iAPX 86, 88, 186 MICROPROCESSORS
PART I

WORKSHOP NOTEBOOK

VERSION 2.0       JUNE 1984
Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

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

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

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

- BX, IC, ICE, iCE, ICS, iDBP, iDIS, iLBX, iM, iMMX,
- Insite, INTEL, Intelevision, Intellec, intelligent identifier™,
- intelBOS, intelligent Programming™, Intelink, iPDS,
- iRMS, iSBC, iSBX, iSDM, iXM, Library Manager, MCS,
- Megachassis, Micromainframe, MULTIBUS, Multichannel™
- Plug-A-Bubble, MULTIMODULE, PROMPT, Ripplemode,
- RMX/80, RUPI, System 2000, and UPI, and the combination of
- ICE, iCS, iRMX, iSBC, MCS, or UPI and a numerical suffix.
iAPX 86,88,186
MICROPROCESSORS
PART I

WORKSHOP NOTEBOOK
VERSION 2.0 JUNE 1984

© Intel Corporation 1984
# TABLE OF CONTENTS

1 THE iAPX 86 PRODUCT FAMILY
   - Products
   - Nomenclature
   - Course Contents

2 INTRODUCTION TO MICROPROCESSORS
   - Registers
   - Number Systems
   - Flags

3 INTRODUCTION TO SEGMENTATION
   - Segments
   - Segment Registers
   - Physical Addresses
   - Segment Usage

4 INTRODUCTION TO THE iAPX 86, 88 INSTRUCTION SET
   - Creating a Segment
   - Assume Statement
   - MOV and XCHG
   - IN and OUT
   - Shift and Rotate

5 MORE INSTRUCTIONS
   - HLT
   - JMP
   - LOOP

6 SOFTWARE DEVELOPMENT
   - Series III Development System
   - File Utilities
   - AEDIT

7 ARITHMETIC, LOGICAL AND CONDITIONAL INSTRUCTIONS
   - ADD, SUB, MUL, DIV and CMP
   - Conditional Jumps
   - AND, OR, XOR, NOT and TEST

8 DEFINING AND ACCESSING DATA
   - Defining Data
   - Initializing Segment Registers
   - Addressing Modes

9 PROGRAM DEVELOPMENT
   - DEBUG-86
   - ASM86
   - SUBMIT Files
10 BASIC CPU DESIGN AND TIMING
- Minimum Mode
- Maximum Mode
- Instruction Queue
- 8086, 8088, 8284A, 8288, 8286 and 8282

11 PROCEDURES
- Procedure Definition
- Stack Creation and Usage
- Parameter Passing
- Example

12 PROGRAMMING WITH MULTIPLE SEGMENTS
- Multiple Code Segments
- Procedure Declaration
- Multiple Data Segments
- Segment Override Instruction Prefix
- Forward References

13 INTERRUPTS
- iAPX 86, 88 Interrupt System
- Creating an Interrupt Routine
- 8259A Programmable Interrupt Control Unit
- Programming the 8259A

14 MEMORY AND I/O INTERFACING
- Memory Organizations
- Speed Requirements
- Address Decoding

15 PROGRAMMING TECHNIQUES
- JMP Table (Indirect Jumps)
- Block Move (String Instructions)
- Table Look-up (XLATB)

16 MODULAR PROGRAMMING
- PUBLIC Declarative
- EXTRN Declarative
- Combining Segments
- LINK86
- LOC86

17 INTRODUCTION TO THE iAPX 186, 188 MICROPROCESSORS
- Description
- Enhancements
- New Instructions
- Peripherals
18 MULTIBUS SYSTEM INTERFACE
- Design Considerations
- Hardware Interface to the Multibus
- Bus Arbitration
- Lock Instructions Prefix
- Byte Swap Buffer

19 MULTI AND COPROCESSING
- 8087 Numeric Data Processor
- 8089 I/O Processor
- 80130 Operating System

20 iAPX 186, 188 HARDWARE INTERFACE
- Bus Interface
- Clock Generator
- Internal Peripherals Interface
- Differences

21 THE iAPX 286 and iAPX 386 MICROPROCESSORS
- Description
- Enhancements

APPENDICES
A Lab Exercises
B Lab Solutions
C Class Exercise Solutions
D Daily Quizzes
E Unpack Decimal Arithmetic Instructions
F ICE 86
## iAPX 86, 88, 186 MICROPROCESSORS
### WORKSHOP SCHEDULE

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Day One</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>THE iAPX 86 PRODUCT FAMILY</td>
<td>Lab 1 - Using the Series III Development System</td>
</tr>
<tr>
<td>2</td>
<td>INTRODUCTION TO MICROPROCESSORS</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>INTRODUCTION TO SEGMENTATION</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>INTRODUCTION TO THE iAPX 86, 88 INSTRUCTION SET</td>
<td>Optional AEDIT</td>
</tr>
<tr>
<td>5</td>
<td>MORE INSTRUCTIONS</td>
<td>Basic Lab</td>
</tr>
<tr>
<td>6</td>
<td>SOFTWARE DEVELOPMENT</td>
<td></td>
</tr>
</tbody>
</table>

### Day Two

| 7       | ARITHMETIC, LOGICAL AND CONDITIONAL INSTRUCTIONS | Lab 2 - Defining and Accessing Data |
| 8       | DEFINING AND ACCESSING DATA | |
| 9       | PROGRAM DEVELOPMENT | |
| 10      | BASIC CPU DESIGN AND TIMING | |

### Day Three

| 11      | PROCEDURES | Lab 3 - Using Procedures (Linking with PL/M), Multiple Segments, and Interrupts |
| 12      | PROGRAMMING WITH MULTIPLE SEGMENTS | |
| 13      | INTERRUPTS | |
| 14      | MEMORY AND I/O INTERFACING | |

### Day Four

| 15      | PROGRAMMING TECHNIQUES | Lab 4 - Modular Programming |
| 16      | MODULAR PROGRAMMING | |
| 17      | INTRODUCTION TO THE iAPX 186, 188 MICROPROCESSORS (optional) ICE 86 | Optional Lab - ICE Demo |
| 18      | MULTI AND COPROCESSING | |
| 19      | MULTIBUS SYSTEM INTERFACE | |
| 20      | iAPX 186, 188 HARDWARE INTERFACE | |
| 21      | The iAPX 286 and iAPX 386 MICROPROCESSORS | |

Labs are shown for information only. All labs are self-paced and as a result are not scheduled or assigned.
DAY 1 OBJECTIVES

BY THE TIME YOU FINISH TODAY YOU WILL:

* DEFINE THE TERMINOLOGY USED TO DESCRIBE THE iAPX 86,88,186,188 FAMILY OF PRODUCTS

* DEFINE THE THREE BASIC COMPONENTS OF EVERY MICROPROCESSOR DESIGN AND THE BUSSES THAT CONNECT THEM

* MATCH THE CPU POINTER REGISTERS WITH THE TYPE OF MEMORY THEY ARE USED TO ACCESS

* DEFINE TYPICAL SEGMENT REGISTER USE

* USE THE ASSEMBLER DIRECTIVES REQUIRED TO DEFINE A SEGMENT

* CREATE, ASSEMBLE, AND EXECUTE A PROGRAM USING THE SERIES III DEVELOPMENT SYSTEM
CHAPTER 1

THE iAPX 86 PRODUCT FAMILY

- PRODUCTS
- NOMENCLATURE
- COURSE CONTENTS
GENERATIONS OF MICROPROCESSOR SYSTEMS

SOFTWARE

HIGH LEVEL LANGUAGES

PASCAL 86 (APPLICATIONS)
PLM 86 (SYSTEMS IMPLEMENTATION, APPLICATIONS)
FORTRAN 86 (APPLICATIONS, MATH)
C 86 (SYSTEM IMPLEMENTATION, APPLICATIONS)

ASSEMBLY LANGUAGE

ASM 86 ("HIGH LEVEL" ASSEMBLER)

SYSTEM SOFTWARE

IRMX 86 OPERATING SYSTEM (FULL FUNCTION)
IRMX 88 EXECUTIVE (SMALL, FAST)
IMMX 800 MESSAGE EXCHANGE SOFTWARE (MULTIPROCESSOR COMM.)
XENIX OPERATING SYSTEM (FULL FUNCTION)
DEVELOPMENT SUPPORT

SERIES II DEVELOPMENT SYSTEM
(8085 PROCESSOR ONLY, PLM86, ASM86)

SERIES III DEVELOPMENT SYSTEM
(8086 AND 8085 PROCESSORS, FORTRAN 86, PLM86, ASM86, DEBUG-86, PASCAL 86, C86)

SERIES IV DEVELOPMENT SYSTEM
(8086 AND 8085 PROCESSORS, ENHANCED HUMAN INTERFACE)

ICE86 ICE86A
(IN CIRCUIT EMULATOR, POWERFUL SOFTWARE AND HARDWARE DEBUGGING TOOL, USED WITH SERIES II OR III)

I2ICE
(INTEGRATED INSTRUMENTATION AND IN-CIRCUIT EMULATION SYSTEM FOR 8086, 80186, 80286, USED WITH SERIES III OR IV)

LINK86, LOC86, LIB86
(UTILITIES PROGRAMS THAT SUPPORT MODULAR PROGRAMMING, RUN ON SERIES II OR SERIES III)

ISBC 957B PACKAGE
(DOWNLOAD AND DEBUG FOR ISBC88 BOARDS)

1-3

iAPX 86 PRODUCT FAMILY

HARDWARE

SINGLE BOARD COMPUTERS

ISBC 86/30 BOARD (6MHz 8086, 128K RAM, FULL FUNCTION)
ISBC 86/12A BOARD (5MHz 8086, 32K RAM, FULL FUNCTION)
ISBC 86/05 BOARD (6MHz 8086, 80/12A COMPATIBLE, 8K RAM)
ISBC 88/40 BOARD (5MHz 8088, ANALOG IO, PROCESS CONTROL)
PLUS OVER 40 ADDITIONAL IO AND MEMORY EXPANSION BOARDS

PROCESSORS

IAPX 86 (GENERAL 16 BIT DATA PROCESSOR)
IAPX 88 (IAPX 86 WITH 8 BIT EXTERNAL DATA BUS)
IAPX 186 (HIGHER HARDWARE INTEGRATION)
IAPX 188 (IAPX 186 WITH 8 BIT EXTERNAL DATA BUS)
IAPX 286 (HIGHER SOFTWARE INTEGRATION)
8089 IOP (HIGH SPEED DMA AND IO)

PROCESSOR EXTENSIONS

NUMERICS COPROCESSOR (8087, HIGH SPEED MATH)
OPERATING SYSTEM EXTENSION (80130 FAST OPERATING SYSTEM NUCLEUS)

1-4
iAPX 86, iAPX 88 MODEL NUMBERS

<table>
<thead>
<tr>
<th>CPU</th>
<th>MODEL NUMBERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAPX 86/10</td>
<td>8088</td>
</tr>
<tr>
<td>IAPX 86/11</td>
<td>8088 8089</td>
</tr>
<tr>
<td>IAPX 86/20</td>
<td>8088 8087</td>
</tr>
<tr>
<td>IAPX 86/21</td>
<td>8088 8087 8089</td>
</tr>
<tr>
<td>CPU 80130 OSP</td>
<td>8088 80130</td>
</tr>
</tbody>
</table>

SIMILAR FOR
iAPX 88, iAPX 186, iAPX 188

iAPX 86 PRODUCT FAMILY

SOFTWARE

<table>
<thead>
<tr>
<th>PLM 86</th>
<th>ASM 86*</th>
<th>IRMX 86</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASCAL 86</td>
<td>IAPX 88*</td>
<td>IRMX 88</td>
</tr>
<tr>
<td>FORTRAN 86</td>
<td>IAPX 88*</td>
<td>IMMX 800</td>
</tr>
</tbody>
</table>

HARDWARE

<table>
<thead>
<tr>
<th>ISBC 88/12A</th>
<th>IAPX 88*</th>
<th>8087*</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISBC 88/05*</td>
<td>IAPX 88*</td>
<td>8089*</td>
</tr>
<tr>
<td>ISBC 88/40</td>
<td>IAPX 186*</td>
<td>80130*</td>
</tr>
<tr>
<td>IAPX 188*</td>
<td>IAPX 286*</td>
<td></td>
</tr>
</tbody>
</table>

DEVELOPMENT SUPPORT

<table>
<thead>
<tr>
<th>SERIES II*</th>
<th>ICE 88*</th>
<th>1ICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERIES III*</td>
<td>LINK 86*</td>
<td>SDK 86</td>
</tr>
<tr>
<td>SERIES IV</td>
<td>LOC 88*</td>
<td>957 B</td>
</tr>
<tr>
<td></td>
<td>LIB 88</td>
<td></td>
</tr>
</tbody>
</table>

* = COVERED IN THIS COURSE
FOR MORE INFORMATION...

ALL INTEL PRODUCTS ARE DESCRIBED IN
- MICROPROCESSOR AND PERIPHERAL HANDBOOK
- MEMORY COMPONENTS HANDBOOK
- OEM SYSTEMS HANDBOOK

AVAILABLE COURSES
- INTEL WORKSHOPS CATALOG
CHAPTER 2

INTRODUCTION TO MICROPROCESSORS

- REGISTERS
- NUMBER SYSTEMS
- FLAGS
MICROCOMPUTER SYSTEM

FUNCTIONAL SECTIONS

ADDRESS BUS

MEMORY (2)

INPUT/OUTPUT (3)

CPU

MODULE (1)

DATA BUS

CONTROL BUS

1 OPERATIONS
DECISIONS

2 PROGRAMS,
STACK, DATA

3 EXTERNAL
COMMUNICATION

BUS FUNCTIONS

ADDRESS BUS

20 BITS UNI-DIRECTIONAL (OUTPUT ONLY)
MEMORY ADDRESS 0 TO $2^{20}$ (1,048,576)
I/O ADDRESS 0 TO $2^{16}$ (65,536)

DATA BUS

16 BITS BI-DIRECTIONAL (READ/WRITE)
THUS MEMORY AND I/O DATA WIDTH 8 OR 16 BITS

CONTROL BUS

INCLUDES THREE CONTROL LINES
M/Đ = I/O OR MEMORY SELECTOR
RD = READ
WR = WRITE
iAPX 86,88 CPU PROGRAMMING MODEL

15
  \[ \begin{array}{c|c}
    \text{AX} & \text{AH} & \text{AL} \\
    \text{BX} & \text{BH} & \text{BL} \\
    \text{CX} & \text{CH} & \text{CL} \\
    \text{DX} & \text{DH} & \text{DL} \\
  \end{array} \]

WORD

\(15 - 7\)  7 - 0  0

\(\text{SP} \quad \text{BP} \quad \text{SI} \quad \text{DI} \)

\(\text{IP} \quad \text{FLAGS}\)

* POINTER REGISTER

INSTRUCTION POINTER

MEMORY

INSTRUCTION
INSTRUCTION
INSTRUCTION
INSTRUCTION
INSTRUCTION

IP (16)
STACK POINTER

CONTAINS ADDRESS OF TOP OF STACK

DATA POINTERS

BX, SI, DI OR BP

EXAMPLES

MOV CX, 0005
MOV [BX], CX
MOV AX, [SI]
TYPICAL MEMORY USAGE

CPU

- IP (INSTRUCTION POINTER)
- SP (STACK POINTER)
- DATA POINTER

MEMORY

- INSTRUCTION STORAGE AREA
- ROM/PROM/EPROM/RAM
- STACK AREA
- RAM
- VARIABLE STORAGE AREA
- RAM

NUMBER SYSTEMS

<table>
<thead>
<tr>
<th>HEX</th>
<th>BINARY</th>
<th>DECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
<td>13</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
<td>14</td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
<td>15</td>
</tr>
</tbody>
</table>

21H = 0010 0001 B
96H = 1001 0110 B
42H = 0100 0010 B
TWO’S COMPLEMENT ARITHMETIC
SIGNED vs UNSIGNED BINARY NUMBERS

SIGNED: -128 TO +127

UNSIGNED: 0 - 255

TWO’S COMPLEMENT NUMBER REPRESENTATION

EXAMPLE OF TWO’S COMPLEMENT:

<table>
<thead>
<tr>
<th>BINARY</th>
<th>DECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 0000</td>
<td>-128</td>
</tr>
<tr>
<td>1000 0001</td>
<td>-127</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1111 1111</td>
<td>-1</td>
</tr>
<tr>
<td>0000 0000</td>
<td>0</td>
</tr>
<tr>
<td>0000 0001</td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0111 1110</td>
<td>+126</td>
</tr>
<tr>
<td>0111 1111</td>
<td>+127</td>
</tr>
</tbody>
</table>
FLAG WORD

- **FLAGS**:
  - Carry
  - Parity
  - Auxiliary Carry
  - Zero
  - Sign
  - Trap
  - Interrupt-Enable
  - Direction
  - Overflow

- **STATUS FLAGS**
- **CONTROL FLAGS**
- **STATUS FLAG**

FLAG OPERATIONS

- **Operand 1**
- **Operand 2**
- **Result (Operand 1)**
  - BIT 7
  - BIT 6
  - BIT 5
  - BIT 4
  - BIT 3
  - BIT 2
  - BIT 1
- **A.C.**
- **Z**
- **CARRY**
FOR MORE INFORMATION ...

INTRODUCTION TO MICROCOMPUTERS AND THE 8086
- CHAPTER 1 AND 2, iAPX 86/88, 186/188 USER’S MANUAL

REGISTERS AND FLAGS
- CHAPTER 3, iAPX 86/88, 186/188 USER’S MANUAL
- APPENDIX B, ASM86 LANGUAGE REFERENCE MANUAL

SIGNED BINARY NUMBERS
PAGES 3–22, 23, iAPX 86/88, 186/188 USER’S MANUAL
CHAPTER 3

INTRODUCTION TO SEGMENTATION

- SEGMENTS
- SEGMENT REGISTERS
- PHYSICAL ADDRESSES
- SEGMENT USAGE
MEMORY IS USED TO STORE THREE TYPES OF INFORMATION.

* THE 8086 VIEWS MEMORY AS A GROUP OF SEGMENTS.

* A SEGMENT IS A LOGICAL UNIT OF MEMORY.

* SEGMENTS CANNOT BE GREATER THAN 64K LONG.

THE CPU HAS 4 SEGMENT REGISTERS.

THE SEGMENT REGISTER POINTS TO THE BEGINNING OF A SEGMENT.
SEGMENT REGISTERS AND SEGMENTATION

00000H

01000H

03CFOH

FFOOOH

FFFFFH

CS: 0000

DS: 0100

ES: 03CF

SS: FF00

SEGMENTATION

* SEGMENTED ADDRESSING HAS MANY ADVANTAGES OVER LINEAR ADDRESSING

1) REGISTER SIZE
2) DYNAMIC CODE RELOCATION
3) MEMORY MANAGEMENT

* SEGMENTS ARE DEFINED BY APPLICATION
SEGMENTS ARE DEFINED BY APPLICATION

A FEW EXAMPLES

SIMPLE PROGRAM
- 64K CODE
- 64K DATA
- 64K STACK
(OUR MODEL)

MORE CODE

DATA 1
STACK
CODE 2

DATA 2
STACK
CODE 2

TWO PROGRAMS (TASKS)
SHARING ONE PROCESSOR

ACCESSING MEMORY IN A SEGMENT

* TO ACCESS A PARTICULAR BYTE (OR WORD) IN A SEGMENT, THE CPU USES AN OFFSET

* THE OFFSET OF A BYTE (OR A WORD) IS THE DISTANCE IN BYTES FROM THE BEGINNING OR BASE OF THE SEGMENT

* THIS BASE ADDRESS IS SUPPLIED BY THE SEGMENT REGISTER
USING THE SEGMENT REGISTER CONTENTS

FETCHING INSTRUCTIONS
* INSTRUCTIONS ARE ALWAYS FETCHED WITH RESPECT TO THE CS REGISTER.
ACCESSING THE STACK
* THE STACK IS ALWAYS REFERENCED WITH RESPECT TO THE STACK SEGMENT REGISTER.

ACCESSING DATA
* THE OFFSET VALUE CAN BE OBTAINED IN MANY WAYS.
DATA IS TYPICALLY FETCHED WITH RESPECT TO THE DATA SEGMENT REGISTER.
ASSUME AN INSTRUCTION IS LOCATED AT A PHYSICAL ADDRESS OF 05820H.

1. WHAT REGISTER(S) WOULD THE CPU USE TO FETCH THIS INSTRUCTION?

2. NAME THREE COMBINATIONS OF VALUES THAT THE CPU COULD USE TO FETCH THAT SAME INSTRUCTION.

ASSUME A WORD OF DATA IS LOCATED AT AN OFFSET OF 210H FROM A SEGMENT BEGINNING AT PHYSICAL ADDRESS 00020H.

3. WHAT REGISTER(S) WOULD THE CPU TYPICALLY USE TO READ THIS DATA?

4. WHAT IS THE PHYSICAL ADDRESS OF THE DATA?

5. WHAT WOULD BE THE VALUE IN THE SEGMENT REGISTER?
FOR MORE INFORMATION ...

PHYSICAL ADDRESS GENERATION
- CHAPTER 3, iAPX 86/88, 186/188 USER’S MANUAL

SEGMENTATION CONCEPTS
- CHAPTER 3, iAPX 86/88, 186/188 USER’S MANUAL
- CHAPTER 2, ASM86 LANGUAGE REFERENCE MANUAL
CHAPTER 4
INTRODUCTION TO THE iAPX 86.88 INSTRUCTION SET

- CREATING A SEGMENT
- LABELS AND SYMBOLS
- ASSUME STATEMENT
- MOV,XCHG
- IN,OUT
- SHIFT,ROTATE
INSTRUCTIONS ARE CONTAINED IN SEGMENTS.

HOW DO YOU CREATE A SEGMENT?

SEGMENT DECLARATIVE

* A SEGMENT IS DEFINED IN ASSEMBLY LANGUAGE WITH A SEGMENT DECLARATIVE.

* ALL OFFSETS ARE CALCULATED FROM THE SEGMENT DECLARATIVE.
ASM86 FEATURES

IDENTIFIERS
  UPPER AND LOWER CASE ALPHA CHARACTERS (A-Z, a-z)
  NUMERIC CHARACTERS (0-9)
  3 SPECIAL CHARACTERS (?, @, _)
  - ALL IDENTIFIERS MUST BEGIN WITH AN ALPHA CHARACTER OR ONE OF THE 3 SPECIAL CHARACTERS
  - FIRST 31 CHARACTERS ARE SIGNIFICANT

NUMERIC CONSTANTS
  D  DECIMAL
  H  HEXDECIMAL
  Q or O  OCTAL
  B  BINARY
  - DEFAULT BASE IS DECIMAL
  - ALL NUMERIC CONSTANTS MUST BEGIN WITH A DIGIT

- EITHER TABS OR SPACES CAN BE USED AS DELIMITERS
- CERTAIN NAMES HAVE PREDEFINED MEANINGS AND CANNOT BE USED AS IDENTIFIERS

ASSUME DECLARATIVE

THE ASSUME DECLARATIVE ASSOCIATES A SEGMENT REGISTER WITH A SEGMENT NAME

THE ASSUME DOES NOT GENERATE ANY CODE

```
CODE_1 SEGMENT
ASSUME CS:CODE_1

; CODE_1

CODE_1 ENDS
```

MORE ON THIS LATER!!
INSTRUCTIONS

BYTE OR WORD OPERATIONS USE THE SAME MNEMONIC.

BOTH OPERANDS MUST BE THE SAME LENGTH, BYTE OR WORD.

EXAMPLES:

MOV AL, BL ; LEGAL — BOTH BYTE
MOV AX, BX ; LEGAL — BOTH WORD
MOV BX, AL ; ILLEGAL — ONE BYTE, ONE WORD

MOV
XCHG

* MOV BYTES OR WORDS BETWEEN REGISTERS AS WELL AS BETWEEN REGISTERS AND MEMORY

MOV DESTINATION, SOURCE — TRANSFER BYTE OR WORD FROM SOURCE TO DESTINATION

XCHG OP1, OP2 — EXCHANGE BYTE OR WORD, OP1 ↔ OP2

EXAMPLES

MOV AX, BX
XCHG BL, BH
XCHG SI, DI
MOV CX, [BX]
**IMMEDIATE DATA**

*MANY INSTRUCTIONS CAN USE IMMEDIATE DATA*

```
MOV AX, 2345H
MOV BL, 123D
```

*EQU STATEMENTS ARE USEFUL WITH IMMEDIATE DATA*

```
DAYS_IN_YEAR   EQU 365

MOV CX, DAYS_IN_YEAR
```

*EQU IS NOT AN INSTRUCTION AND DOES NOT ALLOCATE ANY MEMORY*

---

**IN, OUT**

- **IN AL, PORT**
- **OUT PORT, AL**

- **IN AX, PORT**
- **OUT PORT, AX**

PORT = 0 TO 255
I/O OPERATION DIRECT PORT

CPU

ADDRESS BUS

DECODE

LOGIC

DATA BUS

LATCH

WR

M/IO

BMX

AX 04

MOV AL, 00000100B
OUT 20H, AL

ANOTHER WAY.....

OR

(HOW DO YOU GET 64K IO ADDRESSES)
IN, OUT

MOV DX, PORT#
IN AL, DX
OUT DX, AL

IN AX, DX
OUT DX, AX

PORT#  0 TO 65,535

I/O OPERATION
(INDIRECT PORT)

MOV AL, 04H
MOV DX, 20H
OUT DX, AL

* BY USING THE DX REGISTER TO POINT TO I/O THE CPU CAN ACCESS UP TO 64K DIFFERENT I/O PORTS.
SHIFT INSTRUCTIONS

* ARITHMETIC SHIFTS CAN BE USED TO MULTIPLY OR DIVIDE NUMBERS BY POWERS OF TWO

- **SHIFT LOGICAL RIGHT**
  - DESTINATION $\rightarrow$ CF $\rightarrow$ SHR

- **SHIFT ARITHMETIC RIGHT**
  - DESTINATION $\rightarrow$ CF $\rightarrow$ SAR

  (SIGN BIT SHIFTED IN)

- **SHIFT LEFT**
  - CF $\rightarrow$ DESTINATION $\rightarrow$ 0 $\rightarrow$ SHL/SAL

ROTATE INSTRUCTIONS

* ROTATE INSTRUCTIONS ARE USED TO MANIPULATE BITS WITHOUT DESTROYING THE BITS

* THE CARRY FLAG CAN BE INCLUDED OR EXCLUDED IN THE OPERATION

- **ROTATE RIGHT**
  - DESTINATION $\rightarrow$ CF $\rightarrow$ ROR

- **ROTATE RIGHT THROUGH CARRY**
  - DESTINATION $\rightarrow$ CF $\rightarrow$ RCR

- **ROTATE LEFT**
  - CF $\rightarrow$ DESTINATION $\rightarrow$ ROL

- **ROTATE LEFT THROUGH CARRY**
  - CF $\rightarrow$ DESTINATION $\rightarrow$ RCL
SHIFT AND ROTATE FORMS

* TYPE OF OPERAND DETERMINES BYTE OR WORD

* SINGLE BIT FORM:
  
  SHL AX,1  :SHIFT LEFT LOGICAL
  ROR BL,1  :ROTATE RIGHT

* VARIABLE BIT FORM:
  
  MOV CL,4  :SET UP THE SHIFT
  SAR AX,CL :SHIFT CL TIMES

* ONLY THE CL REGISTER MAY BE USED TO HOLD THE VARIABLE SHIFT COUNT

* CL IS UNAFFECTED

CLASS EXERCISE 4.1

WRITE A SEQUENCE OF INSTRUCTIONS THAT WILL INPUT AN UNSIGNED BYTE FROM PORT $0FF8H, AND MULTIPLY THE BYTE BY 8. ALLOW THE MULTIPLY TO EXTEND INTO 16 BITS. THE PROGRAM SHOULD THEN OUTPUT THE WORD RESULT TO PORT $8H.
FOR MORE INFORMATION ...

ASSEMBLY LANGUAGE INSTRUCTIONS
-CHAPTER 6, ASM86 LANGUAGE REFERENCE MANUAL
-CHAPTER 3, IAPX 86/88, 186/188 USER'S MANUAL

SEGMENT DECLARATIVE
-CHAPTER 2, ASM86 LANGUAGE REFERENCE MANUAL

RELATED TOPICS ...
IN THIS COURSE WE DO NOT COVER THE BIT ENCODING OF MACHINE INSTRUCTIONS. DUE TO THE MANY ADDRESSING MODES AVAILABLE IN THE 8086, AND THE DESIRE TO MINIMIZE CODE SIZE, INSTRUCTION ENCODING IS MORE DIFFICULT TO UNDERSTAND THAN IN MANY PREVIOUS 8-BIT MACHINES (SUCH AS THE 8085). INFORMATION IS AVAILABLE IN

-CHAPTER 3, IAPX 86/88, 186/188 USER'S MANUAL
-APPENDIX E, ASM 86 LANGUAGE REFERENCE MANUAL
CHAPTER 5
MORE INSTRUCTIONS

• HLT
• JMP
• LOOP
HLT INSTRUCTION

```
MY_SEG
SEGMENT
ASSUME CS: MY_SEG

HLT
ENDS
```

JMP INSTRUCTION

```
MY_SEG
SEGMENT
ASSUME CS: MY_SEG

START:

JMP START
ENDS
```
JMP INSTRUCTION

JMP ±128 BYTE DISPLACEMENT ("SHORT" JUMP, 2 BYTE INSTRUCTION)
JMP ±32K BYTE DISPLACEMENT ("NEAR" JUMP, 3 BYTE INSTRUCTION)
JMP ANY SEGMENT, ANY OFFSET ("FAR" JUMP, 5 BYTE INSTRUCTION)  
(DISCUSSED LATER)

LET THE ASSEMBLER GIVE YOU THE CORRECT FORM!

DISPLACEMENTS AND OFFSETS

* THE DISPLACEMENT OF A BYTE (OR WORD) IS THE DISTANCE IN BYTES FROM THAT BYTE (OR WORD) TO ANOTHER BYTE (OR WORD).

* THE OFFSET OF A BYTE (OR WORD) IS THE DISTANCE IN BYTES FROM THE BEGINNING OF THE SEGMENT.
QUESTION
HOW CAN I EXECUTE MY PROGRAM 10 TIMES THEN STOP?

ANSWER
USE A PROGRAM LOOP.

LOOP INSTRUCTION

A SPECIAL JUMP INSTRUCTION THAT DECREASES THE CX REGISTER AND JUMPS IF CX<0

MY_SEG SEGMENT
ASSUME CS: MY_SEG
START:
AGAIN:

MOV CX, 10
...
...

LOOP AGAIN
HLT
ENDS

MY_SEG ENDS
LOOP INSTRUCTION

MY_SEG
START:
AGAIN:
MY_SEG

SEGMENT
ASSUME
MOV
CS;MY_SEG
CX,10

LOOPS AGAIN

CX=CX-1

NO

CX=0

YES

HLT

ENDS

LOOP INSTRUCTION

ALSO USEFUL FOR DELAYS

- MOV CX,0FFFFH } TAKES ≈0.2 SECONDS @ 5MHZ
SELF: LOOP SELF

- MOV CX,0FFFFH OUTER: MOV DX, CX
SELFZ: LOOP SELFZ

- MOV CX,0FFFFH INNER: LOOP INNER
- OUTER: MOV CX,0FFFFH

- MOV CX,DAX LOOP OUTER

HOW LONG WOULD THESE TAKE?

- MOV CX,0FFFFH
- MOV CX,0FFFFH
- MOV CX,0FFFFH
- MOV CX,0FFFFH
- MOV CX,0FFFFH
- MOV CX,0FFFFH
- MOV CX,0FFFFH
- MOV CX,0FFFFH
STOPPING THE ASSEMBLER

NAME DEMO
MY_SEG SEGMENT
ASSUME CS: MY_SEG

START: MOV CX,10 ;EXECUTE PROGRAM
AGAIN: 
        
        
        LOOP AGAIN
        JMP $ 
MY_SEG ENDS
END START

---

CLASS EXERCISE 5.1

1. Why doesn't the end statement make the CPU stop execution?

2. Which of the following are proper ASM86 identifiers? What is wrong with the others?
   
   a. BEGIN
   b. ?ALPHA
   c. HALT
   d. ?a
   e. 'ELEPHANT'
   f. 5TIMES
   g. GROUP?
   h. LOOP
   i. TOTAL$AMOUNT
   j. NOW IS THE TIME FOR ALL GOOD MEN
FOR MORE INFORMATION ...

ASSEMBLY LANGUAGE INSTRUCTIONS
- CHAPTER 6, ASM86 LANGUAGE REFERENCE MANUAL
- CHAPTER 3, IAPX 86/88, 186/188 USER’S MANUAL

ASSEMBLER DIRECTIVES (E.G. NAME, END)
- CHAPTER 2, ASM86 LANGUAGE REFERENCE MANUAL

RELATED TOPICS ...
THE LOOP INSTRUCTION IS ALSO AVAILABLE AS A CONDITIONAL INSTRUCTION.
LOOPE/LOOPZ
LOOPNE/LOOPNZ
SEE CHAPTER 6, ASM86 LANGUAGE REFERENCE MANUAL
CHAPTER 6
SOFTWARE DEVELOPMENT
• SERIES III DEVELOPMENT SYSTEM
• FILE UTILITIES
• AEDIT TEXT EDITOR
INITIALIZING ISIS-II

1) POWER ON COMPLETE SYSTEM
   (MDS, DISK DRIVES)

2) INSERT SYSTEM DISKETTE INTO DRIVE 0
   (DRIVE 0 IS THE DRIVE ON THE RIGHT)

3) PRESS RESET ON FRONT PANEL
SERIES III ENVIRONMENT

DIRECTORY COMMAND

*LISTS ISIS DISKETTE FILES

DIR [0] [I]

* EXAMPLE

DIR I

DIRECTORY OF :FO:86P1.002
NAME .EXT BLKS LENGTH ATTR NAME .EXT BLKS LENGTH ATTR
ISIS .DIR 26 3200 IF ISIS .MAP 5 512 IF
ISIS .TO 24 2944 IF ISIS .LAB 54 5784 IF
ISIS .BIN 94 11756 SIF ISIS .CLI 25 2984 SIF
ISIS .OVO 11 1279 SIF ATTRIB 40 4909 WS
COPY 69 8489 WS CREDIT 156 19470 WS
DELETE 39 4824 WS DIR 55 5815 WS
IDISK 63 7895 WS RENAME 20 2346 WS
RUN 214 26804 WS SUBMIT 39 4821 WS
EDIT 214 26775 WS ASM86 .86 1056 132988 WS
LINK86 .86 608 75512 WS LOC86 .86 292 36652 WS
DEMO .A86 14 1586 CREDIT.HLP 25 2985 WSI
LARGE .LIB 49 6029 W RUN .MAC 2 9
CI .OBJ 7 763 W C0 .OBJ 6 561 W
RUN .OVO 78 9724 W AEDIT .MAC 2 5 WS
TEST .LAB 3 212

3290/4004 BLOCKS USED
ISIS II NOTES

* FILE NAME CONVENTIONS:

:DEVICE:FILENAME.EXTENSION

2 CHARACTERS OPTIONAL

1 TO 6 CHARACTERS

1 TO 3 CHARACTERS OPTIONAL

:F0: INDICATES DRIVE 0
:F1: INDICATES DRIVE 1
IF NO DEVICE IS SPECIFIED :F0: IS USED

* FOR EASE OF ENTRY OF COMMAND LINES, AND OTHER INPUT:

(RUBOUT) DELETES THE PREVIOUS CHARACTER ENTERED

(CNTL-X) DELETES THE ENTIRE LINE

(CNTL-S) STOPS OUTPUT PROCESS

(CNTL-Q) RESTARTS OUTPUT PROCESS

COPY COMMAND

COPY ISISFILENAME ,ISISFILENAME ... TO ISISFILENAME
COPY :F1:LAB1.LST TO :LP:
COPY :F1:LAB1.ASM TO :F1:LAB4.ASM
DELETE COMMAND

DELETES ISIS DISKETTE FILES FROM THE DIRECTORY

DELETE ISISFILENAME

-DELETE LAB1.LST
-DELETE :F1:LAB1.LST
-DELETE :F1:LAB?.LST

DELETES LAB1.LST FILE FROM DISK IN DRIVE 0
DELETES LAB1.LST FILE FROM DISK PRESENTLY IN DRIVE 1
DELETES LAB1.LST, LAB2.LST, LAB3.LST, LABA.LST

DELETE :F1:LAB1.*

DELETES LAB1.LST, LAB1.OBJ, LAB1.ASM FROM DISK IN DRIVE 1

SOFTWARE DEVELOPMENT
(SERIES III DEVELOPMENT SYSTEM)

AEDIT  

SOURCE FILE    OBJECT FILE
ASM            .OBJ

ASM86  

LIST FILE

OBJ

LINK86  

LIST FILE

NO EXT.

BOUND OBJECT FILE

RUN

6-8
FILE CREATION

WHEN EDITING AN OLD FILE A BACKUP FILE IS CREATED OF THE OLD FILE UPON EXITING AEDIT.
AEDIT IS CALLED FROM ISIS BY ENTERING:

AEDIT FILENAME

WHERE FILENAME IS THE NEW FILE TO BE CREATED OR AN EXISTING FILE TO BE UPDATED.

EXAMPLE:

-AEDIT :F1:LAB1.ASM

IS MENU DRIVEN

INITIAL SCREEN

EOF MARKER

CURSOR TEXT AREA

MESSAGE LINE

PROMPT LINE

TO GET NEXT MENU:

TAB
THE MENUS

MENU 1
TAB

MENU 2
TAB

MENU 3
TAB

• TO INVOKE A COMMAND, KEY THE FIRST LETTER OF THE COMMAND.
• TO ABORT A COMMAND, TYPE CNTL-C.

INSERTING NEW TEXT

• TO INSERT TEXT, TYPE 'I'

6-13

6-14
Now is the time for all good men to bear arms.

[Insert]
CORRECTING MISTAKES

Now is the time
for all good men

ENDING INSERTION

Now is the time
for all good men

Again Block Delete Execute
CURSOR CONTROL

- ARROW KEYS MOVE CURSOR ONE SPACE OR LINE FOR EDITING

6-19

CURSOR MOVEMENT AND PAGING

- MOVES CURSOR TO END OF LINE

- MOVES CURSOR TO BEGINNING OF LINE

- PAGES DOWN

- PAGES UP

6-20
DELETING TEXT

CONTROL F
DELETES CHARACTER AT CURSOR

CONTROL Z
DELETES LINE ON WHICH CURSOR IS POSITIONED

CONTROL U
UNDO-RESTORES DELETED CHARACTERS

THESE ALSO WORK DURING INSERTION

6-21

ENDING AN EDITING SESSION

KEYSTROKES

Q

Insert Jump Macro Other Quit Replace
QUIT

MENU PROMPT LINE

Abort Exit Init Update Write

SUBCOMMANDS:

A - ABORT  ALL CHANGES LOST. RETURN TO OPERATING SYSTEM.
E - EXIT   FILE IS UPDATED. RETURN TO OPERATING SYSTEM.
I - INIT   STARTS NEW EDITING SESSION. DOES NOT RETURN TO OPERATING SYSTEM.
U - UPDATE UPDATES FILE. DOES NOT RETURN TO OPERATING SYSTEM.
W - WRITE  PROMPTS YOU FOR OUTPUT FILENAME. DOES NOT RETURN TO OPERATING SYSTEM.

6-23

EXIT

KEYSTROKES

---

Abort  Exit  Init  Update  Write

6-24
SOFTWARE DEVELOPMENT
(SERIES III DEVELOPMENT SYSTEM)

AEDIT → SOURCE FILE .ASM → ASM86 → OBJECT FILE .OBJ → LINK86 .BIND → SOUND OBJECT FILE NO .EXT → RUN

LIST FILE .LST

DEVELOPMENT STEPS

-AEDIT :F1:LAB1.ASM
-ASSEMBLE FILE - CREATE .LST AND .OBJ FILE
-PRINT .LST FILE LOOK AT ERRORS, IF ANY
-CREATE "RUN TIME LOCATED" FILE
-EXECUTE PROGRAM IN DEVELOPMENT SYSTEM

-RUN ASM86 :F1:LAB1.ASM
-COPY :F1:LAB1.LST TO :LP:
-RUN LINK86 :F1:LAB1.OBJ .BIND
-RUN :F1:LAB1.
ISIS-1 I COMMANDS AND ERROR MESSAGES

- INTELLEC SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM
CONSOLE OPERATING INSTRUCTIONS POCKET REFERENCE

AEDIT TEXT EDITOR

- AEDIT TEXT EDITOR POCKET REFERENCE

AEDIT HAS MANY ADVANCED COMMANDS THAT ARE NOT COVERED IN THIS COURSE. INFORMATION IS AVAILABLE IN THE AEDIT TEXT EDITOR USER'S GUIDE AND THE AEDIT LAB IN APPENDIX A.
DAY 2 OBJECTIVES

BY THE TIME YOU FINISH TODAY YOU WILL:

* WRITE EXECUTABLE PROGRAMS USING THE ARITHMETIC, LOGIC, AND CONDITIONAL INSTRUCTIONS

* ALLOCATE MEMORY SPACE AND INITIALIZE THAT DATA USING THE ASM86 DIRECTIVES

* DEBUG YOUR PROGRAMS USING THE SERIES III DEBUGGER

* WRITE A SUBMIT FILE TO "AUTOMATE" PROGRAM DEVELOPMENT

* DIFFERENTIATE BETWEEN THE MINIMUM MODE AND MAXIMUM MODE OF OPERATION OF THE iAPX 86,88

* DEFINE THE STATE OF THE 8086 AFTER IT IS RESET

* RECOGNIZE THE SYMBOLS USED IN INTEL TIMING DIAGRAMS
CHAPTER 7

ARITHMETIC, LOGICAL AND CONDITIONAL INSTRUCTIONS

- ADD, SUB, MUL, DIV, CMP
- CONDITIONAL JUMPS
- AND, OR, XOR, NOT, TEST
LOGICAL INSTRUCTIONS

EXAMPLES

**AND**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1111</td>
<td>0000 1111</td>
<td></td>
</tr>
<tr>
<td>0000 1111</td>
<td>0000 1111</td>
<td></td>
</tr>
</tbody>
</table>

**OR**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1111</td>
<td>0000 1111</td>
<td></td>
</tr>
<tr>
<td>1001 1111</td>
<td>1001 1111</td>
<td></td>
</tr>
</tbody>
</table>

**XOR**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001 1111</td>
<td>0000 1111</td>
<td>NO CHANGE</td>
</tr>
<tr>
<td>1001 1111</td>
<td>1001 1111</td>
<td>(LOGIC 'AND')</td>
</tr>
</tbody>
</table>

**TEST**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001 1111</td>
<td>0000 1111</td>
<td></td>
</tr>
</tbody>
</table>

NOT (PRODUCES 1'S COMPLIMENT)

7-1

LOGICAL INSTRUCTIONS

* THE AND INSTRUCTION IS USED TO CLEAR BITS
  AND BX,1 ; MASK OUT ALL BITS BUT BIT 0

* THE TEST INSTRUCTION IS USED TO TEST BITS
  TEST CL,2
  JZ NOTSET ; TEST BIT 1 ('AND' CL WITH 00000010B)

* THE OR INSTRUCTION IS USED TO SET BITS
  OR DX,8000H ; SET THE MOST SIGNIFICANT BIT TO 1

* THE XOR INSTRUCTION COMPLEMENTS BITS
  XOR CX, 8000H ; COMPLEMENT HIGH ORDER BIT
  XOR DX,DX ; SET DX TO 0

* THE NOT INSTRUCTION COMPLEMENTS ALL BITS
  NOT AX ; COMPLEMENT THE AX REGISTER
ADDITION

ADD   DESTINATION, SOURCE
ADC   DESTINATION, SOURCE
INC   DESTINATION

DESTINATION = MEMORY OR REGISTER
SOURCE   = MEMORY, REGISTER OR IMMEDIATE DATA

*NO MEMORY TO MEMORY

EXAMPLES
ADD   SI,2
INC   BL
ADD   BX,DL ; ILLEGAL

ADDITION

ADDITION

7-3

ADDITION

7-4
SUBTRACTION

SUB DESTINATION, SOURCE
SBB DESTINATION, SOURCE
DEC DESTINATION
NEG DESTINATION ; FORMS 2'S COMPLEMENT
CMP DESTINATION, SOURCE ; ONLY FLAGS ARE AFFECTED

EXAMPLES
SUB CL, 20
DEC DL

MULTIPLICATION
(ALWAYS USES ACCUMULATOR)

\[
\begin{array}{c}
\text{AX} \\
\leftrightarrow \\
\text{AL} \\
\times \\
\text{8 BIT REGISTER OR MEMORY}
\end{array}
\]

\[
\begin{array}{c}
\text{DX} \\
\leftrightarrow \\
\text{AX} \\

\begin{array}{c}
\text{AX} \\
\times \\
\text{16 BIT REGISTER OR MEMORY}
\end{array}
\end{array}
\]
MULTIPLICATION

- UNSIGNED OPERATIONS
  MUL SOURCE

- SIGNED OPERATIONS
  IMUL SOURCE *

EXAMPLES:
  MUL BL ; AX = AL * BL
  IMUL DX ; DX, AX = AX * DX

* CAN BE IMMEDIATE DATA ON 186 BUT NOT 8086

DIVISION

8 BIT REGISTER OR MEMORY

AL ← AX ← AX \div

AH
REMAINDER

AX ← DX, AX ← AX \div

16 BIT REGISTER OR MEMORY

DX
REMAINDER

7-7

7-8
DIVISION

- UNSIGNED
  DIV SOURCE *

- SIGNED
  IDIV SOURCE *

- ALSO -
  - TO EXTEND SIGN BIT OF AL REGISTER INTO AH
    CBW
  - TO EXTEND SIGN BIT OF OF AX REGISTER INTO DX
    CWD

QUESTION: CBW AND CWD ARE USED WITH SIGNED NUMBERS. HOW DO YOU ACHIEVE THE SAME RESULT WITH UNSIGNED NUMBERS?

* CANNOT BE IMMEDIATE DATA

CLASS EXERCISE 7.1

AN 8 BIT FARENHEIT TEMPERATURE IN THE RANGE OF 40° TO 200° IS INPUT FROM THE SWITCHES (PORT 0). WRITE A PROGRAM TO CONVERT THE TEMPERATURE TO CELSIUS AND OUT THE CONVERTED TEMPERATURE TO THE LIGHTS (PORT 1).

USE THE FORMULA:

CELSIUS = ((FAREN.-32)x 5)/9

7-9

7-10
**CONDITIONAL JUMPS**

- CONDITIONAL JUMPS ARE USED TO TEST ONE OR MORE FLAGS
- ALL CONDITIONAL JUMPS ARE SHORT JUMPS
- THERE IS ONE SET OF JUMPS FOR USE WITH SIGNED NUMBERS AND ONE SET OF JUMPS FOR USE WITH UNSIGNED NUMBERS

---

**CONDITIONAL JUMPS FOR SIGNED OPERATIONS**

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>CONDITION</th>
<th>INTERPRETATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>JL OR JNGE</td>
<td>(SF XOR OF) = 1</td>
<td>&quot;LESS&quot; OR &quot;NOT GREATER&quot; OR &quot;EQUAL&quot;</td>
</tr>
<tr>
<td>JLE OR JNG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNL OR JGE</td>
<td>(SF XOR OF) = 0</td>
<td>&quot;NOT LESS&quot; OR &quot;GREATER OR EQUAL&quot;</td>
</tr>
<tr>
<td>JNLE OR JG</td>
<td>(SF XOR OF) OR ZF) = 0</td>
<td>&quot;NOT LESS OR EQUAL&quot; OR &quot;GREATER&quot;</td>
</tr>
<tr>
<td>JNO</td>
<td>OF = 1</td>
<td>&quot;OVERFLOW&quot;</td>
</tr>
<tr>
<td>JNS</td>
<td>SF = 0</td>
<td>&quot;NOT OVERFLOW&quot;</td>
</tr>
<tr>
<td>JNO</td>
<td>OF = 0</td>
<td>&quot;NOT OVERFLOW&quot;</td>
</tr>
<tr>
<td>JNS</td>
<td>SF = 0</td>
<td>&quot;NOT SIGN&quot;</td>
</tr>
</tbody>
</table>

7-12
### Conditional Jumps for Unsigned Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JB OR JNAE OR JC</td>
<td>CF = 1</td>
<td>&quot;Below&quot; OR &quot;Not Above OR Equal&quot;</td>
</tr>
<tr>
<td>JBE OR JNA</td>
<td>(CF OR ZF) = 1</td>
<td>&quot;Below OR Equal&quot; OR &quot;Not Above&quot;</td>
</tr>
<tr>
<td>JNB OR JAE OR JNC</td>
<td>CF = 0</td>
<td>&quot;Not Below&quot; OR &quot;Above OR Equal&quot;</td>
</tr>
<tr>
<td>JNBE OR JA</td>
<td>(CF OR ZF) ≠ 0</td>
<td>&quot;Not Below OR Equal&quot; OR &quot;Above&quot;</td>
</tr>
</tbody>
</table>

### Conditional Jumps for Signed and Unsigned Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE OR JZ</td>
<td>ZF = 1</td>
<td>&quot;Equal&quot; OR &quot;Zero&quot;</td>
</tr>
<tr>
<td>JP OR JPE</td>
<td>PF = 1</td>
<td>&quot;Parity&quot; OR &quot;Parity Even&quot;</td>
</tr>
<tr>
<td>JNE OR JNZ</td>
<td>ZF ≠ 0</td>
<td>&quot;Not Equal&quot; OR &quot;Not Zero&quot;</td>
</tr>
<tr>
<td>JNP OR JPO</td>
<td>PF ≠ 0</td>
<td>&quot;Not Parity&quot; OR &quot;Parity Odd&quot;</td>
</tr>
<tr>
<td>JCXZ</td>
<td>CX = 0</td>
<td>&quot;CX Register is Zero&quot;</td>
</tr>
</tbody>
</table>
CLASS EXERCISE 7.2

SUPPOSE WE HAVE AN IO DEVICE WHICH HAS A STATUS PORT (PORT 10) AND A DATA PORT (PORT 11).

WRITE A PROGRAM SEQUENCE THAT REPEATEDLY INPUTS FROM THE STATUS PORT UNTIL THE READY BIT BECOMES 1, THEN INPUTS FROM THE DATA PORT. IF THE UNSIGNED NUMBER OBTAINED IS LARGER THAN 43 THEN JUMP TO A LABEL CALLED ERROR.

FOR MORE INFORMATION ...

ASSEMBLY LANGUAGE INSTRUCTIONS
- CHAPTER 6, ASM86 LANGUAGE REFERENCE MANUAL
- CHAPTER 3, IAPX 86/88, 186/188 USER'S MANUAL

MULTIPRECISION ARITHMETIC
- APPENDIX G (EXAMPLES 6 & 7) ASM86 LANGUAGE REFERENCE MANUAL

RELATED TOPICS
THE 8086 PROVIDES A FULL SET OF ADJUST OPERATORS TO ALLOW FOUR FUNCTION ARITHMETIC ON BINARY CODED DECIMAL (BCD) OPERANDS. SEE APPENDIX E IN THE WORKSHOP NOTEBOOK, AND CHAPTER 6 IN THE ASM86 LANGUAGE REFERENCE MANUAL.
CHAPTER 8
DEFINING AND ACCESSING DATA

- DEFINING DATA
- INITIALIZING SEGMENT REGISTERS
- ADDRESSING MODES
DATA DEFINITIONS

ASSEMBLER DECLARATIVES ASSIGN STORAGE SPACE

DB - DEFINE BYTE
DW - DEFINE WORD
DD - DEFINE DOUBLE WORD
DQ - DEFINE QUAD WORD
DT - DEFINE TEN BYTES

8667 DATA TYPES

EXAMPLES:

<table>
<thead>
<tr>
<th>BYTE1</th>
<th>DB 3</th>
<th>;INITIALIZED BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE2</td>
<td>DB ?</td>
<td>;UNINITIALIZED BYTE</td>
</tr>
<tr>
<td>BYTE3</td>
<td>DB 0,7,8</td>
<td>;3 INITIALIZED BYTES</td>
</tr>
<tr>
<td>STRING</td>
<td>DB 'MESSAGE'</td>
<td>;7 INITIALIZED BYTES</td>
</tr>
<tr>
<td>ARRAY</td>
<td>DB 100 DUP(0)</td>
<td>;100 ZERED BYTES</td>
</tr>
<tr>
<td>WORD1</td>
<td>DW $300H</td>
<td>;$00 $03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;(LOW) (HIGH)</td>
</tr>
</tbody>
</table>

MEMORY ALLOCATION

ANGLE DW ?
TEMP DB ?
BARRAY DB 100 DUP (?)
DATA DEFINITION

* DATA IS TYPICALLY DEFINED IN A DATA SEGMENT

DATA_1            SEGMENT
XYZ  DB  ?
ALPHA  DW  ?
MESSAGE  DB  10 DUP (?)
DATA_1  ENDS

WHAT IS THE OFFSET OF THE FIRST BYTE IN MESSAGE?

WHY WOULD WE WANT DATA IN A SEPARATE SEGMENT FROM THE CODE?

ATTRIBUTES OF VARIABLES

* FOR EVERY DATA DEFINITION (VARIABLE), 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
INC  XYZ  ;BYTE OPERATION
INC  YYY  ;WORD OPERATION

WHAT ARE THE OFFSETS OF XYZ AND YYY?
CLASS EXERCISE 8.1

WRITE THE ASSEMBLER DIRECTIVES OR INSTRUCTIONS THAT WOULD:

1. DEFINE WAREA AS A WORD VARIABLE AND INITIALIZE IT TO 2000H.
2. DEFINE BAREA AS A BYTE VARIABLE AND DON'T INITIALIZE IT.
3. SET BAREA TO 10.
4. LOGICALLY 'AND' WAREA WITH 40H.
5. CHECK THE MSB (BIT 15) OF WAREA FOR A 1.

GENERATING ADDRESSES

\[ \text{ADDRESS} = \text{SEGMENT} + \text{OFFSET} \]

- THE ASSEMBLER DECIDES WHICH SEGMENT REGISTER TO USE.
- WHICH SEGMENT REGISTER IS NORMALLY USED TO ACCESS DATA?
- HOW DOES THE ASSEMBLER KNOW WHICH SEGMENT REGISTER IT CAN USE?
ASSUME DECLARATIVE

* THE ASSUME DECLARATIVE TELLS THE ASSEMBLER WHICH SEGMENT REGISTER IS SUPPLYING VALUE FOR THE INSTRUCTION'S DATA ACCESS.

**EXAMPLE**

```asm
DATA_1 SEGMENT
XYZ DB ?
DATA_1 ENDS

CODE_1 SEGMENT
ASSUME DS:DATA_1,CS:CODE_1

CODE_1 ENDS

MOV XYZ,10H
```

XYZ IS IN THE SEGMENT DATA_1, WHICH SEGMENT REGISTER IS POINTING AT DATA_1? THE ASSUME TELLS THE ASSEMBLER DS.

**INITIALIZING SEGMENT REGISTERS**

* THE ASSUME DECLARATIVE IS JUST A PROMISE TO THE ASSEMBLER. IT DOES NOT INITIALIZE THE SEGMENT REGISTER.

- TO WHAT VALUE SHOULD DS BE SET?
- HOW DOES THE SEGMENT REGISTER GET INITIALIZED?

```
DATA_1 SEGMENT
XYZ DB ?
DATA_1 ENDS

CODE_1 SEGMENT
ASSUME DS:DATA_1,CS:CODE_1

; THERE IS NO MOVE IMMEDIATE TO THE ; SEGMENT REGISTER
```

CPU

DS ?

8-7

8-8
ADDRESSING MODES

* THE 8088 PROVIDES SEVERAL WAYS TO ACCESS MEMORY
  - DIRECT
  - INDIRECT
  - INDEXED
  - BASED
  - BASED INDEXED
  - BASED INDEXED AND DISPLACEMENT

* THESE ADDRESSING MODES ARE PROVIDED TO SUPPORT DIFFERENT TYPES OF DATA STRUCTURES.

* DIFFERENT ADDRESSING MODES ARE THE DIFFERENT WAYS AN INSTRUCTION CAN SPECIFY AN OFFSET:

\[
\text{OFFSET} = \text{VARIABLE NAME} + \left[ \begin{array}{c} \text{BX} \\ \text{BP} \end{array} \right] + \left[ \begin{array}{c} \text{DI} \\ \text{SI} \end{array} \right] + \text{DISPLACEMENT}
\]
ADDRESSING MODES

MOV AX, MVAR DIRECT OFFSET = VARIABLE NAME
MOV AX, [BX] INDIRECT OFFSET = [BX]
MOV AX, MVAR [SI] INDEXED OFFSET = VARIABLE NAME + [SI]
MOV AX, [BX] + 5 BASED OFFSET = [BX] + 5
MOV AX, [BX][DI] BASED INDEXED OFFSET = [BX] + [DI]
MOV AX, [BP+SI+15] BASED INDEXED AND DISPLACEMENT OFFSET = [BX]+[SI]+DISPLACEMENT

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

ADDRESSING SIMPLE VARIABLES

* TO ACCESS A SINGLE SIMPLE VARIABLE, THE NAME OF THE VARIABLE IS USED.

EXAMPLE:

LOC OBJ LINE SOURCE
1 NAME DEMO
2 --- SEGMENT
0000 ?? 4 XYZ DB ?
0001 0020 5 BETA DW 2000H
6 DATA_1 ENDS
7 --- SEGMENT
8 CODE_1 ENDS
9 ASSUME CS:CODE_1,DS:DATA_1
10 0000 B8----- R 11 START: MOV AX,DATA_1
12 MOV DS,AX
13 0003 80DB 14 MOV XYZ,10H ;MOV 10H INTO MEMORY LOCATION
15 MOV DS,XYZ ;DS:XYZ
16 0005 C606000010 17 AND XYZ,AL ;AND LOCATION DS:XYZ WITH AL
18 19 000A 20060000 20 MOV BX,BETA ;MOV CONTENTS OF BETA INTO BX
21 22 23 CODE_1 ENDS

* OFFSET = VARIABLE NAME
ARRAYS

* THE 8086,88 HARDWARE AND ASSEMBLER SUPPORT THE REPRESENTATION OF SINGLE DIMENSIONED ARRAYS.

* AN ARRAY IS A COLLECTION OF OBJECTS ALL OF THE SAME TYPE

EXAMPLE: A BYTE ARRAY V

IN MEMORY:

<table>
<thead>
<tr>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>V(0)</td>
</tr>
<tr>
<td>V(1)</td>
</tr>
<tr>
<td>V(2)</td>
</tr>
<tr>
<td>V(3)</td>
</tr>
<tr>
<td>V(4)</td>
</tr>
<tr>
<td>V(5)</td>
</tr>
<tr>
<td>V(6)</td>
</tr>
<tr>
<td>V(7)</td>
</tr>
<tr>
<td>V(8)</td>
</tr>
</tbody>
</table>

IN ASSEMBLY LANGUAGE:

DATA_1 SEGMENT
V DB 10 DUP (?)
DATA_1 ENDS

ACCESSING ARRAYS

* THE ELEMENTS OF THE ARRAY ARE ACCESSED BY USING AN INDEX (SUBSCRIPT)

EXAMPLE:

MOV AL, V[1] ; FETCH THE SECOND BYTE OF V

* OFFSET = VARIABLE NAME + [DISPLACEMENT]
ACCESSING ARRAYS (INDEXED ADDRESSING)

* IN GENERAL V[i] REPRESENTS THE ith ELEMENT OF THE ARRAY. THE INDEX (SUBSCRIPT) CAN BE IN AN INDEX REGISTER OR A BASE REGISTER.

EXAMPLE: TO ACCESS V[i]

```
MOV SI, 8
MOV AL, V[SI]  ; (BX, BP, SI, OR DI ONLY)
```

* OFFSET = VARIABLE NAME + [SI]

* ALL INDEXING IS ON A BYTE LEVEL

EXAMPLE

PROBLEM

AN 8 BIT VALUE REPRESENTING A TEMPERATURE IN THE RANGE 0°C TO 50°C IS IN A MEMORY LOCATION SYMBOLICALLY CALLED "CTEMP". IT IS TO BE CONVERTED TO FAHRENHEIT USING A TABLE OF FAHRENHEIT TEMPERATURE VALUES STORED IN ROM MEMORY STARTING AT A LOCATION SYMBOLICALLY CALLED "CTABLE". THE FIRST TABLE ENTRY IS THE TEMPERATURE VALUE CORRESPONDING TO 0°C. EACH SUCCESSIVE ENTRY CORRESPONDS TO AN INTEGRAL CELSIUS DEGREE 1°C, 2°C, ..., 50°C. THE CONVERTED VALUE IS TO BE STORED AT A BYTE LOCATION CALLED "FTEMP".
EXAMPLE

* IN MEMORY "CTABLE" APPEARS

CTABLE (0) 32°F
CTABLE (1) 33°F
CTABLE (2) 36°F
CTABLE (50) 122

EXAMPLE

SOLUTION

THE VALUE IN CTEMP DEFINES WHERE IN CTABLE THE CORRESPONDING FAHRENHEIT VALUE CAN BE FOUND. THE VALUE IN CTEMP IS LOADED INTO AN INDEX REGISTER AND IS USED AS AN INDEX INTO CTABLE. CTABLE INDEXED BY THE REGISTER IS STORED INTO FTEMP.
CLASS EXERCISE 8.2

* ASSUME THERE IS AN ARRAY OF EMPLOYEE PAYSCALES. ASSUME THERE ARE 100 EMPLOYEES AND 1 BYTE IS NEEDED TO REPRESENT EACH EMPLOYEE'S PAYSCALE. WRITE A PROGRAM THAT ADDS 50 DOLLARS TO EACH EMPLOYEE'S PAYSCALE. USE THE NECESSARY DECLARATIVES TO SET ASIDE MEMORY FOR THE ARRAY AND TO WRITE THE PROGRAM.
FOR MORE INFORMATION . . .

DEFINING DATA
- CHAPTER 3, ASM86 LANGUAGE REFERENCE MANUAL

ACCESSING DATA AND ADDRESSING MODES
- CHAPTER 3, iAPX 86/88, 186/188 USER'S MANUAL
- CHAPTER 4, ASM86 LANGUAGE REFERENCE MANUAL

ASSUME DECLARATIVE
- CHAPTER 2, ASM86 LANGUAGE REFERENCE MANUAL

RELATED TOPICS ...

ASM86 LETS YOU DEFINE VERY COMPLEX DATA ITEMS USING STRUCTURES (A COLLECTION OF DISSIMILAR DATA ITEMS) AND RECORDS (VARIABLE BIT LENGTH FIELDS). USING "HIGH LEVEL" DATA ITEMS SUCH AS STRUCTURES AND RECORDS WILL IMPROVE THE DOCUMENTATION AND RELIABILITY OF YOUR PROGRAMS. READ CHAPTER 3 OF THE ASM86 LANGUAGE REFERENCE MANUAL. CODE EXAMPLES ARE IN CHAPTER 3 OF THE iAPX 86/88, 186/188 USER'S MANUAL.
CHAPTER 9
PROGRAM DEVELOPMENT II

- DEBUG-86
- ASM86
- SUBMIT FILES
SERIES III ENVIRONMENT

ISIS-II SERIES-II 8086 BASED

AEDIT

ISIS-II UTILITIES COPY, DIR, DELETE ...

8080/88 PROGRAMS

SERIES-III 8086-BASED

ASMX86

LINK 86

LOCATE 86

CRTL-D

SERIES III DEBUGGER

* ALLOWS SYMBOLIC DEBUGGING OF 8086,88 PROGRAMS
* DOWNLOADS YOUR 86,88 PROGRAM FROM A DISK FILE
* ALLOWS REAL-TIME EXECUTION OF PROGRAMS
* ALLOWS SINGLE STEP EXECUTION OF PROGRAMS
* DISPLAY AND ALTERATION OF 86,88 REGISTERS, MEMORY LOCATIONS, AND I/O PORTS
* DISASSEMBLE PROGRAMS IN MEMORY
SAMPLE PROGRAM TO BE EXECUTED/DEBUGGED USING DEBUG-86

SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE DEMO
OBJECT MODULE PLACED IN :F1:DEMO.OBJ
ASSEMBLER INVOKED BY: ASM86.86 :F1:DEMO.ASM DEBUG SYMBOLS

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>NAME DEMO</td>
</tr>
<tr>
<td>0000</td>
<td>0100</td>
<td>3</td>
<td>DATA SEGMENT</td>
</tr>
<tr>
<td>0002</td>
<td>(10)</td>
<td>5</td>
<td>ARRAY DB 10 DUP(?)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>BB----</td>
<td>11</td>
<td>START: MOV AX,DATA ;INITIALIZE DS</td>
</tr>
<tr>
<td>0003</td>
<td>B8DB</td>
<td>12</td>
<td>MOV DS,AX</td>
</tr>
<tr>
<td>0005</td>
<td>33F6</td>
<td>13</td>
<td>XOR SI,SI ;ARRAY POINTER</td>
</tr>
<tr>
<td>0007</td>
<td>B9A00</td>
<td>14</td>
<td>MOV CX,LENGTH ARRAY</td>
</tr>
<tr>
<td>000A</td>
<td>BB160000</td>
<td>15</td>
<td>MOV DX,WVAR ;GET ADDRESS OF PORT</td>
</tr>
<tr>
<td>000E</td>
<td>EC</td>
<td>16</td>
<td>AGAIN: IN AL,DX ;INPUT THE VALUE</td>
</tr>
<tr>
<td>000F</td>
<td>B8402</td>
<td>17</td>
<td>MOV ARRAY[SI],AL ;AND SAVE IN ARRAY</td>
</tr>
<tr>
<td>0012</td>
<td>46</td>
<td>18</td>
<td>INC SI</td>
</tr>
<tr>
<td>0013</td>
<td>2EF9</td>
<td>19</td>
<td>LOOP AGAIN ;DO IT 10 TIMES</td>
</tr>
<tr>
<td>0015</td>
<td>EBE9</td>
<td>20</td>
<td>JMP START ;REPEAT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21</td>
<td>CODE ENDS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22</td>
<td>END START</td>
</tr>
</tbody>
</table>

USING THE SERIES III DEBUGGER

* TO INVOKE THE SERIES III DEBUGGER

1. POWER ON DEVELOPMENT SYSTEM
2. INVOKE ISIS-II
   -INSERT SYSTEM DISK INTO DRIVE 0
   -PRESS RESET ON MDS FRONT PANEL ISIS
   WILL SIGN ON
   ISIS-II V x.y.

3. ON DEVELOPMENT SYSTEM TYPE:
   RUN DEBUG
   DEBUGGER WILL SIGN ON:
   DEBUG 8086 V x.y.
   #
USING THE DEBUGGER

* THE SERIES III DEBUGGER CAN EXECUTE/DEBUG ABSOLUTE (LOCATED)
86,88 OBJECT CODE OR LOAD TIME LOCATABLE (LINKED WITH "BIND")
86,88 OBJECT CODE

EXAMPLE: TO LOAD DEMO PROGRAM AND ITS SYMBOLS FROM
DRIVE 1 OF MDS:

* LOAD :F1:DEMO

USING THE DEBUGGER

* DISPLAY COMMANDS

- TO DISPLAY ALL REGISTERS:
  * REGISTER
    RAX=0000H RBX=0000H RCX=0000H RDX=0000H SP=4000H BP=4E50H SI=0000H DI=0000H
    CS=0483H DS=0000H SS=0000H ES=0000H RF=0200H IP=0000H

- TO DISPLAY/CHANGE ONE REGISTER USE THE NAME SPECIFIED
  IN THE DISPLAY.
  * RAX
  RAX=0000H
  * SP=100
  * RBX=50
**USING THE DEBUGGER**

*DISPLAY COMMANDS*

- TO DISPLAY/CHANGE MEMORY USE THE TYPE (BYTE, WORD) WITH AN ADDRESS OR SYMBOLIC NAME.

- **BYTE .ARRAY**
  
  BYT 0481:0002H=FBH

- **WORD .WVAR**
  
  WOR 0481:0000H=0001H

- **BYTE .ARRAY LENGTH 10T**
  
  BYT 0481:0002H=FBH B8H E3H OFH 50H B8H ECH ODH 50H E8H

- **BYTE .ARRAY=FF,00,FF,00,FF,00,FF,00,FF,00**

- **BYTE .ARRAY TO .ARRAY+9**
  
  BYT 0481:0002H=FFH 00H FFH 00H FFH 00H FFH 00H

---

**USING THE DEBUGGER**

*DISPLAY COMMANDS*

- TO DISPLAY INSTRUCTIONS USE THE DISSASSEMBLER WITH AN ADDRESS OR SYMBOLIC NAME.

```
*ASM .START LENGTH 17
ADDR  PREFIX  MNEUMONIC  OPERANDS  COMMENTS
0483:0000H  MOV   AX,0481H
0483:0003H  MOV   DS,AX
0483:0005H  XOR   SI,SI
0483:0007H  MOV   CX,000AH
0483:0008H  MOV   DX,WORD PTR [0000H]
0483:000AH  IN    AL,DX
0483:000BH  MOV   BYTE PTR [SI]+[02H],AL
0483:0012H  INC   SI
0483:0013H  LOOP  $-05H  ;SHORT
0483:0015H  JMP   $-15H  ;SHORT

*ASM CS:IP TO CS:IP+16
ADDR  PREFIX  MNEUMONIC  OPERANDS  COMMENTS
0483:0000H  MOV   AX,0481H
0483:0003H  MOV   DS,AX
0483:0005H  XOR   SI,SI
0483:0007H  MOV   CX,000AH
0483:0008H  MOV   DX,WORD PTR [0000H]
0483:000AH  IN    AL,DX
0483:000BH  MOV   BYTE PTR [SI]+[02H],AL
0483:0012H  INC   SI
0483:0013H  LOOP  $-05H  ;SHORT
0483:0015H  JMP   $-15H  ;SHORT
```

9-7

9-8
USING THE DEBUGGER

DISPLAY COMMANDS

- TO DISPLAY/CHANGE I/O PORTS

* PORT 0
POR 0000H=66H

* PORT 0 LENGTH 2
POR 0000H=55H 01H

* WPORT 1000
WPO 1000H=00FFH

* PORT 0=FF

- PROGRAM EXECUTION COMMANDS

- TO EXECUTE THE PROGRAM WITH NO BREAKPOINTS USE THE GO COMMAND * GO FROM .START FOREVER

- TO STOP THE PROGRAM USE THE CTRL-D KEY. THE NEXT INSTRUCTION IS DISPLAYED

* GO FROM .START TILL .AGAIN

0483:0012H INC SI
PROCESSING ABORTED

* TO EXECUTE FROM THE BEGINNING UNTIL THE OUT INSTRUCTION IS EXECUTED: THE DEBUGGER DISPLAYS THE INSTRUCTION AT THE BREAKPOINT

* GO FROM .START TILL .AGAIN

0483:000EH IN AL,DX

* TO EXECUTE UP TO THE INSTRUCTION AT LABEL START

* GO TILL .START

0483:0000H MOV AX,0481H

*
USING THE DEBUGGER

PROGRAM EXECUTION COMMANDS

- TO EXECUTE ONE INSTRUCTION AND SEE THE NEXT INSTRUCTION.
  * STEP FROM .AGAIN
  0483:000FH MOV BYTE PTR [SI][+02H],AL

- TO EXECUTE THAT INSTRUCTION AND DISPLAY THE NEXT.
  * STEP
  0483:0012H INC SI

THERE ARE ADVANCED COMMANDS THAT YOU CAN USE AFTER MASTERING THESE.

USING THE DEBUGGER

* FINISHING UP
  - TO EXIT THE DEBUGGER TYPE:
    EXIT
    OR
    CNTRL-C

* ONCE A PROGRAM IS DEBUGGED, IT CAN BE LOADED AND EXECUTED BY TYPING:
  RUN :DRIVENUMBER: FILENAME.

* THE DEBUGGER CAN BE INVOKED DURING EXECUTION BY TYPING CNTRL-D.
  THE DEBUGGER CAN BE ABORTED BY TYPING CNTRL-C.
SERIES III ENVIRONMENT

ISIS-II
SERIES-II
8086 BASED

AEDIT

ISIS-II
UTILITIES
COPY, DIR,
DELETE ...

8080/86
PROGRAMS

SERIES-III
8086-BASED

EXIT

AEX

AEX

SERIES-III
8086-BASED

CRTL
C

CRTL-D

8086
PROGRAMS

LINK 86

LOCATE 86

8086/8088 ASSEMBLER CONTROLS

-RUN ASM86 :F1:LAB.ASM OPTIONS

PRIMARY CONTROLS

OBJECT (FILENAME)
NOOBJECT
PRINT(Filename)
NOPRINT
PAGING/NOPAGING
SYMBOLS/NOSYMBOLS
ERRORPRINT(Filename)
DEBUG/NODEBUG

* OJ
* PR
* PI/NOPH
* SB/Noss
* EP/NOEP
* DB/Nosb

CONTROL CREATION AND DESTINATION OF .OBJ FILE
CONTROL CREATION AND DESTINATION OF .LST FILE
PAGINATE/DON'T PAGINATE LISTING
APPEND/DON'T APPEND SYMBOL TABLE TO LISTING
SEND ERRORS TO DEVICE SPECIFIED/DON'T REPORT ERRORS
APPEND/DON'T APPEND SYMBOL TABLE TO OBJECT FILE

* DEFAULT
8086/8088 ASSEMBLER CONTROLS (CONT’D)

GENERAL CONTROLS

LIST/NOLIST  LI NOLI  INCLUDE ALL LINES FOLLOWING IN LISTING FILE
SUSPEND LISTING

EJECT  EJ  FORCE A FORM FEED (OVERRIDEN BY NO PAGING)

INCLUDE(Filename)  IC(Filename)  LINES FROM SPECIFIED FILE ARE INCLUDED IN SOURCE FILE

* DEFAULT

8086/8088 ASSEMBLER CONTROLS (CONT’D)

CONTROLS CAN BE SPECIFIED EITHER:

- AT INVOCATION
  - RUN ASM86 :FI:EXMPL.ASM DEBUG SYMBOLS PRINT(:LP:)

OR

- IMBEDDED IN SOURCE FILE
  - $DEBUG SYMBOLS NAME EXAMPLE
  - $INCLUDE(:FI:EQUS.SRC)
  - $EJECT
  - CODE SEGMENT

PRIMARY CONTROLS MUST BE ON FIRST LINE OF SOURCE FILE
ASSEMBLER FEATURES

* ASM86 HAS SOME BUILT-IN OPERATORS TO AID IN PROGRAMMING
   (THEY MAKE A PROGRAM MORE READABLE AND RELIABLE)

TYPE – RETURNS TYPE OF DATA DEFINITION

- DB 1 BYTE
- DW 2 BYTES
- DD 4 BYTES

LENGTH – RETURNS NUMBER OF UNITS
SIZE – RETURNS NUMBER OF BYTES

EXAMPLE

<table>
<thead>
<tr>
<th>ARRAY</th>
<th>DW 100 DUP(?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>SI,TYPE ARRAY :ADJUST SI TO NEXT ELEMENT</td>
</tr>
<tr>
<td>MOV</td>
<td>CX,LENGTH ARRAY :LOADS CX WITH 100</td>
</tr>
<tr>
<td>MOV</td>
<td>DI,SIZE ARRAY :LOADS SI WITH 200</td>
</tr>
</tbody>
</table>

SERIES III DEVELOPMENT STEPS

- AEDIT :F1:LAB1.ASM
- RUN ASM86 :F1:LAB1.ASM DEBUG
- COPY :F1:LAB1.LST TO :LP:
- RUN LINK86 :F1:LAB1.OBJ BIND
- RUN DEBUG
* LOAD :F1:LAB1

COMPOSE SOURCE PROGRAM
ASSEMBLE PROGRAM
COPY ASSEMBLER OUTPUT LIST FILE TO THE PRINTER
PRODUCE LOAD TIME LOCATABLE CODE FOR EXECUTION ON SERIES III
INVOCER DEBUGGER
LOAD PROGRAM AND DEBUG
YOU WILL PROBABLY HAVE TO EXECUTE
SOME OF THESE STEPS A FEW TIMES
BEFORE YOUR PROGRAM EXECUTES
AS YOU WANT IT.

WOULDN'T IT BE NICE IF YOU DIDN'T HAVE TO TYPE
ALL THOSE COMMANDS EACH TIME?

SUBMIT FILES
ISIS II LETS YOU PUT COMMANDS IN A DISK FILE
TO BE EXECUTED AUTOMATICALLY.
FOR EXAMPLE

WE COULD USE AEDIT TO CREATE A SUBMIT FILE CALLED :F1:SBMT.CSD

RUN ASM86 :F1:LAB1.ASM DB PR(:LP:)

RUN LINK86 :F1:LAB1.OBJ BIND

THIS WOULD GIVE US THE COMMANDS REQUIRED TO:

- ASSEMBLE OUR PROGRAM
- DUMP THE LISTING TO THE LINE PRINTER
- MAKE IT "RUN TIME LOCATED"
IF THERE WERE ERRORS IN THE ASSEMBLY, WE WOULD LIKE TO TAKE CONTROL. EDIT THE FILE AND ASSEMBLE IT AGAIN BEFORE LINKING.

TO TURN CONTROL OF THE SYSTEM OVER TO THE CONSOLE IN A SUBMIT FILE, ADD \E (CTRL-E) COMMAND TO THE SUBMIT FILE.

IN AEDIT COMMAND MODE
1) POSITION CURSOR
2) TYPE H I Ø5 <CR>

:FI:SBMT.CSD (CONT'D)

RUN ASM86 :FI:LAB1.ASM DB PR(:LP:)
\E <--- ALLOWS YOU TO EDIT YOUR MISTAKE AND RETYPE THE ASM86 COMMAND IF THERE WAS AN ERROR. TO GET BACK TO SUBMIT FILE, TYPE A \E WHICH WILL EXECUTE THE LINK86 COMMAND.

RUN LINK86 :FI:LAB1.OBJ BIND
INVOKING A SUBMIT FILE

IF THE SUBMIT FILE WAS THE DEFAULT .CSD EXTENSION,

ENTER:

- SUBMIT :F1:SBMT

PASSING PARAMETERS

USE % N (WHERE N=0 TO 9) IN THE SUBMIT FILE

RUN ASM86 :%0:%1.ASM DB SB
RUN LINK86 :%0:%1.OBJ BIND

EXAMPLES:
SUBMIT :F1:SBMT (F1,LAB5)
SUBMIT :F1:SBMT (F2,LAB3)
CLASS EXERCISE 9.1

WRITE SUBMIT FILE WHICH WILL:

A. ASSEMBLE A PROGRAM WHOSE SOURCE IS CALLED PROBLEM ON A DISK IN DRIVE 1
B. ADD A SYMBOL TABLE TO THE LISTING
C. ADD A SYMBOL TABLE TO THE OBJECT FILE
D. PUT THE LIST FILE ON THE DISK IN DRIVE 1 UNDER THE NAME LISTING
E. PRODUCE A "RUN-TIME LOCATABLE" PROGRAM

FOR MORE INFORMATION ...

DEBUG – 86
- CHAPTER 6, INTELLEC SERIES III M.D.S. CONSOLE OPERATING INSTRUCTIONS

ASM86 (CONTROLS AND OPTIONS)
- CHAPTER 3, ASM86 MACRO ASSEMBLER OPERATING INSTRUCTIONS

ASM86 ERRORS AND RECOVERY
- APPENDIX A, ASM86 MACRO ASSEMBLER OPERATING INSTRUCTIONS

RESERVED WORDS (ASM86)
- APPENDIX C, ASM88 LANGUAGE REFERENCE MANUAL

RELATED TOPICS...
ASM88 SUPPORTS USER DEFINED TEXT MACROS INCLUDING CONDITIONAL ASSEMBLY. SEE CHAPTER 7 OF THE ASM88 LANGUAGE REFERENCE MANUAL.

IT IS POSSIBLE TO MODIFY THE OPERATION OF THE ASSEMBLER TO CHANGE MNEMONICS, DEFAULT CONDITIONS, ETC. THIS ADVANCED TOPIC IS DISCUSSED IN APPENDIX A OF THE ASM86 LANGUAGE REFERENCE MANUAL.
CHAPTER 10
BASIC CPU DESIGN AND TIMING

- MINIMUM MODE
- MAXIMUM MODE
- INSTRUCTION QUEUE
- 8086, 8088, 8284A, 8288, 8286, 8282
THE iAPX 86,88 SYSTEM

* FLEXIBLE PROCESSOR SYSTEM
  - TWO OPERATING MODES
  - ARCHITECTURE SUPPORTS MULTIPROCESSING AND COPROCESSING
  - MEGABYTE MEMORY ADDRESS SPACE
  - 16 BIT DATA BUS (8 OR 16 BIT DATA)
  - INSTRUCTION PREFERENCE QUEUE

iAPX 86,88 ARCHITECTURE (MINIMUM MODE)

- MINIMUM MODE DESIGNED FOR SMALL SYSTEMS
- CONTROL SIGNALS TO MEMORY AND IO SUPPLIED DIRECTLY BY CPU
- USED IN SINGLE PROCESSOR SYSTEMS ONLY

![Diagram of 8284A CPU connected to MIN/MAX with CONTROL BUS](image)
iAPX 86,88 ARCHITECTURE (MAXIMUM MODE)

* MAXIMUM MODE DESIGNED FOR LARGE SYSTEMS

* 8288 BUS CONTROLLER DECODES STATUS SIGNALS TO GENERATE CONTROL SIGNALS

* CPU USES CONTROL PINS FREED BY 8288 TO COORDINATE OTHER PROCESSORS

---

8086, 88 CPU SET AND BUS STRUCTURE
MINIMUM AND MAXIMUM MODE

8086  -  5 MHZ
8086-4  -  4 MHZ
8086-2  -  8 MHZ
8086-1  -  10 MHZ
8086, 88 BASIC BUS CYCLE

8284A CLOCK GENERATOR

- Generates system clock for 8086/8088
- Uses crystal or TTL signal for frequency source
- Provides local ready and multibus ready synchronization
- Generates system reset output from Schmitt trigger input
8284A BLOCK DIAGRAM

RESET CAPTURE LOGIC

CLOCK GENERATOR
33% DUTY CYCLE

READY SYNCHRONIZER
GUARANTEES SETUP
REQUIREMENTS FOR 8086

8284A TIMING

EFI/OSC
CLK
PCLK
RDY
AEN
READY
RESET
RESET-SUPPLIED BY 8284A CLOCK GENERATOR

FLAGS ← 0
CS ← FFFF
IP,D8,88,ES ← 0

+5V

8284A
RES

8086, 88
RESET

READY

• READY IS SYNCHRONIZED WITH THE CPU BY THE CLOCK GENERATOR
• READY IS USED TO EXTEND A BUS CYCLE BY ONE OR MORE CLOCK CYCLES
• INCREASES THE AMOUNT OF TIME THAT CPU GIVES MEMORY TO RESPOND WITH OR ACCEPT DATA
• THE USER MUST DESIGN THE HARDWARE WHICH DECODES THE BUS ADDRESS AND DETERMINES IF "WAIT STATES" ARE REQUIRED.
• THE 8284A HAS 2 RDY-AEN INPUTS WHICH ALLOWS YOU TO DEVELOP TWO DIFFERENT WAIT STATE PERIODS.
SINGLE WAIT STATE GENERATOR

BUS CYCLE WITH WAIT STATES
8086 SIGNAL DESCRIPTION (CONTROL SIGNALS)

**DT/R** - CONTROLS DIRECTION OF DATA THROUGH TRANSCEIVER

**DEN** - OUTPUT ENABLE FOR TRANSCEIVER

**RD, WR** - INDICATES A READ OR WRITE CYCLE TO/FROM MEMORY OR I/O

**M/iO** - INDICATE WHETHER READ OR WRITE IS TO MEMORY OR I/O

---

**READ TIMING MINIMUM MODE**

- **CLK**
- **M/iO**
- **ALE**
- **AD_{15-0}**
- **RD**
- **DT/R**
- **DEN**

Timing intervals: **T_1**, **T_2**, **T_3**, **T_w**, **T_4**, **T_{4} OF NEXT CYCLE**

- **AD_{15-0}** ADDRESS A_{15-0} FLOAT
- **DATA IN D_{15-0}** FLOAT

---

10-15

10-16
HOLD AND HLDA

- HOLD FORCES THE CPU TO RELEASE CONTROL OF THE BUSSES AFTER THE CURRENT BUS CYCLE
- HLDA INDICATES THAT THE CPU HAS TRI-STATE THE BUSSES

HOLD AND HLDA ARE USED BY DMA DEVICES TO "BORROW" BUS CYCLES FOR THEIR DATA TRANSFERS

8088 PIN DIAGRAM (MINIMUM MODE)

DIFFERENCES IN PINOUT FROM 8086:
• NO BHE PIN: SSO ALONG WITH IO/M AND DT/R PROVIDE MACHINE CYCLE STATUS IN MIN MODE
• PIN 28 IS IO/M RATHER THAN M/IO TO BE COMPATIBLE WITH THE 8085
• A8 - A15 NOT MULTIPLEXED WITH DATA
MIN/MAX SELECTION

MIN/MAX - MINIMUM OR MAXIMUM CONFIGURATION STRAPPING OPTION THAT ALTERS THE FUNCTIONS OF 8 OF THE CPU PINS AS FOLLOWS:

<table>
<thead>
<tr>
<th></th>
<th>MINIMUM</th>
<th>MAXIMUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR</td>
<td>LOCK</td>
<td></td>
</tr>
<tr>
<td>INTA</td>
<td>QS₁</td>
<td></td>
</tr>
<tr>
<td>ALE</td>
<td>QS₀</td>
<td></td>
</tr>
<tr>
<td>M/IO</td>
<td>S₀</td>
<td></td>
</tr>
<tr>
<td>DT/R</td>
<td>S₁</td>
<td></td>
</tr>
<tr>
<td>DEN</td>
<td>S₂</td>
<td></td>
</tr>
<tr>
<td>HLDA</td>
<td>RQ/GT₀</td>
<td>RQ/GT₁</td>
</tr>
<tr>
<td>HOLD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MINIMAX 8086, 88
8086 SIGNAL DESCRIPTION (MAXIMUM MODE)

\( S_2, S_1, S_0 \) - STATUS LINES THAT INFORM THE 8288 OF THE TYPE OF BUS CYCLE THAT THE 8076 IS RUNNING

<table>
<thead>
<tr>
<th>( S_2 )</th>
<th>( S_1 )</th>
<th>( S_0 )</th>
<th>SIGNAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>INTA</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>I/O READ</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>I/O WRITE</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>HALT</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CODE ACCESS</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>READ MEMORY</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>WRITE MEMORY</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>PASSIVE</td>
</tr>
</tbody>
</table>

8086, 88
8288 TIMING

The AMWC, AIOWC are provided to generate longer strobes required by some memories. They should not be used with devices that latch data on the leading edge of the strobe since data is not guaranteed to be valid at that time.

8086, 88 CPU BLOCK DIAGRAM

- Two independent units: EU and BIU
- BIU reads data and instructions
- EU executes instructions
- Speeds execution by overlapping instruction fetches with execution
INSTRUCTION PREFETCH QUEUE

- DATA ACCESSES HAVE PRIORITY OVER INSTRUCTION FETCHES
- QUEUE "FLUSHES" AUTOMATICALLY ON JMP
- QUEUE IS 6 BYTES IN 8086, 4 BYTES IN 8088

INVISIBLE TO USER (ALMOST)

PROGRAM TIMING

- IT IS NOT PRACTICAL TO CALCULATE EXACT PROGRAM EXECUTION TIME
  - EXECUTION TIME CAN BE MEASURED WITH A TIMER SUCH AS PROVIDED ON ICE86
  - PROBABLE WORST CASE CAN BE ESTIMATED BY ASSUMING A MINIMUM INSTRUCTION TIME OF 4 CLOCK CYCLES
OUR DESIGN EXAMPLE

iSBC 86/05 SINGLE BOARD COMPUTER

• 8 MHZ 8086 CPU
• 8K BYTES STATIC RAM (EXPANDABLE)
• SOCKETS FOR 32K BYTES ROM (EXPANDABLE)
• 1 SERIAL IO PORT, 3 PARALLEL IO PORTS
• 2 iSBX CONNECTORS
• MULTIBUS COMPATIBLE
• FLEXIBLE DESIGN
CLASS EXERCISE 10.1

1.) IS THIS 8086 IN MINIMUM MODE OR MAXIMUM MODE?

2.) AS CONFIGURED WHAT SPEED WILL THIS 8086 RUN AT?

3.) THERE IS A JUMPER SHOWN AS E181–E182 JUST TO THE LEFT OF THE 8284A. WHAT EFFECT WILL THE REMOVAL OF THIS JUMPER HAVE?

FOR MORE INFORMATION . . .

8086 CPU SET AND OPERATION
-AP-67, 8086 SYSTEM DESIGN APPLICATION NOTE

ISBC 86/05 SINGLE BOARD COMPUTER
-ISHBC 86/05 SINGLE BOARD COMPUTER HARDWARE REFERENCE MANUAL
DAY THREE OBJECTIVES

BY THE TIME YOU FINISH TODAY YOU WILL:

* LIST THE PERIPHERALS AND THEIR FUNCTIONS THAT ARE INCLUDED IN THE iAPX 186,188

* DESCRIBE THE OPERATION OF THE ADDED INSTRUCTIONS TO THE iAPX 186,188

* WRITE A PROCEDURE USING THE PROPER ASSEMBLER DIRECTIVES

* WRITE A PROCEDURE THAT COULD BE CALLED FROM A PL/M PROGRAM WHICH REQUIRES PARAMETERS

* WRITE THE CHANGES REQUIRED TO ELIMINATE FORWARD REFERENCING ERRORS IN A MULTIPLE SEGMENTED PROGRAM

* WRITE AN INTERRUPT SERVICE ROUTINE AND THE ASSEMBLER DIRECTIVES REQUIRED TO CREATE THE PROPER INTERRUPT POINTER TABLE ENTRY
CHAPTER 11
PROCEDURES

• PROCEDURES DEFINITION
• STACK CREATION AND USAGE
• PARAMETER PASSING
• EXAMPLE
PROCEDURES

* SECTIONS OF A PROGRAM THAT ARE CALLED AND RETURNED FROM

THE CALL INSTRUCTION WRITES THE RETURN ADDRESS (THE ADDRESS OF THE NEXT INSTRUCTION) INTO THE STACK.

THE RET INSTRUCTION READS THE RETURN ADDRESS FROM THE STACK.

STACK OPERATION

* REMEMBER THAT STACK IS ALWAYS REFERENCED WITH RESPECT TO THE STACK SEGMENT REGISTER
STACK INITIALIZATION

* A STACK SEGMENT IS LIKE A DATA SEGMENT WITH A POINTER TO THE TOP OF THE SEGMENT

STACK_2 SEGMENT
DW 100 DUP(?)
LABEL WORD
STACK_2 ENDS

CODE_A SEGMENT
ASSUME CS: CODE_A, SS: STACK_2
MOV AX, STACK_2
MOV SS, AX
LEA SP, TOP_OF_STACK

CODE_A ENDS

STACK OPERATION WITH CALL AND RET

1. SP IS SET UP INITIALLY.
2. CALL INSTRUCTION WRITES RETURN ADDRESS TO STACK AND TRANSFERS TO PROC1.
3. RET INSTRUCTION READS RETURN ADDRESS FROM STACK INTO IP AND THUS TRANSFERS TO INSTRUCTION AFTER CALL.
PUSH AND POP INSTRUCTIONS

- **PUSH**
  - WRITES A WORD VALUE INTO THE STACK
    SYNTAX
    PUSH MEMORY OR REGISTER

- **POP**
  - READS A WORD VALUE FROM THE STACK
    SYNTAX
    POP MEMORY OR REGISTER

* PUSH CAN BE IMMEDIATE ON 186

COMMUNICATING WITH A PROCEDURE

* PARAMETERS
  PARAMETERS MAY BE PASSED:

  - REGISTERS
    MOV AX, PARM_1
    CALL PROC_1

  - MEMORY
    MOV PARM_1,30
    CALL PROC_1

  - STACK
    PUSH PARM_1
    CALL PROC_1

* FUNCTIONS, (PROCEDURES THAT RETURN A SINGLE VALUE) MAY USE A
REGISTER OR A MEMORY LOCATION TO HOLD THE RETURN VALUE
PROCEDURE EXAMPLE

* DELAY ROUTINE - EXPECTS A BYTE VALUE IN THE AL REGISTER. THIS NUMBER IS THE NUMBER OF 100 MICROSECOND DELAYS THIS PROCEDURE WILL PRODUCE.

NAME     DEMO
PRO       SEGMENT
ASSUME CS:PRO

:FUNCTION: Delay
:INPUTS:  AL - 8 bit integer denoting number of
          100 microsecond delay periods required.
:OUTPUTS: None
:CALLS:   Nothing
:DESTROYS: AL, CL, FLAGS
DELAY PROC
OR AL,AL  ;Check for 0 delay
JL EXIT   ;If 0 - quit
LOOP:    MOV CL,TBI  ;Count for 100 us
         SHR CL, CL  ;Delay 100 us
         DEC AL    ;Adjust iteration counter
         JNE LOOP_  ;Do again if non-zero
EXIT:     RET
         ;Else go back to calling routine
DELAY ENDP
PRO ENDS
END

* THE ABOVE METHOD WORKS WELL FOR PASSING A SINGLE VALUE. HOW WOULD AN ARRAY BE PASSED TO A PROCEDURE?

COMMUNICATING WITH A PROCEDURE

* WHEN PASSING AN ARRAY (OR EVEN A LARGE NUMBER OF DIFFERENT VALUES) TO A PROCEDURE, THE ADDRESS OF THE ARRAY IS USED.

* TO GET THE OFFSET OF AN ARRAY (OR ANY VARIABLE) INTO A REGISTER, THE LEA INSTRUCTION IS USED.

DATA SEGMENT
BUFFER DB 100 DUP(?)
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE,DS:DATA

MOV CX, LENGTH BUFFER
LEA BX, BUFFER
CALL OUTPROC

11-7

11-8
COMMUNICATING WITH PROCEDURES (BASED ADDRESSING)

* THE PROCEDURE CAN THEN USE THE ADDRESS IN THE REGISTER TO ACCESS THE ARRAY.

```
CRT EQU OFFH

OUTPROC PROC
JZ MORE
EXIT

MORE: MOV AL,[BX]; MOV CONTENTS OF BUFFER POINTED TO
OUT CRT,AL
INC BX; INCREMENT BX TO POINT TO NEXT LOCATION
LOOP MORE

EXIT: RET

OUTPROC ENDP
```

* REMEMBER – OFFSET = \[\text{BX}\] + \[\text{DI}\]

* NOTE THAT THIS PROCEDURE CAN BE USED TO OUTPUT THE CONTENTS OF ANY BUFFER.

---

EXAMPLE
PARAMETER PASSING ON THE STACK

**PROBLEM**
A PROCEDURE IS REQUIRED FOR A PL/M PROGRAM TO CONVERT A TEMPERATURE FROM ONE UNIT OF MEASURE TO ANOTHER USING A TABLE OF CONVERSION VALUES. THE TEMPERATURE VALUE, TABLE ADDRESS, AND TABLE LENGTH ARE PARAMETERS PASSED IN THE STACK FROM THE CALLING PROGRAM. ALLOCATION OF STACK SPACE IS HANDLED BY THE CALLING PROGRAM AND THE ITEMS ARE PUSHED ONTO THE STACK IN THE FOLLOWING ORDER:

```
TMPIN TEMPERATURE 1st WORD
N TABLE LENGTH 2nd WORD
TBLADR TABLE ADDRESS 3rd WORD
```

THE PROCEDURE SHOULD SAVE THE BP REGISTER VALUE, BUT ALL OTHER REGISTERS ARE AVAILABLE. UPON EXIT FROM THE PROCEDURE THE RESULTANT VALUE SHOULD BE LEFT IN THE ACCUMULATOR, AND ALL PARAMETERS DELETED FROM THE STACK.
THIS IS AN EXAMPLE OF WHAT IS CALLED A TYPED PROCEDURE IN PL/M
AND IT WOULD BE CALLED WITH A STATEMENT LIKE THIS:

```
TEMPOUT = CONVERT (TEMPIN, N, TBLADR);
```

PL/M EXPECTS THIS PROCEDURE TO RETURN A VALUE IN THE AL REGISTER

---

TABLE OF CONVERSION VALUES
* TABLE LOCATED SOMEWHERE IN MEMORY.
STACK "FRAME" WITH PARAMETERS AFTER CALL

11-13

STACK "FRAME" WITH PARAMETERS AFTER ENTRY

11-14
EXAMPLE

SOLUTION:

8086/8087/8088 MACRO ASSEMBLER

DMO

LOC OBJ LINE SOURCE

----

1 NAME DMO
2 CODE SEGMENT
3 ASSUME CS:CODE

5

6

0000 7 CONVERT PROC
0001 8BEC 9 PUSH BP,SP ;SEE DIAGRAM

10

0003 8B5E04 11 MOV BX,[BP+4] ;BX <-- TBLADR
0006 8B7E06 12 MOV DI,[BP+6] ;DI <-- LENGTH OF TABLE
0009 8B7608 13 MOV SI,[BP+8] ;SI <-- TMPIN

14

000C 3BF7 15 CMP SI,DI ;CHECK IF TMPIN > LENGTH OF TABLE
000E 7206 16 JB INRANG
0010 8A41FF 17 MOV AL,[BX+DI-1] ;IF NOT IN RANGE USE GREATEST VALUE IN TABLE (LENGTH OF TABLE-1)

18

0013 EB0390 19 JMP EXIT
0016 8A00 20 INRANG: MOV AL,[BX+SI] ;USE SI TO POINT TO TEMP. VALUE
0018 5D 21 EXIT: POP BP
0019 C20600 22 RET 6

23 CONVERT ENDP

24

25

26 CODE ENDS

28 END
STEP 1: SAVES THE VALUE FROM THE CALLING PROGRAM'S BP REGISTER ONTO THE STACK AND LOADS BP (STEP 2) WITH THE CURRENT SP VALUE. THIS ESTABLISHES A BASE REGISTER (BP) WHICH WILL BE USED FOR ADDRESSING THE PARAMETERS BEING PASSED.

DURING EXECUTION OF THE MOVE INSTRUCTION (STEP 3) THE DISPLACEMENT VALUE (4) WILL BE ADDED TO THE CONTENTS OF THE BP REGISTER AND AN EFFECTIVE ADDRESS GENERATED EQUIVALENT TO BP+4. SIMILARLY, INDEX REGISTER DI IS LOADED WITH THE SECOND PARAMETER (N) WHEN BP+6 IS ACCESSED IN STEP 4.

THE PROGRAM FIRST CHECKS THE TEMPERATURE TO SEE IF IT IS WITHIN THE RANGE OF VALUES IN THE TABLE. IF IT ISN'T, THE PROCEDURE CONVERTS IT INTO THE HIGHEST TEMPERATURE IN THE TABLE.

REGARDLESS OF WHETHER THE TEMPERATURE IS WITHIN RANGE OR NOT, THE CONVERTED VALUE IS RETURNED IN AL. THE BP IS THEN RESTORED AND THE RET INSTRUCTION IS EXECUTED. THE RET ALSO ADJUSTS THE SP BY 6, THUS REMOVING THE PARAMETERS FROM THE STACK.

NOTE THAT THE PROCEDURE USES BP TO FETCH PARAMETERS OFF THE STACK, THE CPU. WHEN USING BP AS A POINTER, DEFAULTS TO USING THE SS AS THE SEGMENT REGISTER. ANY OTHER POINTER REGISTER COULD BE USED, BUT WOULD REQUIRE AN EXPLICIT SEGMENT OVERIDE.
CLASS EXERCISE 11.1

WRITE AN ASSEMBLY LANGUAGE PROGRAM TO CALL THE
CONVERT PROCEDURE. SET UP A STACK SEGMENT AND
INITIALIZE THE REGISTERS TO POINT TO IT. SET UP
A DATA SEGMENT WITH VARIABLES FOR THE TEMPERATURE
TO CONVERT, THE CONVERSION TABLE, AND A PLACE TO STORE
THE CONVERTED TEMPERATURE.

FOR MORE INFORMATION ...

ASSEMBLY LANGUAGE INSTRUCTIONS
- CHAPTER 3, iAPX 86/88, 186/188 USER’S MANUAL
- CHAPTER 6, ASM86 LANGUAGE REFERENCE MANUAL

PARAMETER PASSING (EXAMPLES)
- PAGE 3-171, iAPX 86/88, 186/188 USER’S MANUAL
- APPENDIX G (EXAMPLES 3,4,5) ASM86 LANGUAGE
  REFERENCE MANUAL
CHAPTER 12

PROGRAMMING WITH MULTIPLE SEGMENTS

• MULTIPLE CODE SEGMENTS
• PROCEDURE DECLARATION
• MULTIPLE DATA SEGMENTS
• SEGMENT OVERRIDE INSTRUCTION PREFIX
• FORWARD REFERENCES
ONE CODE SEGMENT
NEAR, SHORT JUMP
(REVIEW)

CODE1 SEGMENT
ASSUME CS:CODE1

ABC: ___

JMP ABC

CODE1 ENDS

SHORT JUMP ___ BYTE INSTRUCTION
DISPLACEMENT + ____ BYTES
- ____ BYTES

NEAR JUMP ___ BYTE INSTRUCTION
DISPLACEMENT + ____ BYTES
- ____ BYTES

INTERSEGMENT FAR JUMP

CODE1 SEGMENT
ASSUME CS:CODE1

ABC: ___

CODE1 ENDS

CODE2 SEGMENT
ASSUME CS:CODE2

___

JMP ABC

CODE2 ENDS

FAR JUMP 5 BYTE INSTRUCTION
LOADS CS, LOADS IP

OPCODE
NEW IP
NEW CS
ONE CODE SEGMENT
NEAR CALL, RET
(REVIEW)

CODE1 SEGMENT
ASSUME CS:CODE1

HAL PROC
---
RET ENDP

HAL
---

START:
CALL HAL
---

CODE1 ENDS

PROCEDURE DECLARATION
NEAR RETURN RESTORES REGISTER FROM TOP OF STACK
NEAR CALL BYTE INSTRUCTION SAVES REGISTER ON TOP OF STACK
JUMPS+BYTES -BYTES

INTERSEGMENT FAR CALL, RET

PROCEDURE DECLARATION, TYPE FAR
FAR RETURN RESTORES IP AND CS FROM STACK
FAR CALL 5 BYTE INSTRUCTION
SAVES CS AND IP ON TOP OF STACK LOADS NEW CS AND NEW IP

CODE2 SEGMENT
ASSUME CS:CODE2

CODE2 ENDS

<table>
<thead>
<tr>
<th>OP CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW IP</td>
</tr>
<tr>
<td>NEW CS</td>
</tr>
</tbody>
</table>
PROCEDURE DECLARATION

THE PROCEDURE DECLARATION DEFINES WHETHER THE PROGRAM OR SUBROUTINE HAS ATTRIBUTE NEAR OR FAR.

THIS TELLS THE ASSEMBLER TO GENERATE FAR OR NEAR CALLS AND RETURNS.

EXAMPLE:

```assembly
XYZ PROC {NEAR/FAR}
    
    RET

XYZ ENDP
```

ONE DATA SEGMENT

REVIEW

DATA1 SEGMENT

: 
VAR1 DW ?
: 
DATA1 ENDS

CODE1 SEGMENT

ASSUME CS:CODE1
ASSUME DS:DATA1
MOV AX,DATA1
MOV DS,AX
...
MOV VAR1,12H
: 
CODE1 ENDS

DATA REFERENCE
USES DS SEGMENT REGISTER
SEGMENT OVERRIDE INSTRUCTION PREFIX

- DATA IS NORMALLY ACCESSED USING THE DS SEGMENT REGISTER
- DATA CAN BE ACCESSED WITH ANY SEGMENT REGISTER BY USING A ONE BYTE INSTRUCTION PREFIX
- ASM86 GENERATES SEGMENT OVERRIDE PREFIXES AUTOMATICALLY, USING THE ASSUME STATEMENT
ACCESSING CONSTANT DATA

LOC  OBJ  LINE  SOURCE

1  NAME    SAMPLE
2
3  DATA    SEGMENT
4  ALPHA   DB    ?
5  DATA    ENDS
6  CODE    SEGMENT
7  ASSUME CS:CODE,DS:DATA
8  BETA    DW    2000H

0000 0020 10 START: MOV AX,DATA
0002 B8---- 11 MOV DS,AX
0005 8ED8 12
0007 2E8B0E0000 13 MOV CX,BETA ;CS OVERRIDE
000C 8A0E0000 14 MOV CL,ALPHA ;NO OVERRIDE NECESSARY

15
16
17  CODE  ENDS
18  END    START
USING MULTIPLE DATA SEGMENTS

LOC OBJ | LINE | SOURCE
---------|------|---------
      | 1 | NAME SAMPLE2
      | 2 | DATA SEGMENT
      | 3 | ALPHA DB ?
      | 4 | DATA ENDS
      | 5 | DATA SEGMENT
      | 6 | BETA DW, ?
      | 7 | DATA ENDS
      | 8 | CODE SEGMENT
      | 9 | ASSUME CS:CODE, DS:DATA, ES:DATA_2
      | 10 | START: MOV AX, DATA
      | 11 | MOV DS, AX
      | 12 | MOV AX, DATA_2
      | 13 | MOV ES, AX
      | 14 | MOV CX, BETA ; ASSEMBLER CAUSES ES OVERRIDE
      | 15 | MOV CL, ALPHA ; NO OVERRIDE NECESSARY
      | 16 | CODE ENDS
      | 17 | END START

0000 26B0E0000
0003 0ED8
0005 26B0E0000
00FF 0A0E0000
ADDRESSING DATA USING DS AND ES

• All data that belongs to one code segment should be addressed using the DS register
• Any data that is shared between code segments (each having local data) should be addressed using ES
• This allows the program to access local data many times with no penalty in code size
• Shared data will be accessed a few times with a one byte ES override prefix
LOC     OBJ      LINE      SOURCE

----

-----
0000 (100 ??)

-----
0000 ??
0002 ??

------

0000 B8---- R
0003 8ED8
0005 B8---- R
0008 8EC0

000A 8D0E0000
000E 8A0E0200
0012 26880E0000

----

0000 0003 0005 0008 000A 000E 0012

NAME SAMPLE3

SHARED_DATA SEGMENT
BUFFER DB 100 DUP (?)

SHARED_DATA ENDS

LOCAL_DATA SEGMENT
BETA DW ?
ALPHA DB ?

LOCAL_DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:LOCAL_DATA, ES:SHARED_DATA

START: MOV AX, LOCAL_DATA
        MOV DS, AX
        MOV AX, SHARED_DATA
        MOV ES, AX
        MOV CX, BETA
        MOV CL, ALPHA

        MOV BUFFER, CL

CODE ENDS

END START
EXPLICIT SEGMENT OVERRIDE

* ALLOWS YOU TO EXPLICITLY SPECIFY SEGMENT REGISTER USE
WHEN ASSEMBLER DOESN'T HAVE ENOUGH INFORMATION

NAME       SAMPLE
SEGMENT
ASSUME CS:PRO

LOWEST     EQU     61H
HIGHEST    EQU     7AH
CONVERT_VALUE EQU  20H

; THIS PROCEDURE WILL CONVERT ALL OF THE LOWER CASE ASCII
; CHARs IN THE BUFFER POINTED TO BY THE ES:SI REGISTER PAIR
; TO UPPER CASE. THE CX REGISTER CONTAINS THE BYTE COUNT.
; a=61H, z=7AH, A=41H, Z=5AH

UPPER      PROC     FAR
NEXT:      MOV      AL,ES:[SI]
CMP        AL,LOWEST
JB         MOVE_PTR
CMP        AL,HIGHEST
JA         MOVE_PTR
SUB        AL,CONVERT_VALUE
MOV        ES:[SI].AL-

MOVE_PTR:  INC      SI
LOOP       NEXT
RET
UPPER      ENDP
PRO        ENDS
FORWARD REFERENCING

- ASM86 IS A TWO PASS ASSEMBLER
  
  **PASS 1**
  Allocate space and assign offsets for every instruction.

  **PASS 2**
  Fill in opcodes and instruction fields.

- During Pass 1, if an instruction references a label or a variable not yet encountered, (forward reference), ASM86 will take a guess at the correct length for that instruction.

- ASM86 can make incorrect guesses!

---

FORWARD REFERENCES

- The JMP and CALL instructions default to near (within segment).

- Data references to data in a segment defined later defaults to using the DS register.
FORWARD REFERENCING ERRORS

LOC OBJ   LINE   SOURCE
---       ---    ---
0000 9A9090 1      NAME   SAMPLE5
0003 2E8B 1690 2      MOV   DX,VARI
0007 F4      3      HLT
0008 ??      4      NOP
0001 CB      5      RET  ;Forward Reference to a FAR procedure.
0000 00      6      WIZZY ENDS
0000 00      7      WIZZY ENDp
0000 00      8      WIZZY proc FAR
0000 00      9      WIZZY ENDS
0000 00      10     WIZZY ENDS
0000 00      11     WIZZY ENDS
0000 00      12     WIZZY ENDS
0000 00      13     WIZZY ENDS
0000 00      14     WIZZY ENDS
0000 00      15     WIZZY ENDS
0000 00      16     WIZZY ENDS
0000 00      17     WIZZY ENDS
0000 00      18     WIZZY ENDS

ASSEMBLY COMPLETE, 2 ERRORS FOUND
ASSEMBLY COMPLETE, NO ERRORS FOUND
PTR OPERATORS

* THE PTR OPERATORS EXPLICITLY SPECIFY AN INSTRUCTION TYPE

NEAR PTR
FAR PTR
BYTE PTR
WORD PTR
DWORD PTR

EXAMPLES:
JMP FAR PTR THERE
INC WORD PTR [DI]

NOTE: THERE IS ALSO A "SHORT" OPERATOR WHICH ACTS LIKE A PTR OPERATOR WITHOUT THE PTR e.g. JMP SHORT XYZ
ASSEMBLY COMPLETE, NO ERRORS FOUND
PROGRAMMING MODEL

* YOU CAN CHANGE THE ORDER OF SEGMENTS AT LOCATE TIME. THIS IS JUST FOR THE SAKE OF ASSEMBLER.

- EQUATES
- DATA SEGMENT(S)
- STACK SEGMENT
- CODE SEGMENT(S) WITH PROCEDURE(S)
- MAIN
- CODE SEGMENT
- CONSTANTS
- PROCEDURES
- MAIN PROGRAM

FOR MORE INFORMATION ...

SEGMENTATION AND ASSUME USAGE
- CHAPTER 2, ASM86 LANGUAGE REFERENCE MANUAL

FORWARD REFERENCING
- PAGE 1-3, ASM86 LANGUAGE REFERENCE MANUAL

SEGMENT OVERRIDES AND PTR OPERATOR
- CHAPTER 4, ASM86 LANGUAGE REFERENCE MANUAL
CHAPTER 13

INTERRUPTS

- iAPX 86,88 INTERRUPT SYSTEM
- CREATING AN INTERRUPT ROUTINE
- 8259A PRIORITY INTERRUPT CONTROL UNIT
- PROGRAMMING THE 8259A
PROGRAMMED INPUT/OUTPUT

START DEVICE AND POLL FOR COMPLETION

PROGRAM EXECUTION

START DEVICE

INPUT STATUS
POLL AND WAIT
UNTIL DEVICE READY

13-1

INTERRUPT INPUT/OUTPUT

PROGRAM EXECUTION

INTERRUPT SERVICE ROUTINE

IRET

- INTERRUPTS ARE ASYNCHRONOUS EXTERNAL EVENTS

13-2
Interrupt Sequence

Automatic upon detecting interrupt:
- Current instruction finishes execution
- Flags are pushed on the stack
- If and TF are cleared (disables maskable interrupts and single step)
- Save old CS on the stack
- Save old IP on the stack
- Read new CS and IP from interrupt vector table

IRET:
- Far return (pops IP and CS from stack)
- Pop flags

Interrupt processing (response) time – 61 clocks
does not include:
1. Completion of current instruction
2. Saving register data
3. Any wait states

8086,88 Interrupt Vector Table

Table starts at absolute address 0 in memory space.

Dedicated pointers:
0: Divide error
1: Single step – TF
2: Non-maskable interrupt
3: Breakpoint trap
4: Overflow trap
5-31: Reserved by Intel
### iAPX 186,188 Pre-Assigned Interrupt Types

<table>
<thead>
<tr>
<th>Interrupt Name</th>
<th>Vector Type (Decimal)</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>INTO</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>

### Internal Interrupts

<table>
<thead>
<tr>
<th>Type</th>
<th>CAUSED BY...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVIDE ERROR</td>
<td>QUOTIENT LARGER THAN DESTINATION</td>
</tr>
<tr>
<td>SINGLE STEP</td>
<td>MOST INSTRUCTIONS IF TF IS SET</td>
</tr>
<tr>
<td>ARRAY BOUNDS TRAP</td>
<td>BOUND INSTRUCTION IF ARRAY INDEX IS OUTSIDE BOUNDARY</td>
</tr>
<tr>
<td>UNUSED OPCODE TRAP</td>
<td>CPU DIRECTED TO EXECUTE AN UNUSED OPCODE</td>
</tr>
<tr>
<td>ESCAPE OPCODE TRAP</td>
<td>CPU DIRECTED TO EXECUTE ESC OPCODE AND ESC TRAP SET IN RELOCATION REG</td>
</tr>
</tbody>
</table>
SOFTWARE INTERRUPTS

INT N WHERE $0 \leq N \leq 255$

INT 3 SPECIAL ONE BYTE INSTRUCTION TO REPLACE OPCODE FOR SOFTWARE BREAKPOINTS

INTO TYPE 4 INTERRUPT IF OVERFLOW FLAG IS SET, OTHERWISE NEXT INSTRUCTION

SYSTEM CALLS ADVANTAGES

- HARDWARE INDEPENDENCE
- RELOCATABLE CODE
- EFFICIENT USE OF THE SYSTEM
- MULTITASK SUPPORT
- LESS CODE REDUNDANCY
EXAMPLE SYSTEM CALL OPERATION

PROBLEM:
HOW WOULD YOU WRITE THE CODE TO ASK THE OPERATING SYSTEM TO READ A KEY FROM THE KEYBOARD?

SOLUTION:
INT 52H ; CALL TO OPERATING SYSTEMS READ_KEY
CMP AL,\$0DH; CHARACTER RETURNED IN AL
HARDWARE INTERRUPTS

NMI – NON-MASKABLE INTERRUPT
EDGE TRIGGERED
INVOICES TYPE 2 INTERRUPT

INTR – MASKABLE INTERRUPT REQUEST (IF)
LEVEL TRIGGERED
EXTERNAL HARDWARE MUST SUPPLY
INTERRUPT TYPE NUMBER
COMMUNICATIONS WITH EXTERNAL
HARDWARE SET UP BY INTA

INTERRUPT PROCESSING

8259A

INTR
INTA

IAPX 8688

VECTOR TABLE

INT-PTR

INTERRUPT SERVICE ROUTINE
8259A PROGRAMMABLE INTERRUPT CONTROLLER

- PROVIDES UP TO 8 PRIORITIZED INTERRUPTS WITH FIXED OR ROTATING PRIORITY SCHEMES.

- EXPANDABLE TO 64 INTERRUPTS WITH PRIORITY MODES DEFINABLE IN GROUPS OF 8.

- ABILITY TO INDIVIDUALLY MASK INTERRUPTS.

- SUPPLIES INTERRUPT TYPE NUMBER IN RESPONSE TO INTERRUPT ACKNOWLEDGE.
**8259A OPERATION**

- Looks at current requests and also any interrupts in-service. If requesting level has highest priority, it is put in-service and an interrupt request is sent to CPU.

**INITIALIZATION AND CONTROL**

- To use the 8259A, it must be initialized. This is done using 3 or 4 initialization command words (ICW1-ICW4).

- Once initialized, the 8259A's operation can be controlled or modified with any one of three operational command words (OCW1-OCW3).
INITIALIZATION SEQUENCE

ICW1

ICW2

IN CASCADE MODE

NO

ICW3

YES

ICW4

READY TO ACCEPT INTERRUPT REQUESTS

ICW1 AND ICW2

ICW1

A₀  D₇ D₆ D₅ D₄ D₃ D₂ D₁ D₀
0  0  0  0  1  LTIM 0  SGL  1

1 = SINGLE
0 = CASCADE MODE

ICW2

A₀  D₇ D₆ D₅ D₄ D₃ D₂ D₁ D₀
1  T7  T6  T5  T4  T3  0  0  0

1 = LEVEL TRIGGERED INPUT
0 = EDGE TRIGGERED INPUT

T7 - T3 OF MODE INTERRUPT TYPE
INTERRUPT TYPE SELECTION

5 MBS of INTERRUPT TYPE INserted AUTOMATICALLY, Relative TO IR LEVEL CAUSING INTERRUPT

EXAMPLE:
ASSUME INTERRUPT TYPES 32-39

<table>
<thead>
<tr>
<th>IRO</th>
<th>IRI</th>
<th>IR2</th>
<th>IRI</th>
<th>IR3</th>
<th>IR4</th>
<th>IR5</th>
<th>IR6</th>
<th>IR7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

EXAMPLE: 5 MBS, OF INSERTED INTERRUPT TYPE AUTOMATICALLY, RELATIVE TO IR LEVEL CAUSING INTERRUPT

ICW3

* USED IN CASCADE MODE ONLY

THE MASTER AND EACH SLAVE DEVICE HAVE DIFFERENT ICW3s.

ICW3 (MASTER DEVICE)

A0  D7 D6 D5 D4 D3 D2 D1 D0
1   B7 B6 B5 B4 B3 B2 B1 B0

ICW3 (SLAVE DEVICE)

A0  D7 D6 D5 D4 D3 D2 D1 D0
1   X  X  X  X  X  ID 1  ID 0

1 = IR INPUT HAS A SLAVE
0 = IR INPUT DOES NOT HAVE A SLAVE

SLAVE ID

0 1 2 3 4 5 6 7
0 1 0 1 0 1 1 0
0 0 1 1 0 0 1 1
0 0 0 0 1 1 1

13-19

13-20
SET UP OF ICW3

ICW3 = 1

MASTER

IR1

INTR

IR2

SLAVE

ID = 1

ICW3 = 2

SLAVE

ID = 2

ICW4

A0  D7  D6  D5  D4  D3  D2  D1  D0
1   0   0   0   SFNM  BUF  MS  AEOI  1

1 AUTO EOI
0 NORMAL EOI

USE NORMAL EOI, WILL HAVE NO PROBLEMS THAT WAY.

DETERMINES FUNCTION OF SP/EN

1 SPECIAL FULLY NESTED MODE
0 FULLY NESTED MODE
FULLY NESTED MODE

- ENTERED BY DEFAULT UPON INITIALIZATION

HIGHEST — IR0
IR1
IR2
IR3
IR4
IR5
IR6

LOWEST — IR7

- IF AN INTERRUPT LEVEL IS IN SERVICE, FURTHER INTERRUPTS FROM THAT LEVEL AND ALL LOWER PRIORITY LEVELS ARE INHIBITED UNTIL AN EOI IS ISSUED.

M ASTER/SLAVE CONFIGURATION

SHOULD BE IN SPECIAL FULLY NESTED MODE. PERMITS NESTING OF INTERRUPTS ON SINGLE IR INPUT.

IF NECESSARY, MASTER PLACES SLAVE ID ON CAS0-2

CAS 0 - CAS 2

SLAVE ID
**NON-BUFFERED MODE**

- 8086
- ADDR/DATA
- ALE
- INTA
- INTR

- ADDRESS LATCHES
- STB

- DATA BUS
- ADDRESS BUS

- ADDR/DATA
- ADDR/DATA

- INT
- INTA
- SP
- 8259A
- INT

+ 5

*SP IDENTIFIES 8259A AS MASTER OR SLAVE DEVICE*

---

**BUFFERED MODE**

- 8086
- INTR
- SG-52
- ALE
- 8288
- DEN
- INTA

- ADDRESS LATCHES
- STB

- DATA BUS TRANSCEIVERS
- OE

- DATA BUS
- ADDRESS BUS

- 8259A
- EN
- INT

- INTA
- INT

*EN USED TO CONTROL LOCAL DATA BUS*
OPERATIONAL COMMAND WORDS

OCW1 AND OCW2

OCW1

A0 A1 A2 A3 A4 A5 A6 A7
0 0 1 1 1 1 1 1

OCW2

A0 B1 B2 B3 B4 B5 B6 B7
0 0 1 1 1 1 1 1

Interrupt Mask
1 Mask Set
0 Mask Reset

Interrupt Level to be Acted Upon

0 1 2 3 4 5 6 7
0 1 0 1 0 1 0 1
0 0 1 1 1 1 1 1

Specific EOI Command
Non-Specific EOI Command
Specific EOI Command
Rotating On/Off
Rotate On/Off
Rotate On/Off (Clear)
Rotate On/Off
Reset Priority Command
Set Priority Command
No Operation

END OF INTERRUPT

Automatic Rotation
Specific Rotation

ROTATING PRIORITIES

BEFORE
HIGHEST → IR0
IR1
IR2
IR3
IR4
IR5
IR6
LOWEST → IR7

AFTER
HIGHEST → IR4
IR5
IR6
IR7
IR0
IR1
IR2
LOWEST → IR3

LEVEL 3 SPECIFIED IN ROTATE COMMAND
**OCW3**

**READ REGISTER COMMAND**

- **X**: 0
- **0**: 1

**NO ACTION**

- READ IR REG ON NEXT RD PULSE
- READ IS REG ON NEXT RD PULSE

**1 POLL COMMAND**

- 0 NO POLL COMMAND

**SPECIAL MASK MODE**

- **X**: 0
- **0**: 1

**NO ACTION**

- RESET SPECIAL MASK
- SET SPECIAL MASK

---

**HARDWARE SET UP FOR SAMPLE PROGRAM**

**ADDRESS BUS**

**CONTROL BUS FROM 8088**

**DATA BUS**

- **CS/A0**: PORT ADDRESS 40H, 41H
- **DO-7**: PORT ADDRESS 30H, 31H

**SLAVE 8259A**

**MASTER 8259A**

**SP/EN**

**GND**

**VCC**

**INTA**

**INT**

**...**

**...**
SETTING UP TIMER INTERRUPT

INT VECTOR SEGMENT AT 0
ORG 28H

TIMER_INT_IP DW ?
TIMER_INT_CS DW ?
INT_VECTOR ENDS

INTERRUPTS SEGMENT
ASSUME CS:INTERRUPTS

TIMER:
  STI ;ENABLE INTERRUPTS
  PUSH AX

;PUSH OTHER REGISTERS USED IN INTERRUPT
;HANDLE THE TIMER INTERRUPT
;POP REGISTERS IN REVERSE ORDER OF PUSH

  MOV AL,60H ;SPECIFIC EOI FOR SLAVE
  OUT 40H,AL
  MOV AL,0DH ;COMMAND TO READ ISR
  OUT 40H,AL
  IN AL,40H ;READ ISR
  CMP AL,0 ;CHECK TO SEE IF EMPTY
  JNZ EXIT ;DON'T SEND EOI TO MASTER
  MOV AL,64H ;SPECIFIC EOI FOR MASTER
  OUT 30H,AL

  EXIT:
  POP AX
  IRET

INTERRUPTS ENDS

;SET UP POINTER TO INTERRUPT
MAIN SEGMENT
ASSUME CS:MAIN, ES:INT_VECTOR

INIT: CLI
MOV AX, INT_VECTOR
MOV ES, AX
MOV TIMER_INT_IP, OFFSET TIMER
MOV TIMER_INT_CS; SEG TIMER

; INITIALIZE TIMER AND OTHER PERIPHERALS
; INITIALIZE MASTER 8259A AND SLAVE 8259A
INITIALIZING MASTER 8259A AND SLAVE 8259A

; INITIALIZE THE MASTER

MOV AL, 11H ; ICW1 - CASCADE MODE, EDGE TRIGGER
OUT 30H, AL
MOV AL, 20H ; ICW2 - INTERRUPT TYPES 32 - 39
OUT 31H, AL
MOV AL, 10H ; ICW3 - MASTER HAS ONE SLAVE ON IR4
OUT 31H, AL
MOV AL, 11H ; ICW4 - SPECIAL FULLY NESTED MODE, NON-BUFFERED, NORMAL EOI
OUT 31H, AL

; INITIALIZE THE SLAVE

MOV AL, 11H ; ICW1 - CASCADE MODE, EDGE TRIGGER
OUT 40H, AL
MOV AL, 28H ; ICW2 - INTERRUPT TYPES 40 - 47
OUT 41H, AL
MOV AL, 04H ; ICW3 - SLAVE ID IS 4
OUT 41H, AL ; CONNECTED TO MASTER IR4
MOV AL, 01H ; ICW4 - FULLY NESTED MODE, NON-BUFFERED, NORMAL EOI
OUT 41H, AL
STI ; ENABLE INTERRUPTS

; REST OF MAIN PROGRAM CODE GOES HERE

MAIN ENDS
END INIT
CLASS EXERCISE 13.1

ASSUME THAT YOU HAVE A PROGRAM THAT CONTAINS THE INSTRUCTION

\textbf{DIV BL}

SINCE YOU DO NOT DO ANY RANGE CHECKING BEFORE THE OPERATION, THERE IS A POSSIBILITY OF A DIVIDE ERROR. WRITE AN INTERRUPT PROCEDURE FOR THE DIVIDE ERROR INTERRUPT THAT LOADS THE AH REGISTER WITH FFH AND THE AL REGISTER WITH OOH AND THEN RETURN. ALSO WRITE THE INSTRUCTIONS TO CREATE THE POINTER.
FOR MORE INFORMATION ...

INTERRUPT STRUCTURE
- PAGE 4–6, iAPX 86/88, 186/188 USER’S MANUAL

PROGRAMMING THE 8259A (EXAMPLES)
- PAGE 3–186, iAPX 86/88, 186/188 USER’S MANUAL
CHAPTER 14
MEMORY AND IO INTERFACING

• MEMORY ORGANIZATION
• SPEED REQUIREMENTS
• ADDRESS DECODING
8086 MEMORY ORGANIZATION

TO THE PROGRAMMER:

1 MBYTE CAN BE ADDRESSED AS
1 M BYTES OF MEMORY
512 K WORDS OF MEMORY

NO CONSTRAINTS ON BYTE OR WORD MEMORY ACCESSES.
(WORDS CAN BE ON ODD OR EVEN BOUNDARIES)

* MEMORY ORGANIZED IN TWO BANKS
* ALL ODD ADDRESSES IN ONE BANK—EVEN ADDRESSES IN OTHER
* BYTE ACCESS IN EITHER BANK
* ALIGNED WORD CAN BE ACCESSED IN ONE BUS CYCLE
* NON-ALIGNED WORD REQUIRES TWO BUS CYCLES
8086 MEMORY INTERFACING

STANDARD MEMORY INTERFACE

<table>
<thead>
<tr>
<th>BANK</th>
<th>SELECTED BY</th>
<th>CONNECTED TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEN</td>
<td>A0</td>
<td>D0-D7</td>
</tr>
<tr>
<td>ODD</td>
<td>A1-A19 BHE</td>
<td>D8-D15</td>
</tr>
</tbody>
</table>
THE 8086 WILL INTERNALLY TRANSFER A BYTE FROM ONE SIDE OF ITS DATA BUS TO THE OTHER IF IT NEEDS TO.

E.g. IN ORDER TO MOVE A BYTE OF DATA FROM AN ODD ADDRESS INTO THE CL REGISTER
WHAT IS REQUIRED TO WRITE A WORD FROM MEMORY ADDRESS 4? IS THIS AN ALIGNED WORD?

WHAT IS REQUIRED TO WRITE A WORD FROM MEMORY ADDRESS 5? IS THIS AN ALIGNED WORD?
PROM MEMORY INTERFACING

CURRENT PROM DEVICES

SINGLE 5VOLT POWER REQUIREMENTS
LOW POWER STANDBY MODE
CE/ AND OE/ SELECT LINES

2758      1024 BYTES
2716      2048 BYTES
2732,2732A   4096 BYTES
2764      8192 BYTES
27128     16384 BYTES
27256     32768 BYTES
I/O DEVICE SELECTION

* IN/OUT PORTS CAN TRANSMIT BYTES (8 BITS) OR WORDS (16 BITS).
* BYTE I/O PORTS CAN COMMUNICATE ON THE LOW (D0–D7) DATA BUS LINES OR THE HI (D8–D15) DATA BUS LINES.
* EVEN ADDRESSED I/O PORTS TRANSFER DATA ON LOW (D0–D7) DATA BUS LINES.
* ODD ADDRESSED I/O PORTS TRANSFER DATA ON HI (D8–D15) DATA BUS LINES.

WARNING: CARE MUST BE EXERCISED THAT EACH REGISTER WITHIN AN 8 BIT PERIPHERAL CHIP IS ADDRESSED BY ALL EVEN OR ALL ODD ADDRESSES.
8086 I/O INTERFACE

DO NOT CONNECT "AO" LINE ON PERIPHERAL TO AO LINE OF ADDRESS BUS.

MEMORY SPEED REQUIREMENTS

PROCESSOR
- ALLOWS MEMORY AND IO A SPECIFIC AMOUNT OF TIME TO RESPOND WITH DATA AFTER IT ISSUES AN ADDRESS (MEMORY ACCESS TIME - \( T_{ad} \))
- MEMORY ACCESS TIME IS PROPORTIONAL TO CLOCK SPEED

MEMORY
- REQUIRES FINITE PERIOD OF TIME TO RESPOND WITH DATA TO A VALID ADDRESS (\( T_{acc} \))
CALCULATING PROCESSOR REQUIREMENTS

\[ T_{ad} = 3 \times T_{clcl} - T_{clav} - T_{dvcl} \] (PROCESSOR ACCESS TIME)

WHERE

- \( T_{clcl} \) = CLOCK PERIOD
- \( T_{clav} \) = TIME PERIOD FROM CLOCK TO ADDRESS VALID
- \( T_{dvcl} \) = SET UP TIME FOR DATA IN

FOR A MINIMUM MODE 8086

<table>
<thead>
<tr>
<th>5 MHZ 8086</th>
<th>8 MHZ 8086-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{clcl} = 200 \text{ nsec} )</td>
<td>( T_{clcl} = 125 \text{ nsec} )</td>
</tr>
<tr>
<td>( T_{clav} = 110 \text{ nsec} )</td>
<td>( T_{clav} = 60 \text{ nsec} )</td>
</tr>
<tr>
<td>( T_{dvcl} = 30 \text{ nsec} )</td>
<td>( T_{dvcl} = 20 \text{ nsec} )</td>
</tr>
</tbody>
</table>

\[ T_{ad} = \]

PROCESSOR REQUIREMENTS

(CLK --./ -- 1-- 1-- T_{clcl} --~ -- ALE

ADO–AD15

DATA IN

(\( T_{clav} \) -- \( T_{clav} \)

A0–A15

(\( T_{dvcl} \) -- \( T_{dvcl} \)

T_{ad} = 14-15

14-16
MEMORY TIMING

ADDRESS BUS

DATA BUS

Tacc

DATA

BUS CONFIGURATIONS (MINIMUM MODE)

8086 MINIMUM MODE (MULTIPLEXED BUS)

8086

8282 LATCH

RAM/EPROM/ROM

8086 MINIMUM MODE (BUFFERED BUS)

8086

8282 LATCH

RAM/EPROM/ROM

8286 TRANSCEIVER
BUS CONFIGURATIONS
(MAXIMUM MODE)

8086 MAXIMUM MODE
(BUFFERED BUS)

8088 MAXIMUM MODE
(.DOUBLED BUFFERED BUS)

WAIT STATES

IN ANY SYSTEM YOU MUST CONSIDER ANY DELAYS ENCOUNTERED BY BOTH THE ADDRESS OR THE DATA ON THE "ROUND TRIP".
SYSTEM TIMING FACTORS

* ANY BUFFERS, LATCHES AND DECODE LOGIC IN THE 8086 SYSTEM MUST BE CONSIDERED IN THE TIMING ANALYSIS

DELAY TIMES:

- 8282/8286
- 8283/8287
- 8205/LOGIC

* THESE DELAY TIMES MUST BE SUBTRACTED FROM THE CPU ACCESS TIME.

ARE WAIT STATES NEEDED?

IF THE SYSTEM ARCHITECTURE JUST DOES NOT ALLOW THE CPU TO SEE DATA WITHIN ITS REQUIRED Tad YOU CAN EXTEND THE BUS CYCLE WITH A WAIT STATE (OR MULTIPLE WAIT STATES).

TO DETERMINE HOW MANY WAIT STATES:

1. Evaluate $T_{total} = T_{delay} - (T_{result} + T_{positive})$
2. If positive: NO WAIT STATES REQUIRED
3. If negative: Solve for $N$ in $N \times (T_{result}) = (T_{delay} - T_{positive})$
8086 AND 8088 WAIT STATE CHART
5MHZ

MEMORY MATRIX
NO WAITS STATES

<table>
<thead>
<tr>
<th>MODE</th>
<th>MIN MODE</th>
<th>MAX MODE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MULTIPLEXED BUS</td>
<td>BUFFERED</td>
</tr>
<tr>
<td>STATIC RAM</td>
<td>2114-3</td>
<td>2114-3</td>
</tr>
<tr>
<td></td>
<td>2141-5</td>
<td>2141-5</td>
</tr>
<tr>
<td></td>
<td>2147</td>
<td>2147</td>
</tr>
<tr>
<td></td>
<td>2168</td>
<td>2168</td>
</tr>
<tr>
<td>EEPROM</td>
<td>2816</td>
<td>2816</td>
</tr>
<tr>
<td>EPROM</td>
<td>2716-2</td>
<td>2716-2</td>
</tr>
<tr>
<td></td>
<td>2732A</td>
<td>2732A</td>
</tr>
<tr>
<td></td>
<td>2764</td>
<td>2764</td>
</tr>
<tr>
<td>DYNAMIC RAM</td>
<td>2118-7</td>
<td>2118-7</td>
</tr>
<tr>
<td></td>
<td>2164</td>
<td>2164</td>
</tr>
</tbody>
</table>

14-23

iSBC 86/05 DESIGN EXAMPLE

14-24
ADDRESS DECODING
EXAMPLE USING BIPOLAR PROMS

ADVANTAGES
HIGHLY FLEXIBLE DESIGN ALLOWS:
DIFFERENT MEMORY COMPONENTS
FIELD MODIFICATIONS
EASY UPGRADE TO NEW MEMORY DEVICES

DISADVANTAGES
HIGHER COST (?)
REDUCED ACCESS TIME

CLASS EXERCISE 14.1

1. WHAT IS THE FIRST ADDRESS OF THE FIRST LOCATION IN THE 2186 +4 ON PAGE 14-9?

2. WHY DO WE NEED ONLY ONE ADDRESS DECODER IN A ROM MEMORY AS SHOWN ON PAGE 14-11? WHAT MAKES THIS POSSIBLE?

3. CAN AN 8088 READ A WORD PORT?

4. DOES A 5MHZ 8086 CPU IN MINMODE BUFFERED SYSTEM REQUIRE WAIT STATES TO ACCESS A 2764 EPROM? WHAT IF IT WERE AN 8MHZ 8086? (2764 Tacc = 250 nsec)

5. IF A WAIT STATE IS REQUIRED, WHICH CHIP ACTUALLY GENERATES THE WAIT STATE?
FOR MORE INFORMATION ...

MEMORY INTERFACING AND ADDRESS DECODING
- AP-67, 8086 SYSTEM DESIGN

AVAILABLE MEMORY COMPONENTS
- MEMORY COMPONENTS HANDBOOK

RELATED TOPICS ...

IN SOME SYSTEMS THE TIMING OF THE MEMORY STROBES (RD, WR) MIGHT ALSO BE A CONCERN. AP-67 COVERS THIS CONSIDERATION (Too) IN DETAIL.
DAY 4 OBJECTIVES

BY THE TIME YOU FINISH TODAY YOU WILL:

* IMPLEMENT AN ENCRYPTOR IN SOFTWARE USING THE XLATB INSTRUCTION

* MOVE A BLOCK OF MEMORY USING THE STRING MOVE INSTRUCTIONS

* ADD THE PROPER ASSEMBLER DIRECTIVES TO A MODULE SO THAT IT CAN REFERENCE AND USE AN EXISTING PIECE OF SOFTWARE

* EMULATE ON PAPER AN 8086 INTERFACED TO MEMORY, GENERATING THE PROPER SIGNALS TO ACCESS A BYTE OR A WORD ON ANY BOUNDARY

* DETERMINE WHETHER A PARTICULAR SYSTEM WILL REQUIRE WAIT STATES GIVEN THE SYSTEM CONFIGURATION AND THE DEVICE SPECIFICATIONS

* OPTIONALLY DEBUG USING ICE-86
CHAPTER 15

PROGRAMMING TECHNIQUES

• JUMP TABLE (INDIRECT JUMPS)
• BLOCK MOVE (STRING INSTRUCTIONS)
• TABLE LOOK-UP (XLATB INSTRUCTION)
PROBLEM

A program is to be written that reads the value of an 8 bit input port and transfers to one of a set of routines depending on the value read. Five processing routines are provided as well as one error routine. If the value read is in the range of 0 ... 4 then the program should transfer to routine 0 ... routine 4. If the input value is out of range, greater than 4, the program should transfer to the error routine.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>NAME</th>
<th>JUMP_TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1C00</td>
<td>PORT</td>
<td>EQU 00H</td>
</tr>
<tr>
<td>0002</td>
<td>1E00</td>
<td>CODE</td>
<td>SEGMENT</td>
</tr>
<tr>
<td>0004</td>
<td>2000</td>
<td>ASM</td>
<td>ES:CODE</td>
</tr>
<tr>
<td>0006</td>
<td>2200</td>
<td>TABLE</td>
<td>DW ROUTINE0, ROUTINE1, ROUTINE2, ROUTINE3, ROUTINE4</td>
</tr>
<tr>
<td>0008</td>
<td>2400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00A4</td>
<td>E400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C4</td>
<td>3C04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00E4</td>
<td>770A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0104</td>
<td>32E4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0124</td>
<td>B8F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0144</td>
<td>D1E7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0164</td>
<td>2EFF25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0184</td>
<td>F4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01C4</td>
<td>EBFD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01E4</td>
<td>EBFB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0204</td>
<td>E9F7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0224</td>
<td>EBF5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0244</td>
<td>EBF3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Code**

```assembly
JMP_ROUTINE0:
JMP_ROUTINE1:
JMP_ROUTINE2:
JMP_ROUTINE3:
JMP_ROUTINE4:
```

**Error Handling**

```assembly
IN AL, PORT
CMP AL, 4
JA ERROR
XOR AH, AH
MOV DI, AX
SHL DI, 1
JMP_TABLE[DI]
EXIT: HLT
```

**Jumping to Routines**

```assembly
ROUTINE0:
ROUTINE1:
ROUTINE2:
ROUTINE3:
ROUTINE4:
```

**Assembly Code**

```assembly
CODE ENDS
END START
```
SOLUTION

A table is constructed; each entry in the table is the address of one of the processing routines. The first entry in the table is the address of routine0, the second the address of routine1, .... An indirect jump instruction with indexed addressing will utilize the table.

STEPS

1. Input value from port into AL
2. Check value to see if it is out of bounds. If so, transfer to the error routine.
3. Assume that DI will be used as the index register for the indirect jump. Set AH to zero to make a word value
4. MOV AX TO DI
5. Double DI for word indexing
6. Jump indirect to the proper routine

JMP INSTRUCTION ADDRESSING (INDIRECT JUMPS)

- Indirect jumps use an address which is in a register or a memory location.
- Indirect jumps can use any of the 8086,88 addressing modes.
- All jump instructions use the same mnemonic.

Examples:

JMP CX
JMP WORD PTR [Bx]
BLOCK MOVE
(STRING INSTRUCTIONS)

PROBLEM

MANIPULATING LARGE BLOCKS OF MEMORY IS A COMMON AND TIME-CONSUMING TASK OF COMPUTERS. WRITE A PROGRAM THAT MOVES A BLOCK OF DATA FROM ONE MEMORY LOCATION TO ANOTHER. THE CODE SHOULD BE EFFICIENT AND FAST.

MOTIVATION FOR STRING OPERATORS

* WORD BLOCK MOVE WITHOUT STRING OPERATORS

DATA SEGMENT
SOURCE DW 100 DUP (?)
DESTINATION DW 100 DUP (?)
DATA ENDS

CODE SEGMENT
ASSUME CS: CODE, DS: DATA
MOV AX, DATA
MOV DS, AX

LEA SI, SOURCE
LEA DI, DESTINATION
MOV CX, LENGTH SOURCE

BLOCK:

MOV AX, [SI] ; 12 MICROSECONDS PER WORD
ADD SI, 2
ADD DI, 2
LOOP BLOCK

CODE ENDS

15-5

15-6
STRING INSTRUCTIONS

* BYTE AND WORD ORIENTED ONE BYTE INSTRUCTIONS

* USE DS:SI AS SOURCE POINTER

* USE ES:DI AS DESTINATION POINTER

* USE DIRECTION FLAG BIT
  DF = 0 PROCEEDS TO HIGHER MEMORY ADDRESS
  DF = 1 PROCEEDS TO LOWER MEMORY ADDRESS

* ADDITIONAL INSTRUCTION
  STD
  CLD

STRING INSTRUCTION

MOVS B  ;[DI] ← [SI]
MOVS W  ;SI ← SI + 1 (+2 FOR WORD)
         ;DI ← DI + 1 (+2 FOR WORD)

ASSUMING DF=0
**OTHER STRING INSTRUCTIONS**

- **CMPSB**
- **CMPSW**
- **SCASB**
- **SCASW**
- **LODSB**
- **LODSW**
- **STOSB**
- **STOSW**

**COMPARE TWO BLOCKS OF MEMORY**

**SCAN FOR AN ITEM IN MEMORY**

**LOAD AX/AL WITH STRING ITEM**

**STORE AX/AL IN MEMORY**

---

**NOTE:** THESE INSTRUCTIONS PERFORM ONE BYTE OR WORD OPERATION ONLY.

---

**REPEAT INSTRUCTION PREFIX**

- **ONE BYTE INSTRUCTION PLACED BEFORE STRING INSTRUCTION TO FORM BLOCK STRING OPERATIONS**

- **FOR STRING INSTRUCTIONS THAT DO NOT AFFECT THE FLAGS:**

  - REP \{ MOV, STOS, LODS \}

- **FOR STRING INSTRUCTIONS THAT DO AFFECT THE FLAGS:**

  - REPZ, REP
  - REPNZ, REPNE \{ CMPS, SCAS \}
OPERATION OF THE
REP PREFIX

START

PREVIOUS
INSTRUCTIONS

REPEAT
PREFIX

PRESENT

CX:0

STOP

DO STRING
OPERATION
USING SI/DI

ADJUST
SI/DI
BY DELTA

CMPS
OR
SCAS

YES

CMPS
OR
SCAS

NO

REPEAT
PREFIX

ABSENT

NEXT
INSTRUCTION

S/D/O, CX
AND OF WOULD
typically be
initialized here

DI

Interrupt

PENDING

DECREMENT
CX BY 1

NOT PENDING

NORMAL
SYSTEM
INTERUPT
SERVICE

STRING

DELTA

BYTE

0

1

BYTE

1

-1

WORD

0

2

WORD

1

-2

PREFIX

REPE

REPZ

REPNE

REPN

0

1

1

1

2

-2

2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2

-2
EXAMPLES OF BLOCK OPERATIONS

BLOCK MOVE

DATA SEGMENT
SOURCE DW 100 DUP(?)
DESTINATION DW 100 DUP(?)
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS: DATA, ES: DATA
MOV AX, DATA
MOV DS, AX
MOV ES, AX

CLD
LEA SI, SOURCE
LEA DI, DESTINATION
MOV CX, LENGTH SOURCE
REP MOVSW

; 3.4 MICROSECONDS PER WORD
TABLE LOOK UP  
(XLATB INSTRUCTION)

PROBLEM

ASSUME WE HAVE A TEMPERATURE SENSOR ATTACHED TO AN 8 BIT ACCURACY ANALOG TO DIGITAL CONVERTER.  THIS CONVERTER IS ATTACHED TO PORT 12 OF OUR 8086 SYSTEM.  UNFORTUNATELY, THE SENSOR DOES NOT PRODUCE A LINEAR OUTPUT.

WE WANT TO WRITE A PROCEDURE THAT INPUTS FROM THIS PORT AND QUICKLY CONVERTS THE INPUTTED VALUE TO THE CORRECT TEMPERATURE VALUE.

SOLUTION

USE A CONVERSION TABLE AND "LOOK-UP" THE CORRECT VALUE.
TABLE LOOK-UP

ASSEMBLY COMPLETE, NO ERRORS FOUND
THE XLATB INSTRUCTION USES THE AL REGISTER AS AN INDEX INTO A BYTE TABLE. THE BYTE ACCESSED IS PUT IN THE AL REGISTER.

XLAT IS USEFUL FOR MANY CONVERSIONS E.G., ASCII TO EBCDIC
CLASS EXERCISE 15.1

WRITE A PROCEDURE THAT WILL ENCRYPT THE CONTENTS OF A BUFFER WHICH CONTAINS NUMBERS IN HEX ASCII FORMAT SO THAT:

30H - ASCII 0 BECOMES AN ASCII 5
31H - 1 4 0
32H - 2 4 7
33H - 3 4 2
34H - 4 4 2
35H - 5 4 8
36H - 6 4 3
37H - 7 4 9
38H - 8 4 1
39H - 9 4 6


FOR MORE INFORMATION...

BRANCH TABLE (EXAMPLE)
- APPENDIX G, ASM86 LANGUAGE REFERENCE MANUAL

STRING AND XLATB INSTRUCTIONS
- CHAPTER 6, ASM86 LANGUAGE REFERENCE MANUAL
- CHAPTER 3, IAPX 86/88, 186/188 USER'S MANUAL

RELATED TOPICS...

THERE ARE MORE 8086 INSTRUCTIONS THAT ARE NOT DISCUSSED IN THIS WORKSHOP. IT WOULD BE A GOOD IDEA TO LEAF THROUGH THE COMPLETE LIST IN CHAPTER 6 OF THE ASM86 LANGUAGE REFERENCE MANUAL.
CHAPTER 16

MODULAR PROGRAMMING

- PUBLIC DECLARATIVE
- EXTRN DECLARATIVE
- COMBINING SEGMENTS
- LINK86
- LOC86
WHAT IS MODULAR PROGRAMMING?

- Problem is broken into manageable parts.
- Modules are developed concurrently.
- Easier to debug and maintain.

SOFTWARE DEVELOPMENT PROCESS
LINKAGE

THE LINK86 PROGRAM COMBINES RELOCATABLE OBJECT FILES TO ACT AS IF THEY WERE CREATED AT ONE TIME. ALL REFERENCES BETWEEN MODULES ARE RESOLVED.

LINK86 ALLOWS A PROGRAM TO BE BROKEN UP INTO MODULES SO THAT THE ENTIRE PROGRAM DOES NOT HAVE TO BE RETRANSLATED EVERY TIME CHANGES ARE MADE.

RELOCATION

THE ABILITY TO ASSIGN MEMORY ADDRESSES TO A PROGRAM AFTER IT HAS BEEN TRANSLATED.

ASM86 AND PLM86 MARK SOME ADDRESSES AS BEING RELOCATABLE. THE ADDRESSES WILL BE CONVERTED TO ABSOLUTE ADDRESSES BY THE LOC86 PROGRAM.
THE QUESTION:
HOW TO REFERENCE LABELS AND VARIABLES IN OTHER ASSEMBLED MODULES?

NAME MOD_A
SEG A SEGMENT
ASSUME CS:SEGA
CALL PROCA
SEGA ENDS
END

NAME MOD_B
SEG B SEGMENT
ASSUME CS:SEG B
PROCA PROC FAR
RET
PROCA ENDP
SEG B ENDS
END

PROCA IS UNDEFINED IN THE SEGA MODULE, THE TWO MODULES WOULD HAVE TO BE REASSEMBLED TOGETHER TO ALLOW THE REFERENCE TO PROCA

16-5

THE ANSWER:
BY USING PUBLIC AND EXTRN DECLARATIVES WITH THE TWO MODULES LINK86 CAN RESOLVE EXTERNAL REFERENCES

NAME MOD_A
EXTRN PROCA:FAR
SEG A SEGMENT
ASSUME CS:SEGA
CALL PROCA
SEGA ENDS
END

NAME MOD_B
PUBLIC PROCA
SEG B SEGMENT
ASSUME CS:SEG B
PROCA PROC FAR
RET
PROCA ENDP
SEG B ENDS
END

16-6
PUBLIC AND EXTERNAL DECLARATIVES

PUBLIC makes a name available to other modules.

EXTERN makes names defined elsewhere usable in this module.

EXAMPLES:

PUBLIC       XYZ, WP, ERS
EXTERN       F00: BYTE *

* ATTRIBUTES

NEAR, FAR
BYTE, WORD, DWORD
ABS
MAIN PROGRAM

8086/8087/8088 MACRO ASSEMBLER DEMO 09/01/80 PAGE 1

LOC OBJ LINE SOURCE

1 ;THIS ROUTINE INPUTS AND OUTPUTS TO THE I/O BOX OF THE MOS.
2 ;IT USES AN EXTERNAL DELAY ROUTINE TO DELAY 1 SECOND
3 ;BETWEEN A INPUT AND A SUBSEQUENT OUTPUT.
4
5 NAME DEMO
6
7 ----
8 9 STACK SEGMENT
10 DW 10 DUP (?)
11
12 ----
13 14 CODE SEGMENT
15 ASSUME CS:CODE,SS:STACK
16
17 2710 SECOND EQU 10000 ;DELAY PARAMETER FOR 1 SECOND
18
19 0000 B8---- R 18 START: MOV AX,STACK
20
21 0003 8ED0 19 MOV SS,AX
22 0005 B0261400 20 LEA SP,TOP
23 0009 BA1027 21 MOV DX,SECOND
24 000C E400 22 LOOP : IN AL,0
25 000E 52 23 PUSH DX ;PUSH DELAY ONTO STACK
26 27 000F A0000---- E 24 CALL DELAY
28 0014 E600 25 OUT 0,AL
29 0016 EBF4 26 JMP LOOP_
30
31 ----
32 33 28 CODE ENDS
34 29 END START
SUB PROGRAM

8086/8087/8088 MACRO ASSEMBLER DEMO2 09/01/80 PAGE 1

1 ;THIS IS THE DELAY ROUTINE. THE ROUTINE WILL DELAY N*100 MICRO SECONDS. N IS PASSED IN ON THE STACK.
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

NAME DEMO2

PUBLIC DELAY ;DECLARE DELAY AS A GLOBAL NAME

PRO SEGMENT

ASSUME CS:PRO

DELAY PROC FAR ;FAR PROC., PARAMETER AT BP+6

PUSH CX ;SAVE CX, NOW PARAMETER AT BP+8

PUSH AX ;SAVE AX, NOW PARAMETER AT BP+10

PUSH BP

MOV BP,SP

MOV AX,[BP+6] ;GET "N" OFF STACK.

OR AX,AX ;CHECK FOR 0

JZ EXIT ;IF 0, QUIT PROCEDURE

LOOP: MOV CL,78H ;TIME DELAY FOR 100 MICRO SECOND

SHR CL,CL

DEC AX

JNZ LOOP_

EXIT: POP BP

POP AX

POP CX

RET 2

DELAY ENDP

PRO ENDS

END
COMBINING SEGMENTS

ONE LOGICAL SEGMENT

USES NEAR CALLS AND JMPs

ONE PHYSICAL SEGMENT

MOD 1
MOD 2
MOD 3
MOD 4
MOD 5
SEG A
SEG A
SEG A
SEG A
SEG A
COMBINING LOGICAL SEGMENTS INTO A PHYSICAL SEGMENT

```
SEGA SEGMENT PUBLIC
  ASSUME CS:SEGA
  .
SEGA ENDS
END

SEGA SEGMENT PUBLIC
  ASSUME CS:SEGA
  .
SEGA ENDS
END
```

PLACEMENT OF SEGMENTS WITH PUBLICS

ALL REFERENCES ARE WITHIN ONE PHYSICAL SEGMENT; NEAR JUMPS AND CALLS CAN BE USED.
LOC    OBJ    LINE    SOURCE

1    ;THIS IS THE SAME ROUTINE AS SHOWN EARLIER.
2    ;IT NOW CONTAINS A PUBLIC CODE SEGMENT SO THAT
3    ;NEAR CALLS AND JUMPS CAN BE USED.

4
5
6
7
8
9

NAME DEMO

0000 (10 ???)

0014

STACK SEGMENT

10    DW     10 DUP (?)

11
12
13

----

0014

EXTRN DELAY:NEAR ;MUST DECLARE TYPE OF EXTRN

14

THIS WORD

15

MUST DECLARE TYPE OF EXTRN

16

STACK ENDS

17

CODE SEGMENT PUBLIC

18

ASSUME CS:CODE, SS:STACK

19

SECOND EQU 10000 ;DELAY PARAMETER FOR 1 SECOND

20

;PUSH DELAY ONTO STACK

21

START: MOV AX, STACK

22

MOV SS, AX

23

LEA SP, TOP

24

MOV DX, SECOND

25

PUSH DX

26

ASSUME CS:CODE, SS:STACK

27

28

29

2710

STACK SEGMENT

ASSUME CS:CODE, SS:STACK

SECOND EQU 10000 ;DELAY PARAMETER FOR 1 SECOND

SECOND EQU 10000 ;DELAY PARAMETER FOR 1 SECOND

STACK ENDS

STACK ENDS

CODE ENDS

CODE ENDS

END START

END START
;THIS IS THE DELAY ROUTINE. THE ROUTINE WILL DELAY N*1100 MICRO SECONDS. N IS PASSED ON THE STACK.

NAME DEMO2

PUBLIC DELAY ;DELAY IS A PUBLIC NAME

ASSUME CS:CODE ;NEAR PROC., PARAMETER AT BP+4

DELAY PROC NEAR ;SAVE CX, NOW PARAMETER AT BP+6

PUSH CX ;SAVE AX, NOW PARAMETER AT BP+8

PUSH AX

PUSH BP

MOV BP,SP

MOV AX,[BP+8]

OR AX,AX

JZ EXIT

LOOP: MOV CL,78H

SHR CL,CL

DEC AX

JNZ LOOP

EXIT: POP BP

POP AX

POP CX

RET 2

DELAY ENDP

DELAY IS A PUBLIC NAME

SOURCE

,THIS IS THE DELAY ROUTINE. THE ROUTINE WILL DELAY N*1100 MICRO SECONDS. N IS PASSED ON THE STACK.

PROC

;ASSUME CS:CODE

;SAVE AX, NOW PARAMETER AT BP+8

;SAVE BX, NOW PARAMETER AT BP+10

;GET "N" OFF STACK FOR DELAY

;CHECK FOR 0, QUIT PROCEDURE

;TIME DELAY FOR 100 MICRO SECONDS
REFERENCING EXTERNAL DATA (ONE ITEM)

<table>
<thead>
<tr>
<th>NAME</th>
<th>MOD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>SEGMENT PUBLIC BUFFER,WBUFFER</td>
</tr>
<tr>
<td>BUFFER</td>
<td>DB 100 DUP(?)</td>
</tr>
<tr>
<td>WBUFFER</td>
<td>DW 100 DUP(?)</td>
</tr>
<tr>
<td>DATA</td>
<td>ENDS</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

REFERENCING EXTERNAL DATA (MULTIPLE ITEMS)

<table>
<thead>
<tr>
<th>NAME</th>
<th>MOD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>SEGMENT PUBLIC</td>
</tr>
<tr>
<td>EXTRN</td>
<td>BUFFER:BYTE,WBUFFER:WORD</td>
</tr>
<tr>
<td>BEGIN:</td>
<td>MOV AX,DATA</td>
</tr>
<tr>
<td></td>
<td>MOV DS,AX</td>
</tr>
<tr>
<td></td>
<td>MOV AL,BUFFER(SI)</td>
</tr>
<tr>
<td></td>
<td>MOV WBUFFER,DX</td>
</tr>
<tr>
<td>CODE</td>
<td>ENDS</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>
DEVELOPMENT CYCLE WITH LINK86 AND LOC86

LINK86 SYNTAX

- RUN LINK86 FILENAME,FILENAME [...] TO FILENAME [NO MAP]
- PRINT (FILENAME)
- BIND [ORDER(SEGMENTS(SEGNAME),...)]

CONSOLE MESSAGES

OBJECT MODULE

ERROR MESSAGES

BOUND OBJECT MODULE

PRINT FILE "MP1"

PRINT FILE "MP1"

CONSOLE MESSAGES

INVOCATION LINE CONTROLS
LOC86 SYNTAX

- RUN LOC86 FILENAME [TO FILENAME] [PRINT (FILENAME)]
  [NO MAP]
  [ADDRESSES(SEGMENTS( segment [,..] ))]
  [ORDER(SEGMENTS( segment [,..] ))]
  [BOOTSTRAP]
  [START]
  [NAME (MODNAME)]
  [INITCODE [ADDRESS]]

USING LINK86 AND LOC86

THE PROBLEM:
- MESSAG.OBJ IS A PROGRAM THAT USES THE ROUTINES IN READ.OBJ AND PRINT.OBJ TO INPUT AND OUTPUT CHARACTER(s).
- MESSAG.OBJ CONTAINS THE FOLLOWING SEGMENTS; STACK, CODE AND DATA.
- THE SEGMENTS ARE TO BE LOCATED WITH THE STACK SEGMENT AT 200H, THE CODE SEGMENT AT 300H AND THE REMAINING SEGMENTS FOLLOWING IN ANY ARBITRARY ORDER.
THE SOLUTION:

RUN \texttt{LINK86 \texttt{MESSAG.OBJ,READ.OBJ,PRINT.OBJ}}

RUN \texttt{LOC86 \texttt{MESSAG.LNK ADDRESSES(SEGMENTS(STACK(200H),CODE(300H))))}

1. RUN IS NECESSARY FOR SERIES III ONLY.

CLASS EXERCISE 16.1

ADD THE ASSEMBLER DIRECTIVES THAT ARE NECESSARY FOR THESE TWO MODULES TO BE LINKED TOGETHER

\begin{verbatim}
NAME MODA
DATA SEGMENT
USEFUL_DATA DB 7
DATA ENDS
A_CODE SEGMENT
ASSUME CS:A_CODE
HANDY PROC
MOV AX, 0
RET
HANDY ENDP
A_CODE ENDS
END

NAME MODB
B_CODE SEGMENT
ASSUME CS:B_CODE
MOV AL, USEFUL_DATA
CALL HANDY
B_CODE ENDS
END
\end{verbatim}
LIB86 IS A UTILITY PROGRAM TO MANAGE COLLECTIONS OF DEBUGGED MODULES. (SEE THE IAPX 86,88 FAMILY USER'S GUIDE)

THERE ARE OTHER WAYS OF COMBINING AND MANIPULATING SEGMENTS DURING ASSEMBLY, LINK, AND LOCATE. CLASSES AND GROUPS ARE TWO SUCH FACILITIES PROVIDED BY ASM86. CLASSES ARE A WAY OF LOCATING A GROUP OF SEGMENTS AT SOME PHYSICAL ADDRESS. THIS IS MOST OFTEN USED TO SEGREGATE ROM-BASED SEGMENTS FROM RAM-BASED SEGMENTS. GROUPS ARE A WAY OF COMBINING DIFFERENT LOGICAL SEGMENTS INTO ONE PHYSICAL SEGMENT. IT WORKS SIMILARLY TO THE PUBLIC SEGMENT COMBINE TYPE EXCEPT THAT THE COMBINING SEGMENTS MAY HAVE DIFFERENT NAMES. SEE CHAPTER 2 OF THE ASM86 LANGUAGE REFERENCE MANUAL.
CHAPTER 17

INTRODUCTION TO THE iAPX 186, 188 MICROPROCESSOR

• DESCRIPTION
• ENHANCEMENTS
• NEW INSTRUCTIONS
• PERIPHERALS
READY LOGIC

IAPX 86,88

DECODE LOGIC

ADDRESS BUS

INTERRUPT CONTROLLER

TIMER

DMA CONTROLLER

SYSTEM MEMORY

SYSTEM I/O

DATA BUS

TYPICAL IAPX 86,88 SYSTEM

SAME SYSTEM USING THE IAPX 186, 188

ADDRESS BUS

SYSTEM MEMORY

SYSTEM I/O

PCS

CS

DATA BUS

17-1

17-2
IAPX 186 BLOCK DIAGRAM

"A CPU BOARD ON A SINGLE SILICON CHIP"

Combines 10 of the most common IAPX 86 system components into one

IAPX 186 PERIPHERAL INITIALIZATION

- On-chip peripherals are controlled via a block of 18-bit registers
- The block uses 256 bytes of address space
- Registers are memory or I/O mapped
- Peripherals are located at the top of I/O space after reset (OFF00H - OFFFFH)
- 256 byte block is relocatable anywhere in the 1 megabyte memory space or 64K I/O space after initialization
**iAPX 186,188 INTERRUPT CONTROL UNIT BLOCK DIAGRAM**

- TIMER 0
- TIMER 1
- TIMER 2
- DMA 0
- DMA 1

**INTERRUPT CONTROL UNIT BLOCK DIAGRAM**

- TIMER CONTROL REG.
- DMA 0 CONTROL REG.
- DMA 1 CONTROL REG.
- EXT. INPUT 0 CONTROL REG.
- EXT. INPUT 1 CONTROL REG.
- EXT. INPUT 2 CONTROL REG.
- EXT. INPUT 3 CONTROL REG.

**INTERRUPT REQUEST TO PROCESSOR**

**ADDRESS/DATA**

**iAPX 186,188 INTERRUPT CONTROL UNIT**

- 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 TIMER/COUNTER BLOCK DIAGRAM

iAPX 186 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:
  - ALTERNATE COUNT BETWEEN INTERNAL MAX COUNT REGISTERS
  - RETRIGGER ON EXTERNAL EVENT
  - COUNT INTERNAL CLOCK/EXTERNAL PULSES
- TIMER 2 OPTIONS:
  - CLOCK COUNTER (REAL-TIME CLOCK, TIME DELAY)
  - CLOCK PRESCALER FOR OTHER TWO TIMERS
  - DMA REQUEST SOURCE
- MAXIMUM CLOCK RATE: 2 MHz (1/4 CPU CLOCK FREQUENCY)
CHIP SELECT/READY GENERATION BLOCK DIAGRAM

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

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

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

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

- **Generation Logic (Wait states)**

---

**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
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

- 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
**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 OPCODES**
  - PRE-DEFINED INTERRUPT VECTOR

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

- **NEW INSTRUCTIONS**
COMPATIBILITY WITH iAPX 86,88

- OBJECT CODE COMPATIBLE WITH THE iAPX 86,88

- LANGUAGES
  - ASM, PL/M, PASCAL AND FORTRAN INCORPORATE 186 CONTROL TO SUPPORT ENHANCED INSTRUCTION SET.

- DEVELOPMENT SYSTEMS
  - SERIES III
  - INTEGRATED INSTRUMENTATION IN-CIRCUIT EMULATION (I^2ICE)

NEW iAPX 186, 188 INSTRUCTIONS

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

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

**INS**
- DM ← I/O DEVICE
- DI ← I/O (DX)
- DI ← DI +/- INCR

**OUTS**
- SI ← I/O DEVICE
- DI ← I/O (DX)
- SI ← SI +/- INCR

**INSB (BYTE TRANSFER)**
- + WHEN DF = 0 (CLD)
- - WHEN DF = 1 (STD)

**OUTSB (BYTE TRANSFER)**
- 1 FOR BYTE TRANSFERS
- 2 FOR WORD TRANSFERS

**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
FOR MORE INFORMATION...

INTRODUCTION TO THE iAPX 186/188
- CHAPTER 5, iAPX 86/88, 186/188 USER'S MANUAL
- AP-186, INTRODUCTION TO THE 80186 MICROPROCESSOR
DAY 5 OBJECTIVES

BY THE TIME YOU FINISH TODAY YOU WILL:

* DEFINE MULTIPROCESSING AND COPROCESSING

* DESCRIBE THE SIGNALS USED TO INTERFACE TO THE MULTIBUS

* DESCRIBE THE SIGNALS USED TO INTERFACE AN 80186 TO EXTERNAL HARDWARE

* DESCRIBE THE BASIC FUNCTIONS OF THE iAPX 286 AND iAPX 386
CHAPTER 18
MULTIBUS SYSTEM INTERFACE

- DESIGN CONSIDERATIONS
- HARDWARE INTERFACE TO THE MULTIBUS
- BUS ARBITRATION
- LOCK INSTRUCTION PREFIX
- BYTE SWAP BUFFER
FUNCTIONAL PARTITIONING SUPPORTS MULTIPROCESSING:

- CRT TERMINAL
- LOCAL MEMORY
- HARD DISK
- MODEM
- 8081
- 8086
- 8087
- 8059
- SYSTEM INTERFACE
- SYSTEM MEMORY

MULTI PROCESSOR

* REFERS TO SYSTEM CONTAINING MORE THAN ONE CPU WHERE ONE CPU IS USUALLY THE "MAIN" CPU AND OTHER CPU'S PERFORM SPECIAL TASKS

* EACH CPU HAS ITS OWN PROGRAM AND OPERATES INDEPENDENTLY

* EACH CPU HAS ACCESS TO GLOBAL RESOURCES
CO-PROCESSORS

* SPECIAL CASE OF MULTIPROCESSING
* SPECIAL PURPOSE PROCESSORS THAT ENHANCE THE HARDWARE CAPABILITIES OF THE 8086
* SHARE COMMON PROGRAM WITH HOST PROCESSOR EXECUTING CERTAIN INSTRUCTIONS
* OPERATE IN A LOCAL CONFIGURATION WITH THE 8086 (SHARE COMMON DATA, ADDRESS, AND CONTROL BUSSES)

NUMERIC PROCESSOR EXTENSION

* COPROCESSOR
* INTEGRAL PART OF THE iAPX 86 AND iAPX 88 ARCHITECTURE
* 68 NUMERIC INSTRUCTIONS
* MULTIPLE AND MIXED MODE DATA TYPE CAPABILITIES (INTEGER, REAL, BCD)
* FULL IMPLEMENTATION OF THE IEEE FLOATING POINT STANDARD
* AUTOMATIC EXCEPTION DETECTION AND RECOVERY
* COMPLETE HARDWARE/SOFTWARE TRANSPARENCY
* EIGHT 80-BIT INTERNAL REGISTERS
THE 8087 CAN BE VIEWED AS AN ARCHITECTURAL EXTENSION OF AN 8086/8088.

TO USE THE 8087, ADDITIONAL OP CODES AND OPERANDS ARE INCLUDED IN THE 8086/8088 INSTRUCTION SET.
DATA FORMATS FOR MEMORY OPERANDS

WORD INTEGER
5 bits
MAGNITUDE (TWO'S COMPLEMENT)

SHORT INTEGER
5 bits
MAGNITUDE (TWO'S COMPLEMENT)

LONG INTEGER
6 bits
MAGNITUDE (TWO'S COMPLEMENT)

PACKED DECIMAL
5 bits
X \( \times \)
MAGNITUDE

SHORT REAL
31 bits
BIASED EXPONENT
SIGNIFICAND

LONG REAL
63 bits
BIASED EXPONENT
SIGNIFICAND

TEMPORARY REAL
79 bits
BIASED EXPONENT
SIGNIFICAND
iAPX 86/20, 88/20 INTERCONNECT

- HOST CPU MUST BE IN MAX MODE TO PROVIDE INTERFACE
- RQ/GT, QS0-QS1, TEST LINES USED FOR COMMUNICATION AND SYNCRONIZATION
**QUEUE STATUS LINES**

**QS₁, QS₀** - QUEUE STATUS LINES: INDICATE THE INSTRUCTION QUEUE STATUS AS FOLLOWS:

<table>
<thead>
<tr>
<th>QS₁</th>
<th>QS₀</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>NO OPERATION</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>FIRST BYTE OF OPCODE</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>EMPTY THE QUEUE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SUBSEQUENT BYTE</td>
</tr>
</tbody>
</table>

**TEST PIN**

TEST - USED BY WAIT INSTRUCTION TO SYNCHRONIZE PROCESSORS
IF TEST PIN IS LOW, EXECUTE CONTINUES
IF TEST PIN IS HIGH, CPU ENTERS AN IDLE STATE
8087 CO-PROCESSOR OPERATION

8087 NUMERICAL DATA PROCESSOR

BEGIN

CONTINUE UNTIL 8087 RESULT IS NEEDED

WAIT

TEST

CONTINUE

ESCAPE

8086

REQUEST/GRANT LINES

\begin{align*}
\text{RQ/GT}_0 \\
\text{RQ/GT}_1
\end{align*}

REQUEST GRANT: BIDIRECTIONAL HANDSHAKE LINES ALLOWS UP TO TWO SEPERATE DEVICES CONTROL OF THE BUSES
EXECUTION TIME FOR SELECTED iAPX 86/20 INSTRUCTIONS

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>APPROXIMATE EXECUTION TIME (µS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IAPX 86/20 (5 MHz CLOCK)</td>
</tr>
<tr>
<td>ADD/SUBTRACT MAGNITUDE</td>
<td>14/18</td>
</tr>
<tr>
<td>MULTIPLY (SINGLE PRECISION)</td>
<td>18</td>
</tr>
<tr>
<td>MULTIPLY (DOUBLE PRECISION)</td>
<td>27</td>
</tr>
<tr>
<td>DIVIDE</td>
<td>39</td>
</tr>
<tr>
<td>COMPARE</td>
<td>10</td>
</tr>
<tr>
<td>LOAD (SINGLE PRECISION)</td>
<td>9</td>
</tr>
<tr>
<td>STORE (SINGLE PRECISION)</td>
<td>17</td>
</tr>
<tr>
<td>SQUARE ROOT</td>
<td>36</td>
</tr>
<tr>
<td>TANGENT</td>
<td>110</td>
</tr>
<tr>
<td>EXPONENTIATION</td>
<td>130</td>
</tr>
</tbody>
</table>

18-13

8089 IO PROCESSOR

* THE I/O PROCESSOR CONTROLS ALL I/O IN THE SYSTEM
* BOTH PROCESSORS OPERATE IN PARALLEL
* SYSTEM THROUGHPUT IS ENHANCED
I/O PROCESSOR FEATURES

- 2 INDEPENDENT CHANNELS
- 1 MEGABYTE SYSTEM SPACE, 64K I/O SPACE
- 2 LOGICAL BUSSES CAN BE TREATED AS 8 OR 16 OR BOTH TO MATCH PERIPHERALS TO SYSTEM
- CHANNEL PROGRAM STORE CAN BE ON SYSTEM OR LOCAL BUS
- INSTRUCTION SET TAILORED FOR I/O FUNCTIONS

I/O PROCESSOR BLOCK DIAGRAM

- INFORMATION FLOWS THROUGH IOP
- INSTRUCTIONS APPLY TO I/O OR SYSTEM
- 2 LOGICAL BUSSES
- 2 CHANNELS
- 2 REGISTER SETS
- 2 INSTRUCTION POINTERS
LOCAL CONFIGURATION
MINIMUM BOARD SPACE AND COST

REMOTE CONFIGURATION ALLOWS PARALLEL PROCESSING
DMA FUNCTIONS

- MEMORY TO MEMORY, I/O TO I/O IN ADDITION TO MEMORY TO I/O
- MASKED/COMPARE FOR DATA PATTERN AS TRANSFER OCCURS
  - 8-BIT MASK, 8-BIT COMPARE
- TRANSLATE DURING TRANSFER
  - BYTE TRANSLATED THROUGH 256-BYTE LOOKUP TABLE
- VERSATILE TERMINATION CONDITIONS
  - BYTE COUNT EXPIRED (UP TO 64K)
  - EXTERNAL SOURCE
  - MASKED/COMPARE PASSES OR FAILS
  - SINGLE BYTE

8089 PERFORMANCE

<table>
<thead>
<tr>
<th></th>
<th>5 MHz</th>
<th>8 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA TRANSFER</td>
<td>1.25 Mbyte</td>
<td>2.0 Mbyte</td>
</tr>
<tr>
<td>(16 BIT TRANSFERS)</td>
<td></td>
<td></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>8 BIT/16 BIT SOURCE</td>
<td></td>
<td></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) SINGLE CHANNEL/DUAL CHANNEL</td>
<td>1.0/2.2μs</td>
<td>0.625/1.375 μs</td>
</tr>
</tbody>
</table>
OPERATING SYSTEM Firmware Component

* 16kbyte control store

* Programmable interrupt controller managed by OS software

* 3 programmable timers
  - SYSTEM (8254 Rate Gen Mode)
  - DELAY (8254 Count Mode)
  - BAUD (8254 Square Wave Mode)

80130 Features

<table>
<thead>
<tr>
<th>HARDWARE</th>
<th>SOFTWARE</th>
</tr>
</thead>
<tbody>
<tr>
<td>□ 128 K-bit kernal control store</td>
<td>□ Task management</td>
</tr>
<tr>
<td>□ Programmable interrupt controller</td>
<td>□ Intertask communication and synchronization</td>
</tr>
<tr>
<td>□ System timer</td>
<td>□ Mutual exclusion control</td>
</tr>
<tr>
<td>□ Delay timer</td>
<td>□ Interrupt management</td>
</tr>
<tr>
<td>□ Baud-rate generator</td>
<td>□ Free memory management/system partitioning</td>
</tr>
</tbody>
</table>
FOR MORE INFORMATION ...

8087 MATH COPROCESSOR
- CHAPTER 6, IAPX 86/88, 186/188 USER'S MANUAL
- CHAPTER 6, ASM86 LANGUAGE REFERENCE MANUAL

8089 I/O PROCESSOR
- CHAPTER 7, IAPX 86/88, 186/188 USER'S MANUAL

80130 OPERATING SYSTEM FIRMWARE COMPONENT
- CHAPTER 8, IAPX 86/88, 186/188 USER'S MANUAL

RELATED TOPICS ...

ICE86A SUPPORTS THE 8087 FOR DEBUGGING PURPOSES. SEE THE ICE86A OPERATOR'S MANUAL. AN ICE86 CAN BE UPGRADED TO AN ICE86A.

RBF89 (REAL-TIME BREAKPOINT FACILITY) IS A DEBUGGING TOOL FOR THE 8089 AND WORKS IN CONJUNCTION WITH ICE86(A).
CHAPTER 19

MULTI AND COPROCESSOR

- 8087 NUMERIC DATA PROCESSOR
- 8089 I/O PROCESSOR
- 80130 OPERATING SYSTEM
WHAT IS THE MULTIBUS SYSTEM INTERFACE?

- 16 MEGABYTE ADDRESS SPACE
- IEEE STANDARD (IEEE 796)
- INDUSTRY STANDARD
- OVER 40 VENDORS OF MULTIBUS BOARDS
- OVER 40 BOARDS AVAILABLE FROM INTEL

WHY USE THE MULTIBUS SYSTEM INTERFACE?

- MODULARIZE HARDWARE/DISTRIBUTED PROCESSING
- SHORTEN DESIGN TIME
- REDUCE COST OF DESIGN AND TEST
- FLEXIBLE
  - SYSTEM CAN BE QUICKLY RECONFIGURED
  - EASY TO ADD MORE PROCESSING POWER, MEMORY OR IO
  - SIMPLIFIES REPAIR
MODULARIZE HARDWARE/DISTRIBUTED PROCESSING

- HARDWARE MODULES CAN BE DEVELOPED INDEPENDENTLY
- CONCURRENT PROCESSING ACHIEVES HIGHER THROUGHPUT
- PROCESSORS COMMUNICATE THROUGH SHARED MEMORY
- HARDWARE MODULES CAN BE REUSED IN FUTURE DESIGNS

REDUCE COST/SHORTEN DESIGN TIME

- CONFIGURE SYSTEM COMPLETELY FROM AVAILABLE BOARDS
- DESIGN CUSTOM IO BOARDS FOR YOUR APPLICATION
MAKE/BUY COMPARISON

CROSSOVER POINT

COST PER BOARD

1K-3K

TOTAL NUMBER OF BOARDS

TYPES OF BUS MASTERS

BASIC MASTER

MASTER WITH RESIDENT BUS

MASTER WITH DUAL-PORTED RAM

MULTIBUS

NOT VERY COMMON

iSBC 86/05 BOARD

iSBC 86/12A BOARD

WHY WOULD THE BASIC MASTER NOT BE VERY COMMON?
LOCAL BUS INTERFACE

(REVIEW)

LOCAL (PRIVATE) I/O AND MEMORY

8259A PROGRAMMABLE INTERRUPT CONTROLLER

8286 BUS CONTROLLER

6282/83 LATCHES

8286/87 TRANSCEIVERS

MULTIMASTER

LOCAL BUS

8086/8088 CPU

CONTROL LINES

ADDRESS LINES

DATA LINES

SYSTEM BUS INTERFACE

8259A PROGRAMMABLE INTERRUPT CONTROLLER

8288/87 TRANSCEIVERS

8282/83 LATCHES

8286 BUS CONTROLLER

8288/87 ARBITER

MULTIMASTER

LOCAL BUS

8086/8088 CPU

ARBITRATION LINES

CONTROL LINES

ADDRESS LINES

DATA LINES

SYSTEM BUS GLOBAL RESOURCES
• 8289 RESB PIN TIED LOW (NO RESIDENT BUS)
• ALL MEMORY AND I/O CYCLES REQUIRE MULTIBUS ACCESS
• ONLY WHEN 8289 GETS CONTROL OF BUS DOES IT ENABLE BUS CONTROLLER (8288) AND ADDRESS LATCHES (8283,S)
- 8289 RESB PIN TIED HIGH (RESIDENT BUS PRESENT)
- ADDRESS DECODING SELECTS THE SYSTEM BUS OR RESIDENT BUS VIA 8289 PIN SYSB/RESB.
HOW CAN WE PREVENT TWO MASTERS FROM ACCESSING THE BUS AT THE SAME TIME?

SERIAL PRIORITY RESOLVING

- A MASTER CAN TAKE THE BUS WHEN
  \( BPRN \) IS LOW (BUS PRIORITY IN)
  NO HIGHER PRIORITY MASTER NEEDS THE BUS
  \( BUSY \) IS HIGH
  THE BUS ISN’T BEING USED NOW

NOTE:
THERE IS A MAXIMUM OF 3 MASTERS WHEN USING THE SERIAL PRIORITY RESOLVING TECHNIQUE
SERIAL PRIORITY RESOLVING

- A MASTER REQUESTS THE BUS BY DRIVING BPRO HIGH (BUS PRIORITY OUT)
  ALL LOWER PRIORITY MASTERS GET OFF THE BUS CBRQ LOW (COMMON BUS REQUEST)
  IF A HIGHER PRIORITY MASTER HAS THE BUS AND DOES NOT NEED IT, RELEASE THE BUS.

SERIAL PRIORITY RESOLVING

- A MASTER WILL RELEASE THE BUS WHEN BPRN GOES HIGH
  OR A HIGHER PRIORITY MASTER WANTS THE BUS CBRQ GOES LOW AND CURRENT MASTER IS NOT USING BUS
  THE ARBITER NORMALLY DOES NOT SURRENDER THE SYSTEM BUS, UNLESS ANOTHER ARBITER IS REQUESTING ITS USE.
PARALLEL PRIORITY RESOLVING TECHNIQUE

ADVANTAGES
- CAN HANDLE ANY NUMBER OF MASTERS
- ALLOW COMPLEX PRIORITY ASSIGNMENT (E.G., ROUND ROBIN, ROTATING, ETC.)

DISADVANTAGE
- REQUIRES EXTRA, USER-SUPPLIED HARDWARE.
PROBLEM:
8086 #2 STARTS READING MESSAGE
8086 #1 STARTS UPDATING MESSAGE BEFORE #2 IS FINISHED
8086 #2 GETS INVALID MESSAGE

SOLUTION:
USE ONE SHARED MEMORY LOCATION AS A FLAG (SEMAPHORE), WHICH INDICATES IF MESSAGE AREA IS BEING USED.
USING A SEMAPHORE WITH THE LOCK INSTRUCTION PREFIX

```
LOOP1:  MOV  AL, 1
        XCHG  SEMA4, AL
        CMP  AL, 1
        JE   LOOP1

; ACCESS MESSAGE

MOV  SEMA4, 0
: RELEASE MESSAGE AREA
```

MULTIBUS INTERFACE

1-MESSAGE AREA IS BEING USED
0-MESSAGE AREA IS NOT BEING USED

LOCKING THE MULTIBUS

- THE 8086 WILL ASSERT ITS LOCK PIN DURING ANY INSTRUCTION PRECEDED BY A LOCK PREFIX.
- THE 8289 WILL NOT RELEASE THE BUS AS LONG AS ITS LOCK PIN IS ASSERTED
SHARING RESOURCES
BETWEEN 8 AND 16 BIT BOARDS

PROBLEM: THE 8086 TRANSFERS ODD ADDRESSED BYTES ON THE UPPER 8 DATA LINES. THE 8085 TRANSFERS ALL DATA ON THE LOWER 8 DATA LINES.

SOLUTION: USE BYTE-SWAP BUFFER SO THAT ALL BYTE TRANSFERS ON THE MULTIBUS INTERFACE USE THE LOWER 8 DATA LINES.

- ALL INTEL MEMORY BOARDS AND 16 BIT PROCESSOR BOARDS HAVE BYTE-SWAP BUFFERS
- INTEL 8 AND 16 BIT BOARDS ARE COMPATIBLE
- TO BE COMPATIBLE WITH INTEL BOARDS, USER BOARDS SHOULD HAVE BYTE-SWAP BUFFERS

BYTE SWAP BUFFER

<table>
<thead>
<tr>
<th>16-BIT DEVICE</th>
<th>MULTIBUS DATA PATH</th>
<th>BYTE-SWAP</th>
<th>AG</th>
<th>MULTIBUS TRANSFER DATA PATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOW, EVEN BYTES</td>
<td>D0-D7</td>
<td>HI</td>
<td>LO</td>
<td>D0-D7</td>
</tr>
<tr>
<td>HIGH, ODD BYTES</td>
<td>D8-D15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOW, EVEN BYTES</td>
<td>D0-D7</td>
<td>LO</td>
<td>HI</td>
<td>8-BIT D0-D7</td>
</tr>
<tr>
<td>HIGH, ODD BYTES</td>
<td>D8-D15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOW, EVEN BYTES</td>
<td>D0-D7</td>
<td>LO</td>
<td>LO</td>
<td>16-BIT D0-D15</td>
</tr>
<tr>
<td>HIGH, ODD BYTES</td>
<td>D8-D15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CLASS EXERCISE 19.1

DIRECTIONS: EACH ITEM IN THE FOLLOWING PROBLEM REPRESENTS A STEP THAT WOULD BE REQUIRED IN A MULTIBUS SYSTEM AS SHOWN ON PAGE 16-13 WITH 3 BUS MASTERS. IF BUS MASTER 3 (BM3) WAS CURRENTLY CONTROLLING THE MULTIBUS AND BM2 WANTED ACCESS TO THE MULTIBUS, IN THE SPACE PROVIDED, NUMBER EACH ITEM SO THEY OCCUR IN THE PROPER ORDER. THE FIRST STEP HAS BEEN NUMBERED CORRECTLY AS AN EXAMPLE.

- BM3 DRIVES BUSY HIGH
- BM2 ISSUES CBRQ LOW
- BM2 DRIVES BPRO HIGH
- BM2 TAKES OVER BUS, DRIVES BUSY LOW
- BM3 SEES CBRQ LOW
- BM3 SEES BPRN HIGH

FOR MORE INFORMATION . . .

MULTIBUS ARCHITECTURE
- CHAPTER 4, iAPX 86/88, 186/188 USER’S MANUAL

8289 BUS ARBITER
- CHAPTER 4, iAPX 86/88, 186/188 USER’S MANUAL

LOCK PIN OPERATION
- CHAPTER 4, iAPX 86/88, 186/188 USER’S MANUAL
CHAPTER 20

iAPX 186,188 HARDWARE INTERFACE

- BUS INTERFACE
- CLOCK GENERATOR
- INTERNAL PERIPHERALS INTERFACE
- DIFFERENCES
BUS INTERFACING

80186 BUS SIGNALS

ADDRESS/DATA
ADDRESS/STATUS
CO-PROCESSOR CONTROL
LOCAL BUS ARBITRATION
LOCAL BUS CONTROL
MULTI-MASTER BUS
STATUS INFORMATION

ADO - AD15
A16/S3 - A19/S6, BHE/S7
TEST
HOLD, HLDA
ALE, RD, WR, DT/R, DEN
LOCK
SO - S2
80186 CONTROL SIGNAL DIFFERENCES

PROVIDES BOTH LOCAL BUS SIGNALS AND STATUS OUTPUTS
NO SEPARATE I/O AND MEMORY READ AND WRITE SIGNALS.
THE WR SIGNAL IS AN EARLY WRITE SIGNAL
ALE GOES ACTIVE A CLOCK PHASE EARLIER
QUEUE STATUS IS PROVIDED IF RD IS TIED TO GROUND
QUEUE STATUS IS AVAILABLE A CLOCK PHASE EARLIER
HOLD/HLDA IS PROVIDED RATHER THAN RQ/GT
S3 – S6 PROVIDE DIFFERENT INFORMATION THAN 8086
THE OUTPUT DRIVERS WILL DRIVE DOUBLE THE LOAD
THE 80186 PROVIDES BOTH LOCAL BUS SIGNALS AND SYSTEM BUS SIGNALS

LOCAL MEMORY AND I/O

80186
WR
RD
ALE
DT/R
DEN
LCS
UCS
PCS

SO-S2
LOCK

8288
SO-S2

8289
SO-S2
SYSB/RESB
LOCK

SYSTEM RESOURCES

GENERATING SEPARATE I/O AND MEMORY READ SIGNALS

S2
ALE

LATCH
D
Q
STB

I/O READ

RD

MEMORY READ
SYNTHESIZING DELAYED WRITE ON 80186

80186 QUEUE STATUS MODE
S3 - S6 STATUS SIGNAL DIFFERENCES

**8086**

- S3 - S4 SEGMENT REGISTER USED
- S5 INTERRUPT ENABLE FLAG CONDITION
- S6 LOW IF CPU BUS CYCLE
  - HIGH IF DMA BUS CYCLE

**80186**

- S3 - S4 SEGMENT REGISTER USED
- S5 INTERRUPT ENABLE FLAG CONDITION
- S6 LOW IF CPU BUS CYCLE
  - HIGH IF DMA BUS CYCLE

---

CLOCK GENERATOR

- CLOCK GENERATOR
- EXECUTION ALU GENERAL REGISTERS
- DMA UNIT
- TIMER UNIT
- BUS INTERFACE UNIT
- SEGMENT REGISTERS QUEUE
- CHIP-SELECT UNIT
- INTERRUPT CONTROL UNIT

- RES
- RESE
- SRDY
- ARDY
- DT/R
- DEN
- RD
- WR
- ALE
- HOLD
- HLD

- A15
- A19/56
- AD0-AD15
- A18/63
- A19/56
- MCSB-3
- UCS LCS PCSB-8
- NMI

- INT3/INT1
- INT2/INTA
- INTI
- INTB
80186 INTERNAL CLOCK GENERATOR

- Generates a main clock for integrated components and system
- Can use a crystal or external frequency source
- Generates a synchronized system reset
- Provides a synchronous and an asynchronous ready input

80186 CLOCK GENERATOR BLOCK DIAGRAM

- Crystal Osc.
- 2
- Ready generation
- Reset circuit
- CPU clock and clockout
- CPU ready
- CPU reset and reset output
80186 AND 8284A CLOCK DIFFERENCES

NO OSCILLATOR OUTPUT IS AVAILABLE FROM THE 80186

THE 80186 DOES NOT PROVIDE A PCLK OUTPUT

THE 80186 CLOCKOUT HAS A 50% DUTY CYCLE CLOCK AND THE 8284A CLK OUTPUT HAS A 33% DUTY CYCLE

THE CRYSTAL OR EXTERNAL OSCILLATOR USED BY THE 80186 IS TWICE THE CPU CLOCK FREQUENCY WHILE ON THE 8284A IT IS THREE TIMES THE CPU CLOCK FREQUENCY

EFFECT OF RESET

SAME EFFECT AS IN THE 8086 PLUS EFFECTS THE INTERNAL PERIPHERALS AS follows

RELOCATION REGISTER = 20FFH

INTERNAL PERIPHERALS ARE ADDRESSED AT THE VERY TOP (FFFFH TO FFOOH) OF THE I/O SPACE

UMCS = FFFBH

UCS LINE WILL PROVIDE A CHIP SELECT FOR THE UPPER 1K BLOCK OF MEMORY WITH THREE WAIT STATES WITH EXTERNAL READY CONSIDERED

THE REST OF THE INTERNAL PERIPHERALS ARE RESET AND ARE INACTIVE UNTIL PROGRAMMED
READY SIGNALS

SYSTEM CONSISTS OF TWO BUSSES - A LOCAL BUS AND A SYSTEM BUS

THE SYSTEM BUS IS ASYNCHRONOUS AND NORMALLY NOT READY

THE LOCAL BUS OPERATES SYNCHRONOUS TO THE PROCESSOR

ARDY WOULD BE USED FOR THE SYSTEM BUS

SRDY AND/OR THE 80186 CHIP SELECT LINES WITH THE
PROGRAMMABLE WAIT STATES WOULD BE USED FOR THE LOCAL BUS
DMA CONTROLLER INTERFACE

20-17

USING DMA REQUEST AND SENDING AN ACKNOWLEDGE

20-18
CHIP SELECTIONS

CLOCK GENERATOR
EXECUTION UNIT
ALU GENERAL REGISTERS
DMA UNIT
TIMER UNIT
BLOCK INTERFACE UNIT
SEGMENT REGISTERS
QUEUE
CHIP-SELECT UNIT
MC6851-9 UCS LCB PC88-8
INTERNAL BUS

SRDY
ARDY
RESET
GND
Vcc
DROD DRO1
TMR IN TMR OUT TMR IN TMR OUT
LOCK TEST S$-S2 BHE
A0-A12 A16/18-A19/20
CHIP SELECTS
TMR IN TMR OUT

USING 80186 CHIP SELECTS

A0 - A19, BHE

A1 - A13

2764
CE
OE

A1 - A13

2764
CE
OE

20-19

20-20
NON-iRMX86 DIRECT INPUT MODE AND CASCADE MODE

80186 ARDY
INT0
INT2
INT1
INT3
AD0-AD7
RD
WR
PCSA

OTHER ARDY

10 EXTERNAL INTERRUPTS

8259A-2
INT
INTA
D0-D7
RD
WR
CS

+5V

iRMX86 MODE

80186
INT#
INT2
ALE
INT1
INT3

8259A
INT
INTA
CAS# CAS1 CAS2
8205
CE1
CE2
E3
CE7

+5V

A1
A2

iRMX86 MODE INTERFACE TO 80130

80186
ALE
ADO-AD15
CLK
MMCS2
PC83
SO-52
BHE
INT0
INT3
INT2
INT1

80130
ADO-AD15
CLK
MEMCS
IRO-
IOCS
IR7
SO-52
BHE
INT

ADOR
ADO-AD15

8205
E2
E3
E1
7

3A8-A10
INTERRUPT REQUESTS

80186/80188 BLOCK DIAGRAM

CLOCK GENERATOR
EXECUTION UNIT
ALU
GENERAL REGISTERS
DMA UNIT
TIMER UNIT

BUS INTERFACE UNIT
SEGMENT REGISTERS
QUEUE

CHIP-SELECT UNIT
MC50-3
UCS LCS
PC88-6

CHIP-SELECT UNIT

INT3/INTA1
INT2/INTA9
INT1
INT0

LOCK TEST S0-S2 BHE
ADO-
AD15
A18/A19/68

DT/R
DEN
RD
WR
ALE
HOLD
HLDA

RST
RESET
SRY
ARDY
80186/80188 DIFFERENCES

80186 HAS A 6 BYTE QUEUE AND THE 80188 HAS A 4 BYTE QUEUE.

AD8 – AD15 ON THE 80186 ARE TRANSFORMED TO A8 – A15 ON THE 80188 AND ARE VALID THROUGHOUT THE BUS CYCLE.

BHE/S7 IS ALWAYS DEFINED HIGH BY THE 80188.

THE DMA CONTROLLER ONLY PERFORMS BYTE TRANSFERS.

EXECUTION TIMES FOR MEMORY ACCESSES ON THE 80188 ARE INCREASED BECAUSE OF 8-BIT EXTERNAL DATA BUS. INTERNAL DATA BUS IS STILL 16-BITS.
TYPICAL iAPX 186, 188 COMPUTER SYSTEM

- BHE NOT IMPLEMENTED ON iAPX 188
FOR MORE INFORMATION

INTRODUCTION TO THE 80186 MICROPROCESSOR
AP-186
CHAPTER 21
THE iAPX 286 AND iAPX 386 MICROPROCESSORS
• DESCRIPTION
• ENHANCEMENTS
**iAPX 286 MICROSYSTEM SOLUTION**

TWO OPERATION MODES TO MATCH YOUR NEEDS:

- REAL ADDRESS MODE
  - PROGRAM ENVIRONMENT IDENTICAL TO iAPX 86, 186
  - HIGHEST-PERFORMANCE SYSTEM (6 TIMES iAPX 86)
  - LARGEST BASE OF AVAILABLE SOFTWARE (iAPX 88, 86, 186)

- PROTECTED VIRTUAL ADDRESS MODE
  - SAME PERFORMANCE AS REAL MODE PLUS NEW FEATURES:
    - VIRTUAL MEMORY
    - SOFTWARE PROTECTION
    - PERFORMANCE BOOST FOR PROTECTED O.S.
  - SIMPLE MIGRATION PATH FOR LARGE BASE OF APPLICATIONS SOFTWARE

**iAPX 286 REAL ADDRESS MODE**

- OPERATES EXACTLY AS iAPX 86 (PLUS UP TO 6 TIMES PERFORMANCE)
- 1 MBYTE ADDRESS SPACE
- EXECUTES SAME iAPX 86 INSTRUCTION SET (BASIC SET)
- HAS ALL iAPX 186 INSTRUCTION EXTENSIONS
- SEGMENTATION SAME AS iAPX 86
- FULLY SOFTWARE COMPATIBLE WITH iAPX 86 AND iAPX 186 INCLUDING ADVANCED NUMERICS
iAPX 286 PROTECTED VIRTUAL MODE SATISFIES SYSTEM REQUIREMENTS

- ADVANCED MEMORY MANAGEMENT WITH NO PERFORMANCE PENALTY
  - 16 MBYTE PHYSICAL ADDRESS
  - 1 BILLION BYTE VIRTUAL ADDRESS/TASK
  - VIRTUAL MEMORY SUPPORT—INSTRUCTION RESTART

- ADVANCED PROTECTION MECHANISM
  - AUTOMATIC INTEGRITY CHECKS (CODE AND DATA TYPING, SIZE, AND PRIVILEGE)
  - TASK ISOLATION CONTROL (USER/USER ISOLATION AND SHARING)
  - MULTILEVEL PROTECTION—UP TO 4 LEVELS—(USER/O.S. ISOLATION AND ACCESS CONTROL)

- OPERATING SYSTEM PERFORMANCE ENHANCEMENTS
  - MULTITASKING (INTEGRATED TASK SWITCH)
  - ABILITY TO PROVIDE DIRECT ACCESS TO O.S. FUNCTIONS

- EXECUTES SAME BASIC IAPX 86 AND IAPX 186 INSTRUCTION SET INCLUDING ADVANCED NUMERICS

MEMORY PROTECTION

- Level 0 (most privileged)
- Level 1
- Level 2
- Level 3 (least privileged)
- Privilege Level Isolation

Task A
Task B
Task C
Task Isolation
PIPILED ARCHITECTURE

ACCESSING MEMORY

REAL ADDRESS MODE

- RELATIVELY SIMPLE ADDRESSING MECHANISM
- DIRECT RELATIONSHIP BETWEEN SEGMENT REGISTER CONTENTS AND SEGMENT ADDRESS
INSTRUCTION

SEGMENT SELECTOR

OFFSET

ADDRESSING MECHANISM

MORE SOPHISTICATED ADDRESSING MECHANISM

UTILIZES MEMORY MANAGEMENT AND PROTECTION MECHANISMS

ADDRESS STILL CONSISTS OF 32 bit QUANTITY SELECTOR: OFFSET

24 bit ADDRESS SUPPORTS 16Mb MEMORY

SEGMENT SELECTOR "SELECTS" A PARTICULAR DESCRIPTOR FROM A DESCRIPTOR TABLE

DESCRIPTOR PROVIDES SEGMENT BASE AND LIMIT
DESCRIPTOR REGISTER LOADING

• DESCRIPTORS ARE AUTOMATICALLY LOADED WHENEVER A SEGMENT REGISTER IS LOADED.

• NO NEW INSTRUCTIONS ARE NEEDED.

EXAMPLES:

MOV DS, AX
POP ES
JMP SELECTOR, OFFSET
CALL SELECTOR, OFFSET
RET
LDS SI, POINTER VARIABLE

• THESE ARE THE ONLY TYPES OF INSTRUCTIONS THAT AFFECT THE PERFORMANCE OF REAL ADDRESS MODE VERSUS PVAM

BEYOND

286
PERFORMANCE
iAPX 386

- EVOLUTION OF THE iAPX 86 FAMILY TO THE FUTURE
  - IMPROVED PERFORMANCE
  - INCREASED FUNCTIONALITY
  - PRESERVATION OF 86, 186 AND 286 SOFTWARE INVESTMENT

iAPX 386 FUNCTIONALITY

- FULL 32 BIT ADDRESS AND DATA
- 286 MODEL PROTECTED SEGMENTATION PLUS OPTIONAL PAGING
- INSTRUCTION SET ENHANCEMENTS
  - BIT OPERATIONS, POINTER OPERATIONS, ETC
- EXTENDED NUMERICS COPROCESSOR (80387)
  - INCREASED PERFORMANCE
  - ENHANCED TRIGONOMETRICS
- IMPROVED SYSTEM RELIABILITY
ARCHITECTURE PLANNED FOR EVOLUTION

1ST GENERATION

8086
8088

2ND GENERATION

286
186
188

3RD GENERATION

386
LAB 1

When you finish this lab you will be able to:

* Write a simple but complete assembly language program using an editor
* Use ASM86 to create object code from a text file
* Use LINK86 to make a run time locatable file
* Execute the program using the SERIES III development system

PROBLEM (part 1)

This lab requires an INTELLEC SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM with an attached I/O box containing LED's and switches. You are to write a program that will input the value on the switches wired to port 1, and then output this value to the LED's attached to port 0. The program should do this continuously.

When you have a written solution, continue with the lab.

PREPARING THE USER DISKETTE

If you are using the network, follow the directions given by your instructor, skip this section and go to CREATING A SOURCE FILE.

Your instructor has two floppy diskettes that you will use for all the labs during the week. One of the diskettes is a system diskette that has the ISIS-II operating system on it. To use the Development System, you must first boot up the system with a system diskette. To boot the system, first power on everything and then place the diskette marked SYSTEM DISKETTE into drive 0 of the development system (this is the right hand slot of the drive unit). Place the diskette into the drive such that the label is to the left or facing upwards (it depends on how the disk slot is orientated). Now press the button marked RESET. The system should sign on:

    ISIS-II V x.y

The "-" tells you that you are in ISIS and that any ISIS command may now be entered.

Now place the other diskette into drive 1. This is your diskette that you will use for the entire week. First initialize the diskette in drive 1 with an ISIS command named IDISK. This command is used typically only once to initialize a new diskette. The command formats the
diskette to make it compatible with ISIS and "erases" everything that was on the diskette previously (so only use the IDISK command once this week). To format your diskette enter the IDISK command exactly as it appears below followed by return.

IDISK :F1:MYDISK

The ":F1:" tells ISIS that you want drive 1 (drive Ø is accessed by :FØ:). The name is arbitrary. The return key enters the command. Once the command is done, ISIS will return with a "-". If you make a mistake while typing, use the key labeled "Rubout" to delete the last character you entered.

CREATING A SOURCE FILE

Now you are ready to create a disk file of your program using an editor. If you wish to use AEDIT and you are unfamiliar with it, go to the optional AEDIT Basics lab in this appendix.

To invoke AEDIT, type:

AEDIT :F1:LAB1.ASM

While you are creating this file, it would be good practice to keep your AEDIT Pocket Reference card with you to help you with unfamiliar commands. You should also use the Tab key to make orderly columns in your program.

Once you have your program entered, you are ready to assemble it. This is accomplished by typing:

RUN ASM86 :F1:LAB1.ASM SYMBOLS

where

RUN is a program that invokes the 8086 processor in the development system (ISIS uses the 8085 processor).

ASM86 is the program that you want the 8086 processor to execute (the assembler).

:F1:LAB1.ASM is what you want the assembler to assemble.

SYMBOLS is a control telling the assembler that you would like a table of all the symbols used in your program. This symbol table will be attached to the program listing.
LAB 1

When the assembler is done, it will return control to ISIS. It will also create two new files on the floppy disk in drive 1. One of these files contains 8086 object code to be executed on an 8086 processor. The other file contains the program listing which gives useful information about the program including any errors the assembler found. Write the names of these two files:

: F1: ________________________
: F1: ________________________

If you cannot remember the names of these files, you can find them by looking at the directory of drive 1. Type:

DIR 1

Copy the listing file to the line printer by typing:

COPY : F1: ____________ TO :LP:

or substitute the printing device given by your instructor to use instead of :LP:.

If the assembler found any errors, now is the time to correct them by changing your source file using AEDIT.

You should be able to identify most of the items in the listing. Try to answer these questions.

How many bytes long is the program?

What is the offset of the last instruction in the program?

How many bytes long is this last instruction?

DON'T PROCEED TO THE NEXT SECTION UNTIL YOU HAVE ASSEMBLED YOUR PROGRAM WITH NO ERRORS!
LAB 1

LOADING AND RUNNING YOUR PROGRAM

As we saw in the last section, the assembler produced an object file called :F1:LAB1.OBJ. This file contains relocatable object code. It does not contain any absolute addresses. It must be assigned an address before it can be executed. To assign an address to a program, it is run through a "locater". The locater assigns absolute addresses to the segments in a file.

The SERIES III development system, however, is designed to accept run time locatable code. Thus the code is assigned an address as it is being loaded into RAM memory from the disk. This saves several steps (and time) during program debugging (eventually the program will need to be located before it can be used with an in-circuit emulator or burned into PROMs). To assign run time locatable addresses to your program, we use the linker with a BIND option. This option allows the program to be run on the SERIES III development system. Type:

RUN LINK86 :F1:LAB1.OBJ BIND

The LINK86 program produces two new files, :F1:LAB1 and :F1:LAB1.MP1.

The file :F1:LAB1.MP1 is a map of the output of the linker. You may want to copy it to the line printer, but for such a small program as this one it won't give you much useful information. :F1:LAB1 is the run time locatable object file.

To run your program type:

RUN :F1:LAB1.

The period after LAB1 is required. If you don't include it, the RUN program will look for a file called :F1:LAB1.86 and not find it. Most 8086 object code programs to be run on the SERIES III have an extension of .86. You may want to look at the directory of your system disk to verify this. By including the period after your file name, you tell the RUN program not to look for the .86 extension.

Verify that your program works correctly. If it does not, study your listing or ask your instructor for help. Tomorrow you will learn techniques for debugging your programs while they are running in the development system. Remember, you can abort your program execution at any time and return to ISIS by entering Ctrl-C (press and hold the Ctrl key and then type a C).
Note: If a HLT instruction is included in your program, you might get some unexpected results. This is due to the way that the HLT instruction works and the way that the SERIES III works. The main use of the HLT instruction is to wait for a hardware interrupt. After an interrupt, the processor continues execution with the instruction after the HLT instruction. The SERIES III normally interrupts the 8086 processor every 50 msec. When interrupted, the 8086 checks to see if any keys had been hit at the keyboard. These interrupts are invisible to you unless you use a HLT instruction to end your program. If you do end with a HLT instruction, the 8086 will execute whatever follows the HLT instruction as soon as it returns from the interrupt routine. The solution is to not use a HLT instruction for ending your program or to use a JMP instruction directly after the HLT which jumps to the HLT instruction.

PROBLEM (part 2)

Write a program that will rotate a pattern of one lit LED on the LED's of port 0. The program should delay about 1 second between each rotate.

PROBLEM (part 3)

Use the program written in part 2, but make the delay a variable that is specified by the switch setting on port 1. You may find it difficult to write a 'bug free' program using only the instructions covered so far in class. If you have problems, speak to the instructor or you may want to look at the solution given. Try your own solution first!!

REVIEW:

In this lab, you have learned how to use the instructions taught in Day 1 of the workshop and some of the ISIS commands discussed in class. You have learned how to create, assemble, link and execute your program using the SERIES III development system. The development steps taken in this lab were:

AEDIT :F1:LAB1.ASM
RUN ASM86 :F1:LAB1.ASM SYMBOLS
COPY :F1:LAB1.LST TO :LP:
RUN LINK86 :F1:LAB1.OBJ BIND
RUN :F1:LAB1.
When you finish this lab you will be able to:

* Define and access a data array
* Debug using DEBUG-86 symbolic debugger

PROBLEM (part 1)

Using the flow chart in the following text, write a program that will continuously search a 50 byte array called BUFFER for the ASCII code for return (0DH). If a return is found, the program should output F0H (for FOund) to port 0 LEDs and continue looking from the beginning of the buffer.

If a return is not found, the program should output 0FH to the LEDs and start looking again from the beginning of the buffer.

When writing your program, don't worry about putting a return in the buffer. We will do this later using the debugger. Use START: as the program label for the first instruction in your program.
When you have your program written, you will have to prepare it for execution as you did in Lab 1. Enter your program on a disk file using AEDIT and assemble it using ASM86. Don't forget to use the DEBUG and SYMBOLS options for the assembler as shown below.

RUN ASM86 :F1:LAB2.ASM SYMBOLS DEBUG

The DEBUG option attaches a copy of the symbol table to the object file. When you load your object file into RAM memory, DEBUG-86 will remember the symbol names and their values. This allows you to use symbolic names to reference parts of your program. You should get a copy of the listing for the DEBUG session that follows.

Remember, the SYMBOLS option attaches a copy of the symbol table to the program listing so that you can look at it.

Prepare your object file for loading with:

RUN LINK86 :F1:LAB2.OBJ BIND USING THE SERIES III DEBUGGER

At this point, you are ready to execute your program. However, instead of just running your program and hoping that it works correctly, your should use DEBUG-86 to analyze its operation and find any errors that you might have made.

To invoke the SERIES III Debugger, type:

RUN DEBUG

The debugger will sign on:

DEBUG 8086 V x.y
*

The asterisk prompt,"*", tells you that you are in the debugger and only DEBUG-86 commands are valid (you can still use Rubout). The DEBUG-86 commands are shown in the Intellec Series III Microcomputer Development System Pocket Reference Card with a full explanation in the Intellec Series III Microcomputer Development System Console Operating Instructions manual Chapter 6.

To load the program into memory type:

LOAD :F1:LAB2
LAB 2

This command will load both your program and all of the symbols that you declared in your program. The symbols will only get loaded if the DEBUG option was used when you assembled your program. The loader will also initialize the CS and IP registers to point to the first instruction in your program. Do not put a period at the end! DEBUG-86 only looks for the filename specified. Before executing the program, check to see where in memory the program was loaded. How can you tell where the program was loaded? (hint--look at the registers.) Type:

REGISTER

The debugger will display all the registers and flags.

Where is the program located?

To see if the program was loaded correctly, display memory. The memory display commands use an address range which can be specified in several ways. Type:

BYTE CS:0 TO CS:20

Compare this memory dump to the object code given in the listing. Do they match? An easier way to determine if the program was loaded correctly would be to disassemble the object code in memory. To do this, type:

ASM CS:0 TO CS:20

This command, like the BYTE (display memory) command, requires an address range. The LENGTH keyword can also be used in specifying address ranges. To try it, type:

ASM CS:IP LENGTH 25

Note: You may see an XCHG AX,AX when you dissemble your program. This is not an error. XCHG AX,AX is the way the assembler generates a NOP (no operation) instruction. It is possible for the assembler to allocate one extra byte for a JMP instruction if the destination of the jump is defined later in the program. This extra byte is filled with a NOP. More on this later.

Before running the program, you should know whether or not a return character is in the buffer. But where is the buffer? One way of finding out the address of the buffer is to look it up in the symbol table. Type:

SYMBOLS
LAB 2

You should see all the symbols in your program including segment names. However, we can also use symbol names directly. To display the buffer, try:

    BYT .BUFFER LENGTH 50T

You must use a period in front of every symbol name. This is to differentiate symbol names from DEBUG-86 commands in case they happen to be the same. The T in 50T indicates base ten. The default base is hex.

Fill the buffer with all zeroes by typing:

    BYT .BUFFER LEN 50T = 0

Now execute the program sing the GO command:

    GO

The GO command defaults to using the current CS:IP value as a starting address. If CS:IP were not correct, you could have typed:

    GO FROM .START

Is the program working correctly? To stop execution, press and hold the Ctrl key and type D (Ctrl-D). Ctrl-D brings you back into the debugger. The program stops executing and the next instruction to be executed is displayed. To place a return (\0DH) in the buffer and see if your program finds it, type:

    BYT .BUFFER+10T = \0DH

This will place a \0DH in the eleventh byte in the buffer. Display the buffer again to see if it is there. Now execute the program from the beginning to see if it works. If your program doesn't work, there are several commands to help you find out why.

    GO FROM .START TILL .FOUNDIT

Breakpoints can be used to stop execution at a certain place in your program. They are very useful for finding out if a program is executing correctly. If you had a program label called FOUNDIT and you wanted to see if your program ever reached this statement, you could type:

    GO FROM .START TILL .FOUNDIT

A-9
LAB 2

To single step the program, use the step command. To single step the first instruction, type:

STEP FROM .START

An address could have been used (STEP FROM 485:0). The debugger displays the next instruction to be executed. To step again type:

STEP

The ports on the I/O box can be directly controlled with the debugger. To read the value of the switches on port 0, type:

PORT 0

To turn on the LEDs on port 1, type:

PORT 1 = FF

The debugger has several advanced commands that are useful during debugging. One of these allows any number of DEBUG-86 commands to be repeated indefinitely. To use this command to repeatedly single step and display the registers after every instruction, type:

REPEAT
STEP
REGISTER
END

Abort with Ctrl-D. Use these commands until you feel comfortable with them. If you have extra time, you should try some of the other DEBUG-86 commands that were not discussed here.

To exit the debugger and return to ISIS, type:

EXIT

or

Ctrl-C.

PROBLEM: (optional)

Modify the previous lab to count the number of returns in the buffer. You should use a variable in memory to hold this count. After going through the entire buffer, output
the count to the LEDs on port 0. If the count is zero, output a value of FFH. Have this repeat continuously. Use DEBUG-86 to add returns to your buffer. The following steps may assist you in development:

1) INITIALIZE CX = LENGTH OF BUFFER, BX = 0, COUNT = 0
2) IF BUFFER[BX] = 0DH THEN COUNT = COUNT + 1
3) BX = BX + 1, CX = CX - 1
4) IF CX DOES NOT EQUAL ZERO GO TO STEP 2
5) IF COUNT = 0 THEN OUTPUT 0FFH OTHERWISE OUTPUT COUNT
6) GO TO STEP 1

REVIEW:
In this lab, you have learned how to use the instructions taught in Day 2 of the workshop and how to define and access data. You have learned how to debug your program using the SERIES III development system and DEBUG-86.

The DEBUG-86 commands used in this lab were:

RUN DEBUG Activates DEBUG-86.
LOAD Loads your program code into 8086 memory.
REGISTER Display the contents of user registers.
BYTE Display and change the contents of byte memory locations.
ASM Display the contents of memory locations in 8086 Assembly language mnemonics.
SYMBOLS Displays symbols and their values.
GO Causes execution of your program until breakpoint conditions are met.
STEP Causes execution of a single program instruction.
PORT Display and change contents of a byte I/O port.
REPEAT Causes looping of a command.
EXIT Exits DEBUG-86 (or use Ctrl-C).
When you finish this lab you will be able to:

* Use and declare procedures in ASM86
* Break up your code into separate segments
* Pass parameters to a procedure
* Create and initialize a stack
* Optionally, create an interrupt routine

PROBLEM (part 1)

In the first part of this lab, you will create a simple typewriter program that inputs characters from the development system keyboard and outputs them to the CRT. For this part of the lab, you will use two procedures provided on your system disk. These procedures are labelled CI and CO.

CI is a procedure that inputs one character from the keyboard and returns its ASCII value in the AL register. It will wait until a key has been hit.

CO is a procedure that outputs one character to the CRT. The character to be output (the parameter) should be passed on the stack. CO will clean up the stack.

CI and CO have already been written for you and the object code is contained in two files on your system disk called CI.OBJ and CO.OBJ. We have provided these to save you the time and effort of writing them on your own. CI and CO are actually written in PL/M-86, a high level language. The listings are given in the lab solutions section.

Write your program as if CI and CO were declared in your own source program. They will actually be added later when you use LINK86 to bind your program. This is modular programming which will be covered later in the course.

Use the following steps to help you write your program:

1) CREATE A STACK
2) INITIALIZE ANY NECESSARY REGISTERS
3) CALL CI
4) CALL CO (Don't forget to pass the character on the stack)
5) JUMP TO STEP 3
Because you are using the procedures CI and CO and you don't declare them anywhere in your program, the assembler will give you an error. To prevent this, you should tell the assembler that the procedures CI and CO are defined "external" to the module. To do this, place the following statement at the very beginning of your program (it must be outside of any segment).

EXTRN CO:FAR,CI:FAR

When you are ready to link your program, use the command:

RUN LINK86 :F1:LAB3.OBJ,CO.OBJ,CI.OBJ,LARGE.LIB BIND

This will include the CI and CO routines. LARGE.LIB is a collection of programs that enables an 8086 program to access I/O devices on the development system.

---Good luck---

PROBLEM (part 2)

In this part of the lab, you should make two additions to the program written for part 1. The first is to write a new procedure called ENCRYPT. Before outputting any character to the CRT, it should first be passed to the ENCRYPT procedure. ENCRYPT should transform the ASCII character in some way that you decide and pass it back to the main program. An easy example would be to add a one to the value. This would transform an "A" into a "B", "B" into a "C", etc. An ASCII table is included in the front of this lab section to help you. Pass this parameter on the stack to ENCRYPT. Place ENCRYPT in the same segment as the main program.

Where would be the best place to put the ENCRYPT procedure in your code segment? (the beginning or the end)

What would you use to access the parameter passed to ENCRYPT on the stack?

Also, you probably noticed that carriage returns did not produce a line feed. Add some code to your main program to detect carriage returns and to output a carriage return and a line feed when a carriage return is entered.
PROBLEM (part 3)

Place ENCRYPT in a separate segment from the main program. Your program should then contain two segments with one of them containing your main code and the other containing only the ENCRYPT procedure.

Where would be the best place to put the ENCRYPT procedure segment in your program? (the beginning or the end)

What changes had to be made to make this work? (procedure type and parameter access changes)

PROBLEM (part 4)

This is a slightly more difficult version of part 2.

Instead of creating an ENCRYPT procedure, write one that implements a shift-lock feature for the keyboard. The TPWR key already does this, but we will implement the feature in software. When the TPWR key is depressed, the Intellec keyboard produces both upper and lower case characters depending on the shift key. You should write a procedure that converts lower case alpha characters to upper case characters depending on whether the shift-lock has been set. The shift-lock is defined as the character "I" (7CH) in the upper right hand corner of the keyboard. After this key is hit for the first time, all alpha characters output should be in upper case only. After it is hit again, alpha characters should be in both upper and lower case. Your procedure should maintain a software flag to keep track of whether the lock is set or not.
LAB 3

OPTIONAL PROBLEM (Interrupts)

You are to implement an interrupt service routine. Your main program will be required to read the values set on the port switches then divide the number set on port 0 by that set on port 1. The result (port 0/port 1) should be displayed on the port 0 LEDs. This should be done in a continuous loop.

A divide error may occur. For example, if the port 1 switches were 0 then the answer of infinity cannot be represented. You will have to write an interrupt service routine for the type 0 interrupt to handle this. This routine should change the state of the port 1 LEDs, delay for a half a second and then return. While there is a divide error being generated in the main program, the LEDs on port 1 will flash, the first interrupt switching them on, the next switching them off, etc. Use a byte in RAM to flag the LEDs on/off.

Remember to do the following:

1) Your main program should set up the stack.
2) Your main program should set up the pointer to the interrupt service routine.
3) The interrupt service routine should save any registers it uses.
4) Use the correct return at the end of the routine.

If you prefer to use an absolute segment with a pointer to your interrupt routine in that segment, you may encounter some problems with DEBUG-86. DEBUG overwrites your pointer table entry when it loads your program. If you wish to reload it, type POINTER 0 = .(error) where "error" is whatever you called your service routine.

Do you need to enable interrupts with an STI instruction?

Why not?

REVIEW:

In this lab, you have learned how to create procedures, placed them in a separate segment from your main program, and passed parameters to your procedure. You have created and initialized the registers to point to your stack. If you did the optional lab, then you have set up interrupt pointers and written an interrupt service routine.
LAB 4

When you finish this lab you will be able to:

* Break up your program into separate modules
* Use a jump table
* Encrypt using the XLAT instruction

PROBLEM (part 1)

In this lab, you are going to write a procedure that will be referenced in another module. Edit the program you developed in part 3. Remove the segment that contained the ENCRYPT procedure and make an external reference to the procedure. Now write a separate module that will only contain the ENCRYPT procedure. Modify this procedure to provide a switch selective encryption technique. The operation of the procedure should be as follows:

The procedure should read the value set on the port 0 switches and use this as an index into a table of offsets of program labels. Using an indirect jump, the procedure will jump to one of several different program labels. Each of these pieces of code will provide a different encryption technique to alter the character that was sent to the ENCRYPT procedure. If the value on the switches is greater than the number of encryption techniques you have provided, the ENCRYPT procedure should return a "*" (2AH) to indicate a nonvalid switch setting.

This purpose of this lab is to implement a jump table and to use multiple modules, not to think of many ways of altering the characters. Two or three simple encryption techniques will suffice (i.e. increment character, decrement character, and shift character). Remember to link these together.

PROBLEM (part 2)

Write another encrypt procedure in a separate module. This time try writing it using the XLAT instruction for encrypting your characters. This is a natural for this instruction. Link this module to your main program instead of the one you created in part 1.

REVIEW:

In this lab, you have used multiple modules and the conventions for linking them together. You have also used the instructions taught in Day 4 of the workshop.
AEDIT Basics Lab

When you finish this lab you will be able to:

* Invoke the editor
* Insert text to make a file
* Position the cursor to make corrections
* Correct mistakes by deleting and exchanging characters
* Move and copy blocks of text
* Exit the editor and save your file

In this lab, you will be learning the basic AEDIT commands so you can create your program files. If you have any problems or errors occur, please see your instructor. You will be editing a file called TEST.LAB. This file is on your system disk. Power up your system following the steps taught in class. To use this file, copy it to your user disk with the following command: (<CR> indicates the return key)

COPY TEST.LAB TO :F1: <CR>

To edit this file, you invoke AEDIT by typing the following line:

AEDIT :F1:TEST.LAB <CR>

AEDIT displays a menu on the bottom of the screen which should look like this:

---- system id AEDIT V x.y
Again Block Delete Execute Find -find Get -- more --

At the end of the text you should see a vertical bar "!" which is the EOF mark. This marks the end of the text file. If this was a new file it would appear at the top of the screen. As you type in text it will move and continue to mark the end of the file.

The solid non-blinking block is the cursor. This marks where you are at in the file.

When you begin a session, AEDIT is in the command mode. The menu at the bottom of the screen shows you what options you have. Press the Tab key (If the terminal you are using does not have a Tab key, press and hold the Ctrl key and then type the I key). Pressing the Tab key will show the other options available in the command mode. Pressing Tab repeatedly will show all the options and wrap around to the beginning of the menu. Several of the commands also have subcommand menus as you will see later.
The Insert command is used to type in new text in front of the current cursor position. To enter any command, you type the first letter of the command. Press the I key. You should see "[insert]" at the bottom of the screen to indicate that you are now in the insert mode. Now type in a word but misspell it. To correct your error, press the RUBOUT key. Each time you press the RUBOUT key, it backs the cursor one column and erases that character. Once the offending character is erased, simply type in the new characters.

Delete the characters you just typed by holding down the Ctrl key and typing the X key. This is the DELETE LEFT command and deletes the text on a line from the cursor to the beginning of the line. At this point, the text should be the same as shown below.

When you type ussing an edior you may often make a mistake that you have to correct. AEDIT will allow you to correct the problem, get rid of bad stuff, and make your life easy. This is the first line.

\;

The arrow keys move the cursor up, down, right, or left. If you type the HOME key after one of the arrow keys, then you can move rapidly to the beginning or end of a line or page forward and backwards through a file. Press the right arrow key followed by the HOME key. Notice the cursor moved to the end of the line. Press the left arrow key followed by the HOME key. This took the cursor to the beginning of the line.

The fourth word in the first line, "ussing", is misspelled. Press the right arrow key to move the cursor to the first "s" in "ussing". To delete the "s", hold down the Ctrl key and type an F. This is the DELETE CHAR command which deletes the character under the cursor.

The sixth word in the first line, "edior", is missing a "t". Move the cursor to the "o" in "edior". Now type a "t". While in the insert mode, you can insert characters anywhere in your text.

Press the Esc key. This takes you out of the insert mode and back to the command mode. Another method to go back to the command level is to use a Control C. Control C aborts the command and all corrections made are lost.

A-l8
The third word on the second line "mistoke" is spelled wrong. Move the cursor to the "o" in "mistoke". Since we wish to change the character "o" for an "a", press X for Xchange mode. Xchange allows you to overtype characters. If you make a mistake, press the RUBOUT key, and the old character is returned as long as you don't press Esc, return, or a cursor movement key. Press an "a" to correct "mistoke", and then press the Esc key to get back to the command mode.

The third line contains "the the" at the end of the line. Since the second "the" is at the end of the line, you can delete from there to the end of the line. To get rid of the second "the", move the cursor to the space in front it. Press and hold the Ctrl key and type an A. This command, DELETE RIGHT, deletes all characters to the right of the cursor to the end of the line.

Control A (DELETE RIGHT), Control X (DELETE LEFT) and Control Z (DELETE LINE) can also be restored. The command to do this is the Undo command which is Ctrl U. Undo is able to restore up to 100 characters deleted by the last Control A, X, or Z at the current cursor position. Press Ctrl and type a U. Notice the "the" you just deleted has reappeared. Now delete it again.

Now you will be deleting characters in the middle of a line. If you wished to delete ", get rid of bad stuff,", you would first block or delimit this section. Move the cursor to the comma in front of "get" and type a B for Block. Notice when you did this an "@" has taken the place of the cursor. Now move the cursor to one character past the last character you want in the block. In this case, you would move it to the space after "stuff, " Notice an "@" moved with your cursor and marks the end of the block.

When you pressed B for Block, you may have noticed that the menu has changed to show Block's subcommands. Since you wish to delete, type a D for Delete. Notice that everything from under the first "@" up to the last "@" was deleted.

The Block command gives you the ability to move and copy text from one part of your file to another. The fifth line which reads "This is the first line." should be moved to the first line. Move the cursor to the first character of the fifth line and type a B for Block. Now type the down arrow key. This will block the line. To move the line, you would first delete it, move the cursor to where you want to move it, and then get the line back. Type a D for the block subcommand Delete. This has deleted the line and
AEDIT Basics Lab

placed it in a buffer. Now move the cursor to the beginning of the text by typing an up arrow and then HOME.

Now you want to get the text you deleted. Type a G for the Get command. The Get command will prompt:

Input file:

on the bottom of the screen. To get the buffer which holds the deleted line, type a return or the Esc key. Notice the line has been retrieved and has been inserted before the old cursor position.

Now let's copy the entire text file. Move the cursor to the beginning of the file if your cursor isn't already there. Now type a B for Block. Move the cursor to the EOF mark by typing a down arrow followed by HOME. Since you are about to copy, type a B for Buffer. This will place the blocked text in the buffer without deleting it. Now get the contents of the block buffer by typing G for the Get command. Answer Get's prompt with a return to get the buffer. Notice the six lines are repeated on the screen. Type G again and answer Get's prompt with a return. Notice the same six lines are repeated. Once text is in the buffer you can get it several times. Get the buffer three more times.

To look at the text that is scrolled off the screen, type a down arrow several times. Notice that when you are at the bottom of the screen the screen scrolls up one line every time you type a down arrow. A faster way to look at the next page is to use the HOME key. Type the HOME key. Since the last arrow key typed was the Down arrow key, this should have taken you to the next page or screenfull of text. Typing HOME again should take you to the next page of text or the EOF marker, if this was the last page of text. To look at the previous page of text, you could type the Up arrow key several times or type the Up arrow key followed by the HOME key. Type the HOME key again. Repeated typing of HOME will take the cursor to the beginning of the text. Go from the beginning to the end of the text several times to get comfortable with the operation.

Now that you are finished editing this file, you are ready to end the editing session. Type Q for the Quit command. The bottom of the screen should look like this:

---- Editing :F1:TEST.LAB
Abort Exit Init Update Write

Notice that Quit has several subcommands that you can choose from. Abort returns to the operating system with
all changes lost. If any changes were made, it will ask
you "all changes lost (y or [n])" to make sure. Exit will
write out the new file and return to the operating system.
Init allows you to edit another file without leaving AEDIT.
Update updates your file without leaving AEDIT. Write
prompts for an output file name and then it writes your
file to the named file without leaving AEDIT. Any legal
filename can be used even :LP:. If you did not specify a
filename at the beginning of the session, only Abort, Init,
and Write are available. Since you want to save the file
and leave AEDIT, type E for Exit. Now your file has been
written to the disk and you should have the operating
system prompt. See if your file has been written by typing
DIR 1<CR>.

You should have two files TEST.LAB and TEST.BAK. When you
edit an old file and exit, AEDIT first changes the name of
your old file, TEST.LAB, to TEST.BAK before saving the
changed file. This way you still have the old file in case
the new one didn't work. To use AEDIT on the old file, use
the ISIS RENAME command. For example:

RENAME :F1:TEST.BAK TO :F1:TEST1.LAB

The AEDIT commands can be found in the AEDIT Text Editor
Pocket Reference and in the AEDIT Text Editor User's Guide.
AEDIT has several other advanced commands that you may wish
to use. Refer to these guides to look at these commands.
The commands you have seen in this lab session are the most
frequent ones that you will use to do most of your editing.
AEDIT Basics Lab

Review:

The AEDIT commands that we have learned are:

Cursor Movement commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrow keys</td>
<td>Moves cursor right, left, up, or down.</td>
</tr>
<tr>
<td>Right arrow-HOME</td>
<td>Move cursor to end of line.</td>
</tr>
<tr>
<td>Left arrow-HOME</td>
<td>Move cursor to the beginning of the line.</td>
</tr>
<tr>
<td>Down arrow-HOME</td>
<td>Move cursor to the next page.</td>
</tr>
<tr>
<td>Up arrow-HOME</td>
<td>Move cursor to previous page.</td>
</tr>
</tbody>
</table>

Delete commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-X</td>
<td>Delete all characters left of the cursor to the beginning of the line.</td>
</tr>
<tr>
<td>Ctrl-A</td>
<td>Delete all characters right of the cursor to the end of the line.</td>
</tr>
<tr>
<td>Ctrl-Z</td>
<td>Delete line.</td>
</tr>
<tr>
<td>Ctrl-U</td>
<td>Undo a Ctrl-A, X, or Z.</td>
</tr>
<tr>
<td>Ctrl-F</td>
<td>Delete character under cursor.</td>
</tr>
<tr>
<td>RUBOUT</td>
<td>Delete the preceding character.</td>
</tr>
</tbody>
</table>

Menu commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>Insert text before cursor.</td>
</tr>
<tr>
<td>Xchange</td>
<td>Type over characters under cursor.</td>
</tr>
<tr>
<td>Block</td>
<td>Allows you to delimit a block of characters with the following subcommands:</td>
</tr>
<tr>
<td></td>
<td>Buffer: Store delimited block in buffer.</td>
</tr>
<tr>
<td></td>
<td>Delete: Delete delimited block and store it in the buffer.</td>
</tr>
<tr>
<td>Get</td>
<td>If responded to with a return, gets the contents of the block buffer.</td>
</tr>
<tr>
<td>Quit</td>
<td>Ends the editing session with the following subcommands:</td>
</tr>
<tr>
<td></td>
<td>Abort: Quit with all changes lost.</td>
</tr>
<tr>
<td></td>
<td>Exit: Write new file to disk and quit.</td>
</tr>
<tr>
<td></td>
<td>Init: Edit a new file without returning to the operating system.</td>
</tr>
<tr>
<td></td>
<td>Update: Update your file without returning to the operating system.</td>
</tr>
<tr>
<td></td>
<td>Write: Writes contents of file to the named file without returning to the operating system.</td>
</tr>
<tr>
<td>Esc</td>
<td>Takes you back to the command mode.</td>
</tr>
<tr>
<td>Ctrl-C</td>
<td>Aborts the command and returns you to the command mode.</td>
</tr>
</tbody>
</table>
SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LABIA
OBJECT MODULE PLACED IN :F2:LABIA.OBJ
ASSEMBLER INVOKED BY: :F3:ASMAC86.86 :F2:LABIA.ASM

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td>1</td>
<td>NAME   LABIA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td>3</td>
<td>LEDS   EQU 0 ;LED PORT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>SWITCH EQU 1 ;SWITCH PORT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>CODE   SEGMENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>ASSUME CS:CODE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>E401</td>
<td>9</td>
<td>START: IN AL,SWITCH</td>
</tr>
<tr>
<td>0002</td>
<td>E600</td>
<td>10</td>
<td>OUT LEDs,AL</td>
</tr>
<tr>
<td>0004</td>
<td>EBFA</td>
<td>11</td>
<td>JMP START</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>CODE ENDS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>END START</td>
</tr>
</tbody>
</table>

ASSEMBLY COMPLETE, NO ERRORS FOUND
MACRO ASSEMBLER LAS1_PART2

SERIES-III 8086/8087/8088/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB1_PART2
OBJECT MODULE PLACED IN :F2:LAB1B.OBJ
ASSEMBLER INVOKED BY: :F3:ASM86.86 :F2:LAB1B.ASM SYMBOLES DEBUG

LOC OBJ  LINE  SOURCE

1  NAME  LAB1_PART2

2

3  LEDS EQU 0

4  SWITCH EQU 1

5  PATTERN EQU 01H ; LED PATTERN

6

7  CODE  SEGMENT

8  ASSUME CS:CODE

0000 B001  9  START:  MOV  AL, PATTERN

0002 E600 10  AGAIN:  OUT  LEDS, AL  ; OUTPUT PATTERN

11

0004 B90500 12  MOV  CX, 5  ; 5 TIMES FOR 1 SEC

0007 BBD1 13  OUTER:  MOV  DX, CX  ; SAVE IT FOR LATER

0009 BFFFF 14  MOV  CX, 0FFFFH  ; 2 SEC DELAY

000C E2FE 15  INNER:  LOOP  INNER

000E EBCA 16  MOV  CX, DX  ; SET IT BACK

0010 E2F5 17  LOOP  OUTER  ; TO DO IT 5 TIMES

18

0012 D0C8 19  ROR  AL, 1  ; ROTATE PATTERN

0014 E8EC 20  JMP  AGAIN  ; REPEAT

----

21  CODE  ENDS

22  END  START
LEDS EQU 0
SWITCH EQU 1
PATTERN EQU 01H

ASSUME CS:CODE

START:  MOV AL,PATTERN
          OUT LEDs,AL ;OUTPUT PATTERN
AGAIN:   MOV BL,AL ;SAVE PATTERN
          MOV AH,0 ;SWITCHES
          MOV CX,AX
          JCAZ CONTIN ;IF CX IS ZERO, THEN
          ;SKIP DELAY. OTHERWISE
          ;DELAY WOULD BE TOO LONG
          MOV AX,0FFFFH ;2 SEC DELAY
          MOV CX,0FFFFH
          LOOP OUTER ;TO DO IT 5 TIMES
          OUTER:  MOV DX,CX ;SAVE IT FOR LATER
          LOOP INNER
          MOV CX,DX ;GET IT BACK
          INNER:  LOOP INNER
          CONTIN:  MOV AL,BL ;PUT PATTERN BACK
          ROR AL,1 ;ROTATE PATTERN
          CONTIN:  MOV AL,BL ;PUT PATTERN BACK
          ROR AL,1 ;ROTATE PATTERN
          CONTIN:  JMP AGAIN ;REPEAT
          CODE ENDS
          END START
LOC OBJ LINE SOURCE

1 ;THIS PROGRAM IMPLEMENTS THE FLOWCHART GIVEN IN LAB 2

2

3 NAME LAB2

4

5 CR EQU 0DH ;;CARRIAGE RETURN

00F0 6 FOUND EQU 0FH ;;LED PATTERN IF CR IS FOUND

00F0 7 NFOUND EQU 0FH ;;LED PATTERN IF CR IS NOT FOUND

0000 8 LED EQU 0 ;;LED PORT

9

0000 (50) 10 DATA SEGMENT

??

11 BUFFER DB 50 DUP (?)

)

12 DATA ENDS

13

14 CODE SEGMENT

15 ASSUME CS:CODE,DS:DATA

0000 B8----- 16 START: MOV AX,DATA ;;INITIALIZE DS SEGMENT

0003 B8DB 17 MOV DS,AX

0005 B93200 18 AGAIN: MOV CX,LENGTH BUFFER ;;LOAD CX FOR LOOP COUNT

0008 33DB 19 XOR BX,BX ;;INITIALIZE INDEX

000A 803F0D 20 CHECK: CMP BUFFER[BX],CR ;;CHECK CONTENTS OF BUFFER FOR 0DH

000D 7409 21 JE FNDIT ;;JMP IF CR WAS FOUND

000F 43 22 INC BX ;;BUMP INDEX

0010 E2FB 23 LOOP CHECK ;;DO IT AGAIN

24

25 ;IF THE CPU FALLS OUT OF THE LOOP TO THIS LOCATION THEN

26 ;A CR WAS NOT FOUND

0012 BE0F 27 NFD: MOV AL,NFOUND ;;SIGNAL OPERATOR THAT CR

0014 E600 28 OUT LED,AL ;;WAS NOT FOUND

0016 EBED 29 JMP AGAIN

30

31 ;IF THE CPU JUMPS HERE THEN A CR WAS FOUND

0018 B0F0 32 FNDIT: MOV AL,FOUND ;;SIGNAL OPERATOR THAT CR

001A E600 33 OUT LED,AL ;;WAS FOUND

001C EB7 34 JMP AGAIN

35 CODE ENDS

36 END START
LOC OBJ | LINE | SOURCE
---|---|---
| | | 
| | | 
| | | 
---|---|---
0000 | B8 | R
0003 | 8ED8 | 15 START: MOV AX, DATA
0005 | B93200 | 16 MOV DS, AX; INITIALIZE DS
0008 | 33DB | 17 AGAIN: MOV CX, LENGTH BUFFER; SET CX WITH LOOP COUNT
000A | C606000000 | 18 XOR BX, BX; INITIALIZE INDEX
000C | 807F010D | 19 MOV COUNT, 0; INITIALIZE COUNT
000E | 00F010 | 20 CHECK: CMP BUFFER[BX], CR; LOOK FOR CR
0013 | 7504 | 21 JNE NFIND; IF NO CR THEN DON'T COUNT IT
0015 | FE000000 | 22 INC COUNT; ELSE COUNT IT
0019 | 43 | 23 NFIND: INC BX; BUMP INDEX
001A | E2F3 | 24 LOOP CHECK
001C | 803E000000 | 25 CMP COUNT, 0; IF COUNT IS ZERO
0021 | 7407 | 26 JE NONFIND; THEN PUT OUT NONE FOUND CODE
0023 | 800000 | 27 MOV AL, COUNT; ELSE PUT OUT NUMBER OF CR
0026 | E600 | 28 OUT AL, LEDS; AL
0028 | EB0B | 29 JMP AGAIN
002A | D0FF | 30 NONFIND: MOV AL, NO_CR; THIS IS WHERE WE PUT OUT NONE FOUND CODE
002C | E600 | 31 OUT LEDS, AL
002E | EB05 | 32 JMP AGAIN
0030 | 38 | 33 CODE ENDS
---|---|---
0038 | CODE ENDS | 34 END START
LOC OBJ LINE SOURCE

1 ; THIS PROGRAM WILL USE TWO EXTERNAL PROCEDURES TO ECHO CHARACTERS
2 ; FROM THE KEYBOARD AND THE CRT OF THE SERIES III. CI IS ONE
3 ; OF THESE PROCEDURES. CI INPUTS 1 CHARACTER FROM THE KEYBOARD AND
4 ; RETURNS IT IN THE AL REGISTER TO THE CALLING ROUTINE. CO
5 ; IS THE OTHER PROCEDURE. CO OUTPUTS A CHARACTER TO THE CRT. CO
6 ; EXPECTS THE CHARACTER ON THE STACK. THEREFORE, THE CALLING ROUTINE
7 ; MUST PUSH THE CHARACTER ONTO THE STACK BEFORE CALLING CO.
8
9 ; THESE ARE THE EXTERNALS FOR CI AND CO
10 EXTRN CI:FAR,CO:FAR
11
12 --- NAME LAB3_PART_1
13
14 STACK SEGMENT
15""
16 DW 100 DUP(?)
17
18 ---
19 CODE SEGMENT
20 ASSUME CS:CODE, SS:STACK
21
22 START: MOV AX, STACK ; INITIALIZE THE
23 MOV SS, AX ; STACK SEGMENT AND
24 LEA SP, TOP ; STACK POINTER REGISTERS.
25
26 AGAIN: CALL CI ; GET CHARACTER FROM THE KEYBOARD
27 PUSH AX ; PLACE CHARACTER ON THE STACK
28 CALL CO ; OUTPUT IT TO THE CRT
29 JMP AGAIN
30
31 ---
32 CODE ENDS
33
34 END START
SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB3_PART_2
OBJECT MODULE PLACED IN :F2:LAB3B.OBJ
ASSEMBLER INVOKED BY: :F3:ASM86.86 ; F2:LAB3B.ASM SYMBOLS DEBUG

LOC OBJ LINE SOURCE

1 ; THIS PROGRAM IS THE SOLUTION TO LAB3 PART 2 OF THE WORKSHOP.
2 ; IT INPUTS CHARACTERS FROM THE KEYBOARD, ENCRYPTS THEM (ADD
3 ; ONE TO THE ASCII VALUE) AND THEN OUTPUTS THE RESULT TO THE
4 ; CRT. THE PROGRAM ALSO DETECTS WHEN A CR IS INPUT, AND INSERT A LF.
5
6 EXTRN CO:FAR,CI:FAR
7
8 NAME LAB3_PART_2
9
0000 CR EQU 0DH
0001 LF EQU 0AH
0002
0004 (100 D M 100 DUP(?)
0005 )
0008 T_O_S LABEL WORD
0009 STACK ENDS
000B CODE SEGMENT
000C ASSUME CS:CODE,SS:STACK
000D
000E ENCRYPT PROC
0010 ; THIS IS A SIMPLE ENCRYPTOR PROCEDURE. ENCRYPT EXPECTS
0011 ; TO RECEIVE AN ASCII CHARACTER AS A PARAMETER ON THE STACK.
0012 ; IT INCREMENTS THE ASCII VALUE BY ONE AND RETURNS THE
0013 ; ENCRYPTED CHARACTER IN THE AL REGISTER.
0014
001C 55 27 PUSH BP ; SAVE BP
001D 8BEC 28 MOV BP,SP ; USE AS REFERENCE IN STACK
001E 8B4604 29 MOV AX,[BP+4] ; GET CHARACTER
0020 FED0 30 INC AL ; INCREMENT IT AND LEAVE IT
0021 5D 31 POP BP ; IN AL
0022 C20200 32 RET 2 ; DELETES PARAMETER FROM STACK
0023 E8DEFF 33 ENCRYPT ENDP
0024
0025 8B80---- R 35 START: MOV AX,STACK ; INITIALIZE STACK
0026 8E00 36 MOV SS,AX
0027 826C800 37 LEA SP,T_O_S
0028 900000---- E 38 AGAIN: CALL CI ; GET CHARACTER FROM KEYBOARD
0029 3C0D 39 CMP AL,CR ; IS IT CARRIAGE RETURN?
002A 740C 40 JE CR LF ; IF YES THEN OUTPUT CR/LF
002B 50 41 PUSH AX ; PASS CHAR. ON STACK
002C EBDEFF 42 CALL ENCRYPT ; TRANSFORM IT
002D 50 43 PUSH AX
002E 900000---- E 44 CALL CO ; OUTPUT CHAR ON SCREEN
002F EBEB 45 JMP AGAIN
0030
0032 ; WE SHOULD ONLY BE EXECUTING CR LF IF A CARRIAGE RETURN WAS INPUT
0033 ; CR LF OUTPUTS A CARRIAGE RETURN AND LINE FEED

B-7
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>002A</td>
<td>B000</td>
<td>49</td>
<td>CRLF:  MOV  AL, CR</td>
</tr>
<tr>
<td>002C</td>
<td>50</td>
<td>50</td>
<td>PUSH AX</td>
</tr>
<tr>
<td>002D</td>
<td>9A0000</td>
<td>51</td>
<td>CALL   CO</td>
</tr>
<tr>
<td>0032</td>
<td>800A</td>
<td>52</td>
<td>MOV    AL, LF</td>
</tr>
<tr>
<td>0034</td>
<td>50</td>
<td>53</td>
<td>PUSH AX</td>
</tr>
<tr>
<td>0035</td>
<td>9A0000</td>
<td>54</td>
<td>CALL   CO</td>
</tr>
<tr>
<td>003A</td>
<td>EBD9</td>
<td>55</td>
<td>JMP    AGAIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>56</td>
<td>CODE   ENDS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>57</td>
<td>ENDS   END</td>
</tr>
</tbody>
</table>

; OUTPUT A CARRIAGE RETURN

; OUTPUT A LINE FEED

; GO BACK TO GET NEXT CHAR.
THIS PROGRAM IS THE SOLUTION TO LAB 3 PART 3 OF THE WORKSHOP.
IT DOES THE SAME AS PART 2 EXCEPT THE PROCEDURE IS IN ANOTHER SEGMENT

EXTERN CD:FAR, CI:FAR

NAME LAB3_PART_3

STACK SEGMENT

DW 100 DUP(?)

T_O_S LABEL WORD

ASSUME CS:CODE, SS:STACK

STACK ENDS

ASSUME CS:CODE, SS:STACK

START: MOV AX, STACK ;INITIALIZE STACK

AGAIN: CALL CI ;SET CHARACTER FROM KEYBOARD

IS IS CARRIAGE RETURN?

PASS CHAR. ON STACK

TRANSFORM IT

OUTPUT CHAR ON SCREEN

AGAIN
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0020</td>
<td>000D</td>
<td>049</td>
<td>WE SHOULD ONLY BE EXECUTING CRLF IF A CARRIAGE RETURN WAS INPUT</td>
</tr>
<tr>
<td>0020</td>
<td>000D</td>
<td>050</td>
<td>CRLF OUTPUTS A CARRIAGE RETURN AND LINE FEED</td>
</tr>
<tr>
<td>0022</td>
<td>50</td>
<td>051</td>
<td>CRLF: MOV AL, CR</td>
</tr>
<tr>
<td>0023</td>
<td>9A0000----</td>
<td>E</td>
<td>052</td>
</tr>
<tr>
<td>0028</td>
<td>B00A</td>
<td>053</td>
<td>CALL CO ; OUTPUT A CARRIAGE RETURN</td>
</tr>
<tr>
<td>0029</td>
<td>50</td>
<td>054</td>
<td>MOV AL, LF</td>
</tr>
<tr>
<td>0029</td>
<td>9A0000----</td>
<td>E</td>
<td>055</td>
</tr>
<tr>
<td>0029</td>
<td>B00A</td>
<td>056</td>
<td>CALL CO ; OUTPUT A LINE FEED</td>
</tr>
<tr>
<td>0030</td>
<td>EBD7</td>
<td>057</td>
<td>JMP AGAIN ; GO BACK TO GET NEXT CHAR.</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>CODE ENDS</td>
</tr>
<tr>
<td>058</td>
<td>END START</td>
<td>059</td>
<td></td>
</tr>
<tr>
<td>059</td>
<td>END START</td>
<td>059</td>
<td></td>
</tr>
</tbody>
</table>

B-10
LOC  OBJ  LINE  SOURCE

0000 (100)
????

000D  12  CR  EQU  0DH
000A  13  LF  EQU  0AH
0007  14  LOCK_KEY  EQU  7CH  ;SHIFT LOCK KEY (ASCII)
0000  15  NULL  EQU  00H  ;NULL ASCII CHARACTER

17  STACK  SEGMENT

0000  18  DW  100  DUP(?)

------

0028  19  T_D_S  LABEL  WORD

20  STACK  ENDS

------

0030  22  CODE  SEGMENT

23  ASSUME  CS:CODE,SS:STACK

0000  24  SHFTFLG  DB  0  ;MEMORY LOCATION WHICH INDICATES

25  ;IF SHIFT LOCK IS CURRENTLY SET

0001  26  SHIFT  PROC

27  ;SHIFT IS A PROCEDURE THAT WILL CHANGE LOWER CAS ALPHA

28  ;CHARACTERS TO UPPER CASE DEPENDENT ON WHETHER A SHIFT LOCK

29  ;HAS BEEN SET OR NOT.  SHIFT IS ALSO RESPONSIBLE FOR DETECTING

30  ;THE SHIFT LOCK KEY (ASCII 7CH, UPPER CASE BACK SLASH) AND

31  ;TOGGING A MEMORY BASED FLAG WHICH INDICATES IF THE SHIFT IS

32  ;CURRENTLY LOCKED OR NOT.  NOTE:  THIS LOCK ONLY AFFECTS ALPHA

33  ;CHARACTERS AND IS NOT THE SAME AS LOCKS FOUND ON A COMMON

34  ;TYPEWRITER.  SHIFT EXPECTS AN ASCII CHARACTER TO BE PASSED

35  ;ON THE STACK, AND WILL RETURN A CHARACTER IN THE AL REGISTER.

36

0001  55  37  PUSH  BP

0002  8BEC  38  MOV  BP,SP  ;USE BP TO REFERENCE STACK

0004  894604  39  MOV  AX,[BP+4]  ;GET INPUT CHARACTER

0007  3C7C  40  CMP  AL,LOCK_KEY  ;LOOK FOR SHIFT LOCK

0009  750B  41  JNE  TST  ;IF HIT, THEN

000B  2E8836000000  42  XOR  SHFTFLG,80H  ;TOGGLE SHIFT FLAG

0011  8000  43  MOV  AL,NULL  ;AND DON'T OUTPUT ANYTHING

0013  E81390  44  JMP  DONE

0015  2EF606000000  45  TST  TEST  SHFTFLG,80H  ;LOOK AT SHIFT FLAG STATUS

001C  7406  46  JZ  DONE  ;IF CLEAR, RETURN THE UNALTERED CHAR.

001E  3C68  47  CMP  AL,60H  ;IF SET, LOCK

0020  7206  48  JB  DONE  ;FOR LOWER CASE

;THIS PROGRAM IS THE SOLUTION TO LAB3 PART 4 OF THE WORKSHOP.
;IT INPUTS CHARACTERS FROM THE KEYBOARD, AND OUTPUTS THEM TO
;THE CRT. IT ALSO IMPLEMENTS A SHIFT LOCK FEATURE. BY TYPING
;AN UPPER CASE BACK SLASH "\" ALL SUBSEQUENT LOWER CASE ALPHA CHARACTERS
;WILL BE CONVERTED TO UPPER CASE. TYPING THE UPPER CASE BACK SLASH
AGAIN RETURNS THE OUTPUT TO UPPER AND LOWER CASE AGAIN.

EXTRN  CO:Far,CI:Far

NAME  LAB3_PART_3

;SHIFT IS A PROCEDURE THAT WILL CHANGE LOWER CAS ALPHA
;CHARACTERS TO UPPER CASE DEPENDENT ON WHETHER A SHIFT LOCK
;HAS BEEN SET OR NOT.  SHIFT IS ALSO RESPONSIBLE FOR DETECTING
;THE SHIFT LOCK KEY (ASCII 7CH, UPPER CASE BACK SLASH) AND
;TOGGING A MEMORY BASED FLAG WHICH INDICATES IF THE SHIFT IS
;CURRENTLY LOCKED OR NOT.  NOTE:  THIS LOCK ONLY AFFECTS ALPHA
;CHARACTERS AND IS NOT THE SAME AS LOCKS FOUND ON A COMMON
;TYPEWRITER.  SHIFT EXPECTS AN ASCII CHARACTER TO BE PASSED
;ON THE STACK, AND WILL RETURN A CHARACTER IN THE AL REGISTER.
LOC OBJ   LINE  SOURCE

0022 3C7A  49    CMP AL,7AH  ;ALPHA CHARACTERS
0024 7702  50    JA DONE  ;IF FOUND, THEN
0026 2C20  51    SUB AL,20H  ;MAKE INTO UPPER CASE.
0028 5D    52    DONE: POP BP
0029 C20200 53    RET 2
002B 54    54    SHIFT ENDP

002C B8----  R  56    START: MOV AX,STACK  ;INITIALIZE STACK
002F 8ED0  57    MOV SS,AX
0031 8D26C200 58    LEA SP,T_O_S
0035 9A0000--- E  59    AGAIN: CALL CI  ;GET CHARACTER FROM KEYBOARD
0038 3C9D  60    CMP AL,CR  ;IS IT CARRIAGE RETURN?
003C 746C  61    JE CRLF  ;IF YES THEN OUTPUT CR/LF
003E 50    62    PUSH AX  ;PASS CHAR. ON STACK
0040 EBFFFE 63    CALL SHIFT  ;CONVERT TO UPPER CASE IF SHIFT LOCKED
0042 50    64    PUSH AX
0044 9A0000--- E  65    CALL CO  ;OUTPUT CHAR ON SCREEN
0048 EBEB  66    JMP AGAIN
004A B00D  67
004C 50    68
004D 9A0000--- E  72    CALL CO  ;OUTPUT A CARRIAGE RETURN
0052 B00A  73    MOV AL,LF
0054 50    74    PUSH AX
0055 9A0000--- E  75    CALL CO  ;OUTPUT A LINE FEED
0058 EB09  76    JMP AGAIN  ;GO BACK TO GET NEXT CHAR.

0060 77  CODE ENDS
0061 78    END START

B-12
## 8086/87/88/186 MACRO ASSEMBLER INTERRUPT_HANDLER

**OBJECT MODULE PLACED IN:** :F2:LAB3E.OBJ  
**ASSEMBLER INVOKED BY:** :F3:ASM86.B6 :F2:LAB3E.ASM  
**SYMBOLS DEBUG**

### Location OBJECT LINE SOURCE

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>NAME</th>
<th>SOURCE</th>
</tr>
</thead>
</table>
| 0400 | 6 | INTERRUPT_HANDLER | ;THIS IS THE OPTIONAL EXERCISE TO WRITE AN INTERRUPT HANDLING ROUTINE  
| | | | ;THIS WILL HANDLE THE INTERRUPT FOR DIVIDE ERROR |
| 0400 | 7 | DIVIDE_SEG | ;PORT FOR DIVIDEND |
| 0400 | 8 | QUOTIENT | ;PORT FOR DIVISOR |
| 0400 | 9 | ERROR | ;PORT FOR DIVIDEND |
| 0400 | 10 | INTERRUPT | ;PORT FOR DIVIDEND |
| 0400 | 11 | INTERRUPT_SEGMENT | ;PORT FOR DIVIDEND |
| 0400 | 12 | INTERRUPT_CS | ;PORT FOR DIVIDEND |
| 0400 | 13 | INTERRUPT_ENDS | ;PORT FOR DIVIDEND |
| 0400 | 14 | INTERRUPT | ;PORT FOR DIVIDEND |
| 0400 | 15 | INTERRUPT | ;PORT FOR DIVIDEND |
| 0400 | 16 | INTERRUPT | ;PORT FOR DIVIDEND |
| 0400 | 17 | INTERRUPT | ;PORT FOR DIVIDEND |

### 1600 (100)  

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>NAME</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>005B</td>
<td>18</td>
<td>TOP</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>19</td>
<td>STACK</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>20</td>
<td>DIVIDE</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>21</td>
<td>SEGMENT</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>22</td>
<td>ASSUME</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>23</td>
<td>CS:DIVIDE</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>24</td>
<td>ALARM</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>25</td>
<td>DB</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>26</td>
<td>DIVIDE_ERROR:</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>27</td>
<td>PUSH AX</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>28</td>
<td>NOT ALARM</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>29</td>
<td>MOV AL,ALARM</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>30</td>
<td>MOV AL,ALARM</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>31</td>
<td>MOV AL,ALARM</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>32</td>
<td>MOV CX,3</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>33</td>
<td>OUTER:</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>34</td>
<td>MOV CX,0FFFH</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>35</td>
<td>INNER:</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>36</td>
<td>MOV CX,AX</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>37</td>
<td>LOOP OUTER</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>38</td>
<td>LOOP OUTER</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>39</td>
<td>LOOP OUTER</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>40</td>
<td>LOOP OUTER</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>41</td>
<td>LOOP OUTER</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>42</td>
<td>LOOP OUTER</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>43</td>
<td>LOOP OUTER</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>44</td>
<td>LOOP OUTER</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>45</td>
<td>LOOP OUTER</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>46</td>
<td>LOOP OUTER</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>005B</td>
<td>47</td>
<td>LOOP OUTER</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
</tbody>
</table>

### 0018—  

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>NAME</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>001F8</td>
<td>48</td>
<td>START:</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
<tr>
<td>001F8</td>
<td>49</td>
<td>MOV AX,STACK</td>
<td>;PORT FOR DIVIDEND</td>
</tr>
</tbody>
</table>

---

**Alarm:**  
- Alarms are used to complement the LED pattern.  
- The `ALARM` pattern is held for a duration of approximately 0.6 seconds.

**Divider Error Handling:**  
- The divider error handling routine is set to be loaded at the starting address 0000000:
  - Push AX to save registers used.
  - Push CX to set the initial value for the division.
  - Move AL, ALARM to set the flash value.
  - Move error, AL to send it out.
  - Set delay to about 0.6 seconds.
  - Pop CX to get back registers.
  - Pop AX to get back registers.
  - IRET to return.

**Main Segment:**  
- The main segment assumes that the main, interrupt, and stack segments are used.
  - The `START:` label is used as the starting address for the main program.
  - Moves AX to the stack to initialize the stack.
LOC OBJ | LINE | SOURCE
--- | --- | ---
0003 B8E0 | 49 | MOV SS,AX
0005 8226C800 | 50 | LEA SP,TOP
0009 B80000 | 51 | MOV AX, INTERRUPT
000C B8E0 | 52 | MOV DS,AX ; HAVE DS POINT TO LOAD VECTOR TABLE

; THESE NEXT TWO INSTRUCTIONS WILL MAKE THE VECTOR POINT TO THE INTERRUPT ROUTINE TO HANDLE A DIVIDE ERROR

000E C70600000100 | 57 | MOV DIV_ERR_IP,OFFSET DIVIDE_ERROR
0014 C7068200 | 58 | MOV DIV_ERR_CS, DIVIDE

; THIS PART OF THE PROGRAM WILL INPUT THE DIVIDEND AND DIVISOR AND DIVIDE.
; THE RESULT OF THE DIVISION WILL BE OUTPUT TO THE PORT 0 LEDS. THIS WILL BE DONE CONTINUOUSLY.

001A E401 | 64 | AGAIN: IN AL,DIVISOR ; SET VALUE TO DIVIDE BY
001C B8D8 | 65 | MOV BL,AL ; AND SAVE IT
001E E400 | 66 | IN AL,DIVIDEND ; SET WHAT TO DIVIDE BY
0020 32E4 | 67 | XOR AH,AH ; AND CONVERT IT TO A WORD
0022 F6F3 | 68 | DIV BL
0024 E500 | 69 | OUT QUOTIENT, AL ; OUTPUT DIVISION RESULT TO LEDS
0026 EBF2 | 70 | JMP AGAIN ; DO THIS CONTINUOUSLY

71 | MAIN ENDS
72 | END START
LOC OBJ LINE SOURCE

1 ; THIS PROGRAM IS THE SOLUTION TO LAB4 PART 1 OF THE WORKSHOP.
2 ; IT DOES THE SAME AS LAB 3 PART 3 EXCEPT THE PROCEDURE IS IN
3 ; ANOTHER MODULE
4
5 EXTRN CO:FAR,CI:FAR,ENCRYPT:FAR
6
7 NAME LAB4_PART_1_MAIN
8
0000 CR EQU 00H
0001 LF EQU 00H
0002
0003 STACK SEGMENT
0004 DW 100 DUP(?)
0005
0006 T_O_S LABEL WORD
0007 STACK ENDS
0008
0009 CODE SEGMENT
0010 ASSUME CS:CODE, SS:STACK
0011
0012 D8---- R 21 START: MOV AX,STACK ; INITIALIZE STACK
0013 8ED0 22 MOV SS,AX
0014 0005 826C8000 23 LEA SP,T_O_S
0015 0009 9A0000---- E 24 AGAIN: CALL CI ; SET CHARACTER FROM KEYBOARD
0016 000C 3C0D 25 CMP AL,CR ; IS IT CARRIAGE RETURN?
0017 0010 740E 26 JE CRLF ; IF YES THEN OUTPUT CR/LF
0018 0012 9A0000---- E 27 PUSH AX ; PASS CHAR. ON STACK
0019 0013 9A0000---- E 28 CALL ENCRYPT ; TRANSFORM IT
001A 0016 9A0000---- E 29 PUSH AX
001B 0019 9A0000---- E 30 CALL CO ; OUTPUT CHAR ON SCREEN
001C 001E EBE9 31 JMP AGAIN
001D
0020 D800 35 CRLF: MOV AL,CR
0021 8ED0 36 PUSH AX
0022 0023 9A0000---- E 37 CALL CO ; OUTPUT A CARRIAGE RETURN
0025 0028 B00A 38 MOV AL,LF
0026 002A 9A0000---- E 39 PUSH AX
002B 0028 9A0000---- E 40 CALL CO ; OUTPUT A LINE FEED
002D 0030 EBD7 41 JMP AGAIN ; GO BACK TO SET NEXT CHAR.
002E
0030 0033 EB0D 42 CODE ENDS
0031
0032 0033 EBEB 43 END START

B-15
NAME LAB4_PART_1_SUB

SWITCHES EQU 0

PUBLIC ENCRYPT

PRO SEGMENT

ASSUME CS:PRO

TABLE DW PLUS_1,MINUS_1,PLUS_2 ;JUMP TABLE

ERROR PROC FAR

; THIS PROCEDURE WILL ENCRYPT THE CHARACTERS ACCORDING TO THE
; VALUE READ FROM PORT 0.

PUSH BP ;SAVE BP

MOV BP,SP ;USE AS REFERENCE IN STACK

IN AL,SWITCHES ;FIND OUT WHICH ONE

CMP AL,2 ;SEE IF OUT OF RANGE

JA ERROR ;YES THEN EXIT

XOR AH,11 ;OTHERWISE CONVERT TO WORD

MOV SI,AX ;PUT IT IN AN INDEX REGISTER

MOV AX,(BP+61) ;BET CHARACTER

JMP TABLE[SI] ;ENCRYPT IT

ERROR:

MOV AL,' ' ;ILLEGAL CHARACTER

EXIT: POP BP ;IN AI..

RET 2 ;DELETES PARAMETER FROM STACK

PLUS_1: INC AL ;INCREMENT CHARACTER

MINUS_1: DEC AL ;DECREMENT CHARACTER

PLUS_2: ADD AL,2 ;ADD 2 TO CHARACTER

ENCRYPT ENDP
MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB4_PART_2_SUB

ASSEMBLER INVOKED BY: :F3:ASM86.86 :F2:LAB4B.ASM SYMBOLS DEBUG

LOC OBJ  LINE  SOURCE

1  NAME  LAB4_PART_2_SUB
2  
3  PUBLIC  ENCRYPT
4  
5  TRANS  SEGMENT
6  41H  DUP  ('*')  ;ONLY  LETTERS  ENCRYPTED
7  2A  
8  2A  
9  2A  

0541   5A595857565554  7  DB  'ZYXVUTSRQOPMLKJIHGFEDCBA'
   535251504FA4EA4D
   4CA4A9484746
   454434241

055B   5A595857565554  9  DB  'ZYXVUTSRQOPMLKJIHGFEDCBA'
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  10  DB  5  DUP  ('*')
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  11  TRANS  ENDS
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  12  PRO  SEGMENT
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  13  ASSUME  CS:PRO,DS:TRANS
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  14  ENCRYPT  PROC  FAR
   535251504FA4EA4D
   4CA4A9484746
   454434241

15  ;THIS  PROCEDURE  WILL  ENCRYPT  THE  CHARACTERS  ACCORDING  TO  THE
16  ;VALUE  READ  FROM  PORT  0.
17  

057B   5A595857565554  18  0000  55  20  PUSH  BP  ;SAVE  BP
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  19  0001  88EC  21  MOV  BP,SP  ;USE  AS  REFERENCE  IN  STACK
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  20  0003  1E  22  PUSH  DS  ;SAVE  DS  AND  BX  SINCE  WE  ARE  USING
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  21  0004  53  23  PUSH  BX
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  22  0085  BB----  R  24  MOV  AX,TRANS
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  23  0006  8EDB  25  MOV  DS,AX
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  24  0004  BDI0000  26  LEA  BX,TABLE
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  25  000C  884606  27  MOV  AX,[BP+61]
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  26  0011  D7  28  XLATB  ;CONVERT  THE  CHARACTER  AND  LEAVE  IT  IN  AL
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  27  0012  5B  29  POP  BX  ;GET  BACK  THE  REGISTERS
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  28  0013  1F  30  POP  DS
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  29  0014  5D  31  POP  BP
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  30  0015  8A0200  32  RET  2  ;DELETE  PARAMETER  FROM  STACK
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  31  0015  C90200  33  ENCRYPT  ENDP
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  32  0015  D7  34  PRO  ENDS
   535251504FA4EA4D
   4CA4A9484746
   454434241

057B   5A595857565554  33  0015  DB0200  35  END
   535251504FA4EA4D
   4CA4A9484746
   454434241

B-17
CO and Cl

/*

THIS PROGRAM DOES THE CONSOLE OUTPUT FROM THE SERIES III
IT IS BEING LINKED WITH AN ASSEMBLY LANGUAGE ROUTINE THAT
EXPECTS IT IN LARGE MODEL. THIS PROGRAM USES SYSTEM CALLS
TO DO THE OUTPUTTING TO THE CONSOLE.*/

/* THESE ARE THE DECLARATIONS FOR THE EXTERNAL PROCEDURES
THAT IMPLEMENT THE CONSOLE OUTPUT FUNCTIONS.*/

COMOD:  DO;
        DECLARE FLAG BYTE INITIAL (0FFH);
        DQ$CREATE: PROCEDURE (PATH$PNTR, EXCP$PTR) WORD EXTERNAL;
                 DECLARE PATH$PNTR POINTER, EXCP$PTR POINTER;
        END;
        DQ$OPEN: PROCEDURE (CONN, ACCESS, NUM$BUF, EXCP$PTR) EXTERNAL;
                 DECLARE CONN WORD, ACCESS BYTE, NUM$BUF BYTE,
                             EXCP$PTR POINTER;
        END;
        DQ$WRITE: PROCEDURE (CONN, BUFF$PTR, COUNT, EXCP$PTR) EXTERNAL;
                 DECLARE CONN WORD, BUFF$PTR POINTER, COUNT WORD,
                             EXCP$PTR POINTER;
        END;
        CO: PROCEDURE (CHAR) PUBLIC;
            DECLARE CHAR BYTE;
            DECLARE CONN WORD, ERR WORD;

/\* WE SHOULD ONLY MAKE ONE CONNECTION AND ONE OPEN ON CO. THEREFORE
WE MUST CHECK FIRST TO SEE IF THIS IS THE FIRST TIME THIS ROUTINE HAS
BEEN CALLED.*/

      IF FLAG THEN
          DO;
              FLAG=0;
              CALL DQ$CREATE ( @4, "CO: " ), @ERR);
              CALL DQ$OPEN (CONN, 2, 0, @ERR);
          END;
          CALL DQ$WRITE (CONN, @CHAR, 1, @ERR);
      END CO;
      END COMOD;
/*

THIS PROGRAM IS WRITTEN FOR USE WITH AN ASSEMBLY LANGUAGE
PROGRAM. THIS PROGRAM DOES THE INPUTTING OF CHARACTERS FROM THE SERIES
III. IT USES SYSTEMS CALLS AND MUST BE LINKED WITH THE SYSTEM
LIBRARIES. THIS PROGRAM IS BEING LINKED WITH AN ASSEMBLY LANGUAGE
ROUTINE THAT EXPECTS THIS ROUTINE IN LARGE MODEL. */

CIMOD: DO;
/* THIS FLAG IS USED BY THE PROCEDURE TO TELL IF ITS BEING CALLED
FOR THE FIRST TIME OR SOME TIME AFTER THE FIRST CALL. */

DECLARE FLAG BYTE INITIAL (0FFH);
CO: PROCEDURE (CHAR) EXTERNAL;
DECLARE CHAR BYTE;
END;

/* THESE ARE THE DECLARATIONS FOR THE EXTERNAL SYSTEM CALLS NECESSARY
FOR CONSOLE INPUT. */

DQ$ATTACH: PROCEDURE (PNTR, EXCP$PTR) WORD EXTERNAL;
DECLARE PNTR POINTER, EXCP$PTR POINTER;
END;

DQ$READ: PROCEDURE (CONN, BUF$PNTR, COUNT, EXCP$PTR) WORD EXTERNAL;
DECLARE CONN WORD, BUF$PNTR POINTER, COUNT WORD,
EXCP$PTR POINTER;
END;

DQ$SPECIAL: PROCEDURE (TYPE, PARAM$PTR, EXCP$PTR) EXTERNAL;
DECLARE TYPE BYTE, PARAM$PTR POINTER, EXCP$PTR POINTER;
END;

DQ$OPEN: PROCEDURE (CONN, ACCESS, NUM$BUFF, EXCP$PTR) EXTERNAL;
DECLARE CONN WORD, ACCESS BYTE, NUM$BUFF BYTE,
EXCP$PTR POINTER;
END;
CO and CI

/*

CI: PROCEDURE BYTE PUBLIC;
DECLARE CONN WORD, ERR WORD,
ACTUAL WORD, BUFFER (80) BYTE,
I BYTE, SIGNON (*) BYTE DATA (1BH,45H,0AH,0AH,0AH,‘COMMUNICATION LINK
ESTABLISHED.’,0DH,0AH);

/* THIS IS THE MAIN ROUTINE. FIRST WE MUST ATTACH CI TO GET
A CONNECTION. THE SYSTEM CALL OPEN IS USED TO OPEN THE CONSOLE
AND THEN WE USE A SYSTEM CALL (DQSPECIAL) TO MAKE
THE CONSOLE INPUT TRANSPARENT. FINALLY WE DO A READ FROM
THE KEYBOARD TO READ IN THE CHARACTER. */

/* WE SHOULD ONLY MAKE A CONNECTION/OPEN ONCE. THEREFORE WE MUST
CHECK TO SEE IF THIS IS THE FIRST TIME THAT THIS PROCEDURE IS
CALLED. IF FLAG IS FF (TRUE), THEN THIS IS THE FIRST TIME. */

IF FLAG THEN
DO;
    FLAG=00;
    CONN= DQ$ATTACH (@(4,’:CI:’),@ERR);
    CALL DQ$OPEN (CONN,1,0,@ERR);
    CALL DQ$SPECIAL (1,@CONN,@ERR); /* THE FIRST PARAM SPECIFIES
        TRANSPARENT MODE*/
    /*OUTPUT A SIGNON MESSAGE*/
    DO I=0 TO LAST(SIGNON);
        CALL CO (SIGNON(I));
    END;
END;

ACTUAL=DQ$READ (CONN,@BUFFER(0),1,@ERR); /* THE 1 SPECIFIES THE
    THE NUMBER OF BYTES TO
    INPUT*/

RETURN BUFFER(0);
END CI;
END CIMOD;
*/
3.1 1. CS:IP
2. Any combination of XXXX and YYYY so that when they are added as shown they will result in 05820H.
   \[ \text{CS} \quad \text{XXXX} \]
   \[ \text{IP} \quad + \quad \text{YYYY} \]
   \[ 05820 \]
3. DS, and BX, BP, SI, or DI
4. 00230H
5. 0002H

REVIEW (FILL IN REGISTER NAMES)

\[
\begin{array}{c|c|c}
\text{CS} & \text{IP} & \text{XXXX} \\
\hline
\text{DS} & \text{BX}, \text{BP}, \text{SI}, \text{DI} & \text{DATA} \\
\hline
\text{SS} & & \text{STACK} \\
\hline
\text{DS} & & \text{DATA} \\
\hline
\text{BX} & & \text{DATA} \\
\hline
\text{BP} & & \text{DATA} \\
\hline
\text{CS} & \text{IP} & \text{FFFFF} \\
\end{array}
\]
CLASS EXERCISE SOLUTIONS

4.1

```
MOV DX, OFFFFBH
IN AL, DX
MOV AH, 0
MOV CL, 3                ; or SHL AX, 1
                     ; SHL AX, 1
SHL AX, CL             ; SHL AX, 1
OUT 8, AX
``` 

5.1

1. The END statement is an assembler directive. It never gets encoded and as a result it never gets executed.

2. GOOD a, b, c, d, g, h, and j
BAD e - ' is an illegal character
f - starts with a number
i - $ is an illegal character

7.1

```
NAME CLASS_EXERCISE_7_1

SWITCHES EQU 0
LITES EQU 1

CODE SEGMENT
ASSUME CS: CODE

START: IN AL, SWITCHES
        SUB AL, 32
        MOV BL, 5
        MUL BL
        MOV BL, 9
        DIV BL
        OUT LITES, AL
        JMP START

CODE ENDS
END START
```
7.2

CLASS_EXERCISE_7_2

<table>
<thead>
<tr>
<th>NAME</th>
<th>CLASS_EXERCISE_7_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS_PORT</td>
<td>EQU 10</td>
</tr>
<tr>
<td>DATA_PORT</td>
<td>EQU 11</td>
</tr>
<tr>
<td>RDY</td>
<td>EQU 00000001B</td>
</tr>
</tbody>
</table>

POLL

SEGMENT
ASSUME CS:POLL
HANDSHAKE:  
IN AL,STATUS_PORT
TEST AL,RDY
JZ HANDSHAKE
IN AL,DATA_PORT
CMP AL,43
JA ERROR

---

HLT

ERROR:

---

HLT

ENDS
END HANDSHAKE

8.1

1. WAREA DW 2000H
2. BAREA DB ?
3. MOV BAREA,10
4. AND WAREA,40H
5. TEST WAREA,3000H

8.2

NAME CLASS_EXERCISE_8_2
PAYROLL SEGMENT
PAYSCALE DB 100 DUP(?)
PAYROLL ENDS

PAYRAISE SEGMENT
ASSUME CS:PAYRAISE,DS:PAYROLL
INIT: MOV AX,PAYROLL
       MOV DS,AX
       XOR SI,SI
       MOV CX,100
AGAIN: ADD PAYSCALE[SI],50
       INC SI
       LOOP AGAIN
       HLT
PAYRAISE ENDS
END INIT

C-3
9.1
RUN ASM86 :F1:PROB.LEM SB DB PR(F1:LISTING)
RUN LINK86 :F1:PROB.OBJ BIND

10.1
1. MAX mode
2. 8 Mhz
3. The CPU will run at 5 Mhz rather than 8 Mhz

11.1
NAME CLASS_EXERCISE_12_1

STACK SEGMENT
DW 100 DUP(?)
T_O_S LABEL WORD
STACK ENDS

DATA SEGMENT
CTEMP DW ?
TABLE DB 51 DUP(?)
FTEMP DB ?
DATA ENDS

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

CONVERT PROC
-----
RET 6
CONVERT ENDP

INIT:
MOV AX,DATA
MOV DS,AX
MOV AX,STACK
MOV SS,AX
LEA SP,T_O_S

CALLPROC:
PUSH CTEMP
MOV AX,LENGTH TABLE
PUSH AX
LEA AX,TABLE
PUSH AX
CALL CONVERT
MOV FTEMP,AL
HLT

CODE ENDS
END INIT

C-4
13.1

NAME

CLASS.EXERCISE_14_1

INTERRUPT

SEGMENT AT 0

DIV_ERR_IP

DW ?

DIV_ERR_CS

DW ?

INTERRUPT

ENDS

ERROR

SEGMENT

DIV_ERROR:

MOV AX,OFF00H

IRET

ERROR

ENDS

MAIN

SEGMENT

ASSUME CS:MAIN,DS:INTERRUPT

START:

MOV AX, INTERRUPT

MOV DS, AX

MOV DIV_ERR_IP,OFFSET DIV_ERROR

MOV DIV_ERR_CS,ERROR

DIV BL

MAIN

ENDS

END START

14.1

1. 04001H

2. a) There is no bank selection using A0 and BHE

b) We do not have to worry about writing extraneous data to the unwanted bank since we never write to a ROM.

3. Yes, but it will take two bus cycles

4. a) no

b) $TAD - Tacc - Tdelay = ?$

$295 - 250 - 60 = ?$

295 - 250 - 60 - 15 = ?$

Yes one wait state
### 15.1

```asm
NAME CLASS_EXERCISE_15_1
DATA SEGMENT
TABLE DB '5047283916'
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE,DS:DATA

ENCRYPT PROC
    JCXZ EXIT
    PUSH DS
    PUSH BX
    MOV BX,DATA
    MOV DS,BX
    LEA BX,TABLE
AGAIN:
    MOV AL,ES:[SI]
    SUB AL,30H
    XLATB
    MOV ES:[SI],AL
    INC SI
    LOOP AGAIN
    POP BX
    POP DS
    EXIT:
    RET
ENCRYPT ENDP
INIT: -------
CODE ENDS
END INIT
```

### 16.1

<table>
<thead>
<tr>
<th>NAME</th>
<th>MODA</th>
<th>MODD</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC</td>
<td>USEFUL_DATA, HANDY</td>
<td>EXTRN USEFUL_DATA:BYTE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXTRN HANDY:FAR</td>
</tr>
<tr>
<td>DATA</td>
<td>SEGMENT</td>
<td>B_CODE SEGMENT</td>
</tr>
<tr>
<td>USEFUL_DATA</td>
<td>DB</td>
<td>ASSUME CS:B_CODE</td>
</tr>
<tr>
<td>DATA</td>
<td>ENDS</td>
<td>&amp; DS:SEG USEFUL_DATA</td>
</tr>
<tr>
<td>A_CODE</td>
<td>SEGMENT</td>
<td>MOV AX,SEG USEFUL_DATA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CALL HANDY</td>
</tr>
<tr>
<td>HANDY</td>
<td>PROC FAR</td>
<td>MOV DS,AX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOV AL,USEFUL_DATA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B_CODE ENDS</td>
</tr>
<tr>
<td>HANDY</td>
<td>ENDP</td>
<td>END</td>
</tr>
<tr>
<td>A_CODE</td>
<td>ENDS</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

C-6
CLASS EXERCISE SOLUTIONS

19.1

1. BM2 DRIVES BUSY HIGH
2. BM2 ISSUES CBRO HIGH
3. BM2 DRIVES BPRN HIGH
4. BM2 TAKES OVER BUSY, DRIVES BUSY LOW
5. BM3 SEES CBRO LOW
6. BM3 SEES BPRN HIGH
Quiz #1

1. Match the pointer with the appropriate memory area:

<table>
<thead>
<tr>
<th>CPU</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>STACK</td>
</tr>
<tr>
<td>SP</td>
<td>INSTRUCTIONS</td>
</tr>
<tr>
<td>DI</td>
<td>ROM/PROM/EPROM/RAM</td>
</tr>
<tr>
<td></td>
<td>VARIABLES</td>
</tr>
<tr>
<td></td>
<td>RAM</td>
</tr>
</tbody>
</table>

2. What is the state (1,0) of the zero flag after the CPU executes the following arithmetic operations?

\[
\begin{align*}
5FH - 5FH & = 0FH \\
5FH - 4FH & = 1FH \\
5FH - 6FH & = 3FH \\
\end{align*}
\]

3. Which SEG REG and offset REG would the CPU use to generate an address for the following types of memory access?

<table>
<thead>
<tr>
<th>Type of Access</th>
<th>SEG</th>
<th>OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op code fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data access</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Daily Quiz Tuesday
4. Where does the CPU get immediate data?

5. What is wrong with the following 8086 instructions and what can be done to make them work?

    IN  AL,0FFFFH

    SAR  AX,5
1. Match the following:

TEST ____

CMP ____

NOT ____

NEG ____

ADC ____

CBW ____

CWD ____

a. 2's complement
b. Used for multi-word addition
c. "Non-destructive" AND
d. Used when dividing one signed word by another
e. 1's complement
f. "Non-destructive" subtract
g. Used when dividing one signed byte by another

2. For every data definition (variable), the assembler keeps track of what three attributes?

3. Fill in the spaces to represent the condition of the registers in an 8086 CPU after being reset.

FLAGS ________

CS ________

IP,DS,SS,ES ________

AX,BX,CX,DX ________

4. What address will the 8086 CPU begin execution after being reset

T F 5. In the MIN mode, the CPU is the source of the control bus signals.

T F 6. DIV 35H is a valid instruction.

T F 7. You can have more than one ASSUME statement in a code segment.
8. What are the abbreviations for the following assembler controls?

NOPRINT__________
LIST__________
DEBUG__________
SYMBOLS__________
EJECT__________

Daily Quiz Wednesday
1. What is the difference between the CALL and JMP instruction?

2. Each item in the following problem represents a step in the response of an 8086 to an interrupt request. Number each item in the space provided so the steps occur in the correct order. The first item has been correctly numbered as a starting point.

   T T T
   1. IF and TF are cleared
   2. CPU completes execution of current instruction
   3. CS and IP loaded from Interrupt Vector Table
   4. Flags pushed onto stack
   5. CS and IP pushed onto stack

   TRUE - FALSE (circle one)

   T F 3. You can PUSH and POP a 16-bit register.
   T F 4. You can PUSH and POP an 8-bit memory location.
   T F 5. You can PUSH immediate data in the 8088.
   T F 6. A procedure with a FAR attribute will always generate a FAR return.

   7. What is the physical address for the Interrupt Vector Table entry for a type 10 interrupt?

   8. What does the assembler use to determine if it must generate a segment override prefix?

   9. What prevents the RAMs shown on page 14-9 from responding to an I/O address such as the one generated by the instruction IN AL,OFFH?
Quiz #4

1. Can a string operation (using the REP prefix) be interrupted?

2. Where can you find the definition of an assembler error code?

3. What directive would be used in a module to allow it to call the FAR procedure INPUT that is in another module?

4. Is IMUL XYZ,BX,7 a legal 80186 instruction?
APPENDIX E

UNPACKED DECIMAL ARITHMETIC

INSTRUCTIONS
PACKED DECIMAL

* BINARY ADDITION AND SUBTRACTION USED

* RESULT IN AL REGISTER ADJUSTED

DAA (DECIMAL ADJUST FOR ADDITION)
ADDS 06 60 AS REQUIRED

DAS (DECIMAL ADJUST FOR SUBTRACT)
SUBTRACTS 06 60 AS REQUIRED
DECIMAL ADJUST ADDITION

* PURPOSE: CONVERTS RESULT OF BINARY ADDITION TO BCD VALUE

RULE 1: IF $A_{\text{LOW}} > 9$ OR IF A.C. = 1 THEN ADD 6

RULE 2: IF $A_{\text{HI}} > 9$ OR IF C = 1 THEN ADD 60

EXAMPLES:

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>0010 1001</td>
</tr>
<tr>
<td>+ 1</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>0010 1010</td>
</tr>
<tr>
<td></td>
<td>0110 (RULE 1)</td>
</tr>
<tr>
<td></td>
<td>0011 0000</td>
</tr>
<tr>
<td>18</td>
<td>0001 1000</td>
</tr>
<tr>
<td>+18</td>
<td>0001 1000</td>
</tr>
<tr>
<td>36</td>
<td>0011 0000</td>
</tr>
<tr>
<td></td>
<td>0110 (RULE 1)</td>
</tr>
<tr>
<td></td>
<td>0011 0110</td>
</tr>
<tr>
<td>72</td>
<td>0111 0010</td>
</tr>
<tr>
<td>+93</td>
<td>1001 0011</td>
</tr>
<tr>
<td>165</td>
<td>0000 0101</td>
</tr>
<tr>
<td></td>
<td>0110 0000 (RULE 2)</td>
</tr>
<tr>
<td></td>
<td>0110 0101</td>
</tr>
</tbody>
</table>
(ASCII) - UNPACKED DECIMAL ARITHMETIC

. FORMAT - 1 BCD DIGIT PER BYTE
. ZONE DIGIT SET TO ZERO
. BINARY ADD AND SUBTRACT USED
. ASCII INSTRUCTIONS:

. ADJUST AL LOW DIGIT +6
. SET AL HIGH DIGIT TO 0
. MODIFY AH BY 1 FOR CARRY/BORROW
. MODIFIES CARRY FLAG

EXAMPLE

MOV     AL,  ALPHA
ADD     AL,  BETA
AAA ; ALPHA + BETA
OR AL,  30H
AAA     ADDS 00 } AS REQUIRED
AAS     SUBTRACTS 06
UNPACKED DECIMAL ARITHMETIC

* BINARY ADD, SUBTRACT, MULTIPLICATION AND DIVISION USED

* INSTRUCTIONS ADJUST VALUE IN AL REGISTER

* INSTRUCTIONS -
  AAA -- ASCII ADJUST AFTER ADDITION
  AAS -- ASCII ADJUST AFTER SUBTRACTION
  AAM -- ASCII ADJUST AFTER MULTIPLY
  AAD -- ASCII ADJUST BEFORE DIVIDE
ASCII ADJUST EXAMPLE

\[
\begin{array}{c}
Z \ 5 \\
+ \ Z \ 6 \\
\hline
X \ B \\
\end{array}
\]

\[
\begin{array}{c}
\text{+ 6} \\
\hline
\text{AH} & \text{AL} \\
\end{array}
\]

\[
\begin{array}{c}
\text{XXX 0101} \\
+ \ \text{XXX 0110} \\
\hline
\text{XXX 1011} \\
\end{array}
\]

\[
\begin{array}{c}
\text{+ 1} \\
\hline
\text{0110} \\
\end{array}
\]

\[
\begin{array}{c}
\text{+1} \\
\hline
\text{0000} & \text{0001} \\
\end{array}
\]

AAA
OPERATION: \( C = A + B \); where \( A \) and \( B \) are strings of ASCII digits, and \( C \) is to be a string of unpacked BCD digits.

\[
\begin{align*}
\text{MOV} & \quad \text{BX, STRING\_LENGTH - 1} \\
\text{CLC} & \\
\text{NEXT: MOV} & \quad \text{AL, A[BX]} \\
\text{ADC} & \quad \text{AL, B[BX]} \\
\text{AAA} & \\
\text{MOV} & \quad \text{C[BX], AL} \\
\text{DEC} & \quad \text{BX} \\
\text{JNS} & \quad \text{NEXT}
\end{align*}
\]

NOTE: The upper nibble after the AAA is set to zero. Any carry is saved in the carry flag for the next ADC. The carry is also added to AH, but this fact is not utilized in the above code.

CLASS PROBLEM:

Write a program segment that will perform the operation \( C = A - B \). Use the same assumptions as above.
(ASCII) UNPACKED DECIMAL DIVIDE

AAD ASCII ADJUST DIVIDE

Adjusts a dividend in AX register prior to a divide operation to provide an unpacked decimal quotient.

Example

```
MOV AL, ALPHA
AAD
DIV BETA ; ALPHA/BETA
```

The AH register data is multiplied by ten and added to AL register. AH is set to zero.

This places the binary equivalent of the two digits from AH, AL into AL, in preparation for a binary division.

The binary division will leave the integer quotient in AL, and the integer remainder in AH.

Note: The remainder in AH will always be smaller than the division and is in correct form for the next AAD instruction. The user must be sure that this condition is true for the first operation.
ASCII ARITHMETIC - DIVISION

OPERATION: \( C = \frac{A}{B} \); WHERE A IS A STRING OF ASCII DIGITS, AND B IS A SINGLE ASCII DIGIT. C IS TO BE A STRING OF UNPACKED BCD DIGITS.

SETUP:

\[
\begin{align*}
\text{MOV} & \quad \text{DL, B} \quad ; \text{GET B} \\
\text{MOV} & \quad \text{SI, OFFSET A} \quad ; \text{POINTER TO A} \\
\text{MOV} & \quad \text{DI, OFFSET C} \quad ; \text{POINTER TO C} \\
\text{MOV} & \quad \text{CX, LENGTH A} \quad ; \text{# OF TIMES TO LOOP} \\
\text{CLD} & \quad ; \text{AUTO INCREMENT} \\
\text{AND} & \quad \text{DL, OFH} \quad \quad ; \text{RID B OF ZONE} \\
\text{XOR} & \quad \text{AH, AH} \quad \quad ; \text{SEED LOOP} \\
\end{align*}
\]

NEXT:

\[
\begin{align*}
\text{LODS} & \quad \text{A} \quad ; \text{GET BYTE} \\
\text{AND} & \quad \text{AL, OFH} \quad \quad ; \text{ZERO ZONE} \\
\text{AAD} & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad ; \text{ADJUST FOR DIVIDE} \\
\text{DIV} & \quad \text{DL} \\
\text{STOS} & \quad \text{C} \quad ; \text{SAVE QUOTENT BYTE} \\
\text{LOOP} & \quad \text{NEXT} \\
\end{align*}
\]

NOTE: THE AAD MULTIPLIES THE REMAINDER FROM THE PREVIOUS DIVIDE, (SAVED IN AH), BY 10 THEN ADDS THIS VALUE TO AL. AH IS CLEARED BEFORE ENTERING THE LOOP SO FIRST AAD WORKS PROPERLY.
(ASCII) UNPACKED DECIMAL MULTIPLICATION

THE AAM INSTRUCTION IS USED TO DIVIDE A NUMBER BY 10 AND IS USEFUL IN CONVERTING A BINARY NUMBER \leq 99 TO TWO BCD DIGITS.

IN APPLICATION, BINARY MULTIPLICATION IS USED ON 2 BCD DIGITS TO PRODUCE A BINARY PRODUCT. THE PRODUCT IS CONVERTED TO DECIMAL USING THE AAM INSTRUCTION. FINALLY, THE DECIMAL ADDITION CAN BE USED TO COMBINE PRODUCTS OF MULTIPLICATION.

**BINARY MULTIPLICATION**

A BCD DIGIT IS A VALID BINARY NUMBER AND CAN BE USED IN BINARY MULTIPLICATION.

**EXAMPLE:**

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>\times 9</td>
<td>\times 1001</td>
</tr>
<tr>
<td>81</td>
<td>1010001</td>
</tr>
</tbody>
</table>

BCD = BINARY

BINARY RESULT

* BINARY MULTIPLY
CONVERSION TO DECIMAL

TO CONVERT THE BINARY RESULT TO BCD IT IS NECESSARY TO DO A BINARY DIVIDE BY TEN.

EXAMPLE:

\[
\begin{align*}
81 & \div 10 = 8 & \text{Remainder 1} \\
1010001 & \div 1010 = 1000 & \text{Remainder 0001}
\end{align*}
\]

THE RESULT INDICATES THE NUMBER OF TENS AND ONES THAT CAN BE USED AS A TWO DIGIT BCD NUMBER. 81
ASCII ARITHMETIC - MULTIPLY

OPERATION: \( C = A \times B \); WHERE \( A \) IS A STRING OF ASCII DIGITS, AND \( B \) IS A SINGLE ASCII DIGIT. \( C \) IS TO BE A STRING OF UNPACKED BCD DIGITS.

SETUP: \[
\begin{align*}
& \text{MOV} \ DL, B & &; \text{GET SINGLE ASCII DIGIT} \\
& \text{MOV} \ CX, \ LENGTH \ A & &; \text{NUMBER OF TIMES TO LOOP} \\
& \text{STD} & &; \text{SET UP FOR AUTO DECREMENT} \\
& \text{MOV} \ SI, \ OFFSET \ A + LENGTH \ A -1 \\
& \text{MOV} \ DI, \ OFFSET \ C + LENGTH \ A -1 \\
& \text{MOV} \ BYTE \ PTR \ [DI], \ 0 & &; \text{CLEAR} \ C(1) \\
& \text{AND} \ DL, \ OFH & &; \text{CLEAR ZONE OF} \ B \\
\end{align*}
\]

NEXT: \[
\begin{align*}
& \text{LODS} \ A & &; \text{LOAD BYTE FROM} \ A \\
& \text{AND} \ AL, \ OFH & &; \text{CLEAR ZONE} \\
& \text{MUL} \ DL & &; \text{MULTIPLY BY} \ B \\
& \text{AAM} & &; \text{ADJUSTED RESULT IN} \ AX \\
& \text{ADD} \ AL, [DI] & &; \text{ACCUMULATE INTO} \ C \\
& \text{AAA} & &; \text{IN UNPACKED FORMAT} \\
& \text{STOS} \ \text{WORD} \ PTR \ C & &; \text{PROPAGATE UPPER DIGIT} \\
& \text{INC} \ DI & &; \text{POINT TO PROPER DIGIT} \\
& \text{LOOP} \ \text{NEXT} \\
\end{align*}
\]

NOTE: \( \text{AAM} \) PLACES THE UPPER DIGIT IN \( AH \). \( \text{AAA} \) PROPAGATES THE CARRY FROM THE LOWER NIBBLE BY ADDING THE CARRY TO \( AH \). THE \( C \) STRING IS ONE BYTE LONGER THAN THE \( A \) STRING.
MULTIPLICATION LOOP

UNPACKED BCD

MULTIPOLAND INDEX \( \text{SI} \)
PARTIAL PRODUCT INDEX \( \text{DI} \)
MULTIPLIER INDEX \( \text{BX} \)
MULTIPLIER LENGTH \( \text{B} \)
MULTIPOLAND LENGTH \( \text{C} \)

ZERO PARTIAL PRODUCT
MULTIPLIER INDEX \( \text{BX} = 1 \)

LOOP1: \( \text{DL} = 0 \)

INITIALIZE MULTIPOLAND INDEX \( \text{SI} = 1 \)

INITIALIZE PARTIAL PRODUCT INDEX: \( \text{DI} = \text{BX} \) (MULTIPLIER INDEX)

FETCH MULTIPOLAND \([\text{SI}]\) TO \(\text{AL}\)

\[
\begin{align*}
\text{MULTIPLY} & \quad \text{MULTIPLIER} \quad \text{[BX]} \quad \text{*} \quad \text{AL} \quad \rightarrow \quad \text{AL} \\
\text{ASCII} & \quad \text{MULTIPLY ADJUST} \quad \text{AX} \\
\text{ADD} & \quad \text{DL} \quad \text{TO} \quad \text{AL} \\
\text{ASCII} & \quad \text{ADD ADJUST} \quad \text{AL} \\
\text{ADD PARTIAL PRODUCT} & \quad \text{[DI]} \quad \text{TO} \quad \text{AL} \\
\text{ASCII} & \quad \text{ADD ADJUST} \quad \text{AL} \\
\text{STORE AL TO PARTIAL PRODUCT} & \quad \text{[DI]} \\
\text{SAVE} & \quad \text{DL} = \text{AH} \\
\text{DI} & = \text{DI} + 1 \\
\text{SI} & = \text{SI} + 1
\end{align*}
\]

IF \( \text{SI} \leq \text{C} \) (MULTIPOLAND LENGTH) TO TO LOOP 2
STORE DL TO PARTIAL PRODUCT \([\text{DI}]\)

\( \text{BX} = \text{BX} + 1 \)
IF \( \text{BX} \leq \text{B} \) (MULTIPLIER COUNT) GO TO LOOP 1
\[
\begin{array}{c}
374 \\
\times 152 \\
\hline
748 \\
1870 \\
374 \\
\hline
56848 \\
\end{array}
\]

2x4 = 08

2x7 = 14

2x3 = 06

5x4 = 20

5x7 = 35

1x4 = 04

1x7 = 07

1x3 = 03

E-13
APPENDIX F

ICE-86,88 IN-CIRCUIT EMULATOR
ICE-86, 88

ICE-86,88

* IN-CIRCUIT EMULATOR Allows HARDWARE AND SOFTWARE DEBUGGING.

* ICE-86 AND ICE-88 COMMANDS ARE IDENTICAL, THE HARDWARE IS NOT

* FEATURES INCLUDE:

  HARDWARE BREAKPOINTS
  TRACE DATA COLLECTION
  SYMBOLIC DEBUGGING
  MEMORY MAPPING
  DEBUGGING MACROS
  BUILT IN DISASSEMBLER

ICE-86 COMPONENTS AND ENVIRONMENT

ICE-86 SOFTWARE

ICE-86 CIRCUIT BOARDS

CAN BE PLUGGED INTO USER HARDWARE

BUFFER BOX: CONTAINS AN 8086 PROCESSOR
ICE-86 COMPONENTS

FM CONTROLLER PCB - 8080 ICEJIP, 12KB Firmware ROM, 3KB Scratchpad RAM
86 CONTROLLER PCB - 2KB ICE RAM, 1K x8 MAP RAM, 0.5K Dual Port RAM
ICE 86 TRACE PCB - Trace RAM

ICE-86 BUFFER BOX ASS'Y - 8086µP, GATING AND CONTROL LOGIC

INTELLEC SERIES II TRIPLE AUXILIARY CONNECTOR
“T” CABLE
GROUND CABLE

ICE-86 DISKETTE -
ICE86,OV5
ICE86,OV6
ICE86,OV7
ICE86,OV8
ICE86,OV9
ICE86,OYE
ICE86,OY8
ICE86,OY9
ICE86,OVE

SERIES II OR SERIES III DEVELOPMENT SYSTEM WITH 3 ADJACENT CARD SLOTS AVAILABLE AND 64KB OF RAM

OPTIONAL:
SERIAL OR PARALLEL PRINTER
EXPANSION MEMORY (ISBC 16,32 OR 64) (SERIES III CONTAINS 128K EXPANSION MEMORY)

ICE-86 INSTALLATION

1. INSURE THAT E-1 TO E-2 AND E-7 TO E-8 ARE JUMPERED ON FM CONTROLLER PCB.
2. INSTALL 3 PCB’S IN CHASSIS SO THAT FM CONTROLLER IS ON TOP, TRACE PCB IS NEXT, AND 86 CONTROLLER PCB IS ON THE BOTTOM.
3. INSTALL “T” CABLE BETWEEN TRACE PCB AND 86 CONTROLLER PCB.
4. ATTACH “X” CABLE TO “X” CONNECTOR AND ON 86 CONTROLLER PCB.
5. ATTACH “Y” CABLE TO “Y” CONNECTOR ON FM CONTROLLER PCB.
6. IF USER HARDWARE IS TO BE USED, REMOVE SOCKET PROTECTOR ASS’Y FROM UMBILICAL ASS’Y AND INSERT UMBILICAL PLUG INTO PROTOTYPE 8086 SOCKET.
7. CONNECT GROUND CABLE FROM CABLE ASS’Y TO PROTOTYPE HARDWARE GROUND.
8. POWER UP DEVELOPMENT SYSTEM AND PROTOTYPE.

NOTE:
TO PREVENT PIN DAMAGE INSTALL A 40 PIN IC SOCKET ON THE END OF THE UMBILICAL CORD. THE SOCKET ASS’Y PROTECTOR SHOULD BE IN PLACE WHENEVER ICE-86 IS NOT CONNECTED TO A PROTOTYPE.
PRODUCT DEVELOPMENT PHASES USING ICE-86

PHASE 1:
NO PROTOTYPE HARDWARE AVAILABLE—USE ICE-86 STANDALONE, DEBUG SOME OR ALL PROGRAM MODULES. PROGRAMS RESIDE IN ICE AND/OR MDS AND/OR DISK MEMORY.

PHASE 2:
SKELETON PROTOTYPE HARDWARE AVAILABLE—DEBUG HARDWARE BY EXECUTING TEST SOFTWARE. DEBUG SYSTEM WITH PROTOTYPE HARDWARE AND SOFTWARE. PROGRAMS RESIDE IN PROTOTYPE AND/OR ICE AND/OR MDS AND/OR DISK MEMORY. DOWN LOADING OF PROGRAMS DONE BY ICE, NO NEED TO BURN PROMS.
PRODUCT DEVELOPMENT PHASES USING ICE-86

PHASE 3:
COMPLETE PROTOTYPE SYSTEM AVAILABLE- DEBUG FULL HARDWARE AND SOFTWARE TOGETHER. USE ICE TO DOWNLOAD PROGRAMS. USE ICE FOR FINAL PRODUCT CHECKOUT.

NOTE:
ICE86 SHOULD NEVER BE USED ON A PRODUCTION LINE FOR PRODUCTION TESTING!

PROGRAM PREPARATION

BEFORE USING ICE-86, AN ABSOLUTE OBJECT FILE MUST BE CREATED. ALSO, HARD COPIES OF ALL DIAGNOSTIC INFORMATION SHOULD BE GENERATED.

RUN ASM86:F1:LAB1.A86 DEBUG
RUN LOC86:F1:LAB1.OBJ MAP SYMBOLS INITCODE
COPY:F1:LAB1LST,:F1:LAB1.MP2 TO :LP:
PREPARATION OF THE MAIN PROGRAM MODULE

SERIES -II

NAME EXAMPLE

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

START:
    MOV AX, DATA
    MOV DS, AX
    MOV AX, STACK
    MOV SS, AX
    LEA SP, STACK_TOP

END START

* SEGMENT REGISTER INITIALIZATION PERFORMED IN MAIN MODULE.

SERIES -III

NAME SERIES III EXAMPLE

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

START:
    MOV DX, USART CMD PORT

END START, DS:DATA, SS:STACK:STACK TOP

* END STATEMENT CREATES SEGMENT REGISTER INITIALIZATION RECORD. THIS RECORD IS REQUIRED THE INITCODE FEATURE OF LOC86.

INVOKING ICE-86

THE ICE-86 SOFTWARE DRIVER IS INVOKED FROM ISIS-II.

-ICE86

ONCE LOADED, CONTROL IS THEN PASSED TO THE SOFTWARE DRIVER. ICE-86 IS READY TO ACCEPT A COMMAND WHEN THE ICE PROMPT * IS DISPLAYED.
PREPARATION OF THE ENVIRONMENT

- MEMORY MAPPING
- CLOCK SELECTION
- READY SELECTION

PREPARATION OF THE ENVIRONMENT
MEMORY MAPPING

LOGICAL PROGRAM ADDRESS

MEMORY MAP

GUARDED
USER
ICE
INTELLEC
DISK

\[
\text{MAP partition} = \begin{cases} 
\text{GUARDED} & \text{[GUARDED [NOVERIFY]} \\
\text{ICE [physical segment number]} & \text{[NOVERIFY]} \\
\text{INTELLEC [physical segment number]} & \text{[NOVERIFY]} \\
\text{DISK [physical segment number]} & \text{[NOVERIFY]} 
\end{cases}
\]

where

\[
\text{partition = \{logical segment number [IOL logical segment number]\}}
\]

F-6
ICE-86 MEMORY MAPPING

* ICE-86 divides the megabyte of memory into 1024 1K blocks

* Each 1K block can be mapped into a physical 1K block

MAP TO USER MEMORY

NO ADDRESS DISPLACEMENT IS ALLOWED
LOGICAL AND PHYSICAL ADDRESS REFERENCES MUST BE THE SAME.

* MAP 0 LEN 32=USER
* MAP 1000=USER
MAPPING TO ICE-86 MEMORY

- MAP 0 = ICE 0
- MAP 1023 = ICE 1

MEMORY MAPPING EXAMPLE

- MAP 0 = USER
- MAP 1023 = ICE 0

Program Reference → Memory Map → 2K

Logical Memory:
- 0FFFFH: Program and Contents
- 0FC00H
- 3FFH: Variable Data and Stack

Physical Memory:
- 1K ROM
- 1K RAM

ICE0

USER
* DISPLAY MAP STATUS COMMAND

Example 1.
MAP 0 TO 3
Display:

0000T = USE 0001T = ICE 0002T = INT 0003T = DIS 0004T = DIS

Example 2.
MAP
Display:

0000T = DIS 0001T = ICE 0002T = DIS 0003T = DIS 0004T = USE 0005T = DIS

1027T = DIS

* RESET MAP COMMAND

RESET MAP

PREPARATION OF THE ENVIRONMENT
CLOCK SELECTION

* CLOCK = INTERNAL : DEFAULT
OR
* CLOCK = EXTERNAL

INTERNAL

ICE-86 CLOCK

USER CLOCK

EXTERNAL

CPU CLOCK

F-9
PREPARATION OF THE ENVIRONMENT

ENABLE/DISABLE READY COMMAND

ENABLE RDY

ICE-88 READY
USER READY

CPU READY

DISABLE RDY

ICE-88 READY
USER READY

CPU READY

LOADING A PROGRAM

BEFORE LOADING THE PROGRAM, THE PREPARATION OF THE EXECUTION ENVIRONMENT MUST BE COMPLETED.

* CLOCK=EXTERNAL

SELECT USER CLOCK FOR USE

BY THE EMULATING PROCESSOR.

* ENABLE RDY

ENABLE USER READY FOR USE

BY THE EMULATING PROCESSOR.

WITH THE EXECUTION ENVIRONMENT NOW PREPARED, THE PROGRAM CAN BE LOADED.

* LOAD :F1:LAB1

LOAD AN ABSOLUTE OBJECT

FILE
ICE-86 PROGRAM

**GO EMULATION** -

* Full speed, or near full speed, program execution.

* During emulation, although ICE monitors program execution, the user has no interaction with the system until a halt in emulation occurs.

* A halt in emulation can occur through a user defined hardware breakpoint, or by depressing the escape (ESC) key on the console keyboard.

* After a halt in emulation, the user may interrogate the current state of the system, view information collected during emulation, and/or change the state of the system.

EX.

* GO FROM .START

---

ICE-86 PROGRAM EXECUTION

**STEP EMULATION** -

* User program is executed by ICE, one instruction at a time.

* During step emulation, effective program execution speed is much slower than that of go emulation.

* Step emulation permits interrogation and/or modification of the user system, after the execution of each instruction.

EX.

* STEP FROM .START

F-11
ICE-86 OPERATION

ICE-86 MONITORS THE BUSSES, (ADDRESS AND DATA CONTROL); EACH FRAME OF A BUS CYCLE IS MONITORED AND CAN BE SAVED.

8086 BUS CYCLE TRACING

FRAME 0

FRAME 1

ADO–AD15

DATA

BUS CYCLE

F-12
ICE-86 BREAKPOINTS

ICE-86 HAS TWO BREAKPOINT REGISTERS THAT MAY BE GIVEN VALUES THROUGH SOFTWARE COMMANDS.

FRAME INFORMATION

BREAKPOINT REGISTER CONTENTS

ICE-86 BREAKPOINTS ARE OF TWO TYPES:

EXECUTION
- TAKES INTO ACCOUNT THE QUEUE
- TRACKS INSTRUCTION THROUGH QUEUE
SYNTAX:
EXECUTED

NON-EXECUTION
- BASED ON BUS ACTIVITY ONLY
SYNTAX:
READ
WRITTEN
INPUT
OUTPUT
FETCHED
HALT
ACKNOWLEDGED
LOADING THE BREAKPOINT REGISTERS

GO FROM .START TILL .PORT2 OUTPUT OR .PARM1 READ

- BR0
- BR1

LOADING THE BREAKPOINT REGISTERS (CON'T.)

* BR0 = .PORT2 OUTPUT
* BR1 = .PARM1 READ
* GO FROM START TILL BR0 OR BR1
THE GO-REGISTER

THE GO-REGISTER (GR) IDENTIFIES THE BREAKPOINT REGISTERS TO BE USED FOR HALTING EMULATION.

* GO FROM .START TILL .PROC1 EXEC

OR

* BRO=.PROC1 EXEC

* GR=TILL BRO

* GO FROM .START

OR

* GR=TILL .PROC1 EXEC

* GO FROM .START

INTERROGATION MODE DISPLAY/CHANGE

<table>
<thead>
<tr>
<th>REGISTERS</th>
<th>FLAGS</th>
<th>PINS (READ ONLY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG</td>
<td>RF</td>
<td>HOLD</td>
</tr>
<tr>
<td>RBX</td>
<td>AFL</td>
<td>NMI</td>
</tr>
<tr>
<td>RAL</td>
<td>TFL</td>
<td>IR</td>
</tr>
<tr>
<td>SP</td>
<td>IFL</td>
<td>RDY</td>
</tr>
</tbody>
</table>

*REG
RAI=0000H RBX=0000H RCX=0000H RDI=0000H SP=0000H BP=0000H SI=0000H DI=0000H
CS=0000H DS=0000H SS=0000H ES=0000H RF=0000H IP=0000H

* RAI=5555
* RCX=FF
*
*REG
RAI=0000H RBX=0000H RCX=FF00H RDI=0000H SP=0000H BP=0000H SI=0000H DI=0000H
CS=0000H DS=0000H SS=0000H ES=0000H RF=0000H IP=0000H

* IFL=1
* RF
RF=0000H
* HOLD
HOL=*
**INTERROGATION MODE (CONT.)**

**ACCESSING MEMORY AND I/O**

<table>
<thead>
<tr>
<th>ASH . START LEN 2#</th>
<th>ADDR</th>
<th>PREFIX</th>
<th>MNEMONIC</th>
<th>OPERANDS</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0210000LH</td>
<td>#0210000LH</td>
<td>MOV</td>
<td>DX:FFEAI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100013H</td>
<td>#02100013H</td>
<td>MOV</td>
<td>AL:00H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100013H</td>
<td>#02100013H</td>
<td>OUT</td>
<td>AL:00H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100016H</td>
<td>#02100016H</td>
<td>MOV</td>
<td>DX:AL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100016H</td>
<td>#02100016H</td>
<td>OUT</td>
<td>DX:AL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100016H</td>
<td>#02100016H</td>
<td>CALL</td>
<td>0000EH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100016H</td>
<td>#02100016H</td>
<td>CALL</td>
<td>0007CH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100016H</td>
<td>#02100016H</td>
<td>MOV</td>
<td>WORD PTR (#0200H):0000H</td>
<td>SHORT</td>
<td></td>
</tr>
<tr>
<td>#02100016H</td>
<td>#02100016H</td>
<td>PUSH</td>
<td>WORD PTR (#0200H):0000H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100017H</td>
<td>#02100017H</td>
<td>MOV</td>
<td>AL:00H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100017H</td>
<td>#02100017H</td>
<td>PUSH</td>
<td>AL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100018H</td>
<td>#02100018H</td>
<td>MOV</td>
<td>AL:01H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100018H</td>
<td>#02100018H</td>
<td>PUSH</td>
<td>AL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#02100019H</td>
<td>#02100019H</td>
<td>CALL</td>
<td>00007H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**INTERROGATION MODE (CON'T.)**

**CODE DISASSEMBLY**

F-16
TRACE DATA COLLECTION

BUS DATA

TRACE DATA

<table>
<thead>
<tr>
<th>ADDR/DATA</th>
<th>BHE</th>
<th>BUS STS</th>
<th>QSTS</th>
<th>QDEPTH</th>
<th>DMUX</th>
<th>MARK</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

* EACH FRAME OF TRACE DATA CONTAINS 32 BITS OF INFORMATION.
TRACE DATA BUFFER
2 FRAMES/MACHINE CYCLE - 511 CYCLE CAPACITY

<table>
<thead>
<tr>
<th>FRAME 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FRAME N</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR/DATA</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FRAME N+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR/DATA</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

CONTROLLING TRACE DATA COLLECTION

• **ENABLE TRACE**

  NOTE: BY DEFAULT THE TRACE IS INITIALLY TURNED ON.

• **DISABLE TRACE**

TRACE DATA CAN ALSO BE COLLECTED CONDITIONALLY
ICE-86 has two trace control registers that may be loaded by software commands.

Using the Trace Control Registers

- ONTRACE = DISPLAY_DATA fetched
- OFFTRACE = LIGHT_PORT output
- Enable trace conditionally now off or
- Enable trace conditionally now on

Trace control registers can only be loaded with non-execution match conditions.
DISPLAYING TRACE DATA

Set TRACE Display Mode Command

TRACE = [FRAME | INSTRUCTION]

Examples:

TRACE = FRAME
TRACE = INSTRUCTION

PRINT Command

1. PRINT ALL
2. PRINT [i:j:::-decimal]

Example:

PRINT
PRINT ALL
PRINT +8
PRINT 9
PRINT -10

EXAMPLES

<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>PREFIX</th>
<th>Mnemonic</th>
<th>OPERANDS</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x31</td>
<td>0x217</td>
<td>MOV</td>
<td>DX,FFF0H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x97</td>
<td></td>
<td></td>
<td>IN</td>
<td>AX:DX</td>
<td></td>
</tr>
<tr>
<td>0x200</td>
<td>0x62</td>
<td></td>
<td>NOT</td>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>0x31</td>
<td>0x210</td>
<td>MOV</td>
<td>DX,FFF8H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td></td>
<td>OUT</td>
<td>DX:AX</td>
<td></td>
</tr>
<tr>
<td>0x4F</td>
<td>0x00</td>
<td></td>
<td></td>
<td></td>
<td>HBEI</td>
</tr>
<tr>
<td>0x17</td>
<td>0x17</td>
<td></td>
<td></td>
<td></td>
<td>STS DSTS</td>
</tr>
<tr>
<td>0x18</td>
<td>0x224</td>
<td>JNZ</td>
<td>#F00H</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

F-20
TRACE BUFFER POINTER

NOTE:
THE PRINT COMMAND FUNCTIONS RELATIVE TO THE POINTER.

MOVE, OLDEST, and NEWEST Commands

MOVE [4 :: -decimal]
OLDEST
NEWEST

Example:

MOVE
MOVE + 0
MOVE -11
OLDEST
NEWEST

MISCELLANEOUS FEATURES AND COMMANDS

Set or Display Console Input Radix Commands

BUFFFX
BUFFFX = Y::Q::O::I::H

Examples:

BUFFFX
BUFFFX = Y

Set or Display Console Output Radix Commands

BASE
BASE = Y::Q::O::I::H::ASCII

Examples:

BASE
BASE = 0

* INITIAL RADIX IS HEX FOR BOTH INPUT AND OUTPUT.
EMULATION TIMER

2 MHz (500ns) CLOCK

GO

*EMULATION BREAK*

FROM clause
CS OR IP MODIFIED
ENABLE/DISABLE TRACE

EMULATION TIMER

HITIMER

START

STOP

RESET

LOAD Command

LOAD [drive:]filename
   NOCODE
   NOLINE
   HOBMCODE
   HOSYMBOL

Examples:
LOAD:F1:TEST,VRF
LOAD:F1:MAPROM,HOMLINE
LOAD:F2:COLUMN,ONE,NOCODE,HOLINE
LOAD:F3:NEWCODE,HOSYMBOL

SAVE Command

SAVE [drive:]filename 
   NOCODE, partition, partition 2
   NOSYMBOL
   NOLINE

Examples:
SAVE:F1:TEST
SAVE:F2:MAPROM TO-MF4,HOMLINE
SAVE:F2:COLUMN,ONE,HOSYMBOL
SAVE:F3:NEWSTR,HOCODE,HOLINE
SAVE:F1:TEST,F1 TO 26,F1 SUB:1 TO, SUB:F8

LIST Command

(a) LIST device:
(b) LIST [drive:]filename

Examples:
LIST C.P:
LIST:CO:
LIST:F1:ICEFILE
CLASS EXERCISE 6.1

SET UP THE ICE-86 COMMANDS TO DO THE FOLLOWING:

1. MAP LOGICAL MEMORY 0-32K INTO USER MEMORY
   *

2. SELECT THE USER CLOCK
   *

3. LOAD THE PROGRAM FILE :F1:DEMO
   *

4. EXAMINE THE SYMBOL TABLE
   *

5. BEGIN EMULATION AT .START AND CONTINUE UNTIL .L5 EXECUTED
   *

F-23
CLASS EXERCISE 6.1 (CON’T.)

6. EXAMINE THE REGISTERS

7. EXAMINE THE BYTE MEMORY LOCATION .XYZ

8. CONTINUE EMULATION UNTIL DATA IS INPUT FROM PORT 0F8H

9. EXAMINE THE CONTENTS OF THE TRACE BUFFER

10. SINGLE STEP THROUGH THE NEXT TWO INSTRUCTIONS

11. EXAMINE THE LAST 5 ENTRIES IN THE TRACE BUFFER

12. EXAMINE THE WORD LOCATION .ABC

13. CONTINUE EMULATION FOREVER

14. BREAK EMULATION

15. GO BACK TO ISIS-II
CLASS EXERCISE 6.1 (CON’T)

16. MATCH THE PCB WITH THE RELATIVE LOCATION IN WHICH IT SHOULD BE INSTALLED.

   TOP _______  A 86 CONTROLLER
   MIDDLE _______ B FM CONTROLLER PCB
   BOTTOM _______ C TRACE PCB

17. WHICH ICE86 PCB CONTAINS THE 8080 MICRO PROCESSOR?

   * ________________________________

WHERE TO FIND MORE INFORMATION...

ICE-86 MICROSYSTEM IN-CIRCUIT EMULATOR OPERATING INSTRUCTIONS
CHAPTER 1 - INTRODUCTION TO ICE-86
CHAPTER 2 - ICE-86 INSTALLATION PROCEDURES
GETTING STARTED WITH ICE-86

The purpose of this lab exercise is to use the commands of the In-Circuit Emulator presented in this appendix. With these commands, you will be able to load and debug programs that you have written. The items to be covered during this lab are as follows:

1. Preparation of the Execution Environment
2. Loading of an Executable Program File
3. GO or "Real-Time" Emulation
4. Implementing User Defined Breakpoints
5. Examining CPU Registers, Memory Locations, and I/O Ports
6. Collection and Display of Trace Information
7. Timing a Section of a Program

Before you get started, make sure that you are at a system which is properly configured. In order to perform this lab, you must be at a workstation which contains the following items:

A. SERIES III Development System
B. ICE 86 connected to an SDK 86

If you have any question or if your ICE unit is not attached to your SDK 86, ask your instructor for assistance. You will also need some software. If you do not have the ICE86 software, you should see your instructor.

Once you are situated at a properly configured workstation with the proper software, you must generate an absolute program file. For this lab, we are going to borrow a program that is already written and use it to create an absolute program file.

There is a file on the system disk which was prepared for this lab exercise. It is :F0:DEMO.A86. DEMO.A86 is a source file for a program which is written in 8086 assembly language. We will use this program in this lab to demonstrate the features of ICE86.

Copy the source file to your user disk. Once you have the file on your user disk, you must assemble the source file into an object module. Make sure you use the DEBUG option of the assembler. Also, get a hard copy of the list file to use during this lab session.

ICE-86 DEMO LAB
By the time it finishes, the assembler will give us a relocatable object module. Although the assembler produced a module which is in code that our CPU can execute, we can't do anything with it until we provide it with some absolute addresses. We can use LOC86 to do this for us. Enter the following command:

```
-RUN LOC86 :F1:DEMO.OBJ ADDRESSES(SEGMENTS(CODE(200H))>&<CR>
>>INITCODE(100H)
```

The "-" and ">>" are prompts from the system. Get a copy of the listing from the locator which is in the file DEMO.LST. First of all there should not be any errors listed. If there are, you should match the invocation line at the top of your listing with the command above to make sure you don't have a cockpit error. If you have an error on your listing and the invocation line was OK, then you should see your instructor.

This program, as you can see from the assembler listing, utilizes the LEDs and switches on your SDK 86. The Module is named ICE_DEMO.

Now let's look at the locate command we just entered. As you can see, we located our program by segments beginning at address 200H. Then we invoked something called INITCODE and gave it an address of 100H. At this time, take a look at your program listing. In particular look at the END statement. You will see that the END statement on this program is more extensive than you would think it needs to be. This END statement contains the initialization information for the segment registers used by this module. The assembler uses this information to create what it calls an initialization record. Now back to our locator and this INITCODE business. ICE-86 requires that the INITCODE control be used. The INITCODE control causes the locator (LOC86) to create a segment which will initialize the segment registers and pointer registers in our CPU when our program is loaded.

Once you have familiarized yourself with the program and the locate map, you are ready to start the ICE session. Make sure the ICE-86 System Software is in Drive 0 and enter the following command:

```
-ICE86
```

This will load the ICE software driver and invoke the ICE hardware. If the invocation is successful, ICE will return an asterisk "*" prompt character.
If you wish to make a record of this ICE session, type the following:

LIST :F1:ICE.LAB

This will copy everything that goes to the screen to a file on your user disk called ICE.LAB.

The first thing we must do is prepare the execution environment for ICE. This consists of mapping memory and making a clock selection.

Memory mapping is our way of informing ICE the memory it can use and where it is located. Since we will be executing out of memory on the SDK-86 board, we will map our memory requirements to the user system. To do this, enter the following command:

*MAP 0 LEN 2 = USER

This command identifies the first two 1K blocks in the 8086's logical address space as being located in the user system (00000H - 007FFH).

Next we must make a clock selection. We have a choice of using a clock supplied by ICE-86 hardware (internal) or one supplied by user hardware. Since we are executing out of user memory, it is necessary that we select the user clock. Enter the following:

*CLOCK = EXTERNAL

At this point, the execution environment has been prepared. So now we can go ahead and load our absolute object file.

*LOAD :F1:DEMO

Now that we have our program loaded into our system, let's take a look at the CPU registers to see where our CS and IP registers are pointing. Enter:

*REG

When we assembled our program we used a switch called DEBUG. At the time we said that this switch added the symbol table to our object module. If we want to see what symbols are available, we can enter:

*SYMBOLS

(Remember that you can use Ctrl-S to stop the display and Ctrl-Q to resume)
As you can see, this will give us a list of all the symbols associated with the module called "ICE_DEMO". Let's add a symbol to the table which will be equal to the address of the first instruction to be executed. We know that the CS:IP currently point to that instruction so let's enter:

*DEFINE .BEGIN = CS:IP

Now look at the symbol table again.

*SYMBOLS

As we can see we now have a new symbol called .BEGIN.

When you displayed the registers, you may have noticed that the CS and IP registers contain values of 0010H and 0006H. This translates to an absolute address of 00106H. But our program was located at an address of 200H. What is going on here? Well, remember that locate command? Remember something called INITCODE? Our locator created an absolute segment at the address we specified (100H) and our loader initialized our CPU so that it would execute this code. If you look at the map from the Locator, you may notice a segment was created called ??LOC86_INITCODE. Let's see what this code is. Enter:

*ASM .BEGIN LEN 19

This code is used to initialize our segment registers and the stack pointer from the information in our END statement. SS is loaded from CS:WORD PTR [0000]. To see what this value is, enter:

*WORD CS:Ø

Is this segment value the same as the one on your locate map?

You may also want to look at the value SP is loaded with and see if it agrees with the assembly listing and the value DS is initialized with and check it against the locate map. The final instruction is to do a FAR JMP to 0020:0000 which is where we told the locator to place our CODE SEGMENT.

We can begin executing our program by issuing the command:

*GO FROM .BEGIN FOREVER

We could have said simply GO FOREVER since CS:IP was pointing to .BEGIN anyway. The term FOREVER indicates that the program will continue executing with no breakpoints.
At this time, verify the operation of the program by placing the switches in various positions and monitoring the reaction of the LEDs with the program description in the listing file.

Now that we know the program executes properly, let's terminate its execution and look at some other ICE commands. To bring about a random breakpoint, the Escape key must be struck.

\(<\text{Esc}>\)

Notice the termination address is printed when emulation comes to a halt.

Now let's see how we can enter some breakpoints of our own. Suppose we wanted to restart this program, but this time we wanted to stop when the switches of port $0FF9H$ are in an illegal setting.

Before you enter the breakpoint, make sure that the command switches are in a legal configuration (refer to the listing). As you can see from the listing, the only time the instruction with the label ILLEGAL_CMD is executed is when an illegal command is decoded. We can set the breakpoint for that instruction by entering:

\[*GO FROM .START TILL .ILLEGAL_CMD EXECUTED\]

You can reference any symbol by referencing it as shown by this command. Notice the period "." before the symbol name. Also notice that we were very explicit in saying that we wanted to break emulation when that instruction was EXECUTED. If we were not explicit, we would break emulation when that instruction was fetched regardless of whether it was executed or not. This is important since our CPU has a pre-fetch queue and may fetch the instruction even though it might never execute it.

Your program should execute until you change the setting of the command switches to an illegal setting. When this happens and execution terminates, you can correlate the address at which the execution terminated as displayed on the screen with the address of ILLEGAL_CMD on the locate map. As you can see, the execution terminated with the CS:IP pointing to the instruction following the one we set our breakpoint at.
With the system halted there are a few thing you can look at. If you enter:

*PRINT -20

you can see what the last 20 instructions were executed before the breakpoint was encountered and what the illegal switch setting was that caused us to terminate.

If you prefer to see the information in each frame, enter:

*TRACE = FRAME
*PRINT -25

This will give you frame by frame information

If you enter:

*REG

you can examine all of the registers.

You may want to look at the Zero flag condition to see that it is cleared from the previous CMP by entering:

*ZFL

You can examine the controls of the memory location called .DISPLAY by entering:

*BYTE .DISPLAY

In response to this command, ICE 86 gives us the address of .DISPLAY and displays its contents.

Now change the settings of the command switches to a valid configuration and enter:

*GO

Once the program begins executing, change the switch settings to an illegal command setting. What happened?

If you notice, we didn't enter a TILL clause in our last GO command. As it turns out, ICE86 maintains breakpoints until they are cleared out. To verify this, enter:

*GR

This causes ICE 86 to display the contents of it GO REGISTER. As you can see, the GO REGISTER contains the
breakpoint BR0. How can you determine what BR0 contains?
You guessed it... type:

*BR0

If you compare this with your locate map, you should see
that the breakpoint was matched when the instruction
associated with the program label ILLEGAL_CMD was executed.
In order to get the program to execute continuously we have
to change the contents of the GO REG. We can do this two
ways. The first way is to do it implicitly by entering GO
FOREVER which sets the contents of the GO REG to FOREVER and
begins execution. The other way to do it is by explicitly
setting the GO REG to FOREVER by entering:

*GR=FOREVER

Before we execute the program again, let's conditionally
collect trace information for later display. In this
example, we would like to collect information from the time
the instruction at location .START is fetched until a value
is output to .DISPLAY_PORT. Enter the following:

*ONTRACE = .START Fetched
*OFFTRACE = .DISPLAY_PORT OUTPUT
*ENABLE TRACE CONDITIONALLY NOW OFF
*GO

Change the switch settings several times and then strike the
Escape key to abort the process. Now let's look at the
trace buffer to see what was collected. If you are still in
frame information mode enter:

*TRACE = INSTRUCTIONS

and then we will print the entire buffer by entering:

*PRINT ALL

If you wish to stop it at any time press the Escape key.

If you look at the assembler listing, you will notice a
delay was written in starting at the program label .DELAY.
Let's use the ICE-86 built in timer to time this delay and
see how long it takes to execute. Enter the following:

*GO FROM .DELAY TILL .START Fetched
Now we can look at the timer to see how long it took to execute this piece of our program. Enter

*HTIMER
*TIMER

HTIMER contains the most significant 16 bits of the timer and TIMER the least significant 16 bits of the timer. To find out how long this part of our program took to execute, we would have to multiply the HTIMER value by 65536 add the TIMER value and then multiply it by the timers clock period of 500 nsec. Since most of us don't like to do hexadecimal multiplication, we need these values in decimal. We can do this two ways. Enter:

*BASE = T
*HTIMER
*TIMER

This changed our output mode to base ten and displays all our values in decimal. Another method is to evaluate using the EVAL command. Enter:

*EVAL HTIMER
*EVAL TIMER

This displays these values in all the bases supported by ICE. To calculate how long this took we now have to take HTIMER and multiply it by 65536. The following chart may help.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>65536</td>
</tr>
<tr>
<td>2</td>
<td>65536 + 131072</td>
</tr>
<tr>
<td>3</td>
<td>65536 + 196608</td>
</tr>
<tr>
<td>4</td>
<td>65536 + 262144</td>
</tr>
<tr>
<td>5</td>
<td>65536 + 327680</td>
</tr>
<tr>
<td>6</td>
<td>65536 + 393216</td>
</tr>
<tr>
<td>7</td>
<td>65536 + 458752</td>
</tr>
<tr>
<td>8</td>
<td>65536 + 524288</td>
</tr>
<tr>
<td>9</td>
<td>65536 + 589824</td>
</tr>
<tr>
<td>10</td>
<td>65536 + 655360</td>
</tr>
</tbody>
</table>

We then add the TIMER value and multiply this by 500 nsec or .5 usec. You should get a result of approximately .5 seconds for this.
Now let's change the value of the delay by changing the
MOV BH,2 instruction at 20:39. Enter the following:

*BYTE CS:3A = 4
*ASM .DELAY TO .LP1
*GO FROM .DELAY TILL .START FETCHED

and check the timers. The delay should be approximately 1
second. You may want to change the LOOP count in the CX
register and try it again.

At this time, you should have a basic idea as to how ICE-86
will be used to execute and debug programs that you write.
By using the GO command with breakpoint, you can test and
verify logical portions of your program. Using the REG
command, you can verify the contents of the CPU registers
whenever emulation has been stopped. You can collect
information in a trace buffer and time sections of your
program.

Whenever emulation is terminated, you may interrogate or
modify the system. Using your system and documentation, you
may wish to experiment at this time with some of the
capabilities of ICE 86. Some of the features that you may
wish to try are to modify the contents of an I/O port or to
look at the switch settings.

When you are satisfied, you may exit ICE86 by entering:

*EXIT

This will cause the system to return to ISIS and close the
LIST file you created. You may want to view this file using
AEDIT or copy it to the printer.
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