iAPX 86,88,186 MICROPROCESSORS
PART II
TABLE OF CONTENTS

1 ARCHITECTURE REVIEW
   - Description of iAPX 86/88
   - Architecture

2 INSTRUCTION SET REVIEW : CONSTRUCTS
   - Segment Definition
   - Data Definitions
   - Instruction Format
   - The ASSUME Statement
   - The Segment Override Prefix

3 INSTRUCTION SET REVIEW : INSTRUCTIONS BY CLASS

4 MODULAR PROGRAM DEVELOPMENT
   - Introduction to Linkage and Location
   - Procedures
   - Referencing External Program Labels and Data Items
   - Segment Combination
   - Segment Options

5 ASSEMBLER FEATURES (INCLUDES MACROS)
   - Assembler Directives
   - Assembler Built-Ins
   - Text Macros

6 COMPLEX DATA STRUCTURES
   - Structures
   - Records
   - Use of Addressing Modes

7 GROUPS
   - What They Are
   - How to Use Them
   - The GROUP Directive

8 LINK86 AND LOC86

9 LINKAGE WITH PLM86
   - PL/M Procedure Declarations
   - Parameter Passing
   - Compatible Data Types
   - Compilations Models
   - Conventions for Memory Allocation
   - Conventions for Procedure and Label Definitions
   - Conventions for Data Definitions
10 LINKAGE WITH OTHER HLLS
   - Pascal
   - Fortran
   - 'C'

11 INTRODUCTION TO 8087
   - Motivation for Using the 8087
   - Architecture
   - Hardware Interface
   - Software Interface

12 PROGRAMMING THE 8087
   - Instructions Format
   - Data Formats
   - Data Transfer Instructions
   - Arithmetic Instructions
   - Transcendental Instructions
   - Constant Instructions

13 MORE ON THE 8087
   - Status Word
   - Logical/Relational Instructions
   - Control Word
   - Initializing the 8087
   - Processor Control Instructions

14 8087 SUPPORT LIBRARIES
   - Emulator Libraries
   - Decimal Conversion Library
   - Common Elementary Library
   - Error Handler Library

15 INTRODUCTION TO THE 186
   - Description
   - Enhancements
   - New Instructions

16 PROGRAMMING THE 186 (CONTROL BLOCK, CHIP SELECTS, WAIT STATES)
   - Peripheral Control Block
   - Chip Select Lines
   - Wait State Logic

17 PROGRAMMING THE 186 (TIMERS)
   - Description
   - Features
   - Programming
18 PROGRAMMING THE 186 (DMA CONTROLLERS)
   - Motivation for Using Direct Memory Access
   - Description of Controller
   - Features
   - Programming

19 PROGRAMMING THE 186 (INTERRUPT CONTROLLER)
   - Description
   - Features
   - Programming

20 LIB86, CREF86
   - The Library Concept
   - Using LIB86
   - Module Cross Referencing Using CREF86

21 OVERVIEW OF THE 8089
   - Description
   - Architecture
   - Hardware Interface to 8086/88
   - Software Interface to 8086/88

APPENDICES

A  Lab Projects
B  Lab Solutions
C  80/88 Design Example
D  Daily Quizzes
E  Class Exercise Solutions
F  Introduction to PROM Programming
## Workshop Schedule

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Day One</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ARCHITECTURE REVIEW</td>
</tr>
<tr>
<td>2</td>
<td>INSTRUCTION SET REVIEW: CONSTRUCTS</td>
</tr>
<tr>
<td>3</td>
<td>INSTRUCTION SET REVIEW: INSTRUCTIONS BY CLASS</td>
</tr>
<tr>
<td>4</td>
<td>MODULAR PROGRAM DEVELOPMENT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Day Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>ASSEMBLER FEATURES</td>
</tr>
<tr>
<td>6</td>
<td>COMPLEX DATA STRUCTURES</td>
</tr>
<tr>
<td>7</td>
<td>GROUPS</td>
</tr>
<tr>
<td>8</td>
<td>LINK86 AND LOC86</td>
</tr>
<tr>
<td>9</td>
<td>LINKAGE WITH PLM86</td>
</tr>
<tr>
<td>10</td>
<td>LINKAGE WITH OTHER HLLS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Day Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>INTRODUCTION TO 8087</td>
</tr>
<tr>
<td>12</td>
<td>PROGRAMMING THE 8087</td>
</tr>
<tr>
<td>13</td>
<td>MORE ON THE 8087</td>
</tr>
<tr>
<td>14</td>
<td>8087 SUPPORT LIBRARIES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Day Four</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>INTRODUCTION TO THE 186</td>
</tr>
<tr>
<td>16</td>
<td>PROGRAMMING THE 186 (CONTROL BLOCK, CHIP SELECTS, WAIT STATES)</td>
</tr>
<tr>
<td>17</td>
<td>PROGRAMMING THE 186 (TIMERS)</td>
</tr>
<tr>
<td>18</td>
<td>PROGRAMMING THE 186 (DMA CONTROLLERS)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Day Five</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>PROGRAMMING THE 186 (INTERRUPT CONTROLLER)</td>
</tr>
<tr>
<td>20</td>
<td>LIB86, CREF86</td>
</tr>
<tr>
<td>21</td>
<td>OVERVIEW OF THE 8089</td>
</tr>
</tbody>
</table>
DAY 1 OBJECTIVES

BY THE TIME YOU FINISH TODAY YOU WILL:

- REVIEW BASIC 8086 ARCHITECTURE AND SEGMENTATION CONCEPTS
- REVIEW BASIC ASM86 CONCEPTS
- SEE THE ENTIRE INSTRUCTION SET OF THE 8086/88
- USE ADVANCED SEGMENT ATTRIBUTES (ALIGN-TYPE, COMBINE-TYPE, CLASSNAMES)
- USE MODULAR PROGRAMMING TECHNIQUES
CHAPTER 1
ARCHITECTURAL REVIEW

• DESCRIPTION OF THE IAPX 86,88
• REVIEW OF THE IAPX 86,88 ARCHITECTURE
INTERNAL ARCHITECTURE

**BIU** PERFORMS ALL BUS TRANSFERS
**EU** EXECUTES ALL INSTRUCTIONS
INSTRUCTION FETCHES OVERLAPPED WITH INSTRUCTION EXECUTION

**GENERAL REGISTERS**

<table>
<thead>
<tr>
<th>ACCUMULATOR</th>
<th>AX</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE</td>
<td>AH</td>
<td>AX</td>
</tr>
<tr>
<td>COUNTER</td>
<td>BL</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>BH</td>
<td></td>
</tr>
</tbody>
</table>

**DATA GROUP**

- AX (AL/AH)
- BX (BL/BH)
- CX (CL/CH)
- DX (DL/DH)

**POINTER AND INDEX GROUP**

- SP
- BP
- SI
- DI

1-1

1-2
### FLAG WORD

<table>
<thead>
<tr>
<th>OF</th>
<th>DF</th>
<th>IF</th>
<th>TF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARRY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARITY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUXILIARY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CARRY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZERO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERRUPT-ENABLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIRECTION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVERFLOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### STATUS FLAGS

### CONTROL FLAGS

### STATUS FLAG

---

### SEGMENT REGISTERS AND INSTRUCTION POINTER

<table>
<thead>
<tr>
<th>CODE SEGMENT</th>
<th>DATA SEGMENT</th>
<th>STACK SEGMENT</th>
<th>EXTRA SEGMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>DS</td>
<td>SS</td>
<td>ES</td>
</tr>
</tbody>
</table>

- MEMORY IS MAPPED INTO LOGICAL SEGMENTS OF UP TO 64K BYTES EACH
- THE SEGMENT REGISTERS POINT TO THE FOUR CURRENTLY ADDRESSABLE SEGMENTS
- THE IP AND CS REGISTER WORK TOGETHER FOR ADDRESSING INSTRUCTION MEMORY
• The 8086 can access any item that resides in a segment currently pointed to by one of the segment registers.

• To access items in other segments, the segment register is changed to point to the other segment.

• What is the maximum amount of memory that the 8086 can access at any given instant?

SEGMENT REGISTER CONTENTS

• The segment register always references a hexadecimal address boundary in memory.
APPLICATION OF THE SEGMENT REGISTERS

WHERE TO FIND MORE INFORMATION

8086, 8088 User's Manual (Programmer's Reference)
Chapter 3 - Architecture and Instructions
CHAPTER 2
INSTRUCTION SET REVIEW : CONSTRUCTS

- INSTRUCTION FORMAT
- DATA DEFINITION
- ASSUME STATEMENT
SEGMENT DEFINITION

<table>
<thead>
<tr>
<th>NAME</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>STACK</td>
<td>SEGMENT</td>
</tr>
<tr>
<td>;</td>
<td>STACK DEFINITIONS</td>
</tr>
<tr>
<td>STACK</td>
<td>ENDS</td>
</tr>
<tr>
<td>DATA</td>
<td>SEGMENT</td>
</tr>
<tr>
<td>;</td>
<td>DATA DEFINITIONS</td>
</tr>
<tr>
<td>DATA</td>
<td>ENDS</td>
</tr>
<tr>
<td>CODE</td>
<td>SEGMENT</td>
</tr>
<tr>
<td>ASSUME CS: CODE, DS: DATA</td>
<td></td>
</tr>
<tr>
<td>ASSUME SS: STACK</td>
<td></td>
</tr>
<tr>
<td>;/executable code</td>
<td></td>
</tr>
<tr>
<td>CODE</td>
<td>ENDS</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

SEGMENT REGISTER INITIALIZATION

```
CODE   SEGMENT

ASSUME CS: CODE, DS: DATA
ASSUME SS: STACK

MOV AX, DATA
MOV DS, AX
MOV AX, STACK
MOV SS, AX

...;

CODE ENDS
```
INITIALIZATION AND MODIFICATION OF THE CS REGISTER

RESET
FFFF0
JMP START

CS = FFFF
IP = 0

CODE1

START:

CALL PROC1

CODE2

PROC1 PROC FAR

PROC1 RET

ENDP

2-3

DATA DEFINITIONS

TYPES

DB - DEFINE BYTE
DW - DEFINE WORD
DD - DEFINE DOUBLE WORD
(8087 SHORT REAL, SHORT INTEGER)
DQ - DEFINE QUAD WORD
(8087 LONG REAL, LONG INTEGER)
DT - DEFINE TEN BYTE
(8087 PACKED DECIMAL, TEMPORARY REAL)

MORE ON 8087 DATA TYPES IN CHAPTER 10!

EXAMPLES

XYZ DB ? : UNINITIALIZED BYTE
ARRAY DB 100 DUP (?) : UNINITIALIZED ARRAY
ABC DB 3 : INITIALIZED BYTE
MSG1 DB 'WORKSHOPS' : INITIALIZED ARRAY
PI DQ 3.142 : INITIALIZED LONG REAL
ANDY DT 5 : INITIALIZED PACKED DECIMAL

2-4
ATTRIBUTES OF DATA ITEMS

- For every data definition, the assembler keeps track of three attributes.
  - Segment
  - Offset
  - Type

- The assembler uses these attributes to generate the correct instruction form.

Example:

```
DATA_1 SEGMENT
  XYZ DB ?,
  YYY DW ?
DATA_1 ENDS

CODE_1 SEGMENT
  ...
  MOV XYZ, 10H ; BYTE OPERATION
  MOVE 10H INTO MEMORY LOCATION XYZ
  MOV YYY, 20H ; WORD OPERATION
  ... ; MOVE 0020H INTO MEMORY LOCATION YYY
```

ASSEMBLY LANGUAGE INSTRUCTIONS

- Byte or word operations use the same mnemonic.
- In general, both operands must be the same type, byte or word.
- Most operations apply to any of the general registers and/or memory.
- Immediate data can also be specified in an instruction.

Examples:

```
MOV AL, BL ; BYTE OPERATION
MOV AX, BX ; WORD OPERATION
MOV BX, AL ; ILLEGAL
MOV AL, 20 ; BYTE OPERATION
MOV BX, 20 ; WORD OPERATION
MOV FRED, 10 ; WORD OPERATION (TYPE OF FRED IS WORD)
```
THE MEMORY OPERAND

• MANY INSTRUCTIONS CAN REFERENCE AN OPERAND IN MEMORY
  EG ADD FRED,1
• OFFSET OF OPERAND MAY BE SPECIFIED BY ......

OFFSET = [VARIABLE NAME] + [BX] + [SI] + [DI] + [DISPLACEMENT]

EG NOT TABLE [BX] - 6

MORE ON THE USE OF ADDRESSING MODES IN CHAPTER 6!

EXERCISE 2.1

IF AN INSTRUCTION HAS THE OPTION OF AN OPERAND IN MEMORY, ANY OF THE AVAILABLE ADDRESSING MODES MAY BE USED.

1. CAN THE XOR INSTRUCTION HAVE A MEMORY OPERAND?
2. IF SO, DOES THE ADDRESSING MODE AFFECT THE TIMING OF THE INSTRUCTION (CLUE - 'EA')?
3. WHY SHOULD THE ADDRESSING MODE AFFECT THE TIMING?
4. WHAT IS THE MINIMUM NUMBER OF CLOCKS THE FOLLOWING INSTRUCTION WOULD TAKE? ....
   XOR ARRAY BX ,AX

LOOK IN THE ASM86 LANGUAGE REFERENCE MANUAL.
EA = EFFECTIVE ADDRESS CALCULATION TIME.
ASSUME AND SEGMENT OVERRIDE PREFIXES

<table>
<thead>
<tr>
<th>NAME</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_1 SEGMENT</td>
<td></td>
</tr>
<tr>
<td>XYZ DW</td>
<td>?</td>
</tr>
<tr>
<td>BUFFER DB</td>
<td>100 DUP(?)</td>
</tr>
<tr>
<td>DATA_1 ENDS</td>
<td></td>
</tr>
<tr>
<td>DATA_2 SEGMENT</td>
<td></td>
</tr>
<tr>
<td>ABC DB</td>
<td>?</td>
</tr>
<tr>
<td>DATA_2 ENDS</td>
<td></td>
</tr>
</tbody>
</table>

CODE SEGMENT

FIVE

ASSUME DB 5

MOV AX, DATA_1
MOV DS, AX
MOV AX, DATA_2
MOV ES, AX

; DS USED. NO OVERRIDE NECESSARY.

MOV DS:XYZ, 0
MOV AL, DS:BUFFER [8]
ADD AL, ES:ABC
SUB AL, FIVE
XOR AX, [9]

; DS USED.

CODE ENDS
END
SEGMENT OVERRIDE

- SEGMENT OVERRIDE PREFIX

```
001 REG 110
```

One byte prefix to a memory reference instruction. The "REG" field identifies the segment register to be used in calculating the physical address.

---

USE OF SEGMENT OVERRIDE

<table>
<thead>
<tr>
<th>OFFSET REGISTER</th>
<th>DEFAULT</th>
<th>WITH OVERRIDE PREFIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP (Code Address)</td>
<td>CS</td>
<td>NEVER</td>
</tr>
<tr>
<td>SP (Stack Address)</td>
<td>SS</td>
<td>NEVER</td>
</tr>
<tr>
<td>BP (Stack Address or Stack Marker)</td>
<td>SS</td>
<td>DS, ES, OR CS</td>
</tr>
<tr>
<td>BX</td>
<td>DS</td>
<td>ES, SS, OR CS</td>
</tr>
<tr>
<td>SI or DI (not incl. strings)</td>
<td>DS</td>
<td>ES, SS, OR CS</td>
</tr>
<tr>
<td>SI (Implicit source addr for strings)</td>
<td>DS</td>
<td>ES, SS, OR CS</td>
</tr>
<tr>
<td>DI (Implicit dest addr for strings)</td>
<td>ES</td>
<td>NEVER</td>
</tr>
</tbody>
</table>

Note: If BP used in addressing mode (eg MOV AX, [BP][SI]), SS is used.
WHERE TO FIND MORE INFORMATION

ASM86 LANGUAGE REFERENCE MANUAL
CHAPTER 2 - SEGMENTATION
CHAPTER 3 - DEFINING AND INITIALIZING DATA

AN INTRODUCTION TO ASM86
CHAPTER 3
INSTRUCTION SET REVIEW

* INSTRUCTION SET BY CLASS
DATA TRANSFER INSTRUCTIONS

GENERAL PURPOSE

MOV  MOV BYTE OR WORD
PUSH PUSH WORD ONTO STACK
POP  POP WORD OFF STACK
XCHG EXCHANGE BYTE OR WORD
XLAT  TRANSLATE BYTE

INPUT / OUTPUT

IN  INPUT BYTE OR WORD
OUT OUTPUT BYTE OR WORD

TRANSLATE INSTRUCTION

• USEFUL FOR TABLE LOOKUP

AL ← [BX]+AL

<table>
<thead>
<tr>
<th>TABLE DB</th>
<th>10,20,14,17,23,41,60,72</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN AL,0</td>
<td></td>
</tr>
<tr>
<td>LEA BX,TABLE</td>
<td></td>
</tr>
<tr>
<td>XLATB</td>
<td></td>
</tr>
<tr>
<td>OUT 0,AL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ADDRESS OBJECT

LEA     LOAD EFFECTIVE ADDRESS
* EXAMPLE – LEA BX, TABLE [SI]

LES/LDS LOAD POINTER USING ES/DS
* USEFUL WITH STRING INSTRUCTIONS
* USEFUL FOR ACCESSING POINTER PARAMETERS PASSED ON STACK
* EXAMPLE – LOAD A POINTER PARAMETER FROM A STACK FRAME INTO ES:DI

<table>
<thead>
<tr>
<th>HI</th>
<th>← [BP] + 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO</td>
<td>← [BP] + 2</td>
</tr>
<tr>
<td>SP</td>
<td>← OLD BP</td>
</tr>
<tr>
<td></td>
<td>← [BP]</td>
</tr>
<tr>
<td>LES DI, [BP] + 4</td>
<td></td>
</tr>
</tbody>
</table>

STRING INSTRUCTIONS

- ONE BYTE INSTRUCTIONS WITH AUTO INCREMENT/DECREMENT OF INDEX REGISTERS
- OPERATE ON BYTES OR WORDS
- CAN USE OPERANDS FOR TYPING (BYTE/WORD) OR MNEMOMIC (EG MOVS B, MOVSW)

PRIMITIVES (OPERATE ON SINGLE BYTES/WORDS ONLY)

<table>
<thead>
<tr>
<th>MOVNS</th>
<th>MOVE BYTE OR WORD FROM SOURCE TO DESTINATION STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPS</td>
<td>COMPARE SOURCE TO DESTINATION STRING</td>
</tr>
<tr>
<td>SCAS</td>
<td>SCAN DESTINATION STRING FOR MATCH/NO MATCH WITH AL/AX</td>
</tr>
<tr>
<td>STOS</td>
<td>STORE AL/AX TO DESTINATION STRING</td>
</tr>
<tr>
<td>LDS</td>
<td>LOAD AL/AX FROM SOURCE STRING</td>
</tr>
</tbody>
</table>
STRING INSTRUCTIONS : REGISTER AND FLAG USE

- **SOURCE**
  - HI
  - DS:SI → LO
- **DESTINATION**
  - HI
  - DF = 1
  - CX
  - DF = 0
  - ES:DI → LO

- DIRECTION FLAG SPECIFIES AUTO INCREMENT/DECREMENT OF SI/DI
- CAN OVERRIDE USE OF DS TO ADDRESS SOURCE SEGMENT

REPEAT PREFIXES (REPEAT PRIMITIVE CX TIMES)
- REP
- REPE/REPZ (REPEAT WHILE EQUAL/ZERO (FLAG SET BY CMPS OR SCAS))
- REPNE/REPNZ (REPEAT WHILE NOT EQUAL/NOT ZERO)

EXAMPLE : STRING INSTRUCTIONS AND REGISTER USAGE

**NAME** CANTEEN_USAGE

**DATA_.1** SEGMENT
STUDENT DB 14 DUP (?)
DATA_.1 ENDS
DATA_.2 SEGMENT
CANT-EEN_SEATS DB 50 DUP (?)
DATA_.2 ENDS
CODE_.1 SEGMENT
ASSUME CS:CODE_.1, DS:DATA_.1, ES: DATA_.2
STUD_PTR DD STUDENT
CANT_PTR DD CANT-EEN_SEATS
MOV_IT PROC
LDS SI,STUD_PTR ; LOAD DS:SI
LES DI,CANT_PTR ; LOAD ES:DI
MOV CX,LENGTH STUDENT ; LOAD REPEAT COUNT
REP MOVS CANT-EEN_SEATS,STUDENT ; MOVE ALL STUDENTS INTO CANT-EEN
RET
MOV_IT ENDP
CODE_.1 ENDS
END
EXAMPLE : STRING COMPARE

NAME CHECK_PASSWORD
PANIC EQU 00H
EMPLOYEE EQU 0FFH

DATA_1 SEGMENT
REPLY DB 80 DUP (?)
DATA_1 ENDS

CODE_1 SEGMENT
ASSUME CS:CODE_1, DS:DATA_1, ES:CODE_1

CORRECT REPLY DD 'OPEN SESAME'
REPLY_PTR DD REPLY

CHECK PROC
LES DI,REPLY_PTR
LEA SI,CORRECT REPLY
MOV CX,LENGTH CORRECT REPLY
AGAIN: REPE CMPS REPLY,CORRECT REPLY
JNE SPY
JCXZ OK
JMP AGAIN

SPY: MOV AX,PANIC

OK: MOV AX,EMPLOYEE
RET

CHECK ENDP

CODE_1 ENDS
END
ITERATION CONTROLS

- UNCONDITIONAL LOOPS:
  LOOP LOOP CX TIMES

- LOOPS WITH CONDITIONAL TERMINATION:
  LOOPE/LOOPZ LOOP CX TIMES WHILE ZERO FLAG IS SET
  LOOPNE/LOOPNZ LOOP CX TIMES WHILE ZERO FLAG IS RESET

- SAFETY FEATURE FOR USE WITH LOOPS:
  JCXZ SPECIAL JUMP TO TEST COUNT IN CX.
  A ZERO COUNT WOULD CAUSE A 64K LOOP COUNT.

UNCONDITIONAL TRANSFERS

CALL CALL PROCEDURE
RET RETURN FROM PROCEDURE
JMP JUMP

INTERRUPTS

INT N SOFTWARE INTERRUPT TYPE N (N = 0 TO 255)
INTO INTERRUPT IF OVERFLOW FLAG SET
IRET RETURN FROM INTERRUPT
FLAG INSTRUCTIONS

FLAG TRANSFER

LAHF  LOAD AH FROM FLAGS (LS BYTE OF FLAGS REGISTER)
SAHF  STORE AH INTO FLAGS
PUSHF PUSH FLAGS ONTO STACK
POPF  POP STACK INTO FLAGS

FLAG OPERATIONS

STC  SET CARRY FLAG
CLC  CLEAR CARRY FLAG
CMC  COMPLEMENT CARRY FLAG
STD  SET DIRECTION FLAG
CLD  CLEAR DIRECTION FLAG
STI  SET INTERRUPT ENABLE FLAG
CLI  CLEAR INTERRUPT ENABLE FLAG

HOW DO I KNOW IF AN INSTRUCTION WILL AFFECT THE FLAGS?

• SEE ASM86 LANGUAGE REFERENCE MANUAL . . . .

EFFECT CODE  EFFECT
X  MODIFIED BY THE INSTRUCTION; RESULT DEPENDS ON OPERANDS
-  NOT MODIFIED
U  UNDEFINED AFTER THE INSTRUCTION
1  SET TO 1 BY THE INSTRUCTION
0  SET TO 0 BY THE INSTRUCTION
CONDITIONAL TRANSFERS

• THREE TYPES OF CONDITIONAL JUMP ...
  - FOR UNSIGNED NUMBERS (USE 'ABOVE' AND 'BELOW')
  - FOR SIGNED NUMBERS (USE 'GREATER' AND 'LESS')
  - FOR EITHER (THEY EXAMINE INDIVIDUAL FLAGS)

• OPTIONAL MNEMONICS FOR SOME CONDITIONAL JUMPS

• ALL CONDITIONAL JUMPS ARE SHORT JUMPS (THEY JUMP +127/-128 BYTES)

UNSIGNED:

| JA / JNBE | JUMP IF ABOVE / NOT BELOW OR EQUAL |
| JAE / JNB  | JUMP IF ABOVE OR EQUAL / NOT BELOW |
| JBE / JNBE | JUMP IF BELOW OR EQUAL / NOT ABOVE NOR EQUAL |
| JB / JNA   | JUMP IF BELOW / NOT ABOVE |

SIGNED:

| JG / JNLE | JUMP IF GREATER / NOT LESS NOR EQUAL |
| JGE / JNL  | JUMP IF GREATER OR EQUAL / NOT LESS |
| JL / JNGE  | JUMP IF LESS / NOT GREATER OR EQUAL |

FLAGS:

| J(N)C   | JUMP IF CARRY FLAG (NOT) SET |
| J(N)Z / J(N)Z | JUMP IF ZERO FLAG (NOT) SET |
| J(N)O   | JUMP IF OVERFLOW FLAG (NOT) SET |
| J(N)S   | JUMP IF SIGN FLAG (NOT) SET |
| JPE / JPO | JUMP IF PARITY EVEN/ODD |

3-13

3-14
BIT MANIPULATION INSTRUCTIONS

LOGICALS

NOT
AND, OR, XOR
TEST

COMPLEMENT ALL BITS
LOGICAL AND, OR, EXCLUSIVE OR
NON-DESTRUCTIVE AND FOR TESTING BITS

SHIFTS (ONE PLACE OR CL TIMES)

SHL/SAL
SHR
SAR

SHIFT LEFT/ARITHMETIC LEFT
SHIFT RIGHT
SHIFT ARITHMETIC RIGHT

ROTATES (ONE PLACE OR CL TIMES)

ROL
ROR
RCL
RCR

ROTATE LEFT
ROTATE RIGHT
ROTATE LEFT THROUGH CARRY
ROTATE RIGHT THROUGH CARRY

ARITHMETIC INSTRUCTIONS

ADDITION

ADD
ADC
INC

ADD
ADD WITH CARRY
INCREMENT

SUBTRACTION

SUB
SBB
DEC
CMP
NEG

SUBTRACT
SUBTRACT WITH BORROW
DECREMENT
COMPARE (NON-DESTRUCTIVE SUBTRACT)
NEGATE
ARITHMETIC INSTRUCTION (CONT.)

MULTIPLICATION (8×8 = 16 BITS OR 16×16 = 32 BITS)

MUL UNSIGNED MULTIPLY
IMUL INTEGER MULTIPLY

DIVISION (16 / 8 = 8 BITS OR 32 / 16 = 16 BITS)

DIV UNSIGNED DIVIDE
IDIV INTEGER DIVIDE

QUESTION: WHAT HAPPENS IF THE RESULT OF A DIVISION WILL NOT FIT INTO THE DESTINATION REGISTER (PROBABLY BECAUSE OF A DIVIDE BY ZERO)?

THE ADJUST INSTRUCTIONS

DECIMAL ADJUSTMENTS:

DAA: DECIMAL ADJUST FOR ADD - ADD TWO BCD NUMBERS, ADJUST RESULT

EXAMPLE:

MOV AL,26H ; BCD 26
ADD AL,27H ; ADD BCD 27, RESULT IS 4DH
DAA ; RESULT IS ADJUSTED TO BCD 53

* CARRY FLAG WILL INDICATE 100 (MAXIMUM SUM WOULD 99 + 99 = 198)
* ONLY WORKS FOLLOWING ADDITION OF TWO PACKED BCD DIGITS
THE ADJUST INSTRUCTIONS
(CONT.)

DAS : DECIMAL ADJUST FOR SUBTRACT
* CARRY FLAG INDICATES 100 'BORROWED'
* ONLY WORKS FOLLOWING SUBTRACTION OF TWO PACKED BCD DIGITS

EXAMPLE :
MOV AL,6 ; AL = BCD 6
SUB AL,27H ; SUBTRACT BCD 27, RESULT IS DFH
DAS ; RESULT ADJUSTED TO 79, CARRY FLAG SET

* RESULT WAS 6 - 27 = -21. 100 WAS BORROWED FROM NEXT MOST SIGNIFICANT BYTE OF THE OPERAND (WHEN SUBTRACTING STRINGS OF BCD NUMBERS). CARRY INDICATES 100 WAS BORROWED, -21 + 100 = 79.

** CAN ADD/SUBTRACT BCD STRINGS USING SEVERAL ADD/SUBTRACT AND DAA/DAS INSTRUCTIONS CONNECTED BY CARRY FLAG.

THE ADJUST INSTRUCTIONS
(CONT.)

ASCII ADJUSTMENTS :

AAA : ASCII ADJUST FOR ADD - ADJUST RESULT OF ADDING TWO UNPACKED DIGITS
AAS : ASCII ADJUST FOR SUBTRACT - AAA FOR SUBTRACTION
AAM : ASCII ADJUST FOR MULTIPLY - HAVING MULTIPLIED TWO DIGITS (RESULT IN HEX), Splits PRODUCT INTO TWO DECIMAL DIGITS IN AH, AL
AAD : ASCII ADJUST FOR DIVIDE - CONVERTS TWO UNPACKED BCD DIGITS INTO THEIR 8-BIT BINARY EQUIVALENT READY FOR A DIVIDE OPERATION

* ASCII OFFSET OF 30H IS LOST. YOU OR IT BACK IN TO AN ADJUSTED RESULT

* SEE ASM86 LANGUAGE REFERENCE MANUAL FOR DETAILS OF THESE INSTRUCTIONS
EXTERNAL SYNCHRONIZATION

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLT</td>
<td>HALT UNTIL INTERRUP OR RESET</td>
</tr>
<tr>
<td>WAIT</td>
<td>WAIT FOR TEST PIN ACTIVE</td>
</tr>
<tr>
<td>ESC</td>
<td>ESCAPE TO EXTERNAL PROCESSOR (EG 8087)</td>
</tr>
<tr>
<td>LOCK</td>
<td>LOCK BUS FOR DURATION OF NEXT INSTRUCTION</td>
</tr>
</tbody>
</table>

NO OPERATION

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>NO OPERATION (XCHG AX,AX)</td>
</tr>
</tbody>
</table>

EXERCISE 3.1

1. WHAT IS THE PURPOSE OF THE ARGUMENT TO XLAT IN THE FOLLOWING CODE:

```asm
CODE SEGMENT
ASSUME CS:CODE
TABLE DB 1,3,5,7 ...
.....
LEA BX,TABLE
XLAT TABLE
```

2. LIST THE TYPES OF UNCONDITIONAL CALLS (eg indirect FAR) AND HOW ASM86 RECOGNIZES WHICH TYPE IT IS TO ENCODE.

3. YOU ARE WRITING A DEBUGGER ROUTINE WHICH IS TO SINGLE STEP THROUGH YOUR APPLICATION CODE. ASSUME THAT YOU ENTERED THE DEBUGGER WITH A STACK FRAME AS SHOWN. USE POP, PUSH, IRET TO SINGLE STEP YOUR CODE FROM THE ADDRESS CURRENTLY IN ES:DI. DO THIS BY SETTING THE TRAP FLAG ON 'RETURN' TO THE APPLICATION CODE

```
FLAGS
RET BASE
RET OFFSET
```

LEFT ON STACK BY APPLICATION CODE WHEN DEBUGGER WAS ENTERED
WHERE TO FIND MORE INFORMATION

IAPX 86/88, 186/188 USER'S MANUAL (PROGRAMMER'S REFERENCE)
CHAPTER 3 - ARCHITECTURE AND INSTRUCTIONS

AN INTRODUCTION TO ASM86

ASM86 LANGUAGE REFERENCE MANUAL

RELATED TOPICS ...

BIT CODINGS FOR INSTRUCTIONS ARE NOT COVERED IN THIS COURSE. FOR INFORMATION, SEE IAPX 86/88, 186/188 USER'S GUIDE. YOU WILL FIND THE ASM86 MACRO ASSEMBLER POCKET REFERENCE USEFUL (SEE FRONT OF IT FOR BIT ENCODING INFORMATION).

3-23
CHAPTER 4
MODULAR PROGRAM DEVELOPMENT

• INTRODUCTION TO LINKAGE AND LOCATION
• PROCEDURES
• LINKAGE DIRECTIVES
• REFERENCING EXTERNAL PROGRAM LABELS AND DATA ITEMS
• SEGMENT COMBINATION
• SEGMENT OPTIONS
SINGLE MODULE PROGRAM DEVELOPMENT

MODULAR PROGRAM DEVELOPMENT

4-1

4-2
LINKAGE

MOD1

SEGA
SEGB

MOD2

SEGA
SEGB
SEGC
SEGD

LINKED MODULE

MOD3

SEGA
SEGB
SEGC
SEGD

LOCATION

LINKED MODULE

SEGA
SEGB
SEGC
SEGD

- LOCATION IS PERFORMED ON A SEGMENT BASIS.
PROCEDURES

- MODULES OF A PROGRAM USUALLY LINKED BY PROCEDURE CALLS
- MULTI MODULE PROGRAM WILL USUALLY HAVE A 'MAIN MODULE'
- MAIN MODULE CONTAINS PROGRAM START ADDRESS
- OTHER MODULES CONTAIN PROCEDURES AND DATA DEFINITIONS
- IN ASM86 'END START' DEFINES MAIN MODULE (ALL OTHERS JUST HAVE END)
- LINKER WILL TRAP 'MORE THAN 1 MAIN MODULE'

PROCEDURE DEFINITION

```
CODE_1 SEGMENT
ASSUME CS:CODE_1

WALLY PROC FAR ; default is NEAR
               ; insert useful
               ; code here

RET

WALLY ENDP

CODE_1 ENDS
```
EXERCISE 4.1 NEAR AND FAR PROCEDURES

TRUE OR FALSE?

* GIVING A PROCEDURE THE FAR ATTRIBUTE DOES THE FOLLOWING THINGS...
1. ENCODES A FAR RET INSTRUCTION
2. TAGS THE PROCEDURE AS FAR
3. BECAUSE OF 2, ALL CALLS TO THIS PROCEDURE WILL TAKE 3 BYTES

* CALLING A FAR PROCEDURE FROM THE SEGMENT IN WHICH IT WAS DEFINED PRODUCES A NEAR CALL

* IF IN IGNORANCE I NEAR CALL A PROCEDURE WHICH IS DEFINED IN ANOTHER MODULE AS FAR THE RET INSTRUCTION PRINTS AN ERROR MESSAGE

"HELP - I CAN'T FIND A SEGMENT TO RETURN TO!"

... AND DON'T FORGET THE STACK!

STACK SEGMENT
DW 100 DUP (?)
T_O_S LABEL WORD
STACK ENDS
MAIN SEGMENT
ASSUME CS:MAIN, SS:STACK
START: MOV AX,STACK
       MOV SS,AX
       LEA SP,T_O_S
       CALL WALLY
MAIN ENDS
END START
## INTER-MODULE REFERENCES

By using **PUBLIC** and **EXTRN** declaratives with the two modules, **LINK86** can resolve external references.

<table>
<thead>
<tr>
<th>NAME</th>
<th>MODA</th>
<th>NAME</th>
<th>MODB</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTRN</td>
<td>PROCA:FAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEGMENT</td>
<td>CS:SEG</td>
<td>SEGMENT</td>
<td>CS:SEGB</td>
</tr>
<tr>
<td>ASSUME</td>
<td>CS:SEG</td>
<td>ASSUME</td>
<td>CS:SEGB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>PROCA</td>
<td>PROC</td>
<td>FAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEGA</td>
<td>ENDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

## PUBLIC AND EXTERNAL DECLARATIVES

- **PUBLIC** makes a name available to the linker.
- **EXTRN** tells **ASM86** to let the linker resolve the symbol.

**EXAMPLES:**

<table>
<thead>
<tr>
<th>PUBLIC</th>
<th>EXTRN</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>XYZ, WP, ERS</td>
<td>F00: BYTE</td>
<td>: VARIABLES AND PROCEDURES DEFINED ELSEWHERE IN THIS MODULE</td>
</tr>
</tbody>
</table>

**ATTRIBUTES OF AN EXTERNAL REFERENCE:**

- Near, Far: (EXTERNAL PROCEDURE)
- Byte, Word, DWORD: (EXTERNAL VARIABLE)
- Abs: (EXTERNAL CONSTANT)
REFERENCING EXTERNAL PROCEDURES AND PROGRAMS LABELS

WHY DID PROCA HAVE TO BE A FAR PROCEDURE?

COMBINING SEGMENTS
SEGMENT COMBINATION USING PUBLIC SEGMENTS

<table>
<thead>
<tr>
<th>NAME</th>
<th>MOD1</th>
<th>NAME</th>
<th>MOD2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC</td>
<td>XYZ</td>
<td>PUBLIC</td>
<td>PROCA</td>
</tr>
<tr>
<td>EXTRN</td>
<td>PROCA:NEAR</td>
<td>EXTRN</td>
<td>XYZ:NEAR</td>
</tr>
</tbody>
</table>

CODE SEGMENT PUBLIC
ASSUME CS:CODE

CALL PROCA

XYZ:

CODE ENDS
END

REFERENCE EXTERNAL DATA ITEMS

<table>
<thead>
<tr>
<th>DATA</th>
<th>PUBLIC</th>
<th>BUFFER</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODULE A:</td>
<td>SEGMENT</td>
<td>BUFFER</td>
</tr>
<tr>
<td>BUFFER DB</td>
<td>100 DUP (?)</td>
<td></td>
</tr>
<tr>
<td>DATA ENDS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CODE EXTRN BUFFER:BYTE

<table>
<thead>
<tr>
<th>CODE</th>
<th>MODULE B:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSUME CS:CODE,DS: ______</td>
<td></td>
</tr>
<tr>
<td>MOV AX, ______</td>
<td></td>
</tr>
<tr>
<td>MOV DS, AX</td>
<td></td>
</tr>
<tr>
<td>MOV BUFFER, AL</td>
<td></td>
</tr>
<tr>
<td>CODE ENDS</td>
<td>END</td>
</tr>
</tbody>
</table>

* HOW WOULD WE REFERENCE MULTIPLE EXTERNAL DATA ITEMS
REFERENCING MULTIPLE EXTERNAL DATA ITEMS

NAME MOD1

PUBLIC XYZ,ABC
DATA SEGMENT PUBLIC
XYZ DB ?
ABC DW ?
DATA ENDS
END

NAME MOD2

DATA SEGMENT PUBLIC ; DUMMY SEGMENT
EXTERN XYZ:BYTE, ABC:WORD
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE, DS:____
MOV AX,____
MOV DS, AX
MOV AL, XYZ
MOV AH, 0
MOV ABC, AX
CODE ENDS
END

ALIGNMENT TYPES

NAME ALIGNMENT EXAMPLE

DATA SEGMENT WORD PUBLIC
XYZ DB ?
EVEN
ARRAY DW 100 DUP(?)
DATA ENDS
CODE SEGMENT BYTE PUBLIC
ASSUME CS:CODE, DS:DATA
TABLE DW 50, 30, 25, 62, 75
START: MOV AX, DATA
MOV DS, AX
; ;
CODE ENDS
END

• EVEN DIRECTIVE ENSURES EVEN BOUNDARY ALIGNMENT WITHIN A SEGMENT.
• EVEN DIRECTIVE CAUSES ERROR IN BYTE ALIGNED SEGMENTS.
COMBINE TYPE : STACK

NAME  MAIN  NAME  PROC_MOD
STACK_SEG  SEGMENT  STACK  STACK_SEG  SEGMENT  STACK
STACK_SEG  DW  20 DUP (?)  STACK_SEG  DW  14 DUP (?)
STACK_TOP  LABEL  WORD  STACK_SEG  ENDS
STACK_SEG  ENDS

CODE  SEGMENT  ASSUME  CS:CODE,SS:STACK
MOV  AX,STACK
MOV  DS,AX
LEA  SP,STACK_TOP
CODE  ENDS
CODE  END

STACK_TOP IS ADJUSTED FOR INITIALIZATION PURPOSES

STACK_SEG SEGMENT 'RAM'
STACK_SEG SEGMENT STACK 'RAM'
STACK_SEG ENDS
STACK_SEG ENDS

CLASS NAMES

• PERMITS CLASSIFICATION OF SEGMENTS UNDER A COMMON NAME THAT CAN BE USED AT LOCATE TIME.

• EXAMPLE

NAME  MODA  NAME  MODB
DATA_1  SEGMENT 'RAM'  DATA_2  SEGMENT 'RAM'
DATA_1  ENDS  DATA_2  ENDS
STACK_SEG SEGMENT STACK 'RAM'
STACK_SEG SEGMENT STACK 'RAM'
STACK_SEG ENDS
STACK_SEG ENDS
CODE_1  SEGMENT 'ROM'  CODE_2  SEGMENT 'ROM'
ASSUME  CS:CODE_1,DS:DATA_1  ASSUME  CS:CODE_2,DS:DATA_2
CODE_1  ENDS  CODE_2  ENDS
CODE_1  ENDS  CODE_2  ENDS
LOCATION BY CLASS NAME

'ROM'

CODE_1
CODE_2

'RAM'

STACK SEG
DATA_2
DATA 1

WHERE TO FIND MORE INFORMATION

ASM86 LANGUAGE REFERENCE MANUAL
CHAPTER 5 - PROGRAM LINKAGE DIRECTIVES

AN INTRODUCTION TO ASM86
CHAPTER 4 - MODULAR PROGRAMMING
DAY 2 OBJECTIVES

BY THE TIME YOU FINISH TODAY YOU WILL:

• USE ASSEMBLER BUILT-IN FUNCTIONS
• USE ASSEMBLER DIRECTIVES
• WRITE ASM86 TEXT MACROS
• DEFINE COMPLEX DATA STRUCTURES IN ASM86
• USE APPROPRIATE ADDRESSING MODES FOR ACCESSING COMPLEX DATA STRUCTURES
• DEFINE AND USE SEGMENT GROUPS
• LEARN HOW TO USE LINK86 AND LOC86
• LINK ASSEMBLER PROGRAMS TO PL/M-86 PROGRAMS
• SEE HOW LINKING TO OTHER HIGH LEVEL LANGUAGES COMPARES TO LINKING WITH PL/M-86
CHAPTER 5
ASSEMBLER FEATURES

• ASSEMBLER DIRECTIVES
• ASSEMBLER BUILT-INS
• TEXT MACROS
THE EVEN DIRECTIVE

DATA SEGMENT WORD
BITE DB ?
EVEN
BERNTH DW 1234 ; GUARANTEED WORD AlIGNED
DATA ENDS
• ENSURES WORD ALIGNMENT
• MIGHT WASTE A BYTE OF STORAGE
• DON'T MAKE DATA SEGMENTS BYTE-ALIGNED !!!!

5-1

THE LABEL DIRECTIVE

• ASSOCIATES A USER DEFINED SYMBOL NAME WITH THE CURRENT ASSEMBLER LOCATION COUNTER
• USEFUL FOR CODE AND DATA LABELS

• EXAMPLE

DATA SEGMENT LABEL BYTE
LO_BYTE WORD_VAR DW ?
DATA ENDS
STACK SEGMENT DW 20 DUP(?)
STACK_TOP LABEL WORD
STACK ENDS
CODE SEGMENT ASSUME CS:CODE, SS:STACK, DS:DATA
ASSUME AX,STACK ; INITIALIZE STACK
MOV AX,DATA
MOV DS,AX
MOV AL, LO_BYTE
;
;
;
COD_ ENDS

5-2
ASSEMBLER BUILT-INS

- ASSEMBLER HAS BUILT-IN OPERATORS TO AID IN PROGRAMMING

<table>
<thead>
<tr>
<th>TYPE</th>
<th>RETURNS TYPE OF DATA DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>1 BYTE</td>
</tr>
<tr>
<td>DW</td>
<td>2 BYTES</td>
</tr>
<tr>
<td>DD</td>
<td>4 BYTES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LENGTH</th>
<th>RETURNS NUMBER OF UNITS</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>SIZE</th>
<th>RETURNS NUMBER OF BYTES</th>
</tr>
</thead>
</table>

- EXAMPLE

```
ARRAY DW 100 DUP(?)
MOV BX, 0
MOV CX, LENGTH ARRAY
NEXT:
ADD ARRAY[BX], 60
ADD BX, TYPE ARRAY
LOOP NEXT
MOV AX, SIZE ARRAY
CALL SAVE_ON_DISK
```

5-3

EQU STATEMENT

- THE EQU STATEMENT PROVIDES MORE MEANINGFUL NAMES FOR EXPRESSIONS

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>THREE</th>
<th>EQU</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS EXPRESSION</td>
<td>XYZ</td>
<td>EQU</td>
<td>ALPHA [81]</td>
</tr>
<tr>
<td>REGISTER</td>
<td>COUNT</td>
<td>EQU</td>
<td>CX</td>
</tr>
</tbody>
</table>

- EXAMPLE

```
MOV AL, THREE ;SAME AS AL,3
MOV AX, XYZ   ;SAME AS AX, ALPHA [91]
MOV COUNT,LENGTH ARRAY ;SAME AS CX,LENGTH ARRAY
```

5-4
...SO!!!

NOW YOU THINK

YOU KNOW EVERYTHING

ABOUT THE ASSEMBLER!!!

NOW FOR A COMPLETELY NEW LANGUAGE....

M.P.L.

(MACRO PROCESSING LANGUAGE)

M. P. L.

• ALL MACRO PROCESSING PERFORMED BEFORE ASSEMBLY COMMENCES
• TEXT SUBSTITUTION MACROS
• MACRO PARAMETERS
• INTERACTION WITH CONSOLE DURING ASSEMBLY
• CONDITIONAL ASSEMBLY
• NESTED MACROS
• EVALUATION OF NUMERIC CONSTANTS
• TEXT STRING OPERATIONS
• AND MORE!!!
ALL MACRO PROCESSING PERFORMED
BEFORE ASSEMBLY COMMENCES

GENERAL FORMAT OF MACRO DEFINITION IS . . .

% DEFINE ( MACRO-NAME [(PARAMETER-LIST)] ) (MACRO-BODY)

MACRO PROCESSING AND ASSEMBLY

SOURCE FILE → MACRO PROCESSING → ASSEMBLY

MAY CONTAIN MACRO DEFINITIONS AND CALLS

LIST FILE
OBJECT FILE
ERROR PRINT FILE
TEXT SUBSTITUTION MACROS


% *DEFINE(WRITE)(
    PUSH    CS
    PUSH    AX
    CALL    PRINT
)

INVOCATION:
    LEA AX,MESSAGE_1 ; AX = OFFSET OF MESSAGE
%WRITE

MACRO PARAMETERS

THIS MACRO WILL PASS A POINTER TO A PRINT PROCEDURE. THE POINTER IS COMPOSED OF AN OFFSET (NOW PASSED AS A PARAMETER) AND THE CURRENT VALUE OF THE CODE SEGMENT REGISTER (THE MESSAGES ARE CONTAINED IN THE CODE SEGMENT).

% *DEFINE(WRITE(STRING_ADDRESS))( 
    LEA AX,%STRING_ADDRESS
    PUSH CS
    PUSH AX
    CALL PRINT
)

INVOCATION:
    %WRITE(MESSAGE_1) ; PRINT MESSAGE 1
* CAN HAVE MANY PARAMETERS IF REQUIRED
INTERACTION WITH CONSOLE DURING ASSEMBLY

THIS MACRO WILL ASK THE USER WHAT BAUD RATE IS REQUIRED FOR THIS SYSTEM

```
%*DEFINE (WHAT_BAUD) (
    %OUT (ENTER REQUIRED BAUD RATE . . )
    BAUD_RATE DW %IN
)

INVOCATION :
DATA_1 SEGMENT
%WHAT_BAUD
DATA_1 ENDS
-RUN ASM86 :F1:TEST.ASM
SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.0
ENTER REQUIRED BAUD RATE .... 9600 <CR>
ASSEMBLY COMPLETE, NO ERRORS
-
YOUR LISTING WILL NOW SHOW BAUD_RATE DW 9600
```

EVALUATION OF NUMERIC CONSTANTS

NOW USE THE EVALUATE COMMAND TO MAKE THIS A LITTLE MORE CLEVER. THE MACRO WILL NOW CALCULATE THE CORRECT NUMBER FOR YOUR COUNTER WHICH WILL PRODUCE THE REQUIRED BAUD RATE

```
%*DEFINE (WHAT_BAUD) (
    %OUT (ENTER REQUIRED BAUD RATE . . )

%SET(BAUD_RATE, % IN)
COUNT_VAL EQU %EVAL( (%BAUD_RATE / 27 ) - 13)
)
```
CONDITIONALS IN MACROS

AND NOW LET'S SEE IF WE CAN MAKE THIS THING FOOLPROOF!

%*DEFINE (WHAT_BAUD) (
%OUT (ENTER REQUIRED BAUD RATE ...)
%SET (BAUD_RATE, %IN)
%IF (%BAUD_RATE LT 100) THEN
  (%OUT (HOW SLOW DID YOU SAY ?????))
  FI
%IF (%BAUD_RATE GT 9600) THEN
  (%OUT (YOU MUST BE JOKING !!!))
  FI
COUNT_VAL EQU %EVAL( (%BAUD_RATE / 27 ) - 13 )
)

5-13

INSTEAD OF COMMENTING ON A BAUD RATE,
LET'S DISALLOW IT, TOO

%*DEFINE (WHAT_BAUD) (
%SET(REPLY,1)
%WHILE(%REPLY) (
  %OUT (ENTER REQUIRED BAUD RATE ...)
  %SET (BAUD_RATE, %IN)
  %IF ((%BAUD_RATE GE 110) AND (%BAUD_RATE LE 9600)) THEN
    (%SET(REPLY,0))
  ELSE
    (%OUT (VALID RANGE IS 110 TO 9600...))
  FI
)
COUNT_Val EQU %EVAL( (%BAUD_RATE / 27 ) - 13 )
)

5-14
WHAT OTHER GOODIES DOES M.P.L. HAVE?

- DOCUMENTATION – CHAPTER 7 OF ASM86 LANGUAGE REFERENCE MANUAL

CLASS EXERCISE 5.1

WRITE A MACRO THAT WILL MOVE A BYTE STRING FROM ONE LOCATION TO ANOTHER IN MEMORY. THE MACRO SHOULD ACCEPT AND USE THREE PARAMETERS. THEY ARE:

1. SOURCE
2. DESTINATION
3. COUNT

ASSUME THAT SOURCE AND DESTINATION ARE BOTH IN A SEGMENT CURRENTLY ADDRESSED BY DS. YOU MAY DESTROY ES AND CX.
WHERE TO FIND MORE INFORMATION

ASM86 LANGUAGE REFERENCE MANUAL
CHAPTER 4 - ACCESSING DATA--OPERANDS AND EXPRESSIONS
CHAPTER 7 - THE MACRO PROCESSING LANGUAGE

ASM86 MACRO ASSEMBLER OPERATING INSTRUCTIONS
CHAPTER 3 - ASSEMBLER CONTROLS

RELATED TOPICS:

THERE IS ANOTHER TYPE OF MACRO CALLED A CODEMACRO. THIS IS ALMOST
ANOTHER LANGUAGE IN ITSELF. IT COULD BE USED TO RE-WRITE INTEL’S
INSTRUCTION SET MNEMONICS OR ADD CUSTOM INSTRUCTIONS TO THE INSTRUCTION
SET TO HANDLE (FOR INSTANCE) YOUR CUSTOM COPROCESSOR. DETAILS MAY BE
FOUND IN THE ASM86 LANGUAGE REFERENCE MANUAL, APPENDIX A.
CHAPTER 6
COMPLEX DATA STRUCTURES

• STRUCTURES
• RECORDS
STRUCTURES

- A STRUCTURE IS A CONTIGUOUS COLLECTION OF DISSIMILAR BUT RELATED DATA ELEMENTS.

- THE STRUC DIRECTIVE ALLOWS YOU TO DEFINE A TEMPLATE THAT CAN BE USED TO FORMAT STORAGE ALLOCATION.

**STRUCTURE EXAMPLE**

```
AIRPLANE STRUC
ALTITUDE DW ?
AIRSPEED DW ?
ENGINES DB 4 DUP(?)
FUEL DW ?
AIRPLANE ENDS
```

- THE STRUCTURE DIRECTIVE ONLY SETS UP A TEMPLATE. IT DOES NOT ALLOCATE ANY STORAGE.
DEFINING STORAGE

- USE THE STRUCTURE DEFINITION AS A NEW DATA TYPE

- EXAMPLE

```
DATA SEGMENT
AIRPLANE STRUC
  ALTITUDE DW ?
  AIRSPEED DW ?
  ENGINES DB 4 DUP(?)
  FUEL DW ?
AIRPLANE ENDS

PHANTOM AIRPLANE <> :ALLOCATES STORAGE FOR
  :ONE STRUCTURE
SQUADRON AIRPLANE 5 DUP(< >) :ALLOCATES STORAGE FOR
  :AN ARRAY OF FIVE
  :STRUCTURES.
```

DATA ENDS

---

STORAGE ALLOCATION

<table>
<thead>
<tr>
<th>TYPE:</th>
<th>LENGTH:</th>
<th>SIZE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHANTOM</td>
<td>SQUADRON</td>
<td>SQUADRON(3)</td>
</tr>
<tr>
<td>ALTITUDE</td>
<td>SQUADRON(1)</td>
<td>ALTITUDE</td>
</tr>
<tr>
<td>AIRSPEED</td>
<td>SQUADRON(2)</td>
<td>AIRSPEED</td>
</tr>
<tr>
<td>ENGINES(0)</td>
<td>SQUADRON(3)</td>
<td>ENGINES(0)</td>
</tr>
<tr>
<td>ENGINES(1)</td>
<td>SQUADRON(4)</td>
<td>ENGINES(1)</td>
</tr>
<tr>
<td>ENGINES(2)</td>
<td></td>
<td>ENGINES(2)</td>
</tr>
<tr>
<td>ENGINES(3)</td>
<td></td>
<td>ENGINES(3)</td>
</tr>
<tr>
<td>FUEL</td>
<td></td>
<td>FUEL</td>
</tr>
</tbody>
</table>
## REFERENCING STRUCTURE ELEMENTS

<table>
<thead>
<tr>
<th>NAME</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>SEGMENT</td>
</tr>
<tr>
<td>AIRPLANE</td>
<td>STRUC</td>
</tr>
<tr>
<td>ALTITUDE</td>
<td>DW ?</td>
</tr>
<tr>
<td>AIRSPEED</td>
<td>DW ?</td>
</tr>
<tr>
<td>ENGINES</td>
<td>DB 4 DUP(?)</td>
</tr>
<tr>
<td>FUEL</td>
<td>DW ?</td>
</tr>
<tr>
<td>AIRPLANE</td>
<td>ENDS</td>
</tr>
<tr>
<td>PHANTOM AIRPLANE</td>
<td>5 DUP (&lt; &gt;)</td>
</tr>
<tr>
<td>SQUADRON AIRPLANE</td>
<td>6 DUP « »</td>
</tr>
<tr>
<td>DATA</td>
<td>ENDS</td>
</tr>
<tr>
<td>CODE</td>
<td>SEGMENT</td>
</tr>
<tr>
<td>ASSUME CS:CODE, DS:DATA ; ACCESS PHANTOM'S ALTITUDE MOV AX, PHANTOM.ALTITUDE : ACCESS PHANTOM'S ALTITUDE ; ACCESS THE AIRSPEED OF THE THIRD AIRPLANE IN SQUADRON MOV SQUADRON.AIRSPEED[5 * TYPE SQUADRON], BX ; ACCESS THE THIRD ENGINE OF THE FIFTH AIRPLANE IN SQUADRON MOV DH,SQUADRON.ENGINES[5 * TYPE ENGINE][5 * TYPE SQUADRON]</td>
<td></td>
</tr>
<tr>
<td>CODE</td>
<td>ENDS</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

---

### REFERENCING STRUCTURE ELEMENTS (cont)

- AIRSPEED - DISPLACEMENT INTO STRUCTURE
- 2 * TYPE SQUADRON = INDEX INTO ARRAY
- SQUADRON - OFFSET OF ARRAY BASE
  - ALL INDEXING ON BYTE LEVEL
  - FINAL OFFSET IS SUM OF COMPONENTS
INITIALIZING A STRUCTURE

• TWO APPROACHES

<table>
<thead>
<tr>
<th>AIRPLANE</th>
<th>STRUC</th>
<th>TEST</th>
<th>STRUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTITUDE</td>
<td>DW 5000</td>
<td>SAMPLES DW ?</td>
<td></td>
</tr>
<tr>
<td>AIRSPEED</td>
<td>DW 600</td>
<td>HIGH SCORE DW ?</td>
<td></td>
</tr>
<tr>
<td>ENGINES</td>
<td>DB 0,0,0</td>
<td>LOW SCORE DW ?</td>
<td></td>
</tr>
<tr>
<td>FUEL</td>
<td>DW 500</td>
<td>MEAN DW ?</td>
<td></td>
</tr>
</tbody>
</table>

JET AIRPLANE < >

;CAN ALSO CHANGE ANY OF ;THE INITIALIZED ELEMENTS

PROP AIRPLANE <250,.,200>

CLASS EXERCISE 6.1

DEFINE AND ALLOCATE STORAGE FOR AN ARRAY OF 100 STRUCTURES. EACH OF THE STRUCTURES SHOULD CONTAIN THE FOLLOWING DATA:

LAST_NAME - 10 BYTES
FIRST_NAME - 10 BYTES
MI - 1 BYTE
DIVISION - 1 WORD
DEPT - 1 WORD

WRITE A PROGRAM LOOP TO MAKE EACH EMPLOYEE'S DIVISION NUMBER EQUAL TO 12.
ADDRESSING VARIABLES

• ADDRESSING MODE MODEL

OFFSET = \[ \text{VARIABLE NAME} + \text{[BX]} + \text{[SI]} + \text{[DISPLACEMENT]} \]

• TAILORED FOR BASED ADDRESSING

OFFSET = \[ \text{[BX]} + \text{[SI]} + \text{[DISPLACEMENT]} \]

BASE REG INDEX REG

BASED ADDRESSING MODES

<table>
<thead>
<tr>
<th>ADDRESSING MODE</th>
<th>APPLICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE REG</td>
<td>BASED SCALARS</td>
</tr>
<tr>
<td>BASE REG + DISP</td>
<td>BASED STRUCTURES</td>
</tr>
<tr>
<td>BASE REG INDEX REG</td>
<td>BASED ARRAYS</td>
</tr>
<tr>
<td>BASE REG INDEX REG + DISP</td>
<td>BASED ARRAYS OF STRUCTURES</td>
</tr>
</tbody>
</table>

6-9
RECORDS

- BIT PATTERN USED TO FORMAT BYTES AND WORDS.
- CAN BE USED STRICTLY FOR FIELD REFERENCING (I.E. MASKING).
- CAN ALSO BE USED TO ALLOCATE STORAGE OF FORMATTED DATA.

USING RECORDS

- EXAMPLE
  COORDINATE_FRAME RECORD X:8,Y:8
  00000000 00000000

- ALLOCATE STORAGE USING THE RECORD DEFINITION
  COORD.0  COORDINATE_FRAME  < >
  00000000 00000000
  COORD.abc  COORDINATE_FRAME  φ φ
  00003100 00000011
  COORD_ARRAY  COORDINATE FRAME  50 DUP( < > ) ; 50 CONTIGUOUS COPIES OF COORDINATE_FRAME

- LOAD A RECORD COPY INTO A CPU REGISTER
  MOV  CX,COORD.abc
  MOV  BX,COORD_ARRAY  [4TYPE COORD_ARRAY]
USING RECORDS FOR FIELD REFERENCING ONLY

• EXAMPLE

STATUS_51 RECORD
&DST:1,SYNE:1,FE:1,OE:1,
&PE:1,TEX:1,RXRDY:1,TXRDY:1

- MASK OUT IRRELEVANT BITS USING THE MASK OPERATOR

WAIT_LOOP:
MOV DX,0F8H
IN AL,DX
TEST AL,MASK RXRDY
JZ WAIT_LOOP

USING RECORDS (cont)

• USE RECORD TO SET UP SHIFT COUNT
• PL/M USES LS BIT TO TEST TRUE/FALSE

EXAMPLE:

8251_READY PROC
IN AL,OF8H ; READ STATUS REGISTER
MOV CL,RXRDY
SHR AL,CL ; PUT RXRDY IN LS BIT
RET
8251_READY ENDP
CLASS EXERCISE 6.2

DEFINE A RECORD THAT WILL ALLOW YOU TO ISOLATE
BITS 2 AND 3 OF A BYTE VALUE AS A SINGLE TWO BIT
FIELD

\[
\begin{array}{cccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\square & \square & \square & \square & ? & ? & \square
\end{array}
\]

2.) WRITE AN ASSEMBLY LANGUAGE INSTRUCTION USING THIS
RECORD TO ISOLATE BITS 2 AND 3 OF THE AL REGISTER.

3.) WHAT IS THE TYPE OF THIS RECORD?

WHERE TO FIND MORE INFORMATION...

ASM86 LANGUAGE REFERENCE MANUAL
CHAPTER 3 - DEFINING AND INITIALIZING DATA
CHAPTER 7

GROUPS

• WHAT THEY ARE?
• HOW TO USE THEM?
DEFINING A SEGMENT

NAME SEGMENT [ALIGN TYPE][COMBINE TYPE][CLASSNAME]

NAME ENDS

COMBINING SEGMENTS : PUBLIC SEGMENTS

CODE_1 SEGMENT WORD PUBLIC

CODE_1 ENDS

FEATURES:

- ONE SEGMENT NAME
- COMMON SEGMENT BASE ALLOWS NEAR CALLS/JUMPS, EASY DATA ACCESS
- SEGMENTS JOINED AT NEXT ADDRESS BOUNDARY SATISFYING ALIGN TYPE
- MAXIMUM TOTAL SIZE 64K
COMBINING SEGMENTS : CLASSNAMES

FEATURES:
- Allows gathering of segments destined for the same area in memory
- Each segment has separate name and base
- Maximum class size 1M

ANOTHER SEGMENT 'RAM'
ANOTHER ENDS

COMBINING SEGMENTS : GROUPS

FEATURES:
- Common group base
- Segments have different names
- Maximum total size 64K

... SO WHY NOT USE PUBLIC SEGMENTS?
COMBINING SEGMENTS : GROUPS
(cont)

- UNIQUE SEGMENT NAMES
- LOCATER CAN SEPARATE SEGMENTS TO SUIT MEMORY MAP
- TYPICALLY USED IN SMALL SYSTEMS

USING THE GROUP DIRECTIVE

```
NAME GROUP EXAMPLE
CGROUP GROUP CODE_1, CODE_2, CODE_3
CODE_1 SEGMENT ASSUME CS:CGROUP
  ;
CODE_1 ENDS
CODE_2 SEGMENT ASSUME CS:CGROUP
  ;
CODE_2 ENDS
CODE_3 SEGMENT ASSUME CS:CGROUP
  ;
CODE_3 ENDS
END
```
NOW THAT WE HAVE A GROUP:

CGROUP  GROUP  CODE_1, CODE_2, CODE_3
DGROUP  GROUP  DATA_1, DATA_2, NV_DATA
CODE_3  SEGMENT BYTE
ASSUME CS:CGROUP, DS:DGROUP, SS:STACK

CONSTANT  DW 1\$ 
MOV  AX, DGROUP  
MOV  DS, AX  
LEA  BX, CONSTANT  ; OK !
MOV  BX,OFFSET CONSTANT  ; OFFSET IS FROM CODE_3 
MOV  BX,OFFSET CGROUP:CONSTANT  
MOV  AL, [BX]

... WHICH OF THOSE LAST TWO OFFSETS IS LIKELY TO GENERATE THE CORRECT ADDRESS?

BEWARE 1! 1

7-7

WOULD YOU USE A GROUP FOR BUILDING ONE STACK?

W{STACK SEGMENT STACK "STACK" }  
DW 100 DUP (?)  

STACK 1  

STACK 2  

T.O.S STACK ENDS

THE STACK IS ALWAYS A SINGLE CONTIGUOUS SEGMENT WITH COMBINE TYPE 'STACK'

7-8
WHERE TO FIND MORE INFORMATION...

ASM86 LANGUAGE REFERENCE MANUAL
CHAPTER 2 - SEGMENTATION

AN INTRODUCTION TO ASM86
CHAPTER 4 - MODULAR PROGRAMMING
CHAPTER 8

LINK86 AND LOC86
DEVELOPMENT CYCLE WITH LINK86 AND LOC86

LINK86 COMMAND

- OUTPUT FILE DEFAULTS TO 1ST INPUT FILE WITH .LNK EXTENSION.
- MAP FILE DEFAULTS TO OUTPUT FILE WITH .MP1 EXTENSION.
LINK86 CONTROLS

LINK86 input list

**NAME**(mod-name)**

**MAP**/**NOMAP**

**SYMBOLS**/**NOSYMBOLS**

**LINES**/**NOLINES**

PRINT (file-name)**/**NOPRINT**

**SYMBOLCOLUMNS** (1/2/3/4)

**TYPE**/**NOTYPE**

**PURGE**/**NOPURGE**

**BIND**/**NObIND**

* - DEFAULT

NOTES:

1) OTHER CONTROLS ARE AVAILABLE.
2) THE CONTROLS CAN BE ABBREVIATED.
3) SEE THE iAPX 86,88 FAMILY UTILITIES USER'S GUIDE CHAPTER 2 FOR DETAILS

How Modules Are Linked

-Run LINK86 MOD1.OBJ, **MOD2.OBJ,** MOD3.OBJ,** MOD4.OBJ TO PROGRM.LNK & **NAME (EXAMPLE) BIND

-COPY PROGRM.MP1 TO :LP:

* INPUT LIST PROCESSED IN LEFT TO RIGHT ORDER.
LOC86 COMMAND

[Fn: LOC86 input_file TO output_file controls]

- OUTPUT FILE DEFAULTS TO INPUT FILE MINUS THE EXTENSION.
- MAP FILE DEFAULTS TO OUTPUT FILE WITH .MP2 EXTENSION.

LOC86 CONTROLS

LOC86 input file

- NAME(mod-name)
- MAP*/NOMAP
- SYMBOLS*/NOSYMBOLS
- LINES*/NOLINES
- PUBLICS*/NOPUBLICS
- PRINT (file-name)*/NOPRINT
- SYMBOLCOLUMNS (1/2*/3/4)
- PURGE/NOPURGE*

* - DEFAULT

NOTES:
1) OTHER CONTROLS ARE AVAILABLE.
2) THE CONTROLS CAN BE ABBREVIATED.
3) SEE THE IAPX 86,88 FAMILY UTILITIES USER'S GUIDE CHAPTER 3 FOR DETAILS
BOOTSTRAP CONTROL

**BOOTSTRAP**

START (symbol/segment,offset)

EXAMPLE

-RUN LOC86 MAIN.LNK BOOTSTRAP START (START_ADDR)

RESET

FFFF:0 JMP START_ADDR

<table>
<thead>
<tr>
<th>NAME</th>
<th>MAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC</td>
<td>START_ADDR</td>
</tr>
<tr>
<td>MAIN_SEG</td>
<td>SEGMENT</td>
</tr>
<tr>
<td>ASSUME</td>
<td>CS:MAIN_SEG</td>
</tr>
<tr>
<td>START_ADDR:</td>
<td></td>
</tr>
<tr>
<td>MAIN_SEG</td>
<td>ENDS</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

BOOTSTRAP CONTROL PLACES FAR JUMP AT RESET POINT IN PROGRAM MEMORY

START CONTROL IDENTIFIES ENTRY POINT OF PROGRAM. IF SYMBOL IS USED, IT MUST BE DECLARED PUBLIC.

INITCODE CONTROL

INITCODE (ADDRESS)

EXAMPLE

-RUN LOC86 MAIN.LNK BOOTSTRAP INITCODE (F0000H)

FFFF:0 JMP INITCODE :from "BOOTSTRAP"

STACKFRAME DW stack frame
DATAFRAME DW data frame
EXTRAFRAME DW extra frame

CLI

MOV SS, CS:STACKFRAME
MOV SP, stack offset
MOV DS, CS:DATAFRAME
MOV ES, CS:EXTRAFRAME
JMP program start

SEGMENT INITIALIZATION RECORD:
1. FROM END START, SS:STACKFRAME, DS:DATAFRAME, ES:EXTRAFRAME, SP:TOS
2. AUTOMATICALLY PRODUCED BY COMPILERS FOR MAIN MODULES
LOCATE CONTROLS

ADDRESSES

\[
\begin{align*}
&\text{SEGMENTS} \ (\text{segname} (\ addr), \ldots) \\
&\text{CLASSES} \ (\text{classname} (\ addr), \ldots) \\
&\text{*GROUPS} \ (\text{groupname} (\ addr), \ldots)
\end{align*}
\]

ORDER

\[
\begin{align*}
&\text{SEGMENTS} \ (\text{segname}, \ldots) \\
&\text{CLASSES} \ (\text{classname}, \ldots)
\end{align*}
\]

SEGSIZE

\[
\text{(SEGNAME} \ (|\pm| \ \text{VALUE}), \ldots)
\]

RESERVE

\[
\text{(addr TO addr, \ldots)}
\]

* *GROUPS* WILL NOT LOCATE A GROUP. IT WILL ASSIGN A GROUP BASE ADDRESS. ALL SEGMENTS IN GROUP MUST ALREADY BE WITHIN 64K (TYPICALLY FROM LOCATION BY CLASS).

LOCATE SEQUENCE

- OUTPUT OF LINKER
- LOCATE ALL ABSOLUTE SEGMENTS, TAG ALL RESERVED AREAS IN MEMORY
- PUT REMAINING SEGMENTS INTO ORDERED LIST (NEXT PAGE)
- REMOVE ADDRESSED SEGMENTS FROM LIST AND LOCATE THEM
- REMOVE ADDRESSED CLASSES FROM LIST AND LOCATE THEM

...CONTINUED
LOCATE SEQUENCE (cont)

WAS THERE AN ORDER CONTROL?

YES

SET SCAN COUNTER TO END OF LAST SEGMENT IN ORDER

NO

SET SCAN COUNTER TO 200H

LOCATE NEXT SEGMENT FROM LIST
AT NEXT FREE (AND LARGE ENOUGH) SPACE

MORE SEGMENTS?

YES

NO

FINISHED

THE ORDERED LIST

- USED TO LOCATE SEGMENTS NOT ASSIGNED TO ADDRESSES OR ORDER CONTROLS.

START

ADD SEGMENTS AND CLASSES SPECIFIED IN ORDER CONTROL

ANY MORE SEGMENTS?

YES

ADD NEXT SEGMENT TO SEQUENCED LIST, SET CURRENT CLASS

ANY MORE SEGMENTS OF CURRENT CLASS?

YES

ADD NEXT SEGMENT TO END OF LIST.

NO

DONE
EXAMPLE: AN ORDERED LIST

RUN LOC86 EXAMPL.LNK ORDER(SEGMENTS(D),CLASSES(X2))

<table>
<thead>
<tr>
<th>SEGMENT NAME</th>
<th>CLASS NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X1</td>
</tr>
<tr>
<td>B</td>
<td>X2</td>
</tr>
<tr>
<td>C</td>
<td>X2</td>
</tr>
<tr>
<td>D</td>
<td>X1</td>
</tr>
<tr>
<td>E</td>
<td>X3</td>
</tr>
<tr>
<td>F</td>
<td>X1</td>
</tr>
<tr>
<td>G</td>
<td>X2</td>
</tr>
<tr>
<td>H</td>
<td>X2</td>
</tr>
<tr>
<td>I</td>
<td>X1</td>
</tr>
</tbody>
</table>

... CONTINUED

EXAMPLE (CON'T.)

... ADDRESSES (SEGMENTS (D(100H),H(8000H)), CLASSES(X2(1500H)))

ORDERED LIST:

<table>
<thead>
<tr>
<th>SEGMENT NAME</th>
<th>CLASS NAME</th>
<th>SEGMENT</th>
<th>CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>X1</td>
<td>100H</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>X2</td>
<td>1500H</td>
<td>1500H</td>
</tr>
<tr>
<td>C</td>
<td>X2</td>
<td>8000H</td>
<td>8000H</td>
</tr>
<tr>
<td>G</td>
<td>X2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>X2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>X1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>X1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>X1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>X3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CLASS EXERCISE: USE OF LINK AND LOCATE

You are required to write the correct link and locate controls to locate your finished program as demanded by the address map of your hardware. The requirements are illustrated on the next page.

The three input modules are:

1. PROG.OBJ written in PL/M. It defines the use of DGROUP
2. PROC.OBJ written in ASM86. It defines the segment NVM
3. SMALL.LIB a support library for the small model of PL/M

The classes in DGROUP should appear in the order shown, with the first class starting at address 200H. Note that if the locater tries to locate a class where a segment is already located, it will locate the class at the next available location (this should help with INITCODE and the class 'CODE').

CLASS EXERCISE: ADDRESS MAP

NON-VOLATILE MEMORY

SEGMENT:NVM

CLASS:'STACK'

CLASS:'CONST'

CLASS:'DATA'

INTERRUPT VECTORS

SEGMENT: NVM

CLASS: CODE

(BOOTSTRAP)

(INITCODE)
WHERE TO FIND MORE INFORMATION...

iAPX 86/88 FAMILY UTILITIES USER’S GUIDE
CHAPTER 9
LINKING ASM86 WITH PL/M 86

- PL/M PROCEDURE DECLARATIONS
- PARAMETER PASSING
- COMPATIBLE DATA TYPES
- COMPILATIONS MODELS
- CONVENTIONS FOR MEMORY ALLOCATION
- CONVENTIONS FOR PROCEDURE AND LABEL DEFINITIONS
- CONVENTIONS FOR DATA DEFINITIONS
LINKING ASSEMBLY LANGUAGE MODULES WITH PL/M MODULES

• A HIGH LEVEL LANGUAGE (HLL) COMPILER USES A STANDARD SET OF RULES AND CONVENTIONS IN DEFINING CODE AND DATA.

• AN ASSEMBLY LANGUAGE MODULE TO BE LINKED TO A HLL MODULE MUST BE DESIGNED SUCH THAT IT SUPPORTS THESE RULES AND CONVENTIONS.

• GENERALLY, THE LINKAGE OF ASSEMBLY LANGUAGE AND A HIGH LEVEL LANGUAGE IS IMPLEMENTED ON A PROCEDURE BASIS.

PL/M PROCEDURE DECLARATION

proc_name: PROCEDURE [(parm1, parm2, ...)] [type];

[DECLARE parm1...]

(PROCEDURE BODY)

[RETURN value]

END proc_name;

• A PL/M PROCEDURE CAN ACCEPT AS MANY INPUT PARAMETERS AS REQUIRED.

• A PL/M PROCEDURE CAN ALSO RETURN A SINGLE ITEM OF THE TYPE DEFINED IN THE PROCEDURE DECLARATION.
UNTYPED PL/M PROCEDURES

• DEFINITION
  CLEAR_PORT: PROCEDURE (PORT);
  DECLARE PORT WORD;
  OUTPUT (PORT)=0;
  END CLEAR_PORT;

• INVOCATION
  CALL CLEAR_PORT (20);

TYPED PL/M PROCEDURES

• DEFINITION
  ADD: PROCEDURE (PARM1,PARM2) WORD;
  DECLARE PARM1 BYTE,
  PARM2 BYTE;
  RETURN PARM1+PARM2;
  END ADD;

• INVOCATION
  SUM=ADD (XYZ,ABC);
## Compatible Data Types

<table>
<thead>
<tr>
<th>PL/M Data Type</th>
<th>Assembly Language Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>DB</td>
</tr>
<tr>
<td>WORD</td>
<td>DW</td>
</tr>
<tr>
<td>INTEGER</td>
<td>DW</td>
</tr>
<tr>
<td>POINTER</td>
<td>DW or DD</td>
</tr>
<tr>
<td>REAL</td>
<td>DD</td>
</tr>
<tr>
<td>STRUCTURE</td>
<td>STRUC</td>
</tr>
</tbody>
</table>

## PL/M Parameter Passing

```
PROC1: PROCEDURE (PARM1,PARM2,PARM3) BYTE;
DECLARE PARM1 BYTE,
PARM2 WORD,
PARM3 WORD,
RESULT BYTE;

RETURN RESULT;
END PROC1;
```

- BYTE - AL
- WORD - AX
- INTEGER - AX
- POINTER (SMALL) - BX OR ES:BX
- POINTER (COMPACT, MEDIUM, LARGE) - ES:BX
- REAL - TOP OF REAL MATH UNIT STACK
PASSING REAL PARAMETERS

* THE FIRST SEVEN REAL PARAMETERS ARE PASSED ON THE MATH UNIT STACK. ANY REMAINING ONES ARE PASSED ON THE CPU STACK.

PROC2: PROCEDURE (PARM1, PARM2, PARM3, PARM4);
DECLARE PARM1 BYTE,
        PARM2 REAL,
        PARM3 REAL,
        PARM4 INTEGER;
END PROC2;

WHICH REGISTERS CAN A PROCEDURE MODIFY?

<table>
<thead>
<tr>
<th>REGISTER</th>
<th>MUST PRESERVE</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>NO</td>
<td>Return BYTE (AL), WORD and INTEGER values</td>
</tr>
<tr>
<td>BX</td>
<td>NO</td>
<td>Return POINTER values</td>
</tr>
<tr>
<td>CX</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>DX</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>YES*</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>BP</td>
<td>YES</td>
<td>Stack marker</td>
</tr>
<tr>
<td>SI</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>FLAGS</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>YES</td>
<td>Caller's code segment</td>
</tr>
<tr>
<td>DS</td>
<td>YES</td>
<td>Caller's data segment</td>
</tr>
<tr>
<td>SS</td>
<td>YES</td>
<td>Caller's stack segment</td>
</tr>
<tr>
<td>ES</td>
<td>NO</td>
<td>Return POINTER values</td>
</tr>
</tbody>
</table>

* SP MUST BE ADJUSTED SO THAT ALL PARAMETERS ARE REMOVED FROM THE STACK UPON RETURN.
ASSEMBLY LANGUAGE INTERFACE RULES

- An assembly language procedure which is called by a HLL program must remove all parameters from the stack.

- An assembly language program can expect the stack, upon return from HLL procedure, to no longer contain the parameters it pushed.

- An assembly language procedure which is called by a HLL program must save DS, SS, SP, and BP, if they are to be modified.

- An assembly language program calling a HLL procedure cannot expect any registers except DS, SS, SP, and BP to be preserved.

EXAMPLE

- A PL/M compatible procedure is required to find the mean of two values. Assume that the procedure must be of type FAR.

Given:

```
MEAN: PROCEDURE (PARM1, PARM2) INTEGER EXTERNAL;
    DECLARE PARM1 INTEGER,
    PARM2 INTEGER;
END MEAN;
```

WHERE DO WE FIND THE INPUT PARAMETERS?
WHERE DO WE LEAVE THE RESULT?
EXAMPLE (CONT.)

PARAMETER PASSING

How do we reference parameters on the stack?

EXAMPLE (CONT.)

Stack frame after saving the BP register

9-11

9-12
EXAMPLE (CONT.)
ASSEMBLY LANGUAGE MODULE

NAME MEAN_VALUE
PUBLIC MEAN
MEAN_SEG SEGMENT 'CODE'
ASSUME CS:MEAN_SEG
MEAN PROC FAR
PUSH BP ;SAVE CALLER'S BP.
MOV BP, SP ;SET UP NEW BP.
MOV AX, [BP+8] ;GET PARM1.
ADD AX, [BP+8] ;ADD IT TO PARM2.
SAR AX, 1 ;DIVIDE RESULT BY 2,
POP BP ;RESTORE BP.
RET 4 ;RETURN AND CLEAN UP STACK.
;RESULT LEFT IN AX.
MEAN ENDP
MEAN_SEG ENDS
END

USING A STRUCTURE AS A STACK TEMPLATE

NAME MEAN_VALUE_1
PUBLIC MEAN
STACK_FRAME STRUCT
OLD_BP DW ?
RET_ADDR DD ?
PARM2 DW ?
PARM1 DW ?
STACK_FRAME ENDS
MEAN_SEG SEGMENT
ASSUME CS:MEAN_SEG
MEAN PROC FAR
PUSH BP ;SAVE CALLER'S BP.
MOV BP, SP ;SET UP NEW BP.
MOV AX, [BP].PARM1 ;GET PARM1.
ADD AX, [BP].PARM2 ;ADD IT TO PARM2.
SAR AX, 1 ;DIVIDE RESULT BY 2,
POP BP ;RESTORE BP.
RET 4 ;RETURN AND CLEAN UP STACK.
;RESULT LEFT IN AX.
MEAN ENDP
MEAN_SEG ENDS
END
CLASS EXERCISE 9.1

WRITE A PL/M COMPATIBLE PROCEDURE THAT WILL COMPARE TWO BYTE ARRAYS FOR "COUNT" NUMBER OF BYTES. IF THE STRINGS COMPARE, RETURN A VALUE OF TRUE (OFFH). OTHERWISE, RETURN A VALUE OF FALSE (0).

REFER TO THE FOLLOWING PL/M PROCEDURE DECLARATION WHEN WRITING YOUR CODE:

```
CMP_STRING: PROCEDURE (STR1_PTR,STR2_PTR,COUNT) BYTE EXTERNAL;
    DECLARE (STR1_PTR,STR2_PTR) POINTER,
    COUNT WORD;
END CMP_STRING;
```

PLACE YOUR CODE IN A GROUP NAMED CGROUP. PLACE ANY DATA YOU DEFINE IN A GROUP NAMED DGROUP. ASSUME THAT THE DS REGISTER IS ALREADY POINTING TO DGROUP. ALSO, ASSUME THAT ALL DATA POINTERS ARE 16 BITS AND THAT ALL PROCEDURES ARE OF TYPE NEAR.

9-15

PL/M MEMORY ALLOCATION

- THE OBJECT MODULE PRODUCED BY THE COMPILER CONTAINS FIVE SECTIONS OR SEGMENTS.
- THE MAXIMUM SIZE ALLOWABLE FOR EACH OF THESE SEGMENTS IS DETERMINED BY THE SELECTED COMPILER SIZE CONTROL.

- SMALL  - MEDIUM
- COMPACT - LARGE
SMALL MODEL

* - THE CONST SEGMENT CAN BE PLACED IN CGROUP IF THE "ROM" CONTROL IS SPECIFIED AT COMPILE TIME.

COMPACT MODEL

* - THE CONST SEGMENT CAN BE PLACED IN CGROUP IF THE "ROM" CONTROL IS SPECIFIED AT COMPILE TIME.
MEDIUM MODEL

**MULTIPLE CODE SEGMENTS**

* - THE CONST SEGMENT WITHIN EACH MODULE CAN BE MERGED WITH ITS CORRESPONDING CODE SEGMENT IF THE "ROM" CONTROL IS USED AT COMPILc TIME.

9-19

---

LARGE MODEL

**MULTIPLE CODE SEGMENTS**

**MULTIPLE DATA SEGMENTS**

NOTE: THE CONST SEGMENT WITHIN EACH MODULE IS MERGED WITH ITS CORRESPONDING CODE SEGMENT.

9-20
PLM CLASS NAMES

<table>
<thead>
<tr>
<th>TYPE OF SEGMENT</th>
<th>CLASS NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>CODE</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>CONST *</td>
</tr>
<tr>
<td>DATA</td>
<td>DATA</td>
</tr>
<tr>
<td>STACK</td>
<td>STACK</td>
</tr>
<tr>
<td>MEMORY</td>
<td>MEMORY</td>
</tr>
</tbody>
</table>

*CONSTANTS ARE MERGED WITH THE CODE SEGMENT WHEN USING LARGE MODEL.

CONVENTIONS FOR PROCEDURES AND PROGRAM LABEL DEFINITIONS
SMALL AND COMPACT MODELS

- ALL LOGICAL CODE SEGMENTS ARE CONTAINED IN ONE PHYSICAL GROUP NAMED CGROUP.
- ALL PROCEDURES AND PROGRAM LABELS COMMON TO BOTH PL/M AND ASSEMBLY LANGUAGE MODULES MUST BE DEFINED AS NEAR.
CONVENTIONS FOR PROCEDURE AND PROGRAM LABEL DEFINITIONS
MEDIUM AND LARGE MODELS

NAME: CODE_EXAMPLE_2
PUBLIC: START, PROC1
EXTERN: PROC2:FAR, PROC3:FAR

CODE1
SEGMENT 'CODE'
ASSUME CS:CODE1

PROC1
PROC FAR
  * * *
  * * *
RET
ENDP

START:
CALL PROC1
CALL PROC2
CALL PROC3
JMP START

CODE1 ENDS

END

• CODE IS CONTAINED IN A NUMBER OF PHYSICAL CODE SEGMENTS.

• ALL PROCEDURES AND PROGRAM LABELS COMMON TO BOTH PL/M AND ASSEMBLY LANGUAGE MODULES MUST BE DEFINED AS FAR.

CONVENTIONS FOR DATA DEFINITIONS
SMALL MODEL

NAME: DATA_EXAMPLE_1

DGROUP
GROUP CONST1, DATA1, STACK, MEMORY

CONST1
SEGMENT PUBLIC 'DATA'
  - Constant data definitions go here.
    Don't forget that constants could be merged with the code segments.
  - CONST1 ENDS

DATA1
SEGMENT PUBLIC 'DATA'
  - Variable data definitions go here.
  - DATA1 ENDS

STACK
SEGMENT STACK 'STACK'
  - Stack definitions go here.
  - STACK ENDS

MEMORY
SEGMENT MEMORY 'MEMORY'
  - Data to be placed in the memory segment
    is defined here.
  - MEMORY ENDS

CGROUP
GROUP CODE1

CODE1
SEGMENT PUBLIC 'CODE'
ASSUME CS:GROUP
ASSUME DS:GROUP
  - CODE1 ENDS

END

• ALL PROGRAM DATA IS CONTAINED IN A GROUP NAMED DGROUP.

• DATA POINTERS IN SMALL MODEL WITH CONSTANTS IN DGROUP ARE 16 BITS (OFFSET ONLY). WITH CONSTANTS IN CGROUP, THE DATA POINTERS ARE 32 BITS. (SEGMENT: OFFSET)
### CONVENTIONS FOR DATA DEFINITIONS

#### MEDIUM MODEL

<table>
<thead>
<tr>
<th>NAME</th>
<th>DATA_EXAMPLE_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGROUP</td>
<td>GROUP CONST1,DATA1,STACK,MEMORY</td>
</tr>
<tr>
<td>CONST1</td>
<td>SEGMENT PUBLIC 'DATA'</td>
</tr>
<tr>
<td></td>
<td>Constant data definitions go here.</td>
</tr>
<tr>
<td></td>
<td>Don't forget that constants could be merged with the code segments.</td>
</tr>
<tr>
<td>CONST1</td>
<td>ENDS</td>
</tr>
<tr>
<td>DATA1</td>
<td>SEGMENT PUBLIC 'DATA'</td>
</tr>
<tr>
<td></td>
<td>Variable data definitions go here.</td>
</tr>
<tr>
<td>DATA1</td>
<td>ENDS</td>
</tr>
<tr>
<td>STACK</td>
<td>SEGMENT STACK 'STACK'</td>
</tr>
<tr>
<td></td>
<td>Stack definitions go here.</td>
</tr>
<tr>
<td></td>
<td>Make sure that the segment definition is identical to the one used by PL/M.</td>
</tr>
<tr>
<td>STACK</td>
<td>ENDS</td>
</tr>
<tr>
<td>MEMORY</td>
<td>SEGMENT MEMORY 'MEMORY'</td>
</tr>
<tr>
<td></td>
<td>Data to be placed in the memory segment is defined here.</td>
</tr>
<tr>
<td></td>
<td>Make sure that the segment definition is identical to the one used by PL/M.</td>
</tr>
<tr>
<td>MEMORY</td>
<td>ENDS</td>
</tr>
<tr>
<td>CODE1</td>
<td>SEGMENT 'CODE'</td>
</tr>
<tr>
<td></td>
<td>ASSUME CS:CODE1</td>
</tr>
<tr>
<td></td>
<td>ASSUME DS:DGROUP, SS:DGROUP</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>CODE1</td>
<td>ENDS</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

- ALL PROGRAM DATA IS CONTAINED IN A GROUP NAMED DGROUP.
- DATA POINTERS IN MEDIUM MODEL ARE 32 BITS (SEGMENT : OFFSET).

#### COMPACT MODEL

<table>
<thead>
<tr>
<th>NAME</th>
<th>DATA_EXAMPLE_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGROUP</td>
<td>GROUP CONST1,DATA1</td>
</tr>
<tr>
<td>CONST1</td>
<td>SEGMENT PUBLIC 'DATA'</td>
</tr>
<tr>
<td></td>
<td>Constant data definitions go here.</td>
</tr>
<tr>
<td></td>
<td>Don't forget that constants could be merged with the code segments.</td>
</tr>
<tr>
<td>CONST1</td>
<td>ENDS</td>
</tr>
<tr>
<td>DATA1</td>
<td>SEGMENT PUBLIC 'DATA'</td>
</tr>
<tr>
<td></td>
<td>Variable data definitions go here.</td>
</tr>
<tr>
<td>DATA1</td>
<td>ENDS</td>
</tr>
<tr>
<td>STACK</td>
<td>SEGMENT STACK 'STACK'</td>
</tr>
<tr>
<td></td>
<td>Stack definitions go here.</td>
</tr>
<tr>
<td></td>
<td>Make sure that the segment definition is identical to the one used by PL/M.</td>
</tr>
<tr>
<td>STACK</td>
<td>ENDS</td>
</tr>
<tr>
<td>MEMORY</td>
<td>SEGMENT MEMORY 'MEMORY'</td>
</tr>
<tr>
<td></td>
<td>Data to be placed in the memory segment is defined here.</td>
</tr>
<tr>
<td></td>
<td>Make sure that the segment definition is identical to the one used by PL/M.</td>
</tr>
<tr>
<td>MEMORY</td>
<td>ENDS</td>
</tr>
<tr>
<td>CODE1</td>
<td>SEGMENT 'CODE'</td>
</tr>
<tr>
<td></td>
<td>ASSUME CS:CODE1</td>
</tr>
<tr>
<td></td>
<td>ASSUME DS:DGROUP, SS:DGROUP</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>CODE1</td>
<td>ENDS</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

- VARIABLE AND CONSTANT DATA IS CONTAINED IN A GROUP NAMED DGROUP.
- STACK AND MEMORY ARE EACH ALLOCATED ONE PHYSICAL SEGMENT.
- DATA POINTERS ARE 32 BITS (SEGMENT : OFFSET).
CONVENTIONS FOR DATA DEFINITIONS
LARGE MODEL

NAME DATA_EXAMPLE_4

DATA1 SEGMENT 'DATA'
; Variable data definitions go here.
ENDS

DATA2 SEGMENT 'DATA'
; Variable data definitions go here.
ENDS

STACK SEGMENT STACK 'STACK'
; Stack definitions go here.
; Make sure that the segment definition
; is identical to the one used by PL/M.
ENDS

MEMORY SEGMENT MEMORY 'MEMORY'
; Data to be placed in the memory segment
; is defined here.
; Make sure that the segment definition
; is identical to the one used by PL/M.
ENDS

CODE1 SEGMENT 'CODE'
ASSUME CS:CODE1,DS:DATA1,SS:STACK
; Constants are defined within the code
; segment.
;
; CODE1 ENDS
END

• VARIABLE DATA IS CONTAINED IN MULTIPLE DATA SEGMENTS.
• CONSTANT DATA IS MERGED WITH A MODULE'S CODE SEGMENT.
• STACK AND MEMORY ARE EACH ALLOCATED ONE PHYSICAL SEGMENT.
• DATA POINTERS ARE ALL 32 BITS (SEGMENT : OFFSET).

EXAMPLE

• A PL/M COMPATIBLE PROCEDURE IS REQUIRED TO SUM THE ELEMENTS OF A BYTE ARRAY. ASSUME THAT THE PL/M MODULE HAS BEEN COMPILED USING THE SMALL MODEL OF SEGMENTATION.

GIVEN:

ARRAY_SUM: PROCEDURE(ARRAY_PTR,ELEMENTS) INTEGER EXTERNAL;
DECLARE ARRAY_PTR POINTER,
   ELEMENTS WORD;
END ARRAY_SUM;

WHAT MAKES UP A POINTER IN SMALL MODEL?
EXAMPLE (CONT.)

STACK FRAME

HIGH MEMORY

STACK GROWS

SP

LOW MEMORY

EXAMPLE (CONT.)

ASSEMBLY LANGUAGE MODULE

NAME ARRAY_SUM_MOD
PUBLIC ARRAY_SUM
CGROUP GROUP ARRAY_SUM_SEG
ARRAY_SUM_SEG SEGMENT 'CODE'
ASSUME CS:CGROUP

ARRAY_SUM PROC NEAR

PUSH BP ;SAVE CALLER'S BP.
MOV BP,SP ;SET UP NEW BP.
MOV BX, [BP+6] ;SET UP ARRAY POINTER.
MOV CX, [BP+4] ;SET UP ITEM COUNT.
MOV AX,0 ;CLEAR SUM.
AGAIN:

ADD AX, DS: [BX] ;ADD ARRAY ELEMENT TO SUM.
INC BX ;UPDATE ARRAY POINTER.
LOOP AGAIN ;IF CX#0, DO IT AGAIN.
POP BP ;RESTORE BP.
RET 4 ;RETURN AND CLEAN UP STACK.

ARRAY_SUM ENDP
ARRAY_SUM_SEG ENDS
END
LOADING POINTERS

16 BIT POINTERS

- PTR1
- RET ADDR
- OLD BP

SP → BP+4

SP → BP

LOW MEMORY

MOV BX,[BP+4]

32 BIT POINTERS

- SEGMENT
- PTR2
- OFFSET
- RET ADDR
- OFFSET
- OLD BP

SP → BP+6

SP → BP

LOW MEMORY

DB

- PTR2(SEGMENT)

LDS BX,[BP+6] → BX

OR

- PTR2(OFFSET)

LES BX,[BP+6] → BX

- PTR2(OFFSET)

CLASS EXERCISE 9.2

REWRITE THE ARRAY SUM PROCEDURE. THIS TIME ASSUME THAT IT MUST INTERFACE WITH A PL/M MODULE COMPILED LARGE
WHERE TO FIND MORE INFORMATION...

AN INTRODUCTION TO ASM86
CHAPTER 5 - COMBINING ASM86 AND PL/M-86 MODULES

PL/M-86 USER'S GUIDE
APPENDIX F - LINKING TO MODULES WRITTEN IN OTHER LANGUAGES
CHAPTER 10

LINKAGE WITH OTHER HIGH LEVEL LANGUAGES

- LINKING WITH 'C'
- LINKAGE WITH PASCAL
- LINKAGE WITH FORTRAN
THINGS TO CONSIDER WHEN LINKING TO HLL'S

- COMPATIBLE DATA TYPES
- COMPILATION MODELS (SMALL, LARGE ETC.)
- PASSING PARAMETERS TO PROCEDURES
- MANY PRINCIPLES OF LINKING TO PL/M ARE APPLICABLE

COMPATIBLE DATA TYPES

<table>
<thead>
<tr>
<th>ASMR85</th>
<th>PL/M86</th>
<th>PASCAL</th>
<th>FORTRAN 'C'</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UNSIGNED DATA TYPES</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB</td>
<td>BYTE</td>
<td>CHAR</td>
<td>CHARACTER</td>
<td>CHAR</td>
</tr>
<tr>
<td>DM</td>
<td>WORD</td>
<td>WORD</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>DD</td>
<td>DWord</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td><strong>INTEGERS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB</td>
<td>INTEGER</td>
<td>INTEGER*1</td>
<td>INTEGER*2</td>
<td>SHORT</td>
</tr>
<tr>
<td>DM</td>
<td>INTEGER*3</td>
<td>INTEGER*4</td>
<td>LONG</td>
<td>8087 SHORT INTEGER</td>
</tr>
<tr>
<td>DD</td>
<td>INTEGER*5</td>
<td>INTEGER*6</td>
<td>INTEGER*7</td>
<td>LONG</td>
</tr>
<tr>
<td><strong>BOOLEAN VALUES</strong> (i.e. TRUE/FALSE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB</td>
<td>BYTE</td>
<td>BOOLEAN</td>
<td>LOGICAL*1</td>
<td>------</td>
</tr>
<tr>
<td>DM</td>
<td>-----</td>
<td>-------</td>
<td>LOGICAL*2</td>
<td>------</td>
</tr>
<tr>
<td>DD</td>
<td>-----</td>
<td>-------</td>
<td>LOGICAL*4</td>
<td>------</td>
</tr>
<tr>
<td><strong>REAL NUMBERS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DD</td>
<td>REAL</td>
<td>REAL</td>
<td>REAL*4</td>
<td>FLOAT</td>
</tr>
<tr>
<td>DD</td>
<td>------</td>
<td>------</td>
<td>REAL*8</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DD</td>
<td>------</td>
<td>------</td>
<td>REAL*9</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DT</td>
<td>LONGREAL</td>
<td>TEMPREAL</td>
<td>TEMPREAL</td>
<td>-------</td>
</tr>
</tbody>
</table>

* OTHER DATA TYPES: THESE LANGUAGES SUPPORT ARRAYS AND STRUCTURES IN VARYING DEGREES. THEY ALSO USE POINTERS (16 BITS IN SMALL, 32 BITS OTHERWISE). SEE APPROPRIATE LANGUAGE REFERENCE MANUAL FOR DETAILS.
**COMPILATION MODELS**

- LANGUAGES SAME CONVENTIONS (CLASS NAMES, GROUPS ETC.) AS PL/M.

<table>
<thead>
<tr>
<th></th>
<th>SMALL</th>
<th>COMPACT</th>
<th>MEDIUM</th>
<th>LARGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/M</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PASCAL</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORTRAN</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**PARAMETER PASSING**

- ALL THREE LANGUAGES PASS PARAMETERS INTO PROCEDURES ON STACK.
- RETURNED VALUES (FUNCTIONS, TYPED PROCEDURES) PASSED IN REGISTERS, REALS ON TOP OF 8087 STACK,
- PARAMETERS PASSED IN ONE OF TWO WAYS:
  1) BY VALUE - PARAMETER READ FROM MEMORY AND PUSHED ONTO STACK.
  2) BY REFERENCE - ADDRESS OF PARAMETER IS PASSED TO PROCEDURE. SAME AS PASSING POINTERS IN PL/M.
PARAMETER PASSING : PASCAL

- PARAMETERS USUALLY PASSED BY VALUE
- 'VAR' PARAMETERS PASSED BY REFERENCE
- PARAMETERS PUSHED LEFT-TO-RIGHT
- 8087 STACK USED FOR FIRST SEVEN REALS
- PROCEDURE CLEANS PARAMETERS FROM STACK

EXAMPLE

PROCEDURE PROC (PARM1, PARM2: INTEGER; PARM3: REAL;
VAR PARM4: INTEGER; PARM5: REAL);

PROC (A,B,C,D,E);

PARAMETER PASSING : FORTRAN

- ALL PARAMETERS PASSED BY REFERENCE (ALL POINTERS 32 BITS)
- PARAMETERS PUSHED LEFT-TO-RIGHT
- REALS ALSO PASSED BY REFERENCE
- PROCEDURE CLEANS PARAMETERS FROM STACK

EXAMPLE

SUBROUTINE SBRTNI (PARM1, PARM2, PARM3, PARM4)
CALL SBRTNI (A,B,C,D)
PASSING PARAMETERS OF THE CHARACTER DATA TYPE

- TO PASS A CHARACTER TYPE DATA ARGUMENT, A POINTER TO THE CHARACTER STRING AND THE ACTUAL LENGTH OF THE STRING (IN BYTES) IS PUSHED ON THE STACK.

SUBROUTINE SUBRTN2(CHAR)
CHARACTER 8 CHAR
STACK

PARM1
LENGTH
RETURN ADDRESS

LOW MEMORY

PARAMETER PASSING: 'C'

- ALL PARAMETERS PASSED BY VALUE
- PARAMETERS PUSHED RIGHT-TO-LEFT
- VARIABLE NUMBER OF PARAMETERS ALLOWED
- 8087 STACK USED FOR FIRST SEVEN REALS
- CALLING PROGRAM REMOVES PARAMETERS FROM STACK

INT X,*P; /*X IS INTEGER, P POINTER TO INT*/
INT F(); /*F IS FUNCTION, NO PARAMETER COUNT*/
F(X,P); /*X PASSED BY VALUE, P IS A POINTER*/
HIGH LEVEL LANGUAGE
INTERFACING : CHECK LIST

1. PUBLIC AND EXTERNAL DATA DEFINITIONS MUST MATCH HLL DATA TYPE
2. FOLLOW COMPILATION MODEL (SMALL, COMPACT ...) RULES
   - USE CORRECT CLASSNAMES/GROUPS
   - IF USING GROUPS:
     - CS, DS, ES ADDRESS GROUP BASE (NOT SEGMENT BASE)
     - USE OF MOV BX, OFFSET DGROUP: VARIABLE (OR USE LEA INSTRUCTION)
   - ARE POINTERS (AND RETURN ADDRESSES) 16 BITS OR 32 BITS?
3. PASSING PARAMETERS
   - IS THE STACK FRAME RIGHT?
   - REMOVE CORRECT BYTE COUNT ON RETURN FROM PROCEDURE
   - LEAVE RETURN VALUES IN CORRECT REGISTERS
4. REGISTERS WHICH ONES WILL/MAY BE DESTROYED? BP IS SACRED!

WHERE TO FIND MORE INFORMATION …

PASCAL-86 USER’S GUIDE
   APPENDIX J – LINKING TO MODULES WRITTEN IN OTHER LANGUAGES

FORTRAN-86 USER’S GUIDE
   APPENDIX H – LINKING TO SUBPROGRAMS WRITTEN IN OTHER LANGUAGES

C-86 COMPILER USER’S GUIDE
DAY 3 OBJECTIVES

BY THE TIME YOU FINISH TODAY YOU WILL:

- SEE THE ARCHITECTURE OF THE 8087

- DEFINE THE 8086-8087 INTERFACE (HARDWARE AND SOFTWARE)

- DEFINE THE DATA FORMATS USED FOR REAL, INTEGER AND BCD NUMBERS

- USE THE 8087 INSTRUCTION SET

- INITIALIZE THE 8087

- DISCUSS EXCEPTION HANDLING FOR ARITHMETIC ERRORS

- DEFINE THE USE OF THE 8087 SUPPORT LIBRARIES
CHAPTER 11

INTRODUCTION TO THE 8087 NUMERIC PROCESSOR EXTENSION

- MOTIVATION FOR USING THE 8087
- ARCHITECTURAL DESCRIPTION
- HARDWARE INTERFACE
- SOFTWARE INTERFACE
8087 80-BIT HMOS NUMERIC PROCESSOR EXTENSION

- FULL INTERNAL 80-BIT ARCHITECTURE FOR HIGH PERFORMANCE
- IMPLEMENTS PROPOSED IEEE FLOATING POINT STANDARD
- EXPANDS HOST CPU DATATYPES TO INCLUDE 32-, 64-BIT INTEGERS, 32-, 64-, 80-BIT FLOATING POINT, AND 18-DIGIT BCD OPERANDS
- ALL HOST CPU ADDRESSING MODES AVAILABLE
- DIRECTLY EXTENDS HOST CPU'S INSTRUCTION SET TO TRIGONOMETRIC, LOGARITHMIC, EXPONENTIAL AND ARITHMETIC INSTRUCTIONS FOR ALL DATATYPES
- 8 x 80-BIT, INDIVIDUALLY ADDRESSABLE, NUMERIC REGISTER STACK
- BUILT-IN EXCEPTION HANDLING FUNCTIONS

NOTE: THE 8087 IS AN EXTENSION OF THE HOST CPU
(APX 86,88 OR APX 186,186)

WHY USE AN 8087?

- TO MAKE IT EASIER TO PROGRAM ACCURATE ARITHMETIC SOFTWARE
- TO BRING ABOUT STANDARDIZATION OF NUMERIC PROGRAMS AND DATA
- TO MEET THE HIGH PERFORMANCE MATH REQUIREMENTS OF VARIOUS APPLICATION PROGRAMS
RELIABILITY - WHAT CAN AN 8087 DO FOR YOU?

- THE 8087 IS DESIGNED TO DELIVER STABLE, ACCURATE RESULTS

- IT CAN PROCESS DECIMAL NUMBERS UP TO 18 DIGITS OF SIGNIFICANCE - WITHOUT ROUND-OFF ERRORS

- IT CAN PERFORM EXACT ARITHMETIC ON INTEGERS AS LARGE AS $2^{64}$ (APPROXIMATELY EQUAL TO $1.845 \times 10^{19}$).

STANDARDIZATION

- THE 8087 IS THE FIRST FULL IMPLEMENTATION OF THE PROPOSED IEEE FLOATING POINT STANDARD

- DATA FORMATS AND BASIC ARITHMETIC FUNCTIONS ARE CONSISTENT WITH OTHER INTEL PRODUCTS
  - iSBC-310
  - 8232
  - FPAL
  - ASM-86
  - PL/M-86
  - FORTRAN-86
  - PASCAL-86
### HIGH PERFORMANCE

#### 8087 Evolution and Relative Performance

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Approximate Execution Time (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8087</td>
<td>8086 Emulation</td>
</tr>
<tr>
<td>Multiply (single precision)</td>
<td>19</td>
</tr>
<tr>
<td>Multiply (double precision)</td>
<td>27</td>
</tr>
<tr>
<td>Add</td>
<td>17</td>
</tr>
<tr>
<td>Divide (single precision)</td>
<td>39</td>
</tr>
<tr>
<td>Compare</td>
<td>9</td>
</tr>
<tr>
<td>Load (single precision)</td>
<td>9</td>
</tr>
<tr>
<td>Store (single precision)</td>
<td>18</td>
</tr>
<tr>
<td>Square root</td>
<td>36</td>
</tr>
<tr>
<td>Tangent</td>
<td>90</td>
</tr>
<tr>
<td>Exponential</td>
<td>190</td>
</tr>
</tbody>
</table>

#### 8087 vs Software Comparison

**YEAR INTRODUCED**

1977

**1977 1978 1979 1980**

**100**

**10**

**100**

#### iAPX 86/20, 88/20, 186/20, 188/20 Architecture

- THE 8087 IS AN ARCHITECTURAL EXTENSION OF THE HOST CPU.
- TO USE THE 8087, ADDITIONAL OPCODES AND OPERANDS ARE INCLUDED IN THE HOST CPU's INSTRUCTION SET.
8087 DATA TYPES AND FORMATS

WORD INTEGER
5 MAGNITUDE
15 0

SHORT INTEGER 4 MAGNITUDE
15 0

LONG INTEGER
5 MAGNITUDE
63 0

PACKED DECIMAL
5 MAGNITUDE
216 0

SHORT REAL
5 BIASED EXPONENT SIGNIFICAND
31 23

LONG REAL
5 BIASED EXPONENT SIGNIFICAND
61 54

TEMPORARY REAL
5 BIASED EXPONENT SIGNIFICAND
95 63

NOTES:
5 = Sign bit (0=positive, 1=negative)
4 = Biased sign integer
3 = Fractional part
2 = Integer bits
1 = Exponent
0 = Digit significance

ALL INTERNAL 8087 DATA IS IN THIS FORM. THE SIZE OF THE TEMPORARY REAL FORMAT CONtributes TO THE OVERALL ACCURACY AND STABILITY OF THE 8087

REAL FORMATS

SIGN - β = POSITIVE NUMBER
1 = NEGATIVE NUMBER

EXPONENT - EXPONENT IS BIASED TO ELIMINATE NEED FOR HANDLING NEGATIVE EXPONENTS. SHORT REAL HAS 8 BIT EXPONENT:
	TRUE EXPONENT: -127→+127
	BIASED EXPONENT: 0→+254 BIAS = +127

SIGNIFICAND - CONTAINS SIGNIFICANT BITS (MANTISSA) OF NUMBER. IT IS USUALLY NORMALIZED, MEANING THAT IT CONTAINS BOTH A FRACTION AND WHOLE NUMBER. THIS ENSURES THE GREATEST PRECISION FOR A GIVEN REAL FORMAT

ASSUME WE HAVE A 5 DIGIT SIGNIFICAND AND WE WANT TO REPRESENT THIS NUMBER:

3,174,231
NORMALIZED NUMBER: 3.1742 x 10^6
UNNORMALIZED NUMBER: 0.0031 x 10^8
TEMPORARY REAL FORMAT

MOST SIGNIFICANT BYTE

<table>
<thead>
<tr>
<th>S</th>
<th>E₁₄</th>
<th>E₀</th>
<th>F₆₃</th>
<th>F₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>07</td>
<td>07</td>
<td>07</td>
<td>07</td>
</tr>
</tbody>
</table>

SIGN EXponent SIGNIFICAND

1 BIT 15 BITS 64 Bits

TEMPORARY REAL (CONT.)

EXponent –

SMALLEST -VE NUMBER = \(2^{-16382} \approx 3.36 \times 10^{-4932}\)
LARGEST +VE NUMBER = \(2^{16384} \approx 1.19 \times 10^{4932}\)
RADIUS OF UNIVERSE \(\approx 13\) BILLION LIGHT YEARS

\[ \text{VOLUME} \approx 7.77 \times 10^{84} \text{ CM}^3 \]

IT WOULD TAKE \(\approx 10^{122}\) ELECTRONS TO FILL THIS VOLUME

SIGNIFICAND

ACCURACY OF ONE PART IN \(2^{64}\)

COMPARES WITH THE RADIUS OF A HYDROGEN ATOM
NEXT TO RADIUS OF MOONS ORBIT ABOUT THE EARTH
8086 ↔ 8087 INTERFACE

- Processors can operate in parallel
- Share single instruction stream
- 8087 tracks queue of 8086 using queue status lines
- All 8086 addressing modes available
- 8086 supplies operand addresses to 8087 by issuing dummy read

HARDWARE INTERFACE

- The 8087 has a direct hardware interface with the host CPU
### Principal Instructions of the 8087

<table>
<thead>
<tr>
<th>Class</th>
<th>Instruction Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Transfer</strong></td>
<td>Load and store (for all data types), exchange, free</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>Add, subtract, multiply, divide, subtract reversed, divide reversed, calculate square root, scale, increment, decrement, use remainder, round to integer, change sign, absolute value, extract mantissa or exponent</td>
</tr>
<tr>
<td><strong>Logical/Relational</strong></td>
<td>Compare, examine, test</td>
</tr>
<tr>
<td><strong>Transcendental</strong></td>
<td>Calculate tangent, arctangent, $2^x - 1$, $X \cdot \log_2 X$, $X \cdot \log_2 X + 1$</td>
</tr>
<tr>
<td><strong>Constants</strong></td>
<td>0, 1, $\pi$, $\log_{10} 2$, $\log_2 2$, $\log_2 10$, $\log_2 e$</td>
</tr>
<tr>
<td><strong>Processor Control</strong></td>
<td>Load control word, store control word, store status word, load environment, store environment, save, restore, set interrupt-enable enable, clear interrupt-enable, clear errors, initialize</td>
</tr>
</tbody>
</table>

*Combining these instructions in very simple routines provides all the common trigonometric, inverse hyperbolic, inverse hyperbolic, logarithmic, and power functions.*

### 8087 Software Support

- **8087 Software Emulators**
  - Full 16K Emulator (E8087)
  - Partial 8K Emulator (PE8087) for PL/M-86

- **Language Support**
  - ASM-86
  - PL/M-86
  - FORTRAN-86
  - PASCAL-86

- **Support Libraries**
  - CEL87 Common Elementary Functions
  - DCON87 Decimal Conversion
  - EH87 Error Handler
WHERE TO FIND MORE INFORMATION...

APPLICATION NOTE AP-113 - GETTING STARTED WITH THE NUMERIC DATA PROCESSOR

IAPX 86/88, 186/188 USER'S MANUAL (PROGRAMMER'S REFERENCE)
CHAPTER 6 - THE 8087 NUMERIC PROCESSOR EXTENSION
CHAPTER 12

PROGRAMMING THE 8087

- INSTRUCTION FORMAT
- DATA FORMATS
- DATA TRANSFER INSTRUCTIONS
- ARITHMETIC INSTRUCTIONS
- TRANSCENDENTAL INSTRUCTIONS
- CONSTANT INSTRUCTIONS
INSTRUCTION FORMAT

OPCODE \[
[\text{OP1 \ [\text{OP2}] }]
\]

- Depending on instruction type, one or two operands may be explicitly specified.

- With some instructions, the operand(s) may be implicitly specified.

OPERANDS

- 3 types
  - Implicit reference to the stack top (ST) and possibly the next stack element (ST(1))
    
    EX. FLDZ ; Push 0.0 onto the stack
  
  - Explicit reference to stack element(s)
    
    EX. FADD ST(1),ST ; ST(1) = ST(1) + ST
  
  - Explicit reference to a memory item
    
    EX. FMUL VAR1 ; ST = ST*VAR1
DATA FORMATS FOR MEMORY OPERANDS

WORD INTEGER
- MAGNITUDE
- TWO'S COMPLEMENT

SHORT INTEGER
- MAGNITUDE
- TWO'S COMPLEMENT

LONG INTEGER
- MAGNITUDE
- TWO'S COMPLEMENT

PACKED DECIMAL
- X
- MAGNITUDE

SHORT REAL
- BIASED EXPONENT
- SIGNIFICAND

LONG REAL
- BIASED EXPONENT
- SIGNIFICAND

TEMPORARY REAL
- BIASED EXPONENT
- SIGNIFICAND

STORAGE ALLOCATION DIRECTIVE

<table>
<thead>
<tr>
<th>DIRECTIVE</th>
<th>MEANING</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW</td>
<td>DEFINE WORD</td>
<td>WORD INTEGER</td>
</tr>
<tr>
<td>DD</td>
<td>DEFINE DOUBLEWORD</td>
<td>SHORT INTEGER, SHORT REAL</td>
</tr>
<tr>
<td>DQ</td>
<td>DEFINE QUADWORD</td>
<td>LONG INTEGER, LONG REAL</td>
</tr>
<tr>
<td>DT</td>
<td>DEFINE TENBYTE</td>
<td>PACKED DECIMAL, TEMPORARY REAL</td>
</tr>
</tbody>
</table>

PTR DIRECTIVES

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD PTR</td>
<td>DWORD PTR</td>
</tr>
<tr>
<td>QWORD PTR</td>
<td>TBYTE PTR</td>
</tr>
</tbody>
</table>
## ADDRESSING MODE EXAMPLES

<table>
<thead>
<tr>
<th>CODING</th>
<th>INTERPRETATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIADD</td>
<td>ALPHA is a simple scalar (mode is direct)</td>
</tr>
<tr>
<td>FDIVR</td>
<td>Beta is a field in a structure that is &quot;overlaid&quot; on Alpha (mode is direct)</td>
</tr>
<tr>
<td>FMUL</td>
<td>QWORD PTR [BX] contains the address of a long real variable (mode is register indirect)</td>
</tr>
<tr>
<td>FSUB</td>
<td>Alpha is an array and SI contains the offset of an array element from the start of the array (mode is indexed)</td>
</tr>
<tr>
<td>FILD</td>
<td>BP contains the address of a structure on the CPU stack and Beta is a field in the structure (mode is based)</td>
</tr>
<tr>
<td>FBLD</td>
<td>BX contains the address of a packed decimal array and DI contains the offset of an array element (mode is based indexed)</td>
</tr>
</tbody>
</table>

## INSTRUCTION SET

- DATA TRANSFER
- ARITHMETIC
- LOGICAL/RELATIONAL
- TRANSCENDENTAL
- CONSTANTS
- PROCESSOR CONTROL
### DATA TRANSFER INSTRUCTION

<table>
<thead>
<tr>
<th>REAL TRANSFERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD</td>
</tr>
<tr>
<td>FST</td>
</tr>
<tr>
<td>FSTP</td>
</tr>
<tr>
<td>FXCH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTEGER TRANSFERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILD</td>
</tr>
<tr>
<td>FIST</td>
</tr>
<tr>
<td>FISTP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PACKED DECIMAL TRANSFERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBLD</td>
</tr>
<tr>
<td>FBSTP</td>
</tr>
</tbody>
</table>

- THESE INSTRUCTIONS MOVE OPERANDS AMONG ELEMENTS OF THE REGISTER STACK, AND BETWEEN THE STACK TOP AND MEMORY

---

### REGISTER LOAD

**EX: FLD STD_DEV**

**BEFORE**

<table>
<thead>
<tr>
<th>ST</th>
<th>ST(1)</th>
<th>ST(2)</th>
<th>ST(3)</th>
<th>ST(4)</th>
<th>ST(5)</th>
<th>ST(6)</th>
<th>ST(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

**AFTER**

<table>
<thead>
<tr>
<th>ST</th>
<th>ST(1)</th>
<th>ST(2)</th>
<th>ST(3)</th>
<th>ST(4)</th>
<th>ST(5)</th>
<th>ST(6)</th>
<th>ST(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.75</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

---

12-7

12-8
**REGISTER STORE**

**EX: FIRST RESULT**

<table>
<thead>
<tr>
<th>BEFORE</th>
<th>AFTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td>7.25</td>
<td>7.25</td>
</tr>
<tr>
<td>ST(1)</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(2)</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(6)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(7)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>RESULT</td>
<td>0007</td>
</tr>
</tbody>
</table>

**REGISTER STORE WITH POP**

**EX: FSTP SCORE**

<table>
<thead>
<tr>
<th>BEFORE</th>
<th>AFTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td>4.35</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(1)</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(2)</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(6)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(7)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SCORE</td>
<td>4.35</td>
</tr>
</tbody>
</table>
## ARITHMETIC INSTRUCTIONS

### Addition
- FADD: Add real
- FADDP: Add real and pop
- FIADD: Integer add

### Subtraction
- FSUB: Subtract real
- FSUBP: Subtract real and pop
- FSUBR: Integer subtract
- FSUBRP: Subtract real reversed and pop
- FSUBR: Integer subtract reversed

### Multiplication
- FMUL: Multiply real
- FMULP: Multiply real and pop
- FIMUL: Integer multiply

### Division
- FDIV: Divide real
- FDIVP: Divide real and pop
- FDIVR: Divide integer divide
- FDIVRP: Divide real reversed and pop
- FDIVR: Integer divide reversed

### Notes:
- Braces ( ) surround implicit operands; these are not coded and are shown here for information only.

### Table: Instruction Forms

<table>
<thead>
<tr>
<th>Instruction Form</th>
<th>Mnemonic Form</th>
<th>Operands Form</th>
<th>ASM-86 Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical stack</td>
<td>Fop</td>
<td>(ST(n),ST)</td>
<td>FADD</td>
</tr>
<tr>
<td>Register</td>
<td>Fop</td>
<td>ST(),ST or ST,ST</td>
<td>FSUB ST,ST(3)</td>
</tr>
<tr>
<td>Register pop</td>
<td>FopP</td>
<td>ST(),ST</td>
<td>FMULP ST(2),ST</td>
</tr>
<tr>
<td>Real memory</td>
<td>Fop</td>
<td>ST, short-real/long-real</td>
<td>FDIV AZIMUTH</td>
</tr>
<tr>
<td>Integer memory</td>
<td>Fop</td>
<td>ST, word/integer/short/integer</td>
<td>FIDIV N. PULSES</td>
</tr>
</tbody>
</table>

### Classical Stack Operation

**Ex: FADD**

<table>
<thead>
<tr>
<th>BEFORE</th>
<th>AFTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>3.50</td>
<td>5.25</td>
</tr>
<tr>
<td>ST(1)</td>
<td>ST(1)</td>
</tr>
<tr>
<td>1.75</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(2)</td>
<td>ST(2)</td>
</tr>
<tr>
<td>0.0</td>
<td>ST(3)</td>
</tr>
<tr>
<td>ST(3)</td>
<td>ST(3)</td>
</tr>
<tr>
<td>ST(4)</td>
<td>ST(4)</td>
</tr>
<tr>
<td>ST(5)</td>
<td>ST(5)</td>
</tr>
<tr>
<td>ST(6)</td>
<td>ST(6)</td>
</tr>
<tr>
<td>ST(7)</td>
<td>ST(7)</td>
</tr>
</tbody>
</table>

- **Note:** Classical Stack Mode Includes a POP
REGISTER OPERATION

**EX: FSUB ST(3),ST**

**BEFORE**

<table>
<thead>
<tr>
<th>ST</th>
<th>2.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(1)</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(2)</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(3)</td>
<td>7.00</td>
</tr>
<tr>
<td>ST(4)</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(5)</td>
<td></td>
</tr>
<tr>
<td>ST(6)</td>
<td></td>
</tr>
<tr>
<td>ST(7)</td>
<td></td>
</tr>
</tbody>
</table>

**AFTER**

<table>
<thead>
<tr>
<th>ST</th>
<th>7.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(1)</td>
<td>2.50</td>
</tr>
<tr>
<td>ST(2)</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(3)</td>
<td>4.50</td>
</tr>
<tr>
<td>ST(4)</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(5)</td>
<td></td>
</tr>
<tr>
<td>ST(6)</td>
<td></td>
</tr>
<tr>
<td>ST(7)</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** ONE OF THE OPERANDS MUST BE ST, ST(i),ST OR ST,ST(i)

---

REGISTER OPERATION WITH POP

**EX: FSUBP ST(3),ST**

**BEFORE**

<table>
<thead>
<tr>
<th>ST</th>
<th>2.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(1)</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(2)</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(3)</td>
<td>7.00</td>
</tr>
<tr>
<td>ST(4)</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(5)</td>
<td></td>
</tr>
<tr>
<td>ST(6)</td>
<td></td>
</tr>
<tr>
<td>ST(7)</td>
<td></td>
</tr>
</tbody>
</table>

**AFTER**

<table>
<thead>
<tr>
<th>ST</th>
<th>7.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(1)</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(2)</td>
<td>4.50</td>
</tr>
<tr>
<td>ST(3)</td>
<td>4.50</td>
</tr>
<tr>
<td>ST(4)</td>
<td>0.0</td>
</tr>
<tr>
<td>ST(5)</td>
<td></td>
</tr>
<tr>
<td>ST(6)</td>
<td></td>
</tr>
<tr>
<td>ST(7)</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** OPERANDS MUST BE IN THIS FORM ST(i),ST

---

12-13

12-14
REVERSED OPERATION

EX: FSUBR ST(3),ST

NOTE: THERE IS ALSO A REVERSED FORM OF THE INSTRUCTION FOR DIVISION

MEMORY OPERATION

EX: FIMUL INT1
OTHER ARITHMETIC INSTRUCTIONS

FSQRT - SQUARE ROOT
FScale - SCALE BY INTEGRAL POWERS OF TWO
FPREM - PARTIAL REMAINDER (MODULO REDUCTION)
FRNDINT - ROUND TO INTEGER
FXTRACT - EXTRACT EXPONENT AND SIGNIFICAND
FABS - ABSOLUTE VALUE
FCHS - CHANGE SIGN

EXAMPLE

NAME PYTHAGORUS
EXTRN INIT87:FAR

; Define a structure used to represent a right triangle.
; TRIANGLE STRUC
BASE DD 3.0 ; The DD memory allocation allows
ALT DD 4.0 ; enough space for the variables
HYP DD ; to be defined in the SHORT REAL
AREA DD ? ; format.
TRIANGLE ENDS

DATA SEGMENT PUBLIC 'DATA'
RIGHT TRIANGLE <> ; Allocate storage for one triangle.
TWO DD 2.0 ; Define a real constant equal to 2.
DATA ENDS

STACK SEGMENT STACK 'STACK'
DW 100 DUP(?)
TOS LABEL WORD
STACK ENDS
FLO
TWO
CODE
SEGMENT PUBLIC 'CODE'
ASSUME CS:CODE,DS:DATA,SS:STACK
;
; INITIALIZE 8087
;
INIT: CALL INIT87 ; This routine is in a library.
; It sets up the default environment
; for the 8087.
;
; PLACE INPUT OPERANDS ON 8087 STACK
;
SETUP: FLD TWO ; Put 2.0 in STACK TOP (ST)
FLD RIGHT.BASE ; ST <-- BASE
FLD RIGHT.ALT ; ST <-- ALT
;
; CALCULATE AREA = (BASE*ALT)/2 AND STORE IN MEMORY
;
CALC: FLD ST(1) ; Duplicate BASE in ST
FMUL ST,ST(1) ; ST <-- BASE * ALT
FDIV ST,ST(3) ; ST <-- ST/2
FSTP RIGHT.AREA ; Store ST in AREA then discard
;
; CALCULATE HYPOTENUSE = ((BASE**2)+(ALT**2))**0.5
;
FMUL ST,ST(0) ; Square ALT
FXCH ST(1) ; Exchange ALT**2 and BASE
FMUL ST,ST(0) ; Square BASE
FADD ; ST <-- BASE**2 + ALT**2
FSQRT ; ST <-- ST**0.5
FSTP RIGHT.HYP ; Store ST in HYP then discard
FFREE ST(0) ; Clear out ST
; Register STACK now empty

DONE: HLT

CODE ENDS
END INIT,DS:DATA,SS:STACK:TOS

REGISTER STACK USAGE

<table>
<thead>
<tr>
<th>ST</th>
<th>ALT</th>
<th>BASE</th>
<th>ST</th>
<th>FMUL ST,ST(1)</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>TWO</td>
<td>ST(1)</td>
<td>3.0</td>
<td>BASE</td>
<td>4.0</td>
</tr>
<tr>
<td>3.0</td>
<td>ST(2)</td>
<td>3.0</td>
<td>BASE</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td>ST(3)</td>
<td>TWO</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FDIV ST,ST(3)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ST</th>
<th>AREA</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.0</td>
<td>ST(1)</td>
</tr>
<tr>
<td>4.0</td>
<td>ST(2)</td>
</tr>
<tr>
<td>3.0</td>
<td>ST(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FST RIGHT.AREA</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ST</th>
<th>ALT</th>
<th>BASE</th>
<th>ST</th>
<th>FMUL ST,ST(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>TWO</td>
<td>ST(1)</td>
<td>3.0</td>
<td>BASE</td>
</tr>
<tr>
<td>3.0</td>
<td>ST(2)</td>
<td>TWO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CONTINUED IN NEXT FRAME
CLASS EXERCISE 12.1

WRITE A MATH PROGRAM THAT WILL PERFORM THE FOLLOWING OPERATION:

RESULT = ((A + B)/C)*D

DEFINE A, B, C AND D AS CONSTANTS USING THE SHORT REAL DATA TYPE. DEFINE RESULT AS A LONG REAL.

USE VALUES OF YOUR OWN CHOICE WHEN SETTING UP THE CONSTANTS.
TRANSCENDENTAL AND CONSTANT INSTRUCTIONS

<table>
<thead>
<tr>
<th></th>
<th>FPTAN</th>
<th>FPATAN</th>
<th>F2XM1</th>
<th>FYL2X</th>
<th>FYL2XP1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PARTIAL TANGENT</td>
<td>PARTIAL ARCTANGENT</td>
<td>$2^x - 1$</td>
<td>$Y \cdot \log_2 X$</td>
<td>$Y \cdot \log_2 (X + 1)$</td>
</tr>
</tbody>
</table>

- THE TRANSCENDENTAL INSTRUCTIONS PERFORM THE TIME-CONSUMING CORE CALCULATIONS OF THE FOLLOWING FUNCTIONS:
  - TRIGONOMETRIC
  - INVERSE TRIGONOMETRIC
  - HYPERBOLIC
  - INVERSE HYPERBOLIC
  - LOGARITHMIC
  - EXPONENTIAL

- IN CONJUNCTION WITH THE CONSTANT AND ARITHMETIC INSTRUCTIONS, THE TRANSCENDENTAL INSTRUCTIONS CAN BE USED TO DERIVE ALL OF THE ABOVE LISTED FUNCTIONS

EXAMPLE

FPTAN

BEFORE

| ST | 0.5236 |
| ST(1) | 0.0 |
| ST(2) | 0.0 |
| ST(3) | 0.0 |
| ST(4) | 0.0 |
| ST(5) | 0.0 |
| ST(6) | 0.0 |
| ST(7) | 0.0 |

ST(8) RADIANS (30°)

$\sqrt{x^2 + y^2}$

30°

$\tan(\theta) = \frac{y}{x}$

$\sin(\theta) = \frac{y}{\sqrt{x^2 + y^2}}$

$\cos(\theta) = \frac{x}{\sqrt{x^2 + y^2}}$

AFTER

| ST | 1.7320 |
| ST(1) | 1.0 |
| ST(2) | 0.0 |
| ST(3) | 0.0 |
| ST(4) | 0.0 |
| ST(5) | 0.0 |
| ST(6) | 0.0 |
| ST(7) | 0.0 |

$\sqrt{3}$

y

12-23
CLASS EXERCISE 12.2

WRITE A PROGRAM TO CALCULATE THE TANGENT, SINE AND COSINE OF A 60° ANGLE (π/3 RADIANS).

USE THE CONSTANT INSTRUCTIONS TO DERIVE π/3. STORE THE DESIRED RESULTS IN MEMORY USING A LONG REAL STORAGE FORMAT.

INSTRUCTION SYNCHRONIZATION

- NORMALLY, THE HOST CPU AND THE 8087 OPERATE ASYNCHRONOUSLY WITH RESPECT TO ONE ANOTHER. HOWEVER, THERE ARE TWO CASES WHEN IT IS NECESSARY TO Synchronize THE PROCESSORS.
  1) AN 8087 INSTRUCTION MUST NOT BE STARTED IF THE 8087 IS BUSY EXECUTING A PREVIOUS INSTRUCTION
  2) THE HOST CPU MUST NOT ACCESS A MEMORY OPERAND BEING REFERENCED BY THE 8087 UNTIL THE 8087 HAS COMPLETED ITS CURRENT OPERATION

- THE FWAIT INSTRUCTION ALLOWS SOFTWARE TO Synchronize THE TWO PROCESSORS, SUCH THAT THE HOST CPU WILL NOT EXECUTE ANY MORE INSTRUCTIONS UNTIL THE 8087 IS FINISHED WITH ITS CURRENT INSTRUCTION
• THE ASSEMBLER AUTOMATICALLY TAKES CARE OF THE FIRST CASE

EXAMPLE: FOR THE FOLLOWING TWO SOURCE STATEMENTS,

FMUL ; MULTIPLY
FDIV ; DIVIDE

THE ASSEMBLER PRODUCES FOUR MACHINE INSTRUCTIONS,

"FWAIT"
FMUL
"FWAIT"
FDIV

• THE FWAIT INSTRUCTIONS INSURE THAT ANY PREVIOUS 8087 INSTRUCTION RUNS TO COMPLETION, BEFORE A NEW 8087 INSTRUCTION IS STARTED

12-27

• TO SATISFY THE SECOND CASE, THE PROGRAMMER SHOULD EXPLICITLY CODE THE FWAIT INSTRUCTION IMMEDIATELY BEFORE A CPU INSTRUCTION THAT ACCESSES A MEMORY OPERAND READ OR WRITTEN BY A PREVIOUS 8087 INSTRUCTION

EXAMPLE: FIST VAR_1 ; STORE INTEGER
FWAIT ; WAIT FOR 8087
MOV AX,VAR_1

• THE FWAIT OPCODE CAUSES THE ASSEMBLER TO CREATE A CPU WAIT INSTRUCTION THAT CAN BE ELIMINATED AT LINK TIME IF THE PROGRAM IS TO RUN ON AN 8087 EMULATOR. THE WAIT OPCODE DOES NOT PROVIDE THIS FLEXIBILITY.

FWAIT - CAN BE ELIMINATED IF EMULATOR USED
WAIT - FIXED WITHIN PROGRAM. TEST PIN MUST BE IMPLEMENTED.
PROCESSOR SYNCHRONIZATION

;ASSUME 8087 REGISTER STACK IS LOADED WITH OPERANDS.
;NEU IS NOT BUSY,
;AND THAT "ALPHA" AND "BETA" ARE WORD
;INTEGERS.

FMUL ;MULTIPLY TOP STACK
;ELEMENTS
FSQRT ;SQUARE ROOT OF PRODUCT
CMP ALPHA.100 ;ALPHA 1007
JG CONTINUE ;YES, LEAVE UNALTERED
MOV ALPHA.100 ;NO, SET TO 100
CONTINUE: FIST BETA ;STORE ROOT AS INTEGER WORD
FWAIT ;WAIT FOR 8087
MOV AX,BETA ;PROCEED TO PROCESS BETA

NOTES:
• FWAIT = ASSEMBLER-GENERATED INSTRUCTION

WHERE TO FIND MORE INFORMATION

APPLICATION NOTE AP-113 - GETTING STARTED WITH THE NUMERIC DATA PROCESSOR

IAPX 86/88, 186/188 USER'S MANUAL (PROGRAMMER'S REFERENCE)
CHAPTER 6 - THE 8087 NUMERIC PROCESSOR EXTENSION

ASM86 LANGUAGE REFERENCE MANUAL
CHAPTER 6 - THE 8086/8087/8088 INSTRUCTION N SET
CHAPTER 13

MORE ON THE 8087

- STATUS WORD
- LOGICAL INSTRUCTIONS
- CONTROL WORD
- INITIALIZING THE 8087
- PROCESSOR CONTROL INSTRUCTIONS
THE REGISTER STACK

- The registers are organized as an 8 element stack
- The stack top pointer within the status word identifies the current top of stack
- The tag word identifies the contents of each register as being valid or invalid

STATUS WORD

- The status word reflects the overall condition of the 8087
- The status word may be examined by storing it into memory with an NDP instruction and then inspecting it with CPU code
**STATUS WORD (CONT)**

<table>
<thead>
<tr>
<th>B</th>
<th>C3</th>
<th>ST</th>
<th>C2</th>
<th>C1</th>
<th>IR</th>
<th>PE</th>
<th>UE</th>
<th>CE</th>
<th>DE</th>
<th>E</th>
</tr>
</thead>
</table>

- **Exception Flags**: Used to identify the type of exception(s) that have occurred since the flags were last initialized.
- **Interrupt Request**: Used to record a pending interrupt.

**Exceptions**

**Invalid Operation**
- Attempt to load a register that is not empty
- Attempt to pop an operand from a register that is empty
- Operands cause operation to be indeterminate (0/0, √-Number)

**Denormalized Operand**
- Attempt to use an operand that is not normalized

**Zero Divide**
- Attempt to divide by zero

**Overflow**
- Result too large for destination format

**Underflow**
- Result too small for destination format

**Precision**
- Result not exactly representable in destination format
- 8087 rounds result

**Note**: Exception bits are "sticky" and can be cleared only by the FCLEX (clear exceptions) instruction.
STATUS WORD (CONT)

BUSY - USED TO IDENTIFY IF THE 8087 IS EXECUTING AN INSTRUCTION

STACK TOP POINTER - USED TO IDENTIFY THE REGISTER THAT IS THE CURRENT STACK TOP

CONDITION CODE - USED TO POST RESULTS OF COMPARE/EXAMINE TYPE INSTRUCTIONS AND ALSO THE FPREM (PARTIAL REMAINDER) INSTRUCTION

COMPARISON INSTRUCTIONS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>C7</th>
<th>C6</th>
<th>C5</th>
<th>C4</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare, Test</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>A &gt; B</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>A = B</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>A = B (not comparable)</td>
</tr>
<tr>
<td>Examine</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Valid, positive, unnormalized</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Invalid, positive, exponent ≠ 0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Invalid, negative, exponent ≠ 0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Invalid, positive, normalized</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Invalid, negative, normalized</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Infinity, positive</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Infinity, negative</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Zero, positive</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Empty</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Empty</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Empty</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Invalid, positive, exponent = 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Invalid, negative, exponent = 0</td>
</tr>
</tbody>
</table>

NOTE: COMPARE INSTRUCTIONS - A = ST, B = SOURCE
TEST INSTRUCTION - A = ST, B = $\emptyset$
- The control word is used to configure the operating mode of the 8087.
- The control word is loaded from memory by the FLDCW (load control word) instruction.
INFINITY CONTROL

PROJECTIVE CLOSURE

DEFAULT - RECOMMENDED FOR MOST COMPUTATIONS

AFFINE CLOSURE

GIVES MORE INFORMATION, BUT THERE ARE OCCASIONS WHEN THE SIGN REPRESENTS MISINFORMATION

\[ \pm \infty = \pm \infty \]

\[ 1/\pm \infty = 1/\pm \infty \]

CONTROL WORD (CONT)

EXCEPTION MASKS - USED TO DETERMINE WHETHER THE 8087 SHOULD FIELD AN EXCEPTION ON ITS OWN OR SEND AN INTERRUPT REQUEST TO THE HOST CPU

NOTE: A MASKED RESPONSE PRODUCES A RESULT AND THEN PROCEEDS WITH THE CURRENT INSTRUCTION

AN UNMASKED RESPONSE TRAPS TO USER SOFTWARE BY INTERRUPTING THE HOST CPU
HOW WOULD THE AVERAGE USER CONFIGURE THE 8087?

1. USE THE DEFAULT CONFIGURATION WITH ALL EXCEPTIONS MASKED. THE 8087 WILL GENERATE A DEFAULT RESULT IF AN ERROR OCCURS.

2. UNMASK THE INVALID OPERATION EXCEPTION, AND KILL THE COMPUTATIONAL ALGORITHM IF AN INTERRUPT OCCURS.

3. UNMASK ALL THE EXCEPTIONS, AND KILL THE COMPUTATIONAL ALGORITHM IF AN INTERRUPT OCCURS.

NOTE: THE 8087 IS A VERY FLEXIBLE MATH PROCESSOR. HOWEVER, MOST OF THIS FLEXIBILITY WOULD BE USED ONLY IF VERY SERIOUS NUMERIC ANALYSIS IS REQUIRED.
INITIALIZATION

THE 8087 CAN BE INITIALIZED BY HARDWARE OR SOFTWARE

- HARDWARE INITIALIZATION (RESET)

8087 IDENTIFIES ITS HOST BY MONITORING THE BHE LINE DURING THE HOST CPU'S FIRST PROGRAM FETCH.

8086, 80186 - WORD FETCH FROM LOCATION 0FFFFFH.
BHE = 0

8088, 80188 - BYTE FETCH FROM LOCATION 0FFFFFH.
BHE (SSB) = 1

- SOFTWARE INITIALIZATION

FINT ; INITIALIZE ONLY
FSAVE ; SAVE 8087 STATE THEN INITIALIZE

8087 STATE AFTER INITIALIZATION

<table>
<thead>
<tr>
<th>FIELD</th>
<th>VALUE</th>
<th>INTERPRETATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL WORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFINITY CONTROL</td>
<td>0</td>
<td>PROJECTIVE</td>
</tr>
<tr>
<td>Rounding Control</td>
<td>00</td>
<td>ROUND TO NEAREST</td>
</tr>
<tr>
<td>Precision Control</td>
<td>11</td>
<td>64 BITS</td>
</tr>
<tr>
<td>INTERRUPT-ENABLE MASK</td>
<td>1</td>
<td>INTERRUPTS DISABLED</td>
</tr>
<tr>
<td>EXCEPTION MASKS</td>
<td>111111</td>
<td>ALL EXCEPTIONS MASKED</td>
</tr>
<tr>
<td>STATUS WORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUSY</td>
<td>0</td>
<td>NOT BUSY</td>
</tr>
<tr>
<td>CONDITION CODE</td>
<td>?? ??</td>
<td>(INDETERMINATE)</td>
</tr>
<tr>
<td>STACK TOP</td>
<td>0000</td>
<td>EMPTY STACK</td>
</tr>
<tr>
<td>INTERRUPT REQUEST</td>
<td>0</td>
<td>NO INTERRUPT</td>
</tr>
<tr>
<td>EXCEPTION FLAGS</td>
<td>000000</td>
<td>NO EXCEPTIONS</td>
</tr>
<tr>
<td>TAG WORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAGS</td>
<td>11</td>
<td>EMPTY</td>
</tr>
<tr>
<td>REGISTERS</td>
<td>N.C.</td>
<td>NOT CHANGED</td>
</tr>
<tr>
<td>EXCEPTION POINTERS</td>
<td>N.C.</td>
<td>NOT CHANGED</td>
</tr>
<tr>
<td>INSTRUCTION CODE</td>
<td>N.C.</td>
<td>NOT CHANGED</td>
</tr>
<tr>
<td>INSTRUCTION ADDRESS</td>
<td>N.C.</td>
<td>NOT CHANGED</td>
</tr>
<tr>
<td>OPERAND ADDRESS</td>
<td>N.C.</td>
<td>NOT CHANGED</td>
</tr>
</tbody>
</table>
PROCESSOR CONTROL INSTRUCTIONS

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINIT/FNINIT</td>
<td>INITIALIZE PROCESSOR</td>
</tr>
<tr>
<td>FDIS/FNDSI</td>
<td>DISABLE INTERRUPTS</td>
</tr>
<tr>
<td>FENI/FNENI</td>
<td>ENABLE INTERRUPTS</td>
</tr>
<tr>
<td>FLD CW</td>
<td>LOAD CONTROL WORD</td>
</tr>
<tr>
<td>FSTCW/FNSTCW</td>
<td>STORE CONTROL WORD</td>
</tr>
<tr>
<td>FSTSW/FNSTSW</td>
<td>STORE STATUS WORD</td>
</tr>
<tr>
<td>FCLEX/FNCLEX</td>
<td>CLEAR EXCEPTIONS</td>
</tr>
<tr>
<td>FSTENV/FNSTENV</td>
<td>STORE ENVIRONMENT</td>
</tr>
<tr>
<td>FLDCW</td>
<td>LOAD ENVIRONMENT</td>
</tr>
<tr>
<td>FSAVE/FNSAVE</td>
<td>SAVE STATE</td>
</tr>
<tr>
<td>FRSTOR</td>
<td>RESTORE STATE</td>
</tr>
<tr>
<td>FNSTP</td>
<td>INCREMENT STACK POINTER</td>
</tr>
<tr>
<td>FDECSTP</td>
<td>DECREMENT STACK POINTER</td>
</tr>
<tr>
<td>FFREE</td>
<td>FREE REGISTER</td>
</tr>
<tr>
<td>FNOP</td>
<td>NO OPERATION</td>
</tr>
<tr>
<td>FWAIT</td>
<td>CPU WAIT</td>
</tr>
</tbody>
</table>

- The opcodes, distinguished by a second character of "n", instruct the assembler not to prefix the instruction with a CPU wait instruction. Instead, a CPU NOP is used.

13-15

MEMORY REQUIREMENTS FOR STORING THE 8087's STATE AND ENVIRONMENT

STATE

<table>
<thead>
<tr>
<th>Increasing Addresses</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>CONTROL WORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATUS WORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOP WORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSTRUCTION POINTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP CODE 15-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 15-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 14-11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 10-23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 6-24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 5-25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 1-30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 0-31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPONENT 14-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPONENT 13-11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPONENT 10-23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPONENT 9-24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST STACK ELEMENT 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST STACK ELEMENT 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST STACK ELEMENT 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ENVIRONMENT

<table>
<thead>
<tr>
<th>Increasing Addresses</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>CONTROL WORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATUS WORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOP WORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSTRUCTION POINTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP CODE 15-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 15-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 14-11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 10-23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 6-24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 5-25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 1-30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNIFICAND 0-31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPONENT 14-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPONENT 13-11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPONENT 10-23</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

13-16
WHERE TO FIND MORE INFORMATION

APPLICATION NOTE AP-113 – GETTING STARTED WITH THE NUMERIC DATA PROCESSOR

IAPX 86/88, 186/188 USER'S MANUAL (PROGRAMMER'S REFERENCE)
CHAPTER 6 – THE 8087 NUMERIC PROCESSOR EXTENSION

ASM86 LANGUAGE REFERENCE MANUAL
CHAPTER 6 – THE 8086/8087/8088 INSTRUCTION SET
CHAPTER 14

OVERVIEW OF THE 8087 SUPPORT LIBRARIES

- INTERFACE LIBRARIES
- DECIMAL CONVERSION LIBRARY
- COMMON ELEMENTARY FUNCTION LIBRARY
- ERROR HANDLER LIBRARY
8087 INTERFACE LIBRARIES

FULL EMULATOR
E8087 - EMULATOR
E8087.LIB - INTERFACE LIBRARY

PARTIAL EMULATOR (PL/M-86 ONLY)
PE8087 - PARTIAL EMULATOR
E8087.LIB - INTERFACE LIBRARY

8087 CHIP
8087.LIB - INTERFACE LIBRARY

LINKING TO LIBRARIES

(PARTIAL) EMULATOR

- LINKER REMOVES ALL FWAIT INSTRUCTIONS, INSERTS "CALLS" TO EMULATOR ROUTINES VIA SOFTWARE INTERRUPTS.
- INIT87 SETS UP INTERRUPT VECTORS TO INTERFACE TO EMULATOR

8087.LIB

- 8087 INSTRUCTIONS LEFT INTACT
- INIT87 INITIALIZES 8087, MASKS ALL EXCEPTIONS
DECIMAL CONVERSION LIBRARY

- DCON87.LIB
  - CONVERT BETWEEN DIFFERENT REAL FORMATS
  - CONVERT FROM DECIMAL STRING TO BINARY FORMAT
  - CONVERT FROM BINARY FORMAT TO DECIMAL STRING

NOTES: 1) SUPPORTS PL/M-86 MEDIUM AND LARGE MODELS.
        2) MUST USE FULL EMULATOR OR ACTUAL CHIP (8087).

COMMON ELEMENTARY FUNCTION LIBRARY

- CEL87.LIB
  - ROUNING AND TRUNCATION FUNCTIONS
  - LOGARITHMIC AND EXPONENTIAL FUNCTIONS
  - TRIGONOMETRIC AND HYPERBOLIC FUNCTIONS

NOTES: 1) SUPPORTS PL/M-86 MEDIUM AND LARGE MODELS
        2) MUST USE FULL EMULATOR OR ACTUAL CHIP (8087)
ERROR HANDLER LIBRARY

- EH87.LIB
  - CONTAINS FIVE UTILITY PROCEDURES FOR WRITING YOUR OWN EXCEPTION HANDLERS

NOTES:
1) SUPPORTS PL/M-86 MEDIUM AND LARGE MODELS
2) MUST USE FULL EMULATOR OR ACTUAL CHIP (8087)

LINKAGE EXAMPLES

- FULL EMULATOR
  - RUN LINK86 :F1:MYPROG.OBJ, E8087.LIB, E8087

- DCON87, CEL87 AND CHIP
  - RUN LINK86 :F1:MYPROG.OBJ, DCON87,LIB, CEL87,LIB, 8087.LIB

- CEL87, EH87 AND EMULATOR
  - RUN LINK86 :F1:MYPROG.OBJ, CEL87, EH87, E8087,LIB, E8087
WHERE TO FIND MORE INFORMATION

8087 SUPPORT LIBRARY REFERENCE MANUAL
DAY 4 OBJECTIVES

BY THE TIME YOU FINISH TODAY YOU WILL:

- DEFINE THE ADVANTAGES OF THE 80186/188

- USE THE ENHANCED INSTRUCTION SET OF THE 80186/188

- DEFINE THE FORMAT OF THE CHIP SELECT LINES AND THE USE OF WAIT STATES IN AN 80186. PROGRAM THE CHIP SELECT LINES TO MEET A REQUIREMENT

- DEFINE THE MODES OF OPERATION OF THE THREE TIMERS ON THE 80186 AND PROGRAM THEM TO OPERATE IN A REQUIRED MODE

- DEFINE THE OPERATIONAL MODES OF THE TWO DMA CHANNELS ON THE 80186 AND PROGRAM THEM TO OPERATE IN A REQUIRED MODE
CHAPTER 15

INTRODUCTION TO THE 186

• DESCRIPTION
• ENHANCEMENTS
• NEW INSTRUCTIONS
TYPICAL iAPX 86,88 SYSTEM

SAME SYSTEM USING THE iAPX 186, 188
TYPICAL iAPX 186, 188 COMPUTER SYSTEM

16 MHz

X1 X2

UCS
ADD.
AD15
ALE

80186

RD
WR

MC6803
BHE
SRDY
ARDY
NMI
HOLD
LCS

TMR IN 0
TMR OUT 0

5V

5V

DEN
DT/R
PCS5
A1
A2
INTO

INT1
PCS4
DRQ0

8282 OR
8283
LATCH
STB
OE

ADDRESS

8282 OR
8283
RESET

ROM

ADDRESS

PROGRAM
RAM

LOW RAM

CLOCK

00-D7
SERIAL
I/O

DISK INTERFACE
HARDWARE

TERMINAL

DISK

BHE NOT IMPLEMENTED ON iAPX 188
COMPATIBILITY WITH iAPX 86,88

- OBJECT CODE COMPATIBLE WITH THE iAPX 86,88

- LANGUAGES
  - ASM, PL/M, PASCAL AND FORTRAN INCORPORATE 188 CONTROL TO SUPPORT ENHANCED INSTRUCTION SET.

- DEVELOPMENT SYSTEMS
  - SERIES III
  - INTEGRATED INSTRUMENTATION IN-CIRCUIT EMULATION (I^2ICE)
**iAPX 186, 188 RELATIVE PERFORMANCE**

(8 MHz STANDARD CLOCK RATE)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>8086 (5MHz)</th>
<th>8086-2 (8MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV REG TO MEM</td>
<td>2.0–2.9X</td>
<td>1.2–1.8X</td>
</tr>
<tr>
<td>ADD MEM TO REG</td>
<td>2.0–2.9X</td>
<td>1.2–1.8X</td>
</tr>
<tr>
<td>MUL REG 16</td>
<td>&gt;5.4X</td>
<td>&gt;3.4X</td>
</tr>
<tr>
<td>DIV REG 16</td>
<td>&gt;6.1X</td>
<td>&gt;3.8X</td>
</tr>
<tr>
<td>MULTIPLE (4-BITS) SHIFT/ROTATE MEMORY</td>
<td>3.1–3.7X</td>
<td>1.95–2.3X</td>
</tr>
<tr>
<td>CONDITIONAL JUMP</td>
<td>1.9X</td>
<td>1.2X</td>
</tr>
<tr>
<td>BLOCK MOVE (100 BYTES)</td>
<td>3.4X</td>
<td>2.1X</td>
</tr>
</tbody>
</table>

OVERALL: 2x PERFORMANCE OF 5 MHz iAPX 86
1.3x PERFORMANCE OF 8 MHz iAPX 86

NOTE: SAME COMPARISONS APPLY TO iAPX 188 and iAPX 88

---

**iAPX 186, 188 CPU ENHANCEMENTS**

- **EFFECTIVE ADDRESS CALCULATIONS (EA)**
  - Calculation of Base + Displacement + Index
  - 3 – 6X Faster in the iAPX 186,188

- **16-BIT INTEGER MULTIPLY AND DIVIDE HARDWARE**
  - 3X The 8MHz iAPX 86,88

- **STRING MOVE**
  - 2X The 8MHz iAPX 86,88

- **TRAP ON UNUSED OP CODES**
  - Pre-defined interrupt vector

- **MULTIPLE-BIT SHIFT/ROTATE SPEED-UP**
  - 1.5 – 2.5X The 8MHz iAPX 86,88

- **NEW INSTRUCTIONS**
NEW iAPX 186, 188 INSTRUCTIONS

• SHIFT/ROTATE IMMEDIATE
  - SHIFT OR ROTATE BY AN 8-BIT UNSIGNED IMMEDIATE OPERAND

<table>
<thead>
<tr>
<th>SHL</th>
<th>ROR</th>
<th>SAR</th>
<th>RCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX, 12</td>
<td>BL, 4</td>
<td>DX, 3</td>
<td>XYZ, 2</td>
</tr>
</tbody>
</table>

• MULTIPLY IMMEDIATE (IMUL)
  - IMMEDIATE SIGNED 16-BIT MULTIPLICATION WITH 16-BIT RESULT
  - IMMEDIATE OPERAND CAN BE A 16-BIT INTEGER OR A SIGNED EXTENDED 8-BIT INTEGER
  - USEFUL WHEN PROCESSING AN ARRAY INDEX

  REG16 ← REG/MEM 16 * IMMED 8/16

  | IMUL | BX, SI, 5 | ;BX = SI * 5 |
  | IMUL | SI, -200 | ;SI = SI * -200 |
  | IMUL | DI, XYZ, 20 | ;DI = XYZ * 20 |
• **PUSH IMMEDIATE (PUSH)**
  - Pushes an immediate 16-bit value or a signed extended 8-bit value onto the stack
  
  ```
  PUSH 50 ; PLACE 50 ON THE TOP
  OF THE STACK
  ```

• **PUSH ALL/POP ALL (PUSHA/POPA)**
  - Pushes/pop all 8 general purpose registers onto/off the stack
  
  ```
  INT_SRV: PUSHA : SAVE Registers
          ;
          ;
  POPA : RESTORE Registers
  IRET
  ```

• **BLOCK I/O (INS,OUTS)**
  - Moves a string of bytes or words between memory and an I/O port
  - Synchronizing possible via ready line
  
  **INS**
  - `INSB (BYTE TRANSFER)`
  - `INSW (WORD TRANSFER)`
  - `*+/- INCR: + WHEN DF = 0 (CLD)
    - WHEN DF = 1 (STD)`

  **OUTS**
  - `OUTSB (BYTE TRANSFER)`
  - `OUTSW (WORD TRANSFER)`
  - `INCR: 1 FOR BYTE TRANSFERS
    2 FOR WORD TRANSFERS`
HIGH LEVEL LANGUAGE SUPPORT

• CHECK ARRAY BOUNDS (BOUND)
  - Checks an array index register against the array bounds which are stored in a 2 word memory block

• ENTER PROCEDURE (ENTER)
  - Saves stack frame pointers from calling procedure and sets up new stack frame for current procedure

• LEAVE PROCEDURE (LEAVE)
  - Restores caller's stack frame upon procedure exit

FORMAT OF "BOUND" INSTRUCTION

BOUND 16 BIT REGISTER, ARRAY LIMITS

DATA SEGMENT
  
  ARRAY_1 DB 100 DUP (?)
  ARRAY_1_LIMITS DW OFFSET ARRAY_1
  DW OFFSET ARRAY_1 + (SIZE ARRAY_1-1)

DATA ENDS
CODE SEGMENT

ASSUME CS:CODE, DS:DATA

BOUND BX, ARRAY_1_LIMITS
MOV AL, BX

IF BX IS OUTSIDE THE LIMITS THEN AN INTERNAL INTERRUPT OF TYPE 5 IS GENERATED.
FORMAT OF "ENTER" INSTRUCTION

1. ENTER LOCAL STORAGE, NESTING LEVEL
   \[ \text{ENTER} \ 20, \ 0 \quad \equiv \quad \text{PUSH } \text{BP} \]
   \[ \text{MOV } \text{BP}, \ \text{SP} \]

   HIGH
   \[ \begin{array}{c}
   \text{PARAMETER 1} \\
   \text{PARAMETER 2} \\
   \text{RET ADDRESS} \\
   \text{OLD BP} \\
   \text{IO WORDS FOR} \\
   \text{LOCAL STORAGE}
   \end{array} \]
   BP

   LOW
   \[ \text{SP} \]

2. LEAVE
   \[ \text{LEAVE} \quad \equiv \quad \text{MOV } \text{SP}, \ \text{BP} \]
   \[ \text{POP } \text{BP} \]

   HIGH
   \[ \begin{array}{c}
   \text{PARAMETER 1} \\
   \text{PARAMETER 2} \\
   \text{RET ADDRESS}
   \end{array} \]
   SP

   LOW
CLASS EXERCISE  15.1

USE A MULTIPLY IMMEDIATE INSTRUCTION TO MULTIPLY THE CONTENTS OF BYTE PORT 0D8H TIMES A VALUE OF -5.
OUTPUT THE RESULT TO WORD PORT 0FFFAH.

WHERE TO FIND MORE INFORMATION

IAPX 86/88, 186/188 USER'S MANUAL (PROGRAMMER'S REFERENCE)
CHAPTER 5 - IAPX 186,188 HARDWARE DESIGN OVERVIEW
CHAPTER 16

iAPX 186, 188 CONTROL BLOCK CHIP SELECT
AND WAIT STATE LOGIC

• CONTROL REGISTER BLOCK
• MEMORY CHIP SELECTS
• PERIPHERAL CHIP SELECTS
• WAIT STATE LOGIC
PERIPHERAL CONTROL

- On-chip peripherals programmed via an internal register block.
- Register block initially placed in I/O address space at a base address of 0FF00H.
- Base address can be changed using relocation register.

<table>
<thead>
<tr>
<th>Register Block</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>relocation registration</td>
<td>0FFH</td>
</tr>
<tr>
<td>DMA descriptors Channel 1</td>
<td>C0H</td>
</tr>
<tr>
<td>DMA descriptors Channel 2</td>
<td>C0H</td>
</tr>
<tr>
<td>DMA descriptors Channel 3</td>
<td>C0H</td>
</tr>
<tr>
<td>Chip select control register</td>
<td>AH</td>
</tr>
<tr>
<td>Timer 3 control register</td>
<td>BH</td>
</tr>
<tr>
<td>Timer 2 control register</td>
<td>BH</td>
</tr>
<tr>
<td>Timer 1 control register</td>
<td>BH</td>
</tr>
<tr>
<td>Interrupt controller register</td>
<td>BH</td>
</tr>
</tbody>
</table>

0FF00H → 0 (After RESET)

ACCESSING INTERNAL REGISTERS

- Registers are referenced as normal I/O ports or memory locations.
- Upon detecting any address within the register block, CPU directs access to appropriate internal register.
- Example – Read interrupt mask register (Offset = 28H).

I/O MAPPED:

```
MOV DX, 0FF28H
IN AX, DX
```

MEMORY MAPPED:

```
LEA BX, REGISTER_BLOCK
MOV AX, [BX + 28H]
```
CHANGING REGISTER BLOCK BASE ADDRESS

• BASE ADDRESS CAN BE MODIFIED USING RELOCATION REGISTER. THIS REGISTER IS FOUND IN REGISTER BLOCK AT AN OFFSET OF 0FEH.

• AFTER RESET, RELOCATION REGISTER CONTAINS 2FFFH. THIS VALUE ESTABLISHES BASE ADDRESS OF 0FF00H.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ET</td>
<td>RMX</td>
<td>X</td>
<td>M/I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RELOCATION ADDRESS BITS (R19–8)

ESC TRAP
1–ENABLED
0–DISABLED

INTERRUPT CONTROLLER MODE
1–RNX COMPATIBLE
0–NORMAL

REGISTER BLOCK LOCATED IN MEMORY OR I/O SPACE
1–MEMORY
0–I/O

USED TO ESTABLISH UPPER ORDER ADDRESS BITS (A19–A8).

A7–A0 DEFAULT TO 0.

NOTE: REGISTER BLOCK MUST BE PLACED ON EVEN 256 BYTE BOUNDARY.

iAPX 186,188 CHIP SELECT/READY GENERATION LOGIC

• PROVIDES CHIP SELECT AND WAIT STATES FOR UP TO 6 MEMORY BANKS

• PROVIDES CHIP SELECT AND WAIT STATES FOR UP TO 7 PERIPHERAL DEVICES

• 0–3 WAIT STATES CAN BE PROGRAMMED FOR EACH RANGE
CHIP SELECT/READY GENERATION BLOCK DIAGRAM

- **Upper memory CS**
  - Base address: from FFFFF down
  - Range: 1K to 256K (1K, 2K, 4K, ..., 256K)

- **Mid range memory CS**
  - Base address: 4X selected range
  - Range: from 2K to 128K
  - 4 Contiguous memory pages

- **Lower memory CS**
  - Base address: from 0 up
  - Range: 1K to 256K (1K, 2K, 4K, ..., 256K)

- **Peripheral CS**
  - Base address: any 1K byte boundary
  - Range: 128 Bytes for each peripheral

MEMORY CHIP SELECTS

- **RESET ROM**
  - Base: from $FFFFFH$ down
  - Range: 1K to 256K

- **1 BLOCK UP TO 512K/4 CONTIGUOUS PAGES**
  - Base: multiple of total block size
  - Range: 1K to 128K for each page

- **VARIABLE DATA**
  - Base: from $ß$ up
  - Range: 1K to 256K
PERIPHERAL CHIP SELECTS

- MUST KEEP I/O DEVICES ON EVEN BOUNDARIES. NOT REQUIRED WITH IAPX 188.

ALTERNATIVE APPROACH

- PCS5 AND PCS6 PROVIDE LATCHED ADDRESS BITS A1 AND A2
### READY/WAIT STATE PROGRAMMING

**READY Bits Programming**

<table>
<thead>
<tr>
<th>R2</th>
<th>R1</th>
<th>R0</th>
<th>Number of WAIT States Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 wait states, external RDY also used.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1 wait state inserted, external RDY also used.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2 wait states inserted, external RDY also used.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3 wait states inserted, external RDY also used.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 wait states, external RDY ignored.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1 wait state inserted, external RDY ignored.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2 wait states inserted, external RDY ignored.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3 wait states inserted, external RDY ignored.</td>
</tr>
</tbody>
</table>

- IMPLEMENTATION OF EXTERNAL RDY

---

### UPPER MEMORY CHIP SELECT PROGRAMMING

**UMCS REGISTER**

<table>
<thead>
<tr>
<th>Offset: A0H</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

- A19–A10 OF BASE ADDRESS
- READY LIMIT = $FFFFFH$

**UMCS Programming Values**

<table>
<thead>
<tr>
<th>Starting Address (Base Address)</th>
<th>Memory Block Size</th>
<th>UMCS Value (Assuming $R0=R1=R2=0$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFC00</td>
<td>1K</td>
<td>FFFBH</td>
</tr>
<tr>
<td>FFB00</td>
<td>2K</td>
<td>FFB0H</td>
</tr>
<tr>
<td>FFB00</td>
<td>4K</td>
<td>FFB3H</td>
</tr>
<tr>
<td>FFC00</td>
<td>8K</td>
<td>FFB0H</td>
</tr>
<tr>
<td>FFC00</td>
<td>16K</td>
<td>FC3BH</td>
</tr>
<tr>
<td>FFB00</td>
<td>32K</td>
<td>FB3BH</td>
</tr>
<tr>
<td>FFB00</td>
<td>64K</td>
<td>FDB3H</td>
</tr>
<tr>
<td>FFB00</td>
<td>128K</td>
<td>E03BH</td>
</tr>
<tr>
<td>FFB00</td>
<td>256K</td>
<td>C03BH</td>
</tr>
</tbody>
</table>

**NOTE:** AFTER RESET, THE UMCS REGISTER IS INITIALIZED TO $FFFFBH$

- BASE ADDRESS = 0FFC00H
- BLOCK SIZE = 1K
- READY MODE = 3 WAIT STATES, EXTERNAL RDY USED
LOW MEMORY CHIP SELECT PROGRAMMING

OFFSET: A2H

A19-A10 OF UPPER ADDRESS
BASE ADDRESS = 0

LMCS Programming Values

<table>
<thead>
<tr>
<th>Upper Address</th>
<th>Memory Block Size</th>
<th>LMCS Value (Assuming R0=R1=R2=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000FFH</td>
<td>1K</td>
<td>0038H</td>
</tr>
<tr>
<td>007FFH</td>
<td>2K</td>
<td>0078H</td>
</tr>
<tr>
<td>00FFFFH</td>
<td>4K</td>
<td>00F8H</td>
</tr>
<tr>
<td>01FFFFH</td>
<td>8K</td>
<td>01F8H</td>
</tr>
<tr>
<td>03FFFFH</td>
<td>16K</td>
<td>03F8H</td>
</tr>
<tr>
<td>07FFFFH</td>
<td>32K</td>
<td>07F8H</td>
</tr>
<tr>
<td>0FFFFFH</td>
<td>64K</td>
<td>0FF8H</td>
</tr>
<tr>
<td>1FFFFFH</td>
<td>128K</td>
<td>1FF8H</td>
</tr>
<tr>
<td>3FFFFFH</td>
<td>256K</td>
<td>3FF8H</td>
</tr>
</tbody>
</table>

NOTE: AFTER RESET, THE LMCS REGISTER IS UNDEFINED.
THE LCS CHIP SELECT LINE REMAINS INACTIVE UNTIL
THE LMCS REGISTER IS PROGRAMMED.

16-11

MID-RANGE MEMORY CHIP SELECT PROGRAMMING

MPCS REGISTER

OFFSET: A6H

MPICS Programming Values

<table>
<thead>
<tr>
<th>Total Block Size</th>
<th>Individual Select Size</th>
<th>M6-M0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8K</td>
<td>2K</td>
<td>00000010</td>
</tr>
<tr>
<td>16K</td>
<td>4K</td>
<td>00000011</td>
</tr>
<tr>
<td>32K</td>
<td>8K</td>
<td>00000010</td>
</tr>
<tr>
<td>64K</td>
<td>16K</td>
<td>00100000</td>
</tr>
<tr>
<td>128K</td>
<td>32K</td>
<td>00100000</td>
</tr>
<tr>
<td>256K</td>
<td>64K</td>
<td>01000000</td>
</tr>
<tr>
<td>512K</td>
<td>128K</td>
<td>10000000</td>
</tr>
</tbody>
</table>

CAUTION: ONLY ONE BIT SHOULD BE SET. OTHERWISE,
UNPREDICTABLE OPERATION OF MCS LINES WILL OCCUR.

OFFSET: A6H

MMCS REGISTER

A19-A13 OF BASE ADDRESS
MUST BE MULTIPLE OF TOTAL BLOCK SIZE.

NOTE: AFTER RESET, THE MPCS AND MMCS REGISTERS ARE UNDEFINED. THE MCS LINES REMAIN INACTIVE UNTIL BOTH THE MPCS AND MMCS REGISTERS ARE PROGRAMMED.

16-12
PERIPHERAL CHIP SELECT PROGRAMMING

PACS REGISTER

OFFSET: A4H

A19 - A10 OF BASE ADDRESS MUST BE MULTIPLE OF 1K

READY MODE SELECTION FOR PCS0 - PCS3

PCS Address Ranges

<table>
<thead>
<tr>
<th>PCS Line</th>
<th>Active between Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCS0</td>
<td>PBA - PBA + 127</td>
</tr>
<tr>
<td>PCS1</td>
<td>PBA + 128 - PBA + 255</td>
</tr>
<tr>
<td>PCS2</td>
<td>PBA + 256 - PBA + 383</td>
</tr>
<tr>
<td>PCS3</td>
<td>PBA + 384 - PBA + 511</td>
</tr>
<tr>
<td>PCS4</td>
<td>PBA + 512 - PBA + 639</td>
</tr>
<tr>
<td>PCS5</td>
<td>PBA + 640 - PBA + 767</td>
</tr>
<tr>
<td>PCS6</td>
<td>PBA + 788 - PBA + 915</td>
</tr>
</tbody>
</table>

PC5S REGIST:R

OFFSET: ABH

MID - RANGE MEMORY CONTROL

READY MODE SELECTION FOR PCS4 - PCS6

MS, EX Programming Values

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS</td>
<td>1 = Peripherals mapped into memory space.</td>
</tr>
<tr>
<td>0</td>
<td>= Peripherals mapped into I/O space.</td>
</tr>
<tr>
<td>EX</td>
<td>0 = 5 PCS lines. A1, A2 provided.</td>
</tr>
<tr>
<td></td>
<td>1 = 7 PCS lines. A1, A2 are not provided.</td>
</tr>
</tbody>
</table>

WHERE TO FIND MORE INFORMATION

IAPX 86/88, 186/188 USER'S MANUAL (PROGRAMMER'S REFERENCE)
CHAPTER 5 - IAPX 186,188 HARDWARE DESIGN OVERVIEW
CHAPTER 17

iAPX 186,188 TIMER

- DESCRIPTION
- FEATURES
- PROGRAMMING
iAPX 186,188 TIMER FEATURES

- 3 INDEPENDENT 16-BIT PROGRAMMABLE TIMER/COUNTERS (64K MAX COUNT)
- TIMERS COUNT UP
- TIMER REGISTERS MAY BE READ OR WRITTEN AT ANY TIME
- TIMERS CAN INTERRUPT ON TERMINAL COUNT VIA INTERNAL INTERRUPT CONTROLLER
- TIMERS CAN HALT OR CONTINUE ON TERMINAL COUNT
TIMER 0 AND TIMER 1 OPTIONS

- COUNT INTERNAL OR EXTERNAL PULSES

- GATE OR RETRIGGER THE TIMER

INPUT USED TO START AND STOP TIMER

GENERATE PULSE OUTPUT USING SINGLE MAX COUNT REGISTER.
MAX COUNT REGISTER A = 4
Input Clock

Pulse is one processor clock wide

GENERATE PULSE OUTPUTS OF ANY DUTY CYCLE USING BOTH MAX COUNT REGISTERS.
MAX COUNT REGISTER A = 5
MAX COUNT REGISTER B = 4

17-3

17-4
TIMER 2 OPTIONS

- CLOCK COUNTER - REAL TIME CLOCK, TIME DELAY

  CLOCK → TIMER 2 → T2 INT REQ

- PRESCALER FOR OTHER TWO TIMERS

  CLOCK → TIMER 2 → TIMER 0 → T0 INT REQ
  CLOCK → TIMER 2 → TIMER 1 → T1 INT REQ

- DMA REQUEST SOURCE - TIMED DMA TRANSFERS

  CLOCK → TIMER 2 → T2 DMA REQ

17-5

TIMER 0 AND TIMER 1 MODE/CONTROL REGISTER

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BN</td>
<td>INH</td>
<td>INT</td>
<td>RIU</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>MC</td>
<td>RTG</td>
<td>P</td>
<td>EXT</td>
<td>ALT</td>
</tr>
</tbody>
</table>

- REGISTER IN USE
  φ = MAX COUNT REG A
  1 = MAX COUNT REG B

- INTERRUPT ENABLE
  φ = DISABLED
  1 = ENABLED

- INHIBIT UPDATE OF EN BIT DURING WRITE TO MODE/CONTROL REGISTER
  φ = EN BIT UNAFFECTED
  1 = EN BIT MODIFIED

- TIMER ENABLE
  φ = DISABLED
  1 = ENABLED

- CONTINUOUS COUNT
  φ = ONE SHOT
  1 = CONTINUOUS COUNT

- ALTERNATE BETWEEN MAX COUNT REGISTERS
  φ = REG A ALWAYS USED
  1 = ALTERNATE

- EXTERNAL INPUT
  0 = INTERNAL CLOCK
  1 = EXTERNAL CLOCK

- PRESCALER MODE ENABLE
  φ = DISABLED
  1 = ENABLED

- RETRIGGER MODE ENABLE
  0 = GATED
  1 = RETRIGGERED

- MAXIMUM COUNT
  0 = MAX COUNT NOT REACHED
  1 = MAX COUNT REACHED

17-6
**TIMER 2 MODE/CONTROL REGISTER**

<table>
<thead>
<tr>
<th>Register Offset</th>
<th>Tmr. 0</th>
<th>Tmr. 1</th>
<th>Tmr. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode/Control Word</td>
<td>56H</td>
<td>5EH</td>
<td>66H</td>
</tr>
<tr>
<td>Max Count B</td>
<td>54H</td>
<td>5CH</td>
<td>not present</td>
</tr>
<tr>
<td>Max Count A</td>
<td>52H</td>
<td>A5H</td>
<td>62H</td>
</tr>
<tr>
<td>Count Register</td>
<td>50H</td>
<td>58H</td>
<td>60H</td>
</tr>
</tbody>
</table>

- THE COUNT REGISTERS CAN BE READ OR WRITTEN AT ANY TIME.
- AFTER RESET, THE FOLLOWING CONDITIONS EXIST:
  1) ALL EN BITS ARE RESET PREVENTING TIMER COUNTING
  2) ALL TIMER OUT PINS ARE HIGH
WHERE TO FIND MORE INFORMATION

iAPX 86/88, 186/188 USER’S MANUAL (PROGRAMMER’S REFERENCE)
CHAPTER 5 - iAPX 186,188 HARDWARE DESIGN OVERVIEW
CHAPTER 18

iAPX 186,188 DMA CONTROLLER

• MOTIVATION FOR DIRECT MEMORY ACCESS
• DESCRIPTION OF CONTROLLER
• FEATURES
• PROGRAMMING
WHY DIRECT MEMORY ACCESS?

- TO BRING ABOUT HIGH SPEED DATA TRANSFERS WITHIN THE SYSTEM'S MEMORY AND/OR I/O ADDRESS SPACES.

- LET'S ASSUME THAT A DISK CONTROLLER UTILIZES A 500 KHZ CLOCK. THIS MEANS THAT EACH BIT CELL ON THE DISK OCCUPIES A WINDOW 2μsec IN WIDTH. THEREFORE, ONE BYTE OF DATA IS TRANSFERRED EVERY 16 μsec.

- USING INTERRUPT DRIVEN I/O, THE INTERRUPT RESPONSE AND EXECUTION TIME MUST BE LESS THAN 16 μsec IN ORDER TO TRANSFER A BYTE TO OR FROM THE CONTROLLER.

FACTORS AFFECTING INTERRUPT RESPONSE AND EXECUTION TIME:

1) WORST CASE INSTRUCTION LENGTH (EXECUTION TIME)
2) PROCESSOR RESPONSE TO INTERRUPT
3) REGISTER
4) I/O SERVICING
5) REGISTER RESTORE
6) INTERRUPT RETURN

WILL WE MAKE IT?

18-1

DMA EXAMPLE

- DMA CONTROLLER ELIMINATES PROCESSOR "HOLD-OFF" WHEN PERFORMING TRANSFERS WITHIN THE MEMORY AND/OR I/O ADDRESS SPACES. BY DOING THIS, SYSTEM THROUGHPUT IS GREATLY ENHANCED. (0.5 TO 2.0 μsec/BYTE TRANSFERRED)
iAPX 186, 188 DMA CONTROLLER FEATURES

- TWO INDEPENDENT HIGH-SPEED CHANNELS

- SUPPORTS ALL COMBINATIONS OF TRANSFER MODES
  - MEMORY-TO-MEMORY
  - MEMORY TO-I/O
  - I/O-TO-MEMORY
  - I/O-TO-I/O
  \[ \text{TWO BUS CYCLE TRANSFER} \]

- BYTE OR WORD TRANSFERS
  - WORDS CAN BE TRANSFERRED TO/FROM ODD OR EVEN ADDRESSES

- 20-BIT SOURCE AND DESTINATION POINTER FOR EACH CHANNEL
  - CAN BE INCREMENTED/DECREMENTED INDEPENDENTLY DURING TRANSFER

- 16-BIT TRANSFER COUNTER
  - PROGRAMMABLE TERMINATE AND/OR INTERRUPT REQUEST WHEN COUNTER REACHES 0

- DMA REQUESTS CAN BE GENERATED BY TIMER 2

- 2MBYTE/SECOND MAXIMUM TRANSFER RATE
POINTER AND TRANSFER COUNT REGISTERS

• POINTER REGISTERS

UPPER 4 BITS

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 g

 Lower 16 Bits

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 g

• TRANSFER COUNT REGISTER

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 g

CHANNEL CONTROL REGISTER

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 g

- DESTINATION SOURCE

M/I0 DEC INC M/I0 DEC INC TC INT SYN P TDRC X CHG ST/ NDOCK STOP B/W

M/I0 - POINTER IN MEMORY/I0
SPACE (1/0)

INC - INCREMENT POINTER
AFTER TRANSFER
f - NO INCREMENT
1 - INCREMENT

DEC - DECREMENT POINTER
AFTER TRANSFER
f - NO DECREMENT
1 - DECREMENT

DMA TERMINATE WHEN
TRANSFER COUNT REACHES 0
f - DISABLED
1 - ENABLED

INTERUPT ON BYTE COUNT
TERMINATION
f - DISABLED
1 - ENABLED

BYTE/WORD (B/W) TRANSFER
START/STOP (1/0) CHANNEL
CHANGE/DO NOT CHANGE
ST/STOP BIT WHEN WRITING
TO CHANNEL CONTROL
REGISTER
TIMER 2 DMA REQUEST
f - DISABLED
1 - ENABLED

CHANNEL PRIORITY-
RELATIVE TO OTHER CHANNEL
f - LOW PRIORITY
1 - HIGH PRIORITY

CHANNELS ALTERNATE CYCLES IF SET TO SAME
PRIORITY.

CHANNEL SYNCHRONIZATION
f - NO SYNC
f1 - SOURCE SYNC
f - DESTINATION SYNC
1 - UNUSED

18-5

18-6
## DMA Control Block Format

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Register Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Word</td>
<td>C4H, C6H</td>
</tr>
<tr>
<td>Transfer Count</td>
<td>C8H, D8H</td>
</tr>
<tr>
<td>Destination Pointer (upper 4 bits)</td>
<td>C0H, D0H</td>
</tr>
<tr>
<td>Destination Pointer</td>
<td>C2H, D2H</td>
</tr>
<tr>
<td>Source Pointer (upper 4 bits)</td>
<td>C4H, D4H</td>
</tr>
<tr>
<td>Source Pointer</td>
<td>C0H, D0H</td>
</tr>
</tbody>
</table>

After reset, both channels are disabled by resetting their ST/STOP bits.

---

**WHERE TO FIND MORE INFORMATION**

IAPX 86/88, 186/188 User's Manual (Programmer's Reference)

Chapter 5 - IAPX 186,188 Hardware Design Overview
DAY 5 OBJECTIVES

BY THE TIME YOU FINISH TODAY YOU WILL:

• DEFINE THE OPERATIONAL MODES OF THE 80186 INTERRUPT CONTROLLER AND PROGRAM IT TO OPERATE IN A REQUIRED MODE

• SEE HOW TO USE THE LIBRARIAN (LIB86) AND THE MODULE CROSS-REFERENCER (CREF86)

• DEFINE THE ROLE OF THE 8089 I/O PROCESSOR

• DEFINE THE SOFTWARE INTERFACE BETWEEN THE 8086 AND THE 8089
CHAPTER 19

iAPX 186,188 INTERRUPT CONTROL UNIT

- DESCRIPTION
- FEATURES
- PROGRAMMING
iAPX 186, 188 INTERRUPT CONTROL UNIT BLOCK DIAGRAM

- ACCEPTS INTERRUPTS FROM INTERNAL SOURCES (DMA, TIMERS) AND FROM 5 EXTERNAL PINS (NMI + 4 INTERRUPT PINS)
- PROVIDES FULLY NESTED, SPECIAL FULLY NESTED FEATURES OF THE 8259A
- EXPANDABLE TO 128 EXTERNAL INTERRUPTS BY CASCADING MULTIPLE 8259A'S
  - IAPX 186 CAN BE CONFIGURED TO SUPPORT TWO MASTER 8259A'S
- EIGHT DISTINCT PRIORITY LEVELS
- PROGRAMMABLE PRIORITY LEVEL FOR EACH INTERRUPT SOURCE
- LEVEL OR EDGE TRIGGERED PROGRAMMABLE MODES FOR EACH EXTERNAL INTERRUPT SOURCE.
iAPX 186,188 PRE-ASSIGNED INTERRUPT TYPES

<table>
<thead>
<tr>
<th>Interrupt Name</th>
<th>Vector Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 0</td>
<td>0</td>
<td>Divide error trap</td>
</tr>
<tr>
<td>Type 1</td>
<td>1</td>
<td>Single step trap</td>
</tr>
<tr>
<td>NMI</td>
<td>2</td>
<td>Non-maskable interrupt</td>
</tr>
<tr>
<td>Type 3</td>
<td>3</td>
<td>Breakpoint trap</td>
</tr>
<tr>
<td>INTO</td>
<td>4</td>
<td>Trap on overflow</td>
</tr>
<tr>
<td>Array bounds trap</td>
<td>5</td>
<td>BOUND instruction trap</td>
</tr>
<tr>
<td>Unused op trap</td>
<td>6</td>
<td>Invalid op-code trap</td>
</tr>
<tr>
<td>ESCAPE op trap</td>
<td>7</td>
<td>Supports 8087 emulation</td>
</tr>
<tr>
<td>Timer 0</td>
<td>8</td>
<td>Internal h/w Interrupt</td>
</tr>
<tr>
<td>Timer 1</td>
<td>18</td>
<td>Internal h/w interrupt</td>
</tr>
<tr>
<td>Timer 2</td>
<td>19</td>
<td>Internal h/w Interrupt</td>
</tr>
<tr>
<td>DMA 0</td>
<td>10</td>
<td>Internal h/w Interrupt</td>
</tr>
<tr>
<td>DMA 1</td>
<td>11</td>
<td>Internal h/w Interrupt</td>
</tr>
<tr>
<td><em>Reserved</em></td>
<td>9</td>
<td><em>Reserved</em></td>
</tr>
<tr>
<td>INT0</td>
<td>12</td>
<td>External interrupt 0</td>
</tr>
<tr>
<td>INT1</td>
<td>13</td>
<td>External interrupt 1</td>
</tr>
<tr>
<td>INT2/INTA0</td>
<td>14</td>
<td>External interrupt 2</td>
</tr>
<tr>
<td>INT3/INTA1</td>
<td>15</td>
<td>External interrupt 3</td>
</tr>
</tbody>
</table>

INTERRUPT VECTORING HIGHLIGHTS

- FASTER INTERRUPT RESPONSE TIME FOR INTERNALLY GENERATED INTERRUPTS (42 CLOCKS) VS. IAPX 86 (61 CLOCKS)
  - 1.5X THE 8086

- SHORTER INTERRUPT (FUNCTION OF THE LONGEST INSTRUCTION MUL AND DIV TIMES ARE 1/3 THE 8MHz IAPX 86)
INTERRUPT CONTROL UNIT OPERATION

INTERRUPT REQUEST REGISTER looks at current requests and also any interrupts in-service. If requesting level has highest priority, it is put in-service.

19-5

- INTERRUPT REQUEST, IN-SERVICE AND MASK REGISTERS

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Interrupt request register - read only
In-service and mask registers - read and write

- EOI REGISTER

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Specific/non-specific (6/1)
End of interrupt
Interrupt vector type

EOI register - write only

19-6
• TIMER AND DMA CONTROL REGISTERS

- MASK BIT
  0 - NO MASK
  1 - MASK

- PRIORITY

• INT0 AND INT1 CONTROL REGISTERS

- SPECIAL FULLY NESTED MODE
  0 - NORMAL FULLY NESTED MODE
  1 - SPECIAL FULLY NESTED MODE

- LEVEL TRIGGER MODE
  0 - EDGE TRIGGER
  1 - LEVEL TRIGGER

- CASCADE MODE
  0 - DIRECT INPUT
  1 - CASCADED INPUT

• INT2 AND INT3 CONTROL REGISTERS

NOTE: EACH INTERRUPT SOURCE IS PROGRAMMED INDEPENDENTLY OF THE OTHERS.

CASCADE MODE

- COULD HAVE UP TO 128 PRIORITIZED INTERRUPT LEVELS BY CASCADING SLAVE UNITS INTO THE MASTER UNITS SHOWN.

- USING EXTERNAL 8259's, INTO AND/OR INT1 WOULD BE CONFIGURED IN SPECIAL FULLY NESTED MODE.
OPERATIONAL CONTROL

• PRIORITY MASK REGISTER

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PRIORITY MASK

USED TO DISABLE INTERRUPTS BELOW A SPECIFIED LEVEL.

• INTERRUPT STATUS REGISTER

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DMA HALT
- SETTING BIT HALTS ALL DMA TRANSFERS
- RESET BY IRET INSTRUCTION
- PERMITS PROMPT SERVICE OF INTERRUPT REQUEST
- AUTOMATICALLY SET BY NMI

OPERATION IN A POLLED ENVIRONMENT

• CPU MUST PERIODICALLY INTERROGATE INTERRUPT CONTROL UNIT TO DETERMINE IF THERE IS A PENDING INTERRUPT REQUEST.

• POLL AND POLL STATUS REGISTERS

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- INTERRUPT REQUEST BIT
  - NO REQUEST
  - REQUEST PENDING

VECTOR TYPE OF HIGHEST PRIORITY INTERRUPTING SOURCE

VALID ONLY WHEN INTREQ = 1

POLL REGISTER—READING THIS REGISTER WILL AUTOMATICALLY SET IN-SERVICE BIT OF HIGHEST PRIORITY PENDING INTERRUPT.

POLL STATUS REGISTER—HAS NO EFFECT ON IN-SERVICE REGISTER.
**INTERRUPT CONTROL BLOCK FORMAT**

<table>
<thead>
<tr>
<th>Register Description</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT3 CONTROL REGISTER</td>
<td>3EH</td>
</tr>
<tr>
<td>INT2 CONTROL REGISTER</td>
<td>3CH</td>
</tr>
<tr>
<td>INT1 CONTROL REGISTER</td>
<td>3AH</td>
</tr>
<tr>
<td>INT0 CONTROL REGISTER</td>
<td>38H</td>
</tr>
<tr>
<td>DMA 1 CONTROL REGISTER</td>
<td>30H</td>
</tr>
<tr>
<td>DMA 0 CONTROL REGISTER</td>
<td>34H</td>
</tr>
<tr>
<td>TIMER CONTROL REGISTER</td>
<td>32H</td>
</tr>
<tr>
<td>INTERRUPT CONTROLLER STATUS REGISTER</td>
<td>30H</td>
</tr>
<tr>
<td>INTERRUPT REQUEST REGISTER</td>
<td>2EH</td>
</tr>
<tr>
<td>IN-SERVICE REGISTER</td>
<td>2CH</td>
</tr>
<tr>
<td>PRIORITY MASK REGISTER</td>
<td>2AH</td>
</tr>
<tr>
<td>MASK REGISTER</td>
<td>28H</td>
</tr>
<tr>
<td>POLL STATUS REGISTER</td>
<td>26H</td>
</tr>
<tr>
<td>POLL REGISTER</td>
<td>24H</td>
</tr>
<tr>
<td>EOI REGISTER</td>
<td>22H</td>
</tr>
</tbody>
</table>

- After reset, all interrupts are disabled.

**WHERE TO FIND MORE INFORMATION**

IAPX 86/88, 186/188 User's Manual (Programmer's Reference)

Chapter 5 - IAPX 186,188 Hardware Design Overview
CHAPTER 20

LIBRARIES & MODULE CROSS-REFERENCES

- LIBRARY CHARACTERISTICS
- LIBRARY COMMANDS
- USING LIBRARIES
- INTER-MODULE CROSS REFERENCING (CREF86)
ISIS-II LIBRARIAN

A COLLECTION OF OBJECT MODULES SUPPLIED BY SYSTEM USER
OR BY INTEL

A SPECIAL FILE CONTAINING A DIRECTORY OF PUBLICS

ALLOWS SELECTION OF JUST THOSE MODULES NEEDED BY
THE PROGRAM BY LINKING TO LIBRARY
LINKING AS PROGRAM WITHOUT A LIBRARY

OVCONT.OBJ

OVENCONTROLBLOCK:
WRITE  EXTERNAL
READTEMP  EXTERNAL

UTIL.OBJ

UTILBLOCK:
DELAY  EXTERNAL
READTEMP  PUBLIC

DELAY.OBJ

DELAYBLOCK:
DELAY  PUBLIC

CONSOL.LNK

CONSOLIBLOCK:
READ  PUBLIC
WRITE  PUBLIC
CI  PUBLIC
CO  PUBLIC

- RUN LINK86 OVCONT.OBJ &
  UTIL.OBJ &
  DELAY.OBJ &
  CONSOL.LNK TO PROCES.LNK

• ENTIRE MODULE CONSOL.LNK IS INCLUDED EVEN THOUGH PUBLIC PROCEDURES 'READ' AND 'CI' ARE NEVER USED
• INSTEAD OF LINKING OBJECT MODULES INTO CONSOL.LNK, PUT THEM IN A LIBRARY...

20-3

LINKING A PROGRAM WITH A LIBRARY

OVCONT.OBJ

OVENCONTROLBLOCK:
WRITE  EXTERNAL
READTEMP  EXTERNAL

UTIL.OBJ

UTILBLOCK:
DELAY  EXTERNAL
READTEMP  PUBLIC

DELAY.OBJ

DELAYBLOCK:
DELAY  PUBLIC

CONSOL.LIB

READ_MODULE
READ  PUBLIC
CI  EXTERNAL
CO  EXTERNAL

WRITE_MODULE
WRITE  PUBLIC
CO  EXTERNAL

CI-MODULE
CI  PUBLIC

CO-MODULE
CO  PUBLIC

- RUN LINK86 OVCONT.OBJ &
  UTIL.OBJ &
  DELAY.OBJ &
  CONSOL.LIB TO PROCES.LNK

• ONLY INCLUDES LIBRARY MODULES REQUIRED TO SATISFY EXTERNAL REFERENCES

20-4
ISIS-II LIB86 COMMAND

- RUN LIB86
  
  * NO PARAMETERS ARE ALLOWED IN THE INVOCATION. LIB86 RESPONDS WITH AN ASTERISK AND WAITS FOR COMMANDS:

  CREATE  - CREATE A NEW LIBRARY
  ADD     - ADD OBJECT MODULES TO A LIBRARY
  DELETE  - DELETE OBJECT MODULES FROM A LIBRARY
  LIST    - LIST THE CONTENTS OF A LIBRARY
  EXIT    - EXIT LIBRARIAN

USING LIB86 COMMANDS

- RUN LIB86
  * CREATE CONSOL.LIB
  * ADD READ.OBJ, WRITE.OBJ, OTHER.LIB (CI, CO) TO CONSOL.LIB
  * DELETE OTHER.LIB (CI, CO)
  * LIST CONSOL.LIB PUBLICS

CONSOL.LIB

  READ_MODULE
  READ
  WRITE_MODULE
  WRITE
  CI_MODULE
  CI
  CO_MODULE
  CO

  * LIST CONSOL.LIB TO :LP: PUBLICS
CLASS EXERCISE

1. THE LIBRARY CAN CONTAIN OBJECT MODULES, LINKED MODULES AND LOCATED MODULES. THE LIBRARY LISTING SHOWS THESE ENTRIES BY MODULE NAME. WHERE DOES THIS MODULE COME FROM?

   a) FOR AN ASSEMBLED OBJECT MODULE
   b) FOR A LINKED MODULE
   c) FOR A LOCATED MODULE

2. WHAT ADVANTAGE MIGHT BE HAD BY HAVING A LARGE NUMBER OF LIBRARY MODULES EACH WITH ONE PUBLIC SYMBOL ONLY, RATHER THAN A FEW LIBRARY MODULES EACH WITH SEVERAL PUBLIC SYMBOLS.

CREF86

- PROVIDES A CROSS REFERENCE LIST OF PUBLICS AND EXTERNALS USED BY MODULES OF A PROGRAM
- TYPE CHECKING OF PUBLICS/EXTERNALS
- TYPICALLY USES SAME INPUT LIST AS YOUR FINAL LINK
EXAMPLE: CROSS-REFERENCE LISTING
WHERE TO FIND MORE INFORMATION

IAPX 86,88 FAMILY UTILITIES USER'S GUIDE
CHAPTER 21

OVERVIEW OF THE 8089 I/O PROCESSOR

• MOTIVATION FOR USING THE 8089
• PRODUCT DESCRIPTION
• INTERFACING WITH THE 8089
• PRODUCT FEATURES
• DEVELOPMENT SUPPORT
8089 I/O PROCESSOR (IOP)

TYPICAL SYSTEM WITH I/O

- CPU EXECUTES APPLICATION AND I/O PROGRAMS
- I/O RESIDES ON SYSTEM BUS
- HIGH SPEED I/O IMPEDES APPLICATION PROGRAM EXECUTION
ANOTHER SYSTEM WITH I/O

- MULTIPLE CHIP SOLUTION
- USUALLY A NONSTANDARD COMMUNICATION INTERFACE WITH HOST CPU
- DMA FACILITIES ARE NOT FLEXIBLE

SYSTEM WITH I/O PROCESSOR

- I/O CONTROL FUNCTIONS INTEGRATED WITH DMA FACILITIES
- FASTER RESPONSE
- FLEXIBLE DMA FACILITIES
8089 CONTAINS 2 INDEPENDENT I/O CHANNELS

- 2 INDEPENDENT I/O CHANNELS
- 2 REGISTER SETS,
  2 INSTRUCTION POINTERS
- 2 LOGICAL BUSES
- 2 I/O PROGRAMS CAN EXECUTE CONCURRENTLY
- I/O PROGRAMS CAN BE LOCATED IN I/O OR SYSTEM SPACE

8089 BLOCK DIAGRAM

- COMMON CONTROL UNIT
- ALU
- ASSEMBLY/DISASSEMBLY REGISTERS
- BUS INTERFACE UNIT
- REGISTERS
  TASK POINTER
  I/O CONTROL
- REGISTERS
  TASK POINTER
  I/O CONTROL
- READY
- LOCK
- DATA BUS
- ADDRESS BUS
- MUXED ADDRESS/DATA BUS
- READY
- LOCK
- DATA BUS
- ADDRESS BUS
- MUXED ADDRESS/DATA BUS
**LOCAL CONFIGURATION**

- IOP SHARES THE SYSTEM BUS INTERFACE LOGIC WITH THE HOST CPU

---

**REMOTE CONFIGURATION**

- REMOTE CONFIGURATION ALLOWS PARALLEL PROCESSING
CPU/IOP COMMUNICATION

- CPU can write to a port mapped to Multibus. Port is on 8089 board with two pins connected to Attention/Select.

INITIALIZATION OF CHANNEL CONTROL BLOCK

- Upon seeing first Channel Attention after reset, 8089 initializes itself.

SYSTEM CONFIGURATION POINTER

SYSBUS:
8/16 BIT (8088/8086)

SYSTEM OPERATION COMMAND:

a) I/O BUS WIDTH

b) REQUEST/GRANT MODE

- UPON SEEING FIRST CHANNEL ATTENTION AFTER RESET, 8089 INITIALIZES ITSELF
MESSAGE STRUCTURE

CHANNEL COMMAND WORD (CCW) SAYS WHAT TO DO (eg START, SUSPEND, ENABLE INTERRUPTS)

TYPICAL TASK FLOW
(DISK EXAMPLE)

IOP PROCESSES IN PARALLEL, ENTERS DMA MODE, THEN RETURNS TO PROCESSING INSTRUCTIONS
INSTRUCTION SET OPTIMIZED FOR I/O PROCESSING AND CONTROL

- TAILORED SPECIFICALLY FOR I/O OPERATIONS
  - LOGIC INSTRUCTIONS (MASKING)
  - BIT MANIPULATIONS, BRANCHING
  - ELEMENTARY ARITHMETICS
  - GENERALIZED MOVE
- CONTROL TRANSFERS
  - BRANCH RELATIVE
  - PROCEDURAL CALL/RETURN
- VERSATILE ADDRESSING MODES OPERATE ON 8 OR 16 BIT DATA
  - BASED
  - BASED RELATIVE
  - BASED INDEXED
  - BASED INDEXED WITH AUTO INCREMENT
- DMA CONTROL
  - SYSTEM AND I/O BUS WIDTH SPECIFICATION
  - DMA ACTIVATION

DMA FACILITIES

- TWO CYCLE TRANSFER
  1) READ → IOP
  2) IOP → WRITE

- FLEXIBLE BUS MAPPING
  - 8 BIT TO 8 BIT
  - 16 BIT TO 16 BIT
  - 8 BIT TO 16 BIT
  - 16 BIT TO 8 BIT

- FLEXIBLE I/O DEVICE SYNCHRONIZATION
  - SOURCE
  - DESTINATION
DMA FACILITIES (CONT.)

- FLEXIBLE TRANSFER CAPABILITIES
  - MEMORY TO I/O
  - I/O TO MEMORY
  - MEMORY TO MEMORY
  - I/O TO I/O

- PERFORM MASKED COMPARSES FOR DATA PATTERN AS TRANSFER OCCURS
  - 8 BIT MASK, 8 BIT COMPARE

- TRANSLATE DURING TRANSFER
  - BYTES TRANSLATED THROUGH 256-BYTE LOOKUP TABLE

- FLEXIBLE TERMINATION CONDITIONS
  - BYTE COUNT EXPIRED
  - MASKED COMPARE PASSES OR FAILS
  - SINGLE BYTE
  - EXTERNAL SOURCE

8089 PERFORMANCE

<table>
<thead>
<tr>
<th></th>
<th>5 MHz</th>
<th>8 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA TRANSFER (16-BIT TRANSFERS)</td>
<td>1.25 Mbyte</td>
<td>2.0 Mbyte</td>
</tr>
<tr>
<td>DMA BYTE SEARCH</td>
<td>0.6125/0.833 Mbyte</td>
<td>1.0/1.33 Mbyte</td>
</tr>
<tr>
<td>DMA BYTE TRANSLATE</td>
<td>0.333 Mbyte</td>
<td>0.533 Mbyte</td>
</tr>
<tr>
<td>DMA BYTE SEARCH AND TRANSLATE</td>
<td>0.333 Mbyte</td>
<td>0.533 Mbyte</td>
</tr>
<tr>
<td>DMA RESPONSE (LATENCY)</td>
<td>1.0/2.2μs</td>
<td>0.6251/1.375μs</td>
</tr>
<tr>
<td>SINGLE CHANNEL/DUAL CHANNEL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DEVELOPMENT SUPPORT

- ASM89 - 8089 MACRO ASSEMBLER
- LINK86
- LOC86
- LIB86
- RBF89 - REAL TIME BREAKPOINT FACILITY
  SOFTWARE DRIVER THAT USES
  EXISTING ICE 86,88 HARDWARE

WHERE TO FIND MORE INFORMATION

IAPX 86,88 FAMILY UTILITIES USER'S GUIDE
CHAPTER 7 - THE 8089 INPUT/OUTPUT PROCESSOR
APPENDIX A

LAB PROJECTS
INTRODUCTION TO LAB EXERCISES

The lab exercises you will be doing this week build on each other. You will be using your solution to today’s lab to implement tomorrow’s lab exercise. These labs are self-paced. You are required to complete a minimum part of today’s exercise to enable you to continue tomorrow. The minimum requirement is marked in each of the exercises.

If you finish the required part of the lab and have time to spare you can then continue with the optional parts of the lab. In cases where there are several options you should choose the option which is of most interest to you rather than follow the options in the order in which they have been written. These options offer greater detail of subject matter and give examples of the less-often used aspects of the assembly language.

Each day’s exercise starts with a description of what you will achieve, then suggests a number of steps to follow in order to complete the exercise. You will have to refer to the ASM86 language reference manual from time to time to find details of how to use instructions. If you have any trouble using this manual, ask your instructor for guidance. Use :F1: for all of your programs. If you are using floppy disks, don’t IDISK the user disk since it contains useful code! If you are working on a network (NDSII), your instructor will tell you how to drive it. Finally, you can use DEBUG (the series III debugger) to debug your code. If you do not know how to use this, ask your instructor for guidance. Good luck !!!

LAB: INTEL TEXT/MATH PROCESSOR

The lab exercise, which you will build on day by day implements a text and math processor. On day one you will write the code to select one of a number of processes (listed below). As the week progresses and you learn more about the capabilities of ASM86, the 8087 and the 80186 you can implement the options offered. You will also be linking your code to a high level language (PL/M).

The processing options are ...

<table>
<thead>
<tr>
<th>OPTION</th>
<th>NAME</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ASCII MATH</td>
<td>Implements math functions on ASCII strings</td>
</tr>
<tr>
<td>2</td>
<td>REAL MATH</td>
<td>Implements real number mathematics</td>
</tr>
<tr>
<td>3</td>
<td>SALARY BOOSTER</td>
<td>186 exercise to award salary increases</td>
</tr>
<tr>
<td>4</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>
INTRODUCTION

In today's lab you will write a program to select one of the processes offered by the text/math processor. You will prompt for a number which will be read from the keyboard and be used to select the appropriate procedure using a branch table. Today all these procedures will do is print a message to indicate that they were properly selected. You will write the options in full in later lab exercises.

PART 1

Your first task is to write a procedure which will print text to the screen (call it PRINT). You should pass the procedure the offset of the message. Use the external (far) procedure CHARACTER_OUT which is provided on your system directory in the file CICOL.OBJ. This procedure expects you to give it a single character passed as a word on the stack. CHARACTER_OUT outputs the character to the screen and will remove it from the stack on return. Terminate your text string with OFFH. Presume that CHARACTER_OUT will destroy all registers except BP and DS. Use LODS to read the character string from memory.

The format of the message string will be, for example ...

CR EQU ODH
LF EQU OAH

MESSAGE DB 'WELCOME TO THE REAL NUMBERS PROCESSOR',CR,LF
    DB 'This procedure not yet written!',CR,LF,OFFH

When you have written the procedure PRINT, write a short program to call it and print a message. Don't forget to set up a stack! Assemble your program and link it like this ....

RUN LINK86 :F1:<YOUR_PROGRAM>.OBJ,CICOL.OBJ,LARGE.LIB BIND

PART 2

In your main program, use your PRINT procedure to prompt for an input number. Read the number in using the external (far) procedure CHARACTER_IN which will return a character entered at the keyboard in AL. Use this index number to implement a branch table (ie use an indirect call of the form CALL TABLE[SI] to call the selected option, where TABLE is a table of procedure addresses). Each procedure in the branch table should print a sign-on message (using your PRINT procedure) to indicate that it has been successfully selected. Check that the number entered was not overrange (don't forget that the number will be in ASCII). The last procedure will print 'Oh dear, I selected an invalid process', or similar rebuke of your choice when an
overrange selection is entered.

** THIS IS AS FAR AS YOU NEED TO GO TO BE ABLE TO CONTINUE TOMORROW **

PART 3: use of Ascii Adjust instructions (OPTIONAL)

One of the procedures is an ascii maths processor. To start with, have it prompt for (ie print 'enter a number > ') and input two ascii digit strings using CHARACTER_IN. Add these ascii strings (use AAA) and print the result. If you have time, have the procedure prompt for a function (+, -, *, /) and implement that function using the appropriate AA- instructions. Do not start the multiply and divide options unless you have a lot of time remaining.
Day 2 | Linkage with PL/M

PART 1

Yesterday you wrote a program which would write a text string. Today you
have been supplied with an executive PL/M program which will link into your
program and use it. Small model of PL/M has been used to compile it. The PL/M
executive module will first print a message on your screen by calling your
PRINT routine, and will then run your program. The PL/M call to your procedure
looks like this ....

CALL PRINT ( @MESSAGE)

Edit your PRINT procedure to make it PL/M compatible. In doing so, you
should remember the following points ...

1/ All of your data segments and your stack segment should be added to
dGROUP.
2/ All of your code segments should be added to CGROUP.

3/ Your segment registers should be assumed to be pointing to group
bases rather than segment bases. Since the PL/M executive module will load
the segment registers, you should not be loading them in your code.

4/ Since in SMALL model of PL/M all data is in a single group (enabling all
pointers to be just a 16 bit offset), your message strings must also
be in the data group. If your messages are defined in your code segment,
use a block move in AEDIT to move them into your data segment which you will
add to the data group.

5/ Since PL/M now provides the main module, you will not need to
specify the program start address in you END statement

6/ Use the correct type of external procedures (near/far) for SMALL
model and make sure that your PUBLIC procedure PRINT is of the correct type
(near/far).

The points listed above if not noted will stop your program from working.
In addition to this you should use the most appropriate combine type, align
type and class names for each of your segments. PL/M dictates these.

Define a structure to describe the stack frame used by your PRINT procedure
and use this structure to access parameters on the stack. When PL/M has
welcomed you it will call your program which should now be made into a
procedure called ENTIRE_PROGRAM.

In order to link your program ..

RUN LINK86 EXECS.OBJ,<YOUR_PROGRAM>.OBJ,CICOS.OBJ,SMALL.LIB BIND &
TO :Fi:MAINS BIND

Take a look at the link map.
PART 2

Now edit your program so that you can link it to the large model of PL/M. This time use LDS to read @MESSAGE (a far pointer in large model PL/M) from the stack frame. Has the stack frame changed much? Edit your stack frame structure to reflect the changes. You will have to change your program quite a lot, so think carefully about near/far procedures, requirements for groups and classes. Remember that you must preserve DS, which you will destroy when you access a pointer from the stack using LDS. Also, since each module in LARGE has its own data segment, you will assume that DS addresses yours. Unlike DGROUP of SMALL model it is now your job to load DS to match the assumption if you need to use DS to access your segment. Test your edited program by linking to large model programs thus ...

RUN LINK86 EXECL.OBJ, YOUR_PROGRAM.OBJ, CICOL.OBJ, LARGE.LIB BIND & TO :Fl:MAINL BIND

Take a look at the link map.

PART 3: modular programming

You have already used external procedures CHARACTER IN and CHARACTER OUT. You are now required to write your program in modules. The rest of the programs you will be writing this week will be assembled in separate modules and linked into the main program you have written thus far. LARGE model of PL/M will be used for the rest of the week.

Use AEDIT to separate out your process procedures (they currently just indicate that the process procedure has been activated) and put them in a separate module. Call your main program :Fl:LAB2L1.ASM and the file containing the procedures :Fl:LAB2L2.ASM. AEDIT is great for this. If you don’t know how best to use it for this purpose, ask your instructor for guidance. The entries in your branch table will now be references to external routines. Will the table implement near or far calls (remember we are compatible with LARGE model PL/M)? Should the code leading up the indirect jump change? Assemble and link your component programs and check that everything still works OK.

** this is as far as you need to get to be able to continue with the lab exercise tomorrow **

PART 4: using LINK86 and LOC86 (OPTIONAL)

Use LINK86 without bind to link together the program modules you used in part 2, then locate your program according to the requirements laid out below ....

INITCODE at address F000H
CLASS 'CODE' following on after INITCODE
CLASS 'DATA' at address 200H (leaving space for interrupt vectors)
SEGMENT STACK following class 'DATA'
Take a look at the locate map to see that all is well

PART 5: Text macros (OPTIONAL)

Large model of PL/M says that each module has its own data segment. Write a macro which you can use as a header to all of your ASMB6 procedures. This macro should push BP, copy SP into BP, push DS then load DS with the data segment you are using in your module. Call it %BEGIN. Write a matching end-of-procedure macro which accepts a parameter to say how many bytes should be removed from the stack by the RET instruction. Try these out with your STRING_READ procedure.

PART 6: Records (OPTIONAL)

Use a record to represent the MODRM field of an instruction. (see ASMB6 language reference manual for the format of instructions. Ask your instructor for guidance if necessary). Using this record, construct simple instructions to replace instructions in your code previously assembled by the assembler. Start with a simple 'MOV reg,immed' and work up to a complex addressing mode. Use DEBUG to dissassemble your code to check it. If you need help, ask your instructor.
Day 3 exercises with real numbers

The PL/M executive module also had a real number math function which we could not use until now. You are going to use the 8087 emulator to provide this function. Write this exercise in a separate module which you can then link with all of the other modules you have used/written so far. You are provided with the means to input real numbers and also print them. The procedures that allow you to do this are contained in the file REAL.OBJ which you will find on your system disk. These programs are written in LARGE model PL/M.

The procedures you can use are as follows ...

READ REAL ; Prompts for input and returns the number you key in on the 8087 stack top ST. (no parameters required)

PRINT REAL ; Prints a real number on the screen. Pass the number to the procedure in ST

PRINT REAL B ; As PRINT REAL, but displays number in binary (short real format)

PART 1: using the real number procedures

First make sure that you can drive these procedures. Write a procedure with a program loop to read in a number and then display it. Display it in its binary format too (a simple number like 2.0 is easiest to understand). Don't spend too much time relating the binary format to the decimal number as you are unlikely ever to have to do this in practise. Call your program: Fl:LAB31.ASM. Make your procedure a public one and give it the same name as the dummy option you had in LAB212.OBJ. It is the second process of the text/math processor and should be linked into the rest of the processor using the command shown below.

** DON'T FORGET TO CALL INIT87 BEFORE USING THE 8087 EMULATOR ! **

SUBMIT :Fl:LAB3(1)

PART 2: some real number calculations

Now that you can input and output real numbers you are ready to do some real number calculations. Call your program: Fl:LAB32.ASM.

Have your program prompt for a number (ie print 'enter a number ...' on the screen). This number will be used as the length of a simple pendulum (expressed in metres). Calculate and display the period of the pendulum using the formula period = (2*π*sqrt(1/g)). Period is in seconds. Use FLDPI to read load the value of PI. The value of g (the acceleration due to gravity) is 9.8065. Use a long real format for this number in memory. Store the result in a short real number format. Print your result on the screen and if you have a calculator to hand, see if the result is correct.
To link your object code ... SUBMIT :F1:LAB3(2)

PART 3 : using DCON87 (OPTIONAL)

DCON87 is a useful library for helping you debug programs. It is difficult to decipher those nasty real number bit patterns and DCON87 was used to enable PRINT_REAL to print a readable decimal number on the screen. Rewrite this procedure under a different name and use your procedure to print out your real results. You will need to read the 8087 SUPPORT LIBRARY REFERENCE MANUAL to find out how to do this. The routine which you will use is mqcBIN_DECLOW. This will provide you with a string of ASCII characters which you can then print out in a format of your choice using CHARACTER_OUT. You can use the submit file used above since it's link command includes DCON87.LIB.
Day 4 using the enhanced instructions set of the 80186

This lab will implement the SALARY BOOSTER option of the text/math processor. Do as much as you can in the time available. There is no requirement for you to finish the lab up to a particular point. We do not have a 186 in the development system, so the 186 instructions are going to be emulated using CODEMACROS. Don't be alarmed at the amount of code produced by your 186 instructions. When you come to debug your program, you will see that the code is for a sequence of 8086 instructions that do the same job (a lot less efficiently). To gain access to these CODEMACROS ...

$INCLUDE (E186.INC)

In this lab you will be calling a PL/M (LARGE model) program which will read a data file from disk containing employee payscale information for a new startup called 'YURE COMPANY'. It only has 7 employees right now. Before commencing you should run a program to initialise the data file on disk ...

(RUN) SCALE.

This will write the file :F1:SCALE.PAY

PART 1: awarding an increase (use of IMUL immed, PUSH immed)

You are going to write a 'friendly' program which writes a lot of messages, so before anything else write a text macro which will print a message on the screen when you invoke it ...

$MESSAGE(NAME_OF_MESSAGE)

This macro will be very similar to one you saw in class on Tuesday. Use the assembler control $NOGEN to avoid expansion of the macro in your listing. This will make your listing very readable and avoid several lines of code each time you want to type out a message on the screen.

You will be reading a data file from disk. Define a structure to match the format of this data file. It has information as follows ..

employee's first name 10 characters
employee's last name 12 characters
employee's salary maximum 65 535 pounds

Now set aside storage space for an array of seven such structures. Call a LARGE model PL/M program to fill this array. The program looks like this ...

READ_FILE: PROCEDURE (ARRAY POINTER, ARRAY LENGTH) PUBLIC;
DECLARE ARRAY POINTER POINTER,
ARRAY LENGTH BYTE;
END;

The length of the array is the number of employees, not the number of bytes in the array. Use PUSH immediate to pass this value to the procedure.
Print a message to ask which employee (0-6) is to get an increase, then use CHARACTER_IN to read in the reply (remember it will be returned in ASCII). In a similar way, ask for the percentage increase (0-9) to be awarded. Use the employee number to index into the array of structures. To locate this employee's salary index into the array by (employee number * type structure). Use IMUL immediate to calculate this index. Having located his salary in this way you can print it out by calling another PL/M procedure which will convert the number to decimal and print it on the screen ...

```
BINOUT: PROCEDURE (NUMBER) PUBLIC;
  DECLARE NUMBER WORD;
END;
```

An appropriate message prior to printing the number would be nice. All employees started on 10 000 pounds. Now add the required increase to the salary. Display the new salary together with an appropriate message. Also, write the salary back into the array of structures. In order to update the data file on disk, yet another (LARGE) PL/M procedure has been provided ...

```
WRITE_FILE: PROCEDURE (ARRAY_POINTER,ARRAY_LENGTH) PUBLIC;
  DECLARE ARRAY_POINTER POINTER,
       ARRAY_LENGTH BYTE;
END;
```

To link your program to everything else you have done so far ...

```
SUBMIT :Fl:LAB4
```

This link is getting large and will take a while to do, so check your program carefully before going ahead.

**PART 2: BOUND check**

Use the BOUND instruction to check that you have not exceeded the bounds of the array of structures. To do this, precede the array with a bound check of the following form ...

```
BOUND_CHECK DW WORKFORCE,(WORKFORCE + SIZE WORKFORCE)-1
WORKFORCE DW EMPLOYEE_STRUCTURE 7 DUP(<>)
```

Since you have used BOUND, you should get an interrupt of type 5 if you specify an increase for employee 7 or above. Do you?

**PART 3: PUSHA, POPA**

One of the principle uses of PUSHA will be in an interrupt service procedure. Write an interrupt service procedure for the BOUND interrupt (ask your instructor for help if you are not sure how to do this) to print out an error message. Since printing a message will destroy registers, use PUSHA and POPA to safeguard registers.
PART 4: SHIFT/ROTATE immed

You have finished the salary booster now. Return to the procedure selection routine you wrote on day 1. You had to multiply your option selection by 4 to index into a table of double words. Now use a single multiple shift instruction to do this.

PART 4: ENTER, LEAVE

These instructions are quite complex. They will typically be used by compilers rather than assembly language programmers, though you might want to use them when interfacing to a high level language. Do not attempt this part of the exercise unless you are clear about everything else so far and have a fair amount of time left to spend on the exercise.

Study the ASM86 LANGUAGE REFERENCE MANUAL until you think you understand the instructions. Enter is quite clear and is ideal for languages such as PASCAL, but is overkill for PL/M which does not copy all the old stack frame pointers down from the previous stack frame. Use the ENTER instruction with a nesting level of 0 to provide the front end of the PRINT procedure which you wrote on day 1. Use LEAVE to exit from the procedure.
APPENDIX B

LAB SOLUTIONS
LAB 1 SOLUTION

SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE LAB1
OBJECT MODULE PLACED IN :F:\:LAB1.OBJ
NO INVOCATION LINE CONTROLS

LDC OBJ
LINE SOURCE
1 +1 $TITLE ('SOLUTIONS TO IAPX86/88/186 PART II LAB EXERCISES ')
2 +1 $DEBUG
3 +1 NAME LAB1
4 +1 EXTRN CHARACTER_IN: FAR, CHARACTER_OUT: FAR
5
6
7 STACK SEGMENT STACK
0000 (100
8 DW 100 DUP (?)
???? )
000B
9 IOS LABEL WORD
10 STACK ENDS

-----
12 DATA SEGMENT
13 ;
14 ; MESSAGES ....
15 ;
000D
16 CR EQU ODH
000A
17 LF EQU OAH
00FF
18 LAST EQU OFFH ; LAST CHARACTER MARKER

-----
J000 0D
20 GREETING DB CR,LF,'WELCOME TO THE REAL NUMBERS PROCESSOR',CR,LF
0001 0A
0002 57454C4443424545
20564F20544844
20524541442044
554442455320
505245434553
4F52
0027 0D
0028 0A
0029 5668693207072
6F65696772616C
20655445592044
74207772697474
49582021
0049 0D
004A 0A
004B FF

-----
004C 0D
22 PROMPT DB CR,LF,LF,'ENTER PROCESSING OPTION ....',CR,LF,LAST
004D 0A
004E 0A
004F 41454544532050
524543455349
4E47204F504449
4F4E202E2E2E2E
005F 0D
0060 0A
0061 FF

B-1
```
006E 4F5054494F4E20 25  OPTION1M  DB  'OPTION 1 HERE !', CR, LF, LAST
  31204845524520  
  0070  0D          21
  007E  0A          21
  00/F  FF          21
0080  4F5054494F4E20 26  OPTION2M  DB  'OPTION 2 HERE !', CR, LF, LAST
  32204845524520  
  0088  0D          21
  0090  0A          21
  0091  FF          21
0092  4F5054494F4E20 27  OPTION3M  DB  'OPTION 3 HERE !', CR, LF, LAST
  33204845524520  
  00A1  0D          21
  00A2  0A          21
  00A3  FF          21
00A4  4F5054494F4E20 28  OPTION4M  DB  'OPTION 4 HERE !', CR, LF, LAST
  34204845524520  
  00B3  0D          21
  00B4  0A          21
  00B5  FF          21
00B6  594F5520524541 29  ERROR1    DB  'YOU REALLY THAT ONE UP !!!!', CR, LF, LAST
  4C4C5920534352 
  45574544205448 
  4154204F4E320  
  00C3  0D          21
  00C4  0A          21
  00C5  FF          21
00C6  594F5520524541 30  DATA ENDS
  4C4C5920534352 
  45574544205448 
  4154204F4E320
00DA  FF

-----

31  DATA ENDS

-----

33  CODE1 SEGMENT
34         ASSUME CS:CODE1, DS:DATA, SS:STACK
35
36  PRINT_STRING PROC
37         ; Procedure to print a text string. The text string will be
38         ; terminated with OFFh and a near pointer to it will be passed on
39         ; the stack
40         ;
41  0000  56 41  PUSH BP          ; SAVE OLD STACK MARKER
42  0001  8BE8 42  MOV  BP, SP   ; LOAD NEW STACK BASE POINTER
43  0003  B87004 43  MOV  SI, IBP+4  ; READ OFFSET OF STRING FROM STACK
44  0006  AC 44  LOOP: MVI8
45  0007  3EFF 45  CMP  AL, LAST  ; CHECK FOR LAST CHARACTER
46  0009  84A8 46  JE  EXIT     ; AND EXIT IF SO
47  0008  56 47  PUSH SI        ; IN CASE CHARACTER_OUT DESTROYS IT
48  000C  50 48  PUSH AX        ; PASS CHARACTER TO CHARACTER_OUT
49  000D  900000------.E 49  CALL CHARACTER_OUT  ; AND PRINT THE CHARACTER
50  0012  5E 50  POP  SI        ; RESTORE POINTER TO CHARACTER STRING
51  0013  EBF1 51  JMP  NEXT   ; REPEAT FOR NEXT CHARACTER

B-2
```
8086/8088 MACRO ASSEMBLER SOLUTIONS TO 1APX86/88/186 PART II LAB EXERCISES 12:09:41 06/01/84 PAGE 3

0015 5D 52 EXIT: POP BP
0016 802000 53 RET 2 ; RETURN AND REMOVE NEAR POINTER FROM STACK
0017 55
0019 57 OPTION1P PROC
0019 8066E00 58 LEA AX,OPTION1A ;
001A 50 59 PUSH AX ; PASS OFFSET OF POINTER TO MESSAGE
001B EB0FF 60 CALL PRINT_STRING
001C 53 61 RET
001D 62 OPTION1P ENDP
0020 64 OPTION2P PROC
0021 8066000 65 LEA AX,OPTION2A ;
0022 50 66 PUSH AX ; PASS OFFSET OF POINTER TO MESSAGE
0023 EB0FF 67 CALL PRINT_STRING
0024 53 68 RET
0025 69 OPTION2P ENDP
0028 71 OPTION3P PROC
0029 8069200 72 LEA AX,OPTION3A ;
002A 50 73 PUSH AX ; PASS OFFSET OF POINTER TO MESSAGE
002B EB0FF 74 CALL PRINT_STRING
002C 53 75 RET
002D 76 OPTION3P ENDP
0030 78 OPTION4P PROC
0031 8064A00 79 LEA AX,OPTION4A ;
0032 50 80 PUSH AX ; PASS OFFSET OF POINTER TO MESSAGE
0033 EB0FF 81 CALL PRINT_STRING
0034 53 82 RET
0035 83 OPTION4P ENDP
0038 85 ERROR PROC
0039 8068600 86 LEA AX,ERRORM ;
003A 50 87 PUSH AX ; PASS OFFSET OF POINTER TO MESSAGE
003B EB0FF 88 CALL PRINT_STRING
003C 53 89 RET
003D 90 ERROR ENDP
003E 92 BUFFER DW ERROR,OPTION1P,OPTION2P,OPTION3P,OPTION4P
0040 1900
0041 2200
0042 2800
0043 3400

0045 B8---- K 94 STAKI: MOV AX,DATA ; LOAD DS
0046 B8---- K 95 MOV DS,AX ; AS ASSUMED
0047 B8---- K 96 MOV AX,STACK ; LOAD SS
0048 B8---- K 97 MOV SS,AX ; AS ASSUMED
0049 B8---- K 98 LEA SP,TO5 ; INITIALISE STACK POINTER
004A B0263000 99
004B B8---- K 100 LEA AX,GREETING ; PASS POINTER TO
004C B8---- K 101 PUSH AX ; MESSAGE
004D EB9AFF 102 CALL PRINT_STRING ; AND PRINT IT

B-3
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0066 B06400</td>
<td>0066</td>
<td>0066</td>
<td>B06400</td>
</tr>
<tr>
<td>006A 50</td>
<td>006A</td>
<td>006A</td>
<td>50</td>
</tr>
<tr>
<td>006B EB2FF</td>
<td>006B</td>
<td>006B</td>
<td>EB2FF</td>
</tr>
<tr>
<td>006E 9A0000</td>
<td>006E</td>
<td>006E</td>
<td>9A0000</td>
</tr>
<tr>
<td>0073 2C30</td>
<td>0073</td>
<td>0073</td>
<td>2C30</td>
</tr>
<tr>
<td>0075 3C04</td>
<td>0075</td>
<td>0075</td>
<td>3C04</td>
</tr>
<tr>
<td>0077 7602</td>
<td>0077</td>
<td>0077</td>
<td>7602</td>
</tr>
<tr>
<td>0079 32C0</td>
<td>0079</td>
<td>0079</td>
<td>32C0</td>
</tr>
<tr>
<td>007e 32E4</td>
<td>007e</td>
<td>007e</td>
<td>32E4</td>
</tr>
<tr>
<td>007D D1E0</td>
<td>007D</td>
<td>007D</td>
<td>D1E0</td>
</tr>
<tr>
<td>007F BB08</td>
<td>007F</td>
<td>007F</td>
<td>BB08</td>
</tr>
<tr>
<td>0081 2EFF546</td>
<td>0081</td>
<td>0081</td>
<td>2EFF546</td>
</tr>
</tbody>
</table>

---

121 CODE1 ENDS
122
123 END START

ASSEMBLY COMPLETE, NO ERRORS FOUND
LAB 2 PART 1 SOLUTION

SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE LAB2_SMALL

OBJECT MODULE PLACED IN :F1:LAB2S.OBJ

INVOCATION LINE CONTROLS

LOC OBJ LINE SOURCE

1 +1 $TITLE ('SOLUTIONS TO IAFX86/88/186 PART II LAB EXERCISES ')
2 +1 $DEBUG
3 NAME LAB2_SMALL
4
5 EXTRN CHARACTER_OUT:NEAR,CHARACTER_IN:NEAR
6 PUBLIC PRINT_STRING,ENTIRE_PROGRAM
7
8 CGROUP GROUP CODE
9 DGROUP GROUP DATA,STACK
10
11
12 STACK SEGMENT STACK 'STACK'
0000 (100
13 DW 100 DUP (?)
14
15)
16
17
18
19
0000 20 CR EQU OAH
0000 21 LF EQU OAH
0000 22 LAST EQU OFFH ; LAST CHARACTER MARKER
23
24 GREETING DB CR,LF,'WELCOME TO THE REAL NUMBERS PROCESSOR',CR,LF
0000 25 DB 'This procedure not yet written !',CR,LF,LAST
0000 26 DB
0000 27 PROMPT DB CR,LF,LF,'ENTER PROCESSING OPTION ....',CR,LF,LAST
0000 28 DB
LOC OBJ    LINE SOURCE

4F4E202E2E2E2E 006B 0D
006C 0A 006D FF

006E 4F5054494F4E20 007D 0D 007E 0A 007F FF
3120485524520
21

0080 4F5054494F4E20 0088 0D 0090 0A 0091 FF
3220485524520
21

008B 0D 008C 0A 008D FF
0092 4F5054494F4E20 009A 0D 009B 0A 009C FF
3320485524520
21

00A1 0D 00A2 0A 00A3 FF
00A4 4F5054494F4E20 00A8 0D 00A9 0A 00AA FF
3420485524520
21

00B3 0D 00B4 0A 00B5 FF
00B6 394F5520545441 00BA 0D 00BB 0A 00BC FF
4C4C592053452
4557464205448
4154204F4E520
355020212121
00BB 0D 00BD 0A 00BE FF

DATA ENDS

CODE1 SEGMENT BYTE 'CODE'
ASSUME CS:GROUP,DS:GROUP,SS:GROUP

PRINT_STRING PROC
; Procedure to print a text string. The text string will be
; terminated with OFFh and a near pointer to it will be passed on
; the stack
;
FRAME STRUC
0000 47 OLD_BP    DW  ?
0002 48 RET_OFF   DW  ?
0004 49 STRING    DW  ?
FRAME ENDS

B-6
LOC    OBJ    LINE    SOURCE

0000  S5    52    PUSH BP    ; SAVE OLD STACK MARKER
0001 BBEC  53    MOV BP, SP    ; LOAD NEW STACK BASE POINTER
0003  IE    54    PUSH DS    ; I NEED IT FOR LOOPS
0004  BB7604 55    MOV SI, [BP].STRING    ; READ OFFSET OF STRING FROM STACK
0006 AC    56    NEXT: LODSB    ; FETCH NEXT CHARACTER
0008  JCCF   57    CMP AL, LAST    ; CHECK FOR LAST CHARACTER
000A  7408   58    JE EXIT    ; AND EXIT IF SO
000C  56    59    PUSH SI    ; IN CASE CHARACTER_OUT DESTROYS IT
000E  50    60    PUSH AX    ; PASS CHARACTER TO CHARACTER_OUT
0010  EB0000 61    CALL CHARACTER_OUT    ; AND PRINT THE CHARACTER
0012  56    62    POP SI    ; RESTORE POINTER TO CHARACTER STRING
0014 EBF3   63    JMP NEXT    ; REPEAT FOR NEXT CHARACTER
0016  IF    64    EXIT: POP DS    ; RETURN AND REMOVE NEAR POINTER FROM STACK
0018  5D    65    POP BP
001A  20200 66    RET 2

0019   FUNCTION PROC
0019 BB066600  R  71    LEA AX, OPTION1A
001B  50    72    PUSH AX    ; PASS OFFSET OF POINTER TO MESSAGE
001E EB0FF    73    CALL PRINT_STRING
0021  C3    74    RET
0022  3000  75    OPTION1 PROC
0022 BB066800  R  78    LEA AX, OPTION2A
0025  50    79    PUSH AX    ; PASS OFFSET OF POINTER TO MESSAGE
0027 EB06FF    80    CALL PRINT_STRING
002A  C3    81    RET
002B  3000  82    OPTION2 PROC
002B BB066920  R  85    LEA AX, OPTION3A
002E  50    86    PUSH AX    ; PASS OFFSET OF POINTER TO MESSAGE
0030 EB0CFF    87    CALL PRINT_STRING
0033  C3    88    RET
0034  3000  89    OPTION3 PROC
0034 BB066A00  R  92    LEA AX, OPTION4A
0037  50    93    PUSH AX    ; PASS OFFSET OF POINTER TO MESSAGE
0039 EB0C4FF    94    CALL PRINT_STRING
003C  C3    95    RET
003D  3000  96    OPTION4 PROC
003D BB066B00  R  99    LEA AX, ERROR
0040  50    100    PUSH AX    ; PASS OFFSET OF POINTER TO MESSAGE
0042  88BF    101    CALL PRINT_STRING
0045  C3    102    RET
0046  3000  103    ERROR PROC
0046 BB066C00  R  105    BTABLE DW ERROR, OPTION1P, OPTION2P, OPTION3P, OPTION4P
0048  1900
LOC  OBJ  LINE  SOURCE

004A  2200  0050  ENTIRE_PROGRAM  PROC
004C  2800  0050  8D660000  R  109  LEA  AX,GREETING  ; PASS POINTER TO
004E  3400  0054  50  110  PUSH  AX  ; MESSAGE
0050  8D6600F  111  CALL  PRINT_STRING  ; AND PRINT IT
0054  50  112
0058  8D64C00  K  113  AGAIN:  LEA  AX,PROMPT  ; PRINT
005C  50  114  PUSH  AX  ; MESSAGE
005D  EB80FF  115  CALL  PRINT_STRING  ; TO INPUT SELECTION
0060  EB0000  E  117  CALL  CHARACTER_IN  ; READ PROCESSING OPTION FROM KEYBOARD
0063  2C30  118  SUB  AL,’O’  ; REMOVE ASCII OFFSET FROM CHARACTER
0065  3C04  119
0067  7602  120  CMP  AL,(LENGTH TABLE)-1  ; TEST FOR OVERRANGE
0069  32C0  121  JBE  INRANGE
006B  32E4  122  XOR  AL,AL  ; ERROR ROUTINE IS OPTION 0
006D  01E0  123
006F  8B08  124  INRANGE:  XOR  AH,AH  ; EXTEND SELECTION NUMBER TO 16 BITS
0071  2EFF974600  R  125  SHL  AX,1  ; DOUBLE, SINCE TABLE CONTAINS WORDS
0073  2EFF975008  126  MOV  BX,AX  ; SINCE AX IS NOT AN INDEX REGISTER
0075  C3  127  CALL  BYTABLE(BX)  ; CALL TO SELECTED ROUTINE
0076  C3  128  RET
129
130  ENTIRE_PROGRAM  ENDP
131
132  CODE1  ENDS
133
134  END

ASSEMBLY COMPLETE, NO ERRORS FOUND
SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE LAB2_L_MAIN
OBJECT MODULE PLACED IN :Fl:LAB2L1.OBJ
YO INVOCATION LINE CONTROLS

LOC OBJ LIME SOURCE

1 +1 $TITLE ("SOLUTIONS TO IAPX86/80/186 PART II LAB EXERCISES ")
2 +1 $DEBUG
3
4
5 EXTIN CHARACTER_OUT: FAR, CHARACTER_IN: FAR
6 EXTRN ERROR:FAR, OPTION1P: FAR, OPTION2P: FAR, OPTION3P: FAR, OPTION4P: FAR
7 PUBLIC PRINT_STRING, ENTIRE PROGRAM

STACK SEGMENT STACK 'STACK'

DW 100 DUP (?)

STACK ENDS

DATA SEGMENT 'DATA'

DATA ENDS

CODE SEGMENT BYTE 'CODE'

ASSUME CS:CODE, DS: DATA1, SS: STACK

 messages ....

CR EQU OOH
IF EQU OAM
LAST H1L IH ; LAST CHARACTER MARKER

GREETING DB
10 THE OPTION SELECTOR !', CR, LF, LAST

PROMPT DB CR, LF, 'ENTER PROCESSING OPTION ....', CR, LF, LAST
PRINT_STRING PROC FAR

; Procedure to print a text string. The text string will be
; terminated with OFFh and a near pointer to it will be passed on
; the stack

FRAME STRUC

OLD_BP DW ?
RET_OFF DW ?
RET_BASE DW ?
STRING_DD ?

FRAME ENDS

FAR BP

MOV BP, SP ; LOAD NEW STACK BASE POINTER
PUSH DS ; I NEED IT FOR LDS
LDS SI, [BP] ; READ BASE:OFFSET OF STRING FROM STACK

NEXT: LODSB ; FETCH NEXT CHARACTER
CMP AL, AL ; CHECK FOR LAST CHARACTER
JE EXIT ; AND EXIT IF SO
PUSH SI ; IN CASE CHARACTER OUT DESTROYS IT
PUSH AX ; PASS CHARACTER TO CHARACTER_OUT
CALL CHARACTER_OUT ; AND PRINT THE CHARACTER
PUSH SI ; RESTORE POINTER TO CHARACTER STRING
JMP NEXT ; REPEAT FOR NEXT CHARACTER
EXIT: POP DS
RET 4 ; RETURN AND REMOVE NEAR POINTER FROM STACK

ENTIRE_PROGRAM PROC FAR

LEA AX, GREETING ; PASS POINTER TO GREETING
PUSH CS ; MESSAGE
CALL PRINT_STRING ; AND PRINT IT
AGAIN: LEA AX, PROMPT ; PRINT
PUSH CS ; PROMPT
PUSH AX ; MESSAGE
CALL PRINT_STRING ; TO INPUT SELECTION
CALL CHARACTER_IN ; READ PROCESSING OPTION FROM KEYBOARD
SUB AL, '0' ; REMOVE ASCII OFFSET FROM CHARACTER
CMP AL, (LENGTH STABLE) - I ; TEST FOR OVERRANGE
JBE RANGE
XOR AL, AL ; ERROR ROUTINE IS OPTION 0
LOC OBJ   LINE SOURCE

0099 32E4  83      INRANGE: XOR AH,AH ; EXTEND SELECTION NUMBER TO 16 BITS
009B D1E0  84      SHL AX,1   ; DOUBLE, SINCE TABLE CONTAINS WORDS
009D D1E0  85      SHL AX,1   ; THEN TWICE FOR DOUBLE WORDS
009F BBD8  86      ADD BX,AX ; SINCE AX IS NOT AN INDEX REGISTER
00A1 2EFF6200 R 87      CALL BTABLE(BX) ; CALL TO SELECTED ROUTINE
00A6 CB    88      RET
             89
             90      ENTIRE_PROGRAM ENDP
             91

----

92      CODE1 ENDS
93
94      END

ASSEMBLY COMPLETE, NO ERRORS FOUND

B-11
SOLUTIONS TO IAPX86/88 PART II LAB EXERCISES

LAB 2 PART 3 SOLUTION: 2 of 2

SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE LAB2_L_PROCS
OBJECT MODULE PLACED IN :F1:LAB2L2.OBJ
NO INVOCATION LINE CONTROLS

LOC OBJ | LINE | SOURCE
1 +1 $TITLE ('SOLUTIONS TO IAPX86/88 PART II LAB EXERCISES ')
2 +1 $DEBUG
3  NAME LAB2_L_PROCS
4
5 EXTRN CHARACTER_OUT:FAR,CHARACTER_IN:FAR,PRINT_STRING:FAR
6 PUBLIC ERROR,OPTION1P,OPTION2P,OPTION3P,OPTION4P
7
----
8 CODE2 SEGMENT 'CODE'
9 ASSUME CS:CODE2
10
0000 0000 CR EQU ODH
0001 000A LF EQU OAH
0002 00FF LAST EQU OFFH ; LAST CHARACTER MARKER
14
0000 4F5054494F4E 20 15 OPTION1P DB 'OPTION 1 HERE !',CR,LF,LAST
31204845524520
21
000F 00
0010 0A
0011 FF
0012 4F5054494F4E 20 16 OPTION2P DB 'OPTION 2 HERE !',CR,LF,LAST
32204845524520
21
0021 0D
0022 0A
0023 FF
0024 4F5054494F4E 20 17 OPTION3P DB 'OPTION 3 HERE !',CR,LF,LAST
33204845524520
21
0033 00
0034 0A
0035 FF
0036 4F5054494F4E 20 18 OPTION4P DB 'OPTION 4 HERE !',CR,LF,LAST
34204845524520
21
0045 0D
0046 0A
0047 FF
0048 594F5520524541 19 ERRORDB DB 'YOU REALLY SCREWED THAT ONE UP !!!!',CR,LF,LAST
4C4C5920534352
455745444205448
4154204F4E520
555020212121
006A 0D
0068 0A
006C FF
20
21 OPTION1P PROC FAR
006D 0E 22 PUSH CS ; PASS BASE OF MESSAGE STRING
006E 80060000 23 LEA AX,OPTION1P ;
<table>
<thead>
<tr>
<th>LOC OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>072 50</td>
<td>24</td>
<td>PUSH AX</td>
</tr>
<tr>
<td>0073 9A0000 E</td>
<td>25</td>
<td>CALL PRINT_STRING</td>
</tr>
<tr>
<td>0078 CB</td>
<td>26</td>
<td>RET</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>OPTION1P ENDP</td>
</tr>
<tr>
<td>0079 9A0000</td>
<td>29</td>
<td>OPTION2P PROC FAR</td>
</tr>
<tr>
<td>007A 0E</td>
<td>30</td>
<td>PUSH CS</td>
</tr>
<tr>
<td>007B 9D061200</td>
<td>31</td>
<td>LEA AX,OPTION2M</td>
</tr>
<tr>
<td>007C 50</td>
<td>32</td>
<td>PUSH AX</td>
</tr>
<tr>
<td>007D 9A0000 E</td>
<td>33</td>
<td>CALL PRINT_STRING</td>
</tr>
<tr>
<td>0084 CB</td>
<td>34</td>
<td>RET</td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>OPTION2P ENDP</td>
</tr>
<tr>
<td>0085 0E</td>
<td>37</td>
<td>OPTION3P PROC FAR</td>
</tr>
<tr>
<td>0086 9D062400</td>
<td>38</td>
<td>PUSH CS</td>
</tr>
<tr>
<td>0087 50</td>
<td>39</td>
<td>LEA AX,OPTION3M</td>
</tr>
<tr>
<td>0088 9A0000 E</td>
<td>40</td>
<td>PUSH AX</td>
</tr>
<tr>
<td>0089 CB</td>
<td>41</td>
<td>CALL PRINT_STRING</td>
</tr>
<tr>
<td>0091 0E</td>
<td>42</td>
<td>RET</td>
</tr>
<tr>
<td></td>
<td>43</td>
<td>OPTION3P ENDP</td>
</tr>
<tr>
<td>0092 8D063600</td>
<td>45</td>
<td>OPTION4P PROC FAR</td>
</tr>
<tr>
<td>0093 50</td>
<td>46</td>
<td>PUSH CS</td>
</tr>
<tr>
<td>0094 9A0000 E</td>
<td>47</td>
<td>LEA AX,OPTION4M</td>
</tr>
<tr>
<td>0095 CB</td>
<td>48</td>
<td>PUSH AX</td>
</tr>
<tr>
<td>0096 9A0000 E</td>
<td>49</td>
<td>CALL PRINT_STRING</td>
</tr>
<tr>
<td>0097 CB</td>
<td>50</td>
<td>RET</td>
</tr>
<tr>
<td>0098 9A0000 E</td>
<td>51</td>
<td>OPTION4P ENDP</td>
</tr>
<tr>
<td>0099 0E</td>
<td>53</td>
<td>ERROR PROC FAR</td>
</tr>
<tr>
<td>009A 8D064800</td>
<td>54</td>
<td>PUSH CS</td>
</tr>
<tr>
<td>009B 50</td>
<td>55</td>
<td>LEA AX,ERRORM</td>
</tr>
<tr>
<td>009C 9A0000 E</td>
<td>56</td>
<td>PUSH AX</td>
</tr>
<tr>
<td>009D CB</td>
<td>57</td>
<td>CALL PRINT_STRING</td>
</tr>
<tr>
<td>009E 9A0000 E</td>
<td>58</td>
<td>RET</td>
</tr>
<tr>
<td>009F 0E</td>
<td>59</td>
<td>ERROR ENDP</td>
</tr>
<tr>
<td></td>
<td>61</td>
<td>CODE2 ENDS</td>
</tr>
</tbody>
</table>

ASSEMBLY COMPLETE, NO ERRORS FOUND
LOC OBJ   LINE   SOURCE

3 +1  $TITLE ('SOLUTIONS TO IAPX86/88/186 PART II LAB EXERCISES')
4 +1  $DEBUG
5  NAME LAB3_PART_1
6  EXTRN CHARACTER_OUT:FAR,CHARACTER_IN:FAR,PRINT_STRING:FAR
7  EXTRN READ_REAL:FAR,PRINT_REAL:FAR,PRINT_REAL_B:FAR,INIT87:FAR
8  PUBLIC OPTION2P

9  STACK SEGMENT 'STACK'
10  DW 50 DUP (?)

11  STACK ENDS
12
13  DATA3 SEGMENT 'DATA'
14  DATA3 ENDS

16  CODE3 SEGMENT 'CODE'
17  ASSUME CS:CODE3,DS:DATA3,SS:STACK
18  ;
19  ;  MESSAGES ....
20  ;

21  CR EQU OOH
22  LF EQU OAH
23  LAST EQU OFFH ; LAST CHARACTER MARKER

25  GREET DB 'THIS IS LAB3, THE REAL NUMBERS LAB ','CR,LF,LAST

26  OPTION2P   PROC   FAR

29  LEA AX,GREET ; PRINT
30  PUSH CS ; GREETING
31  PUSH AX ; MESSAGE
32  CALL PRINT_STRING
33  CALL INIT87
34  CALL READ_REAL ; READ REAL NUMBER ONTO TOP OF 8087 STACK
35  CALL PRINT_REAL ; PRINT NUMBER CURRENTLY ON ST
36  CALL PRINT_REAL_B ; PRINT IT IN BINARY
37  RET
38  OPTION2P   ENDP   B-14
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>41</td>
<td>CODE3 ENDS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>42</td>
<td>END</td>
</tr>
<tr>
<td></td>
<td></td>
<td>43</td>
<td>END</td>
</tr>
</tbody>
</table>

ASSEMBLY COMPLETE, NO ERRORS FOUND
# SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE LAB3_PART_2

**OBJECT**: Module placed in :F1:LAB32.OBJ  
**NO INVOCATION LINE CONTROLS**

**LOC OBJ LINE SOURCE**

1 +1 $TITLE ('SOLUTIONS TO IAPX86/88/186 PART II LAB EXERCISES')
2 +1 $DEBUG
3 NAME LAB3_PART_2
4
5 EXTRN CHARACTER_OUT:FAR, CHARACTER_IN:FAR, PRINT_STRING:FAR
6 EXTRN READ_REAL:FAR, PRINT_REAL:FAR, PRINT_REAL_B:FAR, INIT87:FAR
7 PUBLIC OPTION2P
8
9 STACK SEGMENT STACK 'STACK'
0000 (50 ???)

10 DW 50 DUP (?)

----
11 STACK ENDS

----
12 DATA3 SEGMENT 'DATA'
13 DATA3 ENDS

----
14 CODE3 SEGMENT 'CODE'
15 ASSUME CS:CODE3, SS:STACK
16 ;
17 ; MESSAGES ....
18 ;

0000 54484953204953
0002 204C4142332C20 45484520524541
0005 4C204E55404245
0007 5253204C41
0009 2E

0024 0D
0025 0A
0026 FF

0027 454E544552204C
002A 454E475444204F
002D 462050454144
0030 4C554D20494E20
0033 4D45545244320
0036 2E2E2E

004D 0D
004E 0A
004F FF

0050 0D
0051 0A
0052 504552494F4420

26 PROMPT DB 'ENTER LENGTH OF PENDULUM IN METRES ...', CR, LF, LAST

0070 444E454C4543204F
0073 462050454144
0076 4C554D20494E20
0079 4D45545244320
007C 2E2E2E

009D 0D
009E 0A
009F FF

00A0 0D
00A1 0A
00A2 0A
00A3 0A
00A4 FF

27 RESULT DB CR, LF, 'PERIOD OF PENDULUM IS ', LAST

00A9 0A
00AA 0A
00AB 0A
00AC 0A
00AD 0A
00AE 0A
00AF 0A
00B0 0A
00B1 0A
00B2 0A
00B3 0A
00B4 0A
00B5 0A
00B6 0A
00B7 0A
00B8 0A
00B9 0A
00BA 0A
00BB 0A
00BC 0A
00BD 0A
00BE 0A
00BF 0A

B-16
LOC OBJ  LINK  SOURCE

0068 FF
0069 205345434F4f44 20 UNIT DB ' SECONDS',CR,LF,LAST

0071 0D
0072 0A
0073 FF

0074 05A3923A019D23 30 G DW 9.80665 ; ACCELERATION DUE TO GRAVITY

007C 00000000000000 31 TWO DW 2.0

0084 33 OPTION2P PROC FAR

0084 8D060000 35 LEA AX,GREET ; PRINT

0088 0E 36 PUSH CS ; GREETING

0089 50 37 PUSH AX ; MESSAGE

008A 9A0000------ E 38 CALL PRINT_STRING

008F 9A0000------ E 39 CALL INIT87

0094 8D062700 41 LEA AX,PROMPT ; ASK

0098 0E 42 PUSH CS ; FOR

0099 50 43 PUSH AX ; LENGTH

009A 9A0000------ E 44 CALL PRINT_STRING ; OF PENDULUM

909F 9A0000------ E 46 CALL READ_REAL ; READ REAL NUMBER ONTO TOP OF 8087 STACK

00A4 9B2EDC367400 48 FDIV G ; ST = L/G

00AA 9BD9FA 49 FSQRT ; ST = SQRT(L/G)

00AD 9BD9EB 50 FLDPI ; ST = PI, ST(1) = SQRT(L/G)

00B0 9B0EC9 51 FMUL ; ST = PI * SQRT(L/G)

00B3 9B2EDC67C00 52 FMUL TWO ; ST = 2 * PI * SQRT(L/G)

00B6 9BD65000 53 FMUL RESULT ; PRINT

00BD 0E 54 PUSH CS ; START

00BE 50 55 PUSH AX ; OF RESULT

00BF 9A0000------ E 56 CALL PRINT_STRING ; MESSAGE

00C4 9A0000------ E 57 CALL PRINT_REAL ; PRINT NUMBER CURRENTLY ON ST

00C9 8D066900 59 LEA AX,UNIT ; PRINT

00CD 0E 60 PUSH CS ; END

00CE 50 61 PUSH AX ; OF RESULT

00CF 9A0000------ E 62 CALL PRINT_STRING ; MESSAGE

00D4 CB 63 RET

66 OPTION2P ENDP

-----

68 CODE3 ENDS

69 END

ASSAMBLEY COMPLETE, NO ERRORS FOUND

B-17
LAB 4 SOLUTION

SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE LAB4
OBJECT MODULE PLACED IN :H1:LAB4.0BJ
NO INVOCATION LINE CONTROLS

LOC OBJ

LINE SOURCE

1 +1 $TITLE ('SOLUTIONS TO IAPX86/88/186 PART II LAB EXERCISES ')
2 +1 $DEBUG
3 +1 $INCLUDE (E186.INC)
1 +1 $SAVE
1 +1 $NOLIST
348 +1 $NOGEN
349
350 NAME LAB4
351
352 EXTERN CHARACTER_OUT:FAK,CHARACTER_IN:FAK,PRINT_STRING:FAK
353 EXTERN READ_REAL:FAK,PRINT_REAL:FAK,PRINT_REAL_B:FAK,INITB:FAK
354 EXTERN READ_FILE:FAK,WRITE_FILE:FAK,BINOUT:FAK
355 PUBLIC OPTIONSP
356
357 XDEFINE (MESSAGE(NAME))(LEA AX,%NAME
PUSH CS
PUSH AX
CALL PRINT_STRING)

360 STACK SEGMENT STACK 'STACK'
0000 (50)
361 DW 50 DUP (?)

362 STACK ENDS
363
364 EMPLOYEE STRUC
0000
365 FIRST_NAME DB 10 DUP (?)
000A
366 LAST_NAME DB 12 DUP (?)
0016
367 PAY DW ?

368 EMPLOYEE ENDS
369

370 DATA SEGMENT 'DATA'
371
0000 0400
372 ROUND_CHECK DW WORKFORCE,(WORKFORCE + SIZE WORKFORCE)-1
0002 A000
0004 (?)
373 WORKFORCE EMPLOYEE 7 DUP (○○)

374
375 DATA ENDS
376 EJECT

B-18
LOC OBJ  LINE  SOURCE

377
380  CODE SEGMENT 'CODE'
381  ASSUME CS:CODE,DS:DATA,SS:STACK
382  
383  ; MESSAGES ....
384  
385  ; LAST CHARACTER MARKER
386
387  INCRESSE? DB  CR,LF,' HOW MANY PERCENT INCREASE ? ',LAST
388  WHO?  DB  CR,LF,LF,' AND WHICH EMPLOYEE IS THE LUCKY RECIPIENT ?'
389  CURRENT? DB  CR,LF,' .... CURRENT SALARY IS ',LAST
390  NEW DB  '........ NEW SALARY IS ',LAST
391  CURRENCY DB  ' POUNDS',CR,LF,LAST
392  NEW DB  '........ NEW SALARY IS ',LAST
393  DB  LAST
394  
395  $EJECT
LOC OBJ  LINE  SOURCE

008C  396  OPTION3P  PROC FAR

008C 1E  397  PUSH DS  ; MY PROGRAM HAS IT'S OWN DATA SEGMENT
008D B8-----  R  400  MOV AX,DATA4  ; LOAD DS

0090 B8B8  401  MOV DS,AX  ; TO MATCH ASSUME

0092 B806400  402  LEA AX,WORKFORCE  ; PASS POINTER

0096 1E  403  PUSH DS  ; TO PAYSCALES

0097 50  404  PUSH AX  ; ARRAY

0098 5555B9E5C74602  405  PUSH LENGTH WORKFORCE  ; PASS LENGTH OF ARRAY

07005D

00A2 9A0000----- E  406  CALL READ_FILE  ; CALL PL/M PROGRAM TO READ FILE FROM DISK

00A2 407  CALL MESSAGE(INCREASE?)

00A3 408  XMESSAGE(INCREASE?)

00A4 409  XMESSAGE(WHO?)

00A5 40A  CALL CHARACTER_IN  ; READ EMPLOYEE NUMBER FROM KEYBOARD

00A5 40B  SUB AL,'0'  ; REMOVE ASCII OFFSET

00A5 40C  PUSH AX  ; AND SAVE IT ON THE STACK

00A5 40D  XMESSAGE(WHO?)

00A6 40E  CALL CHARACTER_IN  ; READ EMPLOYEE NUMBER FROM KEYBOARD

00A6 40F  CALL CHARACTER_IN  ; READ INCREASE FROM KEYBOARD

00A6 410  SUB AL,'0'  ; REMOVE ASCII OFFSET

00A6 411  PUSH AX  ; AND SAVE IT ON THE STACK

00A7 412  IMUL AX,TYPE WORKFORCE  ; AX WILL BE INDEX INTO ARRAY OF STRUCTURES

00A7 413  CALL READ_FILE  ; CALL PL/M PROGRAM TO READ FILE FROM DISK

00A7 414  CALL MESSAGE(INCREASE?)

00A8 415  CALL MESSAGE(CURRENT?)

00A8 416  CALL DISPLAY_CURRENT

00A8 417  CALL BINOUT  ; SALARY

00A9 418  CALL READ_FILE  ; CALL PL/M PROGRAM TO READ FILE FROM DISK

00A9 419  CALL MESSAGE(INCREASE?)

00A9 420  CALL MESSAGE(CURRENCY)

00AA 421  CALL BINOUT  ; SALARY

00AB 422  CALL BINOUT  ; SALARY

00AC 423  CALL BINOUT  ; SALARY

00AD 424  CALL BINOUT  ; SALARY

00AE 425  CALL BINOUT  ; SALARY

00AF 426  CALL BINOUT  ; SALARY

00B0 427  CALL BINOUT  ; SALARY

00B1 428  CALL BINOUT  ; SALARY

00B2 429  CALL BINOUT  ; SALARY

00B3 430  CALL BINOUT  ; SALARY

00B4 431  CALL BINOUT  ; SALARY

00B5 432  CALL BINOUT  ; SALARY

00B6 433  CALL BINOUT  ; SALARY

00B7 434  CALL BINOUT  ; SALARY

00B8 435  CALL BINOUT  ; SALARY

00B9 436  CALL BINOUT  ; SALARY

00BA 437  CALL BINOUT  ; SALARY

00BB 438  CALL BINOUT  ; SALARY

00BC 439  CALL BINOUT  ; SALARY

00BD 440  CALL BINOUT  ; SALARY

00BE 441  CALL BINOUT  ; SALARY

00BF 442  CALL BINOUT  ; SALARY

00C0 443  CALL BINOUT  ; SALARY

00C1 444  CALL BINOUT  ; SALARY

00C2 445  CALL BINOUT  ; SALARY

00C3 446  CALL BINOUT  ; SALARY

00C4 447  CALL BINOUT  ; SALARY

00C5 448  CALL BINOUT  ; SALARY

00C6 449  CALL BINOUT  ; SALARY

00C7 450  CALL BINOUT  ; SALARY

00C8 451  CALL BINOUT  ; SALARY

00C9 452  CALL BINOUT  ; SALARY

00CA 453  CALL BINOUT  ; SALARY

00CB 454  CALL BINOUT  ; SALARY

00CC 455  CALL BINOUT  ; SALARY

00CD 456  CALL BINOUT  ; SALARY

00CE 457  CALL BINOUT  ; SALARY

00CF 458  CALL BINOUT  ; SALARY

00D0 459  CALL BINOUT  ; SALARY

00D1 460  CALL BINOUT  ; SALARY

00D2 461  CALL BINOUT  ; SALARY

00D3 462  CALL BINOUT  ; SALARY

00D4 463  CALL BINOUT  ; SALARY

00D5 464  CALL BINOUT  ; SALARY

00D6 465  CALL BINOUT  ; SALARY

00D7 466  CALL BINOUT  ; SALARY

00D8 467  CALL BINOUT  ; SALARY

00D9 468  CALL BINOUT  ; SALARY

00DA 469  CALL BINOUT  ; SALARY

00DB 470  CALL BINOUT  ; SALARY

00DC 471  CALL BINOUT  ; SALARY

00DD 472  CALL BINOUT  ; SALARY

00DE 473  CALL BINOUT  ; SALARY

00DF 474  CALL BINOUT  ; SALARY

00E0 475  CALL BINOUT  ; SALARY

00E1 476  CALL BINOUT  ; SALARY

00E2 477  CALL BINOUT  ; SALARY

00E3 478  CALL BINOUT  ; SALARY

00E4 479  CALL BINOUT  ; SALARY

00E5 480  CALL BINOUT  ; SALARY

00E6 481  CALL BINOUT  ; SALARY

00E7 482  CALL BINOUT  ; SALARY

00E8 483  CALL BINOUT  ; SALARY

00E9 484  CALL BINOUT  ; SALARY

00EA 485  CALL BINOUT  ; SALARY

00EB 486  CALL BINOUT  ; SALARY

00EC 487  CALL BINOUT  ; SALARY

00ED 488  CALL BINOUT  ; SALARY

00EE 489  CALL BINOUT  ; SALARY

00EF 490  CALL BINOUT  ; SALARY

00F0 491  CALL BINOUT  ; SALARY

00F1 492  CALL BINOUT  ; SALARY

00F2 493  CALL BINOUT  ; SALARY

00F3 494  CALL BINOUT  ; SALARY

00F4 495  CALL BINOUT  ; SALARY

00F5 496  CALL BINOUT  ; SALARY

00F6 497  CALL BINOUT  ; SALARY

00F7 498  CALL BINOUT  ; SALARY

00F8 499  CALL BINOUT  ; SALARY

00F9 500  CALL BINOUT  ; SALARY

00FA 501  CALL BINOUT  ; SALARY

00FB 502  CALL BINOUT  ; SALARY

00FC 503  CALL BINOUT  ; SALARY

00FD 504  CALL BINOUT  ; SALARY

00FE 505  CALL BINOUT  ; SALARY

00FF 506  CALL BINOUT  ; SALARY

0100 507  CALL BINOUT  ; SALARY

0101 508  CALL BINOUT  ; SALARY

0102 509  CALL BINOUT  ; SALARY

0103 510  CALL BINOUT  ; SALARY

0104 511  CALL BINOUT  ; SALARY

0105 512  CALL BINOUT  ; SALARY

0106 513  CALL BINOUT  ; SALARY

0107 514  CALL BINOUT  ; SALARY

0108 515  CALL BINOUT  ; SALARY

0109 516  CALL BINOUT  ; SALARY

010A 517  CALL BINOUT  ; SALARY

010B 518  CALL BINOUT  ; SALARY

010C 519  CALL BINOUT  ; SALARY

010D 520  CALL BINOUT  ; SALARY

010E 521  CALL BINOUT  ; SALARY

010F 522  CALL BINOUT  ; SALARY

0110 9A0000----- E  457  CALL BINOUT  ; SALARY

0110 458  CALL BINOUT  ; SALARY

0111 59  459  POP CX  ; POP INCREASE INTO CL

0112 32D0  460  XOR CH,CH  ; AND ADD LEADING ZEROS

0113 8807  461  MOV AX,[BX]  ; FETCH CURRENT SALARY

0114 F7E1  462  MUL CX  ; MULTIPLY ORIGINAL SALARY BY INCREASE

0115 B96400  463  MOV CX,100  ; DIVIDE BY

0116 8F7F  464  DIV CX  ; 100 FOR PERCENT

B-20
LOC OBJ LINE SOURCE
0120 0107 465 ADD [BX],AX ; ADD INCREASE TO SALARY
0120 FF37 466
013C 9A0000----- E 467 PUSH WORD PTR [BX] ; NEW SALARY ON STACK FOR BINOUT
013C 9A0000----- E 468 8MESSAGE (NEW)
013C 9A0000----- E 474 CALL BINOUT ; SALARY
013C 9A0000----- E 475 8MESSAGE (CURRENCY)
014C 8D060400 482 LLA AX, WORKFORCE ; PASS POINTER
0150 1E 483 PUSH DS ; TO PAYSCALES
0151 50 484 PUSH AX ; ARRAY
0152 555589E5C74602 485 PUSH LENGTH WORKFORCE ; PASS LENGTH OF ARRAY
015C 9A0000----- E 486 CALL WRITE_FILE ; WRITE ARRAY BACK ONTO DISK
0161 1F 487
0162 CB 488 POP DS
0163 490 RET
0164 491 OPTIONS ENDP
0165 END

ASSEMBLY COMPLETE, NO ERRORS FOUND

B-21
APPENDIX C

80/88 DESIGN EXAMPLE
iAPX/186 APPLICATION EXAMPLE

In this appendix you have a diagram showing how the 186 (or 188, they look the same from a software standpoint) could be used as the basis for a small business computer. It shows how the various peripherals are connected up, both in terms of addressing and in how they utilize the 186 via interrupts and DMA. The memory address mapping is also shown. In this appendix, which you will fill in as you learn the various functions of the 186, you are going to set up the 186 to handle this computer.

Because we made the 186 versatile it has many options on how to use the interrupts, timers, DMA controllers, etc. As you work through this appendix you will appreciate that there is a lot of work to do in getting all the right bits into the right control registers. Fortunately, this is something which to a large extent you program once (for a given hardware configuration) and that's the 186 set up for your system. There are also status registers which allow you to monitor the state of the various internal peripherals and command registers for run-time control of these peripherals.
NOTES:
You will see that only those control lines which are relevant to the programming exercise have been included here. The others have been omitted for the sake of clarity.
SETTING UP REGISTER BLOCK AND CHIP SELECT LOGIC

1) Locate the register block at location 10000H in memory space, enable trapping of escape codes and set the interrupt controller into normal mode.

REG_BLOCK SEGMENT AT 10000H
TABLE LABEL WORD
; set segment aside for control register block
REG_BLOCK ENDS

CODE_1 SEGMENT
ASSUME CS:CODE_1,DS:REG_BLOCK

DEFAULT EQU OFF00H
START: MOV AX,REG_BLOCK
MOV DS,AX
MOV DX,DEFAULT+OFEH
MOV AX,1001000100000000B
OUT DX,AX

2) Program your upper memory chip select. Your memory needs 1 wait state and no external ready synchronisation is required.

MOV TABLE+0AOH,________________________ ;PROGRAM UMCS

3) Program your lower memory chip select. No wait states are required and no external ready synch is needed.

_______________________________ ;PROGRAM LMCS

4) Program the mid range chip selects. No wait states are required, nor is external ready synch needed. Leave the bits for the peripheral chips blank. We'll return to them later.

_______________________________ ;PROGRAM MMCS
_______________________________ ;PROGRAM MPCS

(... continued)
5) Now program the peripheral chip selects. Your peripherals should be I/O mapped and each requires two wait states and no external ready synch. You need address lines A1 and A2 from PCS5,6. Go back and fill in the rest of the MPCS bits from part 4.

;PROGRAM PACS
SETTING UP THE TIMERS

TIMER 0

This is being used as a straight 16 bit divider to reduce the crystal frequency to a baud rate of 9600. A square wave is required, so use both count A and count B registers. No interrupt is required on terminal count.

\[
\text{XTAL(16MHz)} \rightarrow \div 2 \rightarrow \div 4 \rightarrow \text{TIMER 0 (x1/??)} \rightarrow 9600 \text{ Hz}
\]

TIMER 2

This timer is being used as a prescaler to divide by ???? . A single count register will be used (we have no option about this), and no interrupt on terminal count is required.

\[
\text{XTAL(16MHz)} \rightarrow \div 2 \rightarrow \div 4 \rightarrow \text{TIMER 2 (x1/??)} \rightarrow 1\text{KHz}
\]

TIMER 1

A real time clock interrupt is to be generated from this timer. It is to be fed from the timer 2 prescaler and is to produce an interrupt every 1 second.

\[
\text{TIMER 2} \rightarrow 1\text{KHz} \rightarrow \text{TIMER 1 (x1/??)} \rightarrow 1\text{Hz} \rightarrow \text{interrupt}
\]

continued ....
SETTING UP THE TIMERS (continued)

On the previous sheet I explained the operating modes required of the three timers. The timer control registers appear in contiguous locations inside the control register block. It might make sense to take advantage of this fact. To this end, the basis of a block move solution for loading the registers is suggested here. It is neater than loading all of the registers one at a time using in-line code. You still have to do the nasty bit-picking for some of the registers, but at least you only have to do it once (provided that you got it right!).

CODE_1   SEGMENT
ASSUME CS:CODE_1, ES:REG_BLOCK

MOV AX, REG_BLOCK : ADDRESS REGISTER
MOV ES, AX : BLOCK WITH
LEA DI, TABLE+50H ; ES:DI (includes offset to
               ; first timer control register)

LEA SI, PROG_TIME ; ADDRESS TABLE OF BIT PATTERNS (below)
MOV CX, 12 ; COUNT OF REGISTERS TO LOAD
REP MOVSB TABLE, PROG_TIME ; ASSEMBLE WILL GIVE CS: OVERRIDE
               ; TO ACCESS PROG_TIME

etc....

PROG_TIME: DW ___________; TMR 0 COUNT REGISTER
         DW ___________; MAX COUNT A
         DW ___________; MAX COUNT B
         DW ___________; MODE/CONTROL WORD
         DW ___________; TMR 1 COUNT REGISTER
         DW ___________; MAX COUNT A
         DW ___________; MAX COUNT B
         DW ___________; MODE/CONTROL WORD
         DW ___________; TMR 2 COUNT REGISTER
         DW ___________; MAX COUNT A
         DW ___________; THERE IS NO MAX COUNT B
         DW ___________; MODE/CONTROL WORD

C-6
SETTING UP THE DMA CONTROL BLOCK

You have just located a 128 byte sector on a floppy disc and wish to DMA the data into memory at the address CO00H onward. The floppy disk will synchronise the transfer. Use DMA channel 0 and provide an interrupt when the transfer is complete. Give this DMA channel high priority. You will be transferring bytes and the transfer is to start immediately. Use a block move method for loading the registers like the example given for loading the timer control registers.

```
CODE_1 SEGMENT
ASSUME ________________________

; Prog_dma control block

REP MOV$ TABLE,PROG_DMA   ; ASSEMBLER WILL GIVE CS: OVERIDE PREFIX
; TO ACCESS PROG_DMA

etc....

PROG_DMA: DW ___________    ; SOURCE POINTER LS 16 BITS
          DW ___________    ; SOURCE POINTER MS 4 BITS
          DW ___________    ; DESTINATION POINTER LS 16 BITS
          DW ___________    ; DESTINATION POINTER MS 4 BITS
          DW ___________    ; TRANSFER COUNT
          DW ___________    ; CONTROL WORD
```
THE INTERRUPT CONTROL BLOCK

The requirements for the interrupts are as follows ...

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER 0</td>
<td>no interrupt (mask it out)</td>
</tr>
<tr>
<td>TIMER 1</td>
<td>interrupt, priority level 3 (real time clock)</td>
</tr>
<tr>
<td>TIMER 2</td>
<td>no interrupt</td>
</tr>
<tr>
<td>DMA 0</td>
<td>interrupt, priority level 4 (floppy disk data)</td>
</tr>
<tr>
<td>DMA 1</td>
<td>not used (mask it out)</td>
</tr>
<tr>
<td>INT 0</td>
<td>interrupt, priority level 2, level triggered (UART ready),</td>
</tr>
<tr>
<td>INT 1</td>
<td>interrupt, priority level 5, level triggered (keyboard interrupt)</td>
</tr>
<tr>
<td>INT 2</td>
<td>unused (mask it out)</td>
</tr>
<tr>
<td>INT 3</td>
<td>unused</td>
</tr>
</tbody>
</table>

...(low number = high priority)

Note that when it comes to the timer interrupts, the individual timers are programmed to produce an interrupt or not. In the interrupt control block you will see that all three timers would produce the same interrupt. In this case the interrupt has to be from TIMER 1, but generally you will have to read the interrupt status register to see which one interrupted. Since we are going to mask out interrupts from TIMER 0, TIMER 2, and INT2/3 we don't need to set up their control registers. Use the priority mask register to block out interrupt levels 5, 6 and 7. You might find page 30 of the data booklet helpful here.

MOV TABLE+28H,________________________; SET MASK REGISTER
____________________________________; PRIORITY MASK REGISTER
____________________________________; TIMER CONTROL REGISTER
____________________________________; DMA O CONTROL REGISTER
____________________________________; INT O CONTROL REGISTER
____________________________________; INT 1 CONTROL REGISTER
IN THE EVENT OF AN INTERRUPT ....

There you are, at peace with the world when suddenly you get an interrupt to say that the floppy disk controller (via DMA channel 0) has just finished passing its 128 byte block of data to you. You have written the service routine to handle this event, but before you return from the routine you must remember to tell the interrupt controller that you have finished. It needs to know this in case a lower priority interrupt is pending, waiting for you to finish. Write the code to tell the interrupt controller, via the EOI register that you have finished servicing this interrupt.

_________________________________________________________; SEND END OF INTERRUPT
1. The following is a list of implicit uses of the iAPX 86,88 general register set. Supply the register name for each:
   - Word multiply, word divide, word I/O
   - BYTE multiply, BYTE divide, BYTE I/O
   - Translate
   - Word multiply, word divide, indirect I/O
   - Loops
   - Variable shift and rotate

2. Which four general purpose registers can be used in an address expression?

3. What does the assembler use to associate a particular segment register with a particular segment?

4. What determines the type (near or far) of a RET instruction contained within a procedure?

5. For every variable definition, the assembler tracks what three attributes?

6. Fill in the blank fields in the following chart:

<table>
<thead>
<tr>
<th>TYPE OF MEMORY</th>
<th>DEFAULT SEG REG</th>
<th>ALT SEG REG</th>
<th>OFFSET SUPPLIED BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP CODE FETCH</td>
<td>CS</td>
<td></td>
<td>IP</td>
</tr>
<tr>
<td>STACK OPERATION</td>
<td>SS</td>
<td>NONE</td>
<td></td>
</tr>
<tr>
<td>STRING SOURCE</td>
<td></td>
<td>CS,ES,SS</td>
<td>SI</td>
</tr>
<tr>
<td>STRING DEST</td>
<td>ES</td>
<td></td>
<td>DI</td>
</tr>
<tr>
<td>GENERAL DATA ACCESS</td>
<td>DS</td>
<td></td>
<td>EFFECTIVE ADDRESS</td>
</tr>
<tr>
<td>8P USED AS BASE</td>
<td>CS,ES,DS</td>
<td></td>
<td>EFFECTIVE ADDRESS</td>
</tr>
</tbody>
</table>
1. An assembly language procedure is required which will be linked to a PL/M program. The declaration of the procedure in PL/M and a calling sequence are as follows...

```
ASSEMBLER_CODE: PROCEDURE(ARRAY_PTR,COUNT) EXTERNAL;
DECLARE ARRAY_PTR POINTER,
    COUNT BYTE;
END;

CALL ASSEMBLER_CODE(@TABLE,1);
```

Define a structure in assembly language which will describe the stack frame which your assembly language program will use. The large model of compilation has been used for the PL/M program.

2. List the abbreviation for each of the following LINK86 controls:

MAP
SYMBOLS
BIND
PRINT
NO LINES

Circle the general purpose registers which you must preserve when linking an assembly language procedure to a PL/M program.

AX  BX  CX  DX  SI  DI  BP  SP
4. Given the following data segment:

DATA SEGMENT

DIRECTORY STRUC

LAST NAME   DB   10 DUP (?)
FIRST NAME  DB   ?
DEPT        DW   ?
XTENSION    DB   4 DUP (?)

DIRECTORY ENDS

PHONE   DIRECTORY  1000 DUP (<<>)

DATA ENDS

Evaluate the following expressions:

a. TYPE DEPT
b. TYPE PHONE
c. SIZE LAST NAME
d. TYPE DIRECTORY
e. LENGTH PHONE
f. .DEPT
g. SIZE PHONE
1. Supply the AS86 variable definition required for each of the following 8087 data types:

   - LONG REAL
   - PACKED DECIMAL
   - WORD INTEGER
   - SHORT INTEGER
   - TEMPORARY REAL

TRUE OR FALSE:

2. The 8087 stores all variables internally in the temporary real format. T

3. The 8087 always fetches and stores its operands as bytes so that it will be compatible with the 8088. T F

4. How does the execution of these two instructions differ?

   FADD
   FADDP  ST(1),ST
1. At what address is the 80186 peripheral control block following a reset?

2. Are the following instructions valid on a 186 ...

   - MUL AX,6
   - KOR FRED,13
   - PUSHI 11
   - POP 6
   - IMUL AX,BX,5

3. On reset, which memory bank will be selected by the 186 chip select lines. How large is the memory partition assumed to be?

4. Once the DMA channels have been programmed to start, the first DMA cycle will start (choose one) ...

   1) immediately
   2) next time a DMA request occurs
   3) one instruction after the start command was sent to the DMA channel

5. What are the principle uses of timer 2?

   1) 
   2) 
   3)
CLASS EXERCISE SOLUTIONS

EXERCISE 2.1
1. YES (Memory with Reg, Immed to Memory)
2. YES, by +EA (see page 2 ASMB6 Macro Assembler Pocket Ref.)
3. Because register contents and numbers may have to be added together at run time.
4. 16 + EA = 25 clocks

EXERCISE 3.1
1. To tell the assembler that a CS:override prefix is required
2. DIRECT NEAR - destination is a near label
   DIRECT FAR - destination is a far label
   INDIRECT NEAR - destination is a word register or a word variable
   INDIRECT FAR - destination is a double word variable
3. ADD SP, 4 ; waste return base and offset
   POP AX ; flags into AX
   OR AX, 10H; set trap flag bit
   PUSH AX ; print flags image on stack
   PUSH ES ; print return base on stack
   PUSH DI ; print return offset on stack
   IRET ; return to new address, setting trap flag

EXERCISE 4.1 NEAR AND FAR PROCEDURES
TRUE OR FALSE?

Solutions  * Giving a procedure the FAR attribute does the following things...
TRUE  1. encodes a far RET instruction
TRUE  2. tags the procedure as far
FALSE (5 bytes) 3. because of 2, all calls to this procedure will take 3 bytes
FALSE  * Calling a FAR procedure from the segment in which it was defined produces a near call
FALSE  * If in ignorance I near call a procedure which is defined in another module as far the RET instruction prints an error message ...
  'HELP - I can't find a segment to return to !'

EXERCISE 5.1
%*DEFINE (STRING_MOVE (SOURCE, DEST, COUNT)) (  
  MOV  CX, % COUNT
  LEA  SI, % SOURCE
  LEA  DI, % DEST
  PUSH DS
  POP ES
  REP MOVSB)
EXERCISE 6.1

```assembly
EMPLOYEE STRUC
LAST_NAME DB IØ DUP (?)
FIRST_NAME DB IØ DUP (?)
MI DB ?
DIVISION DW ?
DEPT DW ?
EMPLOYEE ENDS

WORKFORCE EMPLOYEE IØØ DUP (< >)

MOV CX,LENGTH WORKFORCE
LEA BX,WORKFORCE.DIVISION
NEXT: MOV WORD PTR [BX], 12
ADD BX,TYPE WORKFORCE
LOOP NEXT
```

EXERCISE 6.2

1. BITE RECORD B23:2, RUBBIS:2
2. AND AL, MASK B23
3. TYPE BITE IS 1 (it takes 1 byte to store it)

EXERCISE 8.1

```assembly
RUN LINK86 PROG.OBJ, PROC.S.OBJ, SMALL.LIB
RUN LOC86 PROG.LNK &
ORDER (CLASSES(DATA, CONST, STACK)) &
ADDRESSES (CLASSES(DATA(200H), CODE(F0000H))) &
SEGMENTS (NVM(C000H))) &
INITCODE (F0000H) &
BOOTSTRAP
```

EXERCISE 9.1

```assembly
CGROUP GROUP CODE1
CODE1 SEGMENT
ASSUME CS:CODE1
CMP_STRING PROC
PUSH BP
MOV BP, SP
MOV CX, [BP] + 4; STRING COUNT
MOV DI, [BP] + 6; STRING 2 POINTER
MOV SI [BP] + 8; STRING 1 POINTER
PUSH DS
POP ES ; BASE FOR STRING 2
REPE CMPSB
MOV AL,Ø ; ASSUME MISMATCH
JNE EXIT
MOV AL,0FFH ; STRINGS MATCH
EXIT: POP BP
RET 6
CMP_STRING ENDP
CODE1 ENDS
```
EXERCISE 9.2

ARRAY_SUM SEG SEGMENT 'CODE'
ARRAY_SUM PROC FAR
PUSH BP
MOV BP, SP
LES DI, [BP] + 8 ; 32 BIT POINTER TO ARRAY
MOV CX, [BP] + 6 ; LENGTH OF ARRAY
MOV AX, 0 ; CLEAR SUM
AGAIN: ADD AX, ES: [DI] ; ADD ARRAY ELEMENT TO SUM
INC SI ; UPDATE ARRAY POINTER
LOOP AGAIN ; REPEAT CX TIMES
POP BP
RET 6
ARRAY_SUM ENDP
ARRAY_SUM SEG ENDS

EXERCISE 12.1

DATA SEGMENT
A DD 1.234
B DD 234
C DD 1000.
D DD 9.82
RESULT DQ ?
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE, DS:DATA

FLD A
FADD B
FDIV C
FMUL D
FSTP RESULT
EXERCISE 12.2

EXTRN INIT87:FAR

DATA_1 SEGMENT

COS_THETA DW ?
SIN_THETA DW ?
TAN_THETA DW ?

DATA_1 ENDS

CODE_1 SEGMENT

ASSUME CS:CODE_1, DS:DATA_1

SIX DW 6.0 ; MUST BE SHORT OR LONG REAL (NOT TEMP)

* REMEMBER THAT OPPOSITE AND ADJACENT SIDES OF THE RIGHT TRIANGLE INCLUDING
* THE 30 DEGREE ANGLE ARE SWAPPED FROM THE MIRROR IMAGE TRIANGLE CONTAINING
* THE ORIGINAL 60 DEGREE ANGLE. HENCE TAN = X/Y, SIN = X/HYPOT, COS = Y/HYPOT
*
TRIG PROC FAR

CALL INIT87

FLOPI ; PI - - - - -
FDIV SIX ; PI/6 - - - - -
FPFAN ; X Y - - - -
FLD ST(1) ; Y X Y - - -
FLD ST(1) ; X Y X Y - -
FDIV ST,ST(1) ; X/Y Y X Y - -
FSIP TAN_THETA ; Y X Y - - -
FLD ST(1) ; X Y X Y - -
FMUL ST,ST(0) ; X^2 Y X Y - -
RXCH ; Y X^2 Y X Y - -
FMUL ST,ST(0) ; Y^2 X^2 X Y - -
FADD ; X^2+Y^2 X Y - - -

; ** FADD IN CLASSICAL STACK MODE DOES INCLUDE A POP !!! **
FSORT ; HYPLIT X Y - - -
FDIV SI(1),ST ; HYPOT SIN Y - - -
FDIVF SI(2),ST ; SIN COS - - - -
FSIP SIN_THETA ; COS - - - - -
FSTP COS_THETA ; - - - - - - -
RET

TRIG ENDP

CODE_1 ENDS
EXERCISE 15.1

IN    AL, 0D8H
CBW
IMUL  AX, -5
MOV   DX, 0FFFFAH
OUT   DX, AX
APPENDIX F

INTRODUCTION TO PROM PROGRAMMING
PROM PROGRAMMING

WOT? PROGRAM TWO BANKS OF 8-BIT WIDE EPROMS FROM A SINGLE OBJECT FILE!

SHOWN: \textsc{UP-201 PROM PROGRAMMER}

\textbf{\textsc{UP-200/201 INTEL UNIVERSAL PROGRAMMER}}

- PROGRAMS A VARIETY OF PROMS/EPROMS USING VARIOUS PERSONALITY MODULES
- READS ROMS/PROMS/EPROMS
- READS/Writes DISK FILES ON HOST MDS
- ALLOWS EDITING OF OBJECT CODE
- FORMATS OBJECT CODE TO SUIT PROM CONFIGURATION (Eg. 2 BANKS 8 BIT WIDE = 16 BIT WIDE MEMORY)
DRIVING PROGRAMMER FROM MDS

- IPPS

INTEL PROM PROGRAMMING SOFTWARE V.x
PPS>

PROGRAMMER OPERATION

- SIMPLE COMMAND LANGUAGE EG.

    PPS >COPY :F1:LOWER,BYT TO PROM

- DO IT ALL FROM A SUBMIT FILE

- IF YOU CAN'T REMEMBER ... PPS >HELP

- IF TOTALLY LOST ... PPS >HELP HELP

FOR DETAILS ...

t:UP - 200/201 UNIVERSAL PROGRAMMER USER'S GUIDE
EXAMPLE: FORMATTING A FILE

- TAKE INPUT FILE NIBBLE.OLD AND USE FIRST 4K BYTES TO PRODUCE TWO OUTPUT FILES FOR BLOWING ODD AND EVEN BANK PROMS

BOLD TYPE IS OPERATOR ENTRY

```plaintext
PPS> FORMAT NIBBLE.OLD (0,FFFH)
LOGICAL UNIT (BIT=1, NIBBLE=2, BYTE=3, N-BYTE=4)
LU = 3
INPUT BLOCK SIZE (N BYTES)
N = 2
OUTPUT BLOCK SIZE (N BYTES)
N = 1
INPUT BLOCK STRUCTURE:
NUMBER OF INPUT LOGICAL UNITS = 002

LSB
[00][01]
----------

NUMBER OF OUTPUT LOGICAL UNITS = 001
OUTPUT SPECIFICATION (CR TO EXIT):
*0 TO :F1: LOWER.BYT
OUTPUT STORED
*1 TO :F1: UPPER.BYT
OUTPUT STORED
*<CR>
PPS>
```
INTEL WORKSHOPS

Microcomputer Workshops—Architecture & Assembly Language
Introduction to Microprocessors
MCS®-48/49 Microcontrollers
MCS®-51 Microcontrollers
MCS®-96 16-Bit Microcontrollers
MCS®-80/85 Microprocessors
iAPX 86, 88, 186 Microprocessors, Part I
iAPX 86, 88, 186 Microprocessors, Part II
iAPX 286 Microprocessors
Data Communications including Ethernet
Speech Communication with Computers
iCEL™ VLSI Design

Programming and Operating Systems Workshops
Beginning Programming Using Pascal
PL/M Programming
PL/M-iRMX™ 51 Operating System
iRMX™ 86 Operating System
XENIX™/C Programming
System 86/300 Applications Programming
iDIS™ Database Information System
iTPS Transaction Processing System
Development System Seminars

System 2000® Database Management Workshops
System 2000® For Non-Programmers
System 2000® Technical Fundamentals
System 2000® Applications Programming
System 2000® Report Writing
System 2000® Database Design and Implementation
Self-Study Introduction to Microprocessors
System 2000® Multimedia Course

BOSTON AREA
27 Industrial Avenue, Chelmsford, MA 01824 (617) 256-1374

CHICAGO AREA
Gould Center, East Tower
2550 Golf Road, Suite 815, Rolling Meadows, IL 60008 (312) 981-7250

DALLAS AREA
12300 Ford Road, Suite 380, Dallas, TX 75234 (214) 484-8051

SAN FRANCISCO AREA
1350 Shorebird Way, Mt. View, CA 94043 (415) 940-7800

WASHINGTON D.C. AREA
7833 Walker Drive, 5th Fl., Greenbelt, MD 20770 (301) 474-2878

LOS ANGELES AREA
Kilroy Airport Center, 2250 Imperial Highway, El Segundo, CA 90245 (415) 940-7800

CANADA
190 Attwell Drive, Toronto, Ontario M9W 6H8 (416) 675-2105