted.  It would become 11101E3.

010E3
011E3      Neither operand needs adjusting.

010E48
010E-5     010E-5 needs to be adjusted.
```

The last example illustrates the Out-of-RANGE (ORANGE) case. The barrel shifter can't shift more than 48 bits. The barrel shifter detects this case and puts out all zeros in this case, and the algorithm proceeds normally.

Tha ALU automatically selects the correct operand to adjust during step 2 if the microcode uses the ALU mode ADJUST. It is assumed by the table that exp1 was the FAC (A leg operand) and exp2 was the memory (B leg) operand. Table 25-4 illustrates this mode of operation.

TABLE 25-4. ADJUST ALU Mode

FALL00+	ALU Mode
0	(TB,TB,TB)
1	(TA,TA,TA)

The last two bits shifted off the right end during a barrel shifter adjust operation are saved in the guard bits. These bits may be shifted back in during a normalize operation to increase the precision of the result.

25.6.1.2 Normalization

Normalization is done at the end of every floating point operation to insure that there are no duplicate sign bits and that the answer is as precise as possible.

To discard extra sign bits we must shift the data left. The guard bits will be the first two bits shifted in at the right, with all subsequent bits forced to zero by the barrel shifter. Shifting left while holding the (imaginary) binary point stationary implies that the mantissa is getting larger. To keep the number the same we have to subtract the shift count from the exponent.

Normalization is also done in two steps:

1. Shift the data left until the first two bits are different. N = the number of places shifted, Positive for left shift, negative for right shifts. This operation is done in the barrel shifter, which reports N in an ALU accessible register named FNRMCNT (flopped normalize count).

2. Add N to the exponent in the ALU.

Continuing with the example discussed under adjust:

```
  01000 E3
+ 11101 E3
----------
  00101 E3
      |
      |   N = 1
      |
      V
  0101 E2


  0100 E3
+ 0110 E3
---------
 01010 E3
     |
     |  N = -1
     |
     V
  0101 E4
```

The second example shows what happens when the addition causes an overflow. (We are adding two positive numbers and are about to get a negative result.) The barrel shifter monitors the ALU overflow condition and shifts the sign bit back in automatically in this case.

25.6.2 Floating Subtract

The floating point subtract algorithm cheats by converting itself into a floating point add:

1. Two's complement the number to be subtracted.

2. Adjust the mantissa that needs adjusting.

3. Add the mantissas.

4. Normalize the result.

25.6.3 Floating Multiply

A floating point multiply operation is analogous to a scientific notation multiply in base ten. Alignment of the binary points of the operands isn't necessary. The tricky part to remember is that the exponents are in excess 128 notation.

The algorithm:

1. Add the two exponents and subtract 128. (exp1 + 128) + (exp2 + 128) - 128 = (exp3 + 128)

2. Multiply the two mantissas together. This is the same operation as the integer multiply discussed earlier.

3. Normalize the result.

Special hardware is available to assist with each step. The multiply hardware of step 2 and the normalize hardware of step 3 have already been discussed. Step 1 discusses the exponent manipulation. The exponent are actually added in the ALU, L side, and the result is stored in the Register Event Counter (REC) inside the PRFADR chip. This general purpose up/down counter is usually used in a strict increment or decrement mode, but also supports addition or subtraction of 128 for exponent operations.

25.6.4 Floating Divide

The floating divide algorithm needs no further introduction:

1. Subtract the divisor's exponent from the dividend's exponent and add 128. (exp1 + 128) - (exp2 + 128) + 128 = (exp3 + 128)

2. Divide the dividend mantissa by the divisor mantissa. This is the same operation as the integer divide discussed earlier.

3. Normalize the result.

4. Round the result. Rounding is discussed in the next section.

5. Normalize the result.

25.6.5 Rounding

Rounding is a software selectable floating point option. Rounding is enabled by setting a bit in the MODALS register. When this bit is active all floating point results are rounded before storage.

Rounding refers to adding a 1 to the bit of lesser significance than the least significant bit.

For example, double precision arithmetic produces a 48-bit result. To round the result we add a 1 to bit 49. If this produces a carry out, bits 1-48 change accordingly.

What is bit 49 if there are only 48 bits in the answer? The guard bits are bits 49 and 50 in double precision. Earlier, it was stated that the guard bits were saved during an adjust operation. The guard bits are available as inputs to ALX. The addition of a 1 for rounding occurs in ALX for double and quadruple precision. In quadruple precision, the 1 is added to bit 97. Single precision rounding is done at bit 25 in ALL.

Rounding can lead to a performance degradation. We can't perform a rounding operation until we know where bit 49 is (assume double precision for this discussion). We don't where bit 49 is until after we normalize the result. If we normalize the result and then round we have to normalize the result again to make sure the carry didn't propogate all the way to bit 2 or higher. We can solve this problem by looking at the result before it is normalized, determine where bit 49 is, round, and then normalize. The round bit inputs to ALX provide this feature. Round bit 1 is input at bit 49, while round bit 2 is input at bit 50.

Consider the possible positioning of bit 49 before normalization. We can find out where bit 49 is by looking at bits 1-3 of the answer and the overflow indicator. There are four possibilities for how much normalization will be needed. Each is discussed in turn:

- The overflow indicator is not on and bits 1-3 are all the same. Normalization will shift left at least 2 bits (bit 3 becomes bit 1), and both guard bits will be shifted in. Result bit 49 is currently somewhere beyond bit 50. All bits beyond bit 50 are 0, so adding a 1 can't produce a carry out. No rounding needs to be done.

- The overflow indicator is not on and bits 1 and 2 are the same, while bit 3 is different. Normalization will shift exactly 1 bit left (bit 2 becomes bit 1), and 1 guard bit will be shifted in. Result bit 49 is currently at bit 50. Set round bit 2, add it to the guard bits in ALX, propogate any carry outs, and normalize the number.

- The overflow indicator is not on and bits 1 and 2 are different. Normalization will not shift at all, as the number is already normalized. Result bit 49 is currently at bit 49. Set round bit 1, add it to the guard bits in ALX, propogate any carry outs, and normalize the number.

- The overflow indicator is on. Normalization will shift one bit right (bit 0 becomes bit 1). Result bit 49 is currently at bit 48. Set round bit 1, force guard bit 1 to be set with signal RNDDONE-, and add the round and guard bits together in ALX, forcing a carry into bit 48. Normalize the number.

25.6.5.1 Rounding During Floating Multiply

The usual procedure if rounding is required is to do the operation and then round and normalize in two steps at the end. Time can be saved in certain situations during floating multiply by doing the rounding during the multiply. This saves the extra step of specifically letting the logic which produces the round bits look at the result before normalization.

Rounding during multiply is done on the first iteration of the multiply loop. Recall that there is a special feature which makes RI all zeros during the first iteration of the multiply. The carry save adders in each PEALU slice interpret this as the sum of the partial products of the previous iteration. If we can determine where bit 49 will be in the final result before normalization we can inject a 1 into the otherwise all zeros word coming out of RI during the first iteration, rounding on the fly.

Determining where bit 49 will be in the final result is the job of the (appropriately named) guess PROM. This PROM looks at the first 4 bits of each operand and, if it can determine where bit 49 will end up, produces one of two possible floating multiply round bits. If it can't determine where bit 49 will be both bits are set to 0. The microcode is notified of the success or failure of the attempt via jump conditions.

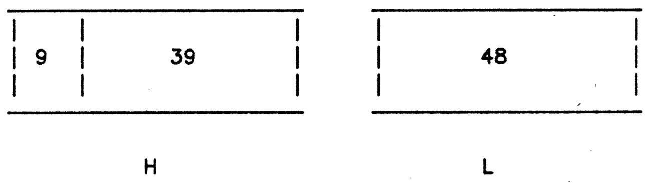
The round bits are injected via a multiplexer into ALL{9:10}+ during the first iteration in single precision, or into ALH{1:2}+ during double precision. The least significant bit of a multiply result ends up at ALE16+. Each multiply iteration effectively shifts the accumulated partial products right 8 bits. Single precision multiplies 24 x 24 bits, requiring three 24 x 8 iterations. Bit 25 is therefore at ALL bit 9 or 10 during the first iteration. A similar analysis of double precision's six 48 x 8 bit iterations places bit 49 at ALH bit 1 or 2 during the first iteration.

25.6.6 Quadruple Precision

Quadruple precision deals with 96-bit mantissas. Quad operations must be done in pieces since there isn't a 96-bit ALU in the E unit. Microcode is responsible for doing these operations correctly.

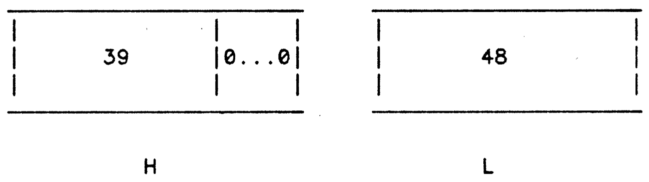
There is logic in the barrel shifter to make the microcode's job a little easier by supporting normalization of 96-bit numbers. This logic is called the reverse mask logic. A reverse mask assumes LADJCNT has been loaded with the number of bits to mask. The operation does a LADJCNT-bit rotate over 48 bits, saving only the bottom LADJCNT bits and forcing the rest to zero. For example, suppose some result needs a 9 bit shift to be normalized correctly. Figure 25-13 shows how this operation would take place.

FIG. 25-13. Quadruple Precision Normalize

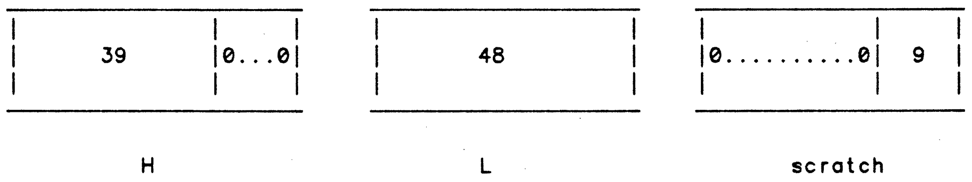


9 bits need to be shifted to normalize the result.

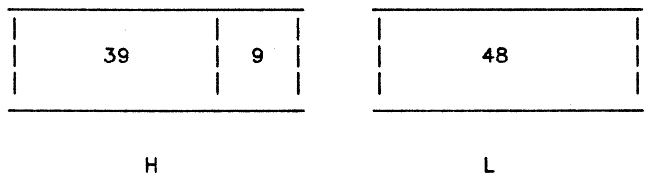
1. Normalize the top 48 bits, zero filled, saving the normalize count as usual in FNORMCNT.



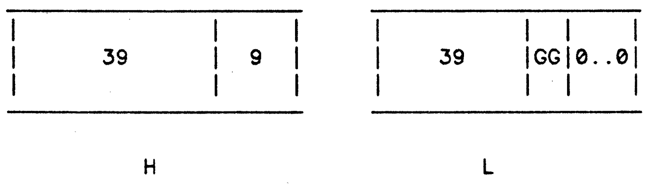
2. Subtract FNORMCNT from the exponent as usual. Load LADJCNT with FNORMCNT. Reverse mask the low 48 bits, saving the result in a scratch register. This effectively moves the top 9 bits to the bottom.



3. Or the top 48 bits with scratch.



4. Shift the bottom bits left LADJCNT places with the guard bits.



25.7 Character and Decimal Instructions

25.7.1 Introduction

The character and decimal instructions in the instruction set are usually discussed together even though the differences between are more numerous than the similarities. The entire topic has historically been treated as voodoo. A few of the incantations about character instructions have leaked out over the years, while decimal has remained a closely guarded black art. This section will attempt to bring these instructions out into the open. The reader is strongly urged to familiarize him/herself with the character and decimal chapter of the System Architecture Reference Guide. before reading this section. Please, kids, don't try this at home, it's dangerous.

The major similarities between character and decimal are:

- Both operate on byte data.
- As a direct consequence of the first point, both have to worry about the alignment of the operands.
- Each operand address points to a string of operands. (In the rest of the instruction set, an operand address points to a single operand.)
- Both use special ALU modes.
- Both use special jump conditions.
- Both are fairly incomprehensible at first (and second) glance.

The major differences between the two modes are:

- Character instructions always deal with byte data. Decimal instructions can deal with either byte or nibble data.
- Character uses Field Length Registers (FLR) for operand lengths. Decimal uses its control register for operand lengths.
- Character uses Field Address Registers (FAR) for operand addressing. Decimal uses general registers for operand addressing.
- All the information needed to process character instructions is present in the opcode, FARs, and FLRs. Decimal instructions also need a control word (L register in V mode, GR2 in I mode).
- Character strings are processed left to right (English). Decimal strings are processed right to left (Hebrew).

25.7.1.1 Alignment

In the discussion of character and decimal instructions the term alignment is used to refer to how a word is aligned in memory. Data operated on by character and decimal instructions can start on byte boundaries. Figure 25-14 illustrates differences in alignment.

FIG. 25-14. Character and Decimal Alignment

```
              1             8   9              16
              ---------------   ----------------
  '1000      |W| | | | | | | | |X| | | | | | | |
              ---------------   ----------------
              17           24   25             32
              ---------------   ----------------
  '1001      |Y| | | | | | | | |Z| | | | | | | |
              ---------------   ----------------
```

In Figure 25-14, if a word begins at W it has alignment 0, at X it has alignment 1, at Y it has alignment 2, and at Z it has alignment 3. The alignment information is stored in the ZFF bits. These bits are set up to contain information on destination alignment in memory (ZFF{0:1}+), source alignment in memory (ZFF{2:3}+), and the number of bytes (mod 4) to be moved during this instruction (ZFF{4:5}+). The bits are saved in latches which are controlled by microcode IACs as shown in Table 25-5. The entries in the table show the source for the bits.

The ZFF bits are referred to as XFF bits in decimal instructions. They are the same bits in hardware. Microcode is schizophrenic, so uses two different names for the same piece of hardware.

TABLE 25-5. ZFF IACs

Dst SRC Bytes left

IAC	ZFF0	ZFF1	ZFF2	ZFF3	ZFF4	ZFF5
LXFF01	RIE01	RIE02	-	-	-	-
LXFF23	-	-	RIE01	RIE02	-	-
LXFF45	-	-	-	-	BBH09	BBH10
LZFF01	RIL16	RIE01	-	-	-	-
LZFF23	-	-	RIL16	RIE01	-	-
LZFF45	-	-	-	-	RIL15	RIL16

25.7.2 Character Instructions

The character instruction set contains 8 instructions. All of them move at least 1 character from one place to another. A few samples, in increasing order of difficulty:

LDC	Loads the character at the effective address into bits 9-16 of the specified register. Bits 1-8 are cleared.
ZMV	A string of characters in memory is copied into another string of characters in memory.
ZED	A string of characters in memory is accessed and edited under the control of character edit string subprogram. The edited characters are stored into another string in memory.

This section will describe all the hardware parts which have a role in character manipulation. After this discussion there will be two examples of ZMV instructions which will demonstrate how all the parts play together.

25.7.2.1 ZMV PROMs

ZMV is an ALU mode which really does most of its work in the barrel shifter. This mode always tells the ALU to pass the B leg input through unchanged to the barrel shifter. It is intended that the microcode is doing a read of a source string from cache during this step.

The barrel shifters goal is to realign the source string so that it has the same alignment as the destination string in memory. Once this is done it is easy for the microcode to pick the bytes it wants from each string, merge them together, and write them to memory.

The entities which provide the control for this operation are the ZMV PROMs. They provide the shift count to the PBDI slices. They also produce some of the special character mode jump conditions necessary to insure proper microcode sequencing.

The action that is taken for various alignments is shown in Table 25-6. In this table and the rest of the tables in this section, X is a don't care entry.

The jump conditions which pertain to character conditions are ZBIT1, ZBIT2, XSRCDST, and ZSTRD. They are defined differently for different ALU modes.

ZMSTRT

ZMSTRT defines the jump conditions used to see how many reads you need to do in order to write 32 bits to the destination at the same time.

- ZBIT1 is always zero.
- ZBIT2 is active if you need an extra read of the source to satisfy the destination. It looks at the alignment of the source, ZFF{2:3}+, and destination, ZFF{0:1}+.

TABLE 25-6. ZMV BDI Alignment

ZFF(0:3)	Alignment DST	Alignment SRC	BDI Mode 32 bits	Result
0 0 0 0	ABCD	ABCD	No Rotate	ABCD
0 0 0 1	ABCD	XABC	Left Rotate 8 bits	ABCX
0 0 1 0	ABCD	XXAB	Left Rotate 16 bits	ABXX
0 0 1 1	ABCD	XXXA	Left Rotate 24 bits	AXXX
0 1 0 0	XABC	ABCD	Left Rotate 24 bits	DABC
0 1 0 1	XABC	XABC	No Rotate	XABC
0 1 1 0	XABC	XXAB	Left Rotate 8 bits	XABX
0 1 1 1	XABC	XXXA	Left Rotate 16 bits	XAXX
1 0 0 0	XXAB	ABCD	Left Rotate 16 bits	CDAB
1 0 0 1	XXAB	XABC	Left Rotate 24 bits	CXAB
1 0 1 0	XXAB	XXAB	No Rotate	XXAB
1 0 1 1	XXAB	XXXA	Left Rotate 8 bits	XXAX
1 1 0 0	XXXA	ABCD	Left Rotate 8 bits	BCDA
1 1 0 1	XXXA	XABC	Left Rotate 16 bits	BCXA
1 1 1 0	XXXA	XXAB	Left Rotate 24 bits	BXXA
1 1 1 1	XXXA	XXXA	No Rotate	XXXA

- XSRCDST is active if there are 4 bytes to move. It looks at the destination alignment, ZFF{0:1}+, and the number of bytes to move, ZFF{4:5}+.
- ZSTRD is active if the number of bytes to be written and the alignment are such that they "straddle" across two 32 bit words.

Tables 25-7 and 25-8 illustrate these conditions.

ZMFAST

For ZMFAST, the information coming out of the ZMV PROMs is used to determine where blank padding should begin. It looks at the destination alignment in ZFF{0:1}+, and number of bytes to be padded in ZFF{4:5}+. ZSTRD is active if the number of bytes to be written and the alignment are such that they "straddle" across two 32-bit words. Table 25-9 illustrates these conditions.

ZMSHRT

ZMSHRT is used if there are less than 4 bytes in the destination. It looks at all of the ZFF bits to define the jump conditions. ZBIT1 is active if ZFF{0:1}+, and ZFF{4:5}+ are all equal to zero. ZBIT2 is active if you need an extra read of the source to meet the number of bits required. XSRCDST is active if the source alignment is less than the destination alignment. ZSTRD is active if the number of bytes to be written and the alignment are such that they "straddle" across two 32-bit words. Tables 25-10, 25-11, and 25-12 illustrate these conditions.

ZMEXIT

TABLE 25-7. ZMSTRT ZBIT Jump Conditions for ZMV

	Alignment		
ZFF(0:3)	DST	SRC	ZBIT(1:2)
0 0 0 0	ABCD	ABCD	0 0
0 0 0 1	ABCD	XABC	0 1
0 0 1 0	ABCD	XXAB	0 1
0 0 1 1	ABCD	XXXA	0 1
0 1 0 0	XABC	ABCD	0 0
0 1 0 1	XABC	XABC	0 0
0 1 1 0	XABC	XXAB	0 1
0 1 1 1	XABC	XXXA	0 1
1 0 0 0	XXAB	ABCD	0 0
1 0 0 1	XXAB	XABC	0 0
1 0 1 0	XXAB	XXAB	0 0
1 0 1 1	XXAB	XXXA	0 1
1 1 0 0	XXXA	ABCD	0 0
1 1 0 1	XXXA	XABC	0 0
1 1 1 0	XXXA	XXAB	0 0
1 1 1 1	XXXA	XXXA	0 0

TABLE 25-8. Other ZMSTRT Jump Conditions for ZMV

ZFF 0 1 4 5	DST	BYTES	XSRCDST	ZSTRD
0 0 0 0	ABCD	0	1	0
0 0 0 1	ABCD	1	0	0
0 0 1 0	ABCD	2	0	0
0 0 1 1	ABCD	3	0	0
0 1 0 0	XABC	0	1	0
0 1 0 1	XABC	1	0	0
0 1 1 0	XABC	2	0	0
0 1 1 1	XABC	3	0	0
1 0 0 0	XXAB	0	1	0
1 0 0 1	XXAB	1	0	0
1 0 1 0	XXAB	2	0	0
1 0 1 1	XXAB	3	0	1
1 1 0 0	XXXA	0	1	0
1 1 0 1	XXXA	1	0	0
1 1 1 0	XXXA	2	0	1
1 1 1 1	XXXA	3	0	1

ZMEXIT is used to merge the last write with destination bits which need to be preserved. ZBIT1 is active if you need to do and extra read alignment and mask/merge of the source. ZBIT2 is active if you need to do a read modify write of the destination. XSRCDST is always active. ZSTRD is active if the number of bytes to be written and the alignment are such that they "straddle" across two 32-bit words. Tables 25-13 and 25-14 illustrates these conditions.

TABLE 25-9. ZMFAST Jump Conditions for ZMV

ZFF 0 1 4 5	DST	BYTES	ZBIT(1:2)	XSRCDST	ZSTRD
0 0 0 0	ABCD	0	0 0	0	0
0 0 0 1	ABCD	1	0 1	0	0
0 0 1 0	ABCD	2	1 0	0	0
0 0 1 1	ABCD	3	1 1	0	0
0 1 0 0	XABC	0	0 1	1	0
0 1 0 1	XABC	1	1 0	0	0
0 1 1 0	XABC	2	1 1	0	0
0 1 1 1	XABC	3	0 0	0	0
1 0 0 0	XXAB	0	1 0	1	0
1 0 0 1	XXAB	1	1 1	0	0
1 0 1 0	XXAB	2	0 0	0	0
1 0 1 1	XXAB	3	0 1	1	1
1 1 0 0	XXXA	0	1 1	1	0
1 1 0 1	XXXA	1	0 0	0	0
1 1 1 0	XXXA	2	0 1	1	1
1 1 1 1	XXXA	3	1 0	1	1

TABLE 25-10. ZMSHRT Jump Conditions for ZMV, Part 1

ZFF 0 1 4 5	DST	BYTES	ZBIT1	ZSTRD
0 0 0 0	ABCD	0	1	0
0 0 0 1	ABCD	1	0	0
0 0 1 0	ABCD	2	0	0
0 0 1 1	ABCD	3	0	0
0 1 0 0	XABC	0	0	0
0 1 0 1	XABC	1	0	0
0 1 1 0	XABC	2	0	0
0 1 1 1	XABC	3	0	0
1 0 0 0	XXAB	0	0	0
1 0 0 1	XXAB	1	0	0
1 0 1 0	XXAB	2	0	0
1 0 1 1	XXAB	3	0	1
1 1 0 0	XXXA	0	0	0
1 1 0 1	XXXA	1	0	0
1 1 1 0	XXXA	2	0	1
1 1 1 1	XXXA	3	0	1

TABLE 25-11. ZMSHRT Jump Conditions for ZMV, Part 2

ZFF(2:5)	SRC	BYTES	ZBIT2
0 0 0 0	ABCD	0	0
0 0 0 1	ABCD	1	0
0 0 1 0	ABCD	2	0
0 0 1 1	ABCD	3	0
0 1 0 0	XABC	0	0
0 1 0 1	XABC	1	0
0 1 1 0	XABC	2	0
0 1 1 1	XABC	3	0
1 0 0 0	XXAB	0	0
1 0 0 1	XXAB	1	0
1 0 1 0	XXAB	2	0
1 0 1 1	XXAB	3	1
1 1 0 0	XXXA	0	0
1 1 0 1	XXXA	1	0
1 1 1 0	XXXA	2	1
1 1 1 1	XXXA	3	1

TABLE 25-12. ZMSHRT Jump Conditions for ZMV, Part 3

ZFF(0:3)	DST	SRC	XSRCDST
0 0 0 0	ABCD	ABCD	0
0 0 0 1	ABCD	XABC	0
0 0 1 0	ABCD	XXAB	0
0 0 1 1	ABCD	XXXA	0
0 1 0 0	XABC	ABCD	1
0 1 0 1	XABC	XABC	0
0 1 1 0	XABC	XXAB	0
0 1 1 1	XABC	XXXA	0
1 0 0 0	XXAB	ABCD	1
1 0 0 1	XXAB	XABC	1
1 0 1 0	XXAB	XXAB	0
1 0 1 1	XXAB	XXXA	0
1 1 0 0	XXXA	ABCD	1
1 1 0 1	XXXA	XABC	1
1 1 1 0	XXXA	XXAB	1
1 1 1 1	XXXA	XXXA	0

TABLE 25-13. ZMEXIT Jump Conditions for ZMV, Part 1

		Alignment				
ZFF(4:5)	ZFF(0:3)	DST	SRC	Bytes	ZBIT(1:2)	ZSTRD
0 0	0 0 0 0	ABCD	ABCD	0	0 0	0
0 0	0 0 0 1	ABCD	XABC	0	0 0	0
0 0	0 0 1 0	ABCD	XXAB	0	0 0	0
0 0	0 0 1 1	ABCD	XXXA	0	0 0	0
0 0	0 1 0 0	XABC	ABCD	0	0 1	0
0 0	0 1 0 1	XABC	XABC	0	1 1	0
0 0	0 1 1 0	XABC	XXAB	0	0 1	0
0 0	0 1 1 1	XABC	XXXA	0	0 1	0
0 0	1 0 0 0	XXAB	ABCD	0	0 1	0
0 0	1 0 0 1	XXAB	XABC	0	1 1	0
0 0	1 0 1 0	XXAB	XXAB	0	1 1	0
0 0	1 0 1 1	XXAB	XXXA	0	0 1	0
0 0	1 1 0 0	XXXA	ABCD	0	0 1	0
0 0	1 1 0 1	XXXA	XABC	0	1 1	0
0 0	1 1 1 0	XXXA	XXAB	0	1 1	0
0 0	1 1 1 1	XXXA	XXXA	0	1 1	0
0 1	0 0 0 0	ABCD	ABCD	1	1 1	0
0 1	0 0 0 1	ABCD	XABC	1	0 1	0
0 1	0 0 1 0	ABCD	XXAB	1	0 1	0
0 1	0 0 1 1	ABCD	XXXA	1	0 1	0
0 1	0 1 0 0	XABC	ABCD	1	1 1	0
0 1	0 1 0 1	XABC	XABC	1	1 1	0
0 1	0 1 1 0	XABC	XXAB	1	0 1	0
0 1	0 1 1 1	XABC	XXXA	1	0 1	0
0 1	1 0 0 0	XXAB	ABCD	1	1 1	0
0 1	1 0 0 1	XXAB	XABC	1	1 1	0
0 1	1 0 1 0	XXAB	XXAB	1	1 1	0
0 1	1 0 1 1	XXAB	XXXA	1	0 1	0
0 1	1 1 0 0	XXXA	ABCD	1	1 0	0
0 1	1 1 0 1	XXXA	XABC	1	1 0	0
0 1	1 1 1 0	XXXA	XXAB	1	1 0	0
0 1	1 1 1 1	XXXA	XXXA	1	1 0	0

TABLE 25-14. ZMEXIT Jump Conditions for ZMV, Part 2

		Alignment				
ZFF(4:5)	ZFF(0:3)	DST	SRC	Bytes	ZBIT(1:2)	ZSTRD
1 0	0 0 0 0	ABCD	ABCD	2	1 1	0
1 0	0 0 0 1	ABCD	XABC	2	0 1	0
1 0	0 0 1 0	ABCD	XXAB	2	0 1	0
1 0	0 0 1 1	ABCD	XXXA	2	1 1	0
1 0	0 1 0 0	XABC	ABCD	2	1 1	0
1 0	0 1 0 1	XABC	XABC	2	1 1	0
1 0	0 1 1 0	XABC	XXAB	2	0 1	0
1 0	0 1 1 1	XABC	XXXA	2	1 1	0
1 0	1 0 0 0	XXAB	ABCD	2	1 0	0
1 0	1 0 0 1	XXAB	XABC	2	1 0	0
1 0	1 0 1 0	XXAB	XXAB	2	1 0	0
1 0	1 0 1 1	XXAB	XXXA	2	1 0	0
1 0	1 1 0 0	XXXA	ABCD	2	0 1	1
1 0	1 1 0 1	XXXA	XABC	2	0 1	1
1 0	1 1 1 0	XXXA	XXAB	2	0 1	1
1 0	1 1 1 1	XXXA	XXXA	2	1 1	1
1 1	0 0 0 0	ABCD	ABCD	3	1 1	0
1 1	0 0 0 1	ABCD	XABC	3	0 1	0
1 1	0 0 1 0	ABCD	XXAB	3	1 1	0
1 1	0 0 1 1	ABCD	XXXA	3	1 1	0
1 1	0 1 0 0	XABC	ABCD	3	1 0	0
1 1	0 1 0 1	XABC	XABC	3	1 0	0
1 1	0 1 1 0	XABC	XXAB	3	1 0	0
1 1	0 1 1 1	XABC	XXXA	3	1 0	0
1 1	1 0 0 0	XXAB	ABCD	3	0 1	1
1 1	1 0 0 1	XXAB	XABC	3	0 1	1
1 1	1 0 1 0	XXAB	XXAB	3	1 1	1
1 1	1 0 1 1	XXAB	XXXA	3	0 1	1
1 1	1 1 0 0	XXXA	ABCD	3	0 1	1
1 1	1 1 0 1	XXXA	XABC	3	0 1	1
1 1	1 1 1 0	XXXA	XXAB	3	1 1	1
1 1	1 1 1 1	XXXA	XXXA	3	1 1	1

25.7.2.2 XMODE PROMs

The XMODE PROMs control the ALU modes for character and decimal instructions. The 4 ALU modes associated with character moves are shown in Table 25-15. These ALU modes behave differently for different combinations of ZFF bits.

The character ALU modes can be invoked without calling them out specifically in the microcode word. This kind of ALU mode is called a "wired" ALU mode. Wired ALU modes are designated by a "W" in t he ALEG - ALU table in the microcoder's handbook. When a wired ALU mode is used the XMODE PROMs produce the jump conditions based on RCMALU{6:8}+. RCMALU6+ equal to 0 implies a character ALU mode, while RCMALU6+ equal to 1 implies a decimal ALU mode.

TABLE 25-15. Character ALU Modes

MODE	RCMALU(07:08)
ZMSTRT	0 0
ZMFAST	0 1
ZMEXIT	1 0
ZMSHRT	1 1

ZMSTRT

ZMSTRT looks at the the destination alignment. It is assumed that the source data has been aligned to match the destination. We need to preserve the part of the destination which will not be written over. To do this, the microcode puts the old destination word on the B leg and the new source word on the A leg, and uses the ZMSTRT ALU mode. The ZFF bits affect how the ALU performs as shown in Table 25-16. In this table, the string ABCDEFH is being moved to the point starting at the J character.

TABLE 25-16. ZMSTRT ALU Modes For Character Instructions

	B	A		
ZFF(0:1)	DST	SRC	ALU mode	Result
0 0	JKLM	ABCD	(TA:TA, TA:TA,)	ABCD
0 1	IJKL	DABC	(TB:TA, TA:TA,)	IABC
1 0	HIJK	CDAB	(TB:TB, TA:TA,)	HIAB
1 1	GHIJ	BCDA	(TB:TB, TB:TA,)	GHIA

ZMFAST

ZMFAST is used to make a 32-bit word to be written to the destination. On the A leg, the microcode puts the old source word which has already been written (either partially or fully) by a ZMSTRT. The new source word is passed on the B leg. ZMFAST is called out for the ALU mode, which causes the XMV PROMs to produce the ALU operations are shown in Table 25-17. The table shows the movement of the string ABCDEFGH. The first byte(s) have already been moved with a ZMSTRT.

ZMSHRT

ZMSHRT is used when there are less than 4 bytes in the destination. The aligned source word is passed on the A leg and the destination is passed on the B leg. ZMSHRT is used to mask merge the two character strings based on destination alignment and number of bytes to be written. ZMSHRT ALU operations are illustrated in Table 25-18. In this table, the string being moved is either A, AB, or ABC, and it is being moved to line up with the Q in the destination.

ZMEXIT

TABLE 25-17. ZMFAST ALU Modes For Character Instructions

ZFF(0:3)	Desired DST	A SRC	B SRC	ALU Mode	Result
0 0 0 0	EFGH	ABCD	EFGH	(TB:TB, TB:TB,)	EFGH
0 0 0 1	EFGH	EFGD	XXXH	(TA:TA, TA:TB,)	EFGH
0 0 1 0	EFGH	EFCD	XXGH	(TA:TA, TB:TB,)	EFGH
0 0 1 1	EFGH	EBCD	XFGH	(TA:TB, TB:TB,)	EFGH
0 1 0 0	DEFG	DABC	HEFG	(TA:TB, TB:TB,)	DEFG
0 1 0 1	DEFG	XABC	DEFG	(TB:TB, TB:TB,)	DEFG
0 1 1 0	DEFG	DEFC	HXXG	(TA:TA, TA:TB,)	DEFG
0 1 1 1	DEFG	DEBC	HXFG	(TA:TA, TB:TB,)	DEFG
1 0 0 0	CDEF	CDAB	GHEF	(TA:TA, TB:TB,)	CDEF
1 0 0 1	CDEF	CXAB	GDEF	(TA:TB, TB:TB,)	CDEF
1 0 1 0	CDEF	XXAB	CDEF	(TB:TB, TB:TB,)	CDEF
1 0 1 1	CDEF	CDEB	GHXF	(TA:TA, TA:TB,)	CDEF
1 1 0 0	BCDE	BCDA	FGHE	(TA:TA, TA:TB,)	BCDE
1 1 0 1	BCDE	CBXA	FGDE	(TA:TA, TB:TB,)	BCDE
1 1 1 0	BCDE	BXXA	FCDE	(TA:TB, TB:TB,)	BCDE
1 1 1 1	BCDE	XXXA	BCDE	(TB:TB, TB:TB,)	BCDE

TABLE 25-18. ZMSHRT ALU Modes For Character Instructions

ZFF 0 1 4 5	# Bytes	B DST	A SRC	ALU Mode	Result
0 0 0 0	0	QRST	ABCX	(TB:TB, TB:TB,)	QRST
0 0 0 1	1	QRST	ABCX	(TA:TB, TB:TB,)	ARST
0 0 1 0	2	QRST	ABCX	(TA:TA, TB:TB,)	ABST
0 0 1 1	3	QRST	ABCX	(TA:TA, TA:TB,)	ABCT
0 1 0 0	0	PQRS	XABC	(TB:TB, TB:TB,)	PQRS
0 1 0 1	1	PQRS	XABC	(TB:TA, TB:TB,)	PARS
0 1 1 0	2	PQRS	XABC	(TB:TA, TA:TB,)	PABT
0 1 1 1	3	PQRS	XABC	(TB:TA, TA:TA,)	PABC
1 0 0 0	0	OPQR	CXAB	(TB:TB, TB:TB,)	OPQR
1 0 0 1	1	OPQR	CXAB	(TB:TB, TA:TB,)	OPAR
1 0 1 0	2	OPQR	CXAB	(TB:TB, TA:TA,)	OPAB
1 0 1 1	3	OPQR	CXAB	(TB:TB, TA:TA,)	OPAB
1 1 0 0	0	NOPQ	BCXA	(TB:TB, TB:TB,)	NOPQ
1 1 0 1	1	NOPQ	BCXA	(TB:TB, TB:TA,)	NOPA
1 1 1 0	2	NOPQ	BCXA	(TB:TB, TB:TA,)	NOPA
1 1 1 1	3	NOPQ	BCXA	(TB:TB, TB:TA,)	NOPA

ZMEXIT merges the last source data to be written with the destination data which needs to be preserved. It looks at destination alignment, and number of bytes to write. Table 25-19 illustrates the actions taken in this mode. In the table, we are at the end of a string which ends in Q, QR, QRS, or QRST.

The bytes left column is different from the value given by ZFF{4:5}+. The reasoning for this is as follows: Take the fifth line in the table as an example (ZFF{0:1}+ = 01, ZFF{4:5}+ = 00). The original number of bytes to be moved was some multiple of 4. On the first store to memory (ZMSTRT) only 3 bytes could be moved because of the destination alignment. All subsequent moves (ZMFAST) stored 4 bytes. Therefore, at the end of the transfer there is one byte of source string left over, which is merged with the destination.

TABLE 25-19. ZMEXIT ALU Modes For Character Instructions

ZFF	bytes	B	A		
0 1 4 5	left	DST	SRC	ALU Mode	Result
0 0 0 0	0	WXYZ	QRST	(TB:TB, TB:TB,)	WXYZ
0 0 0 1	1	WXYZ	QRST	(TA:TB, TB:TB,)	QXYZ
0 0 1 0	2	WXYZ	QRST	(TA:TA, TB:TB,)	QRYZ
0 0 1 1	3	WXYZ	QRST	(TA:TA, TA:TB,)	QRSZ
0 1 0 0	1	WXYZ	QRST	(TA:TB, TB:TB,)	QXYZ
0 1 0 1	2	WXYZ	QRST	(TA:TA, TB:TB,)	QRYZ
0 1 1 0	3	WXYZ	QRST	(TA:TA, TA:TB,)	QRSZ
0 1 1 1	4	WXYZ	QRST	(TA:TA, TA:TA,)	QRST
1 0 0 0	2	WXYZ	QRST	(TA:TA, TB:TB,)	QRYZ
1 0 0 1	3	WXYZ	QRST	(TA:TA, TA:TB,)	QRSZ
1 0 1 0	4	WXYZ	QRST	(TA:TA, TA:TA,)	QRST
1 0 1 1	1	WXYZ	QRST	(TA:TB, TB:TB,)	QXYZ
1 1 0 0	3	WXYZ	QRST	(TA:TA, TA:TB,)	QRSZ
1 1 0 1	4	WXYZ	QRST	(TA:TA, TA:TA,)	QRST
1 1 1 0	1	WXYZ	QRST	(TA:TB, TB:TB,)	QXYZ
1 1 1 1	2	WXYZ	QRST	(TA:TA, TB:TB,)	QRYZ

25.7.2.3 ZMV Examples

As advertised, this section will attempt to show how all the special purpose character hardware is supposed to play together to get the job done. What follows is an oversimplified version of a ZMV algorithm. Any resemblance between this algorithm and the actual one is strictly coincidental. The algorithm given here is perfectly adequate for our discussion, but microcode may have efficiencies built into it not shown here. Also, this algorithm is not guaranteed to be accurate in every case, but is meant for illustrative purposes only.

1. Load the ZFF flops with source and destination string alignment information. Also load the number of bytes left to process into a counter and the ZFF flops.

2. Fetch the first word of the source string. Align it in the barrel shifter using the [ZMV] BDI directive. Fetch the first word of the destination string.

3. If there are less than 4 characters to move go to step 10.

4. Do a ZMSTRT ALU operation. If another read of the source string is not necessary (indicated by ZBIT2), go to step 6.

5. Fetch the next source string, [ZMV] it, and perform a ZMFAST ALU operation to produce an updated source string.

6. Do a ZMSTRT ALU operation and write the result to the destination. Decrement the number of characters left counter by (4 - destination alignment).

7. Fetch the next source string. [ZMV] it. Perform a ZMFAST ALU operation to produce an updated source string.

8. If there are 4 or less characters left to move got to step 15.

9. Write the updated source string to next destination string. Decrement the number of characters remaining counter by 4. Go to step 7.

10. Perform a ZMSHRT ALU operation. If another read of the source string is not necessary (indicated by ZBIT2), go to step 12.

11. Fetch the next source string, [ZMV] it, and perform a ZMFAST ALU operation to produce an updated source string.

12. Perform a ZMSHRT ALU operation and write the results to the destination.

13. If the number of characters to be moved straddles a 32-bit boundary (indicated by ZSTRD), fetch the next destination string. Otherwise end, go to the next instruction.

14. Fetch the next source string, [ZMV] it, and perform a ZMFAST ALU operation to produce an updated source string. Perform a ZMEXIT ALU operation and write the results to memory. End, go to the next instruction.

15. Fetch the next (last) destination string. Perform a ZMEXIT ALU operation and write the results to memory. End, go to the next instruction.

The step numbers shown in the outline of the algorithm are carried through the examples that follow for reference.

<u>ZMV Example, 3 Characters</u>

The first example shows the execution flow of a ZMV instruction which moves three characters from source address '1000 alignment 3 to destination address '2000 alignment 1.

```
                 ___________________________________
                |   |   |   |   ||   |   |   |   |
Source '1000    | Z | R | V | M || D | L | P | Q |
                |___|___|___|___||___|___|___|___|
                              ↑
                              |

                 ___________________________________
                |   |   |   |   ||   |   |   |   |
Dest   '2000    | # | ↑ | $ | ! || % | & | * | ? |
                |___|___|___|___||___|___|___|___|
                      ↑
                      |
```

1. Load the ZFF flops and bytes remaining counter. ZFF = 01 11 11. (ZFF{0:1}+ =

destination alignment, ZFF{2:3}+ = source alignment, ZFF{4:5}+ = number of bytes to move.) Bytes remaining = 3.

2. Fetch first source string. This is the 32-bit word at address '1000, ZRVM. Align using [ZMV] directive in barrel shifter. From Table 25-6, 8th line (ZFF{0:3} = 0111), ZRVM becomes VMZR. Fetch first destination string. This is the 32-bit word at address '2000, #^$!.

3. There are less than 4 characters to move, go to step 10.

10. Perform a ZMSHRT ALU operation and check ZBIT2 to make sure we have enough data. From Table 25-11, last line (ZFF{2:5}+ = 1111), ZBIT2 is set. Another read is necessary.

11. Fetch the next source string. This is the 32-bit word at address '1002, DLPQ. Align via [ZMV], producing PQDL. (The ZFF bits haven't changed, so use the same line of Table 25-6.) Perform a ZMFAST ALU operation. From Table 25-17, 8th line (ZFF{0:3}+ = 0111), the updated source string VMDL is produced.

12. Perform a ZMSHRT ALU operation. From Table 25-18, 8th line (ZFF{0:1}+ = 01, ZFF{4:5}+ = 11), The string #MDL is produced. Write this result to the destination.

```
              _________________________________________
              |   |   |   |   ||   |   |   |   |
Source '1000  | Z | R | V | M || D | L | P | Q |
              |___|___|___|___||___|___|___|___|
                            ↑
                            |

              _________________________________________
              |   |   |   |   ||   |   |   |   |
Dest   '2000  | # | M | D | L || % | & | * | ? |
              |___|___|___|___||___|___|___|___|
                    ↑
                    |
```

13. There is no straddling of the 32-bit boundary, as indicated by ZSTRD (from Table 25-10, 8th line (ZFF{0:1}+ = 01, ZFF{4:5}+ = 11)). End, go to the next instruction.

ZMV Example, 9 Characters

This example moves 9 characters from source address '1000 alignment 2 to destination address '2000 alignment 3. In the discussion, the _ character is used as a place holder for the space character in the string to be moved.

```
              _________________________________________________________________
              |   |   |   |   ||   |   |   |   ||   |   |   |   |
Source '1000  | X | X | R | . ||   | P | A | R || R | O | W | X |
              |___|___|___|___||___|___|___|___||___|___|___|___|
                        ↑
                        |

              _________________________________________________________________
              |   |   |   |   ||   |   |   |   ||   |   |   |   |
Dest   '2000  |   |   |   | S || C | U | M | B || A | L | L |   |
              |___|___|___|___||___|___|___|___||___|___|___|___|
                          ↑
                          |
```

1. Load the ZFF flops and bytes remaining counter. ZFF = 11 10 01. (ZFF{0:1}+ = destination alignment, ZFF{2:3}+ = source alignment, ZFF{4:5}+ = number of bytes to move.) Bytes remaining = 9.

2. Fetch first source string. This is the 32-bit word at address '1000, XXR. . Align using [ZMV] directive in barrel shifter. From Table 25-6, 15th line (ZFF{0:3} = 1110), XXR. becomes .XXR . Fetch first destination string. This is the 32-bit word at address '2000, ____S .

3. There are more than 4 characters to move.

4. Do a ZMSTRT ALU operation. From Table 25-16, 15th line (ZFF{0:3}+ = 1110), ZBIT2 is not set. Another read of the source is not needed, go to step 6.

6. Do a ZMSTRT ALU operation and write the result to the destination. From Table 25-16, last line (ZFF{0:1}+ = 11), the string ____R is produced. Decrement the bytes remaining counter by (4 - destination alignment). The destination alignment (ZFF{0:1}+) is 3, so decrement the bytes remaining counter by 1, leaving it equal to 8.

```
               _________________________________________________
              |   |   |   |   ||   |   |   |   ||   |   |   |   |
Source '1000  | X | X | R | . ||   | P | A | R || R | O | W | X |
              |___|___|___|___||___|___|___|___||___|___|___|___|
                        ↑
                        |

               _________________________________________________
              |   |   |   |   ||   |   |   |   ||   |   |   |   |
Dest   '2000  |   |   |   | R || C | U | M | B || A | L | L |   |
              |___|___|___|___||___|___|___|___||___|___|___|___|
                            ↑
                            |
```

7. Fetch the next source string. This is the 32-bit word at '1002, _PAR . Alignment via [ZMV] (the ZFF bits haven't changed, still at line 15 of Table 25-6) produces R_PA . Perform a ZMFAST ALU operation. From Table 25-17, line 15 (ZFF{0:3}+ = 1110), this produces the updated source string ._PA .

8. There are more than 4 characters left to process.

9. Write the updated source string to the next destination string. This is the 32-bit word at address '2002. Decrement the bytes remaining counter by 4, leaving it equal to 4.

```
               _________________________________________________
              |   |   |   |   ||   |   |   |   ||   |   |   |   |
Source '1000  | X | X | R | . ||   | P | A | R || R | O | W | X |
              |___|___|___|___||___|___|___|___||___|___|___|___|
                        ↑
                        |
                 _______________________________________________________
                |   |   |   |   ||   |   |   |   ||   |   |   |   |
Dest   '2000    |   |   |   | R || . |   | P | A || A | L | L |   |
                |___|___|___|___||___|___|___|___||___|___|___|___|
                              ↑
                              |
```

7. Fetch the next source string. This is the 32-bit word at address '1004, ROWX . Aligning it via [ZMV] produces XROW . Performing a ZMFAST ALU operation produces the updated source string RROW .

8. There are 4 bytes or less remaining to process, go to step 15.

15. Fetch the next destination string. This is the 32-bit word at address '2004, ALL_ . Perform a ZMEXIT ALU operation. From Table 25-19, 14th line (ZFF{0:1}+ = 11, ZFF{4:5}+ = 01), the string RROW is produced. Write this result to the destination string, the 32-bit word at address '2004. End, go to the next instruction.

```
                 _______________________________________________________
                |   |   |   |   ||   |   |   |   ||   |   |   |   |
Source '1000    | X | X | R | . ||   | P | A | R || R | O | W | X |
                |___|___|___|___||___|___|___|___||___|___|___|___|
                          ↑
                          |

                 _______________________________________________________
                |   |   |   |   ||   |   |   |   ||   |   |   |   |
Dest   '2000    |   |   |   | R || . |   | P | A || R | R | O | W |
                |___|___|___|___||___|___|___|___||___|___|___|___|
                              ↑
                              |
```

25.7.3 Decimal Instructions

The decimal instructions operate under V or I mode only. There are some similarities between decimal instructions and floating point instructions, in that both the concept of a decimal point. Decimal instructions operate on decimal ASCII characters or Binary Coded Decimal (BCD) digits, and thus bear a closer resemblance to character instructions.

Decimal instructions can be divided into three different groups:

1. decimal arithmetic instructions
2. decimal string instructions
3. decimal conversion instructions

Some examples of the decimal arithmetic instructions:

Decimal Add (XAD)
: Add the contents of two decimal fields and store the results in the destination field.

Decimal Multiply (XMP)
: Multiply the contents of the source and destination fields and store the result in the destination field.

Decimal Divide (XDV)
: Divide the contents of the destination field by the contents of the source field and store the result and the remainder in the destination field.

Some examples of decimal string instructions:

Decimal Move (XMV)
: Move the contents of the source field into the destination field.

Decimal Compare (XCM)
: Compare the contents of the source and destination field and set the condition codes depending on the outcome of the compare.

Decimal Edit (XED)
: Edit a decimal string under control of an edit subprogram.

Some examples of decimal conversion instructions:

Binary to Decimal Conversion (XBTD)
: Convert a binary number contained in a register to a decimal number and store the result in a memory location.

Decimal to Binary Conversion (XDTB)
: Converts a decimal number in memory to a binary number and stores the result in a register.

Decimal instructions need more information than can be contained in one opcode. To use these instructions a control word must be provided in the L register (V mode), or register 2 (I mode). The control word format is shown in the Table 25-20. Table 25-21 further defines the control word fields.

TABLE 25-20. Decimal Instruction Control Word Format

```
General Setup - VMODE:

        EAFA    0,source_address
        EAFA    1,destination_address
        LDL     controlword_address
        operation

General Setup - IMODE:

        EAFA    0,source_address
        EAFA    1,destination_address
        L       2,controlword_address
        operation
```

Control Word Definition

```
 _________________________________   _________________________________
|a|a|a|a|a|a|0|0|b|c|0|d|e|f|f|f|   |g|g|g|g|g|g|h|h|h|h|h|h|h|i|i|i|
 ---------------------------------   ---------------------------------
 0         0   0 1   1 1 1   1       0         0 0           1 1   1
 1         6   9 0   2 3 4   6       1         6 7           3 4   6

               H                                  L
```

a	=	source field length (digits)
b	=	source field negation flag
c	=	destination field negation flag (XAD, XMP, XDV, XCM only)
d	=	force sign positive flag (XAD, XMP, XDV, XMV)
e	=	round flag (XMV only)
f	=	source field type (see below)
g	=	destination field length (digits)
h	=	scale differential (XAD, XMV, XCM only)
i	=	destination field type

TABLE 25-21. Decimal Field Type Definitions

```
 ______         ______
|f|f|f|        |i|i|i|
 ______         ______
 1   1          1   1
 4   6          4   6
```

bit encoding		
0 0 0	–	leading separate sign
0 0 1	–	trailing separate sign
0 1 1	–	packed decimal (requires odd # digits)
1 0 0	–	leading embedded sign
1 0 1	–	trailing embedded sign
X X 1	–	trailing sign
X 1 1	–	packed
1 X X	–	embedded sign

Embedded Sign Characters

Digit	Positive	Negative
0	0 + sp {	– }
1	1 A	J
2	2 B	K
3	3 C	L
4	4 D	M
5	5 E	N
6	6 F	O
7	7 G	P
8	8 H	Q
9	9 I	R

25.7.3.1 The Decimal Instruction Game Plan

Some general comments about decimal instructions are in order before launching into a description of the hardware. As mentioned above, decimal data may be stored in memory as either ASCII characters or as BCD digits. The first format is known as "unpacked", while the second is known as "packed". An ASCII digit is an 8-bit data element, and thus one digit can fit in a byte. A BCD digit is a 4-bit data element, allowing two digits to fit in a byte. The BCD digits are therefore more compact, more digits can be "packed" into a given space. (Now that the two terms have been defined I will drop the quotation marks.) Decimal data is almost always dealt with in packed format internally in the E unit. Unpacked data is packed when it comes in. Packed data is unpacked, if necessary, before storage.

Memory can be thought of as one long string of continuous bytes, starting with byte 0 (address 0) on the left and with increasing addresses proceeding to the right. Decimal numbers are stored in memory with the most significant digit on the left, at the lower address, and the lesser significant digits proceeding to the right at higher addresses. Character strings are thought of in a similar fashion. Although no particular character has any more significance than another (from a data processing point of view - spelling is out of the scope of this discussion), we took the first character in the string to be at the lowest address and proceeded to the higher addresses in the course of our processing. Decimal numbers can't be handled in this fashion. Decimal arithmetic instructions (XAD, for example) require that operations proceed from least significant digit to most significant digit (or from the highest address down to the lowest address). In short, character strings are processed from right to left (in English, if you will), while decimal strings are processed from left to right (in Hebrew). (As of this writing there are no instructions that process data from top to bottom, in Chinese.) Decimal strings must be processed right to left. There is no reason character strings couldn't be processed right to left also. However, in the English speaking world, processing character strings from right to left would is counterintuitive. Oh, to be in Israel, now that decimal instructions are here.

The tables for decimal string processing are frequently the mirror images of those for character string processing because of the Hebrew nature of the job. The most important difference is that the alignment information given for decimal strings refers to the alignment of the least significant digit, not the most significant digit.

As with the character instructions discussed earlier, the hardware pieces necessary to process decimal instructions will be described first. After this, an illustrative example will be given to show how all the pieces play together.

25.7.3.2 ZMV PROMs

The ZMV PROMs create the decimal jump conditions. Like the character jump conditions ZBIT1 and ZBIT2, the decimal jump conditions are affected by the ALU mode which appears in the instruction before the jump. XBIT{1:2}+ are determined in a similar fashion. Tables 25-22, 25-23, 25-24, and 25-25 show what XBIT{1:2}+ are for each decimal mode.

- For ADJUST, the ZMV PROMs look at the destination alignment.
- For OBTAIN, the ZMV PROMs look at the destination alignment and number of bits to be processed.
- For UNLPCK, the ZMV PROMs look at the source and destination alignment.
- For UNLUNP, the ZMV PROMs look at the source alignment.

TABLE 25-22. ADJUST Decimal Jump Conditions

Mode = 00

XFF(0:1)	DST	XBIT(1:2)
0 0	4XXX	0 0
0 1	34XX	0 1
1 0	234X	1 0
1 1	1234	1 1

TABLE 25-23. OBTAIN Decimal Jump Conditions

Mode = 01

XFF 0 1 4 5	DST	BYTES	XBIT(1:2)	
0 0 0 0	4XXX	0	0 0	
0 0 0 1	4XXX	1	0 0	
0 0 1 0	4XXX	2	1 1	default
0 0 1 1	4XXX	3	1 1	default
0 1 0 0	34XX	0	0 0	
0 1 0 1	34XX	1	1 1	
0 1 1 0	34XX	2	0 0	
0 1 1 1	34XX	3	1 1	default
1 0 0 0	234X	0	0 0	
1 0 0 1	234X	1	1 0	
1 0 1 0	234X	2	1 1	
1 0 1 1	234X	3	0 0	
1 1 0 0	1234	0	0 0	
1 1 0 1	1234	1	0 1	
1 1 1 0	1234	2	1 0	
1 1 1 1	1234	3	1 1	

TABLE 25-24. UNLPCK Decimal Jump Conditions

Mode = 10

XFF(0:3)	Alignment DST	Alignment SRC	XBIT(1:2)
0 0 0 0	4XXX	4XXX	0 0
0 0 0 1	4XXX	34XX	1 1
0 0 1 0	4XXX	234X	1 0
0 0 1 1	4XXX	1234	0 1
0 1 0 0	34XX	4XXX	0 1
0 1 0 1	34XX	34XX	0 0
0 1 1 0	34XX	234X	1 1
0 1 1 1	34XX	1234	1 0
1 0 0 0	234X	4XXX	1 0
1 0 0 1	234X	34XX	0 1
1 0 1 0	234X	234X	0 0
1 0 1 1	234X	1234	1 1
1 1 0 0	1234	4XXX	1 1
1 1 0 1	1234	34XX	1 0
1 1 1 0	1234	234X	0 1
1 1 1 1	1234	1234	0 0

TABLE 25-25. UNLUNP Decimal Jump Conditions

Mode = 11

XFF(2:3)	SRC	XBIT(1:2)
0 0	4XXX	0 0
0 1	34XX	0 1
1 0	234X	1 0
1 1	1234	1 1

25.7.3.3 Pack PROMs

The Pack PROMs take data off of BBH and BBL, convert it from an unpacked decimal format to a packed decimal format, and input it on the A leg of the high side ALU. These input are the JUNK inputs to the PEALU chips. This transformation of 4 decimal digits in 32-bits to 4 decimal digits in 16-bits is done because it is much easier to do arithmetic operations and string manipulation in packed format.

No information is lost in the packing. The bits being thrown away are the bits which make a BCD digit an ASCII digit. These bits are easily recreated.

The Pack PROMs also check the sign of an unpacked decimal string if IAC SIGN is used. The PROM looks at RDE01+ to determine if the sign is in the upper or lower byte of BBH.

If RDE01+ is inactive, the sign is in the high byte, and the pack PROMs will put a hex 2 on the most significant nibble if the sign is negative. If RDE01+ is active, the sign is in the lower byte, and the pack PROMs will put a hex 2 on the second nibble if the sign is negative.

NOTE: There can never be a cache miss during this operation. To insure this, the microcode accesses cache before it uses the Pack PROMs.

25.7.3.4 Decimal Corrector PROMs – DALU

After packing is accomplished, the microcode will typically put the data into an excess-6 format. This means that a hex 6 will be added to each of the 4-bit decimal digits. Again, this is an aid in arithmetic operations. If two decimal digits are added together with one of them in excess-6 format, any carry will happen automatically. This eliminates messy decimal-to-binary, binary-to-decimal conversions in the arithmetic algorithms. For example:

```
                   Decimal Operation

                     109
                   +  5
                    ----
                     114

                                         Microcode
              ASCII 8         PACK       Excess-6    DALU
              Digits          PROMs      Operation   PROMs

 source       B0B1 B0B9       0109        676F
 destination  B0B0 B0B5       0005        0005
              ---------       ----        ----       ----
                                          6774       0114
```

The DALU PROMs work in conjunction with the BDI during an XADP or XADU BDI mode. The PROMs are used to subtract out the excess 6 which the decimal algorithm placed into the packed data. If the destination is unpacked, the microcode uses XADU. The High and Low side BDI chips will automatically unpack the data in this mode. For a packed destination XADP is used, which does not involve the BDI chips. Tables 25-26 and 25-27 show the inputs to the PROMs, outputs of the PROMs, and, in the case of XADU, the output of the BDI chip.

The DALU PROMs also are used to create DECMxNEQ-, which is a possible link bit value. (X in this case can be 1, 2, 3, or 4, depending on the nibble which was 0.) The four signals are NAND'ed together to form the signal DECMNEQ+, which is used for zero tracking so that the microcode can tell if it skipped over zeros. Table 25-28 shows how this signal is generated.

TABLE 25-26. XADP BDI mode

ALHCOUT+	ALH(01:04) ALH(05:08) ALH(09:12) ALH(13:16)	DECM(01:04) DECM(05:08) DECM(09:12) DECM(13:16)	
1	Q	Q	**
0	0	A	
0	1	B	
0	2	C	
0	3	D	
0	4	E	
0	5	F	
0	6	0	
0	7	1	
0	8	2	
0	9	3	
0	A	4	
0	B	5	
0	C	6	
0	D	7	
0	E	8	
0	F	9	

** Note: Q represents any number

TABLE 25-27. XADU BDI mode

ALHCOUT+	ALH(01:04) ALH(05:08) ALH(09:12) ALH(13:16)	DECM(01:04) DECM(05:08) DECM(09:12) DECM(13:16)	BDIH(01:08) BDIH(09:16) BDIL(01:08) BDIL(09:16)	
1	Q	Q	BQ	**
0	0	A	BA	
0	1	B	BB	
0	2	C	BC	
0	3	D	BD	
0	4	E	BE	
0	5	F	BF	
0	6	0	B0	
0	7	1	B1	
0	8	2	B2	
0	9	3	B3	
0	A	4	B4	
0	B	5	B5	
0	C	6	B6	
0	D	7	B7	
0	E	8	B8	
0	F	9	B9	

** Note: Q represents any number.

The hex B in the high nibble in ASCII 8
will be a hex 3 if you are in ASCII 7 mode

TABLE 25-28. DECMxNEQ- Link Bit Input

XBIT(1:2)	LINK	ALHCOUT	ALH(01:16)	DECMxNEQ-
00	0	X	XXXX	0
00	1	0	6666	1
00	1	0	<>6666	0
00	1	1	0000	1
00	1	1	<>0000	0
01	0	X	XXXX	0
01	1	0	XXX6	1
01	1	0	<>XXX6	0
01	1	1	XXX0	1
01	1	1	<>XXX0	0
10	0	X	XXXX	0
10	1	0	XX66	1
10	1	0	<>XX66	0
10	1	1	XX00	1
10	1	1	<>XX00	0
11	0	X	XXXX	0
11	1	0	X666	1
11	1	0	<>X666	0
11	1	1	X000	1
11	1	1	<>X000	0

25.7.3.5 XMODE PROMs

These PROMs are used to control the decimal ALU modes.

The ALU modes associated with decimal are:

MODE	RCMALU(06:08)
ADJUST	1 0 0
OBTAIN	1 0 1
UNLPCK	1 1 0
UNLUNP	1 1 1

ADJUST is identical to the floating point ADJUST discussed earlier. It is used to align the decimal points of the operands if the scale differential is nonzero.

OBTAIN is the decimal version of the character ALU mode ZMFAST.

UNLPCK and UNLUNP are the decimal versions of the character ALU mode ZMSHRT. UNLPCK operates on packed data, while UNLUNP operates on unpacked data.

These ALU modes behave differently for different XFF bits. Tables 25-32, 25-31, 25-29, and 25-30 illustrate the operation of these modes. The "/" character is used to differentiate a nibble operation within an 8-bit PEALU slice from a byte operation between two PEALU slices, which have been designated by the ":" character.

TABLE 25-29. UNLPCK ALU Mode, Unload Packed

XFF 0 1 4 5	B DST	A SRC	BYTES	ALU Modes ALH(01:08)	ALH(09:16)	Result
0 0 0 0	6789	1234	0	TA	TA	1234
0 0 0 1	6789	1234	1	TA/TB	TB	1789
0 0 1 0	6789	1234	2	TA/TB	TB	1789
0 0 1 1	6789	1234	3	TA/TB	TB	1789
0 1 0 0	6789	1234	0	TA	TA	1234
0 1 0 1	6789	1234	1	TB/TA	TB	6289
0 1 1 0	6789	1234	2	TA/TA	TB	1289
0 1 1 1	6789	1234	3	TA	TB	1289
1 0 0 0	6789	1234	0	TA	TA	1234
1 0 0 1	6789	1234	1	TB	TA/TB	6739
1 0 1 0	6789	1234	2	TB/TA	TA/TB	6239
1 0 1 1	6789	1234	3	TA	TA/TB	1239
1 1 0 0	6789	1234	0	TA	TA	1234
1 1 0 1	6789	1234	1	TB	TB/TA	6784
1 1 1 0	6789	1234	2	TB	TA	6734
1 1 1 1	6789	1234	3	TB/TA	TA	6234

TABLE 25-30. UNLUNP ALU Mode, Unload Unpacked

XFF 0 1 4 5	B DST	A SRC	BYTES	ALU Mode	Result
0 0 0 0	6789	1234	0	(TA,TA)	1234
0 0 0 1	6789	1234	1	(TA:TB,TB)	1789
0 0 1 0	6789	1234	2	(TA:TB,TB)	1789
0 0 1 1	6789	1234	3	(TA:TB,TB)	1789
0 1 0 0	6789	1234	0	(TA,TA)	1234
0 1 0 1	6789	1234	1	(TB:TA,TB)	6289
0 1 1 0	6789	1234	2	(TA:TA,TB)	1289
0 1 1 1	6789	1234	3	(TA,TB)	1289
1 0 0 0	6789	1234	0	(TA,TA)	1234
1 0 0 1	6789	1234	1	(TB,TA:TB)	6739
1 0 1 0	6789	1234	2	(TB:TA,TA:TB)	6239
1 0 1 1	6789	1234	3	(TA,TA:TB)	1239
1 1 0 0	6789	1234	0	(TA,TA)	1234
1 1 0 1	6789	1234	1	(TB,TB:TA)	6784
1 1 1 0	6789	1234	2	(TB,TA)	6734
1 1 1 1	6789	1234	3	(TB:TA,TA)	6234

TABLE 25-31. OBTAIN ALU Mode

XFF(0:3)	A SRC	B SRC	ALH (01:08)	ALH (09:16)	Result
0 0 0 0	2345	6789	TB	TB	6789
0 0 0 1	2345	6789	TA/TB	TB	2789
0 0 1 0	2345	6789	TA	TB	2389
0 0 1 1	2345	6789	TA	TA/TB	2349
0 1 0 0	2345	6789	TA	TA/TB	2349
0 1 0 1	2345	6789	TB	TB	6789
0 1 1 0	2345	6789	TA/TB	TB	2789
0 1 1 1	2345	6789	TA	TB	2389
1 0 0 0	2345	6789	TA	TB	2389
1 0 0 1	2345	6789	TA	TA/TB	2349
1 0 1 0	2345	6789	TB	TB	6789
1 0 1 1	2345	6789	TA/TB	TB	2789
1 1 0 0	2345	6789	TA/TB	TB	2789
1 1 0 1	2345	6789	TA	TB	2389
1 1 1 0	2345	6789	TA	TA/TB	2349
1 1 1 1	2345	6789	TB	TB	6789

TABLE 25-32. ADJUST ALU Mode

FALL00+	ALU Mode
0	(TB,TB,TB)
1	(TA,TA,TA)

25.7.3.6 Binary to Decimal PROMs – BTD

There is one Binary to Decimal PROM, which doubles as a decimal to binary PROM (see next section). It receives ALH(09:15) and a signal which tells it whether a decimal to binary conversion is being done. (If you are not doing a decimal to binary conversion it is assumed that you are doing a binary to decimal conversion.)

During a binary to decimal conversion the PROM takes the 7-bit input and converts it two BCD digits. These digits are passed through the H PBDI slice for storage.

25.7.3.7 Decimal to Binary PROMs – DTB

The two Decimal to Binary PROMs work in conjunction with the BTD PROM to convert a BCD number in the range 0 to 9999 into it's binary equivalent. The DTB PROMs work on the 2 most significant digits of the number. Therefore, they can be thought of as converting multiples of 100 from 100 to 9900. The outputs are passed through the E PBDI slice for storage. The conversion of 0 to 99 is done in the BTD PROM when the signal FDTBSIG+ is active. The two pieces of the conversion are added together to complete the conversion.

25.7.3.8 Introduction To Decimal Examples

The general execution of a decimal arithmetic instruction goes something like this:

1. Fetch the source and destination operands. Load the XFF flops with the alignment information. Figure out the signs of the operands based on the data type and compute the sign of the result. If the scale differential is nonzero, shorten the appropriate operand and update the XFF flops.
2. Fetch the source and/or destination operands and pack them if necessary.
3. Add excess 6 to one operand.
4. Do the operation.
5. Remove the excess 6 from the results. Unpack the result if necessary.
6. Store the result.
7. If more digits remain to be processed, got to step 2, Else end, go to next instruction.

Looks pretty simple, right ?

A few comments about some of the operations just mentioned:

Pack	If data is not packed in memory, it must be packed as it is brought into the E unit. Packing takes 4 digits per 32 bits and crams (packs) it into 4 digits per 16 bits. No information is lost during this operation because the bits being thrown away are the bits that make a digit an ASCII character.
Excess 6	Microcode add 6 to every packed digit in one operand during decimal arithmetic instructions. This makes decimal carry outs happen naturally in the hexadecimal world of the E unit. Obviously, the excess 6 must be removed before the result can be stored.
Unpack	Unpacking is the inverse of packing. Four BCD digits per 16 bits are converted into 4 ASCII characters per 32 bits. This is done by the barrel shifter, and can be done in the same microstep as excess 6 removal.

One other difficulty encountered in decimal instructions is the concept of a scale differential. The System Architecture Reference Guide claims decimal instruction operate only on decimal integers. The scale differential introduces a decimal point, which throws that claim out the window.

To illustrate the concept, consider two decimal numbers in memory, 9 and 29. Further suppose the scale differential of the two is equal to 1. This means that the second number is really 2.9, not 29.

```
    9             9
 + 29         + 2.9
```

```
                 ----           -----
                  38             11.9
```

When the scale differential is nonzero, the longer operand is "shortened" to align the decimal points. (This is similar to adjusting in floating point.) If the scale differential is negative, there are more destination digits than source digits. The destination is "shortened". If the scale differential is positive, there are more source digits than destination digits. The source is "shortened". When shortening the destination, a check is done to see if any nonzero digits are skipped over. If this is the case, the condition codes will not be set to EQ when exiting the microcode routine, causing a decimal exception.

25.7.3.9 Decimal Arithmetic Example -- XAD

The sign of each operand is controlled by the two sign control fields of the control word, so there is no need for a decimal subtract instruction. XAD covers both decimal add and subtract.

The scale differential can be considered to be equivalent to moving the decimal point: left if negative, right if positive. If there is a scale differential then the "shortening" of the source or destination is done first. For addition operations you shorten the source by shortening the source length register, updating the pointer to the least significant digit of the source, and latching XFF{2:3}+ with the new alignment information. For subtraction you check to see if there will be an initial borrow before you shorten the source. A borrow occurs when you skip over a nonzero digit while shortening the source. For example:

```
                         Borrow  Non-borrow
                         ------  ----------

destination                 9        9
source                     20       31
scale differential          1        1
effective source          2.0      3.1

                            9
                         - 2.0   no significant figures lost
                          ----
Prime Decimal Answer        7

                                     9
                                 - 3.1  need a borrow for the 1
                                  ----
                                   5.9  "real" result
Prime Decimal Answer               5    destination has only 1 digit
```

When the subtract is done borrowing, it updates the source length register, the pointer to the least significant digit, and XFF{2:3}+.

To shorten the destination you merely update the destination least significant digit pointer. The destination length and destination alignment are unchanged. Remember, if the destination is being shortened, microcode keeps track of nonzero digits which may be skipped.

Now that step 1 of the general decimal algorithm has been elaborated on, here is the example.

XAD Example

In this example we will be adding two 5 digit numbers as shown:

```
  12345
+ 67890
-------
  80235
```

The numbers are given in trailing separate sign format at the addresses shown. Each number has a length of 6, and the scale differential is 0.

```
                 _____________________________________
                |   |   |   |   ||   |   |   |   ||
Source   '1000  | 1 | 2 | 3 | 4 || 5 | + | 7 | 3 ||  Alignment = 1
                |___|___|___|___||___|___|___|___||
                                       ↑
                                       |

                 _____________________________________
                |   |   |   |   ||   |   |   |   ||
Dest     '2000  | 9 | 6 | 7 | 8 || 9 | 0 | + | 2 ||  Alignment = 2
                |___|___|___|___||___|___|___|___||
                                           ↑
                                           |
```

1. The XFF flops are loaded to facilitate the extraction of the sign information. XFF{0:1}+ = 10 (destination alignment), XFF{2:3}+ = 01 (source alignment), and XFF{4:5}+ = 10 (length). Fetching the source and destination operands and extracting the separate sign character shows that both operands are positive. An addition is to be done. Since two positive numbers are being added the result will be positive. The destination sign will not change. The two sign digits don't have to be added together, so we can move the alignment pointers to point to the least significant digits. This requires an update of the XFF flops. XFF = 01 00 10.

The scale differential is zero, so no further update of the source or destination pointers is needed.

```
                 _____________________________________
                |   |   |   |   ||   |   |   |   ||
Source   '1000  | 1 | 2 | 3 | 4 || 5 | + | 7 | 3 ||  Alignment = 0
                |___|___|___|___||___|___|___|___||
                                   ↑
                                   |

                 _____________________________________
                |   |   |   |   ||   |   |   |   ||
Dest     '2000  | 9 | 6 | 7 | 8 || 9 | 0 | + | 2 ||  Alignment = 1
                |___|___|___|___||___|___|___|___||
                                       ↑
                                       |
```

2. Fetch the source string at '1002, 5+73. Pack it to produce the string 5B73. (The ASCII

+ character becomes a B when packed.) Fetch the destination string at '2002, 90+2. Pack it to produce the string 90B2. Do an UNLPCK ALU operation to see whether the source and destination have equal alignment. If so, XBIT{1:2}+ will both be 0. From Table 25-24, 5th line, XBIT{1:2}+ = 01 in this example. This means the source least significant digit is one digit to the left of the destination least significant digit. (XBIT{1:2}+ is a measure of the difference between the two least significant digits, positive for source to the left, negative for source to the right.) Since the source is aligned to the left of the destination, more source digits are needed. Fetch the next source string from address '1000, 1234. Perform an OBTAIN ALU operation with the old source on the A leg and the new source on the B leg. From Table 25-31, 5th line, the updated source string is 5B74. We already know the source is aligned 1 digit to the left of the destination, so rotate the updated source string right 1 digit, producing the string 45B7. The source digit s are now aligned with the destination digits.

There may be digits in our strings which aren't to be added together. To handle this, perform an UNLPCK ALU operation with the destination string, 90B2, on the B leg, and all zeros on the A leg. From Table 25-29, 7th line, this produces the updated destination string 9000.

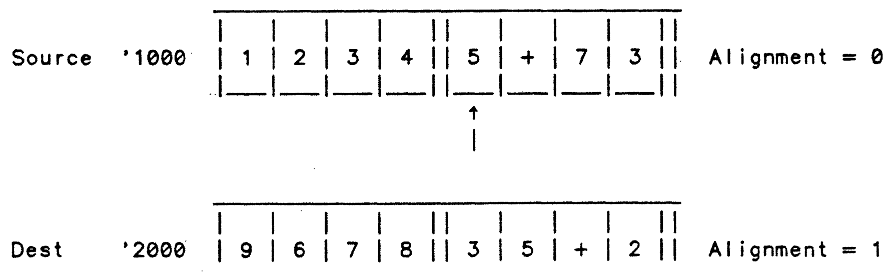
3. Add excess 6 to each nibble of the updated source string. This produces the string AB1D.

4. Add the two updated strings together. Remember that this addition produced a carry out for the next iteration.

```
  AB1D
+ 9000
------
 13B1D
```

5. Remove the excess 6 from the result. As shown by Table 25-26 or 25-27, only nibbles which don't produce carry outs need the excess 6 removed. In this case, the three least significant nibbles didn't produce a carry, while the most significant nibble did. The excess 6 removed result is therefore 35B7. Unpacking this result produces the ASCII characters 35+7. (Both operations can be done at once by using Table 25-27.)

6. Perform an UNLUNP ALU operation with the result on the A leg and the original destination string, 90+2, on the B leg. From Table 25-30, 7th line, this produces the string of ASCII characters 35+2. Write this result to the destination address '2002.



```
|___|___|___|___||___|___|___|___||
                       ↑
                       |
```

7. More digits remain to be processed. XFF{4:5}+ are only important during the first iteration of the loop, and must be = 00 on all subsequent iterations. Clear them and go to step 2.

2. The source string has been exhausted, the entire thing has already been fetched. Some of the left over digits may have already been used. To remove them, perform an OBTAIN ALU operation with the old source string, 1234, on the A leg, and all zeros on the B leg. This produces the updated source string 1230 (from Table 25-31, 5th line). Recalling from last time through the loop, this needs to be rotated right 1 digit, producing 0123. Fetch the nex destination string from address '2000, 9678.

3. Add excess 6 to each nibble of the updated source string, producing 6789.

4. Add the strings together with the carry out of the last iteration.

```
     1   <-------- Carry from last iteration
  6789
+ 9678
------
  FE02
```

5. Remove the excess 6 from the result and unpack it using Table 25-27. This produces the ASCII character string 9802.

6. Write the result to destination address '2000.

```
               _________________________________________
              |   |   |   |   ||   |   |   |   ||
Source  '1000 | 1 | 2 | 3 | 4 || 5 | + | 7 | 3 ||  Alignment = 0
              |___|___|___|___||___|___|___|___||
                                 ↑
                                 |

               _________________________________________
              |   |   |   |   ||   |   |   |   ||
Dest    '2000 | 9 | 8 | 0 | 2 || 3 | 5 | + | 2 ||  Alignment = 1
              |___|___|___|___||___|___|___|___||
                                     ↑
                                     |
```

7. No more digits to process. go to next instruction.

25.7.3.10 Decimal Conversion Example -- XBTD

In the simplest case of 16-bit conversions, we take the 16-bit binary number and divide it by 10000 to get the most significant decimal digit. Then the remainder will be divided by 100 to get two 7-bit pieces which correspond to two decimal digits each. Therefore, from a 16-bit binary number you end up with 5 BCD digits.

The first step is to divide the 16-bit number by a normalized 10000 and save the 2 bits of information which define the most significant decimal digit. (In a signed 16-bit number the maximum number is a hex 7FFF. This corresponds to a decimal 32767. The most significant digit is a 3, which takes only 2 bits.)

The 14-bit remainder from the first division is divided by 100. The 7-bit remainder from this operation is sent through the BTD PROMs by using the BTD ALU operation. These bits are stored in a scratch register. The 7-bit quotient is then sent through the BTD PROMs and put in a scratch register. The first 8 bits are added to the second 8 bits to produce the lowest 4 BCD digits.

For larger conversions (32 or 64-bits), at this point in each iteration of the loop you perform an OBTAIN ALU operation to mask/merge your new 4 digits with the last 4 digits that you created.

When you have completed all of the digits except the last one, you can mask/merge this digit directly since it will be less than 9, and therefore already is a BCD digit.

25.7.3.11 PMA Level Restrictions

Decimal Multiply XMP

For a Decimal Multiply, the user should always insure that there are at least (sourcelength) leading zeroes in the destination field before executing an XMP.

Decimal Divide XDV

The decimal divide instruction (XDV) requires both source and destination fields to be trailing embedded sign types. The destination field must contain at least (sourcelength) leading zeroes in order to insure room for both quotient and remainder. The quotient is returned in character positions 1 through (destlength-sourcelength) and the remainder in positions (destlength-sourcelength+1) through (destlength). Both quotient and remainder are trailing embedded signed fields. No overflow checking is done on this instruction.

25.8 System Timers

The E unit contains two system timers which keep track of process and wall clock time. They are called the Phantom Interrupt Code (PIC) timer and the microsecond timer. Both timers are clocked by a 1 MHZ clock, and cause fetch cycle traps when they overflow. The PIC timer is a 12-bit counter which overflows once every 4 milliseconds, while the microsecond timer is a 10-bit counter which overflows once every 1.024 milliseconds.

25.9 Trap Logic

There are two types of E units traps which may occur, fetch cycle traps and non-fetch cycle traps. Fetch cycle traps are traps that can occur only during a fetch cycle (the microsequencer is executing a RTN to fetch), while non-fetch cycle traps occur anytime.

There are times when we want to prevent traps from interrupting an instruction. Traps can be turned completely on or off under microcode control by the IACs ENTRAPS and DISTRAPS. DMx traps may be prevented during certain operations by the TR microcode field. Other traps may be turned off individually by software via the modals register.

25.9.1 Fetch Cycle Traps

Fetch cycle traps can only cause the microsequencer to vector at the boundary between instructions. The following are the possible fetch cycle traps:

ECCC	Error Correction Code Correctable, MC detected a correctable error during a memory read operation
MINKTRAP	Diagnostic processor request
ISSOFTPE	S unit soft cache parity error
EOI	End Of Instruction
EXTINT	EXTernal INTerrupt, requested by I/O controllers
PIC	PIC timer overflow condition
TIMER	microsecond timer overflow condition
DNETPE	decode net parity error

25.9.2 Non-Fetch Cycle Traps

Non fetch cycle traps allow the microsequencer to vector out of the middle of a PMA instruction and handle an event, and then resume execution of that instruction. The following are the possible non-fetch cycle traps:

ECCU	Error Correction Code Uncorrectable
RCCPE	RCC Parity Error, control store RAM parity error
MISMOD	MISsing memory MODule
DMx	Direct Memory transfer, requested by I/O controllers
MCHK	Machine CHecK, fatal parity error
ADRTRW	ADdRess TRap Write, sent from the S unit.
BCGAFFE	Branch Cache GAFFE
RXM	Restricted PMA instruction

25.10 Parity Reporting

The E unit checks parity on all operands entering the ALU unless specifically told not t: by microcode. It also collects the parity error reporting signals of the other functional units and combines them into one signal, MCHK+. (MCHK+ is clocked at CS7+ to report errors which occurred during the previous stage 8. It is displayed 1 beat later on the PDA.) If the MODALS are set to allow the reporting to software of such errors, the appropriate trap signals are sent to the PCU.

Odd parity is assumed over each byte of data. The selected data and its associated parity bits are sent to the ALU's B leg inputs, while the data only is sent to the A leg inputs. The ALU generates a parity error signal if it detects a B leg parity error. It also generates a a parity bit based on the A leg input. These generated parity bits are compared against the original parity bits. If there is a difference, an error signal is generated.

The E unit sends a code to the microcode trap handler to report on errors it detected. Table 25-33 shows the code and its interpretation. The E unit is only prepared to handle the first parity error. If two errors occur simultaneously or if a second error occurs, the parity code will indicate which error occurred first. In the case of two simultaneous errors, the error code is prioritized from top to bottom of Table 25-33. For example, If a BBH left error and a BBH right error occurred simultaneously, the error code would indicate a BBH left error.

TABLE 25-33. E Unit Parity Error Codes

Code	Interpretation
0	No error
1	BBH left error
2	BBH right error
3	BBL left error
4	BBL right error
5	BAH error
6	BAL error
7	BAE error

25.11 VLSI Usage

The ALU is implemented with seven PEALU slices. Each VLSI slice defines boundary, so the ALU data path is 56 bits wide. The regular data path uses 4 and the rest of the data path is used for multiplies and divides exclusively.

The barrel shifter implements the 48 bit data path using three PBDI slices, with 16 bits per slice.

The addressing of the register file is done by one PRFADR VLSI. All timing, as to when the addressing switches between source and destination modes, is done with external logic.

25.12 Critical Paths

The critical paths on the E unit include:

- Generation of jump conditions for the CS, 1 beat
- Partial product generation during multiply operations, 1 beat
- Partial quotient generation during divide operations, 1 beat
- Rounding operations, 1 beat
- Register file write cycle, 1 beat

Some of the jump conditions in the E unit don't meet the 1 beat requirement. These jump conditions (see list) require a TX= 1 in the microcode step prior to branching.

The first three jump conditions made the 1 beat limit for logical ALU operations, but didn't mae it for arithmetic operations. The time shown are the logical times.

Path	Time (ns)
CS75+A -> FALH02-	60.5

CS75+A -> FALH05+	61.0
TRI+A -> FALEEQ+	56.5

The following 35 jump conditions need the additional TX= 1. The times shown are for arithmetic ALU operations.

Path	Time (ns)	Comment
CS75+A -> FALHOV+	86.5	
CS75+A -> F24PWROF2+, FMIN1+, or F48PWROF2-	82.0	
CS75+A -> FALLENE+	78.0	
CS75+A -> FAL32EQ+	79.0	
CS75+A -> FAL48EQ+	83.0	
CS75+A -> FAL32NE+	79.0	
CS75+A -> FALLNE+	73.0	
CS75+A -> FALLLE+	80.0	
CS75+A -> FBA48EQ+	67.5	
CS75+A -> FALL00+ or FALL00+T	83.5	
CS75+A -> FALLOV+	80.5	
CS75+A -> F8IEXTRAP-	105.5	1.5 beat path
CS75+A -> FALLHBZ-	66.0	
CS75+A -> FALL02+	67.5	
CS75+A -> FALL16+	63.5	
CS75+A -> FALH04+	75.0	
CS75+A -> FALH02+T	68.0	
CS75+A -> FALH07+	74.0	
CS75+A -> FALH08+	77.0	
CS75+A -> FALH15+	73.0	
CS75+A -> FALH16+	73.5	
CS75+A -> FALH14+	73.0	
CS75+A -> FALH13+	75.0	
CS75+A -> FALH00+T or FALHLT+	75.0	
CS75+A -> FALHNE+ or FALHNE+T	76.0	
CS75+A -> FALHEQ+	79.5	
CS75+A -> FALHGT+	84.0	
CS75+A -> FALHLE+	86.3	
CS75+A -> FALHGE+	74.3	
CS75+A -> FALHCOUT+T	83.9	

25.13 9755 Comparisons

The 9955 E unit logic existed on two boards- the E1 and E2. Due to the use of VLSIs the logic fit on the E board only on the 4050. The 9755 did not have a barrel shifter. It did however have some muxing hardware for performing two, three or four bit shifts in one beat. The barrel shifter is essentially a large set of muxes capable of performing arbitrary number of shifts/rotates on the 48 bit data path. This makes it useful for normalizes and adjusts in floating point operations.

The ALUs in the 4150 have multiply and divide logic incorporated in them. This sets them apart from the ALUs of the 9755. Other than this difference, all other features are the same.

The PIC timer in the 9755 was implemented with BCD counters, while the 4150's PIC is implemented using binary counters. This difference required the addition of an initialization circuit to keep the PIC overflows at the desired 1 per 4 millisecond rate.

The 4150 uses a 56-bit ALU data path to support multiplication. The 9755 had a 48-bit data path.

The 9755 used a non-performing divide algorithm. The 4150 uses a non-restoring divide algorithm.

The 9755 implemented a 2-bit Booth's multiply. The 4150 implements a 3-bit Booth's multiply.

Floating point out-of-range detection is implemented in the PBDI on the 4150, and is implemented somewhat differently than the 9755's.

The 9755 had one RS, while the 4150 has three.

25.14 Partitioning

All E unit functions are implemented on the E board, which is discussed in Chapter 32.

26. Memory Controller Unit Detailed Description

26.1 Overview

The MC unit is responsible for the following functions:

- Writing, reading, and refreshing the memory array boards.
- Generating Error Correcting Code (ECC) check bits on memory writes and checking the returning data for ECC errors during memory reads
- Checking BD parity on data coming into the MC during CPU write operations and generating good parity on data returning from memory during memory reads
- Speeding up main memory write operations from the CPU's viewpoint through the use of a Write Buffer (WB).

The MC receives 26 Bus B (BB) bits from the Storage Management (S) unit which it uses as memory address bits. It both drives and receives all of Bus D (BD), which is used as data to/from memory or status bits sent from the MC to the rest of the CPU.

The main sections of the MC are:

- Address
- Data
- Write Buffer
- Memory Timer
- Refresh
- Status & Error Reporting
- Control

The basic goal behind the MC is to make main memory read and write operations occur as quickly as possible from the CPU's viewpoint. The MC contains a Write Buffer (WB) to assist in achieving this goal. It consists of 2 basic parts:

- The address portion of the WB holds addresses sent from the S unit for main memory writes and reads. It also contains status bits corresponding to data being transferred to/from memory. It attempts to smooth out memory operations by combining 16 and 32-bit memory write operations into 64-bit operations to reduce the number of memory accesses. By combining memory operations, the performance of the CPU is improved because of the higher availability of the memory bus.

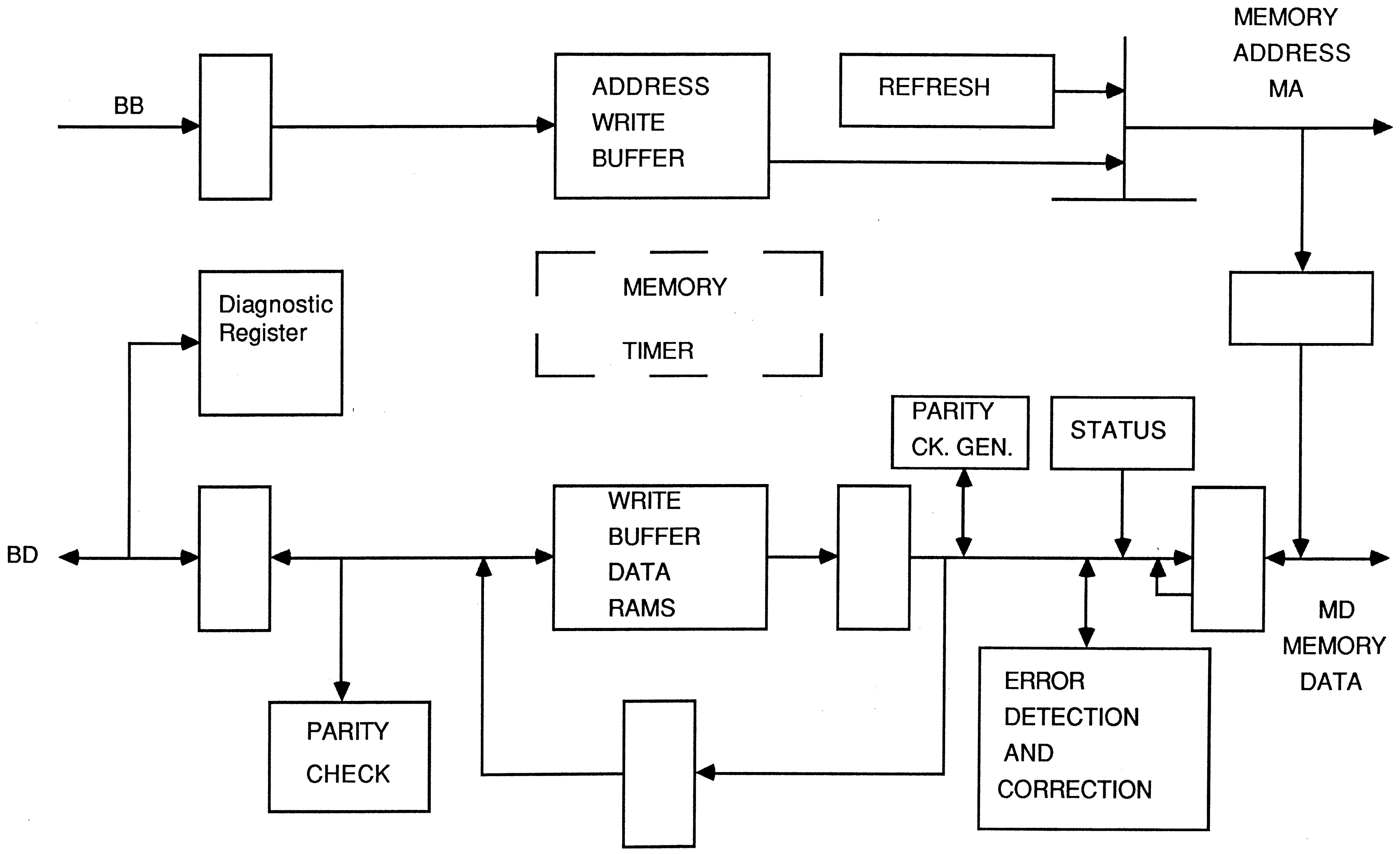


Figure 26-1 Block Diagram of Memory Controller

- The data section consists of RAMs which hold data destined for memory. ECC circuitry, which is associated with the WB data section, generates and checks ECC bits to improve the integrity of main memory.

Other major functional sections of the MC include:

- The Memory Timer (MT), which consists of a sequencer and PROMs whose data outputs are clocked and then sent to the memory boards as control signals or are utilized within the MC as control bits.

- The refresh circuitry initiates a memory refresh routine approximately every 16 microseconds and increments the refresh address each time so that it refreshes the entire memory array approximately every 4 milliseconds.

- A status register, accessible by microcode, holds status and error information.

26.2 Write Buffer

The Write Buffer (WB) is a fully associative 4 x 64-bit memory which takes 16 or 32 of data from the CPU and attempts to concatenate them into 32 or 64-bit quantities transfer to main memory. The goal is to smooth out traffic on the memory bus by doing fewer write operations, thereby providing higher memory availability to the CPU.

The WB has four main functional areas: the Data Write Buffer (DWB), Address Write Buffer (AWB), the Valid Write Buffer (VWB), and the Memory Timer OPerations Scheduler (MTOPS). Refer to Figure 26-2.

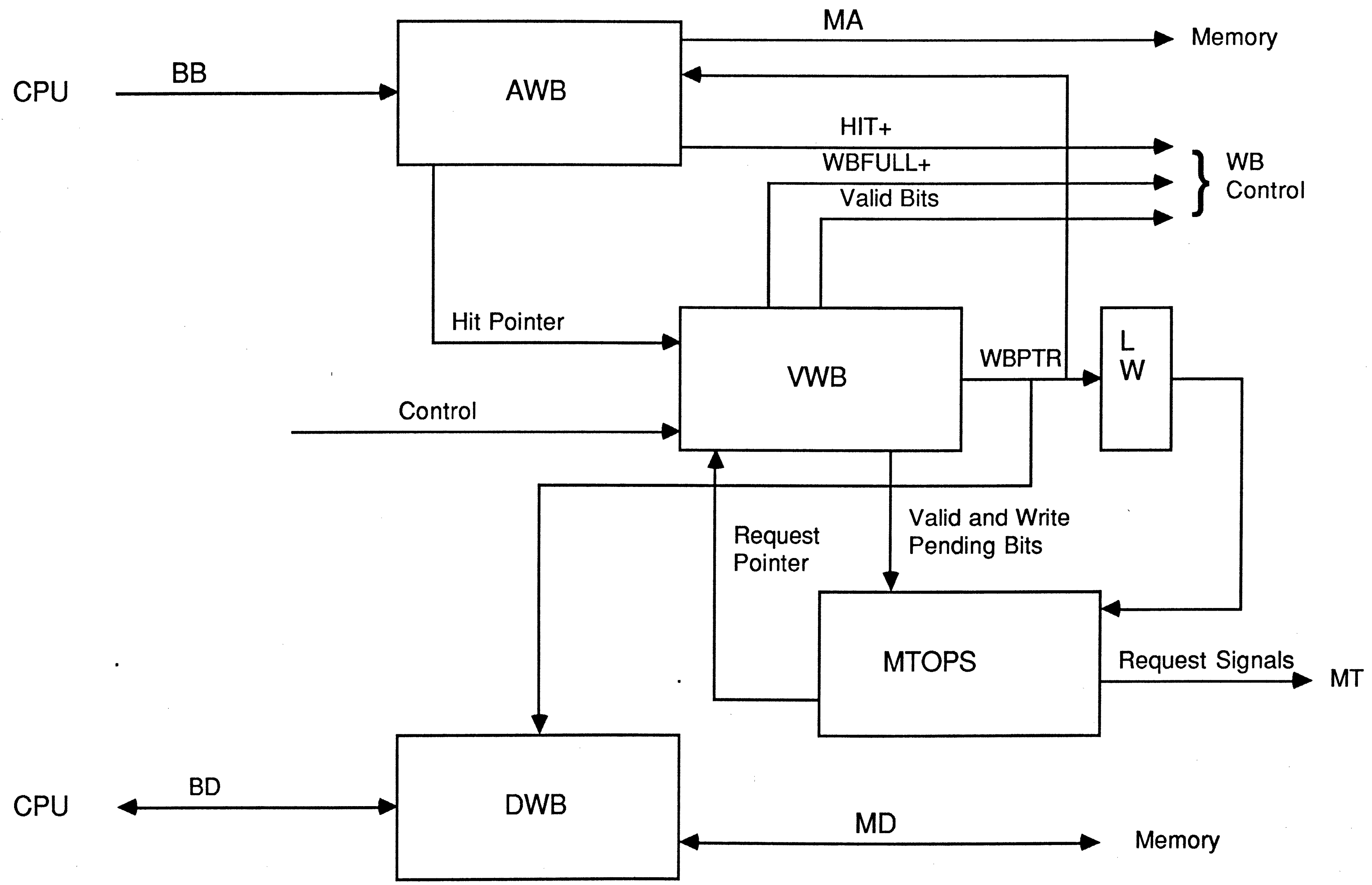


Figure 26-2 Write Buffer Block Diagram

26.2.1 Data Write Buffer

The DWB is an 8 x 32-bit memory. Each 16-bit half of any location is independently controlled by a separate write pulse. This allows any 16-bit write from the CPU to be handled easily.

The DWB can be thought of as a 4 x 64-bit memory in the following fashion: Each DWB location is designed to represent one 64-bit data word as it might appear in main memory. (Refer to Figure 26-3.) The least significant bits of the memory address are BBL15+ and BBL16+. Thus, BBL15+ chooses which 32 bits of the 64-bit word are desired, while BBL16+ further specifies which 16 bits are desired. The first two DWB locations make up a single 64-bit word, with BBL15+ serving the same purpose to the DWB as it does to main memory. This view of the DWB organization is the one to bear in mind throughout the discussion of the WB.

FIG. 26-3. Data Write Buffer Organization

```
DWB
addresses
           ________________
   0      |    32 bits     | \
           ----------------   > DWB location 0
   1      |    32 bits     | /
           ________________
   2      |    32 bits     | \
           ----------------   > DWB location 1
   3      |    32 bits     | /
           ________________
   4      |    32 bits     | \
           ----------------   > DWB location 2
   5      |    32 bits     | /
           ________________
   6      |    32 bits     | \
           ----------------   > DWB location 3
   7      |    32 bits     | /
           ________________
```

The DWB is addressed by 3 bits, referred to as the WB pointer. WBPTR{1:2}+ are produced by the VWB, and specify which of the 4 DWB locations is to be used during the next operation. The third address bit is based on BBL15+ as described above during CPU write operations, and distinguishes which half of the DWB location is to be accessed. Each 16-bit half of the resulting 32-bit location has a separate write pulse, TWBDH- and TWBDL-.

During a Write ACKnowledge (WACK) chain, the VWB supplies the WB pointer, and data is written into the DWB under the control of the write pulses.

During a write to memory, data is read out of the DWB under the control of the MT.

During a CPU read operation data may be read from the DWB instead of from memory. This occurs when the data in the WB hasn't been written out to memory yet. The VWB is responsible for recognizing this condition and replacing the stale data coming back from memory with the valid data from the DWB.

Data may also be written into the DWB during a CPU read operation. This will occur if the ECCC logic discovers there was a correctable error in the data that came back from main memory. In this case the corrected data is sent back to the CPU and also written into the DWB. Eventually this data will be written back to main memory to correct the error in memory.

The DWB is implemented with 74ALS870 static RAMs.

26.2.2 Address Write Buffer

The AWB is a 4 x 26-bit content addressable memory. When a new memory address is latched (in latch LAIN) a comparison is done between LAIN and all locations of the AWB. The result of this comparison is reported on HIT+. The location which found a hit sends its address to the VWB for possible use in WB pointer generation. This value is referred to as the hit pointer.

The contents of the 4 AWB locations are forced to be unique during system initialization, and should stay that way forever.

The 26 bits stored in an AWB location are the valid physical address bits which could be used to address main memory. BBL15+ and BBL16+ are not included, as they only specify which 16-bit section of a 64-bit word is desired. Thus, AWB addresses are meant to represent MOD 4 memory boundaries. The AWB is addressed by WBPTR{1:2}+.

Addresses are written into the AWB during either WACK or STart READ (STREAD) operations. If a hit wasn't found, the VWB supplies a new WB pointer, and the new address in written into the AWB when ADDMEMG- is active.

The AWB supplies the memory address for all main memory accesses except refresh. The VWB generates the WB pointer and the MT causes the output of the AWB to be driven onto bus MA. This address can be latched (in latch LAOUT) before it is driven onto MA.

The AWB has an address shifter on its output. This logic rearranges the address bits based on a 2-bit code which represents the memory configuration. This code is read at system initialization time, set in the CMI diagnostic register, and forgotten about. The code enables the shifter to swap address bits as necessary to prevent shadow memory or work around memory discontinuities.

The AWB is implemented in the ADdress BUFfer (PADBUF) VLSI chip.

26.2.3 Valid Write Buffer

The VWB consists of a 4 x 6-bit memory and its associated circuitry. Its goal in life is to keep track of data flow in and out of the WB.

Each VWB location is broken up into four valid bits and two write pending bits, as shown in Figure 26-4. The VWB is addressed by WBPTR{1:2}. Each valid bit represents 16 bits of valid data in the corresponding DWB location. A write pending bit set means that some valid data referenced by the valid bits has not been written out to memory yet, but needs to be.

FIG. 26-4. VWB Location Format

V0	V1	V2	V3	WP0	WP1

V = Valid bit
WP = Write Pending bit

A WB location is said to be "free" if neither write pending bit is set in that VWB location. The VWB generates a "free address" to be used in the possible generation of WBPTR{1:2} from this information.

As mentioned earlier, the VWB is responsible for generating WBPTR{1:2}. The WB pointer may be the "free pointer" generated by the VWB, the "hit pointer" generated by the AWB, or the "request pointer" generated by the MTOPS logic. The choice is controlled by the signals WBADMUX{2:3}+ as shown in Table 26-1. The selected pointer is latched (in latch LWBA) under the control of signal LWBAG-.

TABLE 26-1. WB Pointer Selection Chart

Desired Pointer	WBADMUX{2:3}+
Free pointer	00
Hit pointer	01
Request pointer	10

The request pointer is selected whenever a main memory access is to be done. The MT takes care of this operation. Otherwise, during WACK and STREAD operations, the choice of the WB pointer is limited between either "free" or "hit". The free pointer must be used whenever the memory address presented on BB does not cause the AWB to assert the HIT+ signal.

As we have seen, BBL15+ and BBL16+ determine which 16 bits of a 64-bit word are of interest. The CPU may also wish to write 32 bits at a time. This option is specified by the signal WRT32+. The setting and resetting of bits in a VWB location are controlled by these three signals, along with a series of command lines. These command signals may be broken down into those that control valid bits and those that control write pending bits. The valid bit control lines are:

- SETVAL+, which causes the current pattern of valid bits to be replaced by the pattern dictated by the three CPU signals,
- MERGEVAL+, which sets any bits indicated by the three CPU signals while leaving the rest of the pattern unchanged,
- SALLVAL+, which causes all the valid bits to be set, and
- CLRVAL+, which causes all the valid bits to be cleared.

The write pending bit control lines are:

- SETWP+, which causes the write pending bit dictated by BBL15+ to be set,
- CLRWP+, which causes both write pending bits to be cleared, and
- SETBTHWP+, which causes both write pending bits to be set.

Only one control signal in each group may be active at any time. Assertion of more than one command in each group leads to unpredictable results. The commands are actually executed when the clock signal VALMEMG- is asserted.

If all four VWB locations have a write pending bit set the WB is said to be full. This state is reported over the signal WBFULL+. No further CPU requests can be handled by the WB when this condition occurs until a WB location is emptied by writing its contents to memory. The reason is readily apparent when you consider the CPU trying to write into the full WB. If the AWB doesn't detect a HIT+, the external logic will ask the VWB for the free pointer. The free pointer can't have an indeterminate value, so the WACK chain will write the new data into the WB, overwriting data which hasn't yet gone out to memory.

The VWB is implemented in the PADBUF VLSI chip.

26.2.4 The WB as a Whole

Stepping back, we can look at the big picture to see how the WB pieces just described play together. Suppose we have just initialized the AWB to have four unique addresses, and the VWB to have no valid or write pending bits set. (The system microcode is required to do this at system initialization time.) The CPU does a write request. The AWB does not detect a hit, so the external logic requests the free pointer from the VWB, which is latched in the LWBA latch. SETVAL+ and SETWP+ are asserted, and the WACK chain writes the new address into AWB (using ADDMEMG-) , sets the appropriate valid bit(s) and the correct write pending bit (using VALMEMG-), and writes the data into the DWB (using TWBDH- and/or TWBDL-). A CPU write request during which the AWB detects a hit works similarly. The important differences are that the external logic would have to ask for the hit pointer instead of the free pointer, and that MERGEVAL+ should be asserted in place of SETVAL+.

CPU read requests are handled in a similar fashion. If the AWB does not detect a hit the external logic must ask the VWB for the free pointer. CLRVAL+ is asserted, and the STREAD chain writes the new address into the AWB and clears all the valid bits. If the AWB does detect a hit, the STREAD chain must not clear the valid bits. In either case, when the data is returned from memory, the VWB detects if the DWB contains more up-to-date data than the memory by using the valid bits, and replaces memory data with data from the DWB if necessary. This data is sent to the CPU and written into the DWB. SALLVAL+ is asserted, and all the valid bits are set. If there was a correctable error, SETBTHWP+ is also asserted, so both write pending bits will be set. (A design limitation makes it impossible to tell the VWB which 32-bit word had the correctable error in it, so both write pending bits must be set to insure the error is corrected in main memory.)

CLRWP+ is asserted and both write pending bits are cleared after data is removed from the WB and written to main memory.

26.2.5 Memory Timer OPeration Scheduler (MTOPS)

The WB may contain data destined for as many as four different main memory locations at any given time. Choosing which main memory operation to do next, or even to wait a while before doing the next memory operation, is the job of the MTOPS. The original goal of smoothing out traffic on the main memory bus is achieved by the algorithms implemented in the MTOPS.

Two conflicting forces are at work on the WB all the time. First, we want to save data in the WB as long as possible. The longer we save it, the greater the chance that the CPU will write more data destined for the same 64-bit location. (Imagine a program which is updating all the elements of a contiguous array.) If we save the data from the first CPU write until after the CPU does a second write to the same location we have saved ourselves one memory access. Opposing this desire is the WB full problem discussed earlier. If we save all the data we can, the WB will fill up very quickly, and we may have to stall the CPU. This defeats our purpose in life and makes us very depressed.

A modified Least Recently Used (LRU) algorithm is used by the MTOPS to decide which memory operation to do next. The classic LRU algorithm, briefly stated, says that when trying to decide which data to move from an easily accessible place to a less accessible place, choose that data which was least recently used. The chances are very small that you will need that data again soon if you haven't needed it lately. The LRU algorithm has been shown to be very efficient over time on a large number of computer systems in which it has been implemented. A system to keep track of just how recently a particular piece of data has been used is inherent in the algorithm, adding overhead for each piece of data you keep track of.

Before the MTOPS can apply the algorithm, however, it must sort the pending operations. A sorting operation can be done by examining the pattern of the valid and write pending bits in each VWB location. Figure 26-5 shows the 5 "aligned write" patterns. When any of these patterns occurs in a VWB location it means the data could be written to main memory directly in one access. The patterns shown in Figure 26-6 are examples of the multitudes of "unaligned write" patterns that may occur. These VWB locations cannot be written directly to main memory, since main memory can only write even multiples of 32 bits at a time. A read of main memory would have to be done to fill in the gaps in the valid bit pattern. Recall that any read of main memory causes the VWB to replace stale data being returned from memory with fresh data from the DWB, and that SALLVAL+ is asserted to set all the valid bits in the VWB location. Applying this to the examples shown in Figure 26-6, we see that all of these examples turn into one of the patterns shown in Figure 26-5. Thus, "unaligned writes" require two memory accesses to be written into memory.

The WB uses a modified LRU algorithm referred to as Last Written (LW), in which only the last piece of data used is tracked. The MTOPS includes a LW latch which is controlled by signal LWG-. Whenever the latch is opened the current value of the WB pointer (in latch LWBA) is copied into the LW latch. In practice, the LW latch is only updated during WACK operations. The WB pointer value of the last piece of data that entered the WB is therefore always known.

The MTOPS must decide which memory access should be done next using the sorted VWB patterns and the value in the LW latch. The decision is easiest when no write pending bit is set. In this case, the MTOPS doesn't assert any of its request lines, and everyone is happy. The decisions get harder as the number of locations with write pending bits set increases. The WB location whose pointer value is currently in the LW latch is disregarded. The data referred to by the value in the LW latch is the MOST recently used, and therefore is most likely to be used again soon. Data in the WB location pointed to by the LW latch is therefore NEVER a candidate to be written out to memory.

The following paragraphs describe the decision process for the four combinations of write pending bits.

- If only one location has a write pending bit set it is the LW location. No request line is asserted.

FIG. 26-5. Aligned Write VWB Patterns

1	1	1	1	1	1

64-bit aligned write

1	1	1	1	1	0

1	1	0	0	1	0

32-bit even aligned writes

1	1	1	1	0	1

0	0	1	1	0	1

32-bit odd aligned writes

- If two locations have a write pending bit set, the one whose value matches the LW latch's value is disregarded. If the other one is an aligned write the MTOPS asserts its write request signal. If it is an unaligned write no request line is asserted.

- If three locations have a write pending bit set, the one whose value matches the LW latch's value is disregarded. If one or both of the remaining locations is a 64-bit aligned write, the MTOPS asserts its write request signal. If no 64-bit aligned write is present but one or both of the remaining locations is an aligned write, the write request signal will also be asserted. If both remaining locations are unaligned writes, the read request signal is asserted. In the cases where both locations have the same type of operations to do, the location with the higher WB pointer value is chosen by default.

- If all four locations have a write pending bit set, the one whose value matches the LW latch's value is disregarded. Of the remaining three locations, if any of them is an aligned write the MTOPS asserts its write request signal. If all three are unaligned writes, the read request signal is asserted. Ties among multiple locations with the same type of operation pending are again broken by WB pointer value. Aligned 64-bit writes take precedence over aligned 32-bit writes.

FIG. 26-6. Unaligned Write VWB Patterns (Examples)

```
 ______________________________________
|      |     |     |     ||      |     |
|  1   |  0  |  0  |  0  ||  1   |  0  |
|      |     |     |     ||      |     |
 ______________________________________
 ______________________________________
|      |     |     |     ||      |     |
|  1   |  1  |  1  |  0  ||  1   |  1  |
|      |     |     |     ||      |     |
 ______________________________________
 ______________________________________
|      |     |     |     ||      |     |
|  1   |  1  |  1  |  0  ||  0   |  1  |
|      |     |     |     ||      |     |
 ______________________________________
 ______________________________________
|      |     |     |     ||      |     |
|  0   |  1  |  1  |  0  ||  1   |  1  |
|      |     |     |     ||      |     |
 ______________________________________
 ______________________________________
|      |     |     |     ||      |     |
|  0   |  0  |  0  |  1  ||  0   |  1  |
|      |     |     |     ||      |     |
 ______________________________________
```

Table 26-2 summarizes this information.

TABLE 26-2. MTOPS Decision Chart

Number of Locations w/ Write Pendings set	Request ?	1st choice	2nd choice	3rd choice
0	No			
1	No			
2	Only if aligned	64 bit	32 bit	
3	Yes	64 bit	32 bit	unaligned
4	Yes	64 bit	32 bit	unaligned

When a decision has been reached the MTOPS sends the request pointer to the VWB for possible use in WB pointer generation.

The request signals referred to in the discussion above are sent to the Memory Timer (MT) which evaluates them when it decides which MT routine to execute next. In addition to the read and write request signals there are two other signals which help the MT vector to the correct routine. The first indicates a 64-bit write is present, while the second indicates whether a 32-bit write is even or odd. These signals are only meaningful when the write request signal is asserted.

One special case is not shown in Table 26-2. (Yes, there ARE exceptions to every rule.) The decision rules just described make it possible for data to go into the WB and never come out. Consider the following sequence of CPU writes:

1. The CPU writes 16 bits to memory which are placed in WB location 0. (This can only occur if locations 3, 2, and 1 are currently in use, and none of them detects a hit.)

2. The CPU writes data to memory which is placed in some WB location other than location 0.

Location 0 is not the last written location. The MTOPS can consider it as a candidate to be the next memory operation it requests. Location 0 contains an unaligned write. Unaligned writes aren't requested until there are three write pending bits set in the VWB. One of those locations with its write pending bit set is the last written location, leaving one other location besides location 0 under consideration. If that location contains an aligned write it gets requested before the unaligned write in location 0. If that location contains an unaligned write it gets requested before the unaligned write in location 0 because its WB pointer value is higher. Either way, when the access for the other location is finished its write pending bit is cleared, leaving two write pending bits set. The unaligned write in location 0 is no longer considered.

This scenario doesn't happen frequently, but it can happen. When it does occur, it effectively makes the WB only three locations deep, impacting system performance. To get around this, the MTOPS detects when there is an unaligned write in location 0 and that location 0 is no longer the last written location, and empties the WB by requesting all pending accesses in turn. This is an automatic function similar to DENALLOPS+ described in the next section.

The MTOPS is implemented in the PADBUF VLSI chip.

26.2.6 WB Diagnostics

Three diagnostic signals are available in the MC diagnostic register to control MTOPS functions. They are:

DSINGOPS+	Forces the WB to appear full whenever one write pending bit is set.
DENALLOPS+	Instructs the MTOPS to request any pending memory operation immediately, regardless of the value in the LW latch or the number of locations with

write pending bits set. Memory operations are requested in the order shown in Table 26-2 until the WB is empty.

DISLW+	Instructs the MTOPS to include all locations in the decision process, even the location referenced by the LW latch. This has the effect of altering the "1 write pending" line in Table 26-2 to look identical to the "2 write pendings" line.

Another diagnostic function in the MC diagnostic register is the FBADP+ bit. This bit instructs the AWB to force the MA parity bits to be a "1" during all memory accesses. This is a useful way to force bad parity onto MA to test the memory arrays.

These four bits are all set inactive by microcode at system initialization time. They can be accessed by the PMA instruction MDWC.

There are three other diagnostic signals in the MC diagnostic register which are used in initializing the WB. These bits force the WB pointer to some known value. These bits are accessible only by microcode.

26.3 CPU Request & Acknowledge

A memory write is initiated when the microcode DEST field specifies MEMWRITE or MEM32. The MC decodes the fact that a write request is pending, and at CS8+ of the microcode step it:

1. Latches the address sent across BB from the S unit with signal BBL+
2. Clocks the data sent across BD from the E unit with signal TDATIN+
3. Sets a busy signal (MBSY-) so no other CPU request can occur until the address and data just received have been processed
4. Sets an internal write request (CPWRTREQ-) to notify the write buffer logic that information is available to be processed if the write buffer is available

A memory read operation is initiated by 1 of 2 methods:

- a cache miss, or
- when the microcode IAC field specifies READHSM64 (Read High Speed Memory 64 bits)

In both cases the memory address is latched as in step 1 above, and an internal read request (CPREADREQ-) is activated to tell the WB that a read has been requested. MBSY- is also asserted here as in step 3 above.

Step 4 above refers to WB availability. The WB is available for STREAD and WACK operations if both of the following are true:

- The previous WACK operation is finished.
- The MT is inactive, or, if it is active, it is finished using the WB for the memory operation it is controlling.

26.3.1 CPU Write

When a write request is posted, the MC logic starts the Write ACKnowledge (WACK) chain as soon as the WB is available. The signal PWACK- is asserted and clocked at End of Beat (EOB) to become WACK+/-. This starts off the WACK chain of events.

WACK+ goes through a series of registers which produce delayed versions of WACK+ every beat. These signals are used to differentiate among the different beats of the WACK chain, and assist in the proper sequencing of operations. There are three delayed versions of WACK+ in the WACK chain, D1WACK+, D2WACK+, and D3WACK+ respectively. Each one beat long. The operations carried out during a WACK chain are discussed below on beat by beat basis.

- Prior to the WACK chain starting, the latched main memory address is sent to the WB which compares it against the four AWB locations. If the address matches one of the locations, the WB asserts the HIT+ signal.
- WACK - The WACK chain asks for the WB pointer as shown in Table 26-1. If a hit is detected, the hit pointer is used. If a hit is not detected, the free pointer is used instead. The signals WBADMUX{2:3}+ tell the WB whether to use the hit or free pointer. The WB pointer is sent to the DWB RAMs to select a location into which to store the data.

  Signal WACK+ is sent to the WB on input pin LAING- which latches the address at the input to the WB for the duration of the write cycle.

  It is also gated to become signal LMTOPSG- which closes the WB latch that contains the MT operations to be requested. This prevents the upcoming update of the VWB from propagating spurious requests through the MTOPS logic to the MT.
- D1WACK - Once the WACK chain has started, the 74AS646 transceivers which hold the data that was clocked from the E unit are turned on to drive data in the direction of the WB RAMs. Either 16 or 32 data bits will be written into the RAMs based on the microcode destination field. The write pulses to the data RAMs, TWBDH- and TWBDL-, are half beat signals and are active during the second half of D1WACK.

  The appropriate valid and write pending control signals are also clocked into the WB during the second half of D1WACK with signal VALMEMG-. In addition, the address is clocked during the last half of this beat with signal ADDMEMG-. The latched main memory address is written into the AWB, while the VWB is clocked with the SETWP+ command and either the MERGEVAL+ command, if there was a hit, or SETVAL+ if there wasn't.

- D2WACK - Signal MBSY- was asserted at CS8+ of the microcode step that issued the memory write. This is sent to the PCU. Its purpose is to hold up the pipeline should the microcode issue any other memory reads or writes before the MC logic has had a chance to process the existing write. Once the address, data, and appropriate status information have been written into the WB, the busy signal (MBSY-) goes inactive and the MC is then ready to accept more reads or writes from the CPU. This happens at the beginning of D2WACK. The system's processing of instructions will not be affected in any way by this signal except, if another memory operation is requested before the present one has been processed, the pipeline will be held up.

  D2WACK- also goes into the WB on input LSTWRTG- to tell the write buffer to update the value in the LW latch.

- D3WACK - This is used to end the ABORT sequence during a write. ABORTs will be discussed in detail in section 26.4.2.

26.3.2 CPU Read

The STREAD chain is started in a fashion similar to that described for WACK. The STREAD chain generates and uses the delayed STREAD signals D1STREAD+, D2STREAD+, and D3STREAD+. Each is one beat long. The operations performed by the STREAD chain are discussed below on a beat by beat basis.

- STREAD - The latched memory address is compared as it was in the WACK chain, and the same hit decision is made. STREAD+ is also delayed by one TMCLK to become DSTREAD+, which is gated further to become MUXLAOUT+. This signal controls a mux in the WB that sends the proper address to the output latch which in turn will be sent to the memory arrays.

- D1STREAD, D2STREAD - Delay beats to allow time for subsequent operations.

- D3STREAD - During the last half of this beat, the address is clocked into the WB with signal ADDMEMG-. The valid bits are cleared with signal VALMEMG- if there was a miss, else they remain set.

  ABORT, which was mentioned in the WACK chain, goes active with STREAD and remains active until D3STREAD goes away.

- The MT operation starts up in parallel with the STREAD chain of events and continues on after the STREAD chain is finished until the read operation has completed.

  The Memory Timer (MT) logic is activated with a request to the priority encoding logic by signal STREAD+. The MT addressing logic vectors to the MT routine which controls the reading of main memory. The memory address is sent out through the AWB to all memory array boards over memory address bus MA. Each memory array board examines the received address to see if that address is within its address range. The board that detects an address range match starts up its internal read logic.

When the data is ready, the MT logic clocks the first 32 data bits from the memory arrays into the memory data bus (MD) data transceivers with signal MTDIN+. From there the data bits are sent to the ECC circuitry to check for any ECC Correctable (ECCC) or ECC Uncorrectable (ECCU) problems. Parity is generated on the returning data at the same time.

All the data and parity bits are latched in a latch which connects MD with the BD transceivers. Stale data returning from memory is replaced with valid data from the DWB if necessary. This is accomplished by logic which examines the valid bit pattern for the current WB location in use. This guaranteed fresh, corrected data is written into both the BD transceivers and the DWB RAMs. The data is automatically written into the WB just in case there was an ECCC detected so we may then write the corrected data back out to memory. The pending bits are set only in the case of an ECCC.

Assuming no ECC problems, the MT logic brings the second 32 bits into th receivers and processes them as it did the first 32. After the first 32 bits have been clocked into the BD transceivers, the MC notifies the CPU that it has data ready for it by asserting the signal MDATAV+. The CPU issues IAC MRDY when it wants the first 32 data bits. This IAC will hold up the pipeline until it sees signal MDATAV+ sent from the MC unit. When MDATAV+ is asserted, the CPU takes the first 32 data bits from the transceivers and puts them either in cache or where the microcode specifies, or both. The MC logic knows that the first 32 bits have been taken because it sees signal MCTOBD- sent from the BD arbitration logic, which says the data can be put onto BD. The MC logic will keep MDATAV+ asserted until it sees MCTOBD-.

After the first 32 bits have been taken, the second 32 data bits are then placed in the DWB RAMs and the BD transceivers. The second 32 bits will be taken by the CPU for a cache miss, and may or may not be taken otherwise depending on what is specified by the microcode. If the microcode wants the second 32 bits, it issues IAC MCONBD which places the data from the BD transceivers onto BD. If it does not want the data, the data will remain in the transceiver until overwritten by data from a subsequent read of main memory.

All reads from memory are 64 bit reads, whether or not all 64 bits are required. When the memory cycle is completed, the MT releases MBSY- so the CPU will be able to process the next read or write when required.

26.4 Memory Timer

The MT logic is used whenever it is necessary to read, write, or refresh the main memory arrays. This logic consists of:

- Prioritization logic which resolves conflicts in case of simultaneous requests

- Addressing logic which sequences through the routines
- PROMs whose data is the control for the memory and other MC logic
- Registers to stabilize the PROMs' outputs
- Various control circuitry

The priority encoding logic takes request inputs from various subsections of the MC logic and encodes the outputs in the following priority (highest to lowest):

1. REFRESH - requested by refresh logic
2. NOP - requested by a flip-flop set by SYSCLR-
3. CPU STATUS READ - requested by microcode
4. CPU READ - requested by S unit (cache miss) or microcode (IAC READHSM64)
5. 64-bit MT WRITE - requested by WB MTOPS logic
6. 32-bit MT WRITE (odd) - requested by WB MTOPS logic
7. 32-bit MT WRITE (even) - requested by WB MTOPS logic
8. MT READ - requested by WB MTOPS logic

CPU requests always have priority over WB requests, and refresh always has priority over everything else.

The NOP routine, as the name suggests, does nothing. Its only purpose in life is to keep WB requests from occurring when returning power to the CPU after having been in battery backup mode. When returning from battery backup mode, there is no guarantee what kind of requests may be coming from the WB until it has been initialized. Therefore it may start requesting memory accesses which could destroy the data in memory which the MC labored so hard to keep correct during battery backup mode. When power is restored, a flip-flop is asynchronously set by SYSCLR-. It asserts a MT NOP request constantly. Whenever a refresh request isn't pending (most of the time) the NOP routine will be executed, keeping the WB from getting a request in. The flip-flop is reset by microcode after it has initialized the WB.

When one or more request(s) are pending, the prioritization logic outputs the highest priority request as an encoded value. These signals, MTREQ-, MTOP2-, MTOP1-, and MTOP0- flow through an open latch to go to control circuitry and to some of the MT address register inputs. After the latch closes with signal MTACTIVE-, the three signals MTOP{2:0}+ are clocked to become three of the four most significant address bits to the MT PROMs. These three bits break the MT PROMs into eight different routines. Each of these routines may be up to 32 beats (PROM locations) long.

Once the PROMs are addressed to start off a routine, the subsequent addresses are controlled by the clocked outputs of one of the PROMs. We thus may go anywhere within a 32 location routine by changing the five least significant address bits. This is analogous to a microcode next address field. Indeed, there are even MT conditional branches.

Once an MT routine has been started, the signal TMT+ is activated. TMT+ clocks the outputs of the PROMs and the next PROM address. This signal cycles until the MT routine is completed.

The following example of a REFRESH request will show the manner in which all MT operations cycle.

1. The refresh circuitry posts a refresh request which is fed into the prioritization logic as signal GMTRFREQ-.

2. The prioritization logic output MTREQ- goes low signifying that some request is pending. The encoded signals MTOP{2:0}- all go low signifying that a REFRESH request is the highest priority request pending.

3. These signals flow through an open latch to become signals LMTREQ- and LMTOP{2:0}-. From there they go to the inputs of the MT address register, and to circuitry which sets signals MTACTIVE+/- and, one beat later, P1MTACTIVE+/-.

4. The combination of MTACTIVE+ active and P1MTACTIVE+ inactive causes the LMTOP{2:0}- signals to be clocked as MTAD{7:5}+ and clears the five least significant MT address bits (MTAD{4:0}+). The eight address bits now address location 0 of the MT PROMs, which is the first location of the REFRESH routine. (The three signals LMTOP{2:0}- will remain the same while the other five bits will change to point to subsequent locations within the routine as we execute the MT operation.)

5. Signal P1MTACTIVE- allows signal TMT+ to occur, which clocks the PROM outputs to sequence through the operation. During the routine, appropriate MT signals flow from the MT to other parts of the MC and to the memory arrays to control the various operations.

6. When the MT operation is complete, the signal MTDONE- shuts off MTACTIVE+ and, one beat later, P1MTACTIVE+ signals. This in turn stops TMT+ from cycling and opens the LMTOP{2:0}- latch to allow any new pending requests to be honored.

All MT operations occur in the same manner. The main differences between the different operations being:

- The prioritization logic will point the upper three address bits to a different 32 location routine based on the highest priority input request.

- Different MT PROM output control signals will become active based on whichever routine is executing.

All MT operations sequence in basically the same manner. In order to access the memory arrays, whether refreshing, writing, or reading we need the following items:

- Memory address provided either from the WB or the refresh circuitry
- MREQ- which is the memory request signal

The following paragraphs highlight some of the MT signals, and show when and how they are used.

When writing to memory, we will do either a 64-bit, 32-bit odd, or a 32-bit even write, based on the request from the WB. The sequence through each of the three routines is approximately the same with the exception of which write control signals are sent to the memory arrays.

- On a 64-bit write, the even word is sent to the memory array first. MT signal MLDIEV+ latches these bits onto the arrays. The odd word is sent to the array next. These bits are latched with signal MTLDIOD+. Signals MWEEV- and MWEOD- tell the memory array to write the even and odd 32 bits respectively into memory.

- On a 32-bit write, only the appropriate 32-bits would be sent along with either MLDIEV+ and MWEEV-, or MTLDIOD+ and MWEOD-.

As the data destined for memory is latched at the output of the DWB RAMs, it is also sent to the ECC logic to generate the 7 check bits which are written with each 32 data bits. When we say we write 32 data bits to memory we always write the check bits with them, so each write is actually 39 or 78 bits.

During a cache miss or a Read High Speed Memory operation, the memory array is started up by signal MREQ- as usual. When the data is ready to be returned, the signal MDOSELEV- is sent active to the array if the even word was requested first, or inactive if the odd word was requested first.

When data is brought back from the memory arrays, it is clocked into a set of transceivers and driven onto bus MDATxx (an MC internal version of MD) while parity is being generated on the returning data. At the same time, the data is sent into the 74ALS632 ECC chip where it checks for ECCCs and ECCUs. MT signal MTCORRERR- becomes active for one beat to tell the MT next address logic to look at signal ERR- which comes from the ECC chip. If ERR- is active, there has been an error detected and the next MT step will be in the error correction portion of the MT routine. ERR- is the jump condition for the MT's two-way branch to different sections of the read routine. While in the error correcting routine, the MT signal MTEDACS0+ is asserted to latch the data into the ECC chip, and MTMDEN- is removed to stop the transceivers from driving the returned data. MTOEBX- and MTOECBX- are asserted to drive the corrected data from the ECC chip to bus MDATxx where it takes the normal path of data being returned from memory. The data is written into the WB (as is all data returned from memory), and the write pending bit is set if there was an ECCC. It is not set on an ECCU. If it were, the ECC chip would generate proper check bits on the bad data on the subsequent write to memory. When this data was next read from memory, there would be no error detected and we would think we are operating on good data. When an error is detected, the MT ECCC/ECCU routine is executed, which adds 3 beats to the memory read routine.

The code used to generate ECC check bits is shown in Table 26-3. Table 26-4 shows the interpretation of the syndrome field. This field is returned during a read status MT routine.

TABLE 26-3. ECC Check Bit Generation

	Data Bit																															
Check Bit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
0	x	x	x	x	x	x	x	x																	x	x	x	x	x	x	x	x
1	x	x	x	x	x	x	x	x									x	x	x	x	x	x	x	x								
2	x	x							x	x	x	x	x	x			x	x							x	x	x	x	x	x		
3			x	x	x				x	x	x				x	x			x	x	x				x	x	x				x	x
4	x		x			x	x		x			x	x			x	x		x			x	x		x			x	x			x
5				x		x		x		x		x		x	x	x				x		x		x		x		x		x	x	x
6	x		x	x		x					x		x	x	x			x			x		x	x	x	x		x				x

x bits on each row are XOR'ed together to form the indicated check bit.

TABLE 26-4. Syndrome Bits

Bits 1234567	Meaning
0001010	correctable error at data bit 1
0001111	correctable error at data bit 2
0010010	correctable error at data bit 3
0010100	correctable error at data bit 4
0010111	correctable error at data bit 5
0011000	correctable error at data bit 6
0011011	correctable error at data bit 7
0011101	correctable error at data bit 8
0100010	correctable error at data bit 25
0100100	correctable error at data bit 26
0100111	correctable error at data bit 27
0101000	correctable error at data bit 28
0101011	correctable error at data bit 29
0101101	correctable error at data bit 30
0110000	correctable error at data bit 32
0110101	correctable error at data bit 31
0111111	correctable error at check bit 0
1001011	correctable error at data bit 17
1001110	correctable error at data bit 18
1010011	correctable error at data bit 19
1010101	correctable error at data bit 20
1010110	correctable error at data bit 21
1011001	correctable error at data bit 22
1011010	correctable error at data bit 23
1011100	correctable error at data bit 24
1011111	correctable error at check bit 1
1100011	correctable error at data bit 9
1100101	correctable error at data bit 10
1100110	correctable error at data bit 11
1101001	correctable error at data bit 12
1101010	correctable error at data bit 13
1101100	correctable error at data bit 14
1101111	correctable error at check bit 2
1110001	correctable error at data bit 16
1110100	correctable error at data bit 15
1110111	correctable error at check bit 3
1111011	correctable error at check bit 4
1111101	correctable error at check bit 5
1111110	correctable error at check bit 6
1111111	no error

All other syndromes indicate an uncorrectable error.

26.4.1 Read Status

During certain times it may be necessary to read the status of the MC. This is accomplished by doing a modified Read High Speed Memory operation. The microcode first issues IAC RDSTAT, which sets the addressable latch RDSTAT+. When this is set, all READHSM64 operations are detected as status reads, which have higher priority in the MT priority logic than data reads. The MT routine vectors to the read status routine which, instead of accessing main memory, gets status information from the logic and places it on BD as though it were data from memory. The first 32 bits brought back consist of:

- 4 parity bits from the AWB
- 2 unused bits
- 1 of 5 encoded DNET and CS parity bits (other 4 in 2nd word)
- 25 address bits from the AWB

The second 32 bits brought back consist of:

- Two memory size bits
- Control Store parity error
- Error bit 15 for ECCCs and ECCUs
- Missing Memory Module error
- ECCU error
- Data or Memory address parity error
- I/O parity error
- 4 bus MCBD parity error bits
- 4 bus MDAT parity error bits
- 4 of 5 encoded DNET and CS parity bits (other 1 in 1st word)
- 4 Memory address parity bits
- ECCC error
- 7 syndrome bits indicating which bit was corrected on an ECCC

The data is brought back in the following manner: Signal MTRDSTAT- is gated and enables MA to be driven onto MD. MD is then enabled onto MDATxx just as if it were data returning from memory. The ECC checking circuitry is not enabled, parity is generated on the bits, and they are then sent via the BD transceivers to the E unit. The second 32 bits are likewise placed onto MDATxx when MT signal MTSTAT- becomes active. Parity is generated and the data sent to the E unit as before.

26.4.2 MT Abort

Refresh, CPU reads, cache misses, and CPU writes should and do take priority over WB requested memory operations. One method of doing this is utilizing the priority encoding scheme with the WB operations having lower priorities. ABORT logic allows a WB operation to be aborted even after it has been started if a higher priority request becomes active. WB operations may only be aborted if they have not yet issued MREQ- to the memory arrays. If they were to be aborted after this time the memory array contents could become unpredictable.

Once the MT operation has started and before it makes signals GOKWRT+ and GOKRD+ inactive, (OK to write and OK to read, respectively) they may be aborted. When a read is requested and the signal GOKRD+ is still active the STREAD chain starts. The signal ABORT+ is made active with STREAD+ and stays active through D3STREAD+. If a write is requested and GOKWRT+ is still active, the WACK chain starts. ABORT+ is made active with WACK+ and stays active through D3WACK+.

ABORT+ prevents the lower priority WB requests from happening in the MT prioritization logic. If one is already happening, it shuts the MT off and allows the higher priority request to be honored.

26.4.3 MT Diagnostic Features

The following MT diagnostic registers are accessible only by microcode:

- DGSELBFAD+ forces the WB address mux to point to the WB location specified by signals DGBFAD{2:3}+ instead of the normal hit mechanism.
- DISERCOR+ inhibits the MT addressing logic from looking at ECCCs and ECCUs.
- DSMPLEM{1:4}- tell the logic to ignore any parity error received from memory array boards 1 thru 4 respectively, one bit per board.
- PDMTMODE+ is clocked and becomes the high order address bit of the MT PROMs, which allows diagnostic testing of the MT logic with routines in the private half of the PROMs.

26.5 Memory Configurations

When the memory array boards are inserted into the backplane they are each connected to common signals MEMSIZEA- and MEMSIZEB-. These signals are pulled up by two resistors on the MC. The purpose of these signals is to show the largest size memory array board installed in the backplane. This is translates directly into MB/slot. Table 26-5 shows how these signals are interpreted.

TABLE 26-5. Memory Configuration Slot Sizes

AB	Slot size	SHFTCTR{1:2}+
11	4 MB	01
10	8 MB	00
0x	16 MB	10

The initialization routine reads these two bits, the two highest order bits of the status word. It then sets two bits in the MC diagnostic register, SHFTCTR{1:2}+. These bits tell the AWB shift logic how to shift the upper memory address bits based on the memory available. Table 26-5 shows the required settings for these bits.

The shift logic necessary to do slot selection was designed into the PADBUF VLSI chip very early in the project. While it was anticipated that 32 MB memory array boards would be available during the 4150's lifetime, 4 MB was thought to be required, and 64 MB was considered to be an acceptable upper limit on memory size.

The memory configuration rules are as follows:

1. Only 8 MB Abel, 16 MB Cain, and 32 MB Cain boards may be used.

2. Always begin in slot #1 and do not skip slots unless the 32 MB Cain boards are used.

3. Always skip the next slot when using a 32 MB Cain board. 32 MB Cain boards may be placed only in slots 1 and 3. If there is one 32 MB Cain board, slot 2 may not be used. If there are two 32 MB Cain boards, no other boards may be used.

4. Always put the highest density remaining board in the next legal slot. Thus, exhaust the supply of 32 MB boards before inserting 16 MB boards and exhaust the supply of 16 MB boards before inserting 8 MB boards.

Table 26-6 shows the legal memory configurations.

TABLE 26-6. Memory Configurations

64 MB Configurations

Slot	1	2	3	4
	32		32	
	32		16	16
	16	16	16	16

56 MB Configurations

Slot	1	2	3	4
	32		16	8
	16	16	16	8

48 MB Configurations

Slot	1	2	3	4	
	32		16		
	16	16	16		
	16	16	8	* 8	(See Note 1)

40 MB Configurations

Slot	1	2	3	4	
	32		8		
	16	16	8		
	16	8	* 8	* 8	(See Note 1)

32 MB Configurations

Slot	1	2	3	4	
	32				
	16	16			
	16	8	* 8		(See Note 1)
	8	8	8	8	

24 MB Configurations

Slot	1	2	3	4
	16	8		
	8	8	8	

16 MB Configurations

Slot	1	2	3	4
	16			
	8	8		

8 MB Configurations

Slot	1	2	3	4
	8			

NOTE 1: 8 MB holes are created in memory where indicated by the asterisks in the configuration tables. These produce some additional Primos memory management overhead and should be avoided when possible.

26.6 Refresh

The MT refresh operation has already been discussed in the MT section. What remains to be discussed is the generation of the refresh address and the request. The signal BPCFCLK+ is provided by the buffered output of a 5 MHz crystal. The refresh circuitry takes this same crystal output and uses it as a clock to a counter whose carry out pulses approximately every 16 microseconds. This carry out is also used as a clock to another counter whose outputs are the refresh addresses. We thus change to a new address with each carry out of the first counter, RFREQ-. This signal is clocked and recirculated until the MT priority logic has accepted it and started a MT REFRESH operation. It is then reset until the next counter overflow, which is in approximately 16 microseconds.

When in the MT Refresh routine, signal ENRFAD+ allows the twelve refresh address bits onto MA in place of the address lines from the AWB.

26.7 Parity Checking/Generating

Parity is checked in two places:

1. on bus MCBD as data comes into the WB RAMs on a write, and
2. on MDATxx data latched at the output of the WB RAMs before the data is written to memory.

During a CPU write the data is clocked into the BD transceivers. When the WB is available, the drivers are enabled onto MCBD (an MC internal version of BD). Parity checkers check the parity and the result is clocked at the end of D2WACK.

The parity circuitry on MDATxx has two functions:

1. Check parity on the data out of the WB RAMs destined for memory, and
2. Generate parity on data read from memory, or on corrected data from the ECC logic, which is to be sent to the rest of the CPU.

Nine-bit parity checker/generators are used in a dual purpose mode.

When reading data from memory, the eight data bits are placed on the inputs of the checker/generator while the ninth bit input is forced low. The output of the checker/generator becomes the proper parity for the eight data input bits. This signal is placed on the bus and is sent along with all the data bits to be latched and sent to the WB and CPU logic.

On a MT write to memory, the outputs from the RAMs are latched and sent to the same parity checker/generators. All nine bits are sent to the chip. The same output this time going high signifies a parity error, while no error is detected if it remains low. The error signals from all four checker/generators are combined and sent to the error reporting logic.

26.8 Error Reporting

There are four categories of errors reported in order of significance:

1. MISMOD - Missing Memory Module
2. ECCU - Uncorrectable memory error
3. DATADPE - data parity or memory address parity error

4. ECCC - correctable memory error

When a memory array read or write operation takes place, the selected memory array sends back a signal MSELVAL- meaning a valid memory array was accessed. The MT looks at this at the appropriate time and, if no valid signal is seen, causes FMISMOD+ to be set. This causes a Missing Memory Module Trap when FMISMOD+ reaches the PCU. If subsequent MISMODs occur before the trap condition is cleared out, the last trap address to occur will be reported, not the first.

On a read from the memory arrays, the ECC checking circuitry has already been discussed. The resulting error, ECCC or ECCU is sent to the error detecting circuitry, and if no higher priority error already exists, will be clocked and reported to the PCU to be handled appropriately. The address that contained the first ECCU to cause a trap will be reported even if more occur before the trap condition has been cleared out.

On a memory array operation, the array checks the address parity and sends back an error if one was detected. This memory address parity error is combined with the two previously discussed data parity errors. If no higher priority error exists, it is clocked as FDATADPE+ and sent to the PCU. The first occurrence of a parity error will be reported, not any subsequent parity errors until the condition has been cleared out. The same holds true for ECCCs.

Any error detected will remain active unless followed by a higher priority error. The error is be cleared by IAC ACKPE, acknowledge parity error.

26.9 Battery Back Up

Battery Back Up (BBU) is used to keep the main memory refreshed during a power failure. This occurs when the Diagnostic Processor (DP) senses a power failure and sends notification to the MC logic. The MC switches a mux from the normal refresh circuitry to a BBU mode upon receiving this notification. It also asserts ENALLOPS+ so that the MTOPS will empty the WB before the power goes off. (The DWB RAMs are not backed up, so any data remaining in them when the power goes off is lost.)

The signal MRFSH- is asserted constantly to tell the memory array to do a refresh cycle whenever it sees a request (MREQ-). MREQ- is fed from the refresh address counter in this mode of operation.

26.10 VLSI Usage

The AWB, VWB, and MTOPS sections of the WB are implemented in VLSI in a slice fashion. Two PADBUF VLSI chips are used for this purpose.

26.11 Critical Paths

The path from the time the memory address is latched into the MC until the WB pointer is latched during a CPU memory access must be less than 2.5 beats.

Setting MBSY- on the MC unit on a READ or WRITE and sending it to the S unit is a one beat path.

26.12 Timing Diagrams

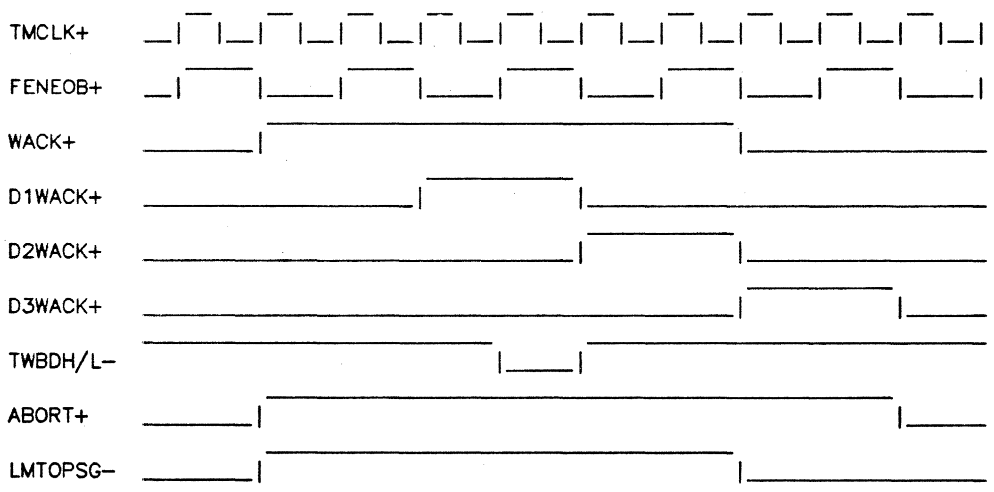


FIG. 26-7. WACK Timing Diagram

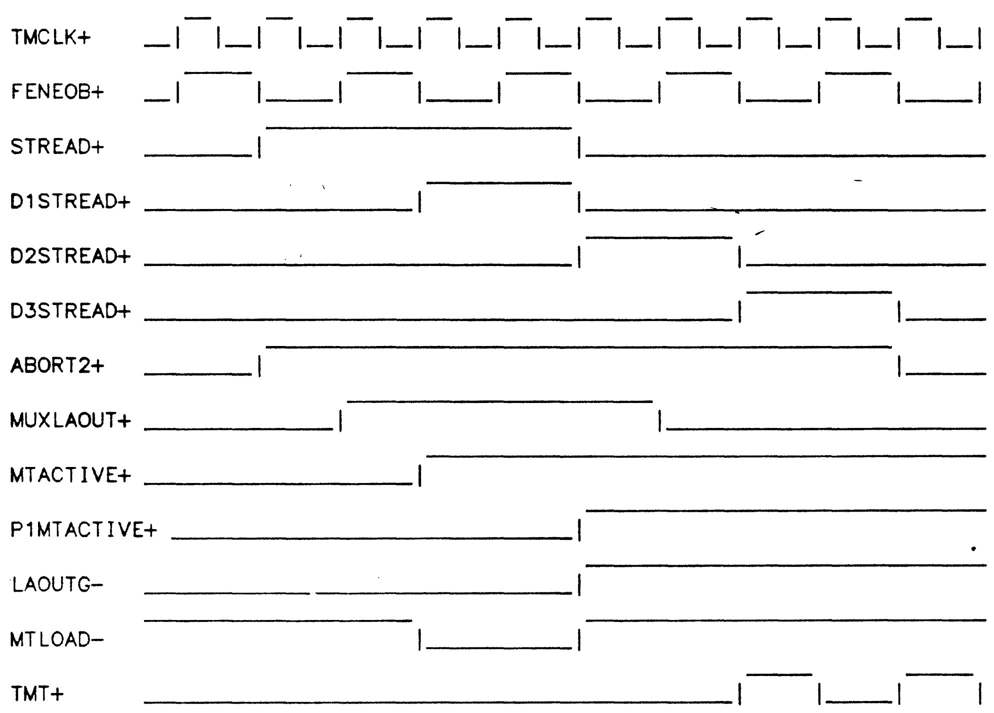


FIG. 26-8. STREAD Timing Diagram

26.13 9755 Comparisons

The MC logic for the 4150 is completely different from that of the 9755. The major differences are:

- The 4150 has one MT, as opposed to the two MTs on the 9755. Each memory word is 64 bits wide (plus check bits) and is broken down into two 32-bit half words. Each half goes to a separate memory board on the 9755, while all bits in the word go to the same board on the 4150. Since each 9755 memory board only contained 32 bits (plus check bits) a separate write or read was required for each 32 bits required, thus a separate MT was required for each half word. Each 4150 memory board contains 64 data bits. Separate MTs are, therefore, no longer needed since all data bits are written to the one board.

- In the 9755 a memory array read could only occur when the WB was empty. This insured that the CPU was not getting stale data from memory. This is not necessary on the 4150. The MT can replace stale data from memory with the fresh data from the WB "on the fly". This results in much faster throughput, since there is no time wasted in emptying the WB whenever a read is requested.

- The 9755 data bus consists of two 32-bit (plus check bits) data buses. The 4150 data bus consists of one 32 bit (plus check bits) data bus. The data is sent over to the memory board 32 bits at a time, latched on the memory array, and written into the array either 32 bits or 64 bits at a time depending on the operation specified.

- Merges on writes are fully associative on the 4150, but could only occur on <u>successive</u> mergable writes on the 9755.

- A completely new memory array board is used with the 4150, employing 256K (8 MB boards) or 1 MB (32 MB boards) DRAMs and supporting the new memory bus design.

- The 4150 is able to support 16 and 32 MB memory arrays and address up to 64 MB of main memory. The 9755 was limited to 32 MB.

- The ECC logic is completely different between the two processors. The 4150 utilizes an ECC chip to check/correct data, while discrete logic with a different code was used on the 9755.

- The MC logic on the 4150 contains DMx support since the I/O logic is tied very closely to the Memory Controller. This allows I/O data transfers to take place between the I/O interface and the MC without the use of BD.

- There are 12 refresh address bits on the 4150 while there are 8 bits on the 9755. This is to accommodate the larger RAM size on the memory array boards and to provide extra bits for future expansion if needed.

- The 4150 has Battery Back Up (BBU) capability. The MC provides refresh to the memory array boards in the case of a short term power failure to maintain the integrity of data in memory.

26.14 Partitioning

All the MC functions described in this chapter are implemented on the CMI board, which is discussed in chapter 30. BD arbitration is handled by the E unit and is discussed in 28. The MC will not drive BD until the arbitration logic on the E unit sends signal MCTOBD- to the MC. The MC drives BD while this signal is active.

27. I/O Interface Detailed Description

The Input Output (I/O) logic has the job of controlling the transfer of data from memory to a peripheral controller (Output), and the transfer of data from a peripheral controller to main memory (Input). The manner in which this transfer occurs is different than on previous Prime machines mainly because the I/O data path logic resides on the same logic board as the Memory Controller (MC). The result of this is that the data being transferred does not go across BD as it did in all previous machines for DMx operations.

Because the data on an input transfer goes directly into the DWB RAMs from the BPD transceivers, and because data for output transfers is placed into these same transceivers on a memory read, the control of I/O transfers is very closely tied to both the microcode and the memory timer logic. The address to/from BPA still goes across BD as on other CPUs, since BPA resides on a different board than the BPD transceivers.

27.1 DMx Control

The Bus Peripheral Control (BPC) signals are asserted under microcode control. They are produced by decoding microcode IACs which are clocked at CS8+ into 74AS194 register parts. The BPC signals have the same meaning and conform to the I/O specification as in all other Prime CPUs.

A request for a DMx transfer is honored by the microcode trapping out of its natural execution sequence. The microcode then enters the DMx microcode routine. This starts an I/O address phase, which is described below:

1. IAC SDEN (Set DMx ENable) is issued, which sets FDEN+ at CS8+. This signal is buffered and sent to all the controllers as signal BPCDEN+.

2. All controllers which have a data request pending assert their request signals, BPCDRQ{1:10}-.

3. The I/O arbitration logic picks the controller with the highest priority and asserts that controller's grant line DMXORINT{01:10}+. All other grant lines are reset. The controller with the highest priority is the one with the highest slot number.

4. IAC RDEN (Reset DMx ENABLE) is issued, which resets FDEN+ and BPCDEN+ at CS8+. The controller which has won control of the bus starts driving the BPC mode lines with a code which indicates what kind of DMx transfer is being requested. Table 27-1 shows these values. It also drives BPA with the necessary address calculation information.

5. IAC LBPAMOD (Latch BPA MODe lines) is issued, which causes the BPC mode lines to be latched for use in the jump net.

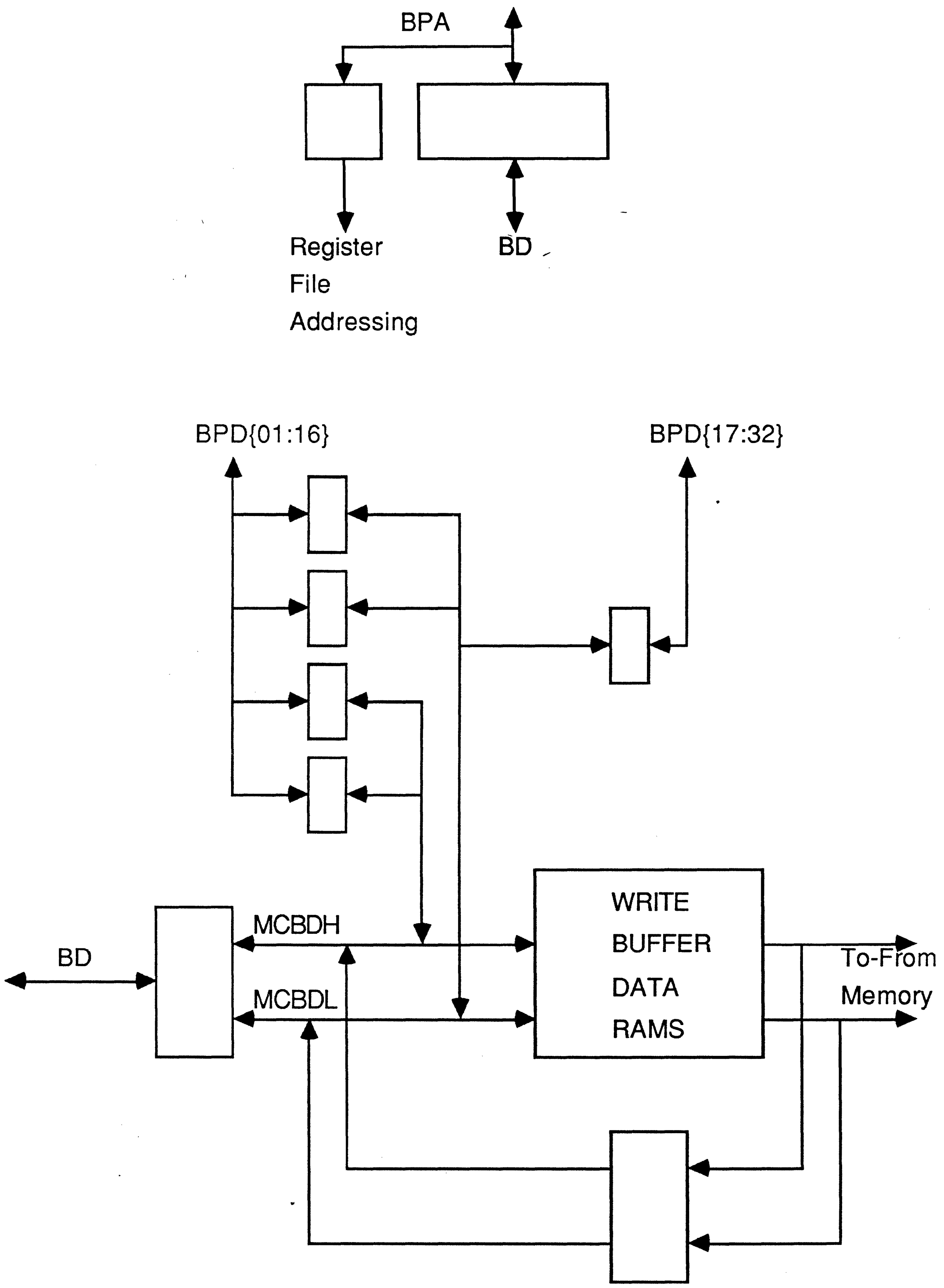


Figure 27-1 Block Diagram of I/O Interface

6. IAC SDCPN (Set DMx Clear Priority Net) is issued, which causes BPCDCPN+ to be set at CS8+. This tells the controllers that the address phase is over and that all unserviced requests should be requested again. All grant lines are reset.

7. IAC RDEN is issued, which resets BPCDCPN+ at CS8+. The microcode branches on the mode lines to determine which type of DMx transfer has been requested.

Note that IAC RDEN resets both BPCDEN+ and BPCDCPN+.

TABLE 27-1. BPC Mode Line Decoding

	BPC Mode Lines				
DMx Modes	MOD0+	MOD1+	MOD2+	MOD3+	INMOD+
DMQ	0	0	0	0	0/1
Illegal	0	0	0	1	0/1
Illegal	0	0	1	0	0/1
Illegal	0	0	1	1	0/1
16-bit DMA	0	1	0	0	0/1
16-bit burst DMA	0	1	0	1	0/1
Illegal	0	1	1	0	0/1
32-bit burst DMA	0	1	1	1	0/1
DMT	1	0	0	0	0/1
16-bit burst DMT	1	0	0	1	0/1
Illegal	1	0	1	0	0/1
Illegal	1	0	1	1	0/1
DMC	1	1	0	0	0/1
Illegal	1	1	0	1	0/1
Illegal	1	1	1	0	0/1
Illegal	1	1	1	1	0/1

BPCINMOD+ = 1 implies input transfer (to CPU)
BPCINMOD+ = 0 implies output transfer (to controller)

An illegal request will cause a machine check.

The following sections describe the data transfer when the controller requested a DMA/DMC/DMT/DMQ IN and then a DMA/DMC/DMT/DMQ OUT transfer. The data transfer phase is the same for all four types of transfers. The address calculation is slightly different in each, and may be based on data from the register file, main memory, or from the peripheral controller, depending on the type of request.

27.2 DMA/DMC/DMT/DMQ IN

1. The microcode issues IAC SSTROB (Set STROBe) which is clocked at CS8+ as ISTRB+. This is clocked one TMCLK later, is buffered, and sent to the controller as BPCSTRB+. This makes the controller start driving BPD with the data to be transferred.

2. The microcode sets and resets BPCDEN and latches the mode lines as discussed in section 27.1. This action overlaps the next address phase with the current data phase, which yields an overall time savings in cases when there are more DMx requests pending.

3. The address calculation is done:

   - For DMA, the address is taken from the address register of the DMA register pair pointed to by BPA. It is written into RMA. Both the address register and the range register are incremented. If the incremented range register value is 0, BPCEOR+ is asserted by the I/O interface hardware.

   - For DMC, the address is taken from the first memory location pointed to by BPA. This address is written into RMA. The memory location where it came from is updated with the incremented value. The original unincremented value is compared with the value in the next memory location. If these values are equal, BPCEOR+ is asserted by the I/O interface hardware.

   - For DMT, the address on BPA is written into RMA. BPCEOR+ is never asserted.

   - For DMQ, the address on BPA is the address of a Queue Control Block (QCB) in memory. A DMQ input transfer is analogous to the PMA ABQ (Add to Bottom of Queue) instruction. Queues and the ABQ instruction are discussed in the System Architecture Reference Guide. BPCEOR+ is asserted by the I/O interface hardware if the current transfer fills the queue.

4. The microcode issues IAC ENBPDIN (ENable BPD IN) to allow the BPD transceivers to turn on and drive from BPD toward the Write Buffer (WB). The same step also issues a memory write request and IAC TBPD (Trigger BPD). IAC TBPD is clocked at CS8+, and is used to clock the BPD data into the 74ALS646 transceivers. The memory write request causes the S unit to drive RMA onto BB to be used as the memory address as in a normal write operation.

5. From this point on, the MC logic takes over as it would on a normal memory write operation with one exception. The data to be written into the WB is taken from BPD instead of BD. The I/O data is written into the write buffer and then proceeds to memory as does all other data.

6. The microcode issues IAC RSTROBE (Reset STROBe), which resets BPCSTRB+. This tells the controller to stop driving BPD.

27.3 DMA/DMC/DMT/DMQ OUT

1. The beginning steps in DMA/DMC/DMT/DMQ OUT routines are the same as for DMA/DMC/DMT/DMQ IN routines. The microcode issues IAC SSTROB, overlaps another address phase, and does the address calculation as for the corresponding input operation.

   - The address calculation for DMQ OUT is an exception. A DMQ output transfer is analogous to the PMA RTQ (Remove from Top of Queue) instruction. Queues and the RTQ instruction are discussed in the System Architecture Reference Guide. BPCEOR+ is asserted by the I/O interface hardware if the queue is empty when the transfer begins. In this case, a special microcode path is taken which sends 0 to the controller as data.

2. A memory read operation is executed in place of a memory write.

3. IAC ENBPDOUT (ENable BPD OUT) is issued, which turns on the BPD transceivers and enables them to drive BPD.

4. When the data returns from memory, it is handled in the normal manner (checks and corrects for ECCC, stale memory data replaced by the WB), and it is put into the normal BD transceivers and sent to the rest of the CPU. The data is not needed on the E or S units, so it is simply ignored. As it is being processed normally, it also is being clocked into the 74ALS646 BPD transceivers via signal TBPD+, which is generated from IAC TBPD.

5. The microcode issues IAC RSTROBE, which resets BPCSTRB+. The controller uses this trailing edge to clock the data on BPD.

27.4 Burst DMA Mode

Burst DMA mode is basically the same as normal DMA except that four 16-bit words are transferred during each data phase instead of the normal one word. In order for this to happen, certain requirements must be met. There must be at least four words remaining to be transferred, and the address must be aligned on a MOD4 boundary. These conditions are verified by DMx microcode, using the E unit for any calculations. The result is sent to the EOR logic of the I/O interface, which will assert BPCEOR+ during the data phase if the burst increments the range register to zero.

If the requirements are not met the DMx microcode reverts to the normal DMA transfer sequence.

27.4.1 Burst DMA IN

1. The address phase is the same as for any DMA request.

2. IAC SSTROB is issued to set BPCSTRB+. This tells the controller to send the first 16-bit data word.

3. IAC SBSTRB (Set Burst STRoBe) sets signal FBSTRB- at CS8+. This signal is gated such that BPCBSTRB+ is active during stage 8 and inactive during stage 7. Each trailing edge of BPCBSTRB+ causes the controller to send the next 16-bit data word. The first 16-bit word is clocked in a set of BPD transceivers by signal TBPD+, generated from IAC TBPD.

4. The second word is clocked into another set of transceivers with signal TBPD2+, generated from IAC TBPD2.

5. The third word is clocked into another transceiver by signal FW64+, generated from IAC W64 (Write 64 bits). This same microcode step starts up a normal 32-bit memory write operation. What really happens is a 64-bit write operation from the BPD transceivers to the WB. IAC W64 modifies the WACK chain into a Super WACK (SWACK) chain to accomplish this. As the WACK chain starts up we already have three of the four words necessary to do the write.

6. SWACK chain logic clocks the fourth 16-bit data word into another set of BPD transceivers during D1WACK, at the same time as it is clocking the first two words into the WB from the first two sets of BPD transceivers. IAC RSTROB is issued, which resets BPCSTRB+ and also deactivates FBSTRB- and subsequently stops BPCBSTRB+ from pulsing.

7. The SWACK chain logic writes the other two data words into the WB from the last two sets of BPD transceivers during D3WACK.

27.4.2 Burst DMA OUT

Burst DMA OUT is the same as DMA OUT, except that all four data words returned from memory are clocked into the BPD transceivers. They are clocked in 32 bits at a time as they are received from memory. These are enabled onto BPD by microcode in turn. The microcode issues BPCSTRB+ and BPCBSTRB+ as it did for burst DMA IN mode.

27.5 32-Bit Burst DMA Mode

The 4150 was to have supported this new I/O mode which transfers twice as much data per data phase as regular burst mode. The implementation of this mode ran into some problems. The hardware designed into the I/O interface to support this mode was shown to be insufficient to yield the desired performance during the performance testing phase of the processor's development. It should be stressed that the mode works, and that the 4150 can support it. However, since the performance improvement of this mode over regular burst mode was only 10 - 15%, the mode was shut off. This was accomplished by making the microcode go down the normal burst mode path for all wide word burst mode requests.

Should there be a midlife kicker to this machine, this mode could be made active and productive by the addition of three 74ALS646s to the I/O interface.

27.6 Extended DMA

There is a new I/O mode called extended DMA. This new mode allows channel registers to exist in memory or in the register file for DMA transfers. If the BPA address is greater than 31 decimal (37 octal), the channel pair is in memory. If the address is equal or less than 31, the address is in the register file. The channel pairs must reside in the first four segments. Logic looks at the upper 13 of the 18 BPA address bits, and if they are all zero, the channel pair resides in the register file as previous DMA did. If any of these upper bits are non-zero, the channel pair is in memory. Extended DMA transfers proceed exactly like normal DMA transfers, except that each address phase is longer because of the fetch of the channel pair from memory.

27.7 Programmed I/O

There are four Programmed I/O (PIO) instructions which may be executed by the CPU:

1. OCP - Output Control Pulse
2. OTA - Output from the A register of the CPU
3. INA - Input to the A register of the CPU
4. SKS - Skip if Ready Set

The microcode enables the BPA transceivers to drive BPA with 16 bits which contain the opcode, function code, and controller address. It then asserts signal BPCPIO+, which tells all controllers to decode the last 6 BPA bits to see if it is its address. (Each controller has a hard-wired 6-bit device address for the purposes of this comparison. No two controllers are allowed to have the same device address.) The controller that detects its own address then drives BPCREDY-. If a controller responds ready to an INA, OTA, or SKS, the CPU drives BPCSTRB+, which causes the controller to execute the command. The microcode then resets BPCSTRB+ and BPCPIO+. The CPU skips the next instruction if a controller asserted BPCREDY-.

The data destined for the controller on an OTA is placed into the BPD transceivers and enabled onto BPD while BPCSTRB+ is active. IAC IODAT (I/O DATa) allows the BD transceivers to pass live data from the E unit to BPD, bypassing the internal transceiver registers. IODAT is used for both OTA and INA instructions. During an OTA instruction, IAC OTA enables both the BPA and BPD drivers to drive the I/O bus.

27.8 Parity Checking

Parity is checked on the data placed onto MCBD by the BPD transceivers during DMx input transfers. This operation is similar to that which takes place during normal CPU write operations. The parity checkers check for good parity as the data is being written into the WB RAMs. One minor difference is that during a burst input operation the parity is checked twice, once as the first 32 bits are being written, and again as the second 32 bits are written. On a PIO INA instruction the parity is checked on BD as it is transferred from the I/O interface logic to the E unit.

On output DMx and PIO OTA transfers, parity is checked on BPD as signal BPCSTRB+ is being reset. It is not checked on BURST transfers as each BURST controller checks parity anyway.

27.9 Required I/O Controller Revision Levels

Certain I/O controllers must be at a specified revision level to work in a 4150 or 4050 system. Table 27-2 shows these requirements.

TABLE 27-2. I/O Controller Required Revision Levels

Board Name	Slang Name	Model Number	Part Number	Revision
IDC3	Koala	6580	TLA10019-001	R
MSTC	Minnow	2382-003	TLA10234-001	R
MPC4		7010T	SPL91521-91	H
STSC	Streamer		2301-901	AA
ASYNC LAC	ICS3	CLAC304	ESA10063-001	C
BMTC	Marlin		2023-001	N
PNC II			2384-001	G

27.10 Timing Diagrams

FIG. 27-2. BPCSTRB and BPCBSTRB Timing

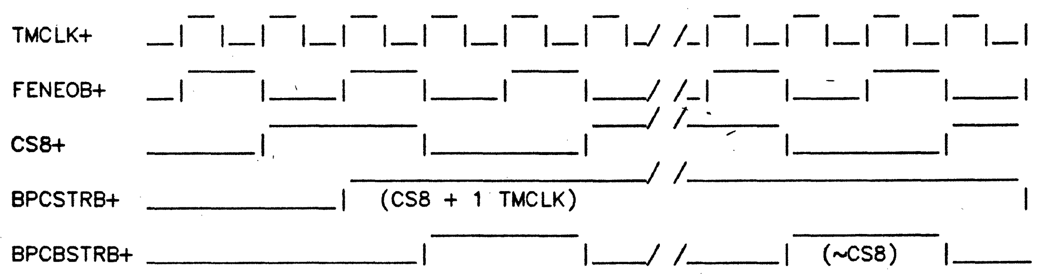
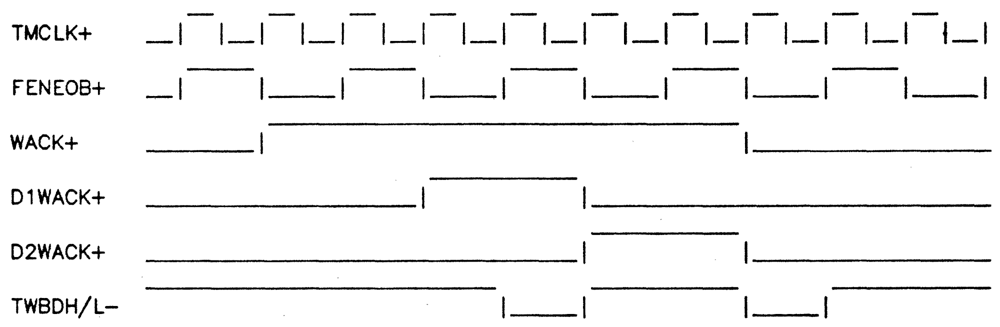


FIG. 27-3. 64 Bit Write Timing (SWACK Chain)



27.11 9755 Comparisons

The logic is implemented completely differently from the 9755 and all other Prime CPUs. The differences are so vast that it is easier to list the similarities:

1. Both machines transfer data between main memory and I/O controllers using a combination of hardware and microcode control.

27.12 Critical Paths

During an input burst mode operation the SWACK chain writes 64 bits into the WB RAMs in the same time a normal write does 32 bits. The timing is critical throughout this operation. Refer to Figure 27-3. After the first 32 bits are written at the end of D1WACK+ three critical things must happen in the next beat:

1. Hold time must be met for the data just written.

2. The least significant bit of the WB pointer must be changed.

3. The BPD drivers which were driving the first 32 bits must be shut off, and the BPD transceivers with the second 32 bits must be turned on.

27.13 Partitioning

The BPA interface and I/O arbitration logic (BPC request and grant lines) are implemented on the E board, which is discussed in chapter 32. The rest of the BPC interface, the BPD interface, and the EOR hardware are implemented on the CMI board, which is discussed in chapter 30.

28. Bus D Arbitration Detailed Description

The Bus D (BD) arbitration logic is needed to prevent TTL tristate clashes. BD can be driven by any CPU unit, but the E unit is normally driving the bus. To prevent any tristate clashes, the E unit always stops driving BD one TMCLK+ before some other unit needs control of the bus. When the other unit no longer needs control of the bus, that unit stops driving BD at CS7+, and the E unit starts driving at least one TMCLK later.

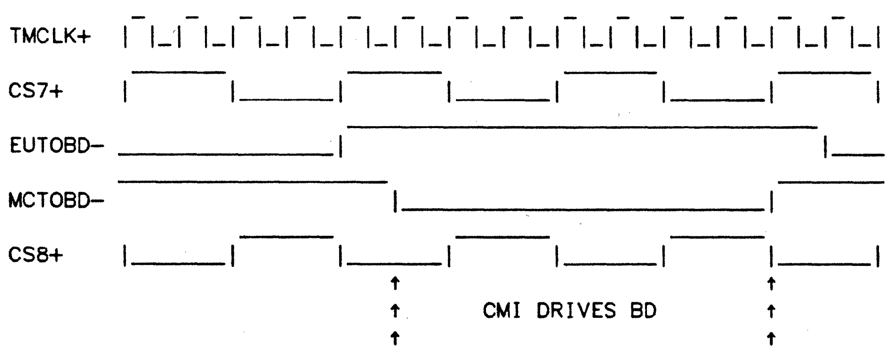
28.1 E Unit

The signal EUTOBD- tells the E unit that no other unit is driving BD and the E unit can now drive it. In most situations the E unit is driving the bus.

28.2 MC Unit

The signal MCTOBD- goes from the arbitration logic to the MC. This signal tells the that no other unit is driving BD and the MC can now drive it. The timing for this tran is shown in Figure 28-1.

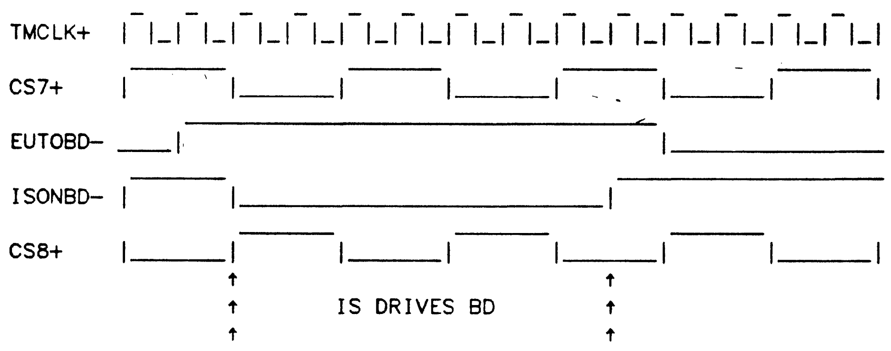
FIG. 28-1. Transfer of BD Control, E to MC



28.3 S Unit

The signal ISONBD- goes from the arbitration logic to the S unit. This signal tells the S unit that no other unit is driving BD and the S unit can now drive the bus. Figure 28-2 shows the timing of this transfer.

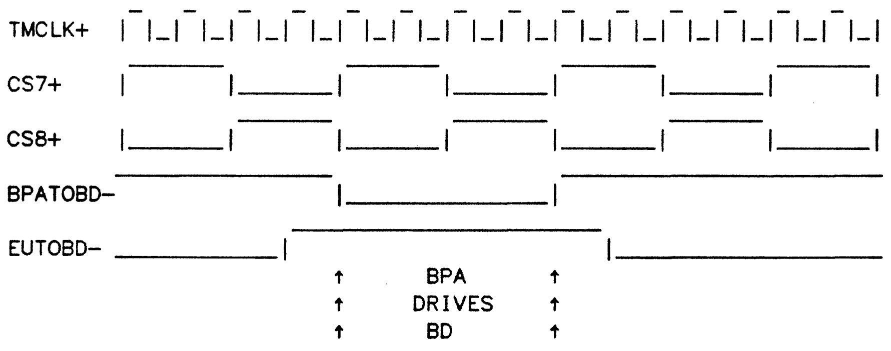
FIG. 28-2. Transfer of BD Control, E to S



28.4 BPA

The signal BPATOBD- tells the I/O interface logic that no one else is driving BD, and th
BPA can be placed on BD. The BPA logic only drives BD during DMx address phase. I
only drives BDH{1:16}+ and BDL{15:16}+, so parity can only be checked on BDL during t
time. Figure 28-3 shows the timing of the control transfer.

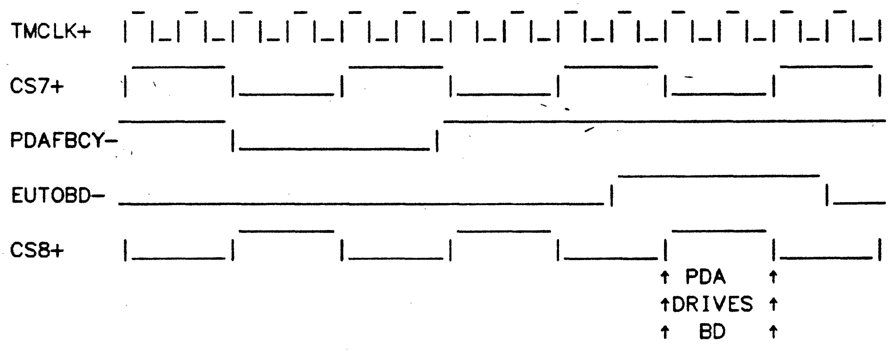
FIG. 28-3. Transfer of BD Control, E to BPA



28.5 PDA

The PDA can drive BD during a FORCEBCY operation, or when loading either memory or the decode net. During this time the PDA drives only BDH{01:16}+, so BD parity must not be checked at this time. Normally, if no other unit is driving BD, the E unit takes control of the bus. During a BDH branch, BD is selected as the ALEG of the ALU, causing the E unit to stop driving BD. By default, the PDA drives BDH data. Figure 28-4 shows the timing of this control transfer.

FIG. 28-4. Transfer of BD Control, E to PDA



28.6 VLSI Usage

No VLSI chips are used in the implementation of the BD arbitration logic.

28.7 Critical Paths

The critical path through the BD arbitration logic starts at CS7+ and ends half a beat later. The current BD driver must be shut off in this time to prevent tristate clashes with the new BD driver, which will be getting on the bus starting at CS7.5+

28.8 9755 Comparisons

The 9755 was an ECL machine. No TTL tristate clashes could occur so there was no BD arbitration logic.

28.9 Partitioning

The BD arbitration logic is implemented on the E board, which is discussed in chapter 32.

29. Processor Diagnostic Aid Detailed Description

This chapter details the operation of the hardware component of the PDA. Details of the host software functionality can be found in the PDA Host User's Guide.

The most useful feature of the PDA is to dynamically capture the the state of an executing machine.

The PDA can be divided into several functional parts: stack, halts/delays, event counter, control store interface, sense registers, BDH interface, and the microprocessor.

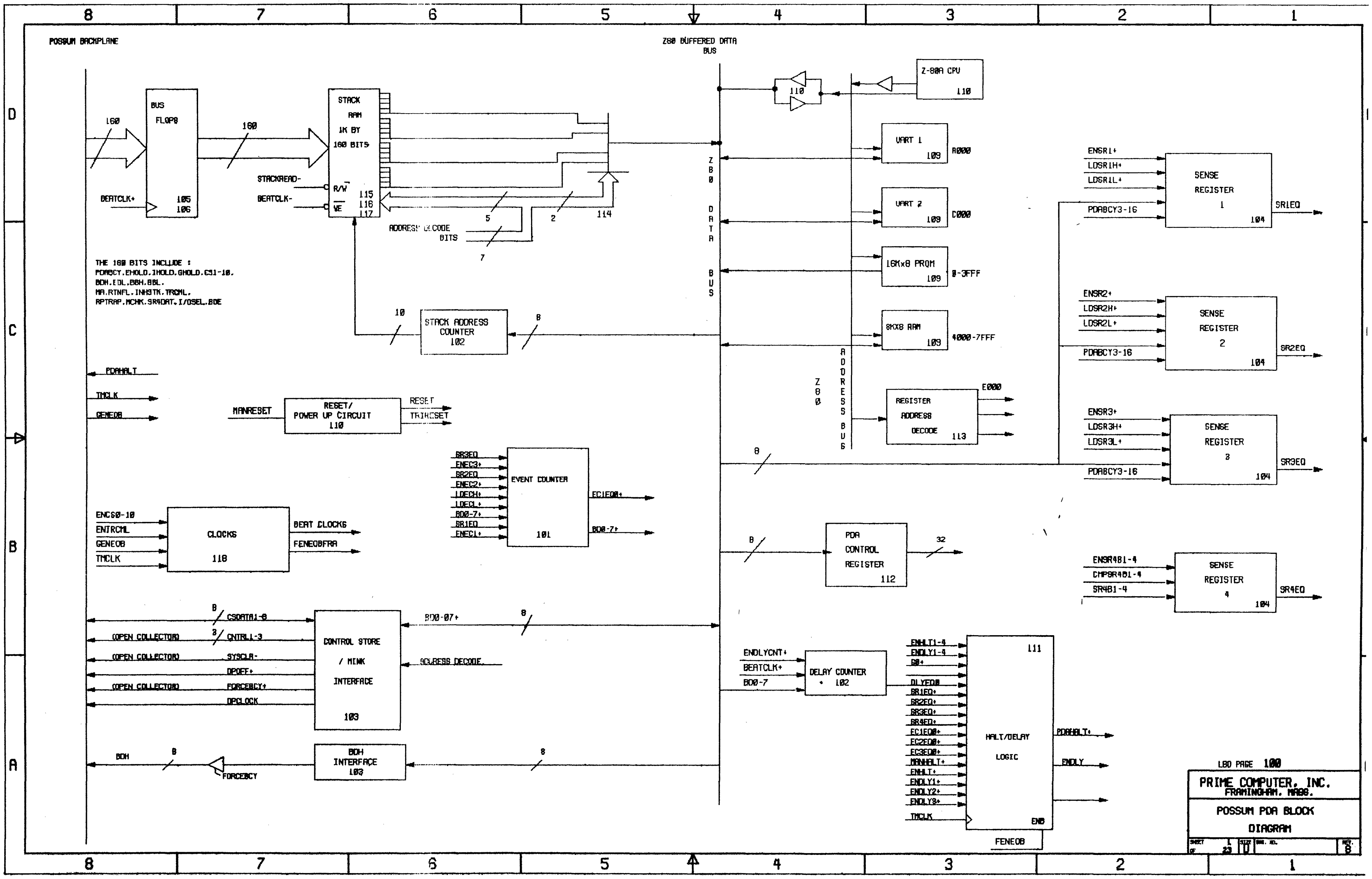

POSSUM BACKPLANE
Z80 BUFFERED DATA BUS
BUS FLOPS 105 106
STACK RAM 1K BY 160 BITS 115 116 117
STACKREAD-
BEATCLK-
BEATCLK+
ADDRESS DECODE BITS
114
Z-80A CPU 110
UART 1 109 A000
UART 2 109 C000
16Kx8 PROM 109 0-3FFF
8KX8 RAM 109 4000-7FFF
REGISTER ADDRESS DECODE 113 E000
Z80 DATA BUS
Z80 ADDRESS BUS
THE 160 BITS INCLUDE : PDABCY.EHOLD.IHOLD.GHOLD.CS1-10. BDH.EDL.BBH.BBL. MA.RTNFL.INHSTK.TRCML. RPTRAP.MCHK.SR4DAT.I/OSEL.BDE
STACK ADDRESS COUNTER 102
PDAHALT
TMCLK
GENEOB
MANRESET
RESET/ POWER UP CIRCUIT 110
RESET
TRIRESET
EVENT COUNTER 101
SR3EQ ENEC3+ SR2EQ ENEC2+ LDECH+ LDECL+ BD0-7+ SR1EQ ENEC1+
EC1EQA+
BD0-7+
ENCS0-10 ENTRCML GENEOB TMCLK
CLOCKS 110
BEAT CLOCKS
FENEOBFRA
PDA CONTROL REGISTER 112
CSDATA1-8
CNTRL1-3
SYSCLR-
DPOFF+
FORCEBCY+
DPCLOCK
(OPEN COLLECTOR)
CONTROL STORE / MINK INTERFACE 103
BD0-07+
ADDRESS DECODE
BDH
FORCEBCY
BDH INTERFACE 103
ENDLYCNT+ BEATCLK+ BD0-7
DELAY COUNTER 102
ENHLT1-4 ENDLY1-4 G0+ DLYEDB BR1EQ+ BR2EQ+ BR3EQ+ BR4EQ+ EC1EQ0+ EC2EQ0+ EC3EQ0+ MANHALT+ ENHLT+ ENDLY1+ ENDLY2+ ENDLY3+ TMCLK
HALT/DELAY LOGIC 111
PDAHALT+
ENDLY
FENEOB
ENSR1+ LDSR1H+ LDSR1L+ PDABCY3-16
SENSE REGISTER 1 104 SR1EQ
ENSR2+ LDSR2H+ LDSR2L+ PDABCY3-16
SENSE REGISTER 2 104 SR2EQ
ENSR3+ LDSR3H+ LDSR3L+ PDABCY3-16
SENSE REGISTER 3 104 SR3EQ
ENSR4B1-4 CMPSR4B1-4 SR4B1-4
SENSE REGISTER 4 104 SR4EQ
LBD PAGE 100
PRIME COMPUTER, INC. FRAMINGHAM, MASS.
POSSUM PDA BLOCK DIAGRAM
SHEET 1 OF 23

Figure 29-1 Block Diagram of PDA

29.1 Stack

The stack on the PDA is a pre-increment stack, meaning that the stack address increments prior to the write pulse. The write pulse is only inhibited during a stack read cycle and therefore is always happening unless a stack read is in progress. If there are any inhibits on any particular beat the stack address remains the same for the next beat, overwriting the inhibited data.

The stack and its associated logic comprise the most critical path in the PDA. Once every beat the logic on the PDA must sample the state of the CPU busses, store that data into the stack RAMs and sample data for the next beat. Due to the timing involved, the data is stored in registers which are clocked by BEATCLOCK+, a PDA version of FENEOB+. Another version of BEATCLOCK+ called STKCLK+ is used to clock the stack address counters that determine the stack address.

Due to the possibility of skew, part delay and write pulse delay, meeting worse case design constraints was accomplished only by considerable tuning of the write pulse.

29.1.1 Inhibits

Stack inhibits are used to effectively get more stack depth by gating off the clock to the stack address counter during certain conditions.

29.1.1.1 Microcode Label Trace

The PDA can be made to trace microcode labels only, by only allowing stack address counter to increment when TRCML+ is active. In this case only the beats during which TRCML+ is active will be stored in the stack.

29.1.1.2 EHOLD / GHOLD

The PDA can **Inhibit on GHOLD** or **Inhibit on EHOLD** which prevents the stack address counter from clocking during a GHOLD+ or an EHOLD+ condition.

GHOLD+ is a one beat signal that happens at the beginning of each cache miss. EHOLD+ is a signal that remains on for the duration of a cache miss or other pipeline hold condition.

29.1.1.3 Multi-Microcode

The operator can set an **Inhibit on Multi-Microcode** condition, in which case the stack address counter is prevented from clocking whenever there is no INIT- clock. This effectively allows only the microcode's PMA entry points to appear in the stack. All other microcode is inhibited from the stack. It appears that the stack is saving only the PMA level instructions. Certain entry points do not indicate the true instruction under execution. Examples of this are indirection and shared PMA entry points such as BCOM16.

29.1.2 Storing Data into the Stack

74AS574s take a snapshot of the backplane every beat. Every rising edge of BEATCLOCK+ makes these devices clock the data and keep it steady for the proper setup and hold times for the stack RAMs. Some of these registers feed the data through 2 to 1 muxes. This allows some testing of the stack RAMs by feeding known good diagnostic data into the data inputs of the stack RAMs (1K x 4 static RAMs with separate input and output pins).

The clock signals are sampled by 74AS194s at TMCLK+ with a clock enable connected to FENEOBFRA+, the free running enable end of beat. BBH and BBL need special treatment because they are driven onto the backplane by a VLSI chip. The PDA uses 74ALS573s to latch this data before the end of beat and 74ALS574 registers to hold this data at the end of beat. This technique is also used on other less critical signals.

29.1.3 Reading Data from the Stack

The PDA stack RAM outputs are tristated together in groups of 5. These groups go through a 4 to 1 mux onto the Z80 data bus. The read of the stack is *much* less critical than the write, since it is done by the Z80 at its speed, and the RAMs are very fast relative to the Z80. The operation of reading the stack is done by disabling the write pulse (under Z80 control), incrementing the stack once (done by a Z80 command), and reading the 20 bytes of data at that stack location. Reading each of the 20 bytes is done by mapped memory on the Z80 microprocessor. The Z80 then increments the stack address and the operation continues until the entire stack is read.

29.2 Sense Registers

The sense registers on the 4150 PDA detect when a specific condition exists in the CPU. This allows the operator to halt the CPU to determine why that condition exists, or to trigger the stack which will make information about how the machine ended up in that state available to the operator.

29.2.1 SR1 and SR2

Sense registers 1 and 2 have the ability to signal a match condition when they determine that a specific microcode address has been executed. Some microcode address of interest is placed in the sense register, which is actually a set of 8-bit comparators with latches on the input. When the microcode address (or BCY) of interest appears at the inputs, it signals the match on the output and is registered. This registered output is used in the halt and delay circuits.

29.2.2 SR3

Sense register 3 will signal the halt/delay circuit that a virtual address match has occurred. SR3 is connected to the Bus Virtual Memory Address (BVMA), and can be set to examine EA (BVMA during TRCML+) or RP (BVMA during CS1+). RP values by definition must be even, that is, the least significant bit must be zero.

29.2.3 SR4

Sense register 4 is the sense register you've been waiting for. Until now, the PDA had to determine in advance what the operator might want to compare data against. Now, SR4 has inputs coming from all the CPU boards, allowing SR4 to look at any signal in the machine.

Each CPU board has an open collector driver that receives input from testpoints which are placed strategically around the board. The output of the driver is connected in a wired-AND configuration with the other CPU boards on the backplane. These four signals feed into the PDA and become SR4. The operator has the option of setting a trigger or halt based on any SR4 pattern.

Caution: Wires left connected to these binding posts *must* be removed in any production systems. This is due to the fact that on some occasion, a board that has something wired to it may be returned for work and at that time a technician may wire something else into SR4 and this may cause misleading results with 2 or more signals driving SR4.

29.3 Halts and Delays

The halt and delay triggering in the 4150 PDA is functionally equivalent to previous machines.

29.3.1 Halts

Halts happen when the operator sets a halt condition on some sense register match. When this condition is set, the Halt/Delay logic generates a signal called GPDAHALT-, which goes to a register. The clocked version, PDAHALT-, goes to all the CPU boards, which suspends all clocks except TMCLK+ and those required for memory refresh.

29.3.2 Delays

Delays work in much the same way as halts except the Halt/Delay logic generates GDLYTRIG+ (instead of GPDAHALT-), which starts the delay counter. When the delay counter reaches zero the stack address counter stops issuing new stack addresses, and the PDA holds the stack data. Write pulses continue to occur, continually overwriting the last pre-incremented address. The last entry in every triggered stack dump is always trash because of this action.

29.4 Event Counter

The event counter in the 4150 PDA allows the operator two basic operations. First, it allows the operator to trigger the stack or halt the CPU when, for example, thirty-five iterations of LDA have happened. The other operation it can do is to count the number of LDAs that have been executed.

The counted object could be such things as SR1, SR2, SR3, SR4 matches, GHOLDs, or DMx operations. One unused input is available on the multiplexer that controls this counted event. This allows seat-of-the-pants debug; if the operator wishes to see some other event counted, s/he can add one wire and see the desired count.

Warning: The event counter counts beats, not occurrences. For example, if it is set to count occurrences of DMx, and one DMx operation occurs during which the signal FDMX+ is active for 15 beats, the event counter will be incremented by 15, not 1.

29.4.1 Loading

The event counter is loaded from the Z80 bus, and it requires that a single enable end of beat (SENEOB+) be generated by the microprocessor.

29.4.2 Reading

The Z80 in the PDA can read the value of the event counter by doing a read from the event counter memory mapped address. There are 4 bytes of data in the event counter and the microprocessor needs to do memory mapped reads from four different locations.

29.4.3 Selecting Events to Count

The event counter incorporates a multiplexer as described above to enable the counter to count a variety of specific events. The output of this mux is gated into the clock circuits of the counter to only allow a clock pulse into the counter when the mux sees the prescribed event.

29.5 Control Store Interface

The control store interface allows the PDA to load and read control store. It employs a parallel tristate interface and four open collector control lines for control.

29.5.1 Reading/Writing Control Store

The PDA can read and write the control store. To do this, it first asserts DPOFF-A and then SYSCLR+B (2 of the control lines). This tells the control store that an access is to occur. There are four open collector control lines that the control store unit looks at to determine which operation to do. These control lines are to be physically connected to identical control lines coming from the diagnostic processor. These control lines are made up of three control bits that determine the operation to be performed by the CS unit, and one clock signal which is sent whenever the data on the DPDATA bus becomes valid. The control signals are be manipulated by Z80 software. The function of DPOFF-A is to tell the Diagnostic Processor that the PDA has control of the bus, and that it should not interpret the switching on the control lines as requests from the CPU. It also forces the Diagnostic Processor to force its tristate data bus drivers into their high impedance state.

The four control signals are CNT1, CNT2, CNT3, and DPCLOCK. Their decoding is explained in Table 29-0. The signals are left in a high state when unused due to their open collector nature.

Operation	CNT1	CNT2	CNT3	DPCLOCK
	CPUACK+A	DPFULL+A	CPUREQ+A	DPREQ+A
NOP	1	1	1	+
Load RBCYL	1	0	1	+
Load RBCYH	1	0	0	+
Read CS	0	1	1	+
Write CS	1	1	0	+
Begin	0	0	1	+
Unused	0	0	0	+
Unused	0	1	0	+

TABLE 29-1. Control Store Command Table

Further details of the Diagnostic Processor interface are discussed in chapter 16.

29.6 BDH Interface

The BDH interface allows the PDA to load CPU memory, decode net, and force microcode execution address (FORCEBCY). Much of what is done with this interface is under control of the CPU microcode. In order to initiate any operation of this interface, the PDA asserts PDAFRBCY- to the CPU. This signals the CS to vector to control store location '77. At this location, the microcode reads BDH and operates on its value. During this time the PDA takes control of BDH and drives it with a command which tells the CPU what operation it is requesting.

When the PDA decides to use the BDH interface for some function, it starts a chain of events that synchronizes with TRCML+ and drives the signal PDAFRBCY- to the CPU for one beat. After two CS8+ clocks occur, the PDA is driving BDH to the E unit for it to operate on. The next CS7+ will cause the PDA to release BDH and cause execution to take place by microcode control.

FIG. 29-2. PDA BDH Interface Fields

```
                 F |               |    C    |    D
                 O |               |    O    |    A
                 R |               |    M    |    T
                 C |               |    M    |    A
                 E |               |    A    |
                 B |               |    N    |
                 C |               |    D    |
                 Y |               |         |
                   |               |         |
               ------------------------------------------------
Bit Number     | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |   9 - 16    |
               ------------------------------------------------
```

```
         ------------------------------------
BDH      | COMMAND BYTE   |     DATA BYTE    |
         ------------------------------------
           1             8  9              16
```

Command byte:	Data byte interpreted as:
\$01	\$dd = high address byte
\$02	\$dd = low address byte
\$03	\$dd = high data byte
\$10	\$dd = low data byte
\$11	\$xx = don't care; write data to memory; increment memory address
\$12	\$dd = last data byte for DNET; write to DNET; increment DNET address

NOTE: DNET is 20 bits wide; therefore three bytes are needed to write a word. Only the top nibble of the third byte is used.

NOTE: When the high bit of BDH is set, the rest of BDH is interpreted as a BCY location for a BDH branch:

```
BDH1     = 1
BDH2-8   = gets latched to BCYH
BDH10-16 = gets latched to BCYL
```

TABLE 29-2. Table of PDA BDH Commands

29.6.1 FORCEBCY

If the PDA puts a "1" in the most significant bit of BDH the microcode interprets the command as a FORCEBCY. In this case, the least significant 14 bits, BDH 3 - 16, are interpreted as a microcode address, and the microsequencer will then vector off to that address.

29.6.2 Load Decode Net

The load of the decode net is executed in three steps, load high address, load low address, and load data.

29.6.2.1 Load High Address

Loading the decode net is accomplished by first signalling to the microcode that a decode net address is to be specified. The code for loading the high side of the address is in bits 6 - 8. Bits 9 - 16 contain the high side of the decode net address to be loaded.

29.6.2.2 Load Low Address

The low side address is specified in a similar manner. The command for loading low side address is in bits 6 - 8, and the address bits are in data bits 9 - 16 again.

29.6.2.3 Load Data

The decode net data is loaded in a similar fashion by asserting the command 'Load first decode net byte', and by sending the most significant byte of the decode net data in the data area. This operation is repeated by loading the second and third decode net data bytes (the decode net has 20 bits per entry). When asserting the command for the third byte of decode net data, the decode net data is taken from temporary registers in the CPU and actually placed in the decode net. This happens under microcode control. The microcode then increments the decode net address so that the PDA doesn't have to load the next address.

29.6.3 Load Memory

To load memory from the PDA a similar operation to loading the decode net is performed. The differences are: the commands for loading the address are different, and there are only 2 data bytes for each address. The second byte writes the data to memory and increments the address pointer under microcode control.

29.7 Microprocessor

As previously mentioned, the 4150 PDA uses a Z80 microprocessor operating at a clock speed of 2.5 MHz for a resultant cycle time of 400 nanoseconds. The following is the memory mapping for the microprocessor.

```
PDASTAT0 EQU     $E000      Read only
PDASTAT1 EQU     $E001         "
CNTR1.   EQU     $E002      Write Only
CNTR2    EQU     $E003         "
CNTR3    EQU     $E004         "
CNTR4    EQU     $E005         "
HSR1H    EQU     $E008         "
HNEWCLK  EQU     $E009         "
```

```
HSR1L     EQU     $E00A         "
HSR2H     EQU     $E00B         "
LDBDHL    EQU     $E00C         "
HSR2L     EQU     $E00D         "
RDCPDAT   EQU     $E00E      Read Only
HSR3B1    EQU     $E010      Write Only
HSR3B2    EQU     $E011         "
HSR3B3    EQU     $E012         "
HSR3B4    EQU     $E013         "
SNDCPDAT  EQU     $E014         "
CLKCPDAT  EQU     $E015         "
ENCSCTRL  EQU     $E016         "
CNTR5     EQU     $E017         "
EXDHNH    EQU     $E01B         "
EXDHNL    EQU     $E01C         "
LDBDHH    EQU     $E01D         "
ENDLY     EQU     $E01E         "
RESET     EQU     $E01F         "
STACK     EQU     $E020      Read Only
BYTE01    EQU     $E020         "
BYTE02    EQU     $E021         "
BYTE03    EQU     $E022         "
BYTE04    EQU     $E023         "
BYTE05    EQU     $E024         "
BYTE06    EQU     $E025         "
BYTE07    EQU     $E026         "
BYTE08    EQU     $E027         "
BYTE09    EQU     $E028         "
BYTE10    EQU     $E029         "
BYTE11    EQU     $E02A         "
BYTE12    EQU     $E02B         "
BYTE13    EQU     $E02C         "
BYTE14    EQU     $E02D         "
BYTE15    EQU     $E02E         "
BYTE16    EQU     $E02F         "
BYTE17    EQU     $E030         "
BYTE18    EQU     $E031         "
BYTE19    EQU     $E032         "
BYTE20    EQU     $E033         "
ENEXT     EQU     $E038      Write Only
DLYCTH    EQU     $E03C         "
DLYCTL    EQU     $E03D         "
PDAREQ    EQU     $E03E         "
LDEC1     EQU     $E040         "
LDEC2     EQU     $E041         "
LDEC3     EQU     $E042         "
LDEC4     EQU     $E043         "
FRCWRT    EQU     $E044         "
RDEC1     EQU     $E048      Read Only
RDEC2     EQU     $E049         "
RDEC3     EQU     $E04A         "
RDEC4     EQU     $E04B         "
HSR4      EQU     $E04C      Write Only
FRBCY     EQU     $E04D         "
SENEOB    EQU     $E04E         "
```

The Serial Ports are configured as follows:

```
STANDALONE PORT READ DATA     -    A000
STANDALONE PORT READ STATUS   -    A001
STANDALONE PORT READ MODE REG -    A002
STANDALONE PORT READ COMMAND  -    A003
STANDALONE PORT WRITE DATA    -    A004
STANDALONE PORT WRITE STATUS  -    A005
STANDALONE PORT WRITE MODE    -    A006
STANDALONE PORT WRITE COMMAND -    A007

HOST PORT READ DATA     -    C000
HOST PORT READ STATUS   -    C001
HOST PORT READ MODE REG -    C002
HOST PORT READ COMMAND  -    C003
HOST PORT WRITE DATA    -    C004
HOST PORT WRITE STATUS  -    C005
HOST PORT WRITE MODE    -    C006
HOST PORT WRITE COMMAND -    C007
```

29.8 PDA Configurations

The 4150 PDA can be configured in a variety of fashions.

29.8.1 PDA Debug Configuration

This typically will use a Mink Diagnostic Processor, an assignable line, and a terminal. The stand alone port of the PDA is set up as a debug port. From this port, which is accessed using the Mink 'MO PDA' command, a variety of debug routines that aid in diagnosing PDA problems are available. The assignable line can be used for verifying that communication is possible between the host system and the PDA board.

The self diagnostics that are available from the stand-alone port are described below:

29.8.1.1 PDA Self Diagnostics

The PDA supports self diagnostics via the stand alone port that is accessible via the Mink's 'MO PDA' command. If the CPU is running and executing functional code, the operator can type 'VERY H' or 'VERY L' once s/he has the PDA> prompt. This will cause the PDA self diagnostics to run. The H argument for the VERY command means that the user wants the diagnostics to stop when an error is encountered, and the L argument means that the operator wants the PDA to loop on the error test, thereby allowing a technician to find the source of the problem. PDA stand alone code will print an error message on the screen indicating the circuit in error and will give the technician ideas on where to start.

The following are the possible PDA error messages while running VERY:

- Delay Counter Carry Out Test Failed. Check for inability to load.
- Memory Test Failed.
- Sense Register Test Failed. Check for functional microcode running first.

- SR1 TEST FAILED.
- SR2 TEST FAILED.
- Delay Trigger Test Failed. Check for proper clocks.
- Event Counter Test Failed.
- End of pass.
- Verify passed.
- Verify failed.

29.8.2 Manufacturing CPU Debug Configuration

The PDA requires an assignable AMLC line from a host CPU which is executing the PDA HOST CODE. Connected to this host CPU will be a PT200 terminal. The PT200 is needed due to host code dependency on its 132 column format.

29.8.3 Customer or Field System Configuration

If a customer system or a system at some other field site has another known good system, then that site can do a similar setup as the manufacturing configuration. If this system is the only one available and it requires the use of a PDA, the PDA can be inserted into the system under test. The host code and microcode database can be installed on the system under test and an assignable line from the system under test must be connected to the PDA. A phantom user can set up the conditions under which the PDA should trigger, and it can then be left for the offending error to happen. At this point, presumably, the system has halted, and once the system is restarted, the system can initiate a process to get the stack dump information and store it in a file. The stack dump of the offending error can be used to determine the problem.

The PDA will retain the stack dump information in all cases unless, of course, the power has been removed from the system.

29.9 How to Debug the PDA

The debug station for the 4150 PDA should include the following:

- PT200 terminal
- Z80 emulator

- Mink Diagnostic Processor
- Fully functional 4150 CPU

The following is the progression of steps in verifying PDA functionality:

- Verify Z80 executes in wait loop
- Verify stand alone communication via Diagnostic Processor.
- Verify communication status from host system.
- Update and load control store from PDA and determine that it in fact is loaded into control store by using Diagnostic Processor "MO ECS" command.
- Execute PDA self verify tests from Diagnostic Processor terminal.
- Start system executing microcode and observe stack dumps.
- Verify sense registers 1, 2, and 4 in both halt and trigger modes.
- Verify SR3 operation by getting virtual memory addresses changing through diagnostics
- Test GBEAT functionality by halting CPU and examining stack dumps

29.10 Timing Diagrams

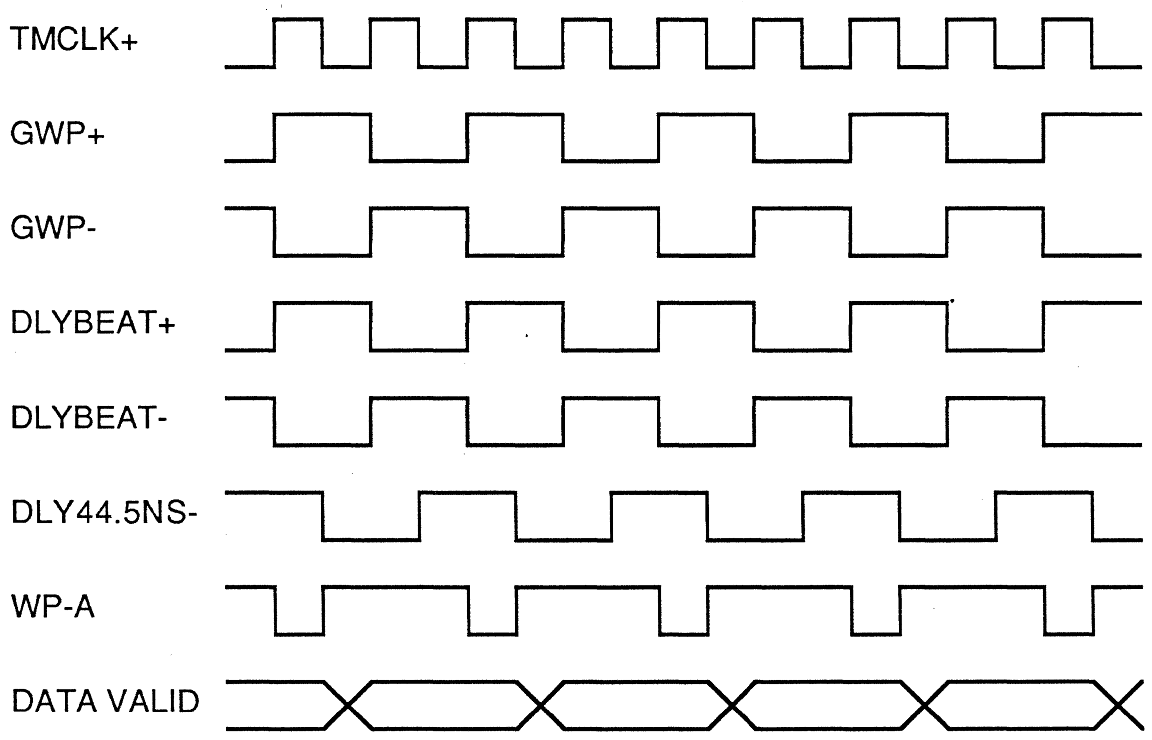


FIG. 29-3. PDA Stack Write Pulse Timing

29.11 Critical Paths

The critical paths on the 4150 PDA are the stack write pulse, the setup and hold times on the stack, and the halt path from a sense register match to the issuance of PDAHALT-.

29.11.1 Stack Write Cycle

The stack write cycle has proven to be the most critical path in the PDA. The path involves sampling data from the backplane and the rest of the CPU and storing it into registers while the address for the stack becomes stable. At this point, the write pulse, which is shaped by the use of delay lines, stores the data into the stack RAM.

29.11.2 Halt Path

The timing from the sense register match to finally result in the PDAHALT- signal going active in 2 beats is a major path.

29.11.3 Trigger Path

This path involves getting a sense register match to result in a stack trigger in two beats.

29.12 VLSI Usage

No VLSIs are used in the 4150 PDA.

29.13 9755 Comparisons

29.13.1 Self Diagnostic Mode

The 4150 PDA does not support stand alone functionality as the 9755 FEP did. It does support self diagnostics.

29.13.2 Stack

The 4150 PDA uses a 1K deep stack in contrast with the 9755 which has a 256 location stack. The stack uses Random Access Memory with separate I/O pins (AMD9150).

29.13.3 Trigger Conditions

29.13.3.1 SR2 Enables SR3 (New)

This new halt/trigger condition is a combination of an SR3 match enabled by an SR2 match. This provides the function of halting/triggering on a match of both the Effective addresses and BCY. For example, the PDA can halt the machine on any specific instruction that references a predefined effective address.

29.13.3.2 SR1 Enables SR2 (New)

This new condition allows the operator to halt/trigger the stack on the occurrence of an SR2 enabled by an SR1 match. This uses a registered SR1 match signal, that cannot be cleared until a RESET- becomes active, logically ANDed with a live SR2 match. This is a new feature in the 4150 PDA.

29.13.3.3 SR1 Logically ANDed with SR4 (Deleted)

There is a capability in the 9755 FEP to halt/trigger on the occurrence of an SR2 match logically ANDed with an SR4 match. This will not be included in the 4150 PDA.

29.13.4 Event Counter

The 4150 PDA has an Event Counter. An event counter is a new feature to this design.

29.13.5 Power Up Circuit

The 4150 PDA has a circuit that disables all tristate busses on power up and manual reset. It does NOT disable the Z80 RAM. This tristate condition can be disabled with a single Z80 write to a known location.

29.13.6 BVMA Interface

The BEMA bus on a 9755 and the BVMA bus on a 4150 are basically the same thing. Similarly, the 4150 PDA has BVMA in the stack and as an input to sense register 3 as in the 9755. The BVMA bus is active low.

29.14 Partitioning

All functions described in this chapter are implemented on the PDA board. The PDA board is discussed in chapter 33.

30. CMI Board Discussion

The CMI board contains logic from three major functional units from which it derives its name:

- Control Store logic,
- Memory Controller logic, and
- Input/Output (I/O) logic.

The CMI board also contains the Diagnostic Processor (DP) interface, as well as parts of the PCU, the decode net, and Effective Address Formation (EAF) logic. These functions are described in detail in the chapters cited below:

- Control Store - chapter 19
- Memory Control - chapter 26
- I/O - chapter 27
- Diagnostic Processor (DP) Interface - chapter 16
- EAF logic - chapter 21
- Decode net - chapter 20
- PCU - chapter 17.2

The CMI board is a combination of the 9755 CS board, which contains the Control Store and I/O logic, the MC board, which contains the Memory Controller logic, and miscellaneous logic from the I board.

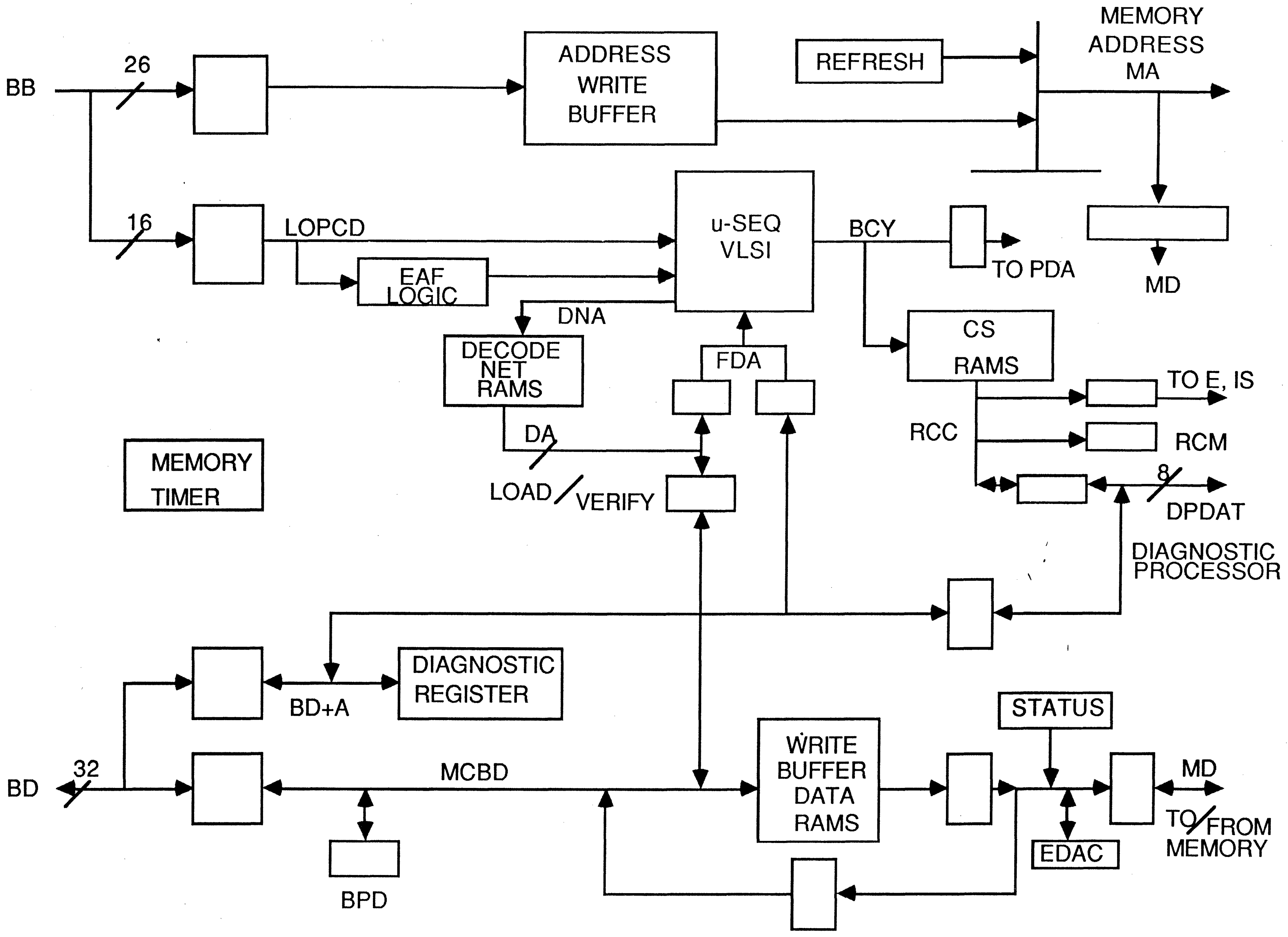


Figure 30-1 CMI BOARD BLOCK DIAGRAM

30.1 4150 And 4050 Differences

The CMI holds the master clock crystal oscillators. The nominal frequency in the 4150 is 32 MHz, while it is 30 MHz in the 4050.

30.2 Programmable Parts

30.2.1 PALs

30.2.1.1 Bus Enable

The PAL at site 04H controls the loading of data into latches MCBD and MDAT, as well as the enabling of data onto MDAT.

30.2.1.2 Burst OK & EOR

The PAL at site 06D contains the logic for determining the BURST OK logic, enable burst strobe, and the End Of Range (EOR) logic.

30.2.1.3 Miscellaneous Memory Control

The PAL at site 08M controls signals to allow the refresh counter to be enabled/disabled. It also controls the margining of the system clock via DP control signals, and controls a mux used for enabling refresh during Battery Back Up (BBU) mode.

30.2.1.4 Memory Address Parity

The PAL at site 10C monitors and can disable address parity error reporting from the memory array boards.

30.2.1.5 MCBD Miscellaneous Control

The PAL at site 10H controls the direction and the driving of the BD transceivers. It also controls the data available signal (MDATAV+).

30.2.1.6 Error Encoding

The PAL at site 10L collects and reports the highest priority error detected on the CMI board.

30.2.1.7 Write Buffer Control PAL 1

The PAL at site 14H controls the write pulses to the DWB RAMs and the latch controls for writing the address, write pending, and valid bits into the Write Buffer.

30.2.1.8 DP Interface Miscellaneous Control

The PAL at 16G gates bit 15 used by the MC logic. It also controls which information is put onto the Decode Net address lines and the direction control for two way DP communication.

30.2.1.9 Write Buffer Control PAL 2

The PAL at site 16K controls the enable signals to the WB for the writing of the Valid bits and Write Pending bits.

30.2.1.10 WACK & STREAD

The PAL at site 16L controls the initiation of the WACK and STREAD chain of events. It also has the ABORT logic.

30.2.1.11 Bus MCBD Control

The PAL at site 18G controls whether the WBD RAMs or the MDAT latch information gets put onto MDAT.

30.2.1.12 CPU Request

The PAL at site 22F accepts the memory write, read, and cache miss signals, and controls the clocking of data from the E unit.

30.2.1.13 MODALS

The PAL at site 24D clocks Modal information when the Modals register is specified as the destination by microcode.

30.2.1.14 EOB

The PAL at site 24G controls the End of Beat (EOB) signal.

30.2.1.15 BPD Drive

The PAL at site 26E controls which set of BPD transceivers is enabled.

30.2.1.16 DP Interface Control

The PAL at site 26F controls DP interface handshaking and also collects decode net parity errors.

30.2.1.17 Decode Net Address

The PALs at sites 38D, 40D, and 40G control the decode net addressing.

30.2.1.18 MT Priority

The PAL at site 38N is the priority encoder for Memory Timer operations.

30.2.1.19 Clock Control

The PAL at site 42G enables TRCML, and INIT clocks. It also controls the writing of the CS address to the microsequencer.

30.2.1.20 CS & DNET Error Reporting

The PALs at sites 42K and 48N encode the CS and decode net errors.

30.2.1.21 Addressable Latch

The PAL at site 46D controls addressable latches FSPLUS, FSMINUS, ADL02, and I

30.2.1.22 EAF

The PAL at site 46G controls the resetting of FGEAF+/-.

30.2.1.23 CS Write/Read

The PAL at site 52F controls the writing and reading of CS memory based on signals received from the DP.

30.2.1.24 BPD Parity Check

The PAL at site 54E controls checking of BPD parity on output non-burst transfers at the trailing edge of BPCSTRB+.

30.2.2 PROMs

30.2.2.1 Memory Timer

The PROMs at sites 02F, 02L, 04G, 02K, 06L, and 44N control the Memory Timer sequencing and operations.

30.2.2.2 Destination

The PROMs at sites 52G and 54G decode the microcode DST field for all CMI board destinations.

30.2.2.3 IAC

The PROMs at sites 44F, 46E, 48F, and 50E decode the microcode IAC field.

30.2.2.4 EAF

The PROMs at sites 28D, 28E, and 30C decode the opcode for Effective Address Formation (EAF).

30.2.2.5 BOP

The PROMs at sites 52C and 54B control register file tracking.

30.3 Diagnostic Features

The CMI board contains a diagnostic register which is loaded from BD with microcode DST MDIAGR. Table 30-1 shows the layout of the register and briefly describes each bit's function.

TABLE 30-1. CMI Diagnostic Register Layout

Bit #	Name	Function
1	DGBFAD2+	MSB of write buffer pointer when DGBFSELAD+ is asserted.
2	DGBFAD3+	Middle bit of write buffer pointer when DGBFSELAD+ is asserted.
3	DGBFAD15+	LSB of write buffer pointer when DGBFSELAD+ is asserted.
4	DISERCOR+	Disables error correction.
5	DISERCK-	Disables error checking.
6	DMEMDIS-	Disables memory arrays.
7	DFBPARITY+	Forces MA parity bits to 1 regardless of address.
8	DGSELBFAD+	Forces write buffer pointer to diag. reg. bits{1:3}
9	SHFTCTR1	MSB of MA shift control field
10	SHFTCTR2	LSB of MA shift control field
11	DSMPLEM1-	Disables sampling of errors from memory array 1
12	DSMPLEM2-	Disables sampling of errors from memory array 2
13	DSMPLEM3-	Disables sampling of errors from memory array 3
14	DSMPLEM4-	Disables sampling of errors from memory array 4
15	DLCKB-	Locks check bits on memory arrays
16	DISRF+	Disables refresh
17	DISLW+	Disables write buffer last written algorithm
18		
19	DGENRFAD-	Puts refresh address on MA
20	DFRCBYPAS+	Forces write buffer contents onto BD during reads
21	DCLRWB+	Forces valid and write pending bits to 0
22	DFRCNOP-	Forces Memory Timer to do only NOP and Refresh
23	DIAGOPS+	Makes write buffer appear 1 location deep
24	DENALLOPS+	Forces write buffer to empty
25	DLOOP+	Forces write buffer contents onto MDAT during reads.
26	DMAONMD-	Forces MA on MD during reads
27	PDMTMODE+	Forces memory timer into diagnostic routine.
28		
29		
30		
31		
32		

30.4 Critical Paths

- The critical path through the CS involves receiving a jump condition from another unit, changing the BCYs, and accessing the CS RAMs. This is a two beat path from CS8+ to TRCML+.

- Conditional ReTurNs (CRTNs) are critical, but TXNX logic is used to make this path work because of the I unit intervention needed. This is a two beat path from CS8+ to TRCML+.

- The internal PUSEQ push/pop stack path is critical, due to the time required from TSTACK+ to TWRITE-.

- The path from the time the memory address is latched into the MC until the WB pointer is latched during a CPU memory access must be less than 2.5 beats.

- Setting MBSY- on the MC unit on a READ or WRITE and sending it to the IS unit is a one beat path.

- During an IO input burst mode operation the SWACK chain writes 64 bits into the WB RAMs in the same time a normal write does 32 bits. The timing is critical throughout this operation. Refer to Figure 27-3. After the first 32 bits are written at the end of D1WACK+ three critical things must happen in the next beat:

  1. Hold time must be met for the data just written.

  2. The least significant bit of the WB pointer must be changed.

  3. The BPD drivers which were driving the first 32 bits must be shut off, and the BPD transceivers with the second 32 bits must be turned on.

31. IS Board Discussion

The IS board implements the Instruction (I) and Storage management (S) units, as well as the PCU, cache, EAF logic, and branch cache. These functions are described in detail in the chapters cited below:

- I unit - chapter 19
- S unit - chapter 23
- PCU - chapter 17
- Cache - chapter 22
- EAF logic - chapter 20
- Branch cache - chapter 21

The IS board is a combination of the 9755 I and S boards. The exceptions to this are the decode net and EAF decode logic, which have moved to the CMI, and the trap addressing logic, which is on the E board.

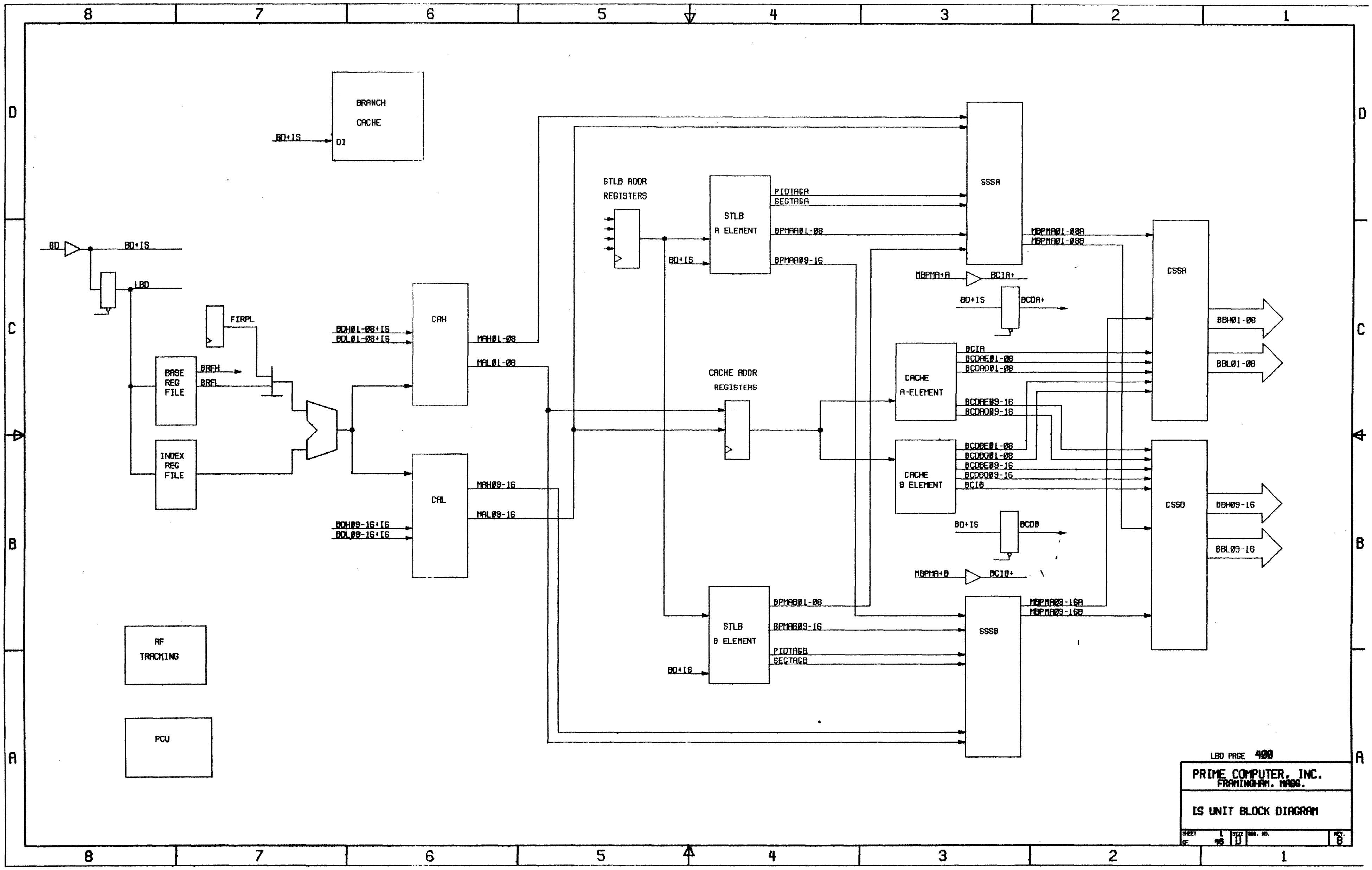


Figure 31-1 IS Board Block Diagram

31.1 4150 And 4050 Differences

There are no differences between the IS boards in the two machines.

31.2 PALs

31.2.1 PCU PALs

The PCU is partially implemented in the PALs at sites 06A, 06C, 02A, 04A, 20A, 30A, 28A, 22A, 22H, 58A, and 60M.

31.2.2 Parity Clocks

The PAL at site 56M generates clock enables for TCSSPE+, TSOFTPE+, TISPE+, TSSSPE+, TCSSPE+, and TFATALPE+ parity clocks.

31.2.3 EAS, EAD, and INSTAT Clocks

The PAL at site 58L generates clock enables for TEAS+, TEASH+, TEAD+, and TINSTAT+ clocks.

31.2.4 Branch Cache Control

The PAL at site 60A generates branch cache control information to be clocked by TINSTAT+.

31.2.5 Register Collisions

The PALs at sites 02C and 56N contains logic to detect register file collisions. The PAL at 02C generates a guess at the branch address used for stage 4-6 register collision detection. The PAL at 02C is used to generate IWAIT- for stage 5-9 register collisions.

31.2.6 BVMA Select Control

The PALs at sites 20J and 18J generate the BVMA selects for the PCADR VLSI.

31.2.7 Cache Write Control

The PALs at sites 52L and 54L generate the write enables for the cache RAMs.

31.2.8 RMA Clocks

The PAL at site 56L generates the enables for the TCADR+, TSADR+, and TERMA+ clocks.

31.2.9 Unaligned Read and Cache Data Clocks

The PAL at site 52M generates the enables for TRCDIE+, TRCDO+, and CS2OR7+ clocks used to clock cache data. The PAL also generates the enables for DFUNABLIP+ and FUNALBLIP+, which are used during unaligned reads.

31.2.10 Displacement, Opcode, and BB Latch Control

The PAL at site 44M generates the enables for TOPCD+, TDISP+, and TEBBLCH+, which are used to control the opcode, displacement, and BB latches. The first two of these latches are on the CMI board, while the last is on the E board.

31.2.11 BB and PMA selects

The PAL at site 56D generates the selects for the BB mux on the PCSS VLSI and PMA mux in the PSSS VLSI.

31.2.12 IRPL, IRPH, and RPST clocks

The PAL at site 04C generates the enables for TIRPL+, TIRPH+, and TRPST+ clocks.

31.2.13 STLB Write Pulses Enables

The PAL at site 48L generates the enables for the STLB write pulses.

31.2.14 Short/Long Instruction Control

The PAL at site 60E controls the correct alignment of opcodes on BB.

31.2.15 TRAPVLD Clock Enable

The PAL at site 58E is used to generate the enable for FTRAPVLD clock.

31.2.16 Displacement ALU and EAS mux control

The PAL at site 14J is used to control the displacement ALU inside the PCADR VLSI. The PAL also generates the selects for the EAS muxes in the PCADR VLSI.

31.2.17 Base Register ALU control

The PAL at site 42M is used to control the base register/index register ALU.

31.2.18 Control of Ring Bits

The PAL at site 40M is used to weaken the ring bit.

31.2.19 Base Register File Address Generation

The PAL at site 42P decodes the opcode bits and generates the base register file address.

31.2.20 Base Register Parity Checking

The PAL at site 34P is used to check parity on data coming out of the register file.

31.3 PROMs

31.3.1 Destination

The PROMs at sites 14D and 16D decode the microcode DST field for all IS board destinations.

31.3.2 IAC

The PROMS at sites 20D, 20E, and 22E decode the microcode IAC field for all IS board IACs.

31.3.3 Trap

The PROM at site 58D generates the IS board trap address for use by the microsequencer during trap sequences.

31.4 Critical Paths

One Beat Paths [62.5 nsec]

FTRAP+ holding off TCADR+ clock	62.5
Detection of 5-9 collision to generation of FIWAIT-	58.0
Unaligned read detection to ERMAL+1 to RMA	62.5
Memtrap to holding off cache miss on E-unit	60.5
Cache miss (GHOLD) to holding off E-unit clocks	62.5
FGEAF+ to inhibiting of CS1+ and CS3+ clocks	61.0
NXDONE- to TCADR+	60.0

Two Beat Paths [125 nsec]

Clocking base register address at CS4+ to clocking in the effective address at TRCML+.	114.0
Clocking opcode at CS2+ to generating F4UNAL+ control bit at CS4+	106.0

Half Beat Paths [31.25 nsec]

Memtrap holding off increment of IRPL	30.0
IRPL to PCADR registers and RMA	25.0

32. E Board Discussion

The E Board implements the Execution unit, the system timers, BD arbitration, and the I/O address interface. These topics are discussed in detail in the chapters cited below:

- E unit - chapter 25
- System timers - chapter 25
- BD arbitration - chapter 28
- I/O address interface - chapter 27

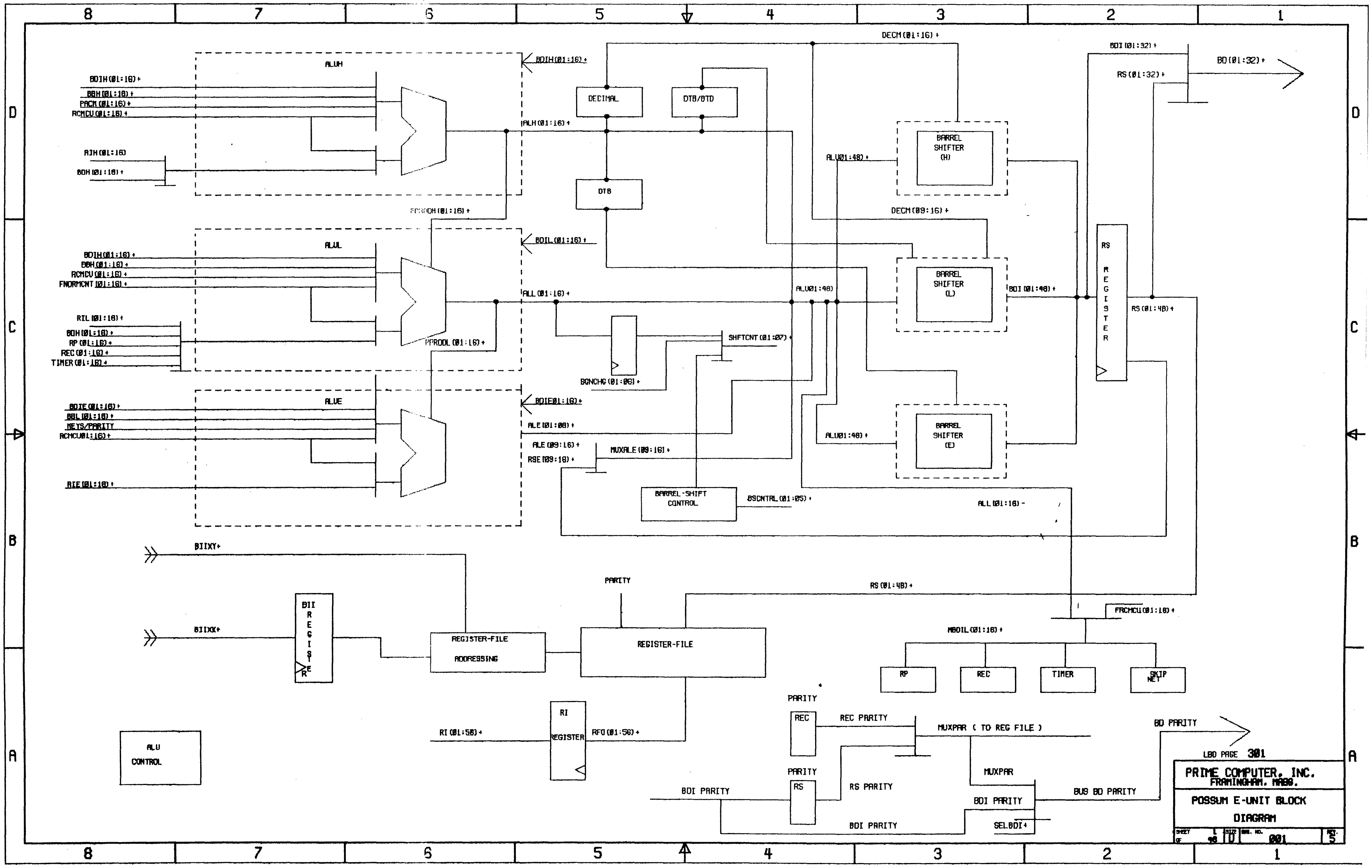


Figure 32-1 E Board Block Diagram

32.1 4150 And 4050 Differences

Two PALs are different between the 4150 and 4050 E boards.

1. The PAL at site 58E divides the nominal clock rate down into the 1 MHz signal used to clock the microsecond and PIC timers. Since the operating frequency of the two machines is different, this PAL needs a different program depending on the machine.

2. The PAL at site 32H provides the final error report to the microcode. One of the bits provided indicates whether the hardware is a 4150 or a 4050. The microcode uses this bit during its SYSCLR routine to verify that it is running on the correct processor.

32.2 Programmable Parts

32.2.1 PALs

32.2.1.1 Clock Control

The PALs at sites 28D, 38C, 34E, 38D, 40E, and 42E control the clock generation for the stage clocks.

32.2.1.2 RP16

The PAL at site 22A generates bit 16 of the RP register.

32.2.1.3 Register File RAM Control

The PAL at site 30M generates the register file RAM chip select.

32.2.1.4 BD Arbitration

The PAL at site 50C controls which unit of the CPU can drive BD.

32.2.1.5 Addressable Latches

The PAL at site 54D implements the following microcode addressable latches that are decoded from the microcode IAC field.

- ADL00
- ALD01
- ENTIMER

- INTEOI
- DISPIC
- ENMINK
- ENBPA
- FDMX

32.2.1.6 Multiply

The PAL at site 48J generates fix factor logic needed by the ALUs during multiply operations.

32.2.1.7 ZFF

The PAL at site 20G generates ALU information needed during decimal and ZMV instructions.

32.2.1.8 One MHz

The PAL at site 58E generates a 1 MHz clock needed for the PIC and microsecond timer logic. This clock is derived from TMCLK+.

32.2.1.9 I/O Requests

The PALs at sites 54B and 58A look at the Interrupt and DMx request from the I/O controllers to determine which controller to grant the bus to. it selects the highest priority request.

32.2.1.10 I/O Grant Enable

The PAL at site 46D generates the I/O grant enable for the I/O controllers.

32.2.1.11 I/O Grants

The PALs at sites 58C and 60C are priority generator PALs which generate the grant lines for the I/O controllers.

32.2.1.12 Parity Reporting

The PALs at sites 52C, 32K, 42J, 44J, and 32H generate parity and machine check information.

32.2.1.13 Fetch Cycle Traps

The PAL at site 48C generates either a FCYTRAP or FCYJC fetch cycle condition.

32.2.1.14 Traps

The PAL at site 30C generates trap information for the trap logic.

32.2.2 PROMs

32.2.2.1 ALU PROMs

The PROMs at sites 26L, 12D, 16F, 20L, 26K, 10D, 10N, 18N, and 26N control the mode of operation for the 56-bit ALU by decoding the microcode ALU field.

32.2.2.2 BB PROMs

The PROMs at sites 10F and 08F specify the B leg source input to the ALUs by decoding the microcode BB field.

32.2.2.3 BDL PROMs

The PROMs at sites 06G, 02J, 02M, and 02H control the mode of operation for the BDI barrel shifter by decoding the microcode BDL field.

32.2.2.4 DST PROMs

The PROMs at sites 30E, 30F, and 32D decode the microcode DST field for all E board destinations.

32.2.2.5 IAC PROMs

The PROMs at sites 32F, 36E, 48E, 34C, 56D, 58G, and 56E decode the microcode IAC field for all E board IACs.

32.2.2.6 Register File PROMs

The PROMs at sites 48B and 50B specify which register file location is to be used in the current operation.

32.2.2.7 Multiply PROM

The PROM at site 60J guesses the number of places to normalize during a floating point multiply operations.

32.2.2.8 PACK PROMs

The PROMs at sites 04K, 14P, 16E, and 18M do ASCII to packed BCD conversion.

32.2.2.9 Decimal PROMs

The PROMs at sites 44H, 42H, 28H, 26J, and 30H subtract 6666 hex from packed decimal data.

32.2.2.10 Decimal To Binary (DTB) PROMs

The PROMs at sites 34K and 36K do decimal to binary conversion.

32.2.2.11 ZFF and ZMV PROMs

The PROMs at sites 12F and 22M are used during decimal and character instructions.

32.2.2.12 Trap PROM

The PROM at site 06A calculates the E board trap address for use by the microsequencer during trap sequences.

32.3 Diagnostic Features

The E board contains a PAL at site 54A which looks at the I/O controller grant lines. The slot number of the controller being granted the bus is sent to the PDA for stack display.

32.4 Critical Paths

The critical paths have been identified as follows:

- Register File addressing - One beat path from TRCML to clocking of RI.
- Jump Conditions - One beat path from CS7+ to TJC+ flops.
- RS data - The output of BDIs clocked at CS9+ to S unit's CS10+ data latches.
- Condition codes - The path from the E unit JC mux selects at TRCML+ to the CS unit's TRCML+.
- Cache --> ALU --> RD - CS7+ clock to the CS8+ clock for RD inside the ALU.
- Register file read on register to register operations.

33. PDA Board Discussion

The 4150 Processor Diagnostic Aid (PDA) consists of the board, its hardware, resident Z80 code, and the host code that runs on a 50 series system. The PDA board plugs into the 4150 backplane in its dedicated slot, the IS debug slot, or the IS slot, and requires an RS-232 connection to the host system to allow communication between the host and the PDA. The common method of getting to the stand alone port is to enter the command MO PDA to the Diagnostic Processor.

The following 4150 PDA functions are discussed in detail in chapter 29:

- Z80 Microprocessor
- Stack
- Halts / Delays
- Delay Counter
- Sense Registers
- Forcing Microcode Address
- Reading and Writing Control Store

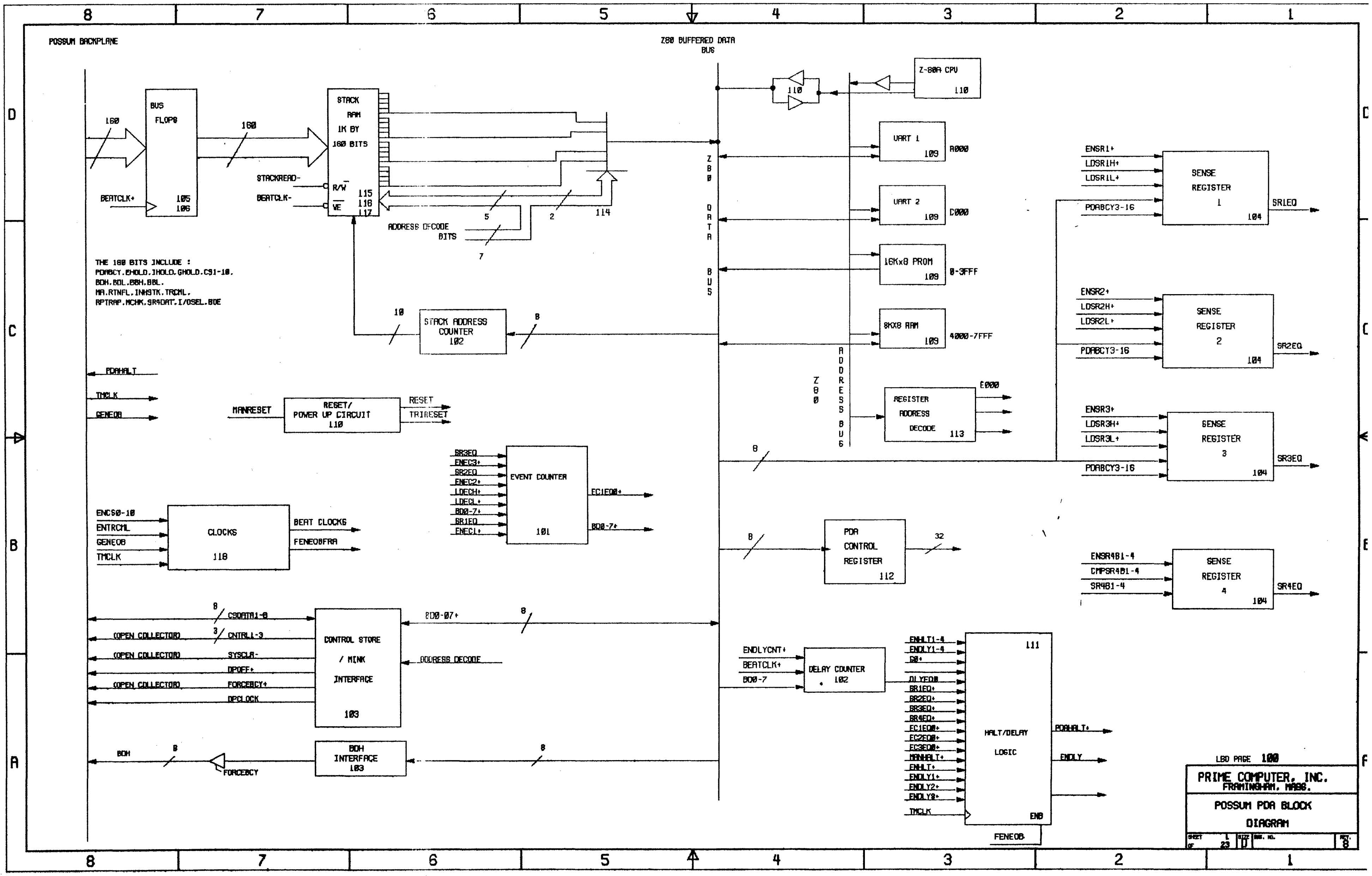


Figure 33-1 PDA Board Block Diagram

33.1 4150 And 4050 Differences

There are no differences between the PDA boards in the two machines.

33.2 Programmable Parts

33.2.1 PALS

33.2.1.1 Stack Inhibit PAL

The PAL at site 15B controls the inhibits to the stack. It uses inhibit conditions to generate a signal called INHSTK-.

33.2.1.2 Stack Read PAL

The PAL at site 31K generates the stack read signals under Z80 control. It uses the Z80 address bus as inputs and EBYTE01-04-, STKRD-, STKSELA+, STKSELB+ as outputs.

33.2.1.3 SR4 Comparator PAL

The PAL at site 31H is the sense register 4 PAL. It uses SR4 enables and data to generate SR4EQ1- and SR4EQ2-.

33.2.1.4 Halt and Delay PALs

The PAL at site 21D generates one of the halt and one of the delay signals. It uses inputs of sense register equal signals and enables and generates PSRHLT- and PSRDLY-.

The PAL at site 19D generates one of the halt and one of the delay signals. It uses inputs of sense register equal signals and enables and generates QSRHLT- and QSRDLY-.

33.2.1.5 Delay Inhibit PAL

The PAL at site 15C generates a delay inhibit signal. It uses various inhibit signals and generates INHDLY-.

33.2.1.6 Z80 Decode PALs

The PALS at sites: 29K, 25K, 23K, 21K, 17K, and 15K are Z80 decode PALs. They generate clocks to registers by decoding the Z80 address bus.

33.2.2 PROMs

33.2.2.1 Z80 Program PROMs

PROMs and RAMs (8K x 8) are interchangable in the following locations : 34K, 36K, 38K, 40K, 42K. The purpose of these locations is to provide executable Z80 code for the microprocessor to run. The typical configuration is with 2764 EPROM in locations 34K and 36K. Locations 38K and 42K typically will have pin compatible RAMs installed.

33.3 Diagnostic Features

The 4150 PDA has on board Z80 diagnostics that are executed from the Diagnostic Processor in the 'MO PDA' environment. Please refer to section 29.8.1.1 for additional information.

33.4 Critical Paths

The critical paths on the 4150 PDA are the stack write pulse, the setup and hold times on the stack, and the halt path from a sense register match to the issuance of PDAHALT-.

33.4.1 Stack Write Cycle

The stack write cycle has proven to be the most critical path in the PDA. The path involves sampling data from the backplane and storing it into registers while the address for the stack becomes stable. At this point, the write pulse, which is shaped by the use of delay lines, stores the data into the stack RAM.

```
                             |
           ______            |           ______
          |      |   Diag  --|           |      |
          |      |           |-----------| 9150 |
Data    --|      |-----------|           |   Ram|
          |      |           |           |      |
Beatclk+ -|>     |           |           |      |
           ______       _______          ______
          3-9 ns.      1-5 ns.         5ns setup before
                                      write pulse rising edge

                       +/-2ns.
                       ______       2ns Controlled
                      | 15ns |        etch run         AS1804
       AS574       ---| Delay|-----------------------  |\
       ______      |  | Line |   DLY44.5NS-          | | \
      |      |     |   ______                       -|N \    WP-A,B,C,D
GWP-  -      |-----                                  |A  )O-----
GWP+  -|     |-------------------------------------- |N /
      |      |   DLYDBEAT+       Rises Midbeat       |D/
TMCLK+ -|>   |                                       |/
       ______                                       1-4 ns.
      3-9 ns.
```

```
    3ns AS574 min                  9ns  AS574 max
    1ns AS1804 min                 2ns  etch max
  -----------------                4ns  AS1804 max
    4ns min TMCLK to WP falling  -----------------
                                   15ns max TMCLK to WP falling


    3ns AS574 min                  9ns  AS574 max
   15ns delay min                 17ns  delay max   Etch Delay not factored
    1ns AS1804 min                 4ns  AS1804 max  due to controlled lengths
  -----------------              -----------------
   19ns TMCLK to WP rising        30ns max TMCLK to WP rising
```

Using a 15ns minimum write pulse width in the above exercise forces us to carefully look at the path. Address and data setup and hold times are well within the specifications.

33.4.2 Halt Path

The timing from the sense register match to finally result in the PDAHALT- signal going active in 2 beats is a major path.

```
19ns   AS866 max delay d -> = (37B)
 2ns   setup on as574         (19C)
-----------------------------
21ns   BCY -> 19C (1 beat path)

 9ns   AS574 max clk -> out   (19C)
15ns   PAL delay              (21D)
 5.5ns AS08 delay             (05C)
 3ns   AS194 setup            (21A)
-----------------------------
32.5ns FSRn -> PDAHALT (1 beat path)
```

33.4.3 Trigger Path

This path involves getting a sense register match to result in a stack trigger in two beats.

```
19ns   AS866 max delay d -> = (37B)
 2ns   setup on as574         (19C)
-----------------------------
21ns   BCY -> 19C (1 beat path)

 9ns   AS574 max clk -> out   (19C)
15ns   PAL delay              (21D)
 4.5ns AS10 delay             (29D)
 4.5ns AS74 setup             (31E)
-----------------------------
33ns FSRn -> PDAHALT (1 beat path)
```