iAPX 86,88,186 MICROPROCESSORS PART I

WORKSHOP NOTEBOOK

VERSION 2.0 JUNE 1984

Order No. 210976-002

iAPX 86,88,186 MICROPROCESSORS PART I

WORKSHOP NOTEBOOK

VERSION 2.0 JUNE 1984

Order No. 210976-002

TABLE OF CONTENTS

iAPX 86, 88, 186 MICROPROCESSORS
WORKSHOP SCHEDULE

	CHAPTER	Lab
	Day One	
1	THE iAPX 86 PRODUCT FAMILY	Lab 1 -
2	INTRODUCTION TO MICROPROCESSORS	Using the Series III
3	INTRODUCTION TO SEGMENTATION	Development System
4	INTRODUCTION TO THE iAPX 86, 88	
	INSTRUCTION SET	Optional AEDIT
5	MORE INSTRUCTIONS	Basic Lab
6	SOFTWARE DEVELOPMENT	
	Day Two	
7	ARITHMETIC, LOGICAL AND	Lab 2 -
	CONDITIONAL INSTRUCTIONS	Defining and
8	DEFINING AND ACCESSING DATA	Accessing Data
9	PROGRAM DEVELOPMENT	
10	BASIC CPU DESIGN AND TIMING	
	Day Three	
11	PROCEDURES	Lab 3 -
12	PROGRAMMING WITH MULTIPLE SEGMENTS	Using Procedures
13	INTERRUPTS	(Linking with PL/M),
14	MEMORY AND I/O INTERFACING	Multiple Segments,
		and Interrupts
	Day Four	
15	PROGRAMMING TECHNIQUES	Lab 4 -
16	MODULAR PROGRAMMING	Modular Programming
17	INTRODUCTION TO THE iAPX 186, 188	
	MICROPROCESSORS	Optional Lab -
	(optional) ICE 86	ICE Demo
	Day Five	
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

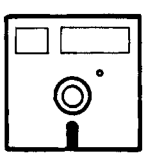
THE iAPX 86 PRODUCT FAMILY

- PRODUCTS
- NOMENCLATURE
- COURSE CONTENTS

GENERATIONS OF MICROPROCESSOR SYSTEMS

iAPX 86 PRODUCT FAMILY

SOFTWARE



HIGH LEVEL LANGUAGES

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

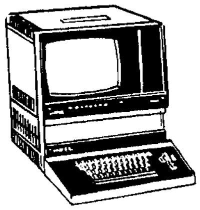
ASSEMBLY LANGUAGE

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)

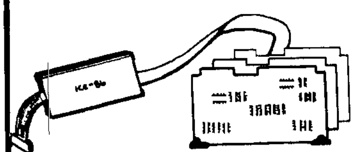
DEVELOPMENT SUPPORT



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

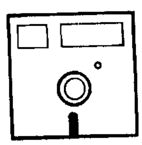
SERIES III DEVELOPMENT SYSTEM
(8086 AND 8085 PROCESSORS, FORTRAN 86, PLM86, ASM86, DEBUG-86, PASCAL 86, C86)

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)

I^2ICE
(INTEGRATED INSTRUMENTATION AND IN-CIRCUIT EMULATION SYSTEM FOR 8086, 80186, 80286, USED WITH SERIES III OR IV)



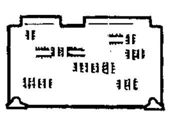
LINK86, LOC86, LIB86
(UTILITIES PROGRAMS THAT SUPPORT MODULAR PROGRAMMING, RUN ON SERIES II OR SERIES III)

iSBC 957B PACKAGE
(DOWNLOAD AND DEBUG FOR iSBC86 BOARDS)

iAPX 86 PRODUCT FAMILY

HARDWARE

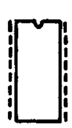
SINGLE BOARD COMPUTERS



iSBC 86/30 BOARD	(8MHz 8086, 128K RAM, FULL FUNCTION)
iSBC 86/12A BOARD	(5MHz 8086, 32K RAM , FULL FUNCTION)
iSBC 86/05 BOARD	(8MHz 8086, 86/12A COMPATIBLE , 8K RAM)
iSBC 88/40 BOARD	(5MHz 8088,ANALOG IO, PROCESS CONTROL)

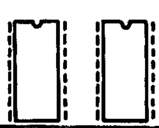
PLUS OVER 40 ADDITIONAL IO AND MEMORY EXPANSION BOARDS

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)

iAPX 86, iAPX 88 MODEL NUMBERS

iAPX 86

CPU	iAPX 86/10 8086
CPU & IOP	iAPX 86/11 8086 8089
CPU & 8087 NPX	iAPX 86/20 8086 8087
CPU & 8087 NPX & IOP	iAPX 86/21 8086 8087 8089
CPU 80130 OSP	iAPX 86/30 8086 80130

SIMILAR FOR

iAPX 88,
iAPX 186,
iAPX 188

iAPX 86 PRODUCT FAMILY

SOFTWARE

PLM 86	ASM 86 *	IRMX 86
PASCAL 86		IRMX 88
FORTRAN 86		IMMX 800

HARDWARE

ISBC 86/12A	iAPX 86*	8087*
iSBC 86/05*	iAPX 88*	8089*
ISBC 88/40	IAPX 186*	80130*
	iAPX 188*	
	IAPX 286*	

DEVELOPMENT SUPPORT

SERIES II *	ICE 86*	I²ICE
SERIES III*	LINK 86*	SDK 86
SERIES IV	LOC 86*	957 B
	LIB 86	

* = 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

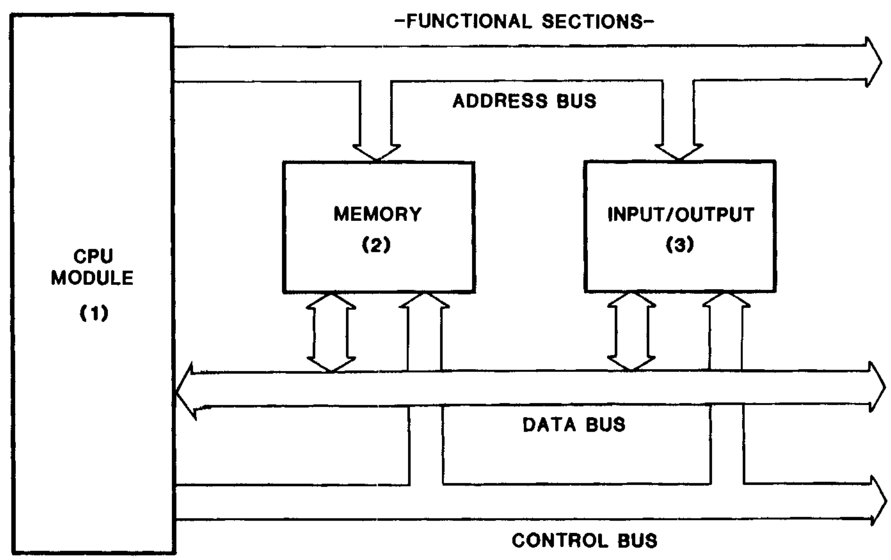
- INTEL WORKSHOPS CATALOG

CHAPTER 2

INTRODUCTION TO MICROPROCESSORS

- REGISTERS
- NUMBER SYSTEMS
- FLAGS

MICROCOMPUTER SYSTEM



1 **OPERATIONS DECISIONS**

2 **PROGRAMS, STACK, DATA**

3 **EXTERNAL COMMUNICATION**

2-1

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)

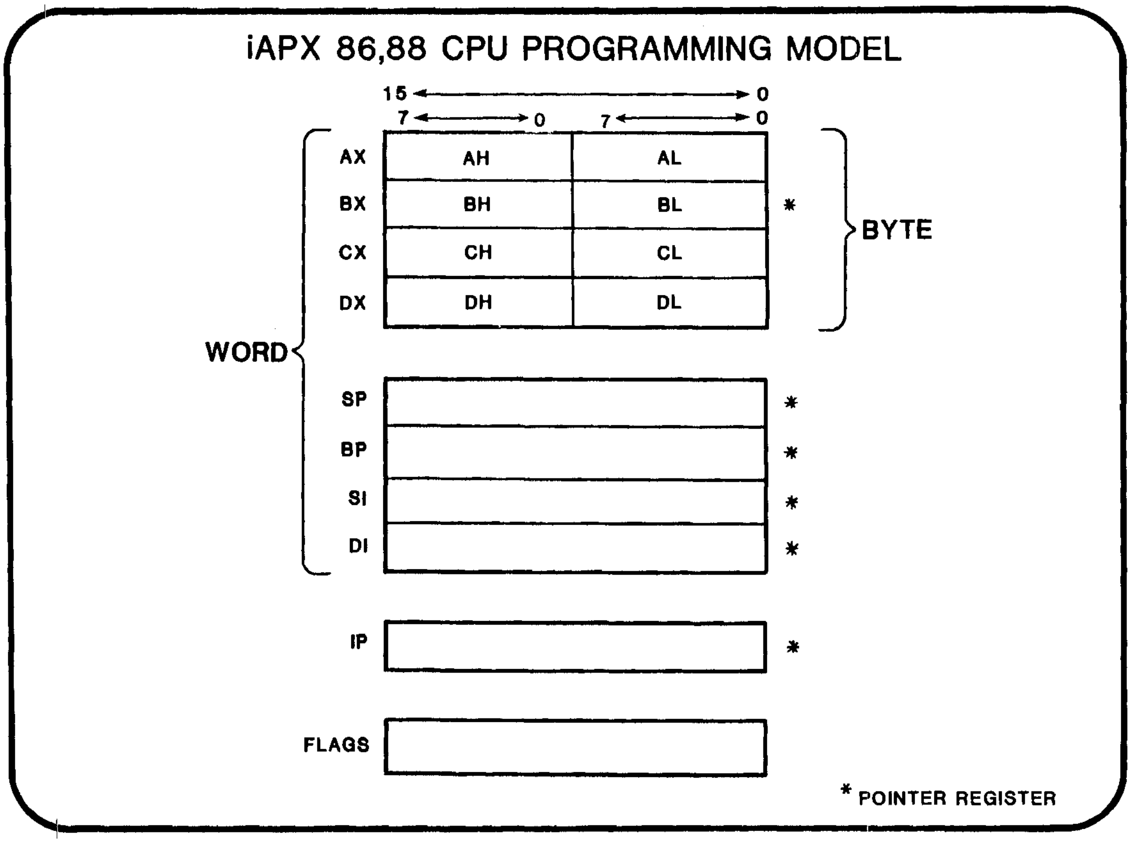
DATA BUS

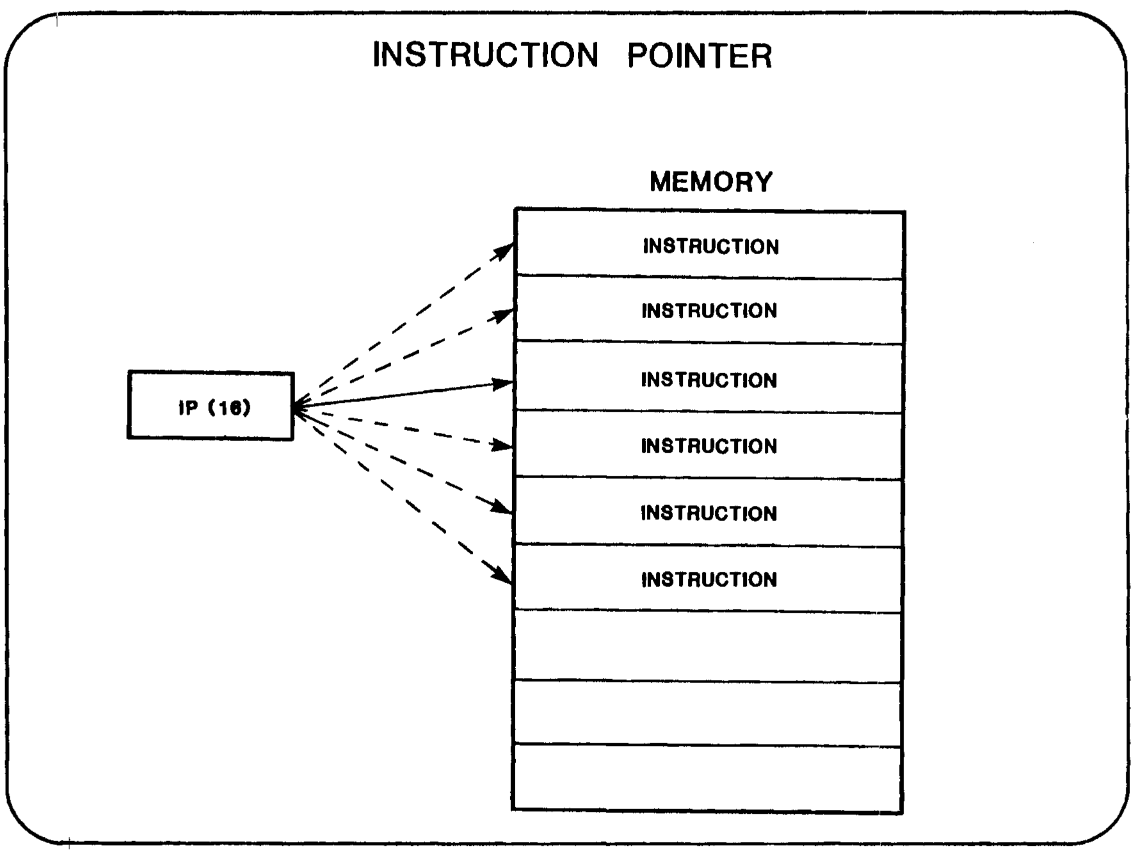
16 BITS BI-DIRECTIONAL (READ/WRITE)
THUS MEMORY AND I/O DATA WIDTH 8 OR 16 BITS

CONTROL BUS

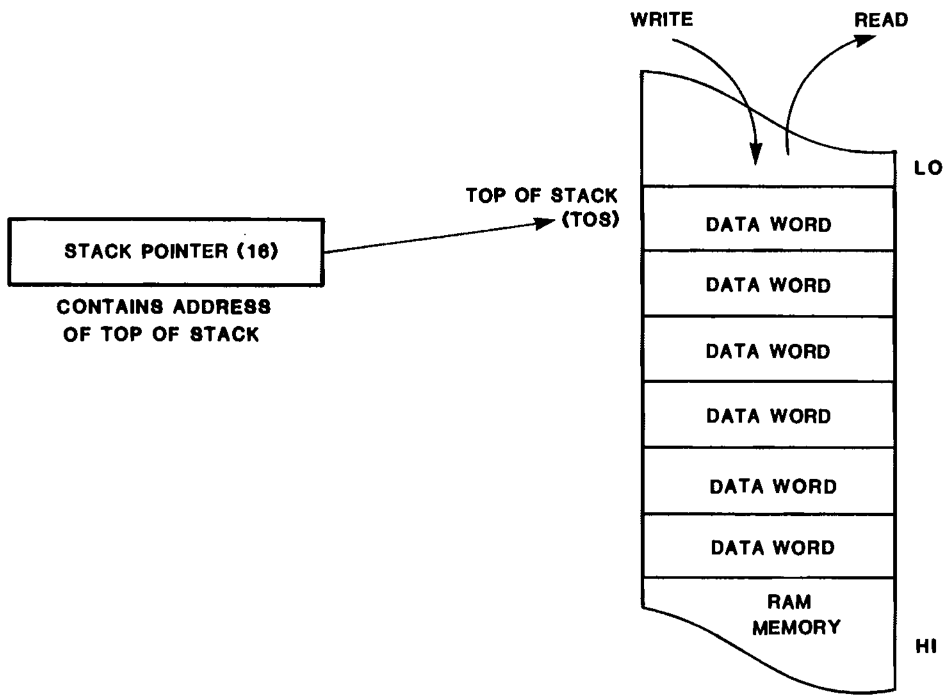
INCLUDES THREE CONTROL LINES

$M/\overline{IO}$ = I/O OR MEMORY SELECTOR
$\overline{RD}$ = READ
$\overline{WR}$ = WRITE

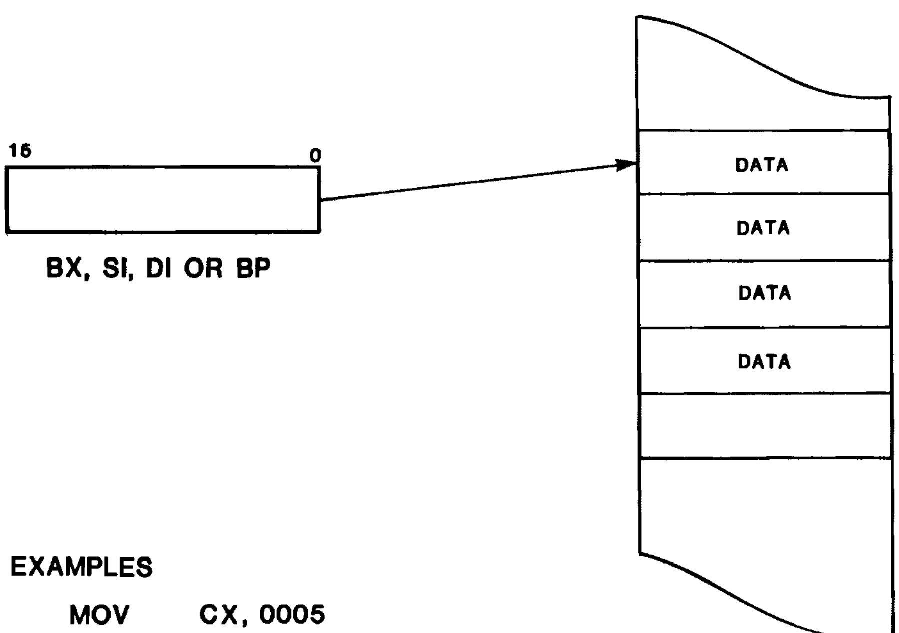

iAPX 86,88 CPU PROGRAMMING MODEL
15 0
7 0 7 0
AX AH AL
BX BH BL *
CX CH CL
DX DH DL
BYTE
WORD
SP *
BP *
SI *
DI *
IP *
FLAGS
* POINTER REGISTER


INSTRUCTION POINTER
MEMORY
IP (16)
INSTRUCTION
INSTRUCTION
INSTRUCTION
INSTRUCTION
INSTRUCTION
INSTRUCTION

STACK POINTER



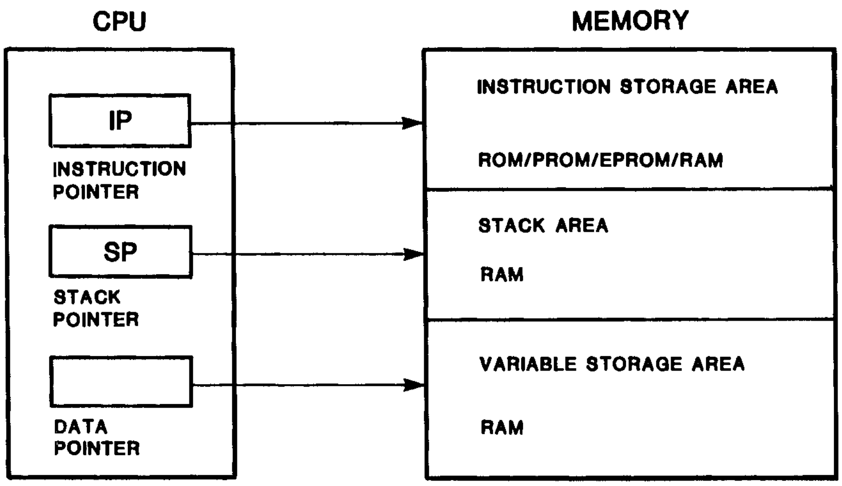
DATA POINTERS



EXAMPLES

```
MOV    CX, 0005
MOV    [BX], CX
MOV    AX, [SI]
```

TYPICAL MEMORY USAGE



2-7

NUMBER SYSTEMS

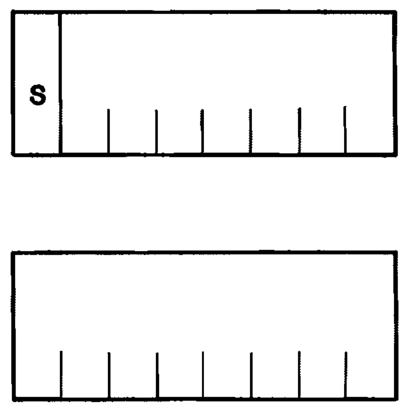
HEX	BINARY	DECIMAL
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

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**

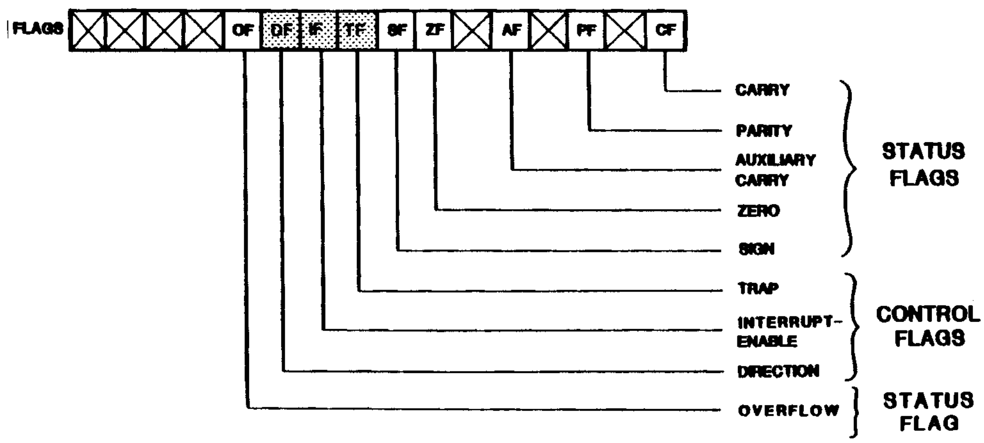
UNSIGNED: **0 − 255**

TWO'S COMPLEMENT NUMBER REPRESENTATION

EXAMPLE OF TWO'S COMPLEMEMT:

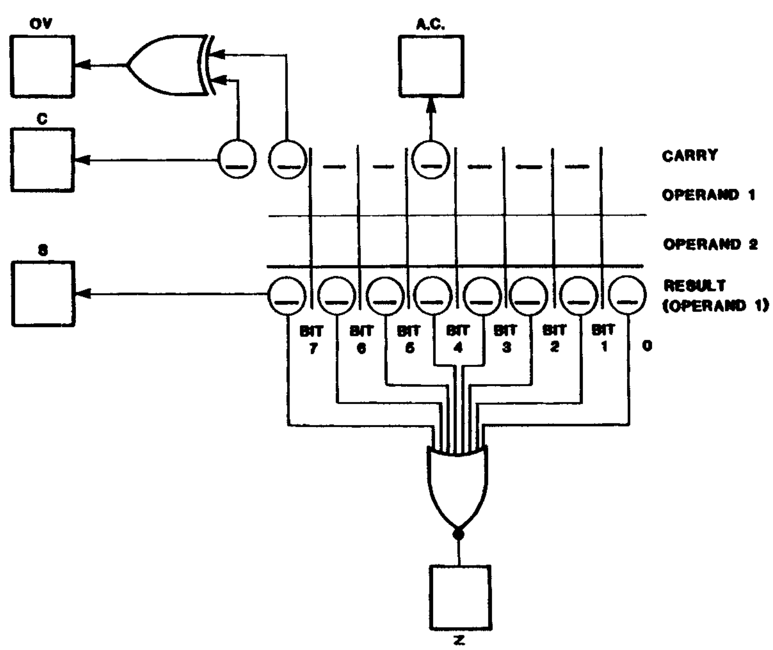
BINARY	DECIMAL
1000 0000	− 128
1000 0001	− 127
•	•
•	•
1111 1111	− 1
0000 0000	0
0000 0001	+ 1
•	•
•	•
0111 1110	+ 126
0111 1111	+ 127

FLAG WORD



2-11

FLAG OPERATIONS



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

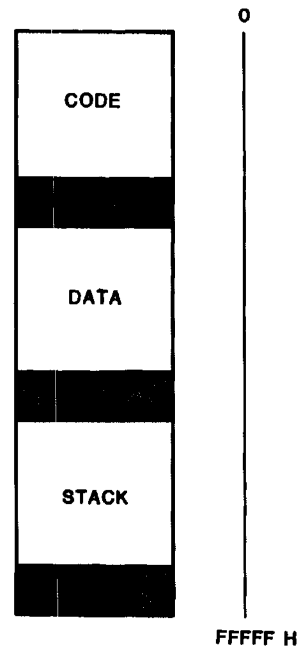
iAPX 86,88 MEMORY TERMINOLOGY

* MEMORY IS USED TO STORE THREE TYPES OF INFORMATION.

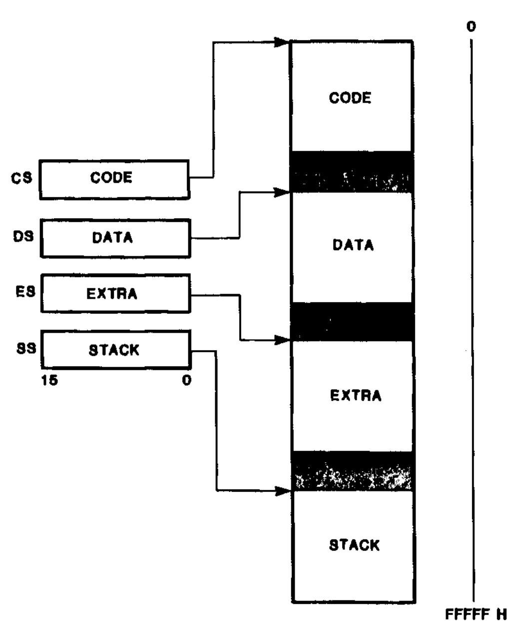
* THE 8086 VIEWS MEMORY AS A GROUP OF SEGMENTS.

* A SEGMENT IS A LOGICAL UNIT OF MEMORY.

* SEGMENTS CANNOT BE GREATER THAN 64K LONG.



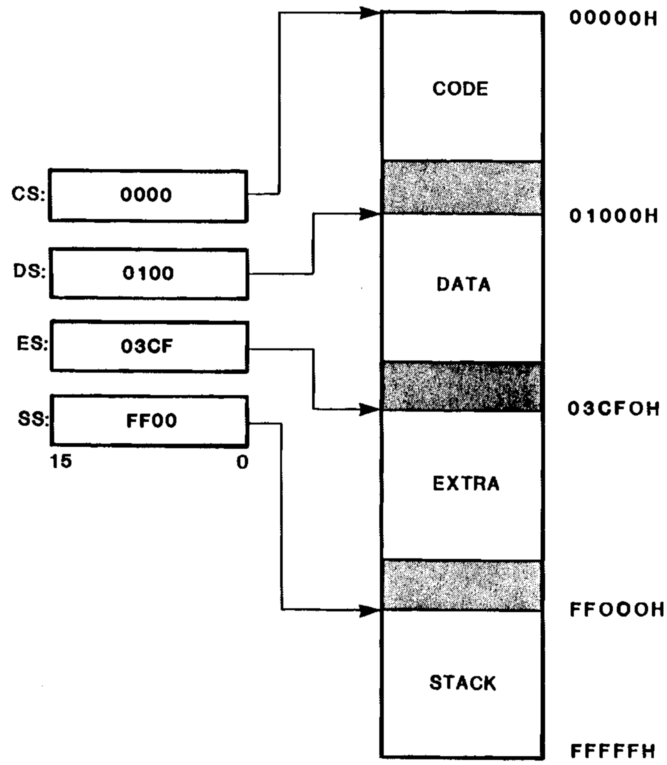
SEGMENT REGISTERS AND SEGMENTATION



* THE CPU HAS 4 SEGMENT REGISTERS.

* THE SEGMENT REGISTER POINTS TO THE BEGINNING OF A SEGMENT.

SEGMENT REGISTERS AND SEGMENTATION



SEGMENTATION

* SEGMENTED ADDRESSING HAS MANY ADVANTAGES OVER LINEAR ADDRESSING

  1) REGISTER SIZE

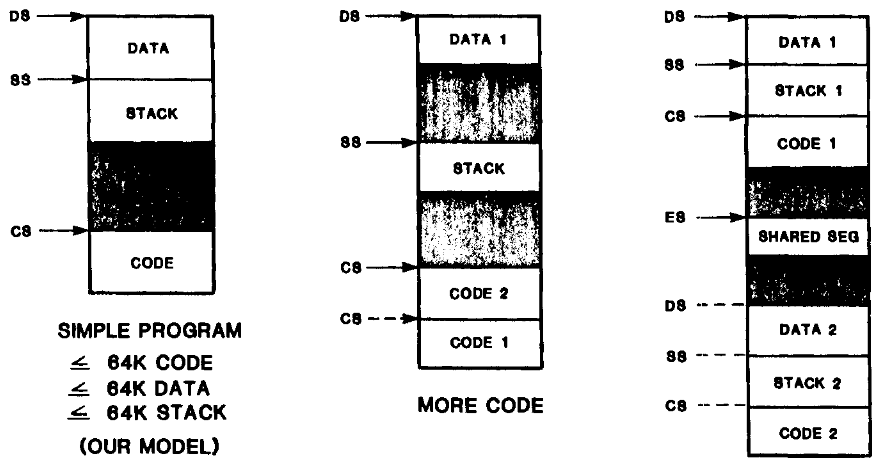
  2) DYNAMIC CODE RELOCATION

  3) MEMORY MANAGEMENT

* SEGMENTS ARE DEFINED BY APPLICATION

SEGMENTS ARE DEFINED BY APPLICATION

A FEW EXAMPLES



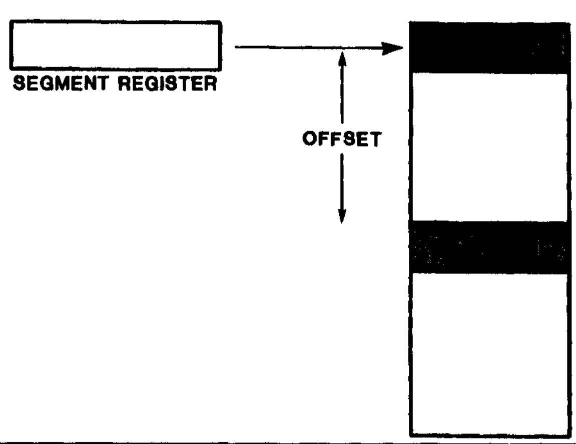
SIMPLE PROGRAM
≤ 64K CODE
≤ 64K DATA
≤ 64K STACK
(OUR MODEL)

MORE CODE

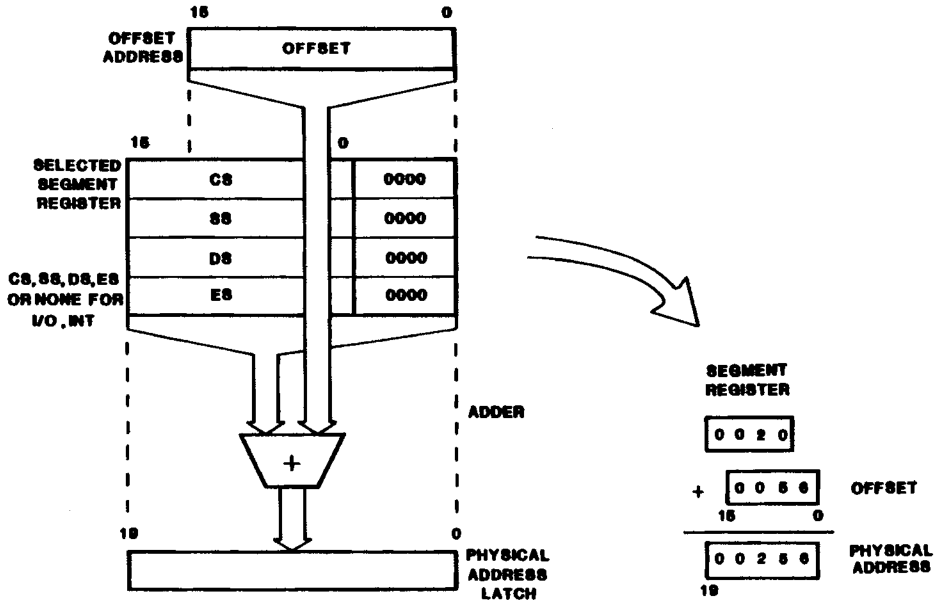
TWO PROGRAMS (TASKS) SHARING ONE PROCESSOR

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

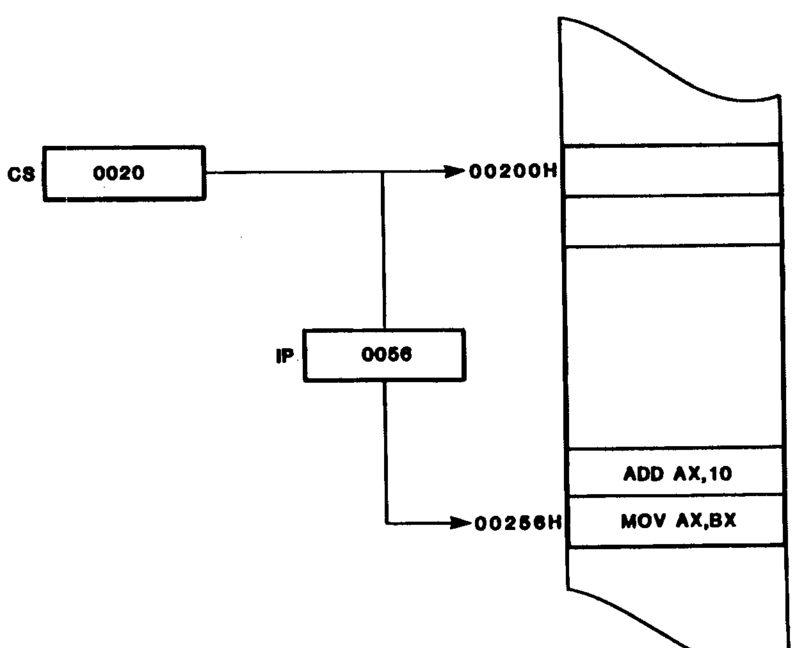


USING THE SEGMENT REGISTER CONTENTS



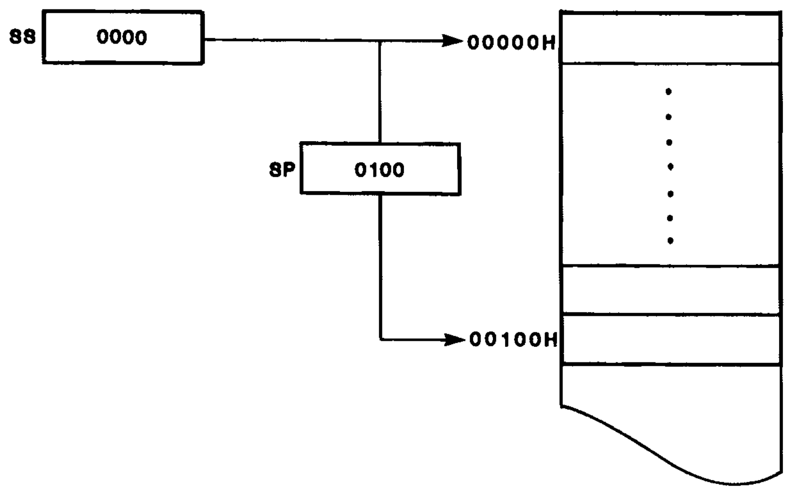
FETCHING INSTRUCTIONS

* INSTRUCTIONS ARE <u>ALWAYS</u> FETCHED WITH RESPECT TO THE CS REGISTER.



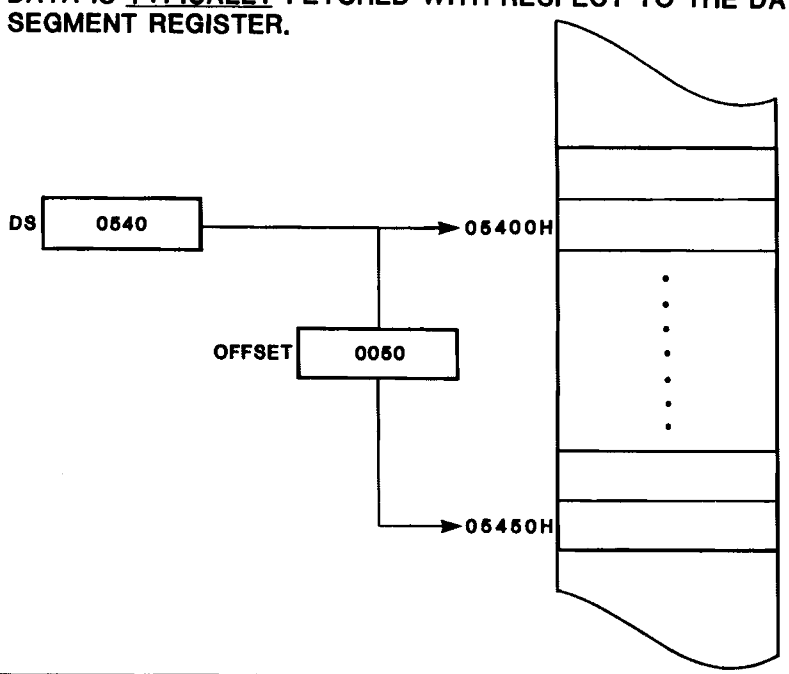
ACCESSING THE STACK

* 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.



CLASS EXERCISE 3.1

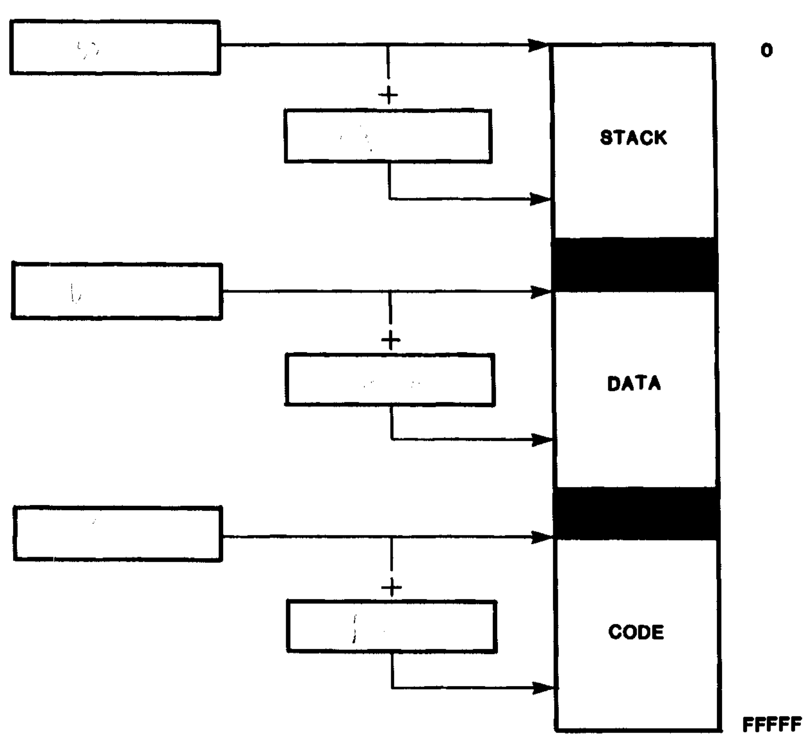
ASSUME AN INSTRUCTION IS LOCATED AT A PHYSICAL ADDRESS OF 05820H.

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?

REVIEW (FILL IN REGISTER NAMES)



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

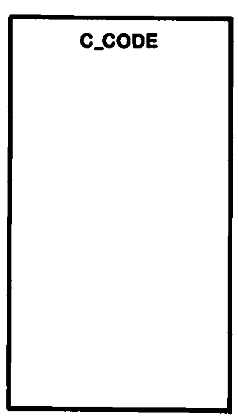
- CREATING A SEGMENT
- LABELS AND SYMBOLS
- ASSUME STATEMENT
- MOV,XCHG
- IN,OUT
- SHIFT,ROTATE

INSTRUCTIONS ARE CONTAINED IN SEGMENTS.

HOW DO YOU CREATE A SEGMENT ?

SEGMENT DECLARATIVE

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




```
C_CODE    SEGMENT
     |
     |
     |
C_CODE    ENDS
```

* 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	HEXIDECIMAL
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      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, [SI]
```

IMMEDIATE DATA

* MANY INSTRUCTIONS CAN USE IMMEDIATE DATA

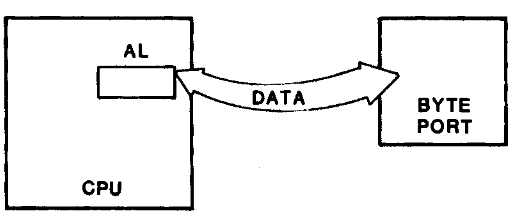
```
MOV   AX, 2345H
MOV   BL, 123D
```

* EQU STATEMENTS ARE USEFUL WITH IMMEDIATE DATA

```
DAYS_IN_YEAR   EQU 365
      ⋮
MOV       CX, DAYS_IN_YEAR
```

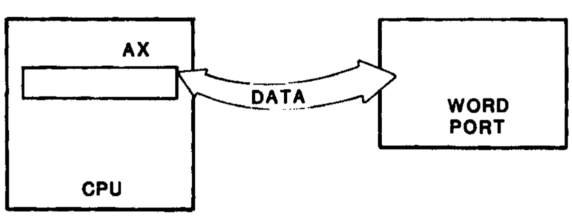
*EQU IS NOT AN INSTRUCTION AND DOES NOT ALLOCATE ANY MEMORY

IN, OUT



IN AL, PORT#

OUT PORT#, AL

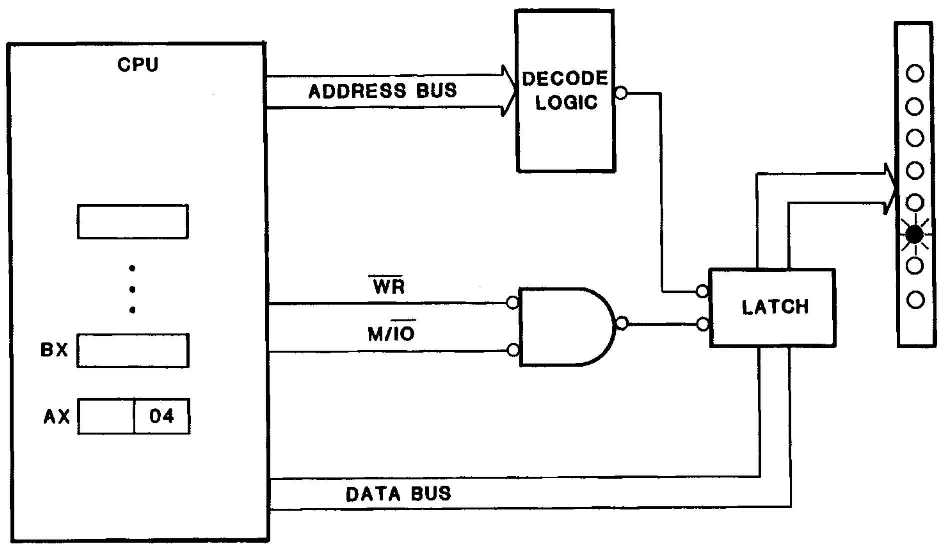


IN AX, PORT#

OUT PORT#, AX

PORT# = 0 TO 255

I/O OPERATION DIRECT PORT



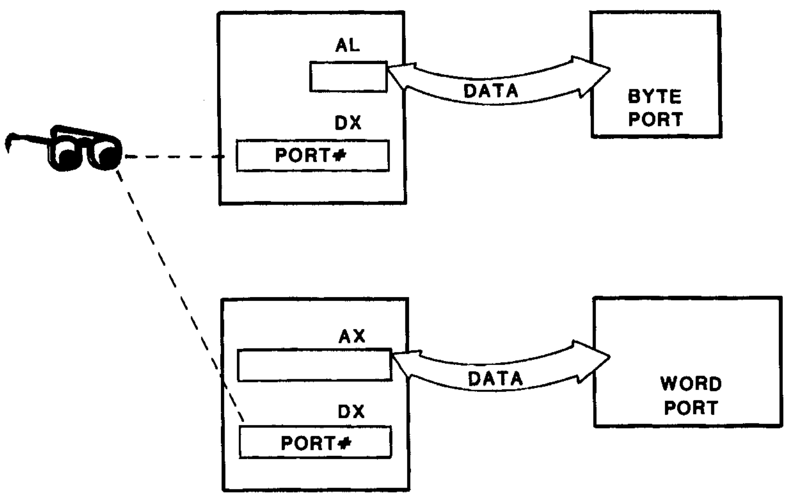

```
MOV    AL, 00000100B
OUT    20H, AL
```


ANOTHER WAY.....

OR

(HOW DO YOU GET 64K IO ADDRESSES)

IN, OUT



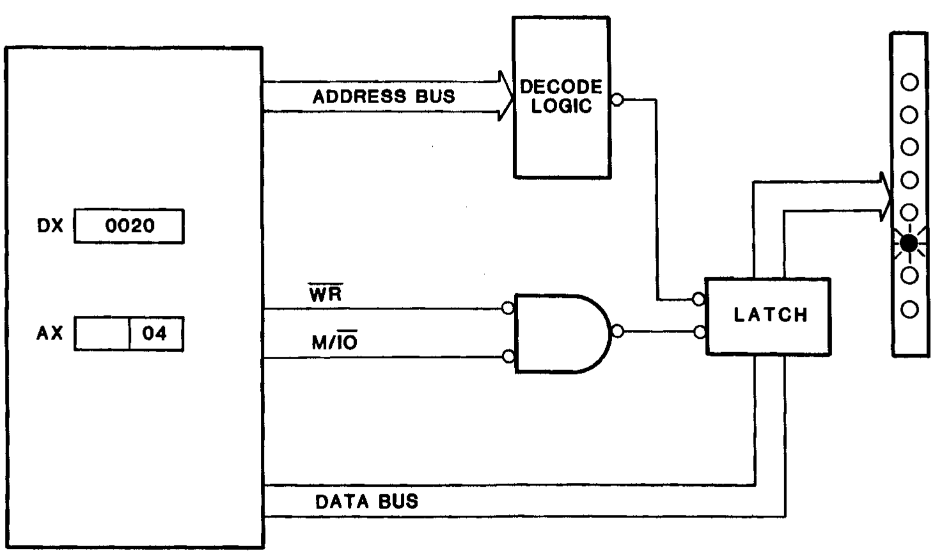
MOV DX, PORT#

IN AL, DX
OUT DX, AL

IN AX, DX
OUT DX, AX

PORT# 0 TO 65,535

I/O OPERATION
(INDIRECT PORT)

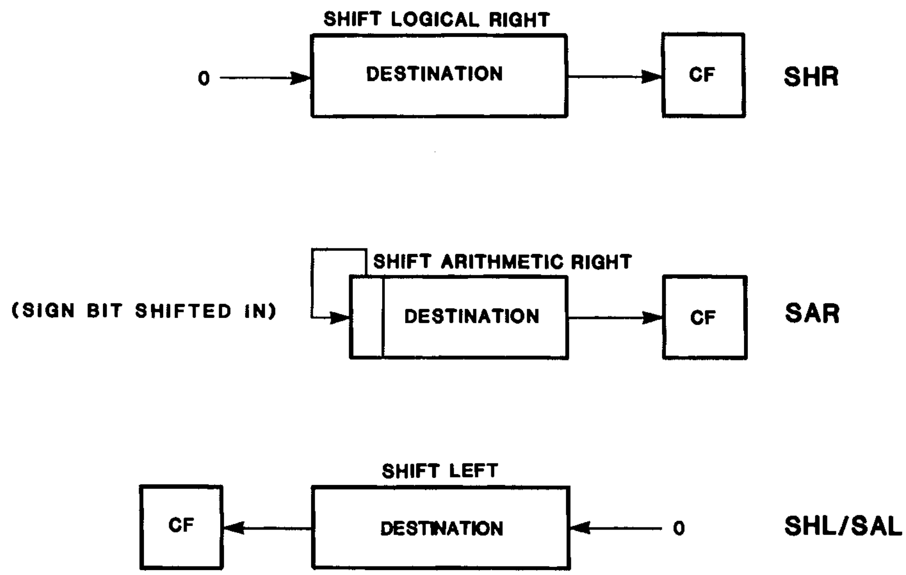


MOV AL,04H
MOV DX,20H
OUT DX,AL

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

SHIFT INSTRUCTIONS

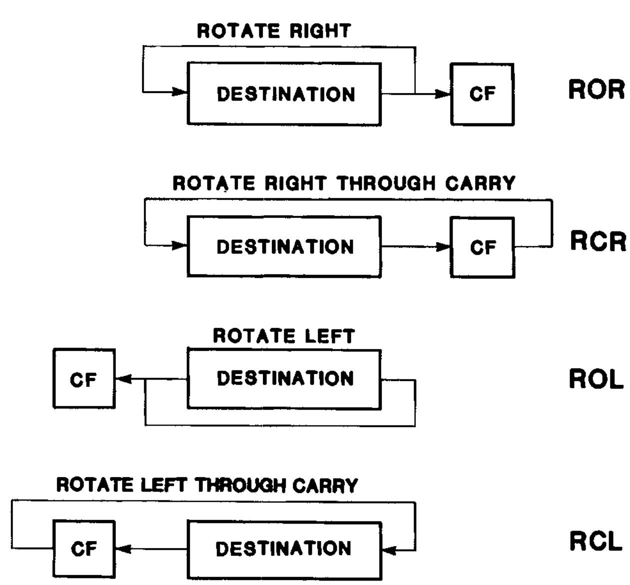
* ARITHMETIC SHIFTS CAN BE USED TO MULTIPLY OR DIVIDE NUMBERS BY POWERS OF TWO



ROTATE INSTRUCTIONS

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

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



SHIFT AND ROTATE FORMS

* TYPE OF OPERAND DETERMINES BYTE OR WORD
* SINGLE BIT FORM:

```
SHL     AX,1     :SHIFT LEFT LOGICAL
                 :ONE BIT

ROR     BL,1     :ROTATE RIGHT
```

* VARIABLE BIT FORM:

```
MOV     CL,4     :SET UP THE SHIFT
                 :COUNT

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 ØFFF8H, 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 8Ø86, AND THE DESIRE TO MINIMIZE CODE SIZE, INSTRUCTION ENCODING IS MORE DIFFICULT TO UNDERSTAND THAN IN MANY PREVIOUS 8-BIT MACHINES (SUCH AS THE 8Ø85). 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

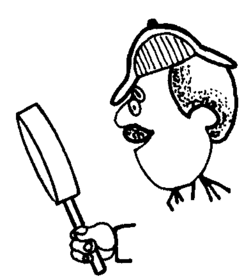
- HLT
- JMP
- LOOP

HLT INSTRUCTION

```
MY_SEG          SEGMENT
                 ASSUME CS: MY_SEG
                   -
                   -
                   -
                  HLT
MY_SEG           ENDS
```

JMP INSTRUCTION

```
MY_SEG          SEGMENT
                ASSUME CS:MY_SEG
START:
                  -
                  -
                  -
                JMP START
MY_SEG          ENDS
```



JMP INSTRUCTION

JMP ±128 BYTE DISPLACEMENT ("SHORT" JUMP, 2 BYTE INSTRUCTION)

JMP ±32K BYTE DISPLACEMENT ("NEAR" JUMP ,3 BYTE INSTRUCTION)

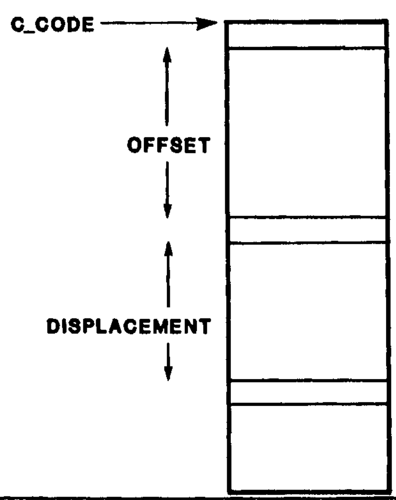
JMP ANY SEGMENT, ANY OFFSET ("FAR" JUMP , 5 BYTE INSTRUCTION)
(DISCUSSED LATER)

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.

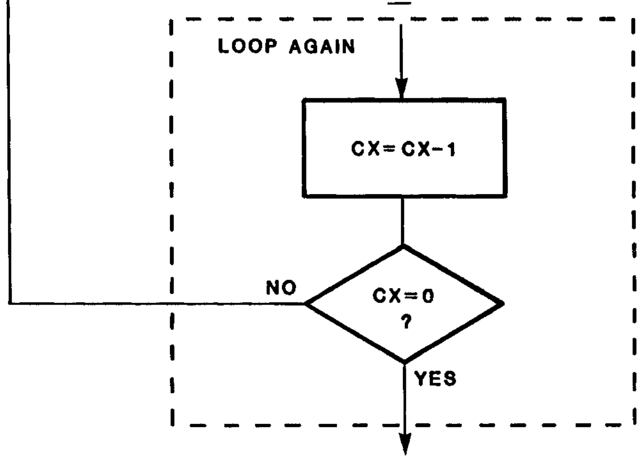
LOOP INSTRUCTION

A SPECIAL JUMP INSTRUCTION THAT DECREMENTS THE CX REGISTER AND JUMPS IF CX≠0

```
MY_SEG          SEGMENT
                  ASSUME CS: MY_SEG
START:            MOV CX,10
AGAIN:              -
                    -
                    -
                  LOOP AGAIN
                  HLT
MY_SEG            ENDS
```

LOOP INSTRUCTION

```
MY_SEG      SEGMENT
            ASSUME      CS;MY_SEG
START:      MOV         CX,10
AGAIN:      —
            —
            —
            LOOP AGAIN
            HLT
MY_SEG      ENDS
```



LOOP INSTRUCTION

ALSO USEFUL FOR DELAYS

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

HOW LONG WOULD THESE TAKE?

```
        :
        MOV CX,0FFFFH
SELF:   LOOP SELF
SELFZ:  LOOP SELFZ
        :
```

```
          :
          MOV CX,0FFFFH
OUTER :   MOV DX, CX
          MOV CX,0FFFFH
INNER:    LOOP INNER
          MOV CX,DX
          LOOP OUTER
          :
```

STOPPING THE ASSEMBLER

```
NAME          DEMO
MY_SEG        SEGMENT
              ASSUME CS: MY_SEG

START:        MOV CX,10    ;EXECUTE  PROGRAM
AGAIN:                     ;10 TIMES
                 -
                 -
                 -
                 -
                 -
                 -
                 -
              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. GROUP7
   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

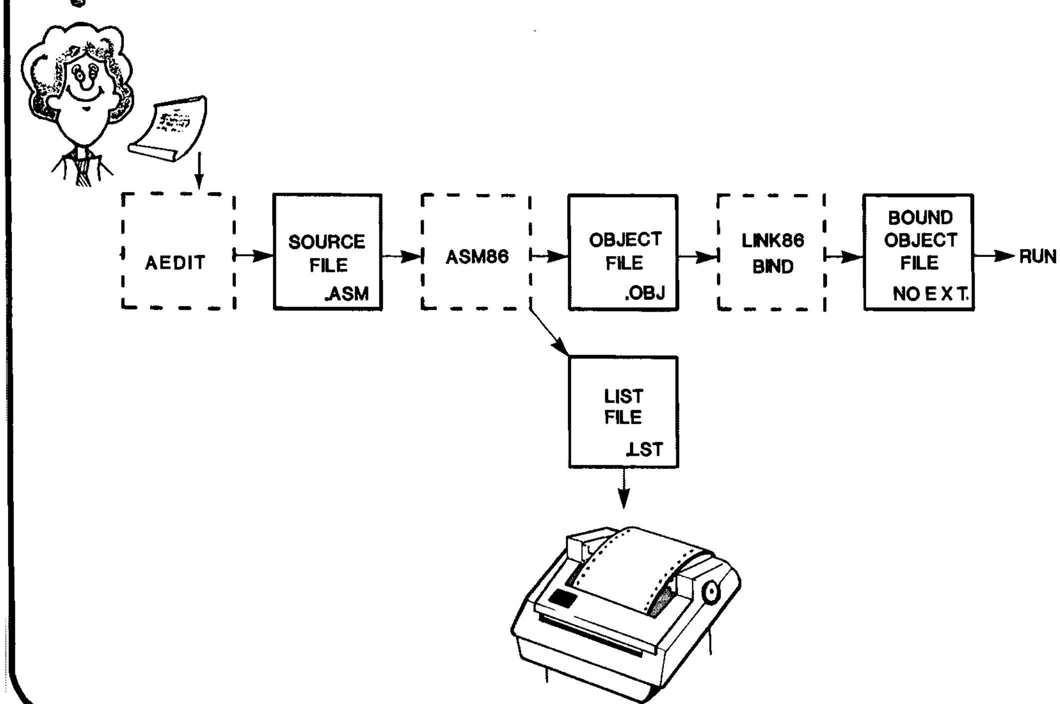
SEE CHAPTER 6, ASM86 LANGUAGE REFERENCE MANUAL

CHAPTER 6

SOFTWARE DEVELOPMENT

- SERIES III DEVELOPMENT SYSTEM
- FILE UTILITIES
- AEDIT TEXT EDITOR

SOFTWARE DEVELOPMENT
(SERIES I I I DEVELOPMENT SYSTEM)



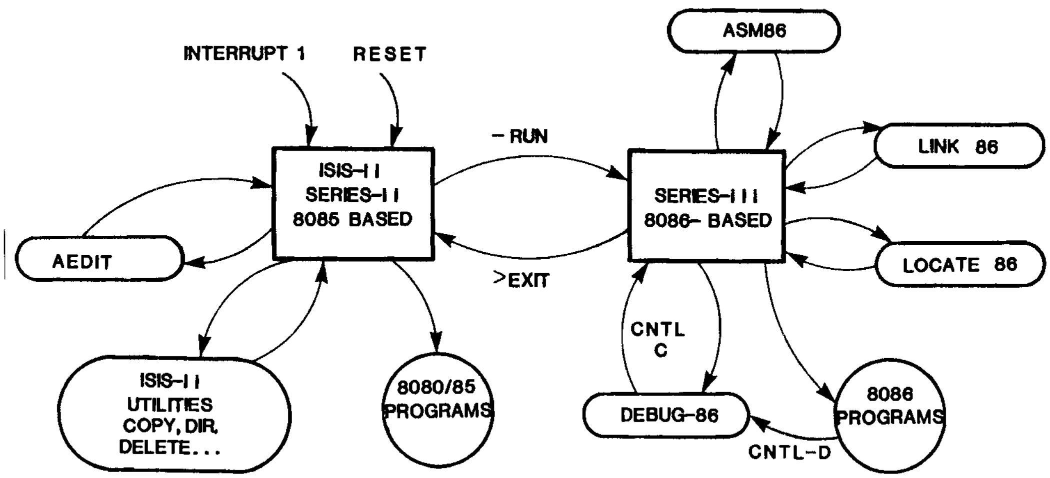
INITIALIZING ISIS-II

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 / 1] [I]

*** EXAMPLE**

DIR I

```
DIRECTORY OF :F0:86P1.002
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    6784   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    6815 WS
IDISK         63    7895 WS        RENAME        20    2346 WS
RUN          214   26804 WS        SUBMIT        39    4821 WS
AEDIT        214   26775 WS        ASM86 .86   1056  132988 WS
LINK86.86    608   76512 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         CO    .OBJ     6     561 W
RUN   .OVO    78    9724 W         AEDIT .MAC     2       5 WS
TEST  .LAB     3     212
                                   3290
3290/4004 BLOCKS USED
```

ISIS II NOTES

*** FILE NAME CONVENTIONS:**

:DEVICE:FILENAME.EXTENSION

- DEVICE: 2 CHARACTERS OPTIONAL
- FILENAME: 1 TO 6 CHARACTERS
- EXTENSION: 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

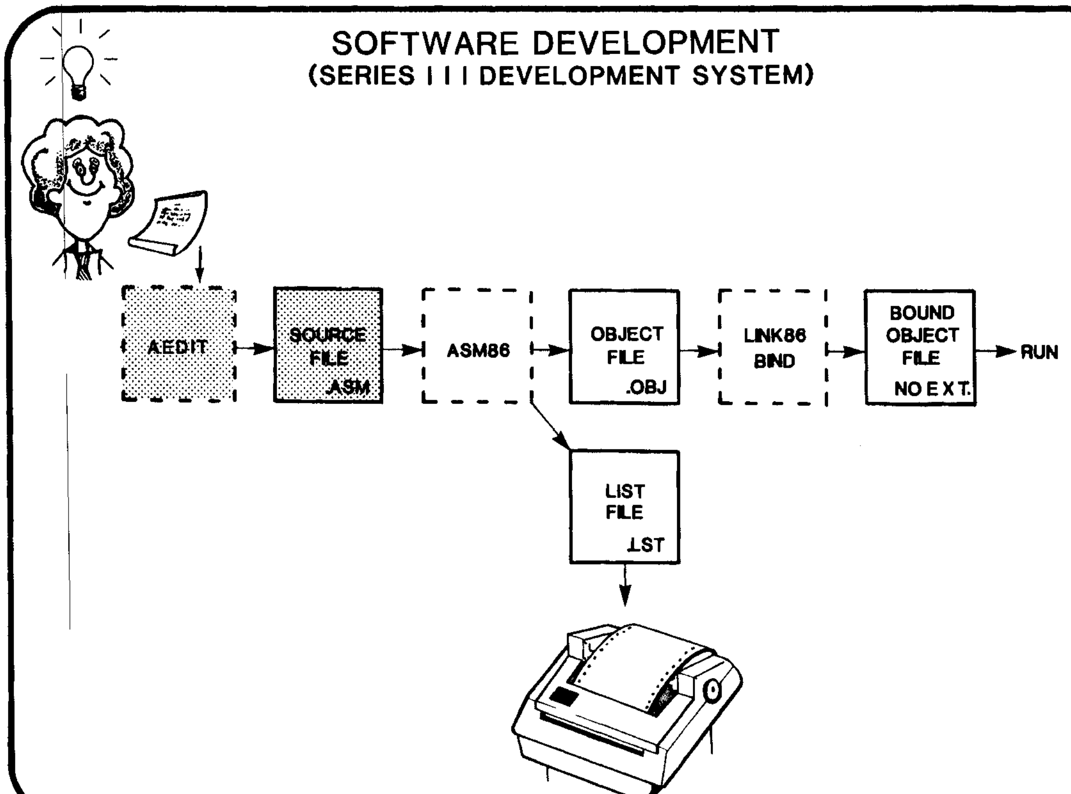
DELETE COMMAND

DELETES ISIS DISKETTE FILES FROM THE DIRECTORY

DELETE ISISFILENAME

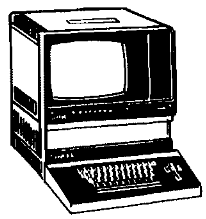
-DELETE LAB1.LST	DELETES LAB1.LST FILE FROM DISK IN DRIVE 0
-DELETE :F1:LAB1.LST	DELETES LAB1.LST FILE FROM DISK PRESENTLY IN DRIVE 1
-DELETE :F1:LAB?.LST	DELETES LAB1.LST, LAB2.LST, LAB3.LST, LABA.LST FROM DISK IN DRIVE 1
DELETE :F1:LAB1.*	DELETES LAB1.LST, LAB1.OBJ, LAB1.ASM FROM DISK IN DRIVE 1

SOFTWARE DEVELOPMENT
(SERIES I I I DEVELOPMENT SYSTEM)

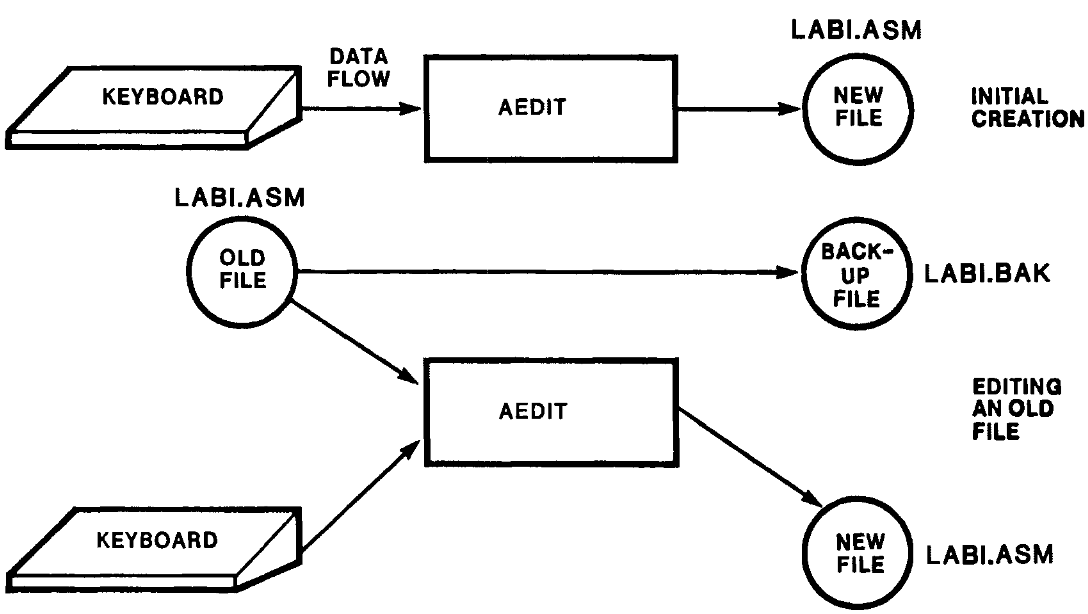


AEDIT

SERIES II/III/IV TEXT EDITOR



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

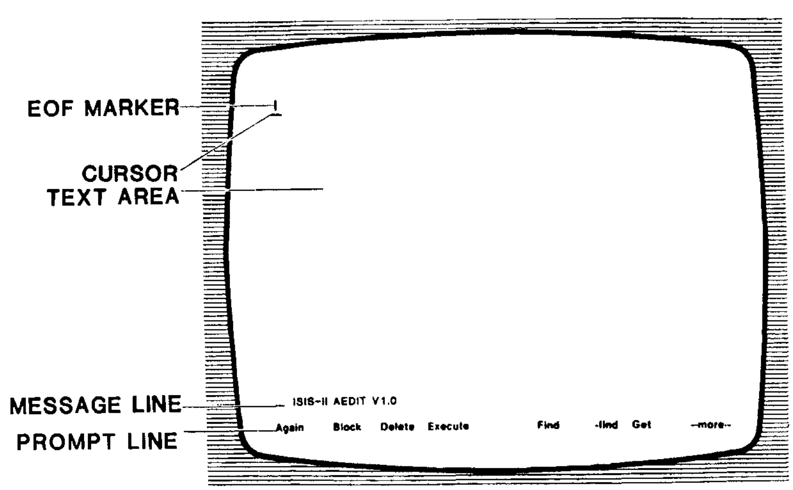
WHERE FILENAME IS THE NEW FILE TO BE CREATED OR AN EXISTING FILE TO BE UPDATED.

EXAMPLE :

-AEDIT :F1:LAB1.ASM

IS MENU DRIVEN

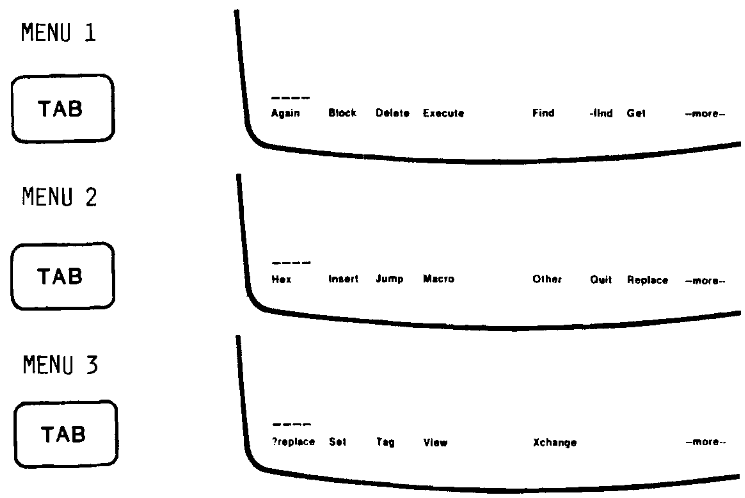
INITIAL SCREEN



- TO GET NEXT MENU:

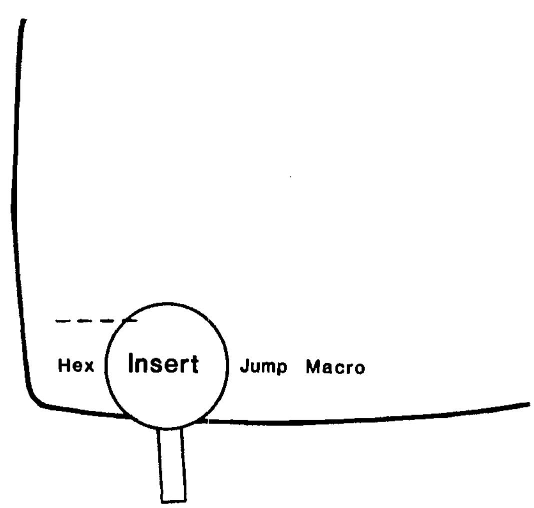
TAB

THE MENUS

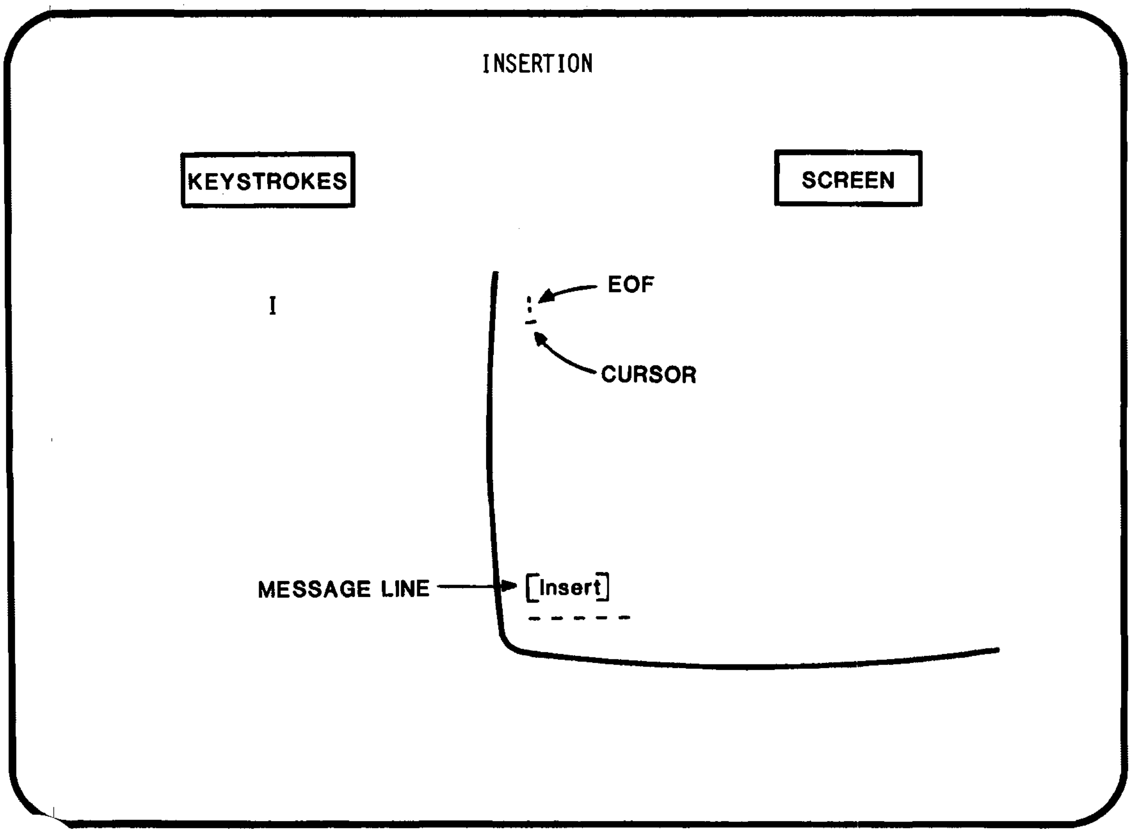


- TO INVOKE A COMMAND, KEY THE FIRST LETTER OF THE COMMAND.
- TO ABORT A COMMAND, TYPE CNTL-C.

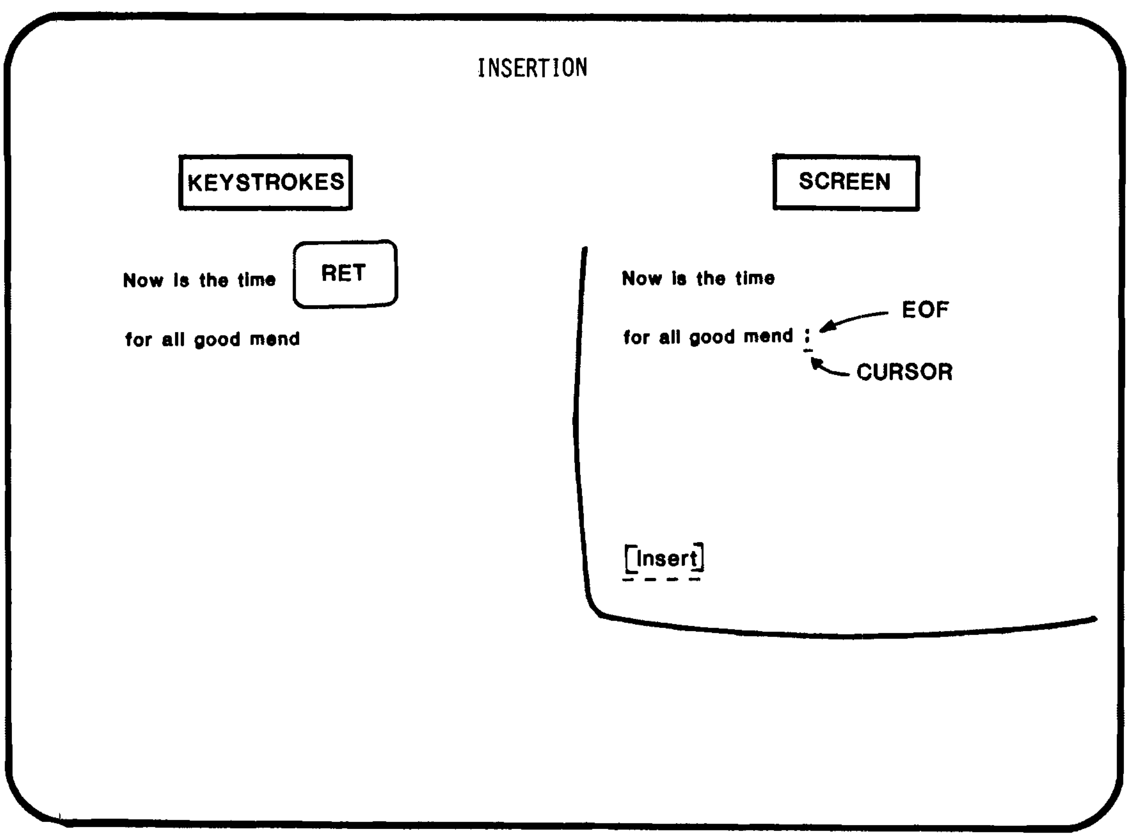
INSERTING NEW TEXT

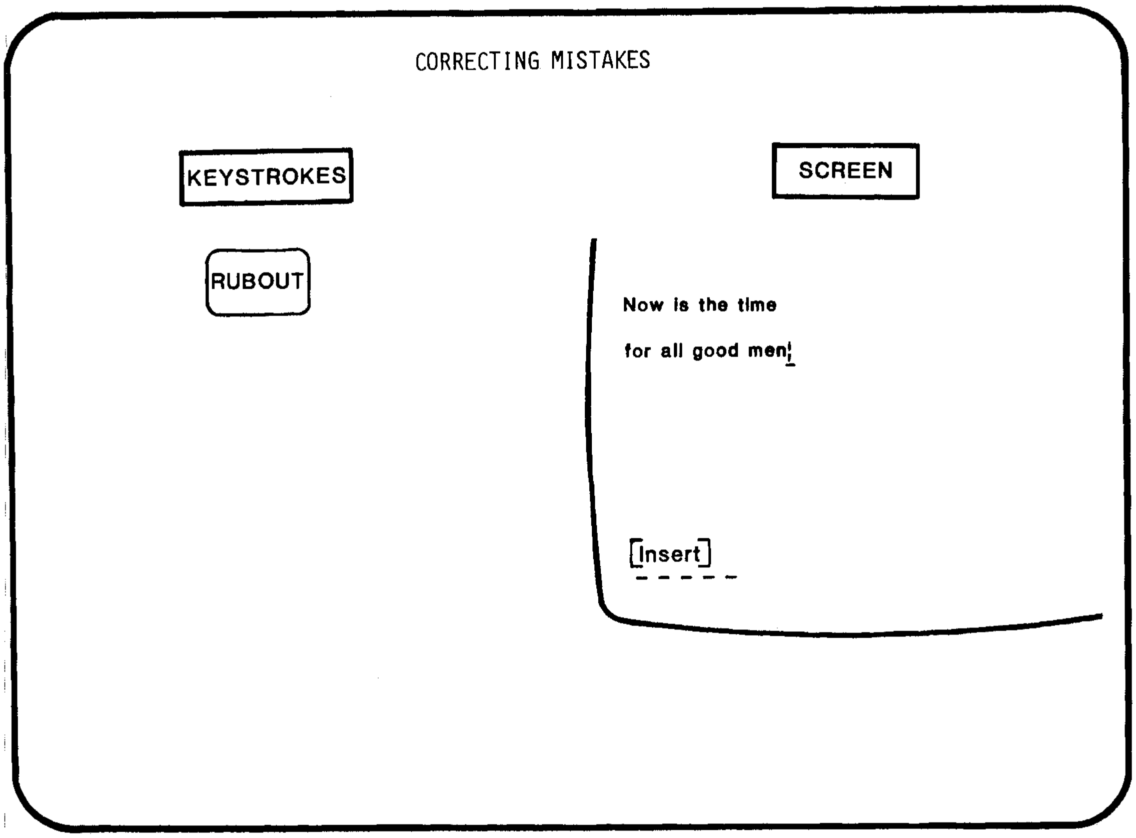


- TO INSERT TEXT, TYPE I

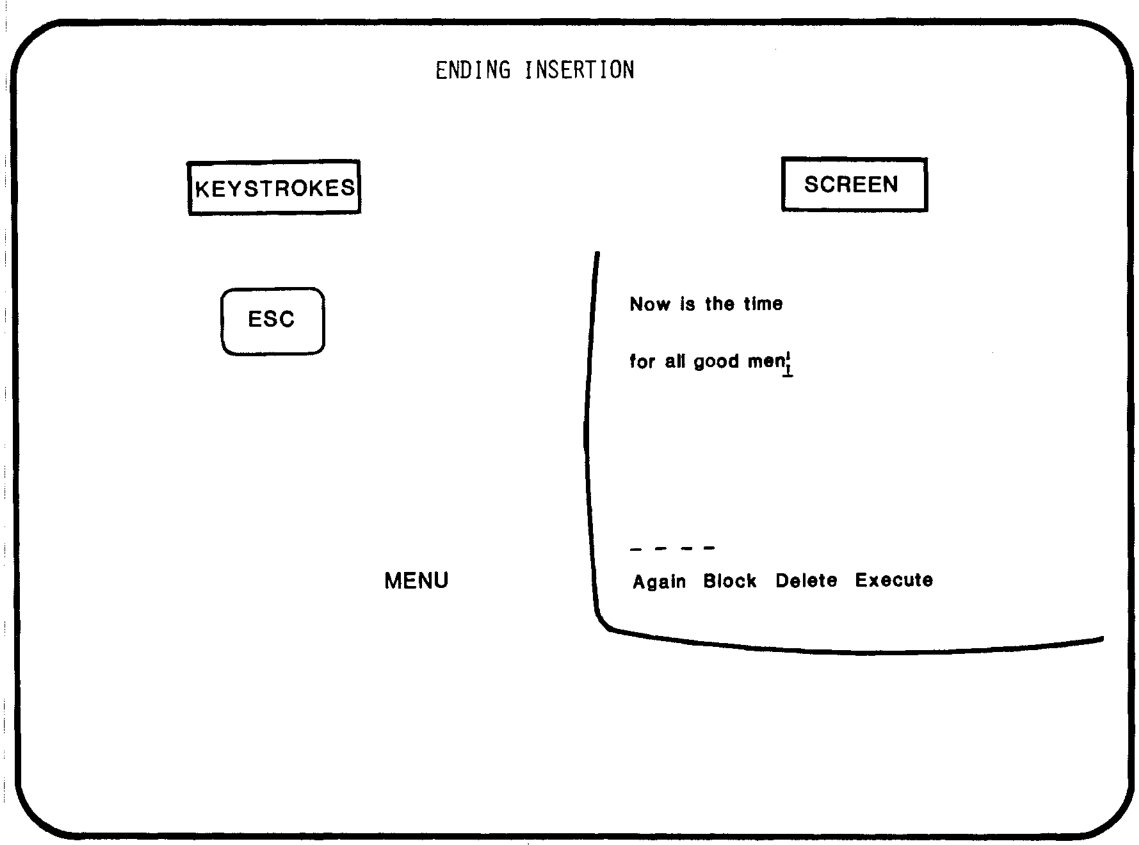

INSERTION
KEYSTROKES
SCREEN
I
EOF
CURSOR
MESSAGE LINE
[Insert]

6-15

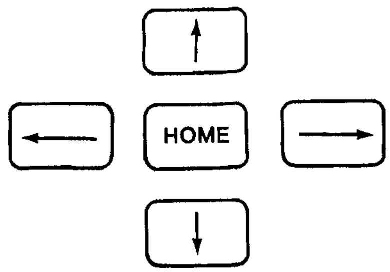

INSERTION
KEYSTROKES
SCREEN
Now is the time
RET
for all good mend
Now is the time
EOF
for all good mend
CURSOR
[Insert]


CORRECTING MISTAKES
KEYSTROKES
RUBOUT
SCREEN
Now is the time
for all good men
[Insert]

6-17

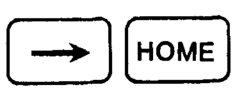

ENDING INSERTION
KEYSTROKES
ESC
MENU
SCREEN
Now is the time
for all good men
Again Block Delete Execute

CURSOR CONTROL

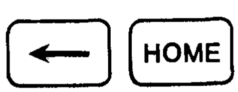


- ARROW KEYS MOVE CURSOR ONE SPACE OR LINE FOR EDITING

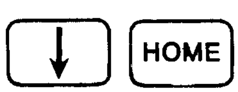
CURSOR MOVEMENT AND PAGING



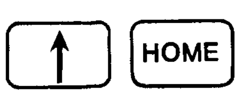
- MOVES CURSOR TO END OF LINE



- MOVES CURSOR TO BEGINNING OF LINE



- PAGES DOWN



- PAGES UP

DELETING TEXT

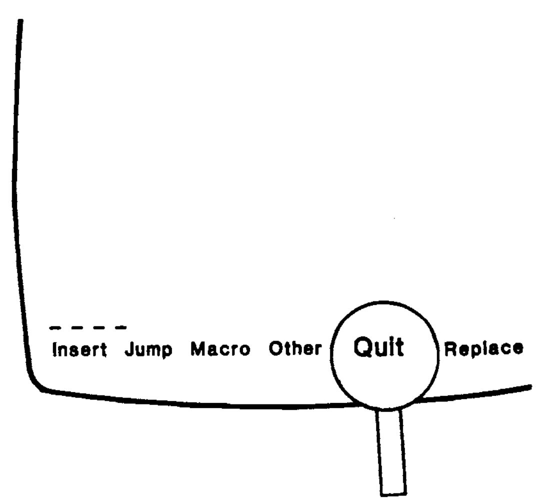
CONTROL F	DELETES CHARACTER AT CURSOR
CONTROL Z	DELETES LINE ON WHICH CURSOR IS POSITIONED
CONTROL U	UNDO-RESTORES DELETED CHARACTERS

THESE ALSO WORK DURING INSERTION

ENDING AN EDITING SESSION

KEYSTROKES

Q



QUIT

MENU PROMPT LINE

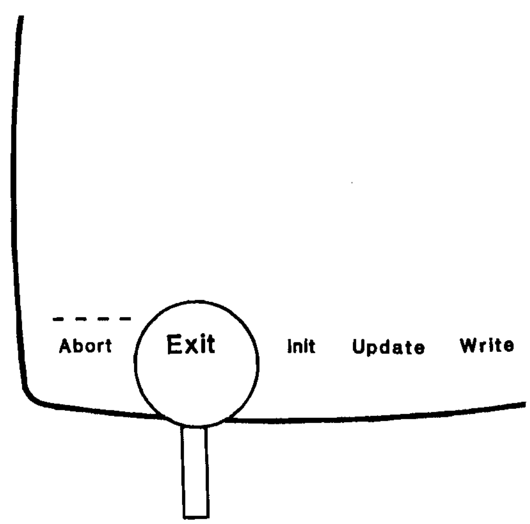
ABORT EXIT INIT UPDATE WRITE

SUBCOMMANDS:

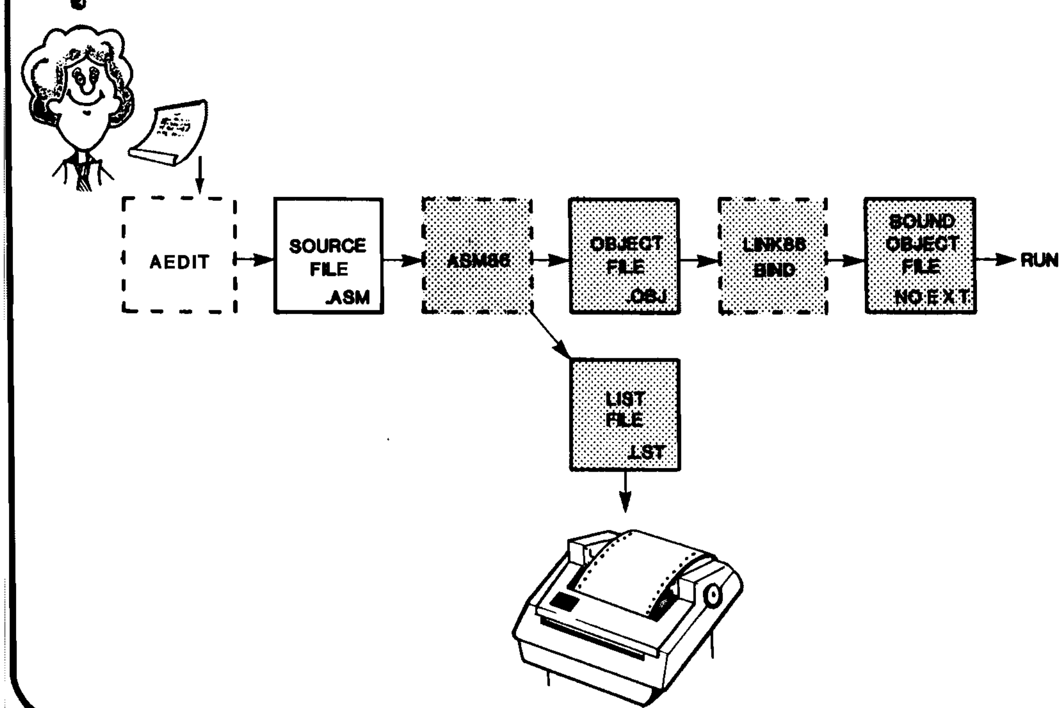
A - ABORT	ALL CHANGES LOST. RETURN TO OPERATING SYSTEM.
E - EXIT	FILE IS UPDATED. RETURN TO OPERATING SYSTEM
I - INIT	STARTS NEW EDITING SESSION. DOES NOT RETURN TO OPERATING SYSTEM.
U - UPDATE	UPDATES FILE. DOES NOT RETURN TO OPERATING SYSTEM.
W - WRITE	PROMPTS YOU FOR OUTPUT FILENAME. DOES NOT RETURN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

EXIT

KEYSTROKES



SOFTWARE DEVELOPMENT
(SERIES I I I DEVELOPMENT SYSTEM)



DEVELOPMENT STEPS

-AEDIT :F1:LAB 1.ASM	CREATES FILE
⋮	
-RUN ASM86 :F1:LAB1.ASM	ASSEMBLE FILE - CREATE .LST AND .OBJ FILE
-COPY :F1:LAB1.LST TO :LP:	PRINT .LST FILE LOOK AT ERRORS, IF ANY
-RUN LINK86 :F1:LAB1.OBJ BIND	CREATE "RUN TIME LOCATED" FILE
-RUN :F1:LAB1.	EXECUTE PROGRAM IN DEVELOPMENT SYSTEM

FOR MORE INFORMATION. . .

ISIS-I 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

	1001 1111	source	
AND	0000 1111	destination	
	0000 1111	destination	RESULT
	1001 1111	source	
OR	0000 1111	destination	
	1001 1111	destination	RESULT
	1001 1111	source	
XOR	0000 1111	destination	
	1001 0000	destination	RESULT
	1001 1111	source	
TEST	0000 1111	destination	
	NO CHANGE	destination	(LOGIC 'AND') NO REGISTERS CHANGED FLAGS REFLECT RESULT
NOT	(PRODUCES 1'S COMPLIMENT)		

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             ; TEST BIT 1 ('AND' CL WITH 00000010B)
JZ NOTSET
```

* 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

ADDING TWO 32-BIT NUMBERS

CY [0]	CY [1]	
	0010001101110011	1011101101100101
	0001001110001000	1110001100011100
	0011011011111100	1001111010000001

SUBTRACTION

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

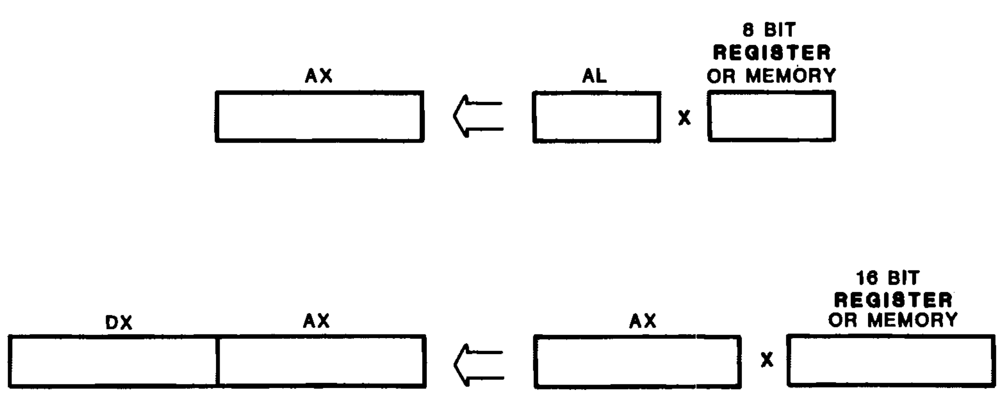
EXAMPLES

```
SUB   CL,20
DEC   DL
```

7-5

MULTIPLICATION

(ALWAYS USES ACCUMULATOR)



7-6

MULTIPLICATION

— UNSIGNED OPERATIONS

```
MUL     SOURCE
```

— SIGNED OPERATIONS

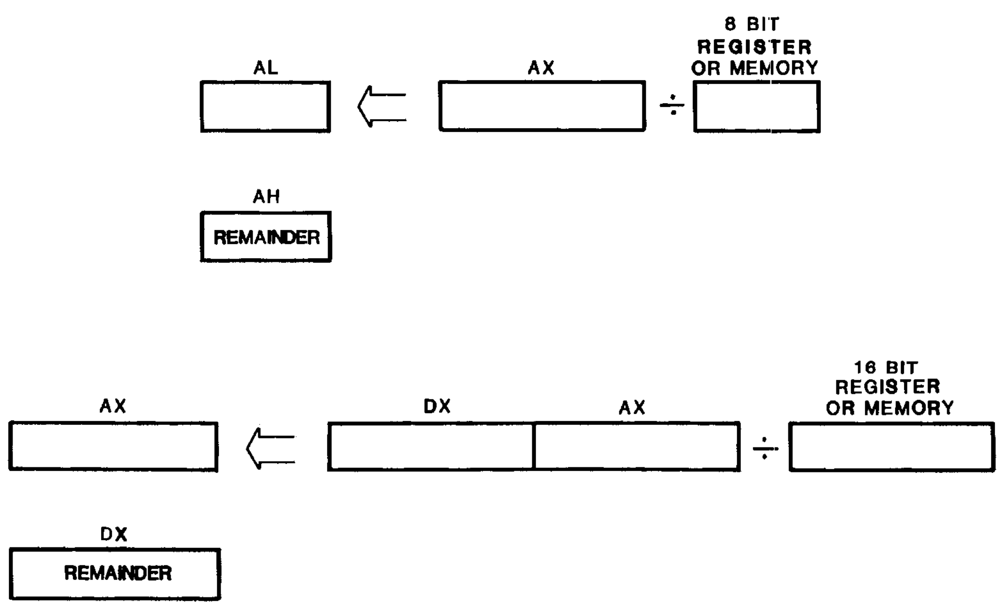
```
IMUL    SOURCE *
```

EXAMPLES:

```
MUL     BL          ;AX= AL * BL
IMUL    DX          ;DX,AX= AX * DX
```

* CAN BE IMMEDIATE DATA ON 186 BUT NOT 8086

DIVISION



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

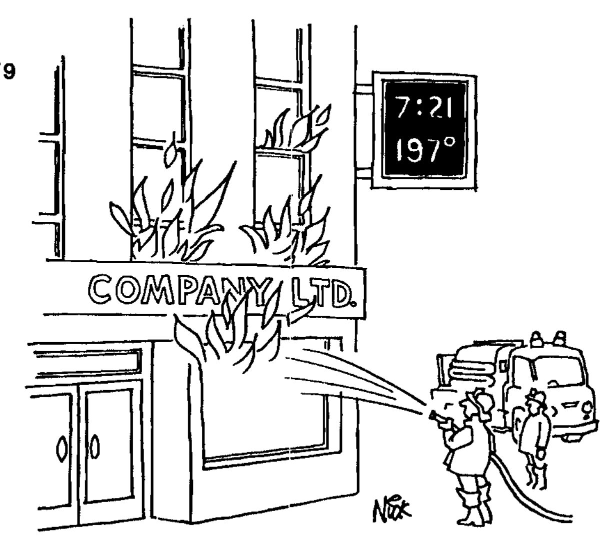
QUESTION: CBW AND CWD ARE USED WITH SIGNED NUMBERS. HOW DO YOU ACHIEVE THE SAME RESULT WITH UNSIGNED NUMBERS?

* 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



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

INSTRUCTION	CONDITION	INTERPRETATION
JL OR JNGE	(SF XOR OF)=1	"LESS" OR "NOT GREATER"OR EQUAL"
JLE OR JNG	((SF XOR OF) OR ZF)=1	"LESS OR EQUAL" OR "NOT GREATER"
JNL OR JGE	(SF XOR OF)=0	"NOT LESS" OR "GREATER OR EQUAL"
JNLE OR JG	((SF XOR OF) OR ZF)=0	"NOT LESS"OR"EQUAL" OR "GREATER"
JO	OF=1	"OVERFLOW"
JS	SF=1	"SIGN"
JNO	OF=0	"NOT OVERFLOW"
JNS	SF=0	"NOT SIGN"

CONDITIONAL JUMPS FOR UNSIGNED OPERATIONS

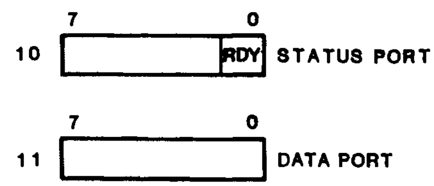
INSTRUCTION	CONDITION	INTERPRETATION
JB OR JNAE OR JC	CF=1	"BELOW" OR "NOT ABOVE"OR"EQUAL"
JBE OR JNA	(CF OR ZF)=1	"BELOW OR EQUAL" OR "NOT ABOVE"
JNB OR JAE OR JNC	CF=0	"NOT BELOW" OR "ABOVE OR EQUAL"
JNBE OR JA	(CF OR ZF)=0	"NOT BELOW"OR"EQUAL" OR "ABOVE"

CONDITIONAL JUMPS FOR SIGNED AND UNSIGNED OPERATIONS

INSTRUCTION	CONDITION	INTERPRETATION
JE OR JZ	ZF= 1	"EQUAL" OR "ZERO"
JP OR JPE	PF=1	"PARITY" OR PARITY EVEN"
JNE OR JNZ	ZF=0	"NOT EQUAL" OR "NOT ZERO"
JNP OR JPO	PF=0	"NOT PARITY" OR "PARITY ODD"
JCXZ	CX=0	"CX REGISTER IS ZERO"

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

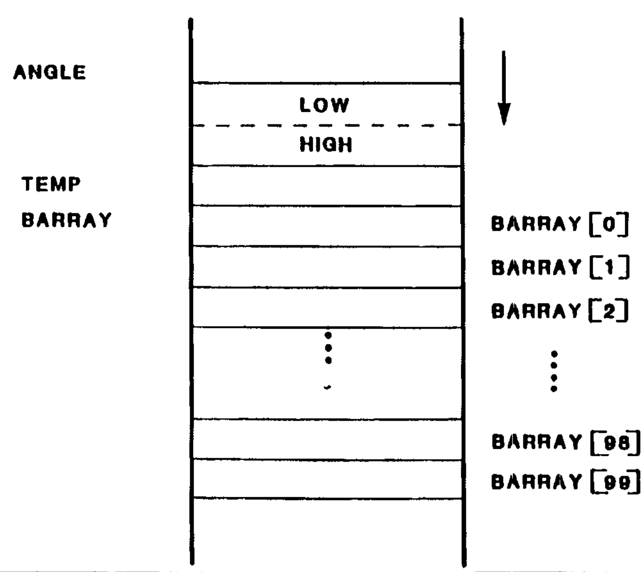
DB – DEFINE BYTE
DW – DEFINE WORD
DD – DEFINE DOUBLE WORD
DQ – DEFINE QUAD WORD } 8Ø87 DATA TYPES
DT – DEFINE TEN BYTES }

EXAMPLES:

```
BYTE1   DB  3              ;INITIALIZED BYTE
BYTE2   DB  ?              ;UNINITIALIZED BYTE
BYTE3   DB  6,7,8          ;3 INITIALIZED BYTES
STRING  DB  'MESSAGE'      ;7 INITIALIZED BYTES
ARRAY   DB  1ØØ DUP(Ø)     ;1ØØ ZEROED BYTES

WORD1   DW  Ø3ØØH          ;ØØ    Ø3
                           ;(LOW) (HIGH)
```


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
         :
```

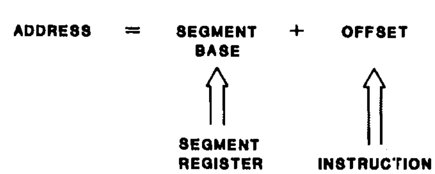
WHAT ARE THE OFFSETS OF XYZ AND YYY?

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



- 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 <u>CAN</u> USE?

ASSUME DECLARATIVE

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

EXAMPLE

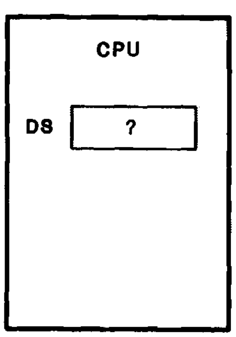
```
DATA_1      SEGMENT
XYZ         DB          ?
DATA_1      ENDS
CODE_1      SEGMENT
            ASSUME      DS:DATA_1,CS:CODE_1

            MOV         XYZ,10H
CODE_1      ENDS
```

* 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

                ?
```

TOTAL SOLUTION

```
8086/8087/8088 MACRO ASSEMBLER    DEMO1                                  09/01/80   PAGE    1

LOC  OBJ                  LINE    SOURCE

                             1    NAME    DEMO1
                             2
----                         3    DATA_1  SEGMENT
0000 ??                      4    XYZ     DB       ?
----                         5    DATA_1  ENDS
                             6
                             7
----                         8    CODE_1  SEGMENT
                             9            ASSUME   CS:CODE_1,DS:DATA_1
                            10
0000 B8----           R     11    START:  MOV      AX,DATA_1
0003 8ED8                   12            MOV      DS,AX
                            13
0005 C606000010             14            MOV      XYZ,10H          ;MOV 10H INTO MEMORY
                            15                                      ;LOCATION DS:XYZ
----                        16    CODE_1  ENDS
                            17
                            18            END      START
```


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:

OFFSET = [VARIABLE NAME] + [[BX] / [BP]] + [[SI] / [DI]] + [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] + DISPLACEMENT**
MOV AX, [BX] [DI]	**BASED INDEXED**	**OFFSET = [BX] + [DI]**
MOV AX, [BP + SI + 15]	**BASED INDEXED AND DISPLACEMENT**	**OFFSET = [BP] + [SI] + DISPLACEMENT**

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

ADDRESSING SIMPLE VARIABLES

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

EXAMPLE:

```
LOC  OBJ                    SOURCE

                            NAME    DEMO1

----                        DATA_1  SEGMENT
0000 ??                     XYZ     DB      ?
0001 0020                   BETA    DW      2000H
----                        DATA_1  ENDS

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

0000 B8----         R       START:  MOV     AX,DATA_1
0003 8ED8                           MOV     DS,AX

0005 C606000010                     MOV     XYZ,10H         ;MOV 10H INTO MEMORY LOCATION
                                                            ;DS:XYZ

000A 20060000                       AND     XYZ,AL          ;AND LOCATION DS:XYZ WITH AL

000E 8B1E0100                       MOV     BX,BETA         ;MOV CONTENTS OF BETA INTO BX


----                        CODE_1  ENDS
```

*** OFFSET = VARIABLE NAME**

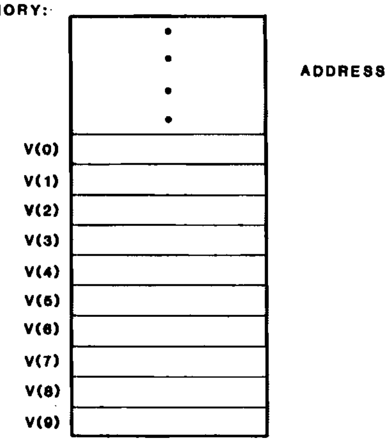
ARRAYS

* 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:



IN ASSEMBLY LANGUAGE:

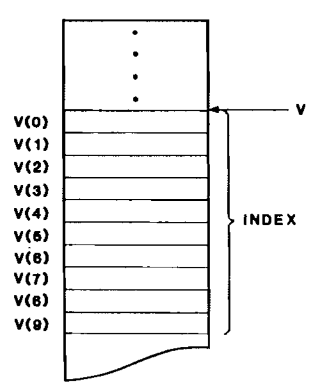
```
DATA_1    SEGMENT
  V       DB      10 DUP (?)
DATA_1    ENDS
```


ACCESSING ARRAYS

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

EXAMPLE:

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



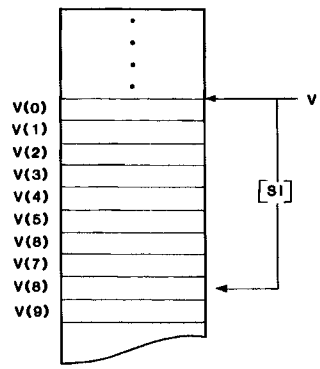
* OFFSET = VARIABLE NAME + [DISPLACEMENT]

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[8]

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



* OFFSET = VARIABLE NAME + [SI]

* ALL INDEXING IS ON A BYTE LEVEL

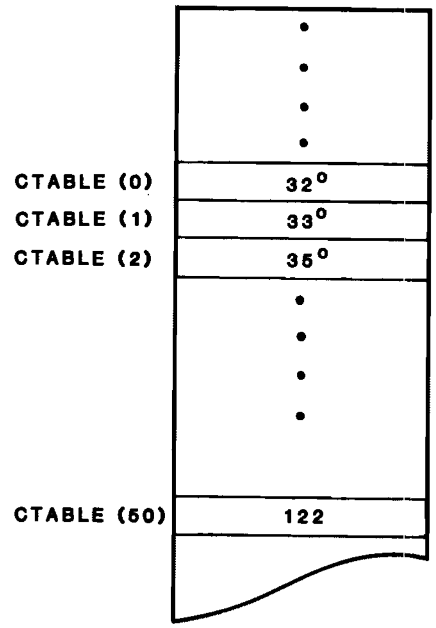
EXAMPLE

PROBLEM

AN 8 BIT VALUE REPRESENTING A TEMPERATURE IN THE RANGE 0^{o} TO 50^{o}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^{o}C. EACH SUCCESIVE ENTRY CORRESPONDS TO AN INTEGRAL CELSIUS DEGREE 1^{o},2^{o},....50^{o}C. THE CONVERTED VALUE IS TO BE STORED AT A BYTE LOCATION CALLED 'FTEMP'.

EXAMPLE

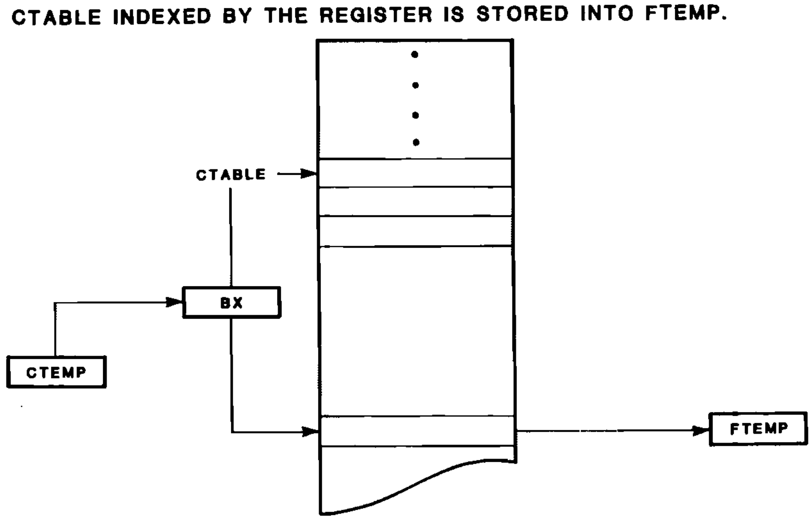
* IN MEMORY "CTABLE" APPEARS



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.



ASSEMBLY LANGUAGE SOLUTION

```
8086/8087/8088 MACRO ASSEMBLER    LESSON_4                              09/01/80  PAGE    1

LOC  OBJ                    LINE     SOURCE

                              1      NAME     LESSON_4
                              2
----                          3      DATA_1   SEGMENT
0000 ??                       4      CTEMP    DB      ?
0001 ??                       5      FTEMP    DB      ?
----                          6      DATA_1   ENDS
                              7
----                          8      CODE_1   SEGMENT
                              9               ASSUME  CS:CODE_1,DS:DATA_1

0000 20                      10      CTABLE   DB      32,33,35,. . .
0001 21
0002 23
0003 7A                      11               DB      122               ;FARENHEIT TEMPERATURES
                             12
                             13
0004 B8----          R       14      START:   MOV     AX,DATA_1
0007 8ED8                    15               MOV     DS,AX
                             16
                             17
                             18
0009 32FF                    19               XOR     BH,BH             ;CLEAR UPPER BYTE OF BX
000B 8A1E0000                20               MOV     BL,CTEMP          ;GET CELCIUS TEMP. INTO BX
000F 2E8A07                  21               MOV     AL,CTABLE[BX]     ;GET CONVERTED TEMP INTO AL
0012 A20100                  22               MOV     FTEMP,AL
                             23
                             24
----                         25      CODE_1   ENDS
                             26               END     START
```


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 ...

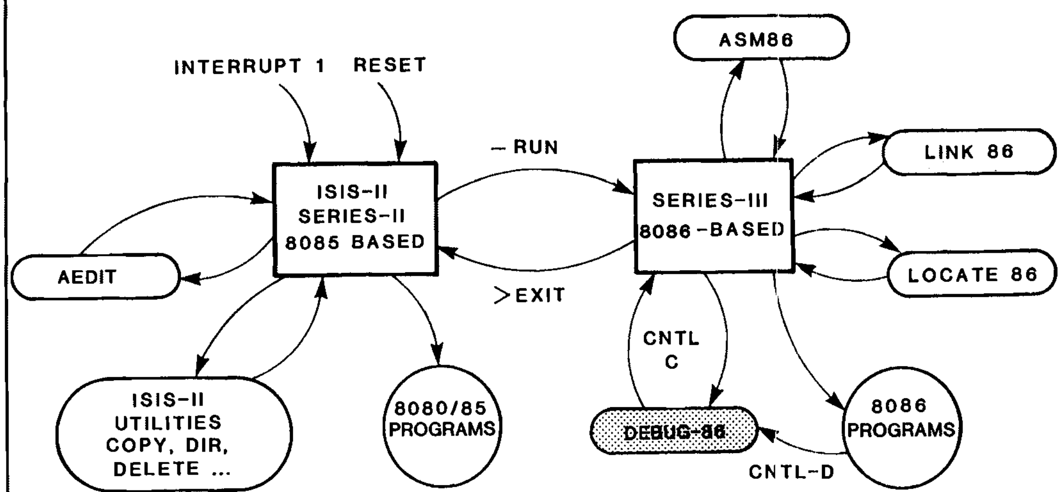
ASM86 LETS YOU DEFINE VERY COMPLEX DATA ITEMS USING STRUCTURES (A COLLECTION OF DISSIMILAR DATA ITEMS) AND RECORDS (VARIABLE BIT LENGTH FIELDS). USING "HIGH LEVEL" DATA ITEMS SUCH AS STRUCTURES AND RECORDS WILL IMPROVE THE DOCUMENTATION AND RELIABILITY OF YOUR PROGRAMS. READ CHAPTER 3 OF THE ASM86 LANGUAGE 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



9-1

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

```
8086/87/88/186 MACRO ASSEMBLER    DEMO                          10:06:11 12/27/83 PAGE 1


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

LOC  OBJ          LINE    SOURCE

                    1     NAME    DEMO
                    2
----                3     DATA    SEGMENT
0000 0100           4     WVAR    DW      1
0002 (10            5     ARRAY   DB      10 DUP(?)
     ??
     )
----                6     DATA    ENDS
                    7
----                8     CODE    SEGMENT
                    9             ASSUME CS:CODE,DS:DATA
                   10
0000 B8----    R   11     START:  MOV     AX,DATA          ;INITIALIZE DS
0003 8ED8          12             MOV     DS,AX
0005 33F6          13             XOR     SI,SI            ;ARRAY POINTER
0007 B90A00        14             MOV     CX,LENGTH ARRAY
000A 8B160000      15             MOV     DX,WVAR          ;GET ADDRESS OF PORT
000E EC            16     AGAIN:  IN      AL,DX            ;INPUT THE VALUE
000F 884402        17             MOV     ARRAY[SI],AL     ;AND SAVE IN ARRAY
0012 46            18             INC     SI
0013 E2F9          19             LOOP    AGAIN            ;DO IT 10 TIMES
0015 EBE9          20             JMP     START            ;REPEAT
----               21     CODE    ENDS
                   22             END     START
```


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:

```
* REGISTERS
  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

* BYT .ARRAY LENGTH 10T
BYT 0481:0002H=FBH B8H E3H OFH 50H B8H ECH ODH 60H E8H

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

*BYTE .ARRAY TO .ARRAY+9
BYT 0481:0002H=FFH  OOH  FFH OOH FFH OOH FFH OOH FFH OOH
```


USING THE DEBUGGER

* DISPLAY COMMANDS

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

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

USING THE DEBUGGER

DISPLAY COMMANDS

- TO DISPLAY/CHANGE I/O PORTS

```
* PORT 0
POR 0000H=55H

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

* WPORT 1000
WPO  1000H=00FFH

*  PORT 0=FF
```


USING THE DEBUGGER

* PROGRAM EXECUTION COMMANDS

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

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

```
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:

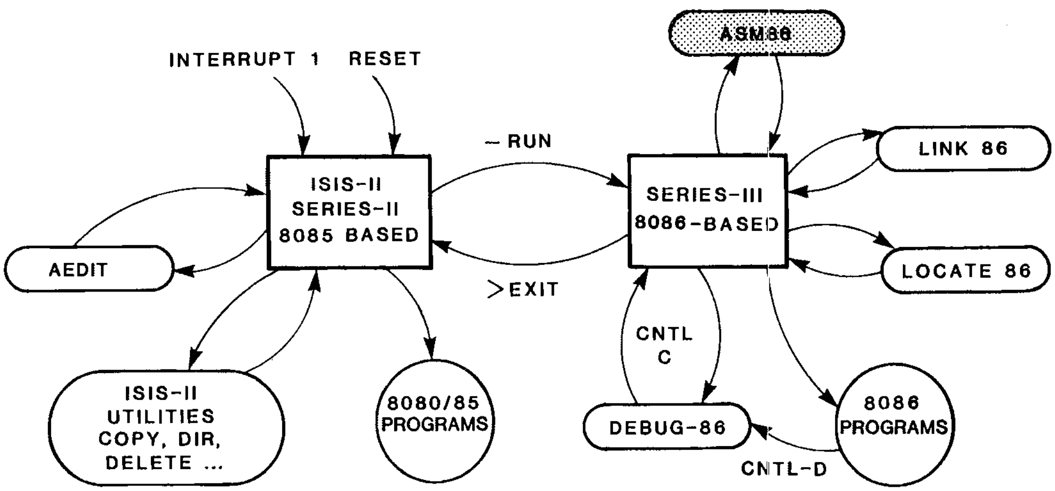
EXIT
OR
CNTRL-C

* 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



8086/8088 ASSEMBLER CONTROLS

-RUN ASM86 :F1:LAB.ASM OPTIONS

PRIMARY CONTROLS

OBJECT (FILENAME) NOOBJECT	* OJ NOOJ	CONTROL CREATION AND DESTINATION OF .OBJ FILE NO .OBJ FILE
PRINT(FILENAME) NOPRINT	* PR NOPR	CONTROL CREATION AND DESTINATION OF .LST FILE NO .LST FILE
PAGING/NOPAGING	* PI/NOPI	PAGINATE/DON'T PAGINATE LISTING
SYMBOLS/NOSYMBOLS	SB/NOSB *	APPEND/DON'T APPEND SYMBOL TABLE TO LISTING
ERRORPRINT(FILENAME)	EP/NOEP *	SEND ERRORS TO DEVICE SPECIFIED/DON'T REPORT ERRORS
DEBUG/NODEBUG	DB/NODB *	APPEND/DON'T APPEND SYMBOL TABLE TO OBJECT FILE

* DEFAULT

8086/8088 ASSEMBLER CONTROLS (CONT'D)

GENERAL CONTROLS

LIST*/NOLIST	* LI	INCLUDE ALL LINES FOLLOWING IN LISTING FILE
	NOLI	SUSPEND LISTING
EJECT	EJ	FORCE A FORM FEED (OVERRIDDEN 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  <-------  PRIMARY CONTROLS MUST
                        NAME   EXAMPLE      BE ON FIRST LINE OF
MUST BE IN                                  SOURCE FILE
 COLUMN 1   --->  $INCLUDE(:FI:EQUS.SRC)
                  $EJECT
                  CODE    SEGMENT
```

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

```
ARRAY     DW 100 DUP(?)
ADD       SI,TYPE ARRAY        ;ADJUST SI TO NEXT ELEMENT
MOV       CX,LENGTH ARRAY      ;LOADS CX WITH 100
MOV       D1,SIZE ARRAY        ;LOADS SI WITH 200
```


SERIES III DEVELOPMENT STEPS

- AEDIT :F1:LAB1.ASM	COMPOSE SOURCE PROGRAM
- RUN ASM86 :F1:LAB1.ASM DEBUG	ASSEMBLE PROGRAM
- COPY :F1:LAB1.LST TO :LP:	COPY ASSEMBLER OUTPUT LIST FILE TO THE PRINTER
- RUN LINK86 :F1:LAB1.OBJ BIND	PRODUCE LOAD TIME LOCATABLE CODE FOR EXECUTION ON SERIES III
- RUN DEBUG	INVOKE DEBUGGER
* LOAD :F1:LAB1	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 :FI: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 :FI:SBMT

PASSING PARAMETERS

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

```
RUN   ASM86    :%O:%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 PROB.LEM 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 LISTIN.G

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, ASM86 LANGUAGE REFERENCE MANUAL

RELATED TOPICS...

ASM86 SUPPORTS USER DEFINED TEXT MACROS INCLUDING CONDITIONAL ASSEMBLY. SEE CHAPTER 7 OF THE ASM86 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

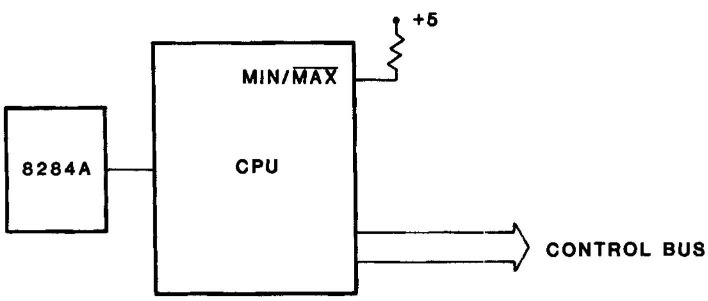
THE iAPX 86,88 SYSTEM

* 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 PREFETCH QUEUE

10-1

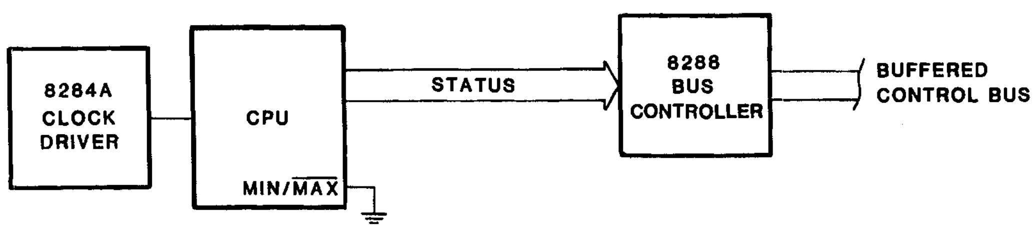
iAPX 86,88 ARCHITECTURE (MINIMUM MODE)

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

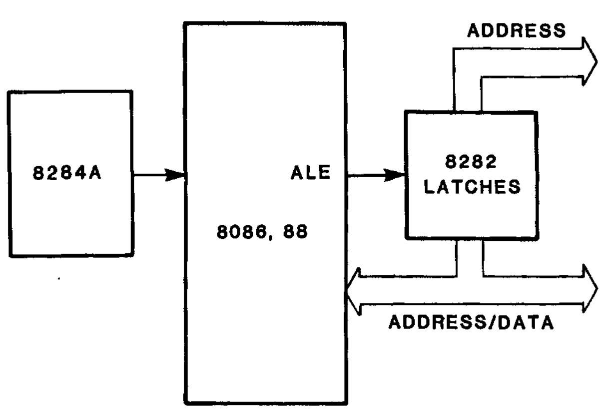


iAPX 86,88 ARCHITECTURE (MAXIMUM MODE)

* 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

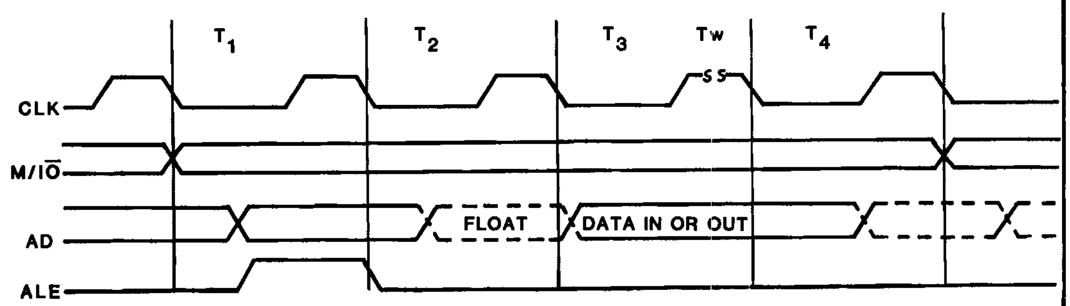


8086, 88 CPU SET AND BUS STRUCTURE MINIMUM AND MAXIMUM MODE



8086	-	5 MHZ
8086-4	-	4 MHZ
8086-2	-	8 MHZ
8086-1	-	10 MHZ

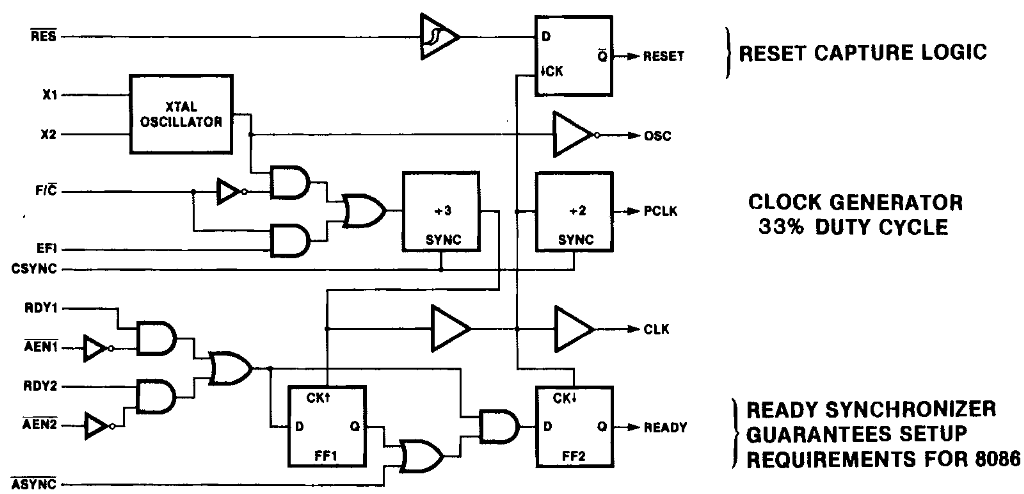
8086, 88 BASIC BUS CYCLE



8284A CLOCK GENERATOR

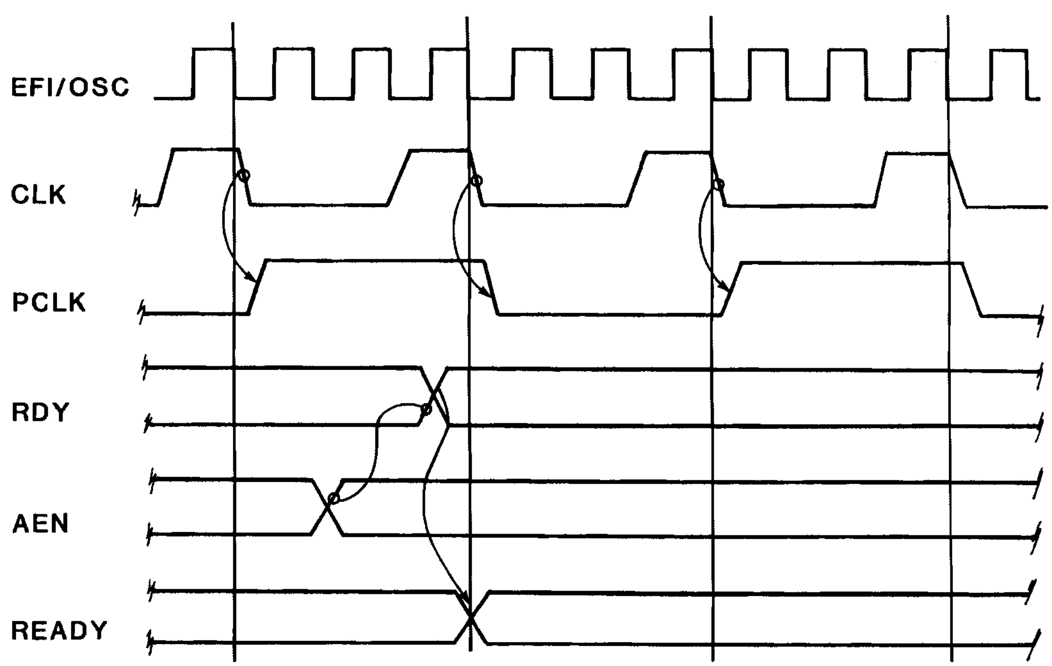
* GENERATES SYSTEM CLOCK FOR 8086/8088
* USES CRYSTAL OR TTL SIGNAL FOR FREQUENCY SOURCE
* PROVIDES LOCAL READY AND MULTIBUS READY SYNCHRONIZATION
* GENERATES SYSTEM RESET OUTPUT FROM SCHMITT TRIGGER INPUT

8284A BLOCK DIAGRAM



10-7

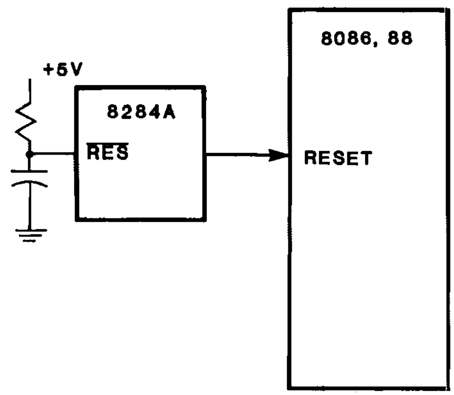
8284A TIMING



RESET

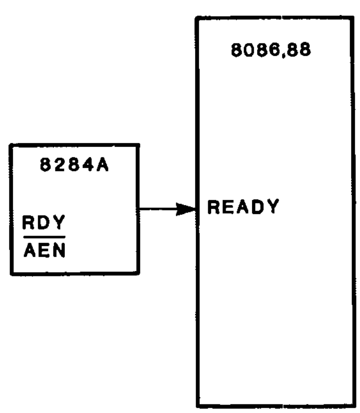
RESET–SUPPLIED BY 8284A CLOCK GENERATOR

FLAGS ← 0
CS ← FFFF
IP,DS,SS,ES ← 0

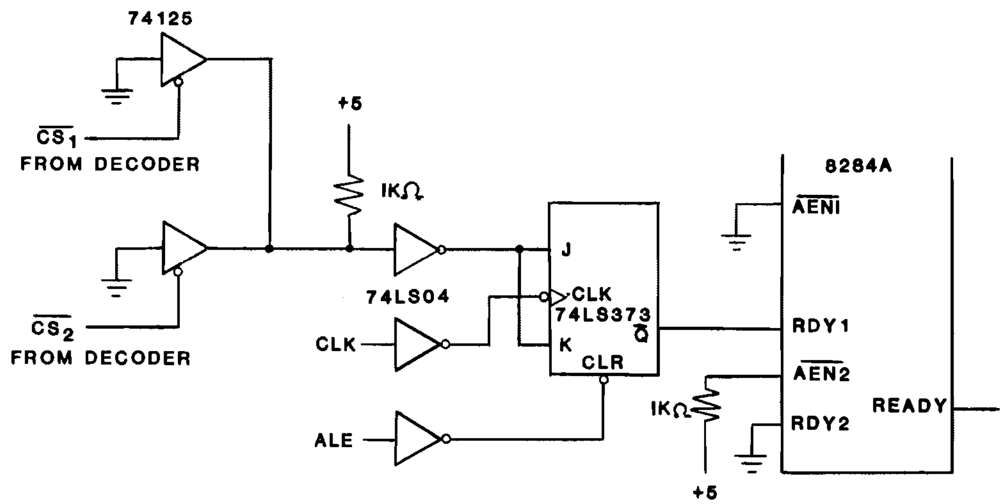


READY

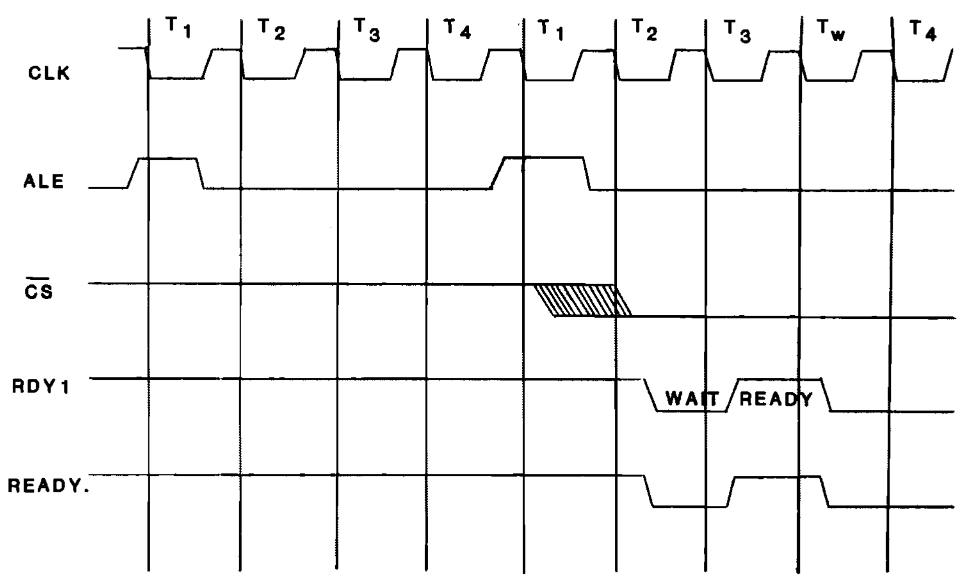
- READY IS SYNCHRONIZED WITH THE CPU BY THE CLOCK GENERATOR
- READY IS USED TO EXTEND A BUS CYCLE BY ONE OR MORE CLOCK CYCLES
- INCREASES THE AMOUNT OF TIME THAT CPU GIVES MEMORY TO RESPOND WITH OR ACCEPT DATA
- THE USER MUST DESIGN THE HARDWARE WHICH DECODES THE BUS ADDRESS AND DETERMINES IF "WAIT STATES"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

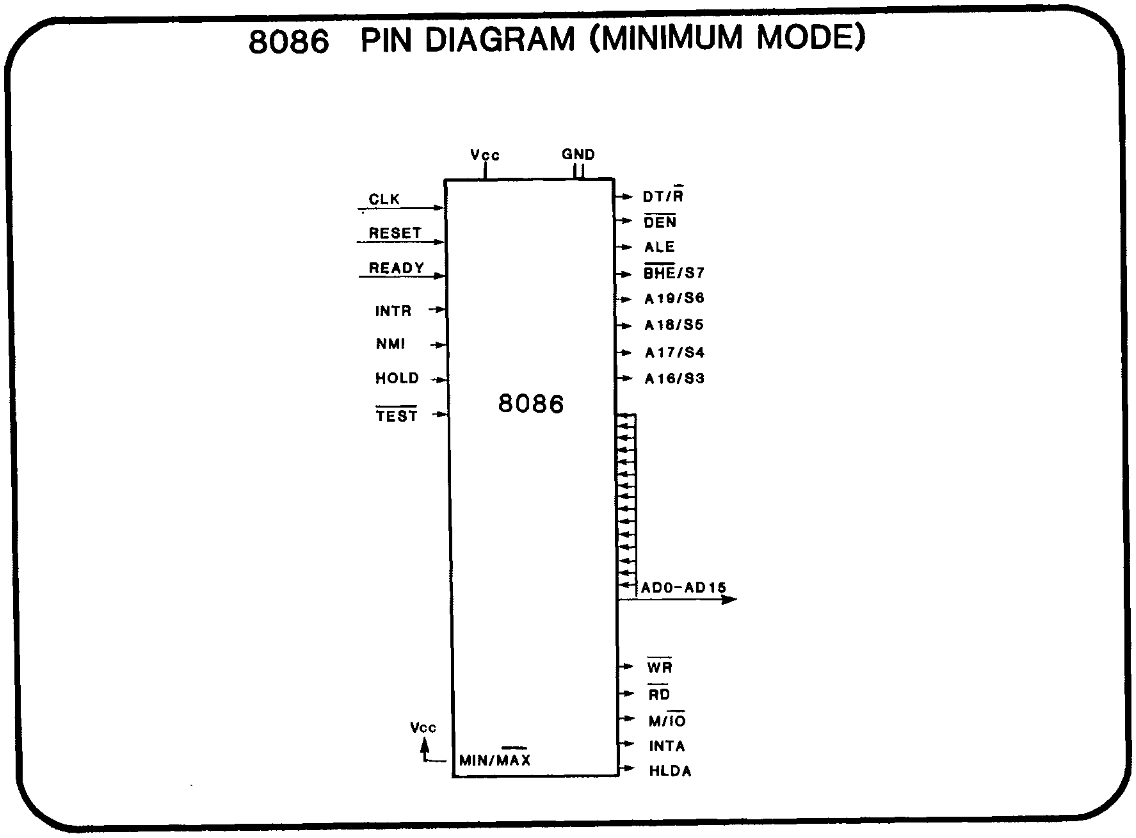


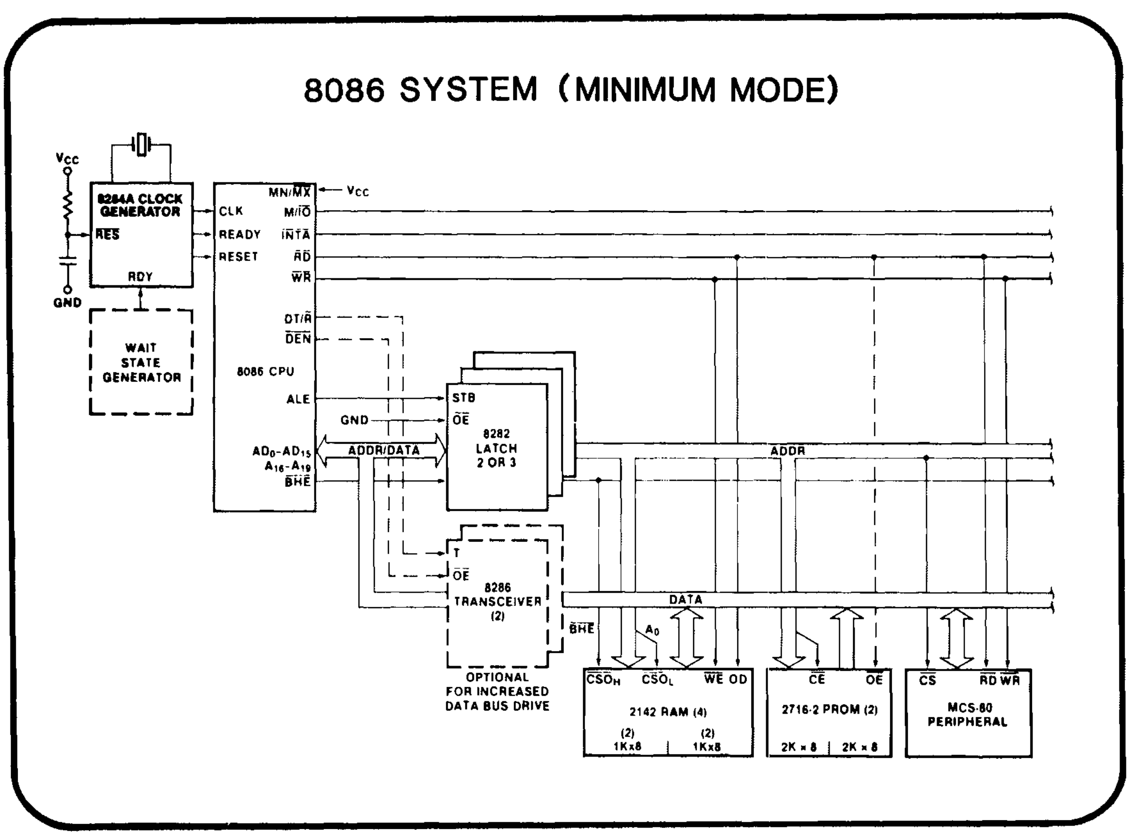
SINGLE WAIT STATE GENERATOR



BUS CYCLE WITH WAIT STATES




8086 PIN DIAGRAM (MINIMUM MODE)
Vcc
GND
CLK
RESET
READY
INTR
NMI
HOLD
TEST
8086
DT/R
DEN
ALE
BHE/S7
A19/S6
A18/S5
A17/S4
A16/S3
AD0-AD15
WR
RD
M/IO
INTA
HLDA
Vcc
MIN/MAX


8086 SYSTEM (MINIMUM MODE)
Vcc
GND
8284A CLOCK GENERATOR
RES
RDY
WAIT STATE GENERATOR
CLK
READY
RESET
MN/MX
M/IO
INTA
RD
WR
DT/R
DEN
8086 CPU
ALE
AD0-AD15
A16-A19
BHE
GND
ADDR/DATA
STB
OE
8282 LATCH 2 OR 3
ADDR
T
OE
8286 TRANSCEIVER (2)
OPTIONAL FOR INCREASED DATA BUS DRIVE
DATA
BHE
A0
CSOH
CSOL
WE OD
2142 RAM (4)
(2) 1Kx8
(2) 1Kx8
CE
OE
2716-2 PROM (2)
2K x 8
2K x 8
CS
RD WR
MCS-80 PERIPHERAL

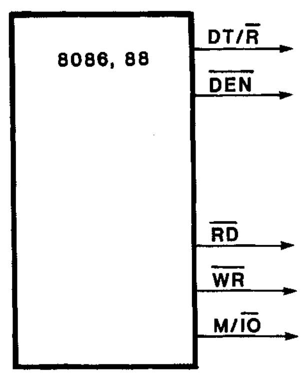
8086 SIGNAL DESCRIPTION (CONTROL SIGNALS)

DT/$\overline{R}$ - CONTROLS DIRECTION OF DATA THROUGH TRANSCEIVER

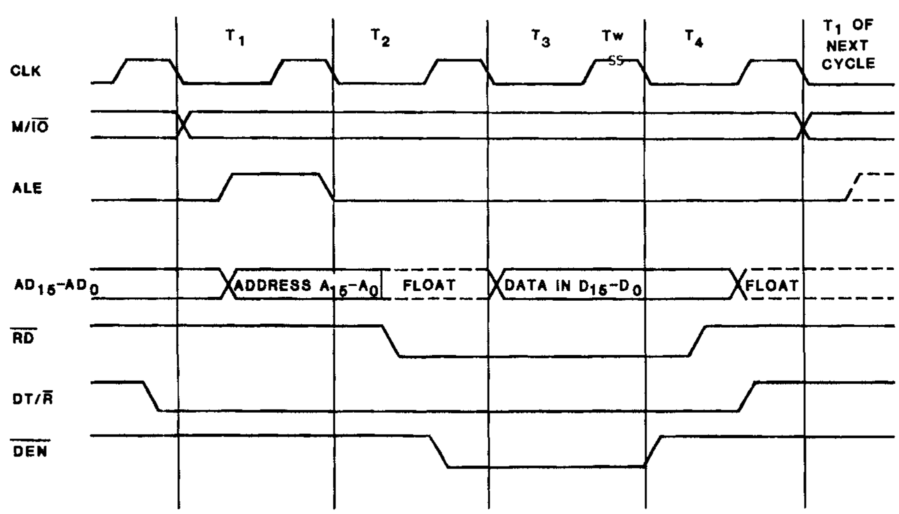
$\overline{DEN}$ - OUTPUT ENABLE FOR TRANSCEIVER

$\overline{RD}$,$\overline{WR}$ - INDICATES A READ OR WRITE CYCLE TO/FROM MEMORY OR I/O

M/$\overline{IO}$ - INDICATE WHETHER READ OR WRITE IS TO MEMORY OR I/O



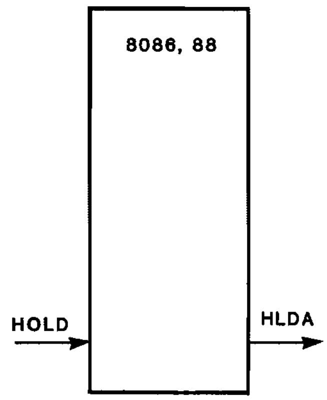
READ TIMING MINIMUM MODE



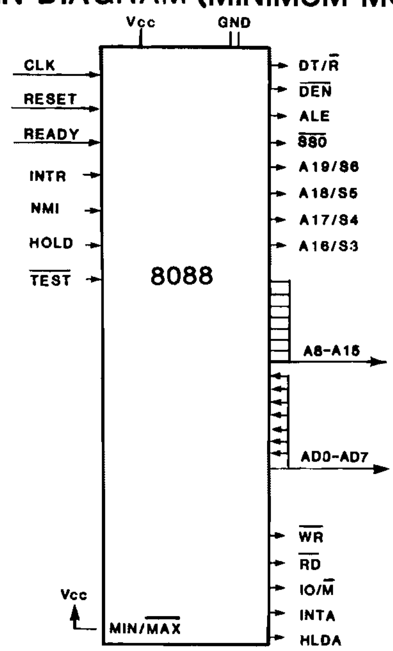
HOLD AND HLDA

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

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



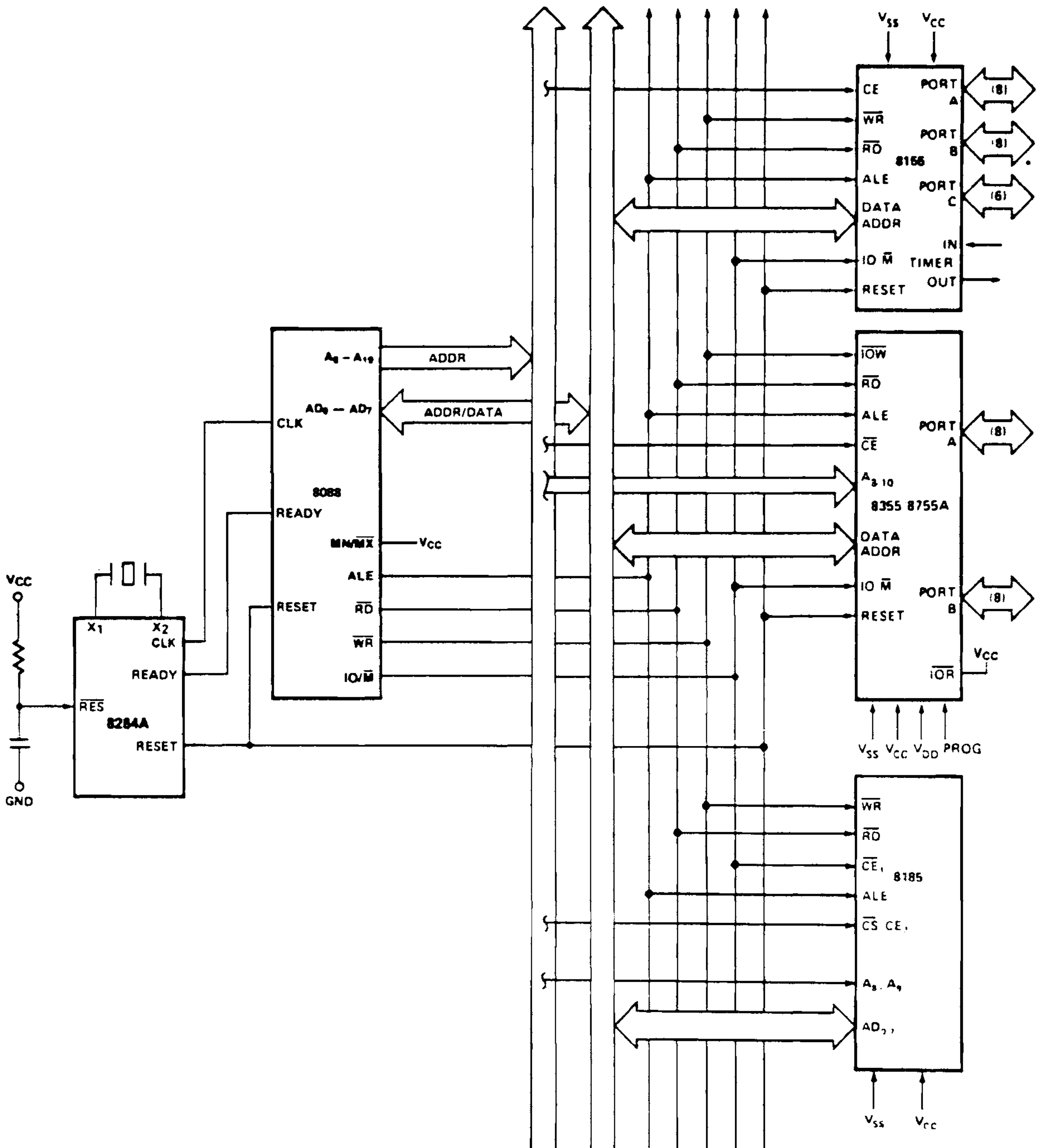
8088 PIN DIAGRAM (MINIMUM MODE)



DIFFERENCES IN PINOUT FROM 8086:

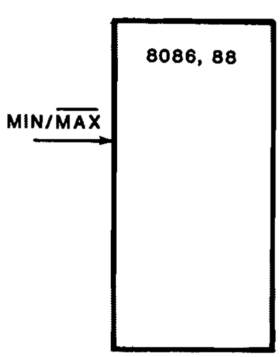
- NO $\overline{BHE}$ PIN: $\overline{SSO}$ ALONG WITH IO/$\overline{M}$ AND DT/$\overline{R}$ PROVIDE MACHINE CYCLE STATUS IN MIN MODE
- PIN 28 IS IO/$\overline{M}$ RATHER THAN M/$\overline{IO}$ TO BE COMPATIBLE WITH THE 8085
- A8 – A15 NOT MULTIPLEXED WITH DATA

8088 MULTIPLEXED BUS



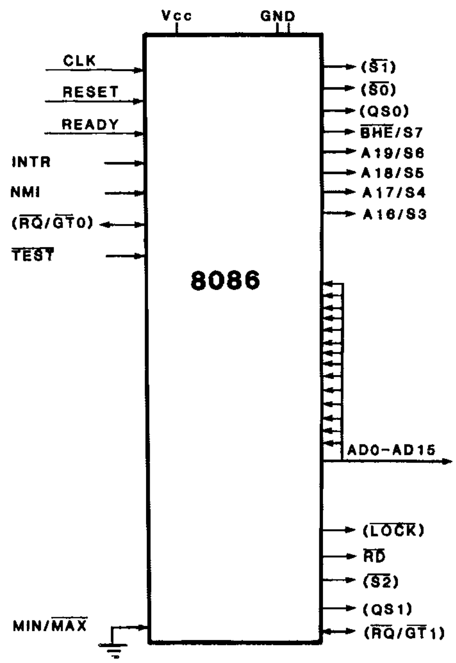
MIN/$\overline{\text{MAX}}$ SELECTION

MIN/$\overline{\text{MAX}}$ – MINIMUM OR MAXIMUM CONFIGURATION STRAPPING OPTION THAT ALTERS THE FUNCTIONS OF 8 OF THE CPU PINS AS FOLLOWS:

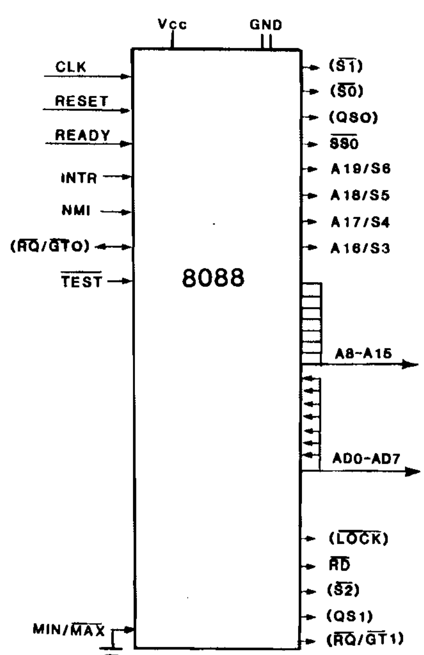


MINIMUM	MAXIMUM
$\overline{\text{WR}}$	$\overline{\text{LOCK}}$
$\overline{\text{INTA}}$	QS_1
ALE	QS_0
M/$\overline{\text{IO}}$	$\overline{S_0}$
DT/$\overline{\text{R}}$	$\overline{S_1}$
$\overline{\text{DEN}}$	$\overline{S_2}$
HLDA	$\overline{\text{RQ}}/\overline{\text{GT}}_0$
HOLD	$\overline{\text{RQ}}/\overline{\text{GT}}_1$

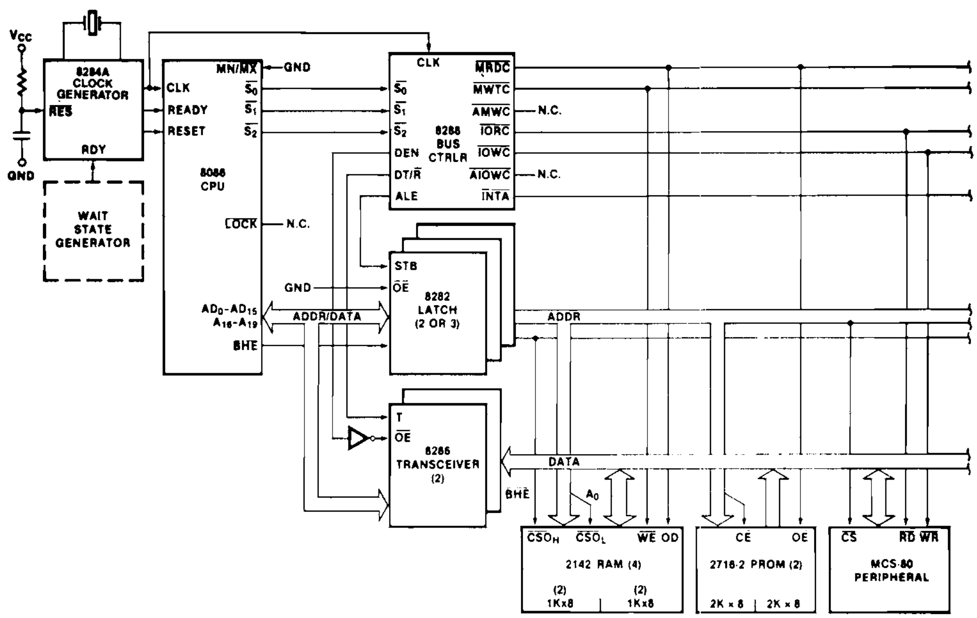
8086 PIN OUT (MAXIMUM MODE)



8088 PIN OUT (MAXIMUM MODE)

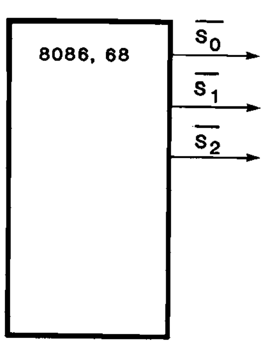


8086 SYSTEM (MAXIMUM MODE)



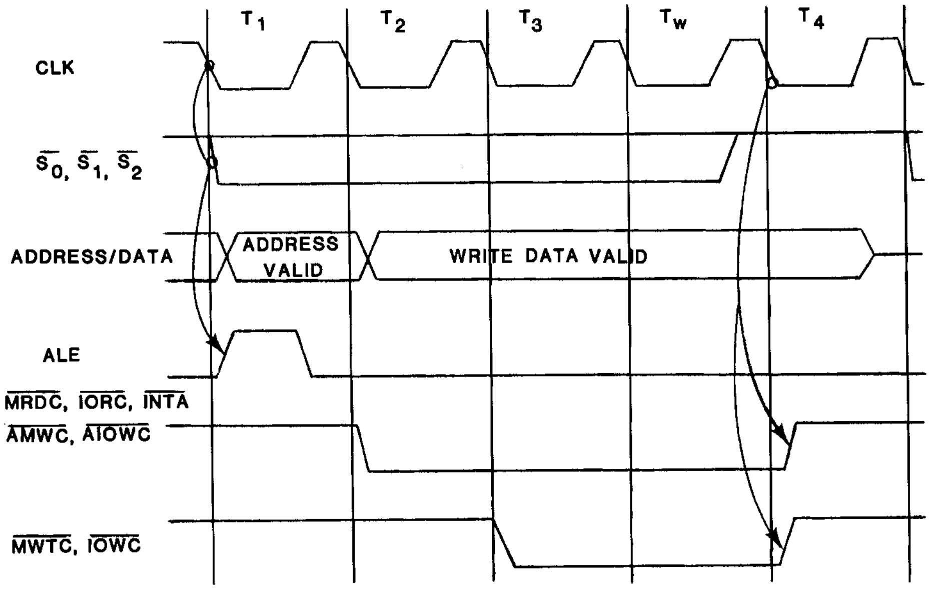
8086 SIGNAL DESCRIPTION (MAXIMUM MODE)

$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ - STATUS LINES THAT INFORM THE 8288 OF THE TYPE OF BUS CYCLE THAT THE 8076 IS RUNNING



$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	SIGNAL
0	0	0	INTA
0	0	1	I/O READ
0	1	0	I/O WRITE
0	1	1	HALT
1	0	0	CODE ACCESS
1	0	1	READ MEMORY
1	1	0	WRITE MEMORY
1	1	1	PASSIVE

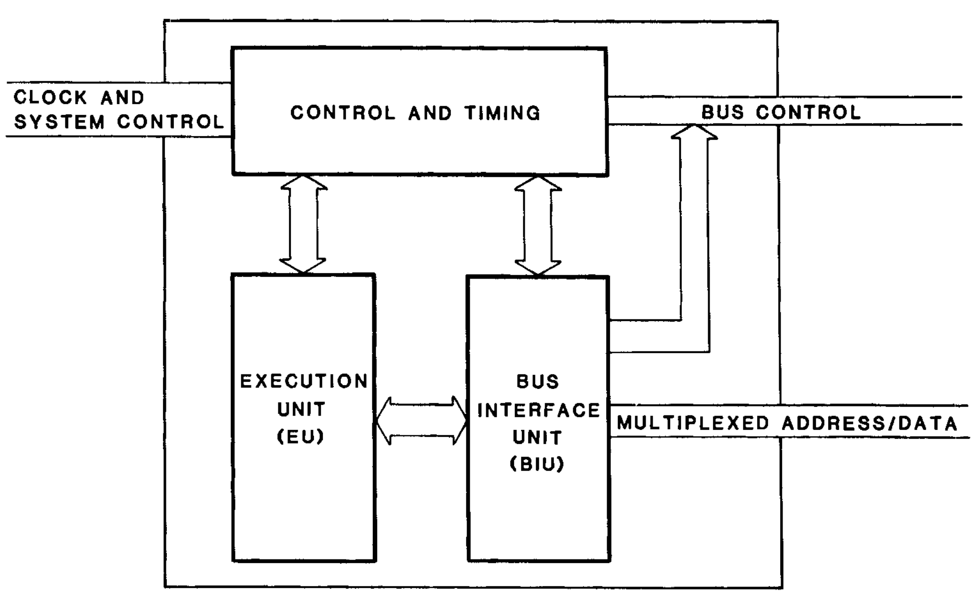
8288 TIMING



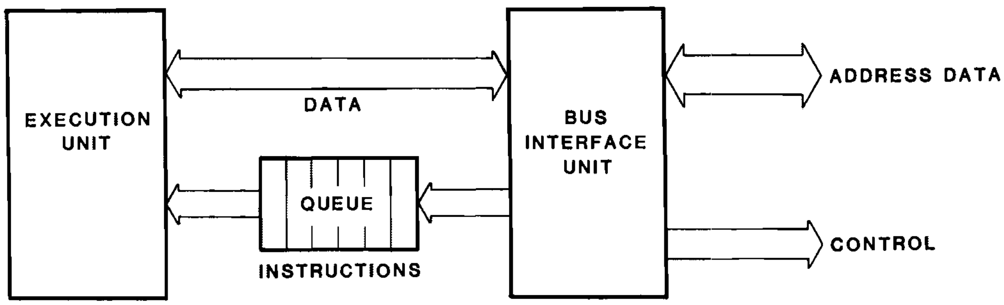
THE $\overline{AMWC}$, $\overline{AIOWC}$ ARE PROVIDED TO GENERATE LONGER STROBES REQUIRED BY SOME MEMORIES. THEY SHOULD NOT BE USED WITH DEVICES THAT LATCH DATA ON THE LEADING EDGE OF THE STROBE SINCE DATA IS NOT GUARANTEED TO BE VALID AT THAT TIME.

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

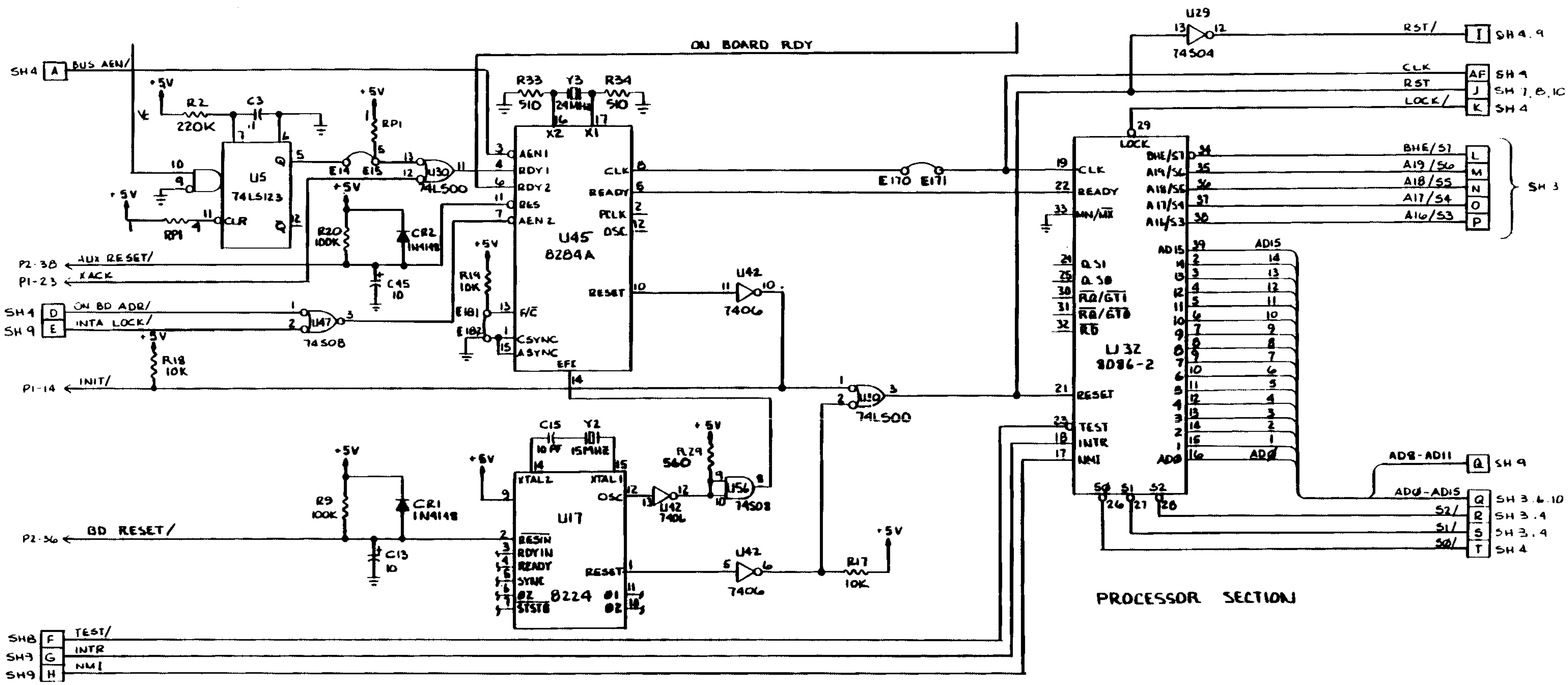
OUR DESIGN EXAMPLE

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

iSBC 86/05 SCHEMATIC

PAGE 2



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
- iSBC 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

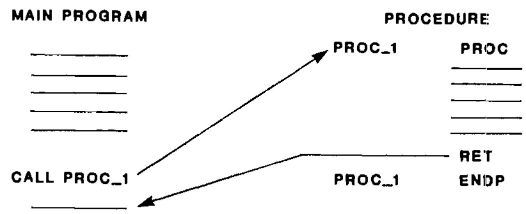
CHAPTER 11

PROCEDURES

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

PROCEDURES

* SECTIONS OF A PROGRAM THAT ARE CALLED AND RETURNED FROM

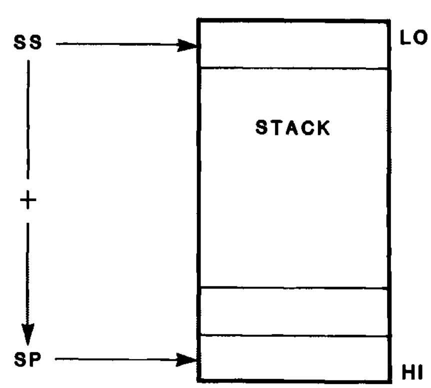


* THE CALL INSTRUCTION WRITES THE RETURN ADDRESS (THE ADDRESS OF THE NEXT INSTRUCTION) INTO THE STACK.

* 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

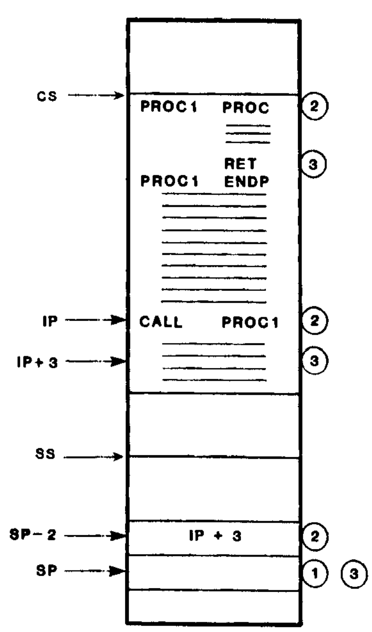


STACK INITIALIZATION

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

```
STACK_2         SEGMENT
                DW      100 DUP(?)
TOP_OF_STACK    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
        JZ      EXIT    ;if 0 - quit
LOOP_:  MOV     CL,78H  ;Count for 100 us
        SHR     CL,CL   ;Delay 100 us
        DEC     AL      ;Adjust iteration counter
        JNZ     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
```

COMMUNICATING WITH PROCEDURES (BASED ADDRESSING)

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

```
CRT       EQU    0FFH

OUTPROC PROC
          JCXZ   EXIT          ;CHECK FOR CX SETUP

MORE:     MOV    AL,[BX]       ;MOV CONTENTS OF BUFFER POINTED TO
                               ;BY BX INTO AL
          OUT    CRT,AL

          INC    BX            ;INCREMENT BX TO POINT TO NEXT LOCATION
                               ;IN BUFFER
          LOOP   MORE

EXIT:     RET

OUTPROC ENDP
```

* REMEMBER - OFFSET = [[BX] / [BP]] + [[DI] / [SI]]

* 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:

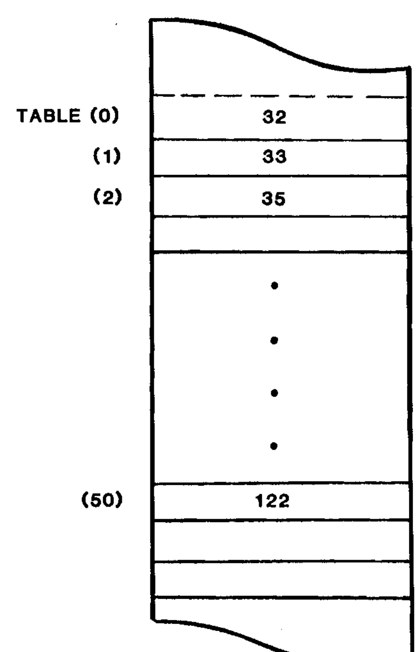
TEMPOUT = CONVERT (TEMPIN, N, TBLADR);

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

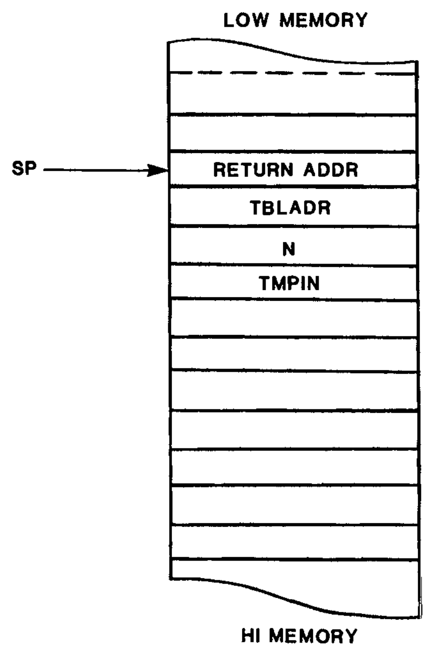
11-11

TABLE OF CONVERSION VALUES

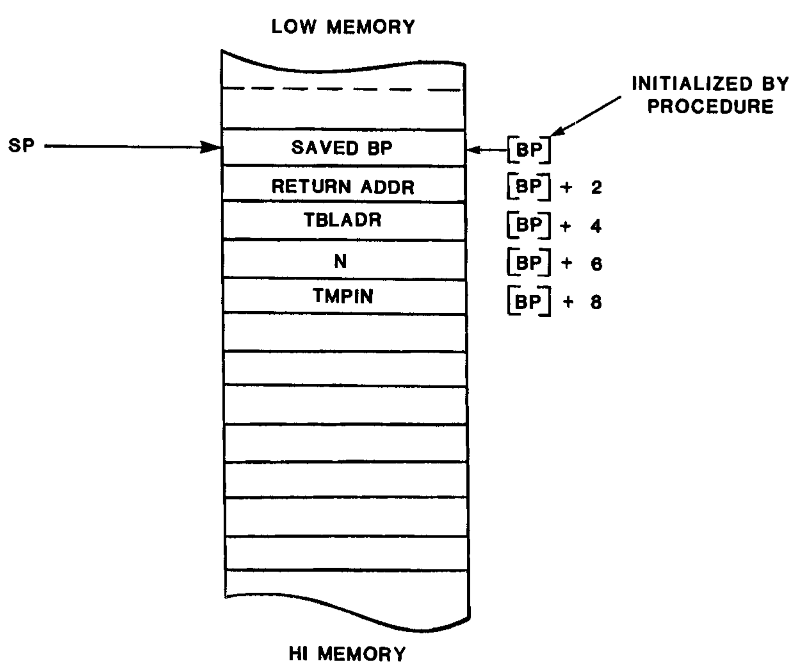
* TABLE LOCATED SOMEWHERE IN MEMORY.



STACK "FRAME" WITH PARAMETERS AFTER CALL



STACK "FRAME" WITH PARAMETERS AFTER ENTRY



EXAMPLE

SOLUTION:

```
8086/8087/8088 MACRO ASSEMBLER    DMO                                09/01/80  PAGE    1

LOC  OBJ                   LINE     SOURCE

                                NAME     DMO
----                            CODE     SEGMENT
                                         ASSUME CS:CODE



0000                            CONVERT  PROC
0000 55                                  PUSH    BP              ;SEE DIAGRAM
0001 8BEC                                MOV     BP,SP

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

000C 3BF7                                CMP     SI,DI           ;CHECK IF TMPIN > LENGTH OF TABLE
000E 7206                                JB      INRANG
0010 8A41FF                              MOV     AL,[BX+DI-1]    ;IF NOT IN RANGE USE GREATEST
                                                                 ;VALUE IN TABLE (LENGTH OF TABLE-1)
0013 EB0390                              JMP     EXIT
0016 8A00                       INRANG:  MOV     AL,[BX+SI]      ;USE SI TO POINT TO TEMP. VALUE
0018 5D                         EXIT:    POP     BP
0019 C20600                              RET     6
                                CONVERT  ENDP


                                CODE     ENDS
                                         END
```

DISCUSSION

STEP1 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 ...

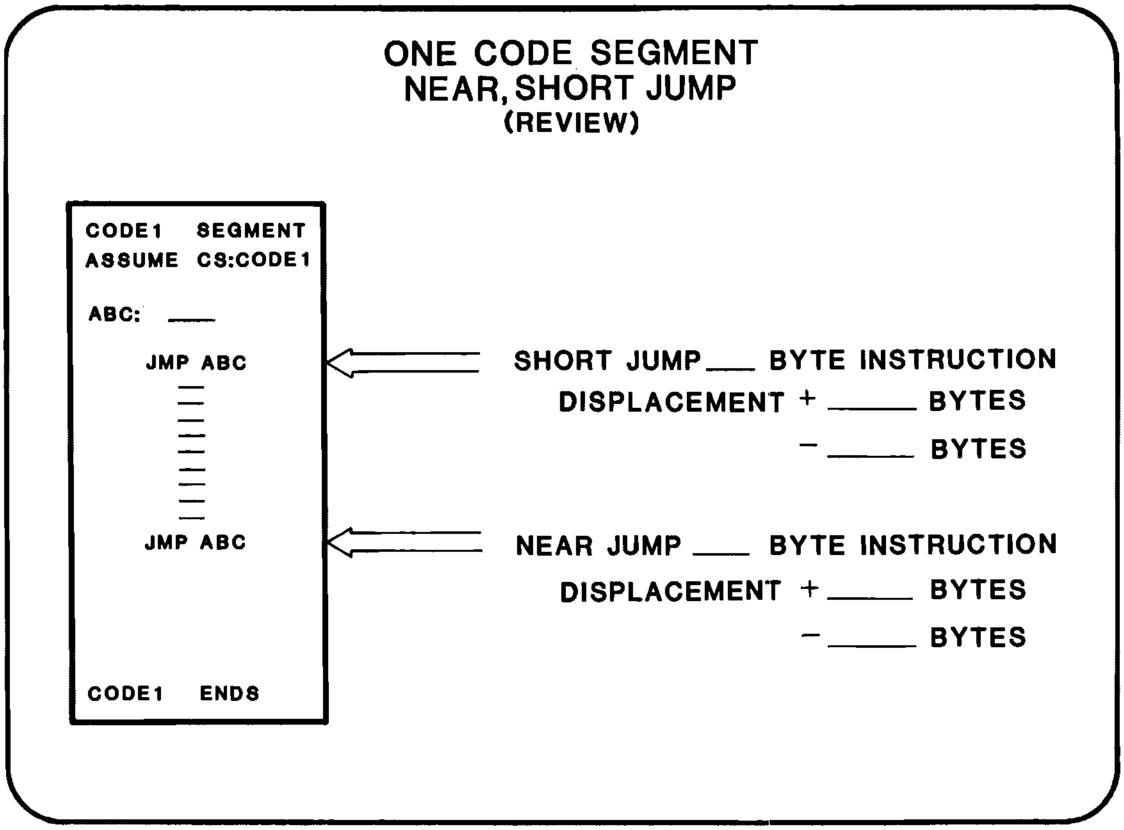
ASSEMBLY LANGUAGE INSTRUCTIONS
- CHAPTER 3, iAPX 86/88, 186/188 USER'S MANUAL
- CHAPTER 6, ASM86 LANGUAGE REFERENCE MANUAL

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

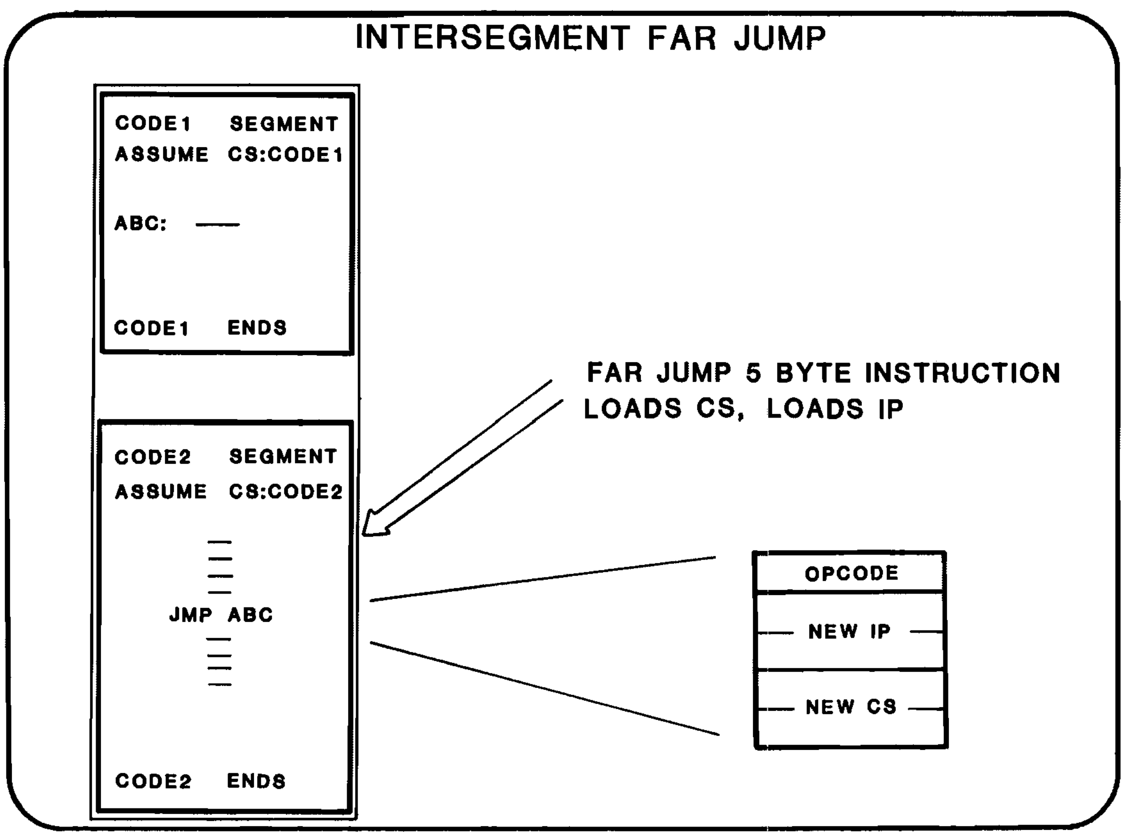
CHAPTER 12

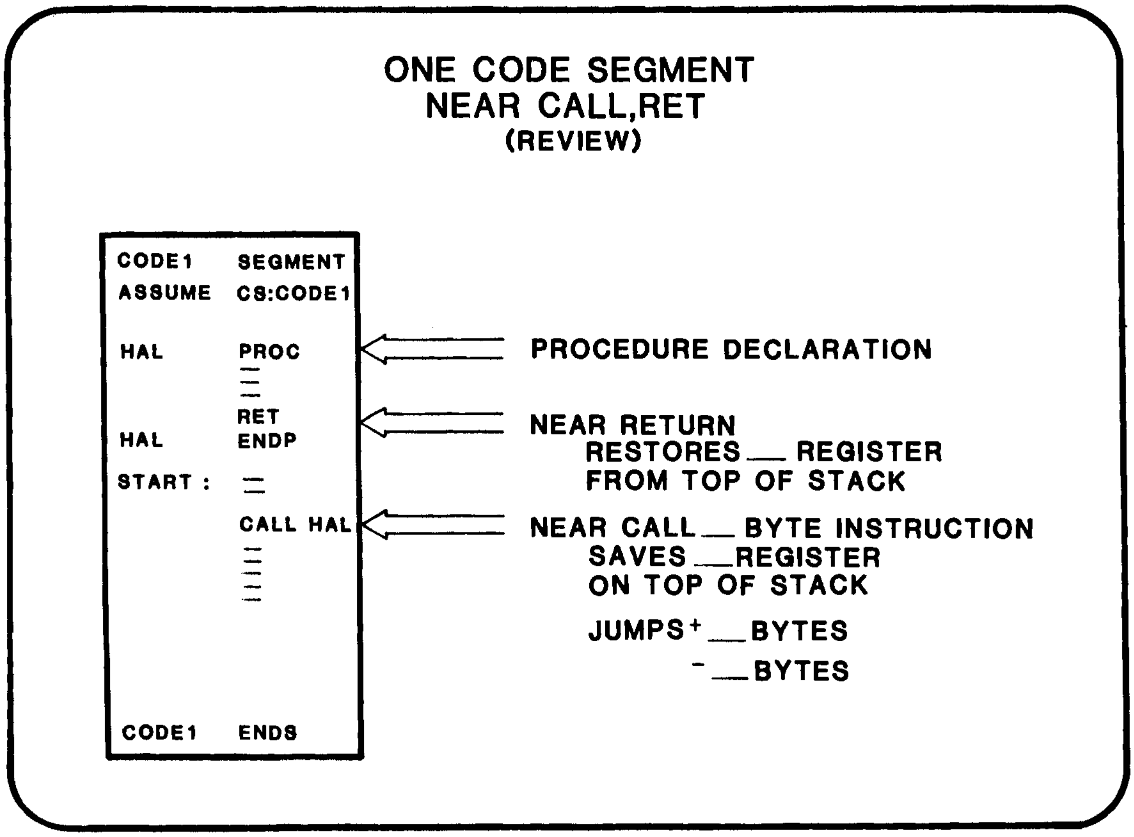
PROGRAMMING WITH MULTIPLE SEGMENTS

- MULTIPLE CODE SEGMENTS
- PROCEDURE DECLARATION
- MULTIPLE DATA SEGMENTS
- SEGMENT OVERRIDE INSTRUCTION PREFIX
- FORWARD REFERENCES

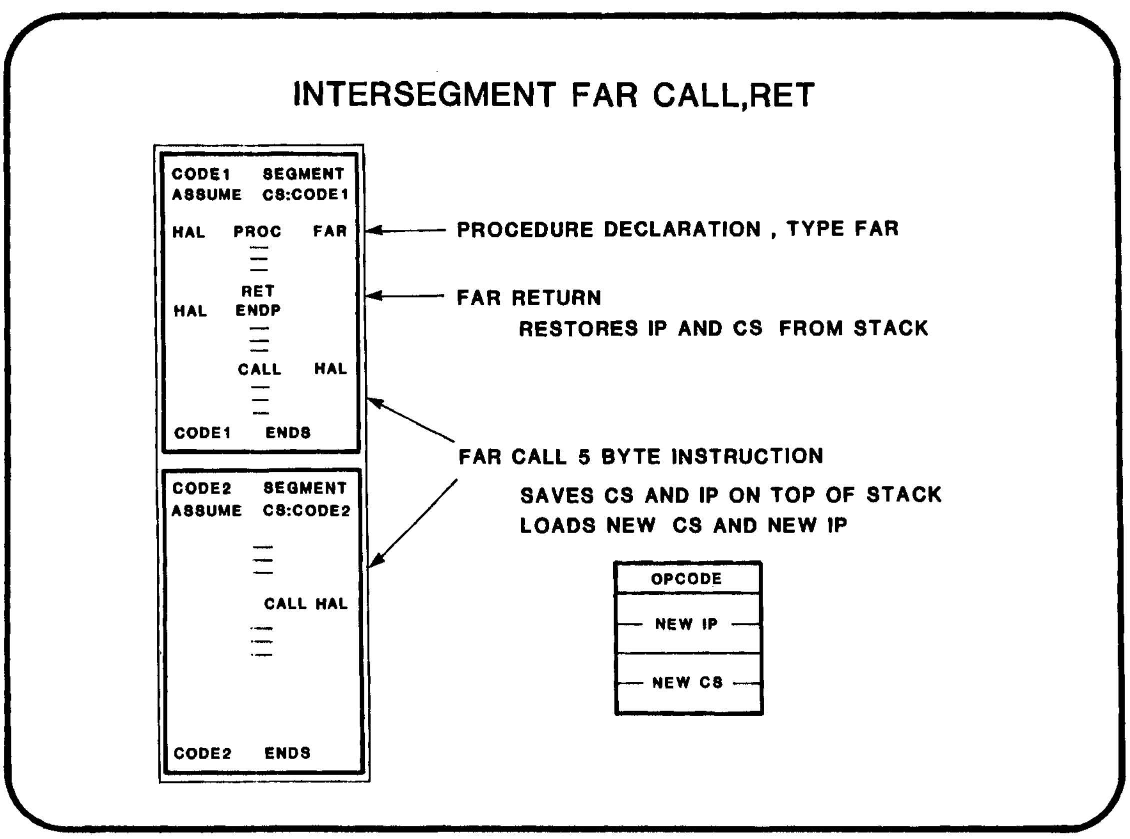

ONE CODE SEGMENT
NEAR, SHORT JUMP
(REVIEW)
CODE1 SEGMENT
ASSUME CS:CODE1
ABC:
JMP ABC
JMP ABC
CODE1 ENDS
SHORT JUMP ___ BYTE INSTRUCTION
DISPLACEMENT + ______ BYTES
− ______ BYTES
NEAR JUMP ___ BYTE INSTRUCTION
DISPLACEMENT + ______ BYTES
− ______ BYTES

12-1


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
HAL ENDP
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

12-3


INTERSEGMENT FAR CALL,RET
CODE1 SEGMENT
ASSUME CS:CODE1
HAL PROC FAR
RET
HAL ENDP
CALL HAL
CODE1 ENDS
CODE2 SEGMENT
ASSUME CS:CODE2
CALL HAL
CODE2 ENDS
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
OPCODE
NEW IP
NEW CS

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:

```
XYZ     PROC     {NEAR/FAR}
         ⋮
        RET
XYZ     ENDP
```


ONE DATA SEGMENT

REVIEW

DS →

```
DATA1        SEGMENT
               :
VAR1        DW ?
               :
DATA1        ENDS
```

CS →

```
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
0000  ??                          4     ALPHA    DB        ?
----                              5     DATA     ENDS
----                              6     CODE     SEGMENT
                                  7              ASSUME CS:CODE,DS:DATA
0000  0020                        8     BETA     DW        2000H
                                  9
0002  B8----            R        10     START:   MOV       AX,DATA
0005  8ED8                       11              MOV       DS,AX
                                 12
0007  2E8B0E0000                 13              MOV       CX,BETA     ;CS OVERRIDE
                                 14
000C  8A0E0000                   15              MOV       CL,ALPHA    ;NO OVERRIDE NECESSARY
                                 16
----                             17     CODE     ENDS
                                 18              END       START
```

USING MULTIPLE DATA SEGMENTS

```
LOC   OBJ                      LINE    SOURCE

                                  1    NAME      SAMPLE2
                                  2
----                              3    DATA      SEGMENT
0000 ??                           4    ALPHA     DB        ?
----                              5    DATA      ENDS
                                  6
----                              7    DATA_2    SEGMENT
0000 ????                         8    BETA      DW,       ?
----                              9    DATA_2    ENDS
                                 10
----                             11    CODE      SEGMENT
                                 12              ASSUME CS:CODE,DS:DATA,ES:DATA_2
                                 13
0000 B8----              R       14    START:    MOV       AX,DATA
0003 8ED8                        15              MOV       DS,AX
0005 B8----              R       16              MOV       AX,DATA_2
0008 8EC0                        17              MOV       ES,AX
                                 18
000A 268B0E0000                  19              MOV       CX,BETA        ;ASSEMBLER CAUSES ES OVERRIDE
                                 20
000F 8A0E0000                    21              MOV       CL,ALPHA       ;NO OVERRIDE NECESSARY
                                 22
----                             23    CODE      ENDS
                                 24              END       START
```

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

EXAMPLE

```
LOC  OBJ                      SOURCE

                              NAME     SAMPLE3

----                          SHARED_DATA      SEGMENT
0000 (100                     BUFFER           DB       100 DUP (?)
     ??
     )
----                          SHARED_DATA      ENDS

----                          LOCAL_DATA       SEGMENT
0000 ????                     BETA             DW       ?
0002 ??                       ALPHA            DB       ?
----                          LOCAL_DATA       ENDS

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

0000 B8----            R      START:   MOV      AX,LOCAL_DATA
0003 8ED8                              MOV      DS,AX
0005 B8----            R               MOV      AX,SHARED_DATA
0008 8EC0                              MOV      ES,AX

000A 8B0E0000                          MOV      CX,BETA          ;NO OVERRIDE

000E 8A0E0200                          MOV      CL,ALPHA         ;NO OVERRIDE NECESSARY

0012 26880E0000                        MOV      BUFFER,CL        ;ASSEMBLER CAUSE ES OVERRIDE

----                          CODE     ENDS
                                       END      START
```

EXPLICIT SEGMENT OVERRIDE

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

```
                    NAME      SAMPLE
PRO                 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

                                1               NAME    SAMPLE5
----                            2     CODE1     SEGMENT
                                3               ASSUME CS:CODE1
0000 9A9090                     4     START:    CALL    WIZZY          ;Forward Reference to a FAR procedure.
*** ERROR #3, LINE #4, (PASS 2) INSTRUCTION SIZE BIGGER THAN PASS 1 ESTIMATE
0003 2E8B 1690                  5               MOV     DX,VAR1        ;Forward Reference to a variable not
*** ERROR #3, LINE #5, (PASS 2) INSTRUCTION SIZE BIGGER THAN PASS 1 ESTIMATE
                                6                                      ; accessible using DS register.
0007 F4                         7               HLT
0008 ????                       8     VAR1      DW      ?
----                            9     CODE1     ENDS
                               10
----                           11     CODE2     SEGMENT
                               12               ASSUME CS:CODE2
0000                           13     WIZZY     PROC    FAR
0000 00                        14               NOP
0001 CB                        15               RET
                               16     WIZZY     ENDP
----                           17     CODE2     ENDS
                               18               END START

ASSEMBLY COMPLETE, 2 ERRORS FOUND
```

ONE SOLUTION

```
LOC  OBJ                LINE   SOURCE

                          1            NAME    SAMPLE6
----                      2    CODE1   SEGMENT
                          3            ASSUME CS:CODE1
0000 9A0000----     R     4    START:  CALL    FAR PTR WIZZY   ;Forward Reference using PTR operator
0005 2E8B160B00           5            MOV     DX,CS:VAR1      ;Forward Reference using explicit
                                                               ; segment override.
000A F4                   6            HLT
000B ????                 7    VAR1    DW      ?
----                      8    CODE1   ENDS
                          9
----                     10    CODE2   SEGMENT
                         11            ASSUME CS:CODE2
0000                     12    WIZZY   PROC    FAR
0000 90                  13            NOP
0001 CB                  14            RET
                         15    WIZZY   ENDP
----                     16    CODE2   ENDS
                         17            END START

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

BETTER SOLUTION

```
LOC  OBJ                   LINE   SOURCE

                              1          NAME    SAMPLE7
----                          2   CODE2  SEGMENT
                              3          ASSUME CS:CODE2
0000                          4   WIZZY  PROC    FAR
0000 90                       5          NOP
0001 CB                       6          RET
                              7   WIZZY  ENDP
----                          8   CODE2  ENDS
                              9
----                         10   CODE1  SEGMENT
                             11          ASSUME CS:CODE1
0000 ????                    12   VAR1   DW      ?
0002 9A0000----        R     13   START: CALL    WIZZY        ;No Forward Reference, no problems.
0007 2E8B160000              14          MOV     DX,VAR1
000C F4                      15          HLT
----                         16   CODE1  ENDS
                             17          END START

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

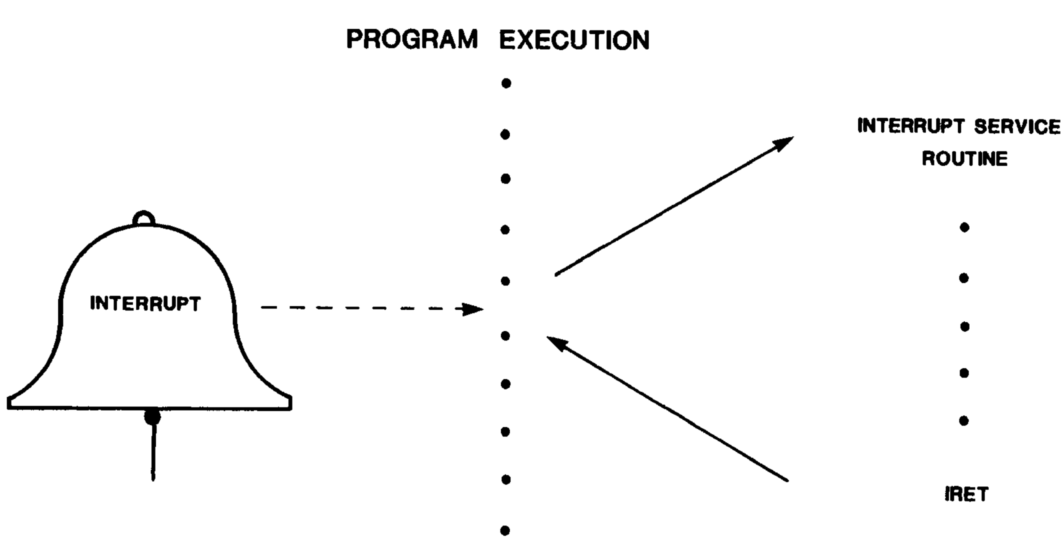
PROGRAMMED INPUT/OUTPUT

START DEVICE AND POLL FOR COMPLETION

PROGRAM
EXECUTION
•
•
•
START
DEVICE

INPUT STATUS ←
POLL AND WAIT
UNTIL DEVICE READY
•
•

INTERRUPT INPUT/OUTPUT



- INTERRUPTS ARE ASYNCHRONOUS EXTERNAL EVENTS

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

SERVICE ROUTINE

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

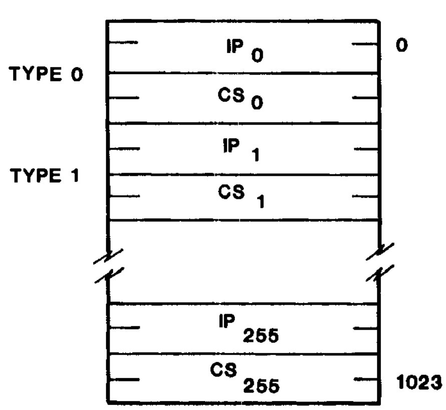


TABLE STARTS AT ABSOLUTE ADDRESS 0 IN MEMORY SPACE.

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

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

INTERNAL INTERRUPTS

	TYPE	CAUSED BY...
DIVIDE ERROR	0	QUOTIENT LARGER THAN DESTINATION
SINGLE STEP	1	MOST INSTRUCTIONS IF TF IS SET
iAPX 186,188 ONLY		
ARRAY BOUNDS TRAP	5	BOUND INSTRUCTION IF ARRAY INDEX IS OUTSIDE BOUNDARY
UNUSED OPCODE TRAP	6	CPU DIRECTED TO EXECUTE AN UNUSED OPCODE
ESCAPE OPCODE TRAP	7	CPU DIRECTED TO EXECUTE ESC OPCODE AND ESC TRAP SET IN RELOCATION REG

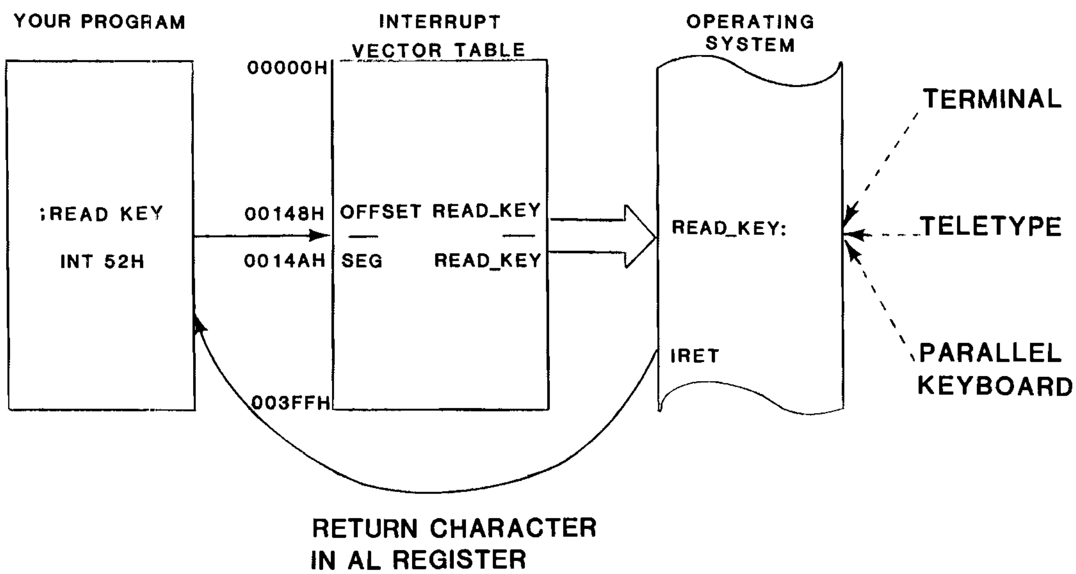
SOFTWARE INTERRUPTS

INT N	WHERE $\emptyset \leqslant N \leqslant 255$
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



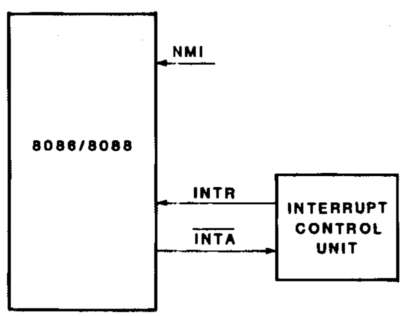
PROBLEM:

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,ØDH; CHARACTER RETURNED IN AL
```

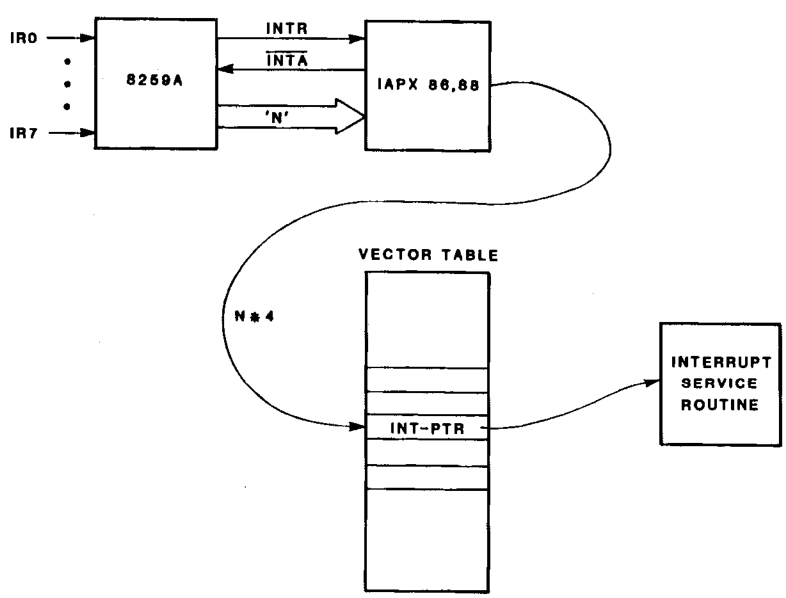
HARDWARE INTERRUPTS



NMI - NON-MASKABLE INTERRUPT
EDGE TRIGGERED
INVOKES TYPE 2 INTERRUPT

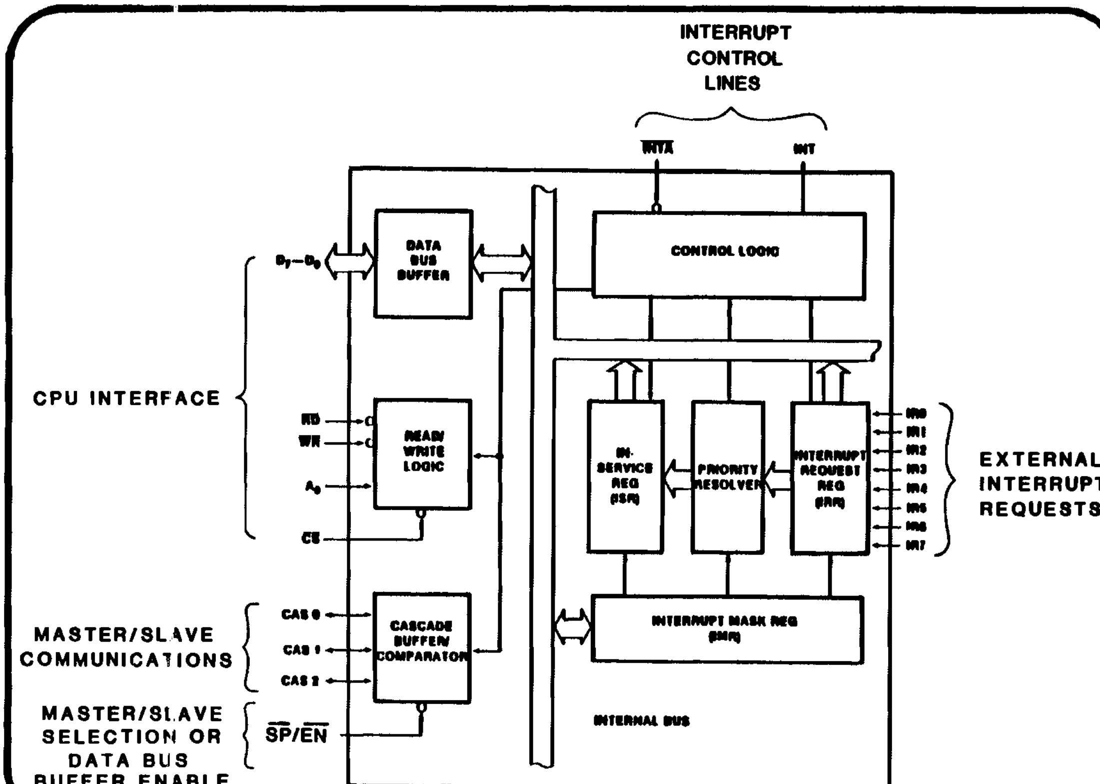
INTR AND $\overline{\text{INTA}}$ - MASKABLE INTERRUPT REQUEST (IF)
LEVEL TRIGGERED
EXTERNAL HARDWARE MUST SUPPLY INTERRUPT TYPE NUMBER
COMMUNICATIONS WITH EXTERNAL HARDWARE SET UP BY $\overline{\text{INTA}}$

INTERRUPT PROCESSING

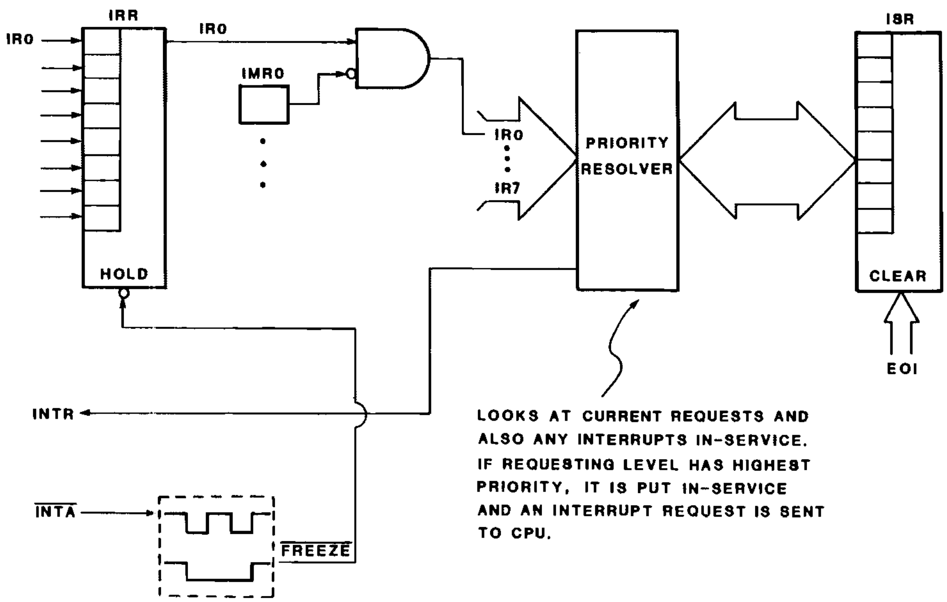


8259A PROGRAMMABLE INTERRUPT CONTROLLER

- 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.

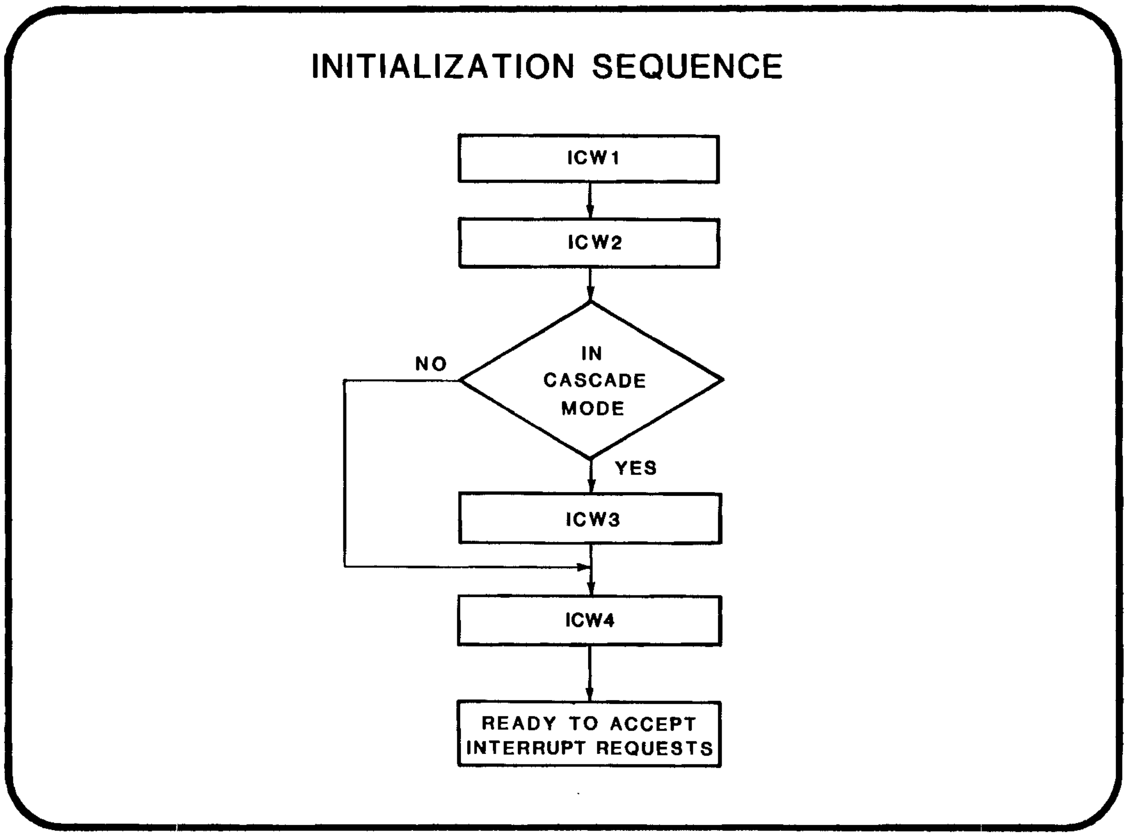


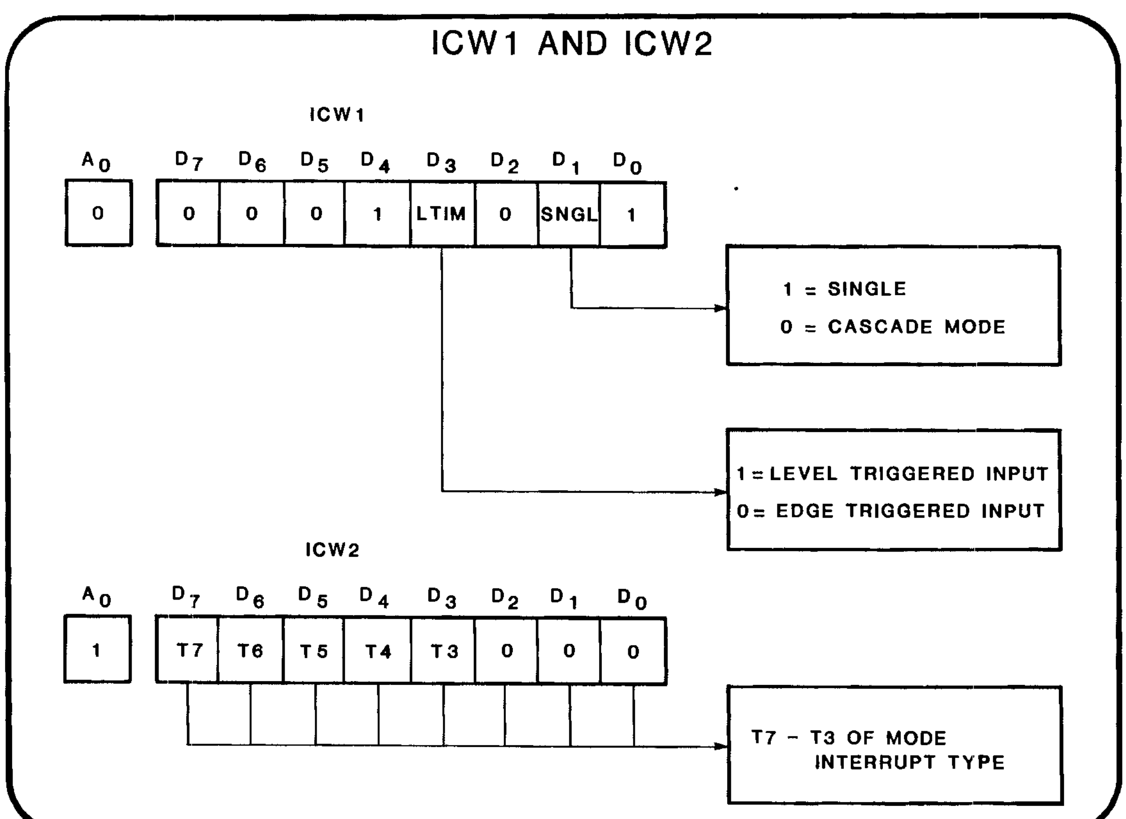
8259A OPERATION



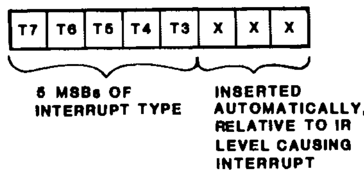
INITIALIZATION AND CONTROL

- 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
YES
ICW3
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 SNGL 1
1 = SINGLE
0 = CASCADE MODE
1 = LEVEL TRIGGERED INPUT
0 = EDGE TRIGGERED INPUT
ICW2
A0
D7 D6 D5 D4 D3 D2 D1 D0
1
T7 T6 T5 T4 T3 0 0 0
T7 – T3 OF MODE
INTERRUPT TYPE

INTERRUPT TYPE SELECTION



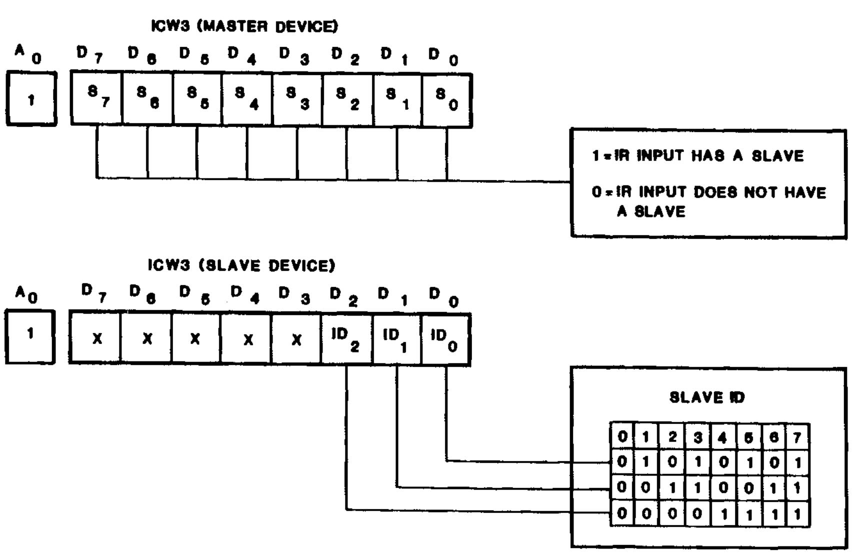
EXAMPLE:

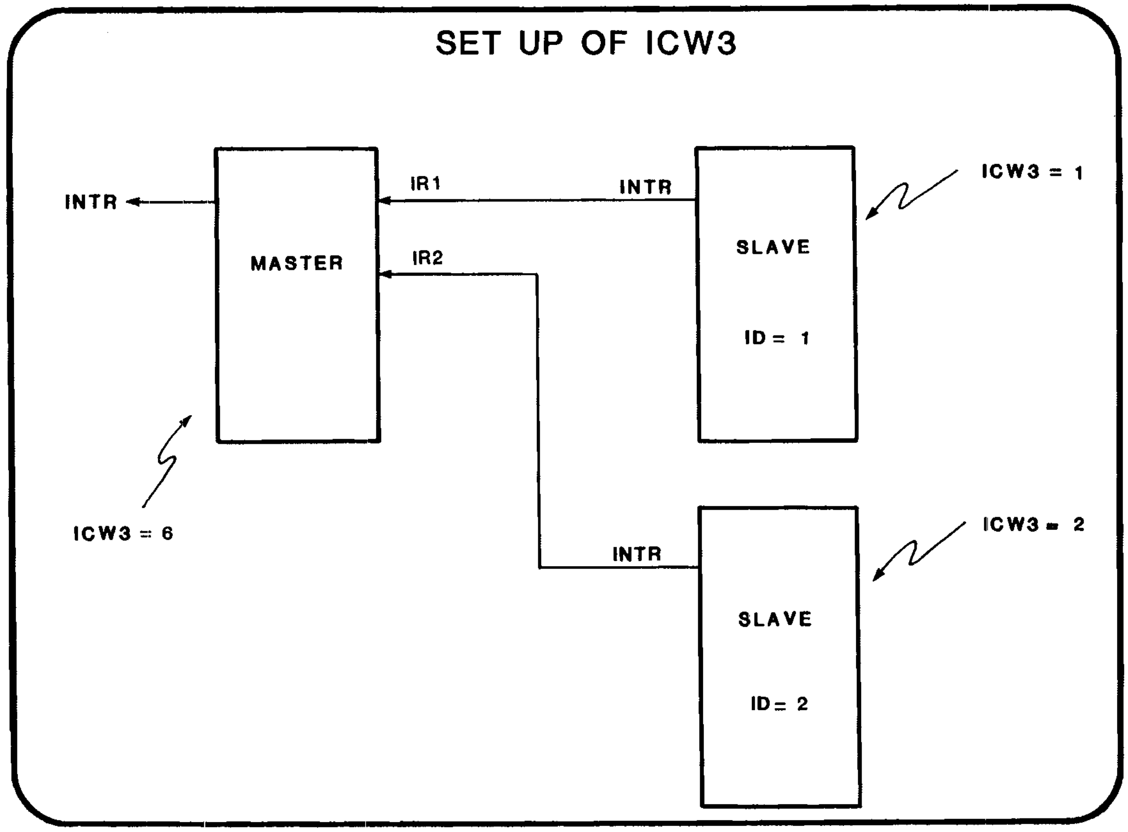
ASSUME INTERRUPT TYPES 32-39

	T7	T6	T5	T4	T3	X	X	X
IR0	0	0	1	0	0	0	0	0 ← USE THIS AS ICW2
IR1	0	0	1	0	0	0	0	1
IR2	0	0	1	0	0	0	1	0
IR3	0	0	1	0	0	0	1	1
IR4	0	0	1	0	0	1	0	0
IR5	0	0	1	0	0	1	0	1
IR6	0	0	1	0	0	1	1	0
IR7	0	0	1	0	0	1	1	1

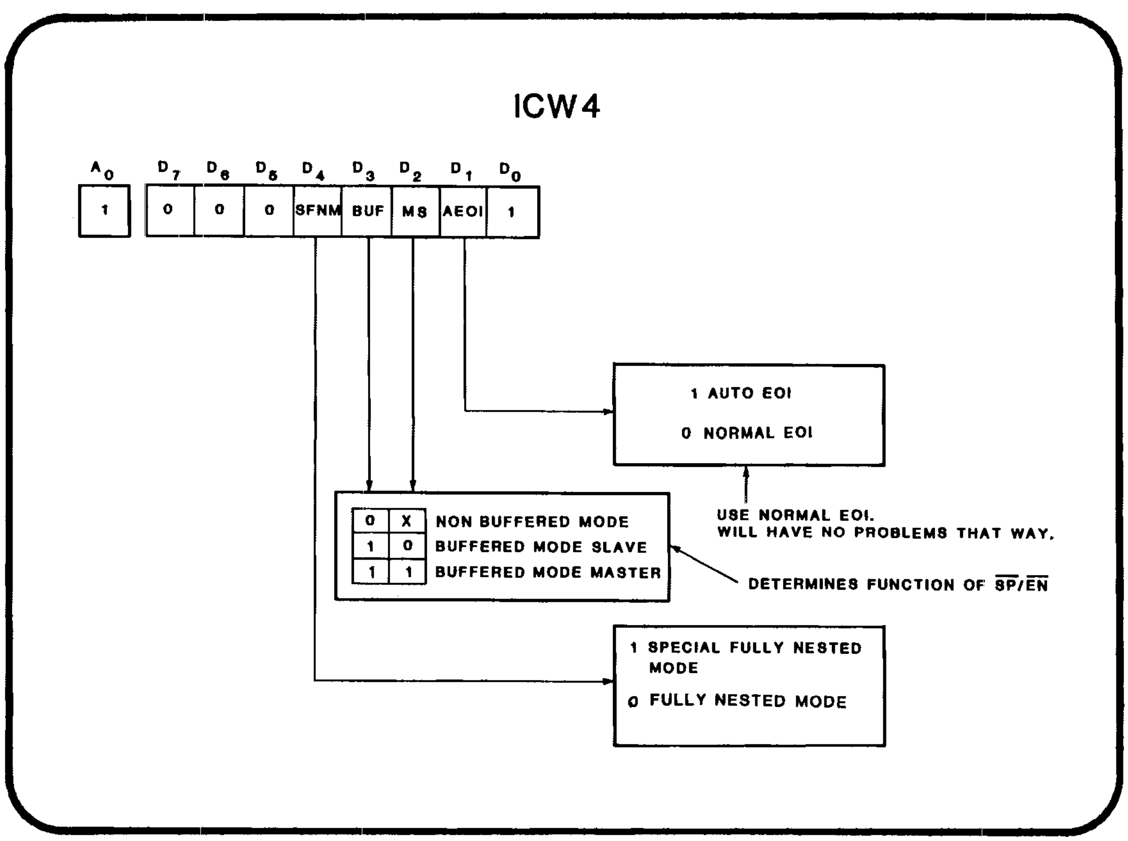
ICW3

- USED IN CASCADE MODE ONLY
- THE MASTER AND EACH SLAVE DEVICE HAVE DIFFERENT ICW3s.




SET UP OF ICW3
INTR
MASTER
IR1
IR2
INTR
SLAVE
ID = 1
ICW3 = 1
ICW3 = 6
INTR
SLAVE
ID= 2
ICW3 = 2

13-21


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.
0 X NON BUFFERED MODE
1 0 BUFFERED MODE SLAVE
1 1 BUFFERED MODE MASTER
DETERMINES FUNCTION OF SP/EN
1 SPECIAL FULLY NESTED MODE
0 FULLY NESTED MODE

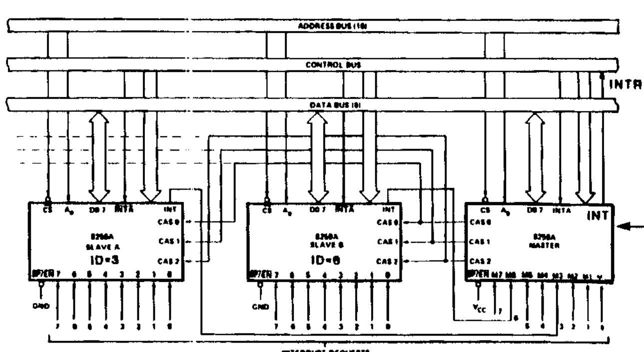
FULLY NESTED MODE

- ENTERED BY DEFAULT UPON INITIALIZATION

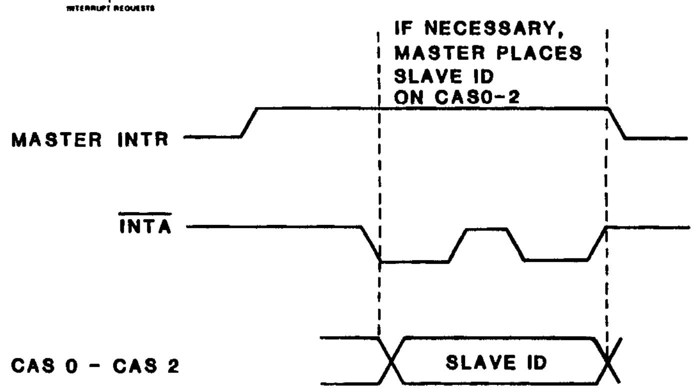
HIGHEST — IR0
IR1
IR2
IR3
IR4
IR5
IR6
LOWEST — IR7

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

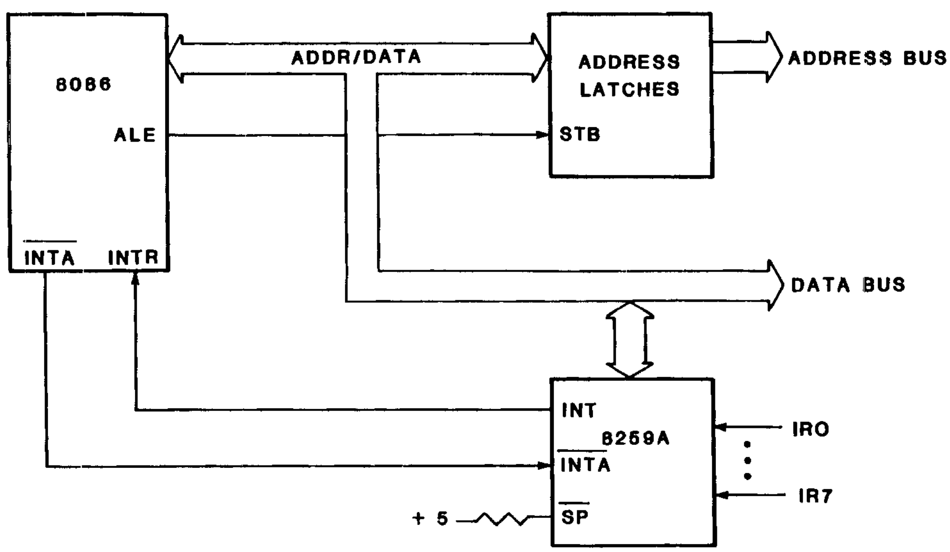
MASTER/SLAVE CONFIGURATION



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



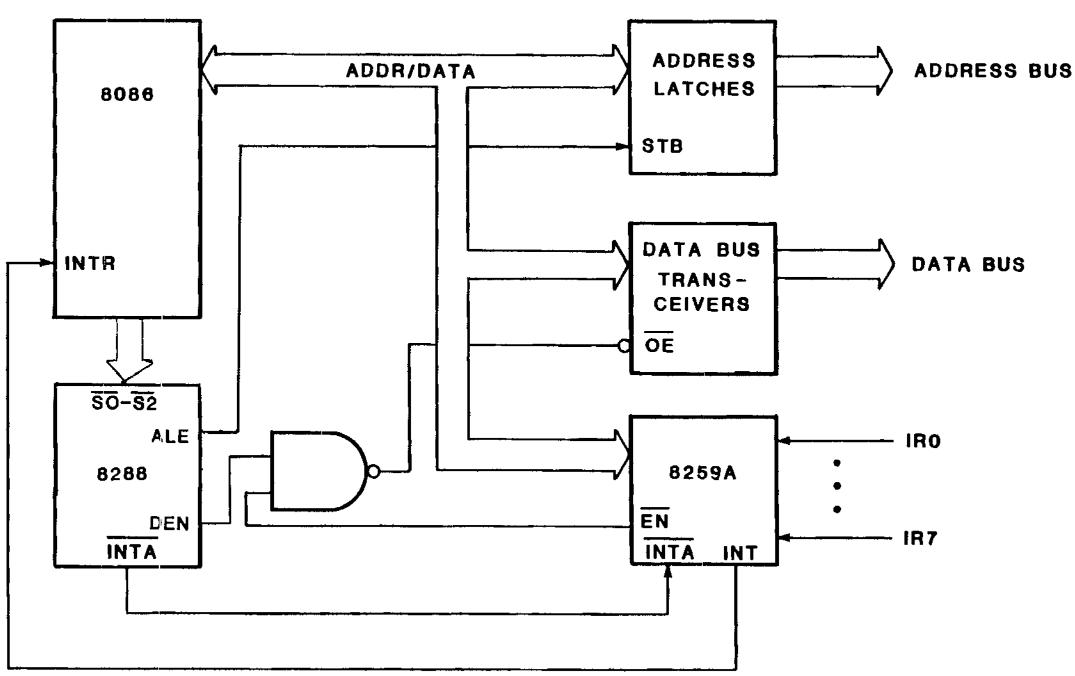
NON-BUFFERED MODE



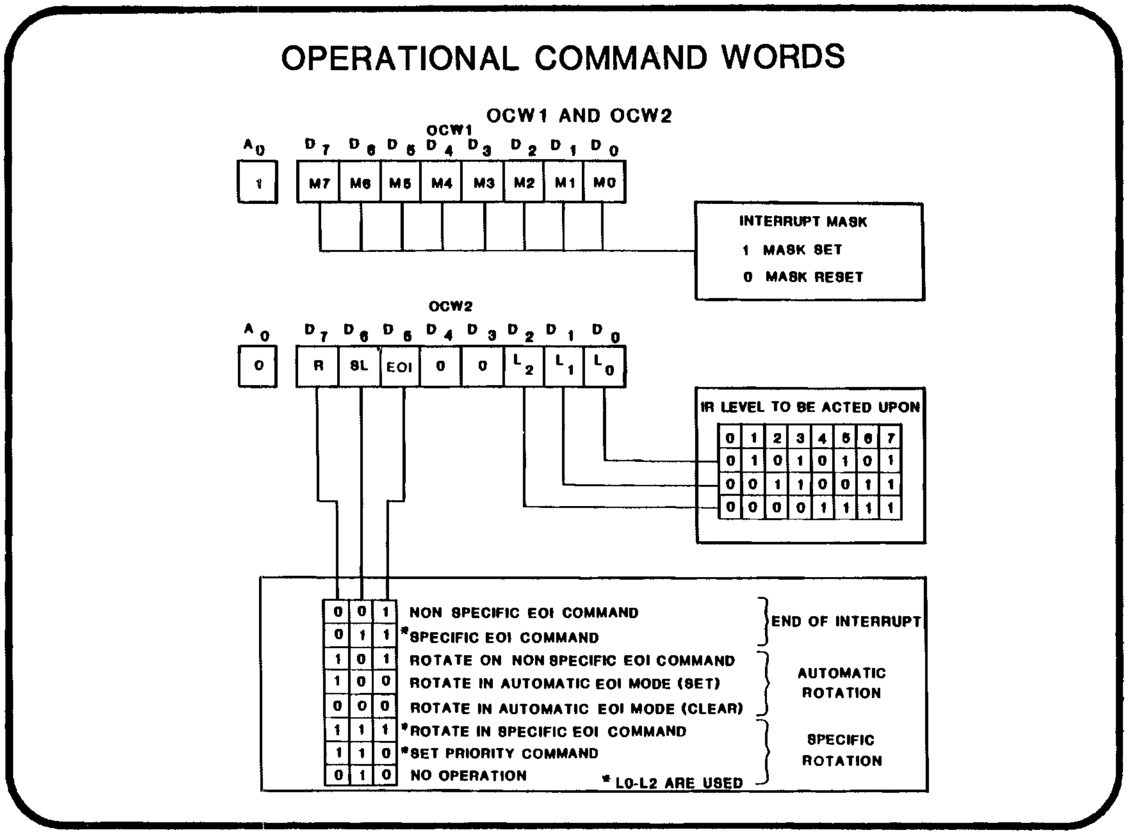
- $\overline{SP}$ IDENTIFIES 8259A AS MASTER OR SLAVE DEVICE

13-25

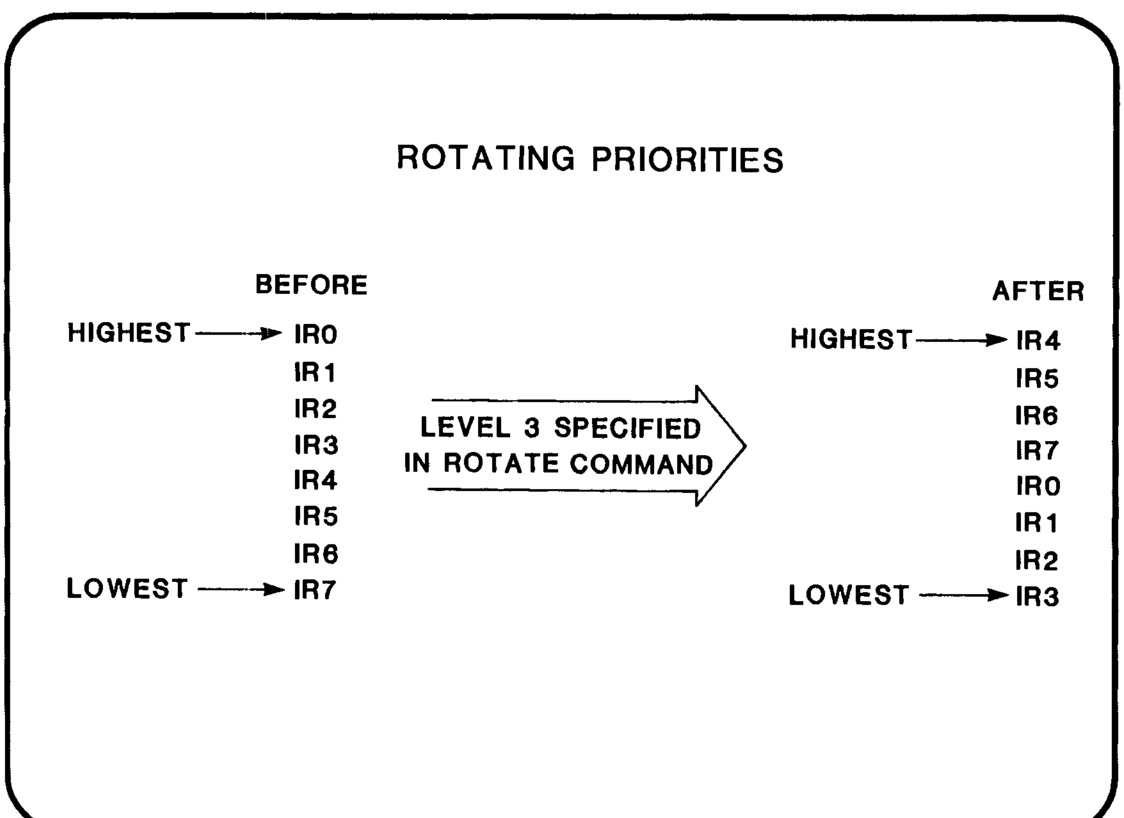
BUFFERED MODE

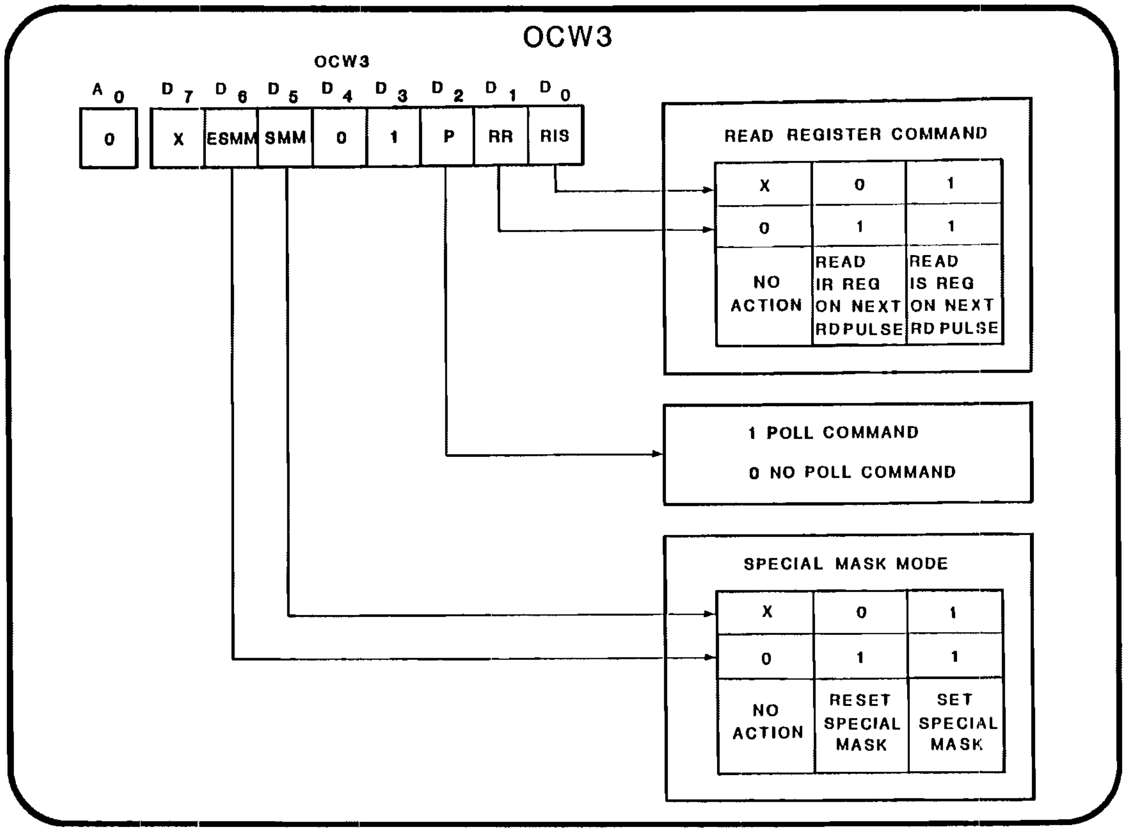


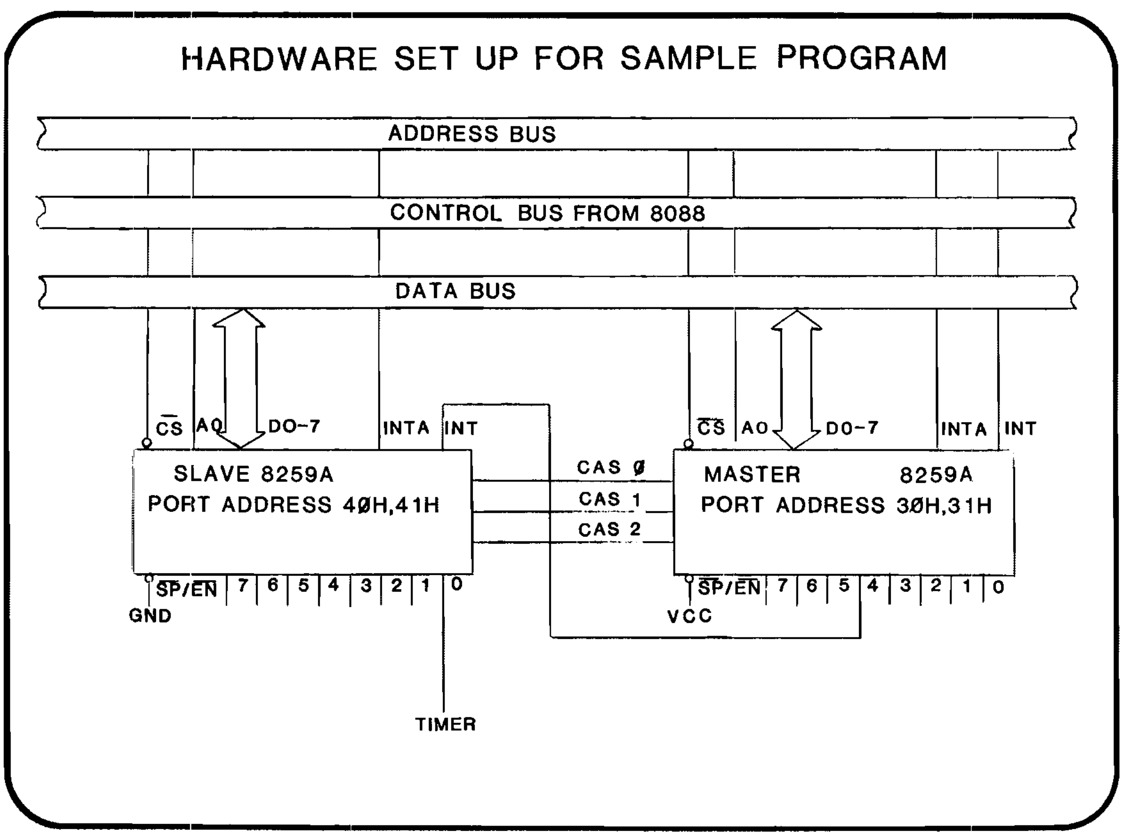
- $\overline{EN}$ USED TO CONTROL LOCAL DATA BUS


OPERATIONAL COMMAND WORDS
OCW1 AND OCW2
OCW1
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 M7 M6 M5 M4 M3 M2 M1 M0
INTERRUPT MASK
1 MASK SET
0 MASK RESET
OCW2
A0 D7 D6 D5 D4 D3 D2 D1 D0
0 R SL EOI 0 0 L2 L1 L0
IR 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 0 0 1 1
0 0 0 0 1 1 1 1
0 0 1 NON SPECIFIC EOI COMMAND
0 1 1 *SPECIFIC EOI COMMAND
END OF INTERRUPT
1 0 1 ROTATE ON NON SPECIFIC EOI COMMAND
1 0 0 ROTATE IN AUTOMATIC EOI MODE (SET)
0 0 0 ROTATE IN AUTOMATIC EOI MODE (CLEAR)
AUTOMATIC ROTATION
1 1 1 *ROTATE IN SPECIFIC EOI COMMAND
1 1 0 *SET PRIORITY COMMAND
0 1 0 NO OPERATION
SPECIFIC ROTATION
* L0-L2 ARE USED

13-27


ROTATING PRIORITIES
BEFORE
HIGHEST → IR0
IR1
IR2
IR3
IR4
IR5
IR6
LOWEST → IR7
LEVEL 3 SPECIFIED
IN ROTATE COMMAND
AFTER
HIGHEST → IR4
IR5
IR6
IR7
IR0
IR1
IR2
LOWEST → IR3


OCW3
OCW3
A0 D7 D6 D5 D4 D3 D2 D1 D0
0 X ESMM SMM 0 1 P RR RIS
READ REGISTER COMMAND
X 0 1
0 1 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 1
0 1 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 D0-7 INTA INT
SLAVE 8259A
PORT ADDRESS 4ØH,41H
SP/EN 7 6 5 4 3 2 1 0
GND
TIMER
CAS Ø
CAS 1
CAS 2
CS A0 D0-7 INTA INT
MASTER 8259A
PORT ADDRESS 3ØH,31H
SP/EN 7 6 5 4 3 2 1 0
VCC

SETTING UP TIMER INTERRUPT

```
INT VECTOR          SEGMENT AT 0
                    ORG 28H*4
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
```

SETTING 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,
        OUT     31H,AL   ;       NON-BUFFERED, NORMAL EOI

;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,
        OUT     41H,AL   ;       NON-BUFFERED, NORMAL EOI
        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

```
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

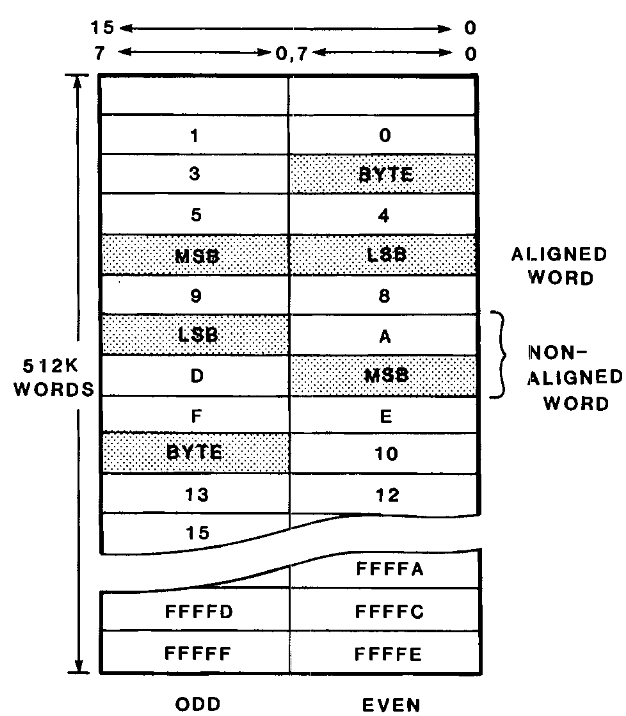
512 K WORDS OF MEMORY

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

14-1

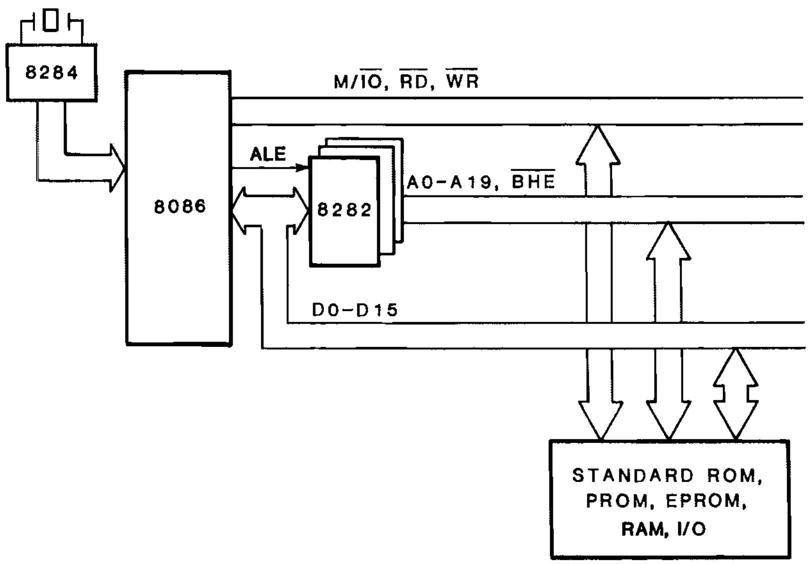
8086 MEMORY ORGANIZATION

* 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

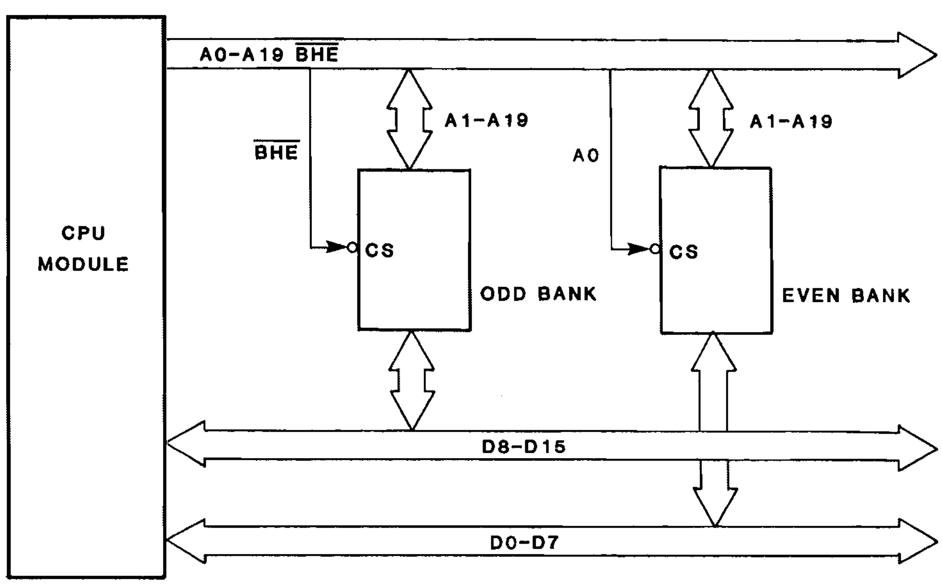


14-2

8086 MEMORY INTERFACING

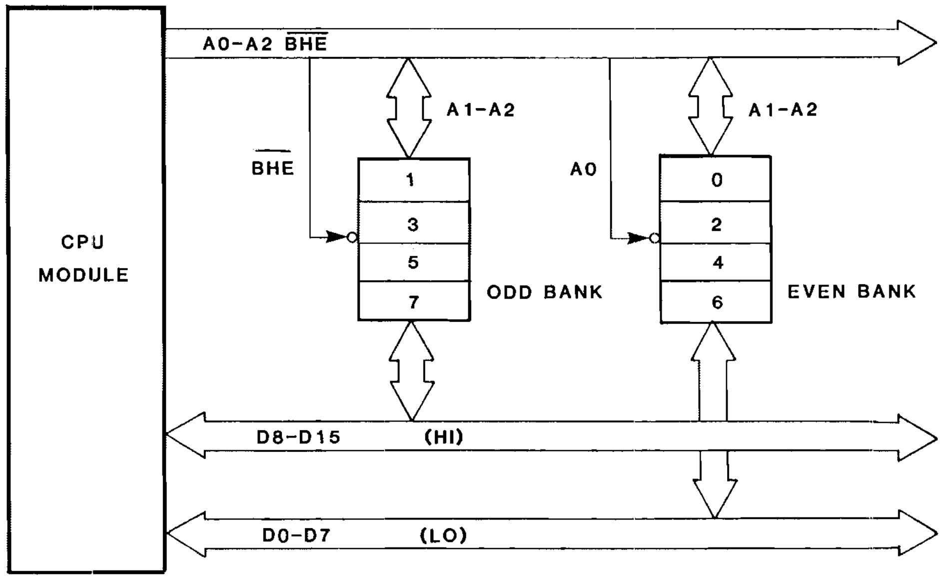


STANDARD MEMORY INTERFACE



BANK	SELECTED BY	CONNECTED TO
EVEN	A0	D0-D7
ODD	$\overline{BHE}$	D8-D15

BANK SELECTION

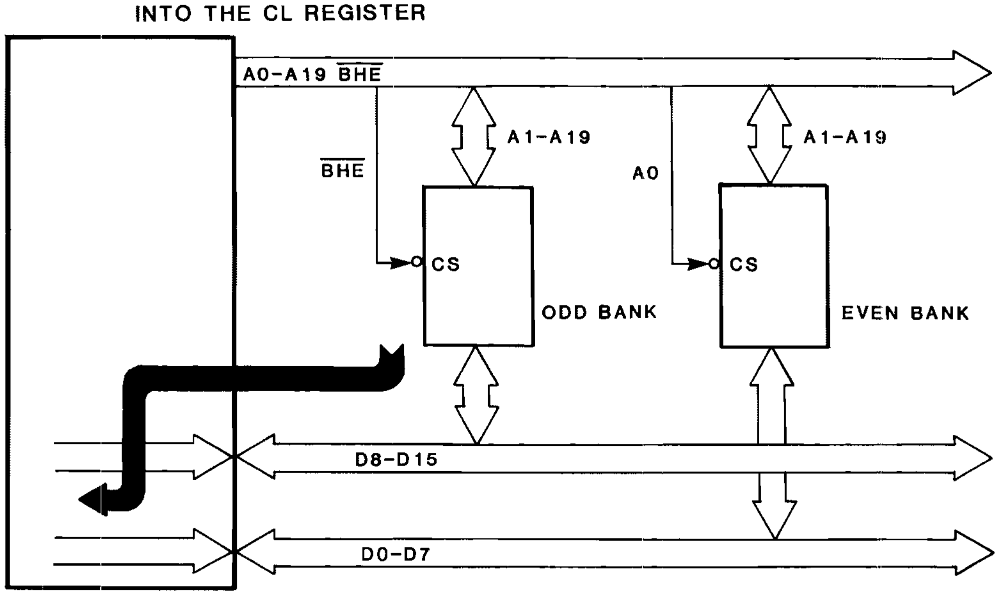


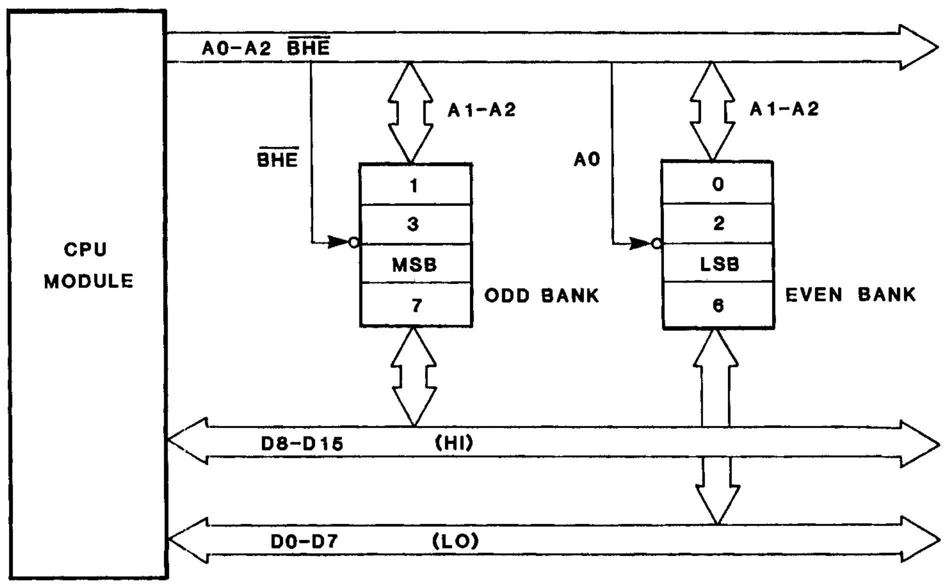
ADDRESS	A2	A1	A0	BHE	DATA BUS
BYTE@0					
BYTE@1					

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

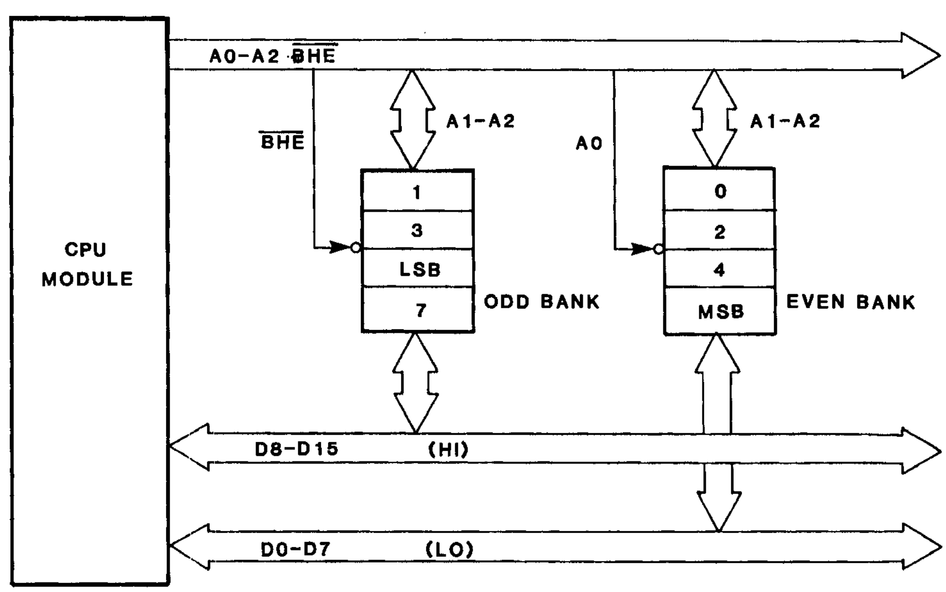




WHAT IS REQUIRED TO WRITE A WORD FROM MEMORY ADDRESS 4?

IS THIS AN ALIGNED WORD?

14-7

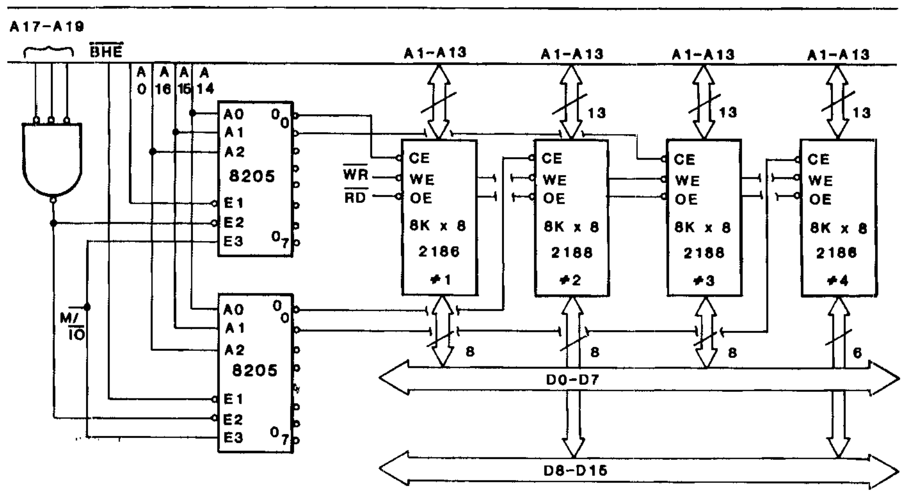


WHAT IS REQUIRED TO WRITE A WORD FROM MEMORY ADDRESS 5?

IS THIS AN ALIGNED WORD?

14-8

STATIC RAM INTERFACE



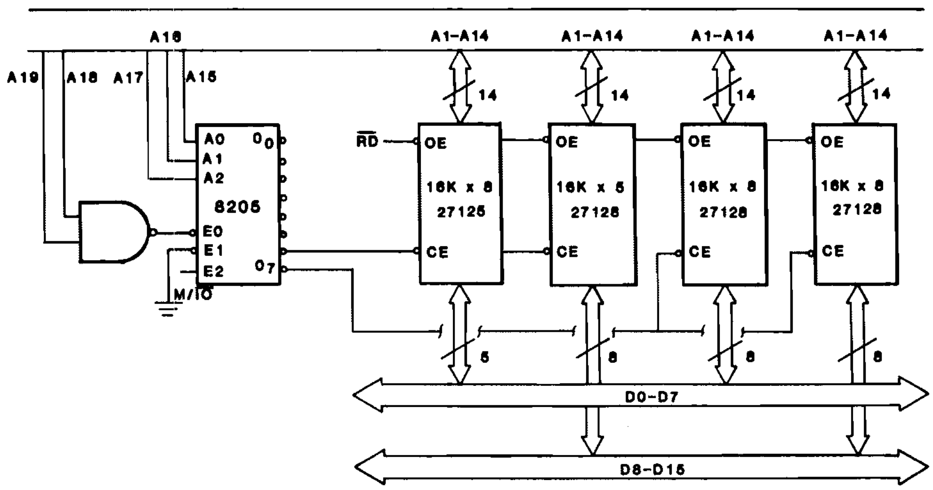
PROM MEMORY INTERFACING

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

ROM INTERFACE

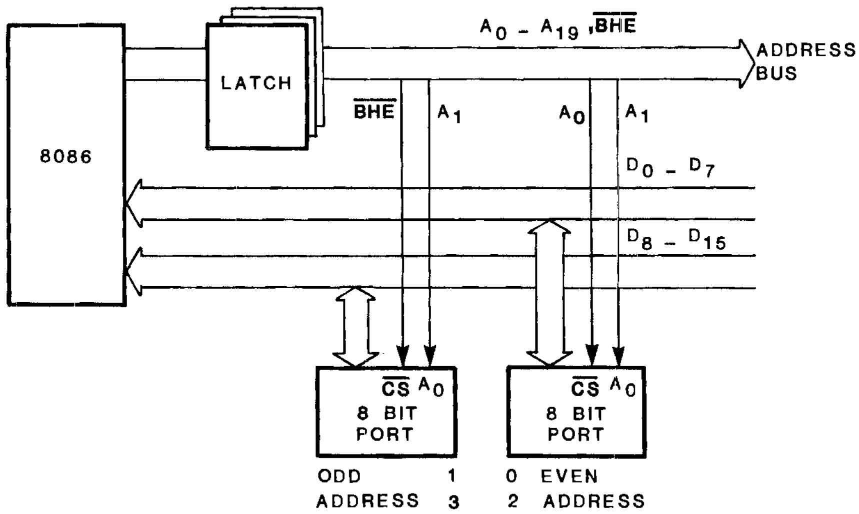


I/O DEVICE SELECTION

* 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-Tad)
- MEMORY ACCESS TIME IS PROPORTIONAL TO CLOCK SPEED

MEMORY

- REQUIRES FINITE PERIOD OF TIME TO RESPOND WITH DATA TO A VALID ADDRESS (Tacc)

CALCULATING PROCESSOR REQUIREMENTS

Tad=3*Tclcl−Tclav−Tdvcl (PROCESSOR ACCESS TIME)

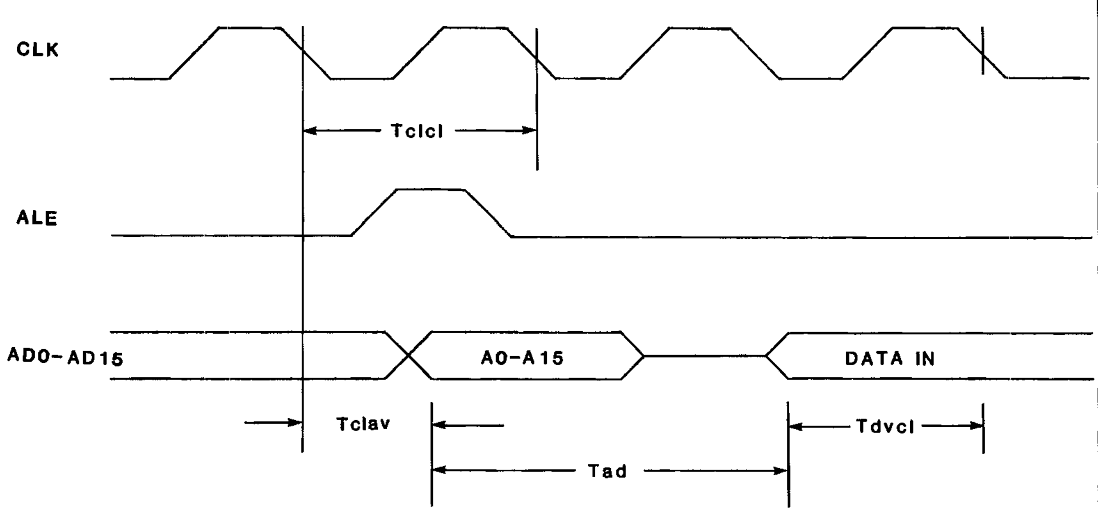
WHERE

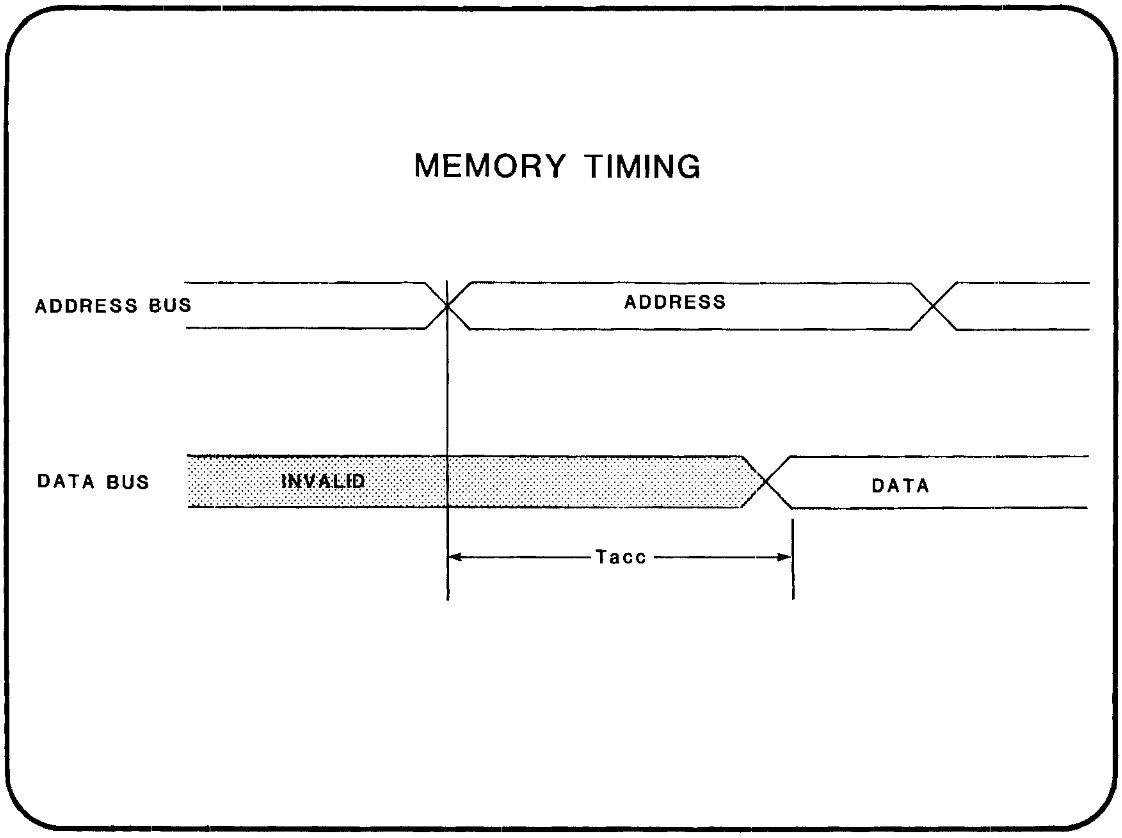
Tclcl = CLOCK PERIOD
Tclav = TIME PERIOD FROM CLOCK TO ADDRESS VALID
Tdvcl = SET UP TIME FOR DATA IN

FOR A MINIMUM MODE 8086

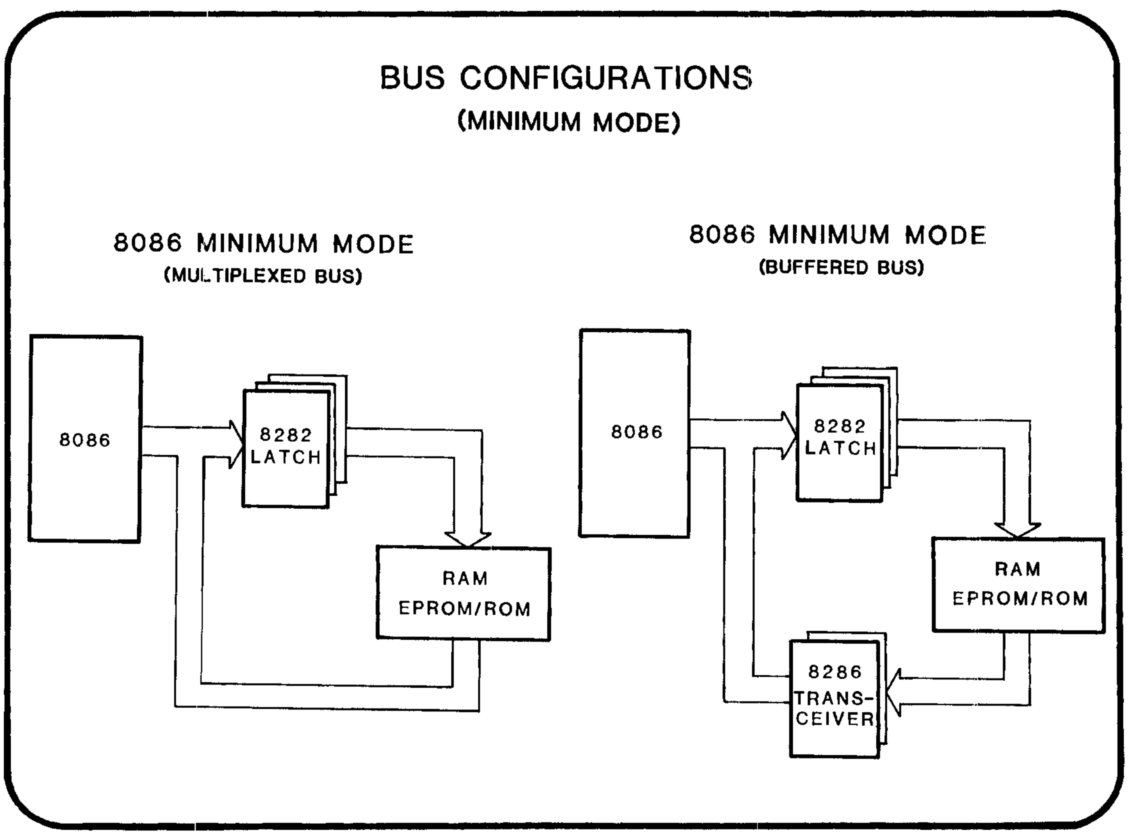
5 MHZ 8086	8 MHZ 8086-2
Tclcl = 200 nsec	Tclcl = 125 nsec
Tclav = 110 nsec	Tclav = 60 nsec
Tdvcl = 30 nsec	Tdvcl = 20 nsec
Tad =	Tad =

PROCESSOR REQUIREMENTS



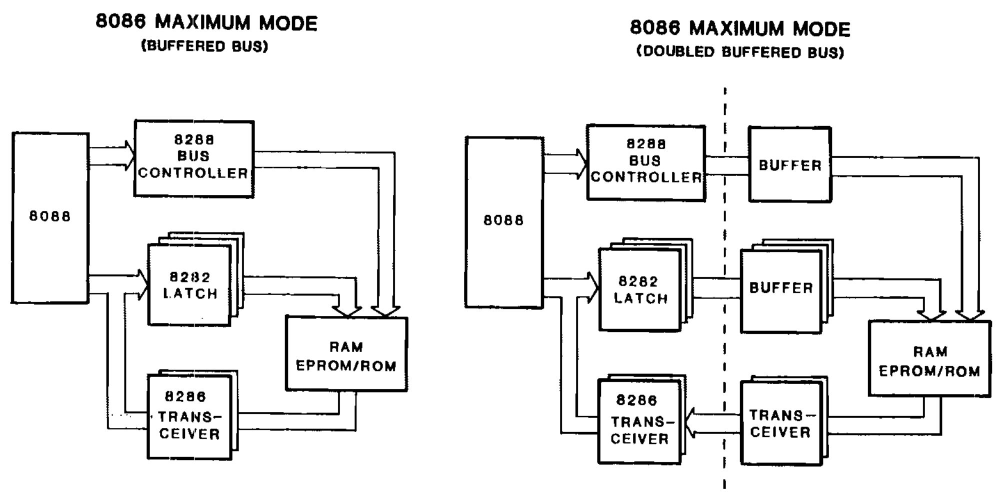

MEMORY TIMING
ADDRESS BUS
ADDRESS
DATA BUS
INVALID
DATA
Tacc

14-17


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
TRANS-
CEIVER

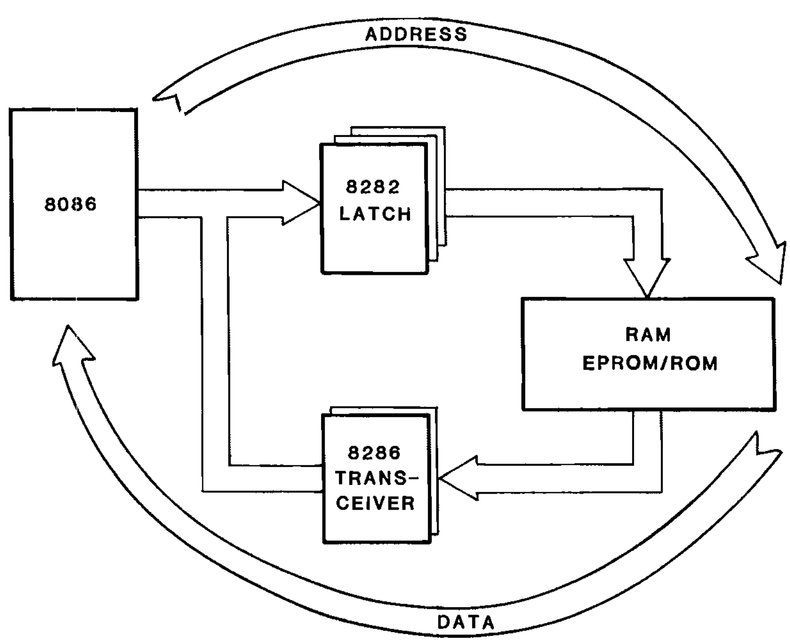
BUS CONFIGURATIONS

(MAXIMUM MODE)



14-19

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:

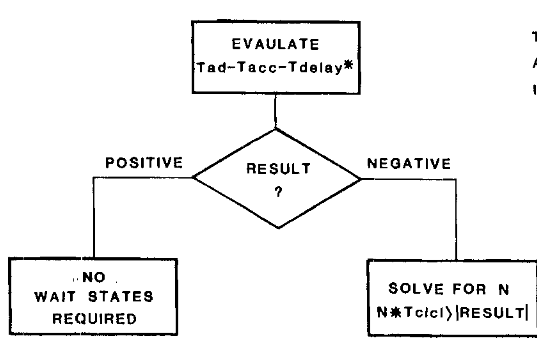
8282/8286	NON INVERTING	30 NSEC
8283/8287	INVERTING	22 NSEC
8205/LOGIC		18 NSEC

* 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:



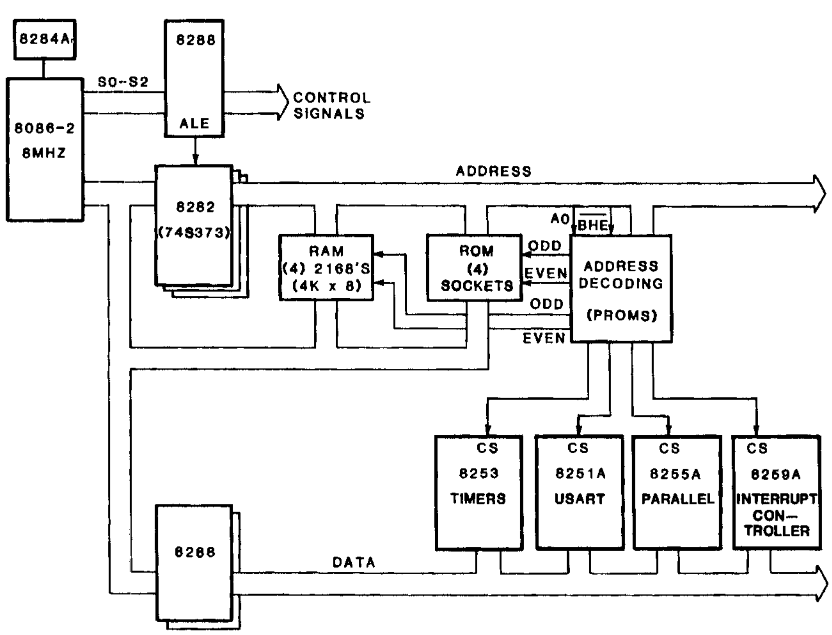
*Tdelay - TOTAL PROPAGATION DELAY FOR ALL BUFFER, TRANSCIEVERS, AND LATCHES IN ADDRESS AND DATA PATHS

8086 AND 8088 WAIT STATE CHART
5MHZ

MEMORY MATRIX
NO WAITS STATES

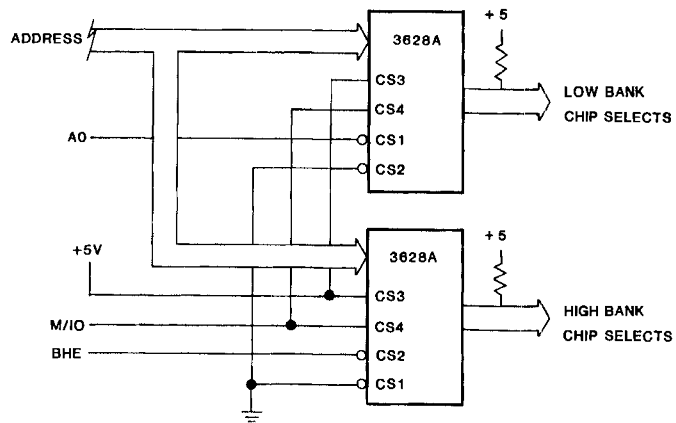
MODE	MIN MODE		MAX MODE	
BUS	MULTIPLEXED BUS	BUFFERED	BUFFERED	DOUBLE BUFFERED
STATIC RAM	2114-3 2141-5 2147 2168	2114-3 2141-5 2147 2168	2114-3 2141-5 2147 2168	2114-3 2141-5 2147 2168
EEPROM	2816	2816	2816	2816
EPROM	2716-2 2732A 2764	2716-2 2732A 2764	2716-2 2732A 2764	2732A 2764
DYNAMIC RAM	2118-7 2164	2118-7 2164	2118-7 2164	2118-7 2164

iSBC 86/05 DESIGN EXAMPLE



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

14-25

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 (Toe) 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)

JUMP TABLE
(INDIRECT JUMPS)

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.

ASSEMBLY CODE

```
LOC   OBJ               SOURCE

                                  NAME     JUMP_TABLE

 0000                   PORT      EQU      00H

----                    CODE      SEGMENT
                                  ASSUME   CS:CODE
0000 1C00               TABLE     DW       ROUTINE0,ROUTINE1,ROUTINE2,
0002 1E00
0004 2000
0006 2200               &                  ROUTINE3,ROUTINE4
0008 2400
000A E400               START:    IN       AL,PORT
000C 3C04                         CMP      AL,4
000E 770A                         JA       ERROR
0010 32E4                         XOR      AH,AH
0012 8BF8                         MOV      DI,AX
0014 D1E7                         SHL      DI,1
0016 2EFF25                       JMP      TABLE[DI]
0019 F4                 EXIT:     HLT
001A EBFD               ERROR:    JMP      EXIT

001C                    ROUTINE0:
001C EBFB                         JMP      EXIT
001E                    ROUTINE1:
001E EBF9                         JMP      EXIT
0020                    ROUTINE2:
0020 EBF7                         JMP      EXIT
0022                    ROUTINE3:
0022 EBF5                         JMP      EXIT
0024                    ROUTINE4:
0024 EBF3                         JMP      EXIT
----                    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]
              MOV   [DI] , AX           ; 12 MICROSECONDS PER WORD
              ADD   SI, 2
              ADD   DI, 2
              LOOP  BLOCK
```

STRING INSTRUCTIONS

* BYTE AND WORD ORIENTED ONE BYTE INSTRUCTIONS
* USE DS:SI AS SOURCE POINTER
* USE ES:DI AS DESTINATION POINTER

} AUTOMATICALLY INCREMENTS/DECREMENTS

* USE DIRECTION FLAG BIT

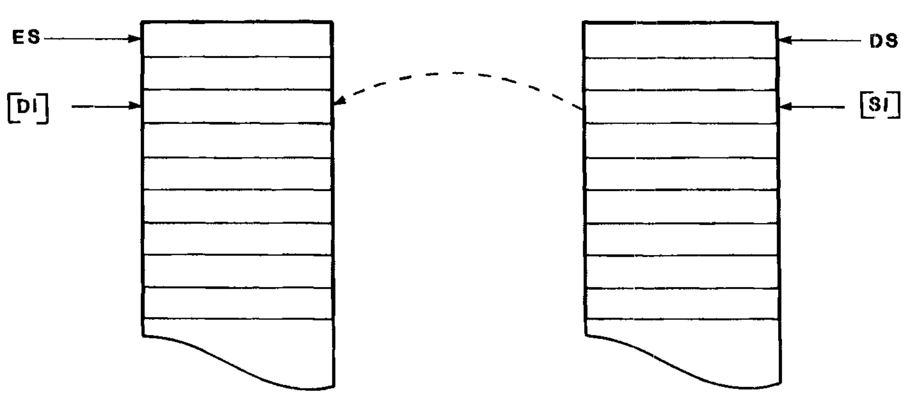
  DF = 0 PROCEEDS TO HIGHER MEMORY ADDRESS
  DF = 1 PROCEEDS TO LOWER MEMORY ADDRESS

* ADDITIONAL INSTRUCTION

  STD
  CLD

STRING INSTRUCTION

MOVSB ;[DI] ← [SI]
MOVSW ;SI ← SI + 1 (+2 FOR WORD)
;DI ← DI + 1 (+2 FOR WORD)



ASSUMING DF=0

OTHER STRING INSTRUCTIONS

CMPSB CMPSW	COMPARE TWO BLOCKS OF MEMORY
SCASB SCASW	SCAN FOR AN ITEM IN MEMORY
LODSB LODSW	LOAD AX/AL WITH STRING ITEM
STOSB STOSW	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:

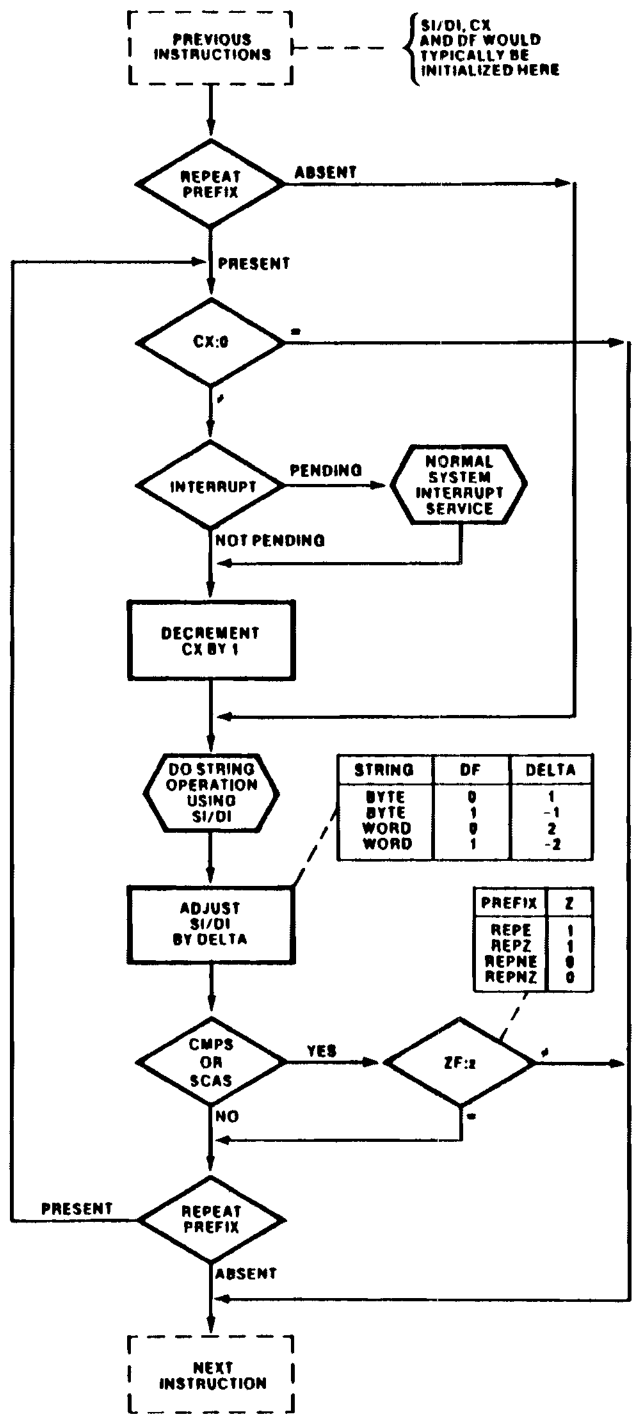
  REP { MOVS, STOS, LODS }

* FOR STRING INSTRUCTIONS THAT DO AFFECT THE FLAGS:

  - REPZ, REPE
  - REPNZ, REPNE

  { CMPS, SCAS }

OPERATION OF THE REP PREFIX



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

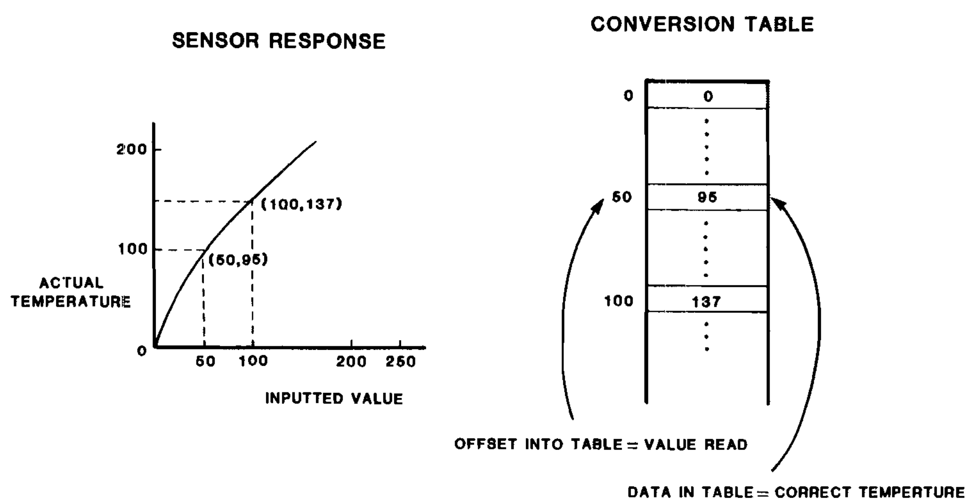
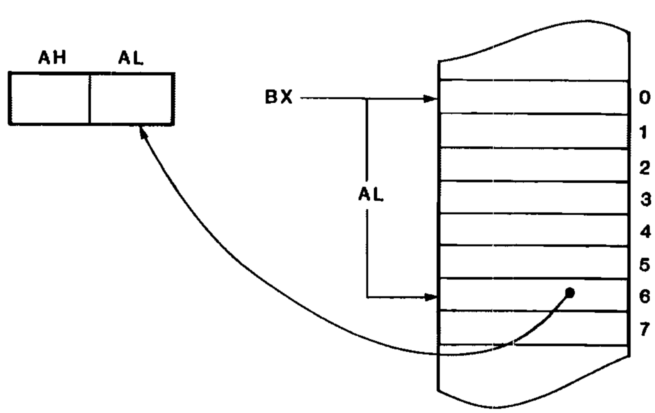


TABLE LOOK-UP

```
LOC  OBJ                  LINE   SOURCE

                                          NAME TABLE_LOOKUP
  000C                           SENSOR   EQU      12
----                             DATA1    SEGMENT
0000 00                          TABLE             DB       0,2,4,6,8,10,12,14,16,18,20,23,25
000D 1B                                            DB       27,29,30,32,34,35 ; ...........etc.
----                             DATA1    ENDS

----                             CODE1    SEGMENT
                                          ASSUME CS:CODE1,DS:DATA1
0000                             INPUT    PROC     FAR
0000 1E                                   PUSH     DS                ;Save registers except AX
0001 53                                   PUSH     BX
0002 B8----            R                  MOV      AX,DATA1          ;Initialize segment register
0005 8ED8                                 MOV      DS,AX
0007 8D1E0000                             LEA      BX,TABLE          ;The XLAT inst. requires BX to
                                                                     ;  point to the lookup table.
000B E40C                        AGAIN:   IN       AL,SENSOR         ;Get input from sensor.
000D D7                                   XLATB                      ;Linearized result is now in AL
000E 5B                                   POP      BX
000F 1F                                   POP      DS
0010 CB                                   RET
                                 INPUT    ENDP
----                             CODE1    ENDS
                                          END

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

SOLUTION

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	"	"	"	0
32H –	"	2	"	"	"	4
33H –	"	3	"	"	"	7
34H –	"	4	"	"	"	2
35H –	"	5	"	"	"	8
36H –	"	6	"	"	"	3
37H –	"	7	"	"	"	9
38H –	"	8	"	"	"	1
39H –	"	9	"	"	"	6

USE THE XLAT B INSTRUCTION. ASSUME THAT WHEN THE PROCEDURE IS CALLED THE ES AND SI REGISTERS CONTAIN THE ADDRESS OF THE BUFFER AND THE CX REGISTER CONTAINS THE NUMBER OF THE CHARACTERS IN THE BUFFER.

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

STRING AND XLATB INSTRUCTIONS (EXAMPLES)

- PAGE 3-191, iAPX 86/88, 186/188 USER'S MANUAL

RELATED TOPICS ...

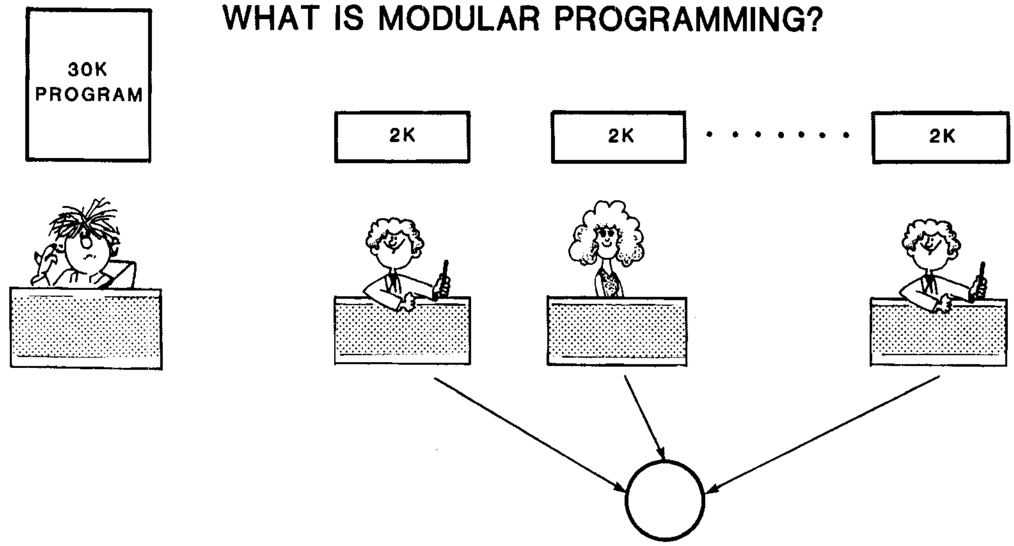
THERE ARE MORE 8086 INSTRUCTIONS THAT ARE NOT DISCUSSED IN THIS WORKSHOP. IT WOULD BE A GOOD IDEA TO LEAF THROUGH THE COMPLETE LIST IN CHAPTER 6 OF THE ASM86 LANGUAGE REFERENCE MANUAL.

CHAPTER 16

MODULAR PROGRAMMING

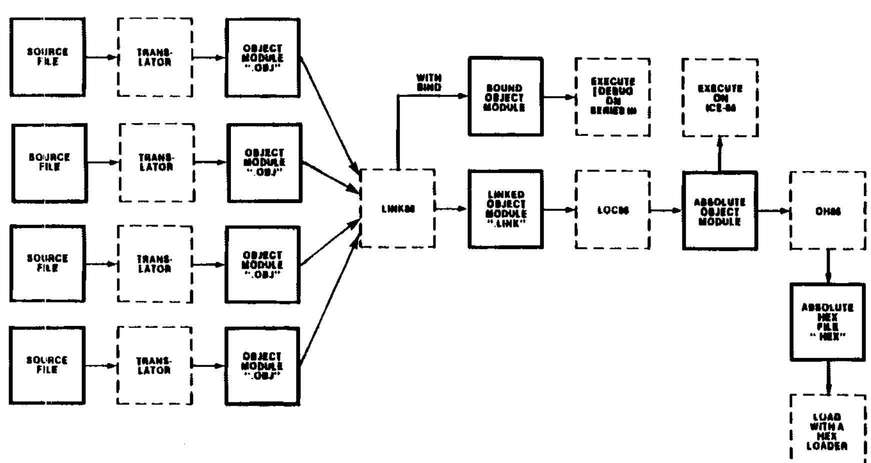
- PUBLIC DECLARATIVE
- EXTRN DECLARATIVE
- COMBINING SEGMENTS
- LINK86
- LOC86

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
SEGA    SEGMENT
        ASSUME CS:SEGA
          .
          .
        CALL     PROCA
          .
          .
SEGA    ENDS
        END
```

```
        NAME  MOD_B
SEGB    SEGMENT
        ASSUME CS:SEGB
          .
          .
PROCA   PROC     FAR
          .
          .
        RET
PROCA   ENDP
SEGB    ENDS
        END
```

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

THE ANSWER:

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

```
        NAME  MOD_A
        EXTRN   PROCA:FAR
SEGA    SEGMENT
        ASSUME  CS:SEGA
          .
          .
        CALL     PROCA
          .
          .
SEGA    ENDS
        END
```

```
        NAME  MOD_B
        PUBLIC  PROCA
SEGB    SEGMENT
        ASSUME CS:SEGB
          .
          .
PROCA   PROC     FAR
          .
          .
PROCA   ENDP
SEGB    ENDS
        END
```

PUBLIC AND EXTERNAL DECLARATIVES

PUBLIC MAKES A NAME AVAILABLE TO OTHER MODULES.

EXTRN MAKES NAMES DEFINED ELSEWHERE USABLE IN THIS MODULE.

EXAMPLES:

```
PUBLIC      XYZ, WP, ERS
EXTRN       FOO: BYTE *
```

* ATTRIBUTES

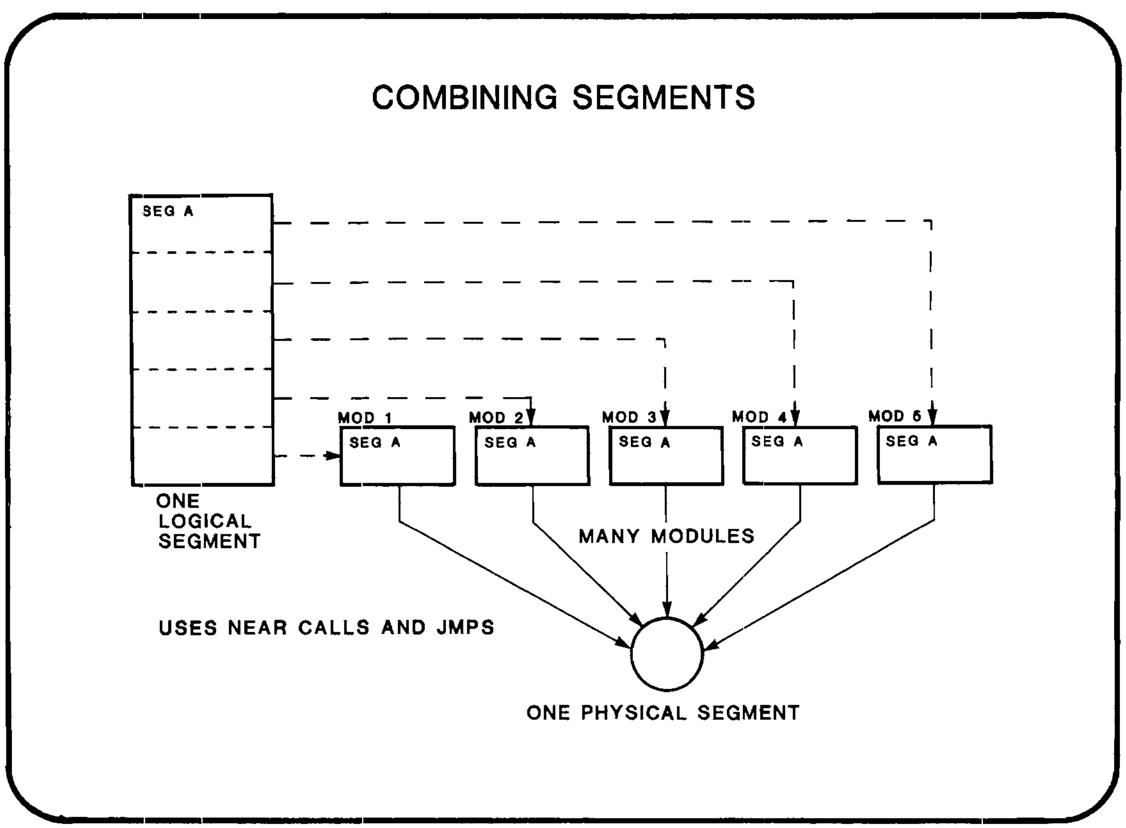
NEAR, FAR
BYTE, WORD, DWORD
ABS

MAIN PROGRAM

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

LOC  OBJ                  LINE     SOURCE

                                 ;THIS ROUTINE INPUTS AND OUTPUTS TO THE I/O BOX OF THE MDS.
                                 ;IT USES AN EXTERNAL DELAY ROUTINE TO DELAY 1 SECOND
                                 ;BETWEEN A INPUT AND A SUBSEQUENT OUTPUT.

                                    NAME DEMO

                                            EXTRN   DELAY:FAR        ;MUST DECLARE TYPE OF EXTRN

----                                STACK   SEGMENT
0000 (10                                    DW      10 DUP (?)
     ????
     )
  0014                              TOP     EQU     THIS WORD
----                                STACK   ENDS

----                                CODE    SEGMENT
                                            ASSUME CS:CODE,SS:STACK
  2710                              SECOND  EQU     10000       ;DELAY PARAMETER FOR 1 SECOND

0000 B8----          R              START:  MOV     AX,STACK
0003 8ED0                                   MOV     SS,AX
0005 8D261400                               LEA     SP,TOP
0009 BA1027                                 MOV     DX,SECOND
000C E400                           LOOP_:  IN      AL,0
000E 52                                     PUSH    DX          ;PUSH DELAY ONTO STACK
000F A0000----       E                      CALL    DELAY
0014 E600                                   OUT     0,AL
0016 EBF4                                   JMP     LOOP_

----                                CODE    ENDS
                                            END     START
```

SUB PROGRAM

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

LOC  OBJ                  LINE     SOURCE

                            1   ;THIS IS THE DELAY ROUTINE.  THE ROUTINE WILL DELAY N*
                            2   ;100 MICRO SECONDS.  N IS PASSED IN ON THE STACK.
                            3
                            4      NAME    DEMO2
                            5
                            6      PUBLIC  DELAY            ;DECLARE DELAY AS A GLOBAL NAME
                            7
----                        8      PRO     SEGMENT
                            9              ASSUME CS:PRO
0000                       10      DELAY   PROC    FAR          ;FAR PROC.; PARAMETER AT BP+6
0000 51                    11              PUSH    CX           ;SAVE CX, NOW PARAMETER AT BP+8
0001 50                    12              PUSH    AX           ;SAVE AX, NOW PARAMETER AT BP+10
0002 55                    13              PUSH    BP
0003 8BEC                  14              MOV     BP,SP
0005 8B460A                15              MOV     AX,[BP+10]   ;GET "N" OFF STACK.
0008 0BC0                  16              OR      AX,AX        ;CHECK FOR 0
000A 7407                  17              JZ      EXIT         ;IF 0, QUIT PROCEDURE
000C B178                  18      LOOP_:  MOV     CL,78H       ;TIME DELAY FOR 100 MICRO SECOND
000E D2E9                  19              SHR     CL,CL
0010 48                    20              DEC     AX
0011 75F9                  21              JNZ     LOOP_
0013 5D                    22      EXIT:   POP     BP
0014 58                    23              POP     AX
0015 59                    24              POP     CX
0016 CA0200                25              RET     2
                           26      DELAY   ENDP
----                       27      PRO     ENDS
                           28              END
```

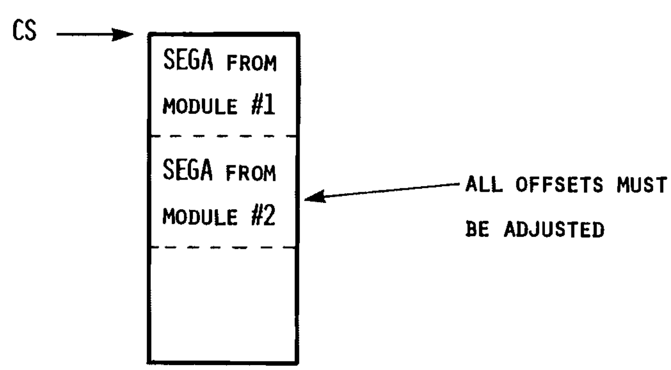

COMBINING SEGMENTS
SEG A
MOD 1
SEG A
MOD 2
SEG A
MOD 3
SEG A
MOD 4
SEG A
MOD 6
SEG A
ONE
LOGICAL
SEGMENT
MANY MODULES
USES NEAR CALLS AND JMPS
ONE PHYSICAL SEGMENT

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.

MAIN PROGRAM

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

LOC  OBJ                    SOURCE

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

                            NAME DEMO

                                    EXTRN   DELAY:NEAR        ;MUST DECLARE TYPE OF EXTRN

----                        STACK   SEGMENT
0000 (10                            DW      10 DUP (?)
     ????
     )
  0014                      TOP     EQU     THIS WORD
----                        STACK   ENDS

----                        CODE    SEGMENT PUBLIC
                                    ASSUME CS:CODE,SS:STACK
  2710                      SECOND  EQU     10000             ;DELAY PARAMETER FOR 1 SECOND

0000 B8----          R      START:  MOV     AX,STACK
0003 8ED0                           MOV     SS,AX
0005 8D261400                       LEA     SP,TOP
0009 BA1027                         MOV     DX,SECOND
000C E400                   LOOP_:  IN      AL,0
000E 52                             PUSH    DX                ;PUSH DELAY ONTO STACK
000F E80000          E              CALL    DELAY
0012 E600                           OUT     0,AL
0014 EBF6                           JMP     LOOP_

----                        CODE    ENDS
                                    END     START
```

SUB PROGRAM

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

LOC  OBJ              LINE    SOURCE

                         1    ;THIS IS THE DELAY ROUTINE.  THE ROUTINE WILL DELAY N*
                         2    ;100 MICRO SECONDS.  N IS PASSED ON THE STACK.
                         3
                         4    NAME     DEMO2
                         5
                         6             PUBLIC  DELAY        ;DELAY IS A PUBLIC NAME
                         7
----                     8    CODE     SEGMENT PUBLIC       ;BOTH SEGMENTS SHARE SAME NAME
                         9             ASSUME CS:CODE
0000                    10    DELAY    PROC    NEAR         ;NEAR PROC.; PARAMETER AT BP+4
0000 51                 11             PUSH    CX           ;SAVE CX, NOW PARAMETER AT BP+6
0001 50                 12             PUSH    AX           ;SAVE AX, NOW PARAMETER AT BP+8
0002 55                 13             PUSH    BP
0003 8BEC               14             MOV     BP,SP
0005 8B4608             15             MOV     AX,[BP+8]    ;GET "N" OFF STACK FOR DELAY
0008 0BC0               16             OR      AX,AX        ;CHECK FOR 0
000A 7407               17             JZ      EXIT         ;IF 0, QUIT PROCEDURE
000C B178               18    LOOP_:   MOV     CL,78H       ;TIME DELAY FOR 100 MICRO SECOND
000E D2E9               19             SHR     CL,CL
0010 48                 20             DEC     AX
0011 75F9               21             JNZ     LOOP_
0013 5D                 22    EXIT:    POP     BP
0014 58                 23             POP     AX
0015 59                 24             POP     CX
0016 C20200             25             RET     2
                        26    DELAY    ENDP
----                    27    CODE     ENDS
                        28             END
```

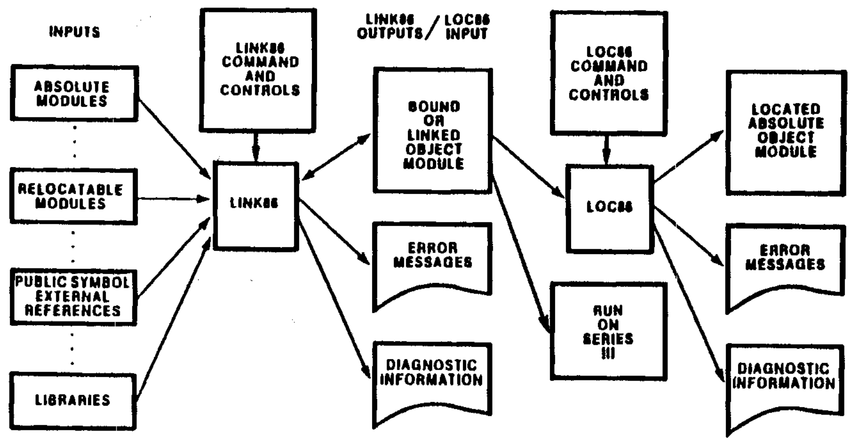
REFERENCING EXTERNAL DATA (ONE ITEM)

```
            NAME      MOD1
DATA        SEGMENT
            PUBLIC    BUFFER,WBUFFER
BUFFER      DB        100 DUP(?)
WBUFFER     DW        100 DUP(?)
DATA        ENDS
            END
-----------------------------------------------------------
            NAME      MOD2
            EXTRN     BUFFER:BYTE
CODE        SEGMENT
            ASSUME    CS:CODE,DS:SEG BUFFER
BEGIN:      MOV       AX,SEG BUFFER
            MOV       DS,AX
             .
             .
            MOV       AL,BUFFER[SI]
             .
             .
CODE        ENDS
            END       BEGIN
```


REFERENCING EXTERNAL DATA (MULIPLE ITEMS)

```
            NAME      MOD1
DATA        SEGMENT   PUBLIC
            PUBLIC    BUFFER,WBUFFER
BUFFER      DB        100 DUP(?)
WBUFFER     DW        100 DUP(?)
DATA        ENDS
            END
-----------------------------------------------------------
            NAME      MOD3
DATA        SEGMENT   PUBLIC
            EXTRN     BUFFER:BYTE,WBUFFER:WORD
DATA        ENDS
CODE        SEGMENT
            ASSUME    CS:CODE,DS:DATA
BEGIN:      MOV       AX,DATA
            MOV       DS,AX
             .
            MOV       AL,BUFFER[SI]
            MOV       WBUFFER,DX
             .
CODE        ENDS
            END       BEGIN
```

DEVELOPMENT CYCLE WITH LINK86 AND LOC86

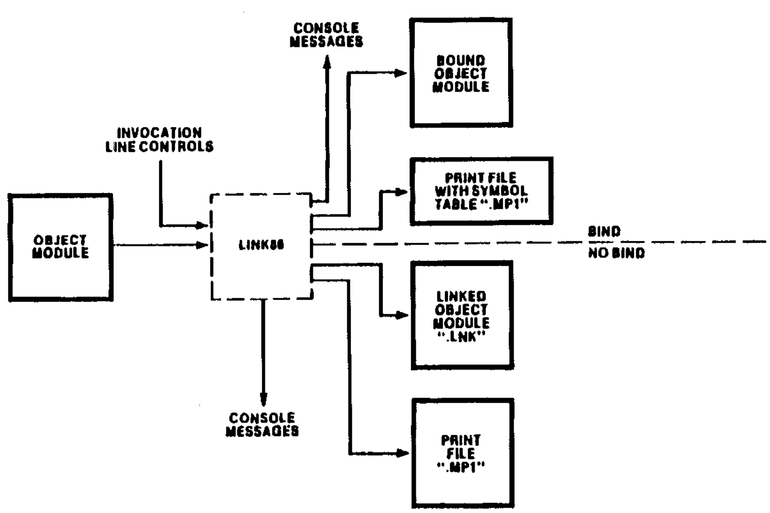


LINK86 SYNTAX

-RUN LINK86 FILENAME,FILENAME[,...][TO FILENAME][NO MAP]

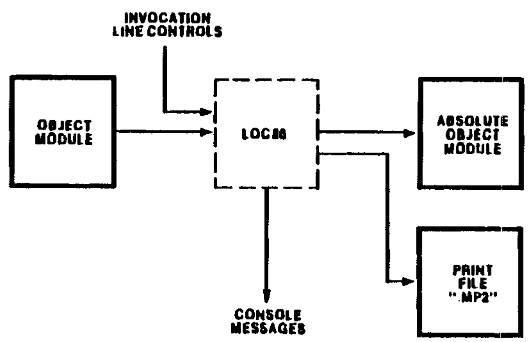
[PRINT (FILENAME)]

[[BIND [ORDER(SEGMENTS(SEGNAME),...]]



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[1] LINK86 MESSAG.OBJ,READ.OBJ,PRINT.OBJ

RUN LOC86 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

```
            NAME MODA

DATA        SEGMENT
USEFUL_DATA     DB      ?
DATA        ENDS
A_CODE      SEGMENT
            ASSUME  CS:A_CODE
HANDY       PROC    FAR
            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
```

FOR MORE INFORMATION ...

LINK86

- iAPX 86,88 FAMILY UTILITIES USER'S GUIDE

LOC86

- iAPX 86,88 FAMILY UTILITIES POCKET REFERENCE CARD

COMBINING SEGMENTS , PUBLIC AND EXTRN DECLARATIVE

- CHAPTER 2, ASM86 LANGUAGE REFERENCE MANUAL

RELATED TOPICS ...

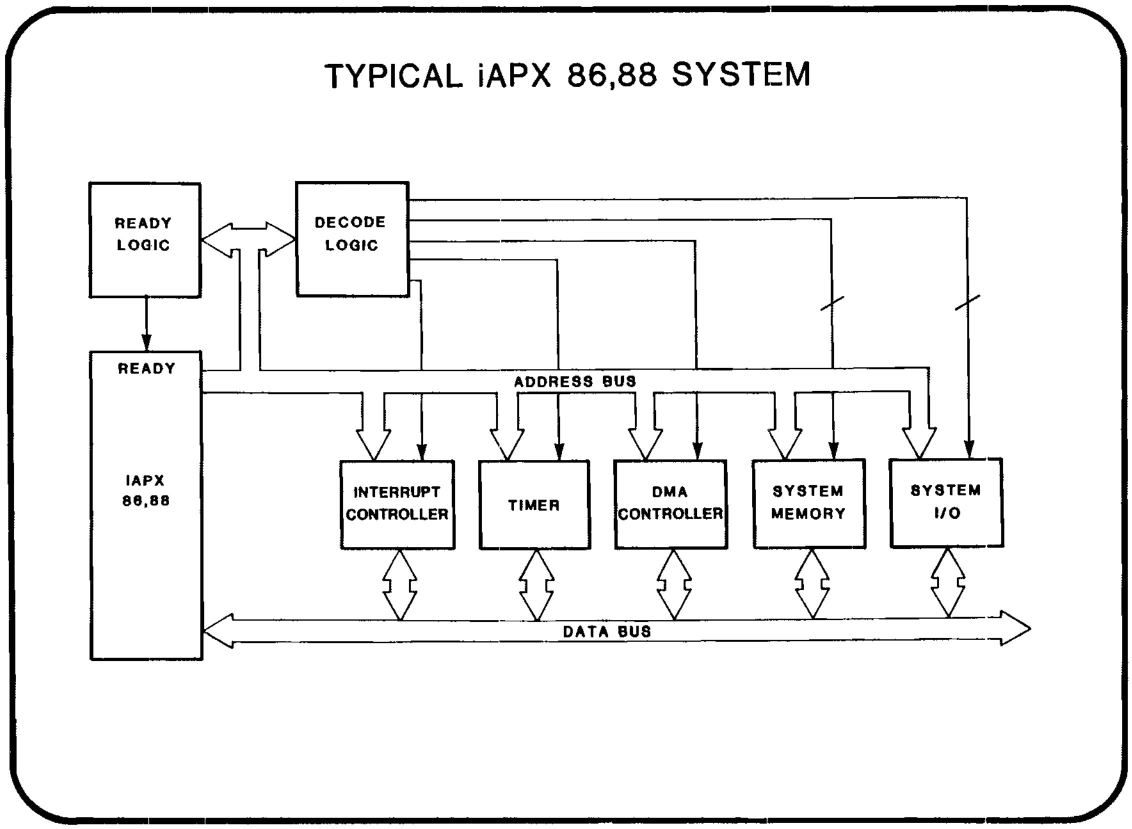
LIB86 IS A UTILITY PROGRAM TO MANAGE COLLECTIONS OF DEBUGGED MODULES. (SEE THE iAPX 86,88 FAMILY USER'S GUIDE)

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.

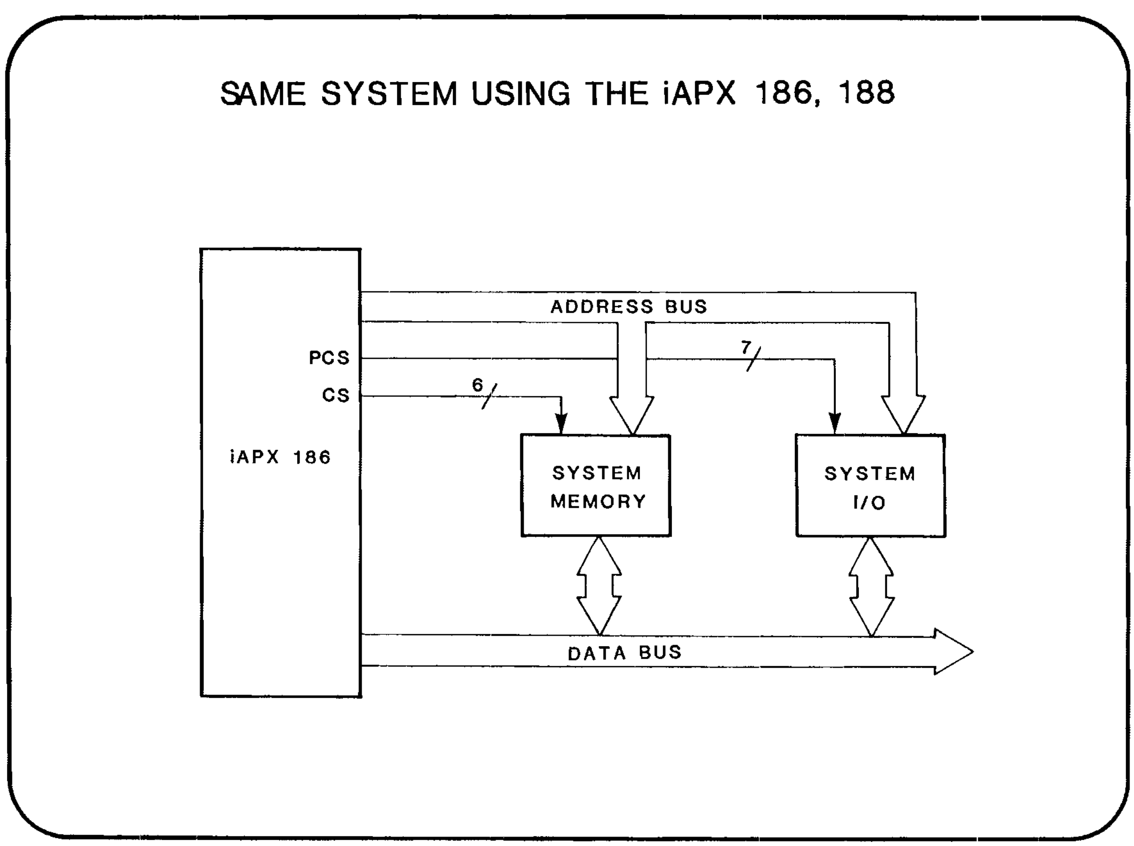
CHAPTER 17

INTRODUCTION TO THE iAPX 186, 188 MICROPROCESSOR

- DESCRIPTION
- ENHANCEMENTS
- NEW INSTRUCTIONS
- PERIPHERALS


TYPICAL iAPX 86,88 SYSTEM
READY LOGIC
DECODE LOGIC
READY
IAPX 86,88
ADDRESS BUS
INTERRUPT CONTROLLER
TIMER
DMA CONTROLLER
SYSTEM MEMORY
SYSTEM I/O
DATA BUS

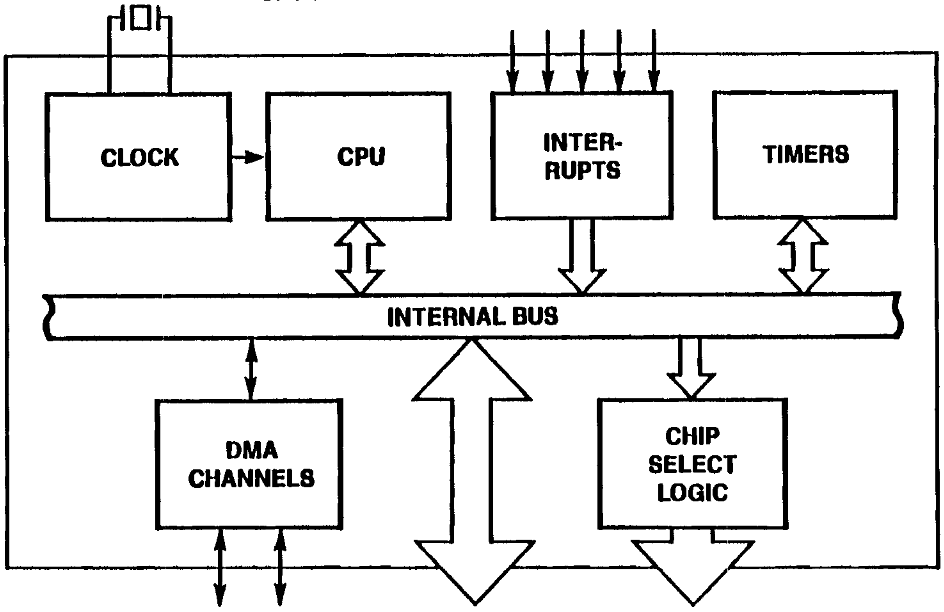
17-1


SAME SYSTEM USING THE iAPX 186, 188
ADDRESS BUS
PCS
CS
6
7
iAPX 186
SYSTEM MEMORY
SYSTEM I/O
DATA BUS

17-2

iAPX 186 BLOCK DIAGRAM

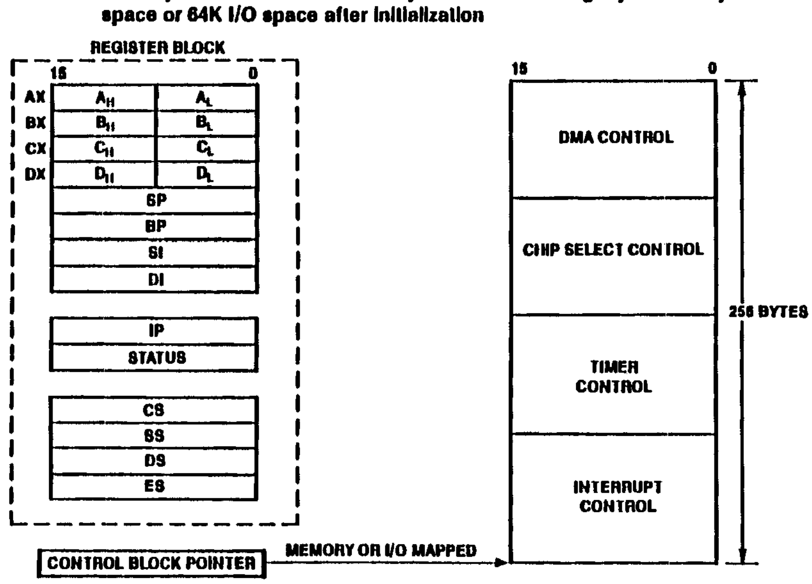
"A CPU BOARD ON A SINGLE SILICON CHIP"



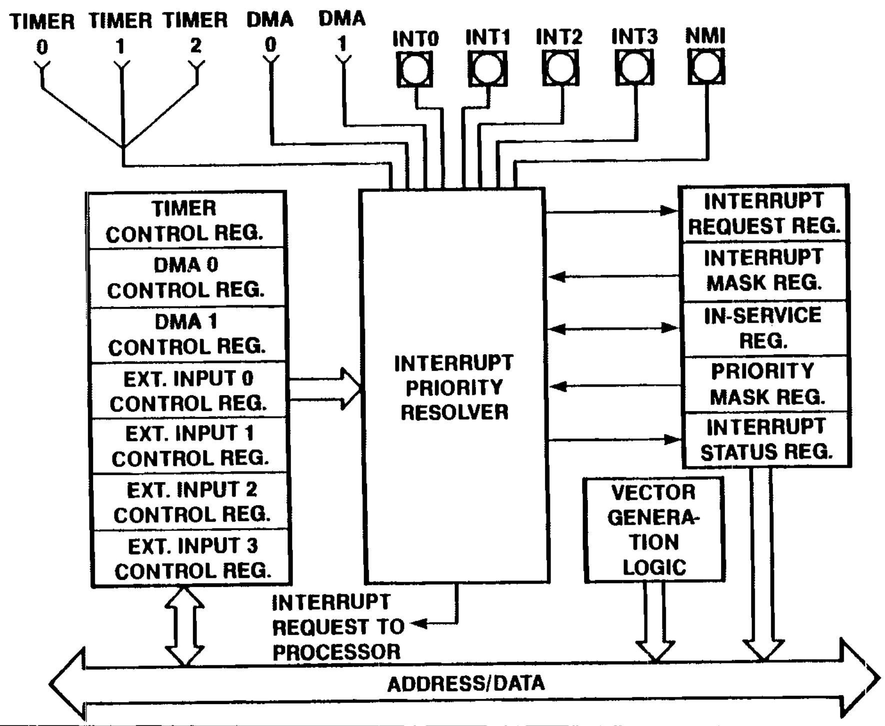
Combines 10 of the most common iAPX 86 system components into one

iAPX 186 PERIPHERAL INITIALIZATION

- On-chip peripherals are controlled via a block of 16-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 (0FF00H – 0FFFFH)
- 256 byte block is relocatable anywhere in the 1 megabyte memory space or 64K I/O space after initialization



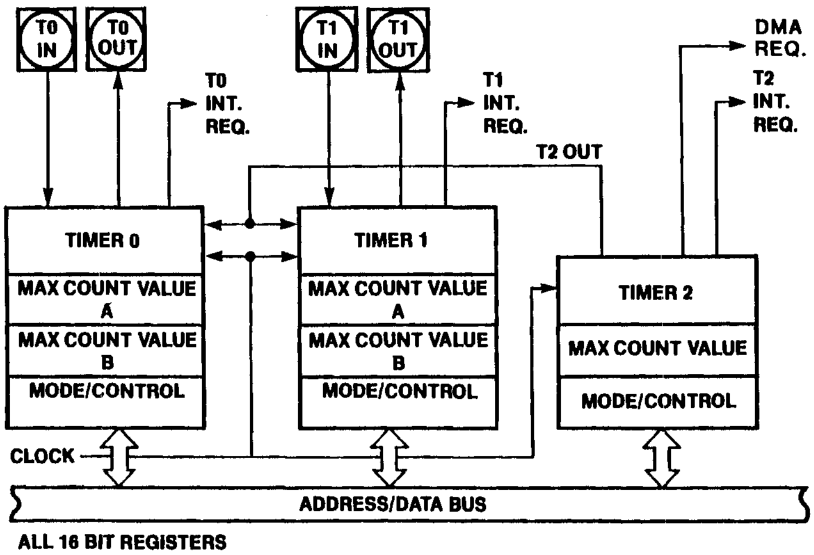
iAPX 186,188 INTERRUPT CONTROL UNIT BLOCK DIAGRAM



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.

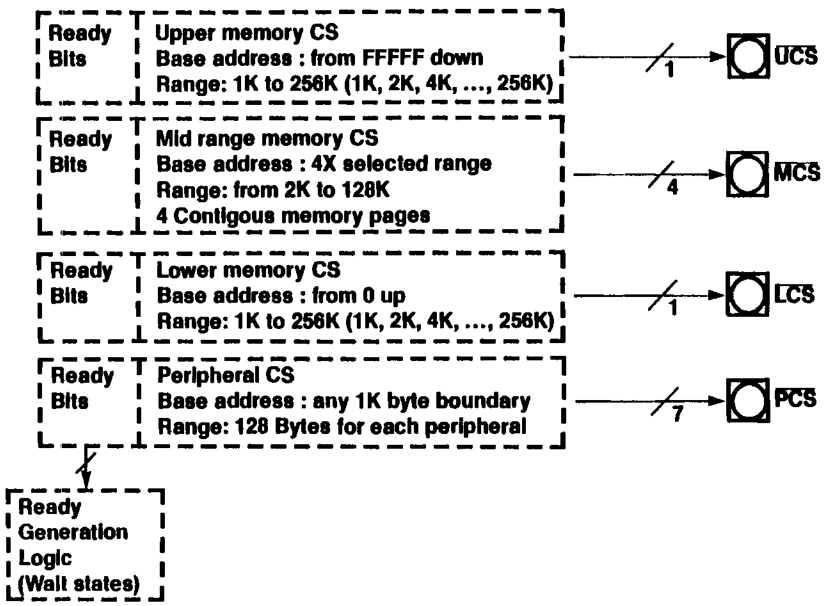
iAPX 186,188 TIMER/COUNTER BLOCK DIAGRAM



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)

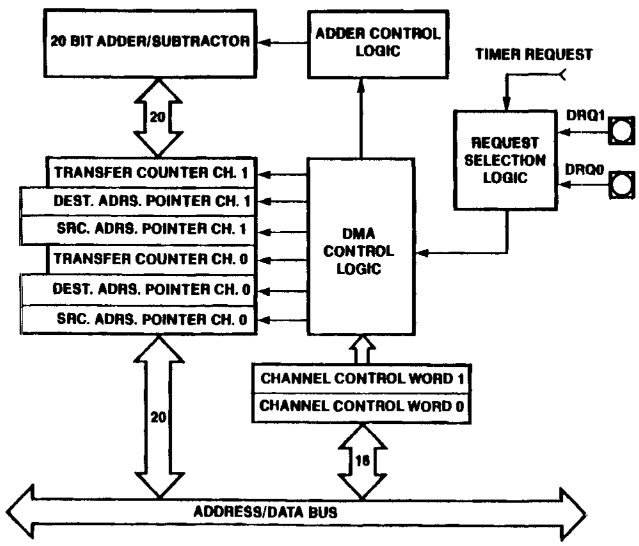
CHIP SELECT/READY GENERATION BLOCK DIAGRAM



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 BLOCK DIAGRAM



17-11

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

  } TWO BUS CYCLE TRANSFER
- BYTE OR WORD TRANSFERS
  - WORDS CAN BE TRANSFERRED TO/FROM ODD OR EVEN ADDRESSES
- 20-BIT SOURCE AND DESTINATION POINTER FOR EACH CHANNEL
  - CAN BE INCREMENTED/DECREMENTED INDEPENDENTLY DURING TRANSFER
- 16-BIT TRANSFER COUNTER
  - PROGRAMMABLE TERMINATE AND/OR INTERRUPT REQUEST WHEN COUNTER REACHES 0
- DMA REQUESTS CAN BE GENERATED BY TIMER 2
- 2MBYTE/SECOND MAXIMUM TRANSFER RATE

iAPX 186, 188 RELATIVE PERFORMANCE (8 MHz STANDARD CLOCK RATE)

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

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

```
SHL    AX, 12
ROR    BL, 4
SAR    DX, 3
RCR    XYZ, 2
```

- 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

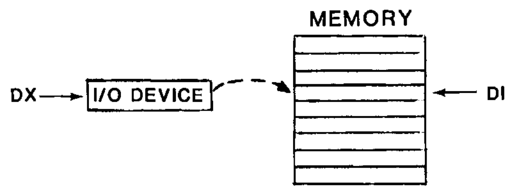
```
PUSH  50          ;PLACE 50 ON THE TOP
                  ;OF THE STACK
```

- 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

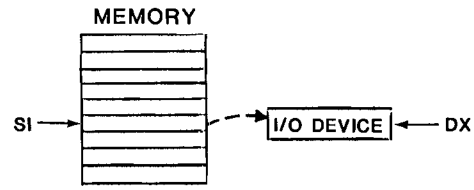


INSB (BYTE TRANSFER)
INSW (WORD TRANSFER)

DI ←— I/O[DX]
DI ←— DI +/- INCR *

* +/- INCR: + WHEN DF = 0 (CLD)
- WHEN DF = 1 (STD)

OUTS



OUTSB (BYTE TRANSFER)
OUTSW (WORD TRANSFER)

I/O[DX] ←— [SI]
SI ←— SI +/- INCR*

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

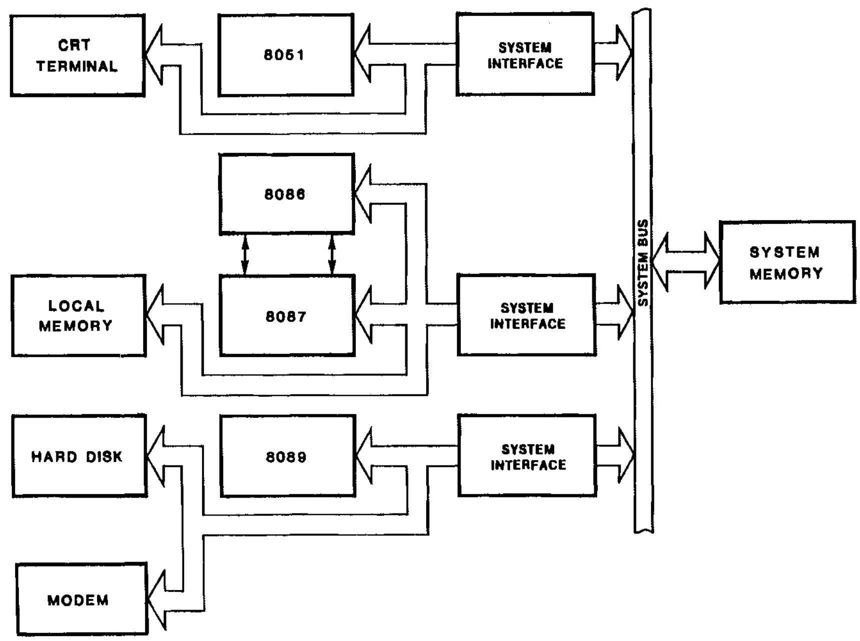
* DESCRIBE THE BASIC FUNCTIONS OF THE iAPX 286 AND iAPX 386

CHAPTER 18

MULTIBUS SYSTEM INTERFACE

- DESIGN CONSIDERATIONS
- HARDWARE INTERFACE TO THE MULTIBUS
- BUS ARBITRATION
- LOCK INSTRUCTION PREFIX
- BYTE SWAP BUFFER

FUNCTIONAL PARTITIONING SUPPORTS MULTIPROCESSING:



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)

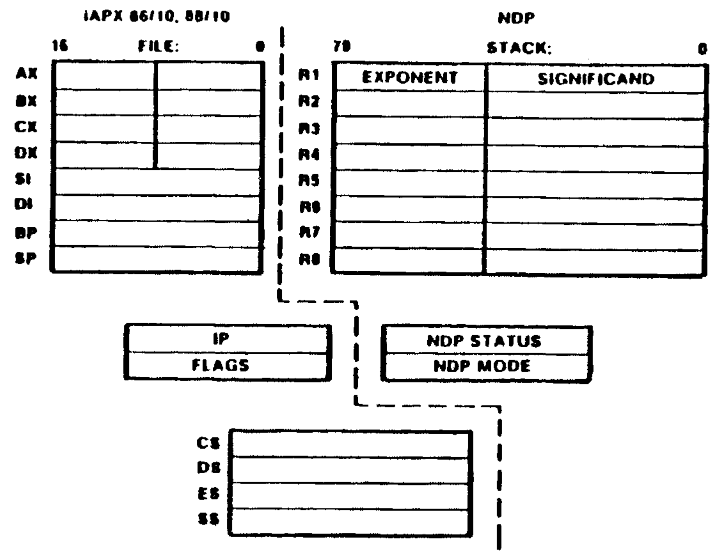
18-3

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

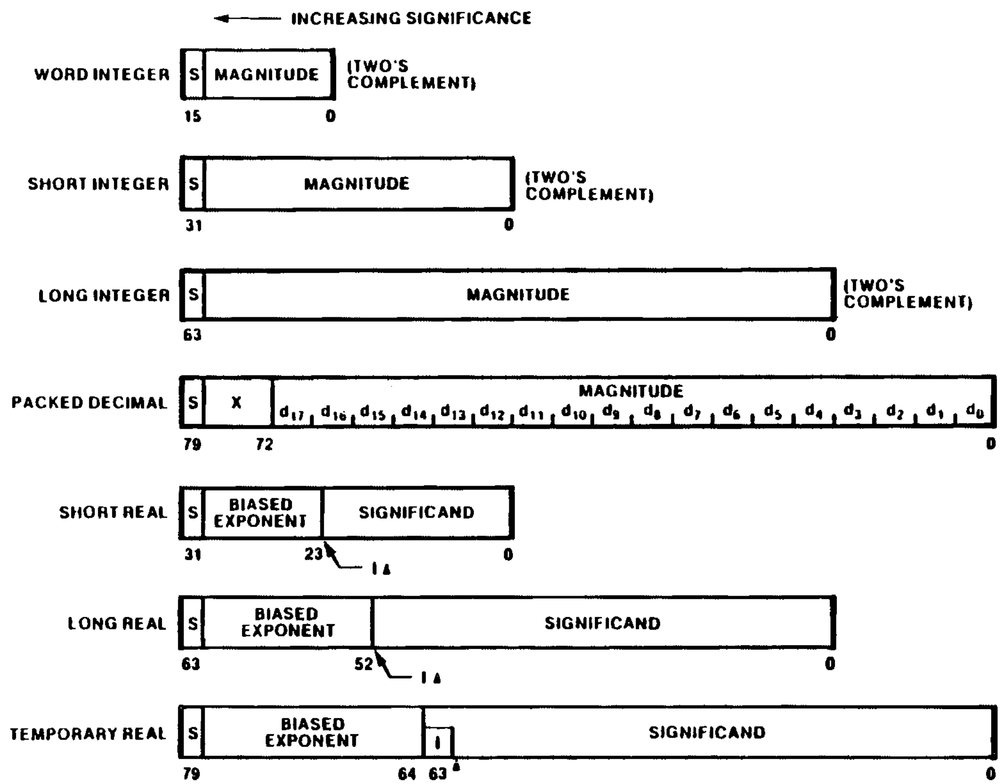
iAPX 86/20, 88/20 ARCHITECTURE

iAPX 86/20, 88/20

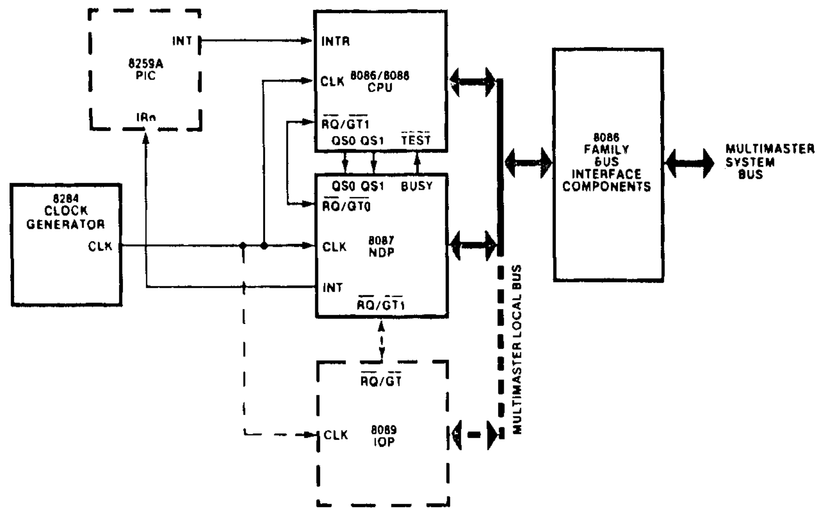


- THE 8087 CAN BE VIEWED AS AN ARCHITECTURAL EXTENSION OF AN 8086/8088.
- TO USE THE 8087, ADDITIONAL OPCODES AND OPERANDS ARE INCLUDED IN THE 8086/8088 INSTRUCTION SET.

DATA FORMATS FOR MEMORY OPERANDS



iAPX 86/20, 88/20 INTERCONNECT

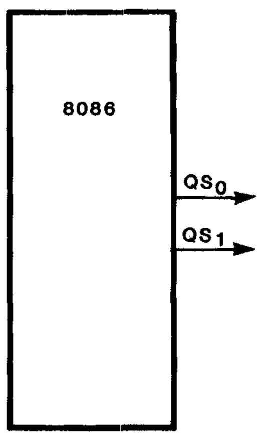


iAPX 86/20 iAPX 88/20 ARCHITECTURE

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

QUEUE STATUS LINES

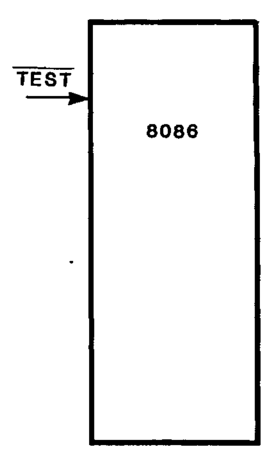
QS_1, QS_0 -QUEUE STATUS LINES: INDICATE THE INSTRUCTION QUEUE STATUS AS FOLLOWS:



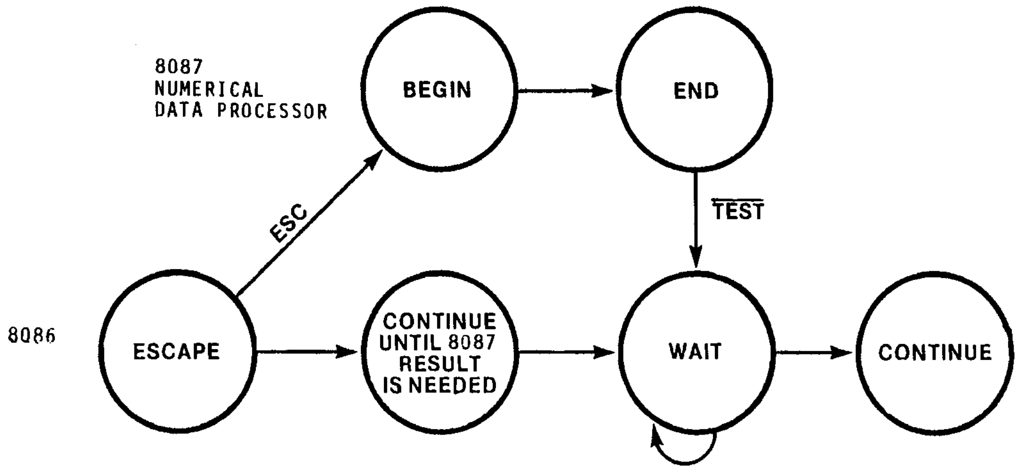
QS_1	QS_0	STATUS
0	0	NO OPERATION
0	1	FIRST BYTE OF OPCODE
1	0	EMPTY THE QUEUE
1	1	SUBSEQUENT BYTE

TEST PIN

$\overline{TEST}$ -USED BY WAIT INSTRUCTION TO SYNCHRONIZE PROCESSORS
IF $\overline{TEST}$ PIN IS LOW, EXECUTE CONTINUES
IF $\overline{TEST}$ PIN IS HIGH, CPU ENTERS AN IDLE STATE



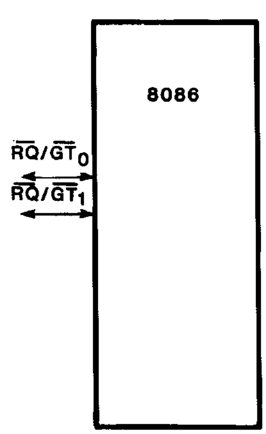
8087 CO-PROCESSOR OPERATION



REQUEST/GRANT LINES

$\overline{RQ}/\overline{GT}_0$
$\overline{RQ}/\overline{GT}_1$

-REQUEST GRANT: BIDIRECTIONAL HANDSHAKE LINES ALLOWS UP TO TWO SEPERATE DEVICES CONTROL OF THE BUSSES

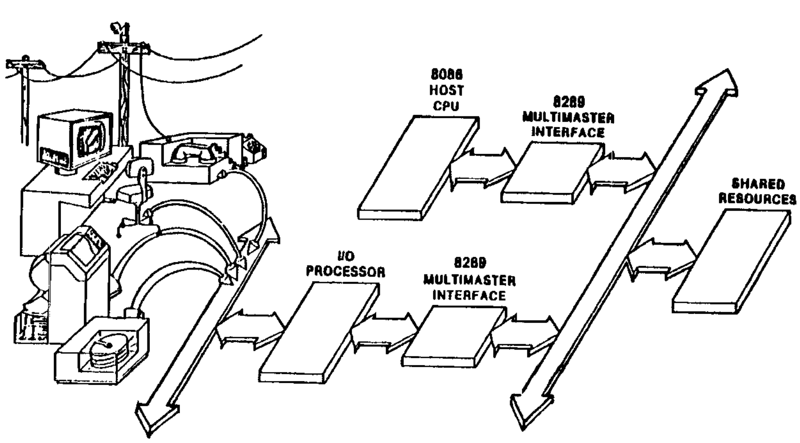


EXECUTION TIME FOR SELECTED iAPX 86/20 INSTRUCTIONS

INSTRUCTION	APPROXIMATE EXECUTION TIME (μs)	
	iAPX 86/20 (5 MHz CLOCK)	iAPX 86/10 EMULATION
ADD/SUBTRACT MAGNITUDE	14/18	1,600
MULTIPLY (SINGLE PRECISION)	18	1,600
MULTIPLY (DOUBLE PRECISION)	27	2,100
DIVIDE	39	3,200
COMPARE	10	1,300
LOAD (SINGLE PRECISION)	9	1,700
STORE (SINGLE PRECISION)	17	1,200
SQUARE ROOT	36	19,600
TANGENT	110	13,000
EXPONENTIATION	130	17,100

8089 IO PROCESSOR

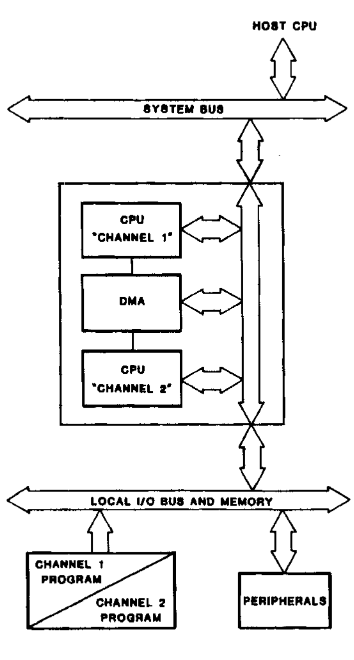
* THE I/O PROCESSOR CONTROLS ALL I/O IN THE SYSTEM
* BOTH PROCESSORS OPERATE IN PARALLEL
* SYSTEM THROUGHPUT IS ENHANCED



I/O PROCESSOR FEATURES

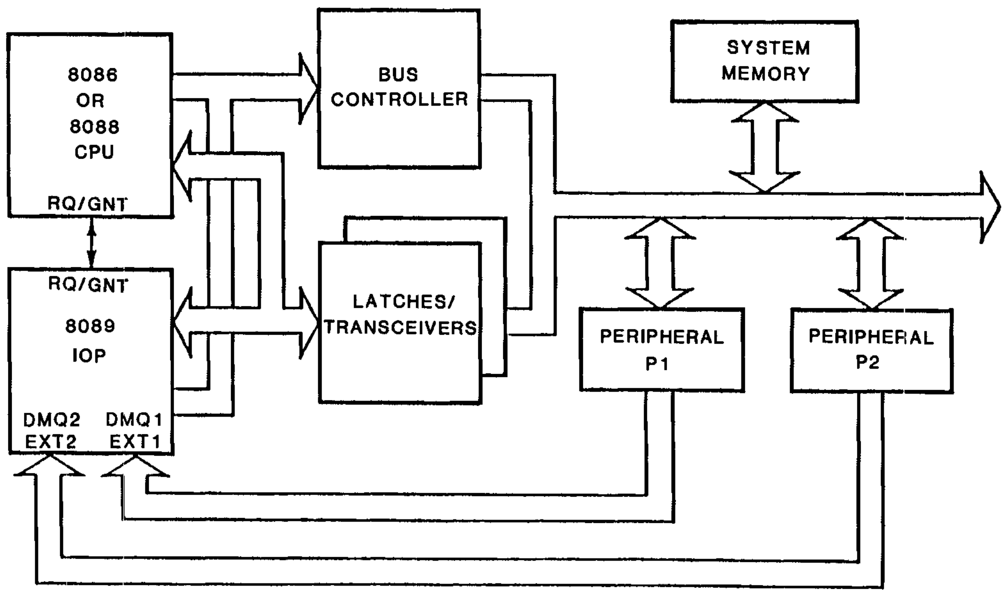
- 2 INDEPENDENT CHANNELS
- 1 MEGABYTE SYSTEM SPACE, 64K I/O SPACE
- 2 LOGICAL BUSSES CAN BE TREATED AS 8 OR 16 OR BOTH TO MATCH PERIPHERALS TO SYSTEM
- CHANNEL PROGRAM STORE CAN BE ON SYSTEM OR LOCAL BUS
- INSTRUCTION SET TAILORED FOR I/O FUNCTIONS

I/O PROCESSOR BLOCK DIAGRAM



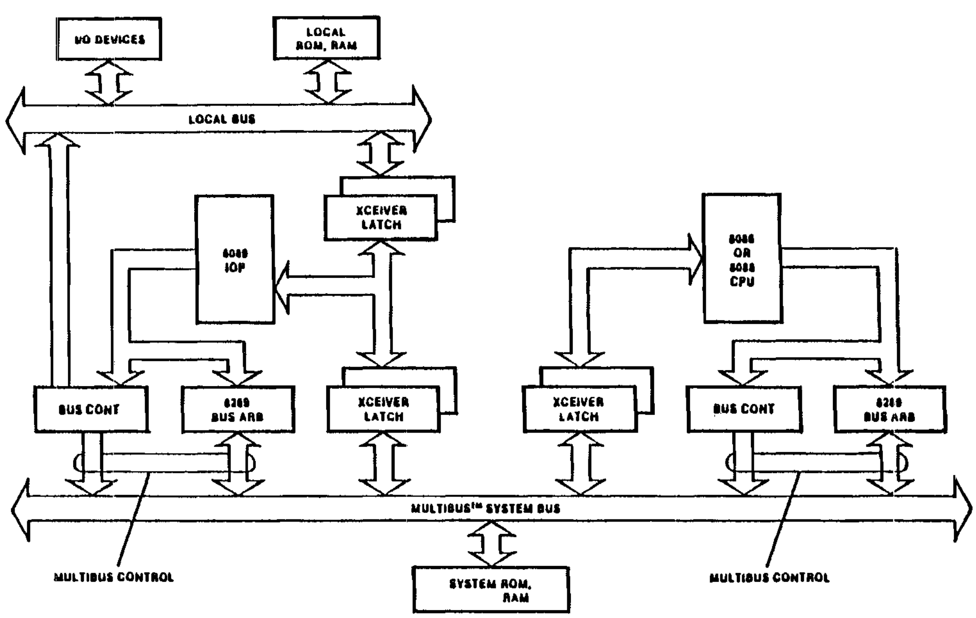
- INFORMATION FLOWS THROUGH IOP
- INSTRUCTIONS APPLY TO I/O OR SYSTEM
- 2 LOGICAL BUSES
- 2 CHANNELS
  2 REGISTER SETS
  2 INSTRUCTION POINTERS

LOCAL CONFIGURATION
MINIMUM BOARD SPACE AND COST



18-17

REMOTE CONFIGURATION ALLOWS
PARALLEL PROCESSING



18-18

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

	5 MHz	8 MHz
DMA TRANSFER (16 BIT TRANSFERS)	1.25 Mbyte	2.0 Mbyte
DMA BYTE SEARCH 8 BIT/16 BIT SOURCE	0.6125/0.833 Mbyte	1.0/1.33 Mbyte
DMA BYTE TRANSLATE	0.333 Mbyte	0.533 Mbyte
DMA BYTE SEARCH AND TRANSLATE	0.333 Mbyte	0.533 Mbyte
DMA RESPONSE (LATENCY) SINGLE CHANNEL/DUAL CHANNEL	1.0/2.2 μs	0.625/1.375 μs

OPERATING SYSTEM FIRMWARE COMPONENT

* 16kbyte CONTROL STORE

* PROGRAMMABLE INTERRUPT CONTROLLER MANAGED BY OS SOFTWARE

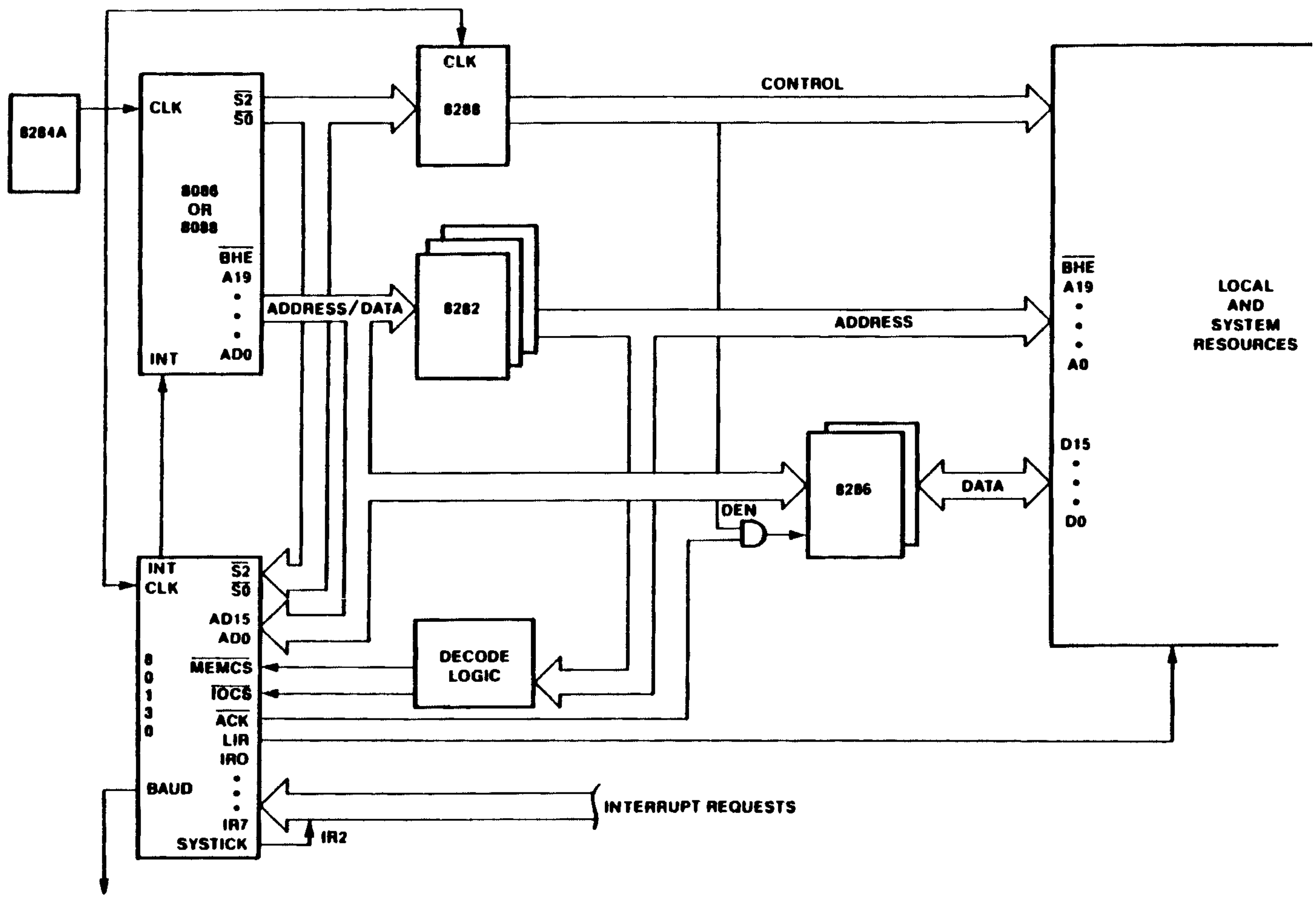
* 3 PROGRAMMABLE TIMERS

    SYSTEM (8254 RATE GEN MODE)
    DELAY (8254 COUNT MODE)
    BAUD (8254 SQUARE WAVE MODE)

80130 FEATURES

HARDWARE	SOFTWARE
☐ 128 K-bit kernal control store	☐ Task management
☐ Programmable interrupt controller	☐ Intertask communication and synchronization
☐ System timer	☐ Mutual exclusion control
☐ Delay timer	☐ Interrupt management
☐ Baud-rate generator	☐ Free memory management/ system partitioning

TYPICAL SYSTEM USING OPERATING SYSTEM PROCESSOR



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.

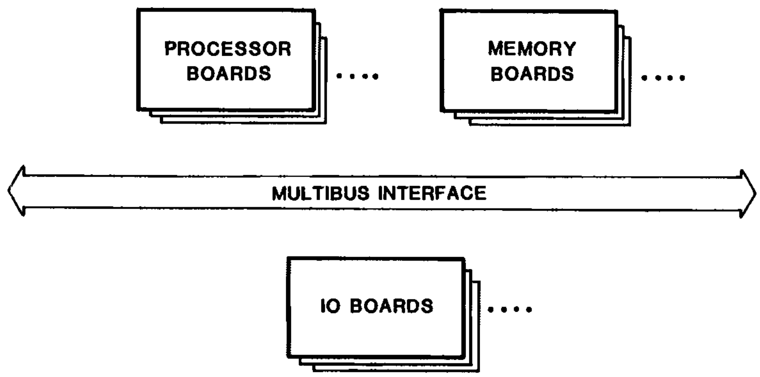
RBF89 (REAL-TIME BREAKPOINT FACILITY) IS A DEBUGGING TOOL FOR THE 8089 AND WORKS IN CONJUNCTION WITH ICE86(A).

CHAPTER 19

MULTI AND COPROCESSOR

- 8087 NUMERIC DATA PROCESSOR
- 8089 I/O PROCESSOR
- 80130 OPERATING SYSTEM

WHAT IS THE MULTIBUS SYSTEM INTERFACE?

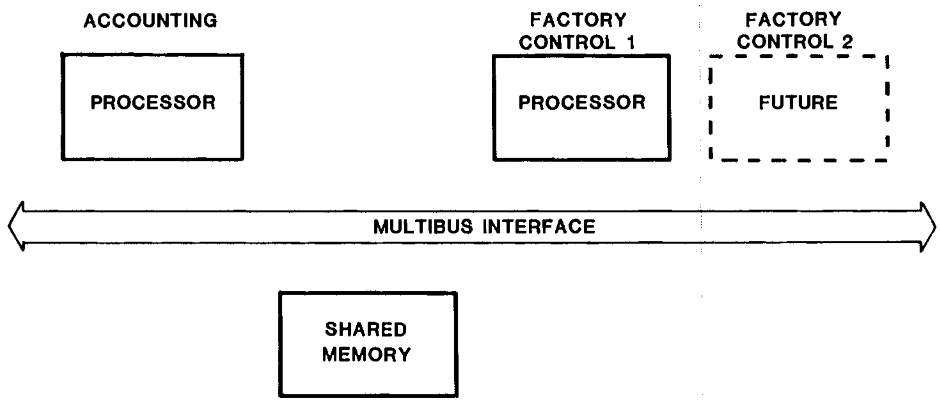


- 16 MEGABYTE ADDRESS SPACE
- IEEE STANDARD (IEEE 796)
- INDUSTRY STANDARD
  - OVER 40 VENDORS OF MULTIBUS BOARDS
  - OVER 40 BOARDS AVAILABLE FROM INTEL

WHY USE THE MULTIBUS SYSTEM INTERFACE?

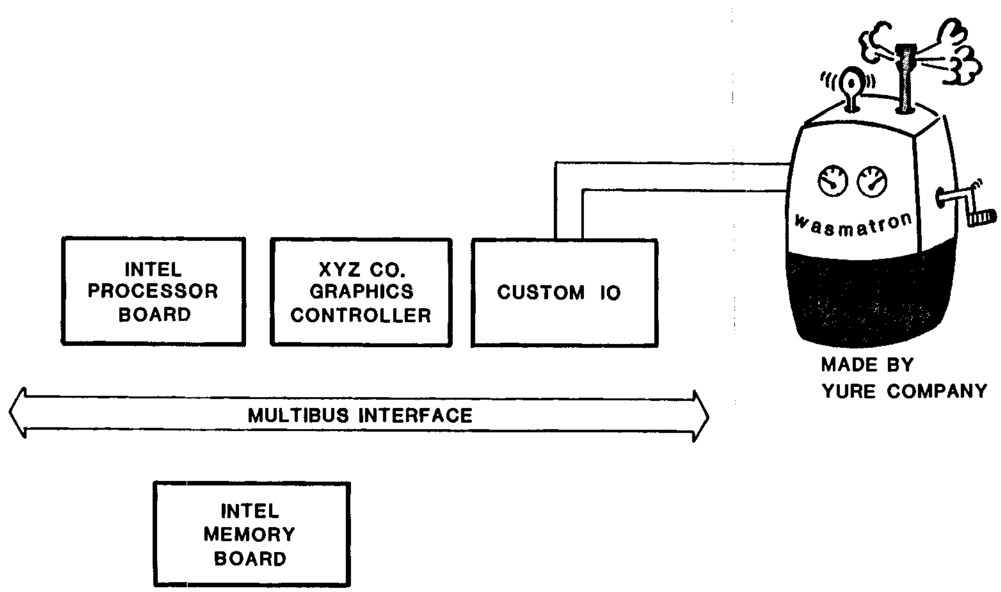
- MODULARIZE HARDWARE/DISTRIBUTED PROCESSING
- SHORTEN DESIGN TIME
- REDUCE COST OF DESIGN AND TEST
- FLEXIBLE
  - SYSTEM CAN BE QUICKLY RECONFIGURED
  - EASY TO ADD MORE PROCESSING POWER, MEMORY OR IO
  - SIMPLIFIES REPAIR

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

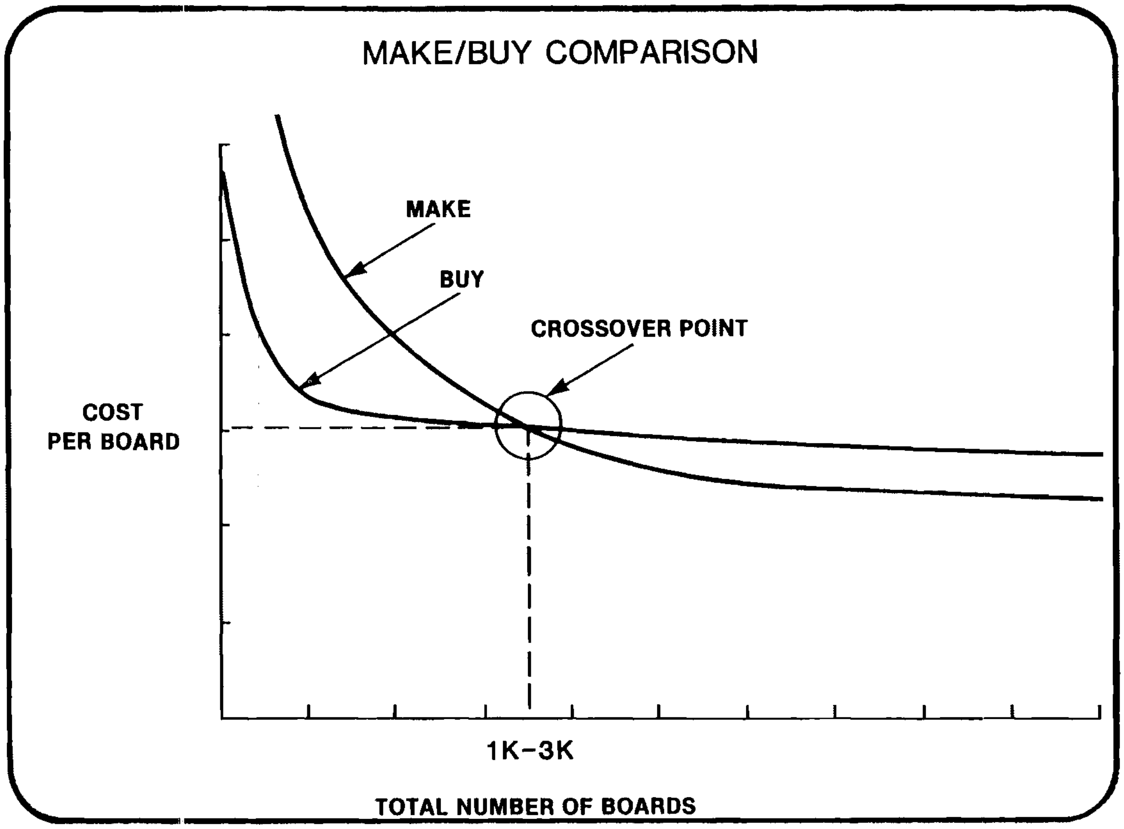
REDUCE COST/SHORTEN DESIGN TIME



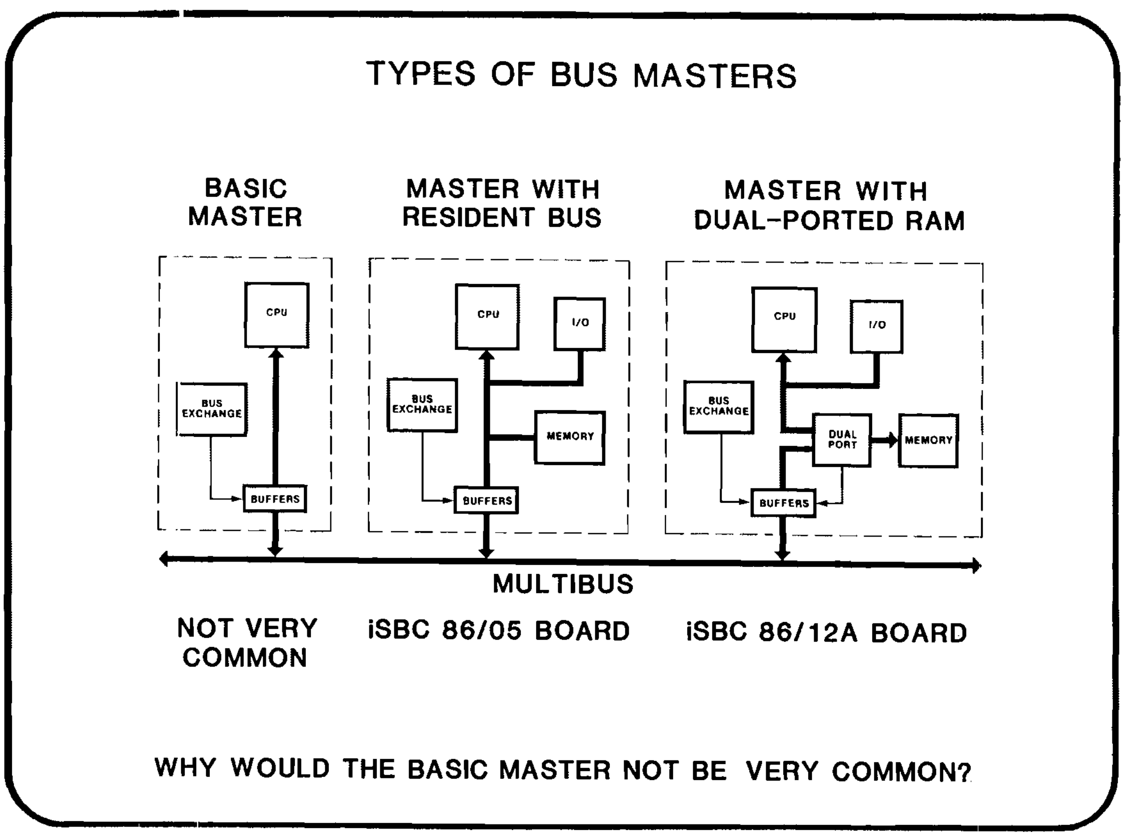
- CONFIGURE SYSTEM COMPLETELY FROM AVAILABLE BOARDS

OR

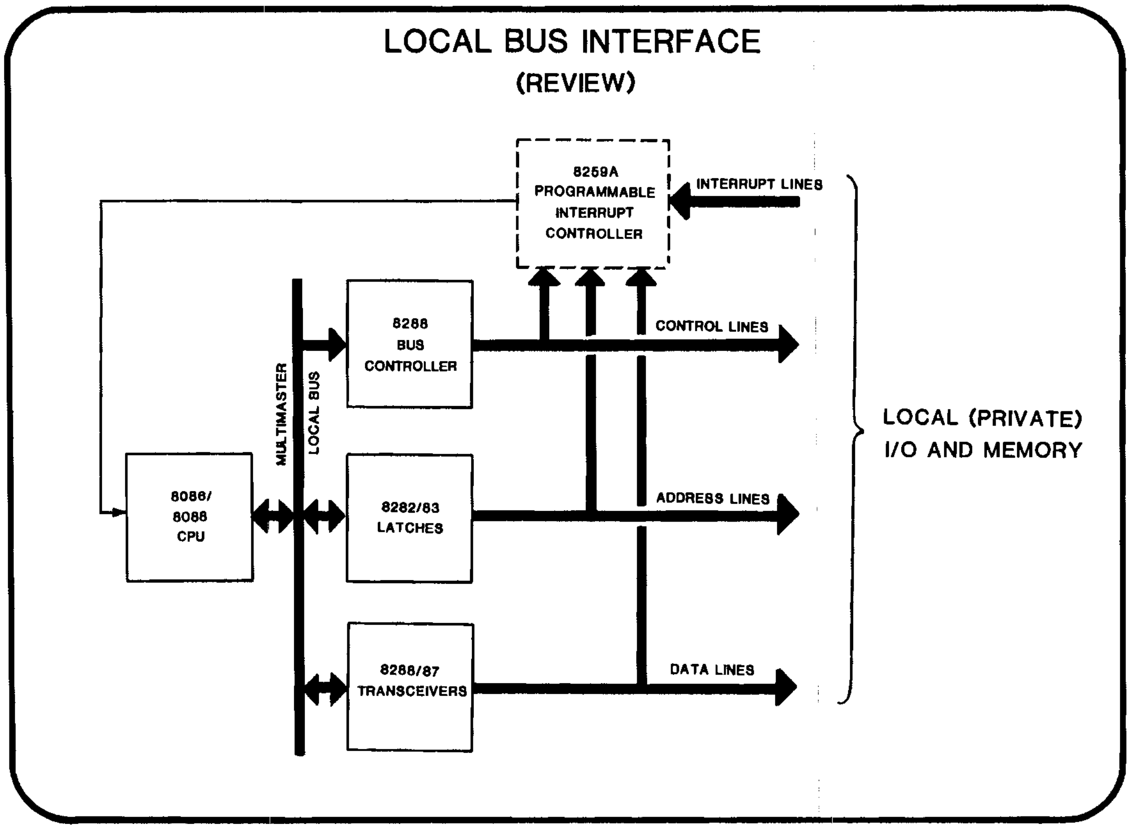
- DESIGN CUSTOM IO BOARDS FOR YOUR APPLICATION


MAKE/BUY COMPARISON
MAKE
BUY
CROSSOVER POINT
COST
PER BOARD
1K-3K
TOTAL NUMBER OF BOARDS

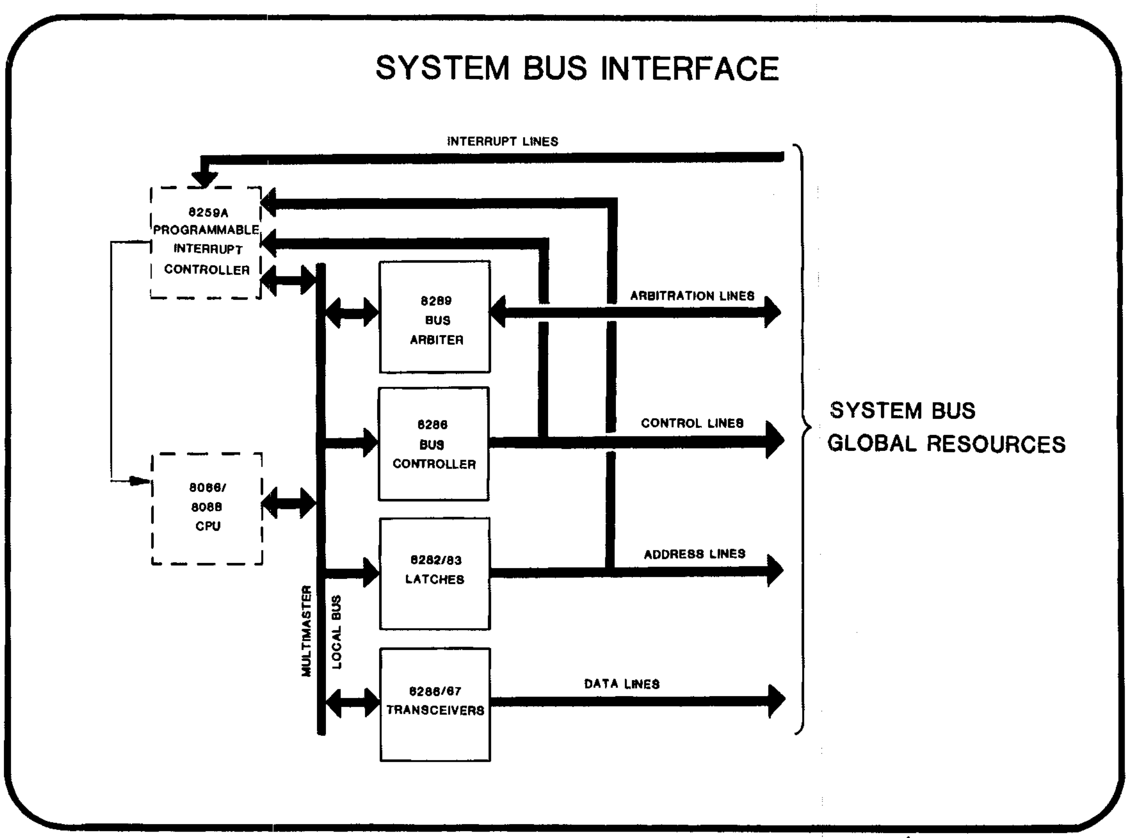
19-5


TYPES OF BUS MASTERS
BASIC
MASTER
MASTER WITH
RESIDENT BUS
MASTER WITH
DUAL-PORTED RAM
CPU
BUS
EXCHANGE
BUFFERS
CPU
I/O
BUS
EXCHANGE
MEMORY
BUFFERS
CPU
I/O
BUS
EXCHANGE
DUAL
PORT
MEMORY
BUFFERS
MULTIBUS
NOT VERY
COMMON
iSBC 86/05 BOARD
iSBC 86/12A BOARD
WHY WOULD THE BASIC MASTER NOT BE VERY COMMON?

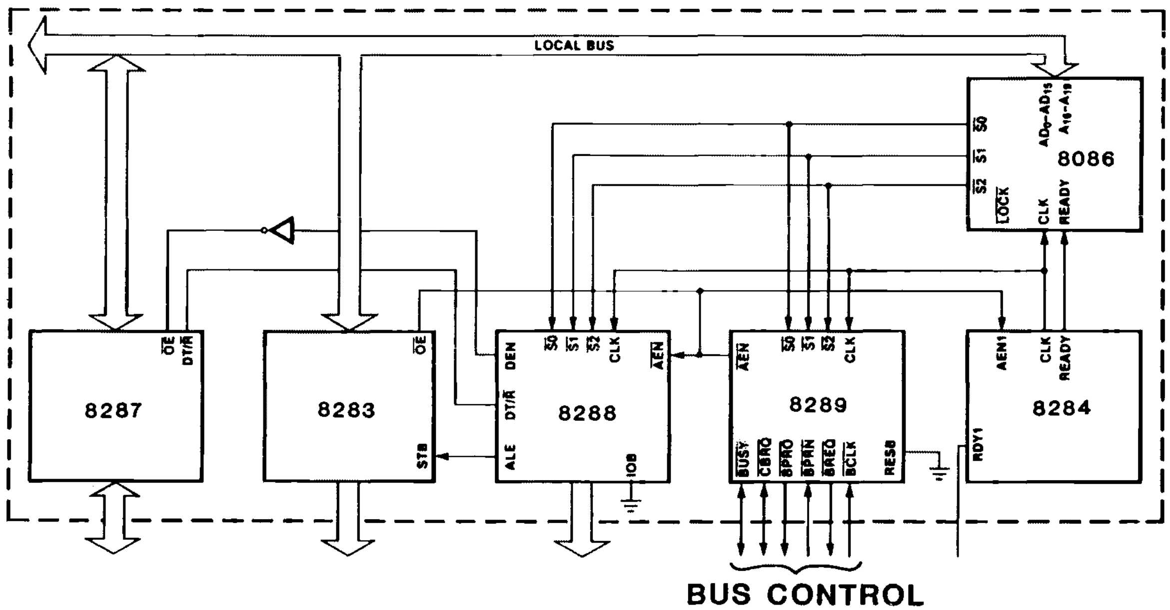
19-6


LOCAL BUS INTERFACE
(REVIEW)
8259A
PROGRAMMABLE
INTERRUPT
CONTROLLER
INTERRUPT LINES
8288
BUS
CONTROLLER
CONTROL LINES
MULTIMASTER
LOCAL BUS
LOCAL (PRIVATE)
I/O AND MEMORY
8086/
8088
CPU
8282/83
LATCHES
ADDRESS LINES
8286/87
TRANSCEIVERS
DATA LINES

19-7

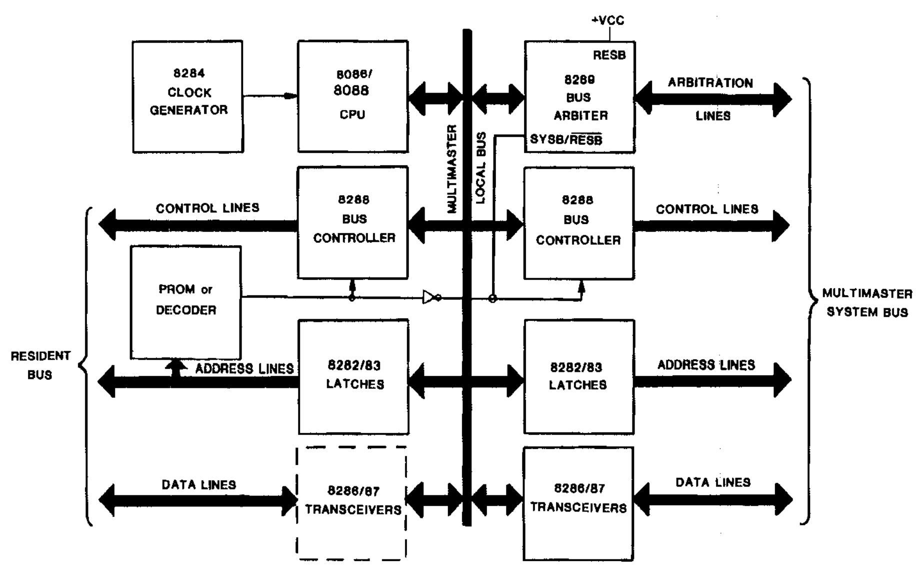

SYSTEM BUS INTERFACE
INTERRUPT LINES
8259A
PROGRAMMABLE
INTERRUPT
CONTROLLER
8289
BUS
ARBITER
ARBITRATION LINES
8288
BUS
CONTROLLER
CONTROL LINES
SYSTEM BUS
GLOBAL RESOURCES
8086/
8088
CPU
8282/83
LATCHES
ADDRESS LINES
MULTIMASTER
LOCAL BUS
8286/87
TRANSCEIVERS
DATA LINES

BASIC MASTER



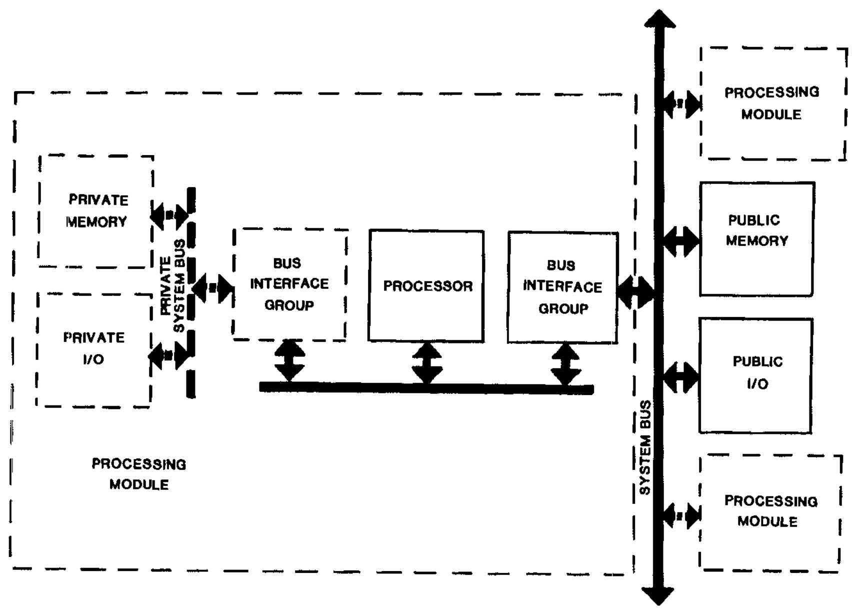
- 8289 RESB PIN TIED LOW (NO RESIDENT BUS)
- ALL MEMORY AND I/O CYCLES REQUIRE MULTIBUS ACCESS
- ONLY WHEN 8289 GETS CONTROL OF BUS DOES IT ENABLE BUS CONTROLLER (8288) AND ADDRESS LATCHES (8283,S)

MASTER WITH RESIDENT BUS



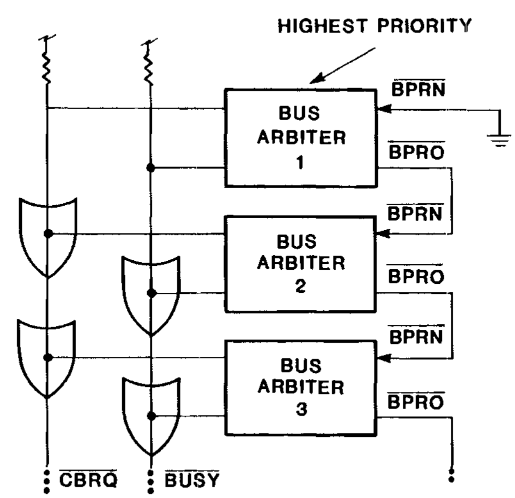
- 8289 RESB PIN TIED HIGH (RESIDENT BUS PRESENT)
- ADDRESS DECODING SELECTS THE SYSTEM BUS OR RESIDENT BUS VIA 8289 PIN SYSB/$\overline{\text{RESB}}$.

BUS ARBITRATION



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)

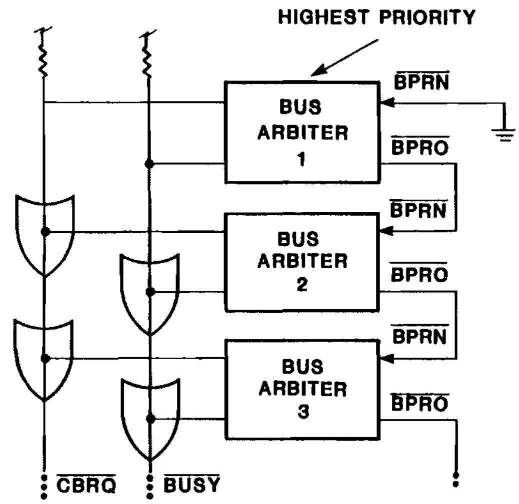
  NO HIGHER PRIORITY MASTER NEEDS THE BUS

  BUSY IS HIGH

  THE BUS ISN'T BEING USED NOW

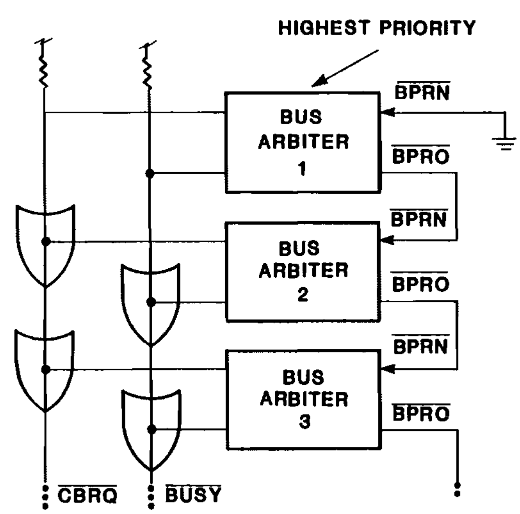
NOTE:
THERE IS A MAXIMUM OF 3 MASTERS WHEN USING THE SERIAL PRIORITY RESOLVING TECHNIQUE

SERIAL PRIORITY RESOLVING



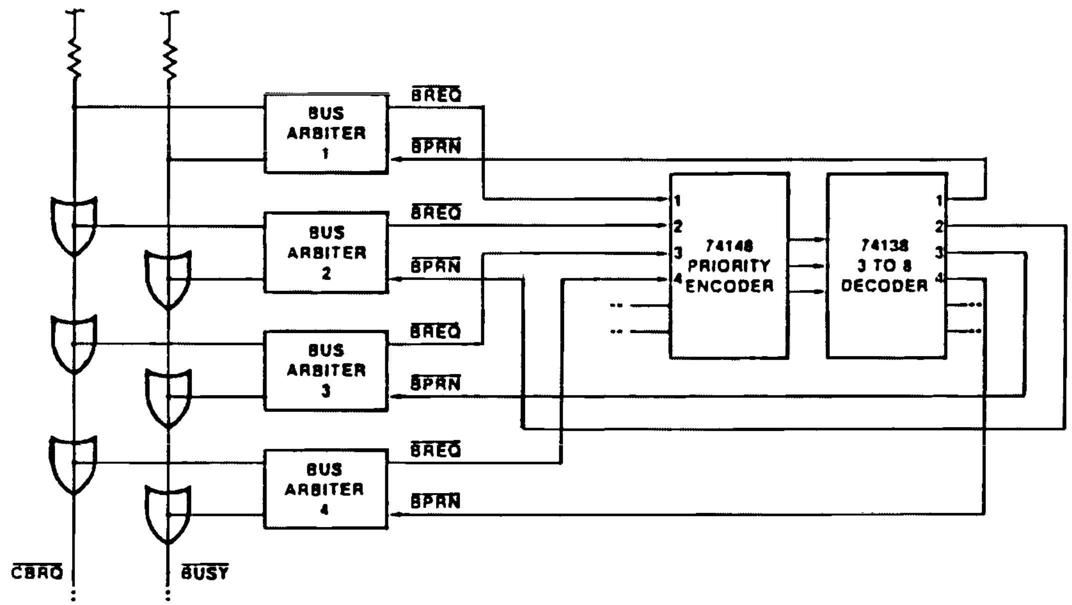
- A MASTER REQUESTS THE BUS BY DRIVING
  - BPRO HIGH (BUS PRIORITY OUT)
    - ALL LOWER PRIORITY MASTERS GET OFF THE BUS
  - CBRQ LOW (COMMON BUS REQUEST)
    - IF A HIGHER PRIORITY MASTER HAS THE BUS AND DOES NOT NEED IT, RELEASE THE BUS.

SERIAL PRIORITY RESOLVING



- A MASTER WILL RELEASE THE BUS WHEN
  - BPRN GOES HIGH
    - A HIGHER PRIORITY MASTER WANTS THE BUS
  - OR 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



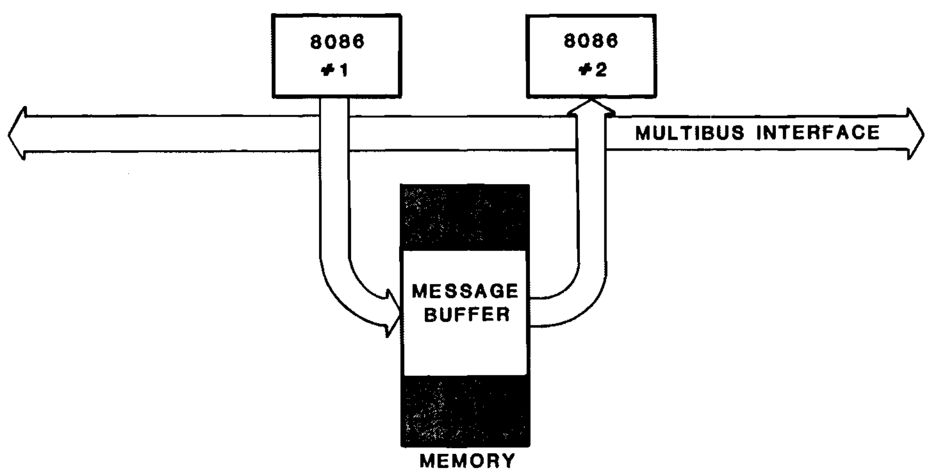
ADVANTAGES

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

DISADVANTAGE

- REQUIRES EXTRA , USER-SUPPLIED HARDWARE.

MUTUAL EXCLUSION PROBLEM



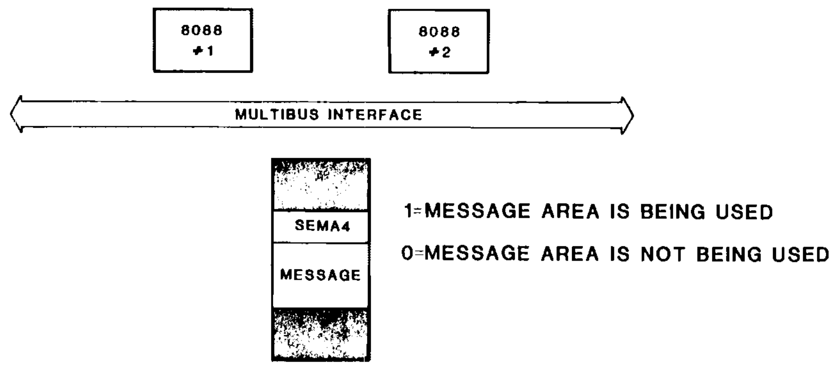
PROBLEM:

8086 #2 STARTS READING MESSAGE
8086 #1 STARTS UPDATING MESSAGE BEFORE #2 IS FINISHED
8086 #2 GETS INVALID MESSAGE

SOLUTION:

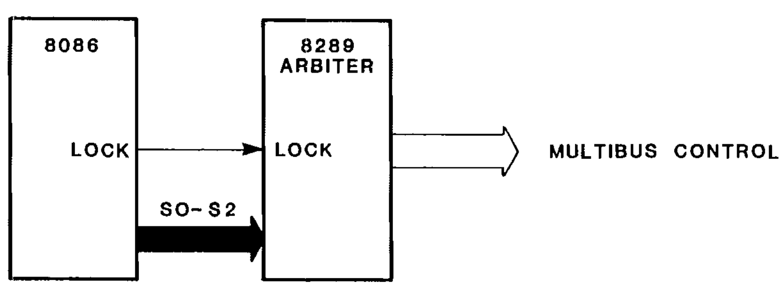
USE ONE SHARED MEMORY LOCATION AS A FLAG (SEMAPHORE), WHICH INDICATES IF MESSAGE AREA IS BEING USED.

USING A SEMAPHORE WITH THE LOCK INSTRUCTION PREFIX




```
LOOP1:  MOV   AL,1
LOCK    XCHG  SEMA4,AL     ;GET AND SET SEMA4 WITHOUT
        CMP   AL, 1        ;RELEASING THE BUS
        JE    LOOP1        ;TRY AGAIN IF SEMA4 WAS SET
        :
        ;ACCESS MESSAGE
        :
        MOV   SEMA4,0      ;RELEASE MESSAGE AREA
```

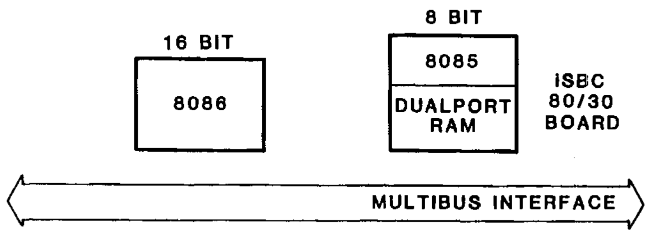

LOCKING THE MULTIBUS



- 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

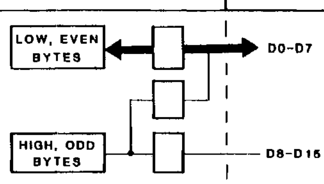
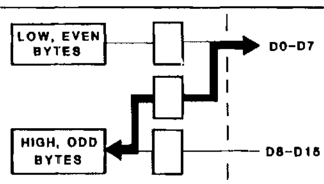
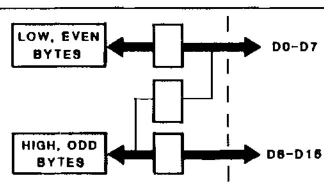
BETWEEN 8 AND 16 BIT BOARDS



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.

BYTE SWAP BUFFER

16-BIT DEVICE	MULTIBUS	$\overline{BHE}$	A0	MULTIBUS TRANSFER DATA PATH
		HI	LO	D0-D7
		LO	HI	8-BIT D0-D7
		LO	LO	16-BIT D0-D15

- 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

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 $\overline{BUSY}$ HIGH

___ BM2 ISSUES $\overline{CBRQ}$ LOW

_1_ BM2 DRIVES $\overline{BPRO}$ HIGH

___ BM2 TAKES OVER BUS, DRIVES $\overline{BUSY}$ LOW

___ BM3 SEES $\overline{CBRQ}$ LOW

___ BM3 SEES $\overline{BPRN}$ 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

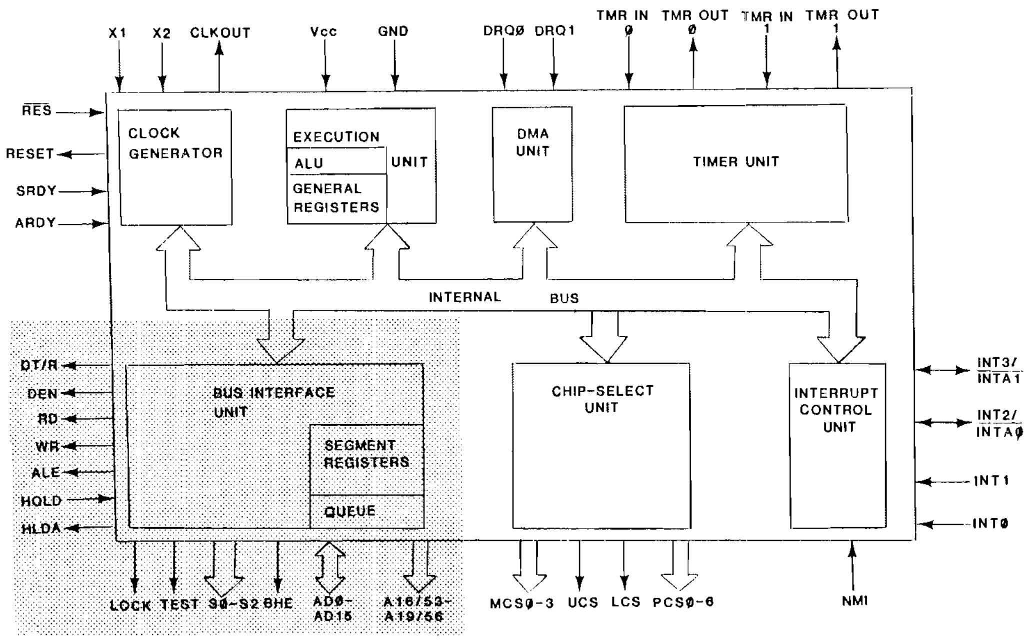
- CHAPTER 4, iAPX 86/88, 186/188 USER'S MANUAL

CHAPTER 20

iAPX 186,188 HARDWARE INTERFACE

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

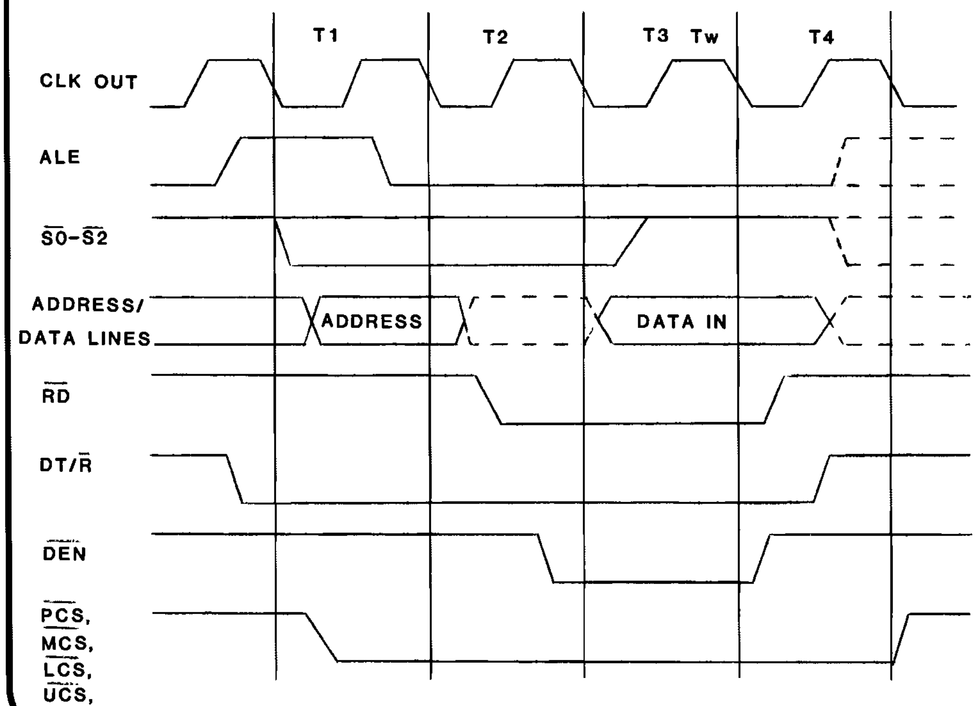
BUS INTERFACING



80186 BUS SIGNALS

ADDRESS/DATA	ADO - AD15
ADDRESS/STATUS	A16/S3 - A19/S6, BHE/S7
CO-PROCESSOR CONTROL	TEST
LOCAL BUS ARBITRATION	HOLD, HLDA
LOCAL BUS CONTROL	ALE , RD, WR, DT/R, DEN
MULTI-MASTER BUS	LOCK
STATUS INFORMATION	SO - S2

READ CYCLE



20-3

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

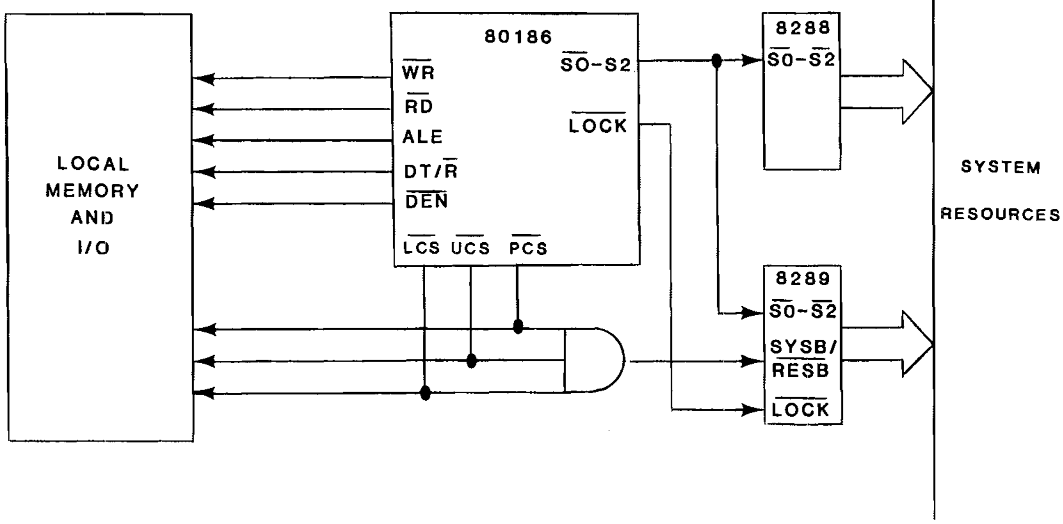
HOLD/HLDA IS PROVIDED RATHER THAN RQ/GT

S3 – S6 PROVIDE DIFFERENT INFORMATION THAN 8086

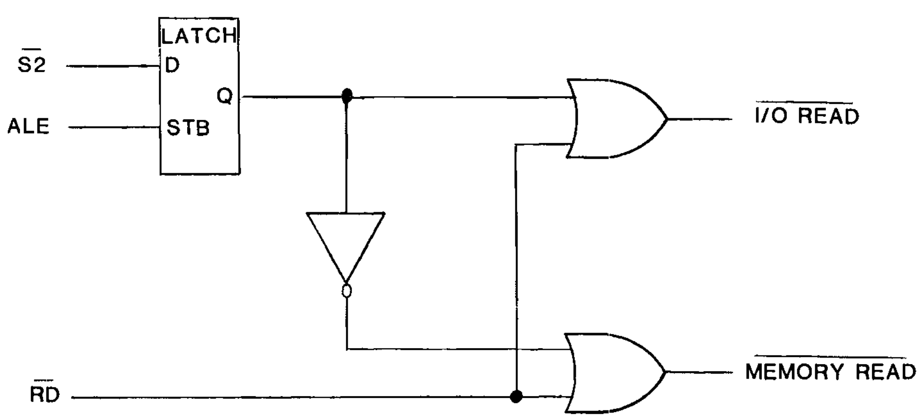
THE OUTPUT DRIVERS WILL DRIVE DOUBLE THE LOAD

20-4

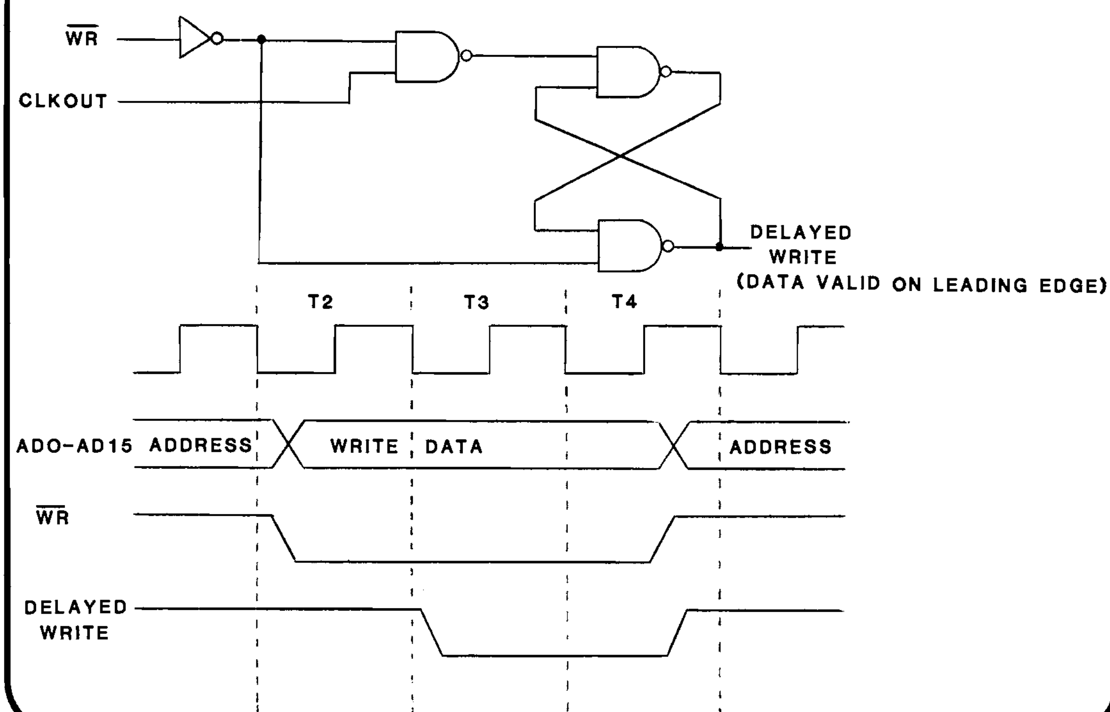
THE 80186 PROVIDES BOTH LOCAL BUS SIGNALS AND SYSTEM BUS SIGNALS



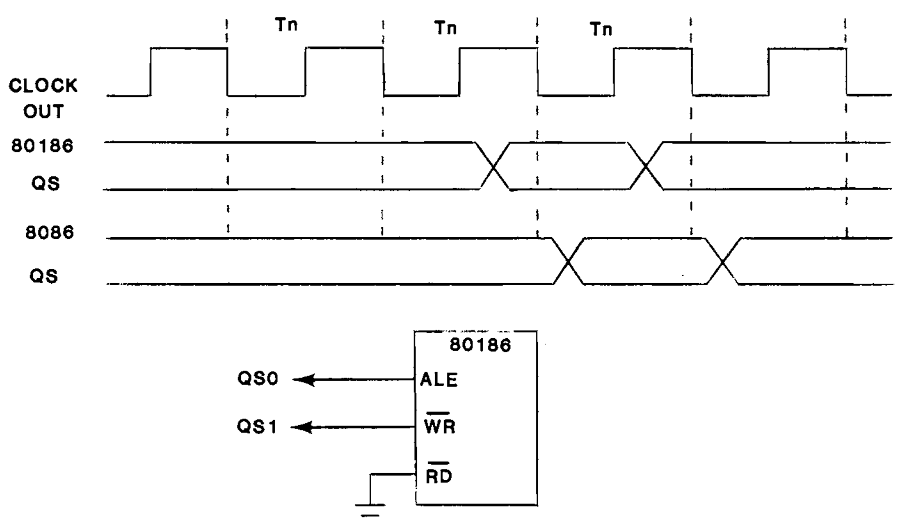
GENERATING SEPARATE I/O AND MEMORY READ SIGNALS



SYNTHESIZING DELAYED WRITE ON 80186



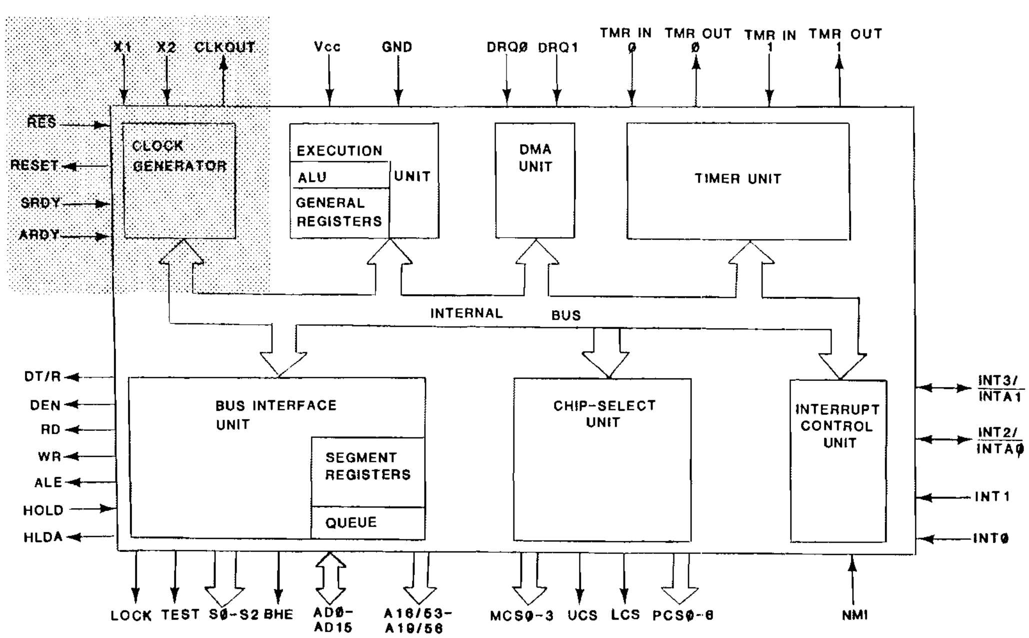
80186 QUEUE STATUS MODE



S3 – S6 STATUS SIGNAL DIFFERENCES

	8086	80186
S3 – S4	SEGMENT REGISTER USED	LOW
S5	INTERRUPT ENABLE FLAG CONDITION	LOW
S6	LOW	LOW IF CPU BUS CYCLE HIGH IF DMA BUS CYCLE

CLOCK GENERATOR

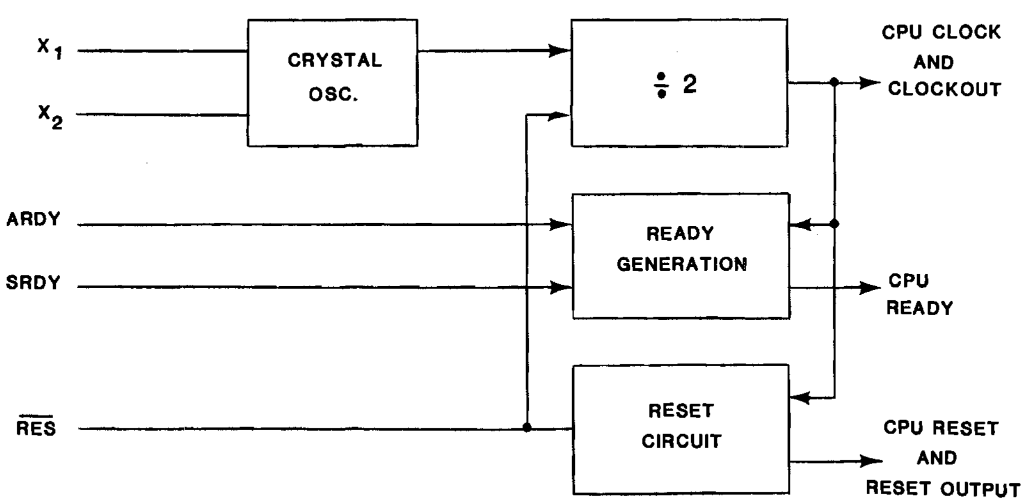


80186 INTERNAL CLOCK GENERATOR

- 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

20-11

80186 CLOCK GENERATOR BLOCK DIAGRAM



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 (FFOOH TO FFFFH) OF THE I/O SPACE

UMCS = FFFBH

> $\overline{\text{UCS}}$ 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

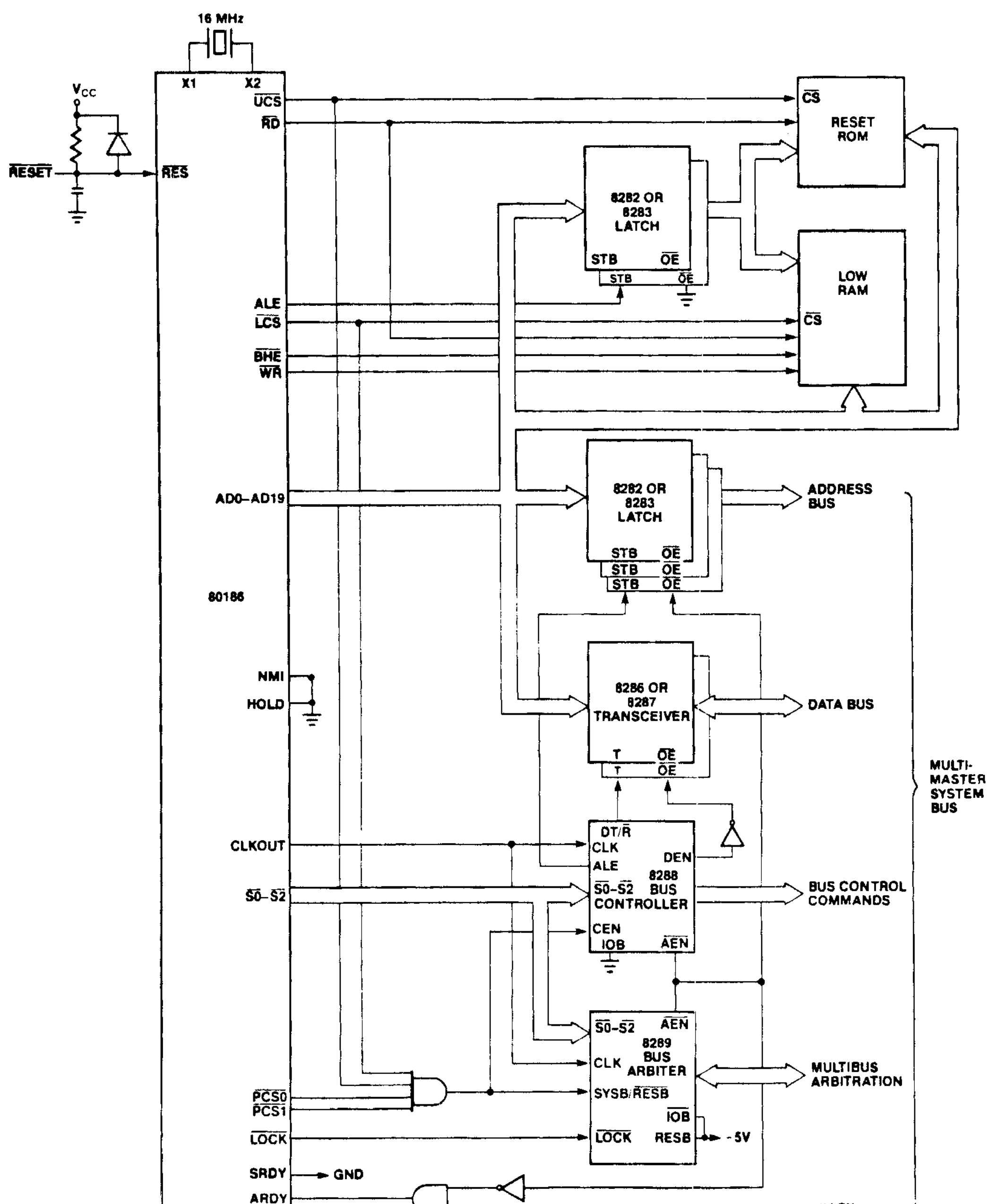
THE SYSTEM BUS IS ASYNCHRONOUS AND NORMALLY NOT READY

THE LOCAL BUS OPERATES SYNCHRONOUS TO THE PROCESSOR

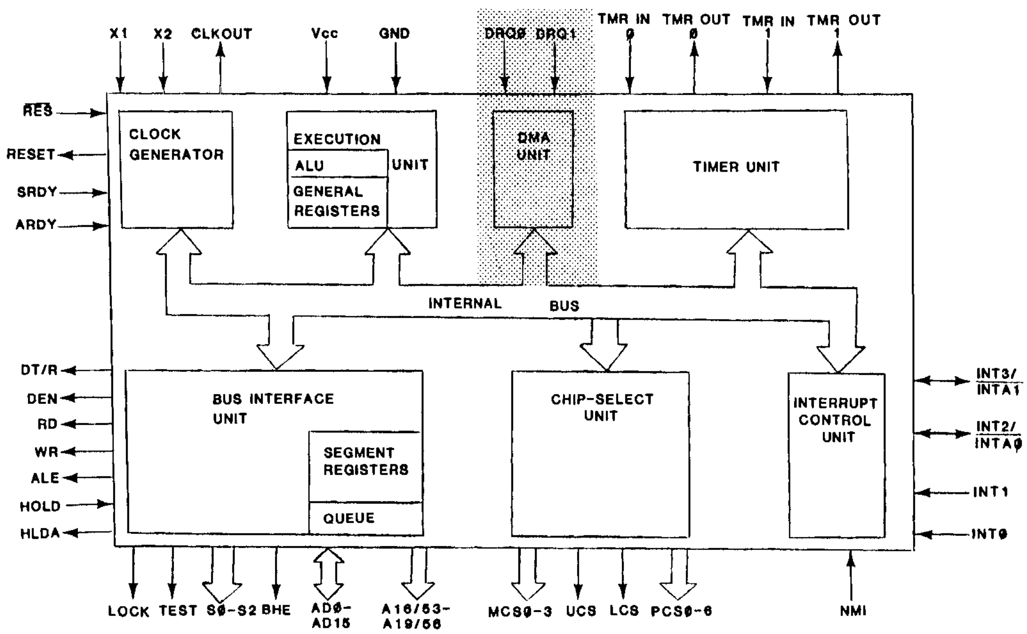
ARDY WOULD BE USED FOR THE SYSTEM BUS

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

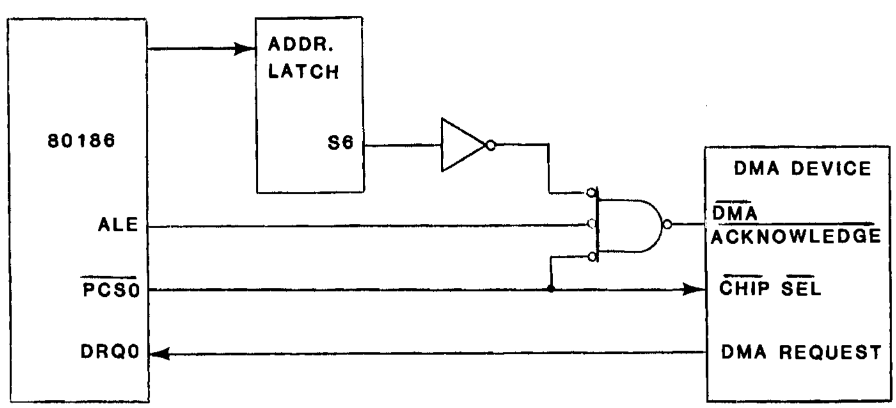
MULTIMASTER BUS INTERFACE



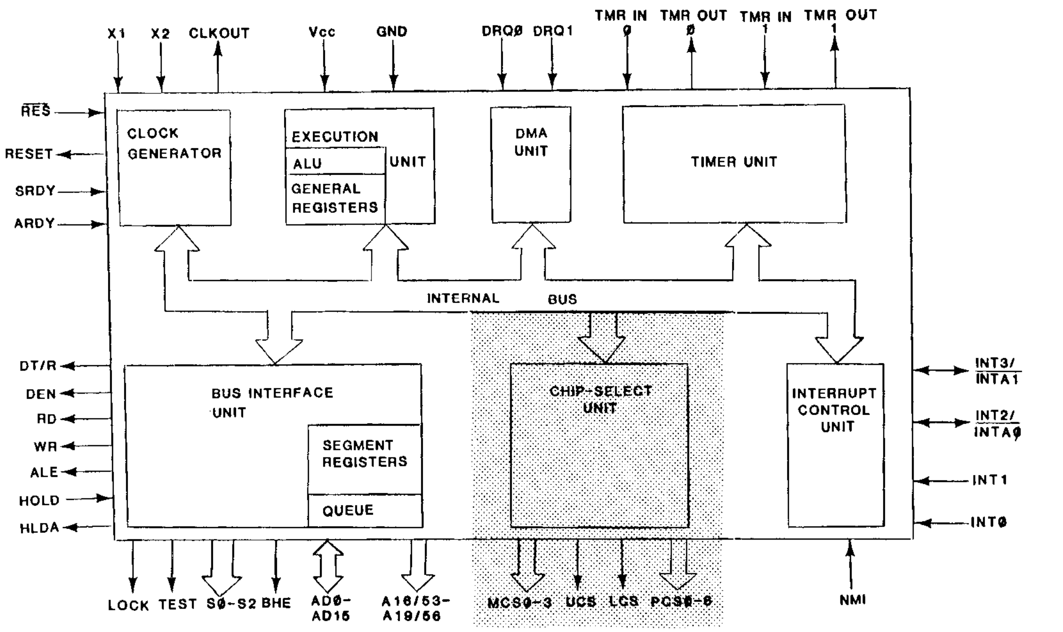
DMA CONTROLLER INTERFACE



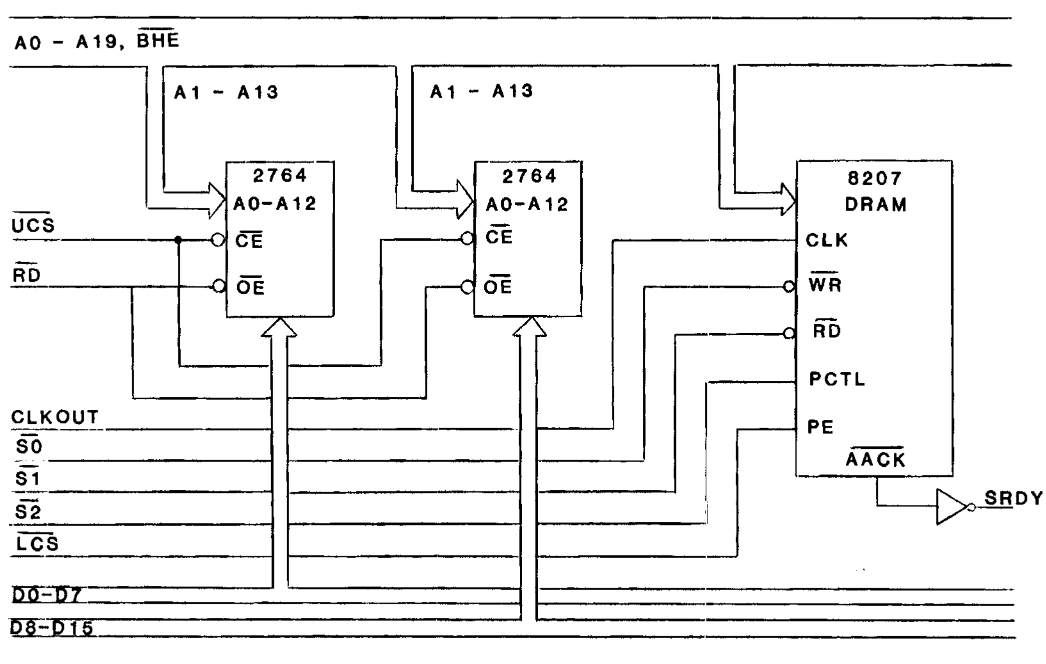
USING DMA REQUEST AND SENDING AN ACKNOWLEDGE



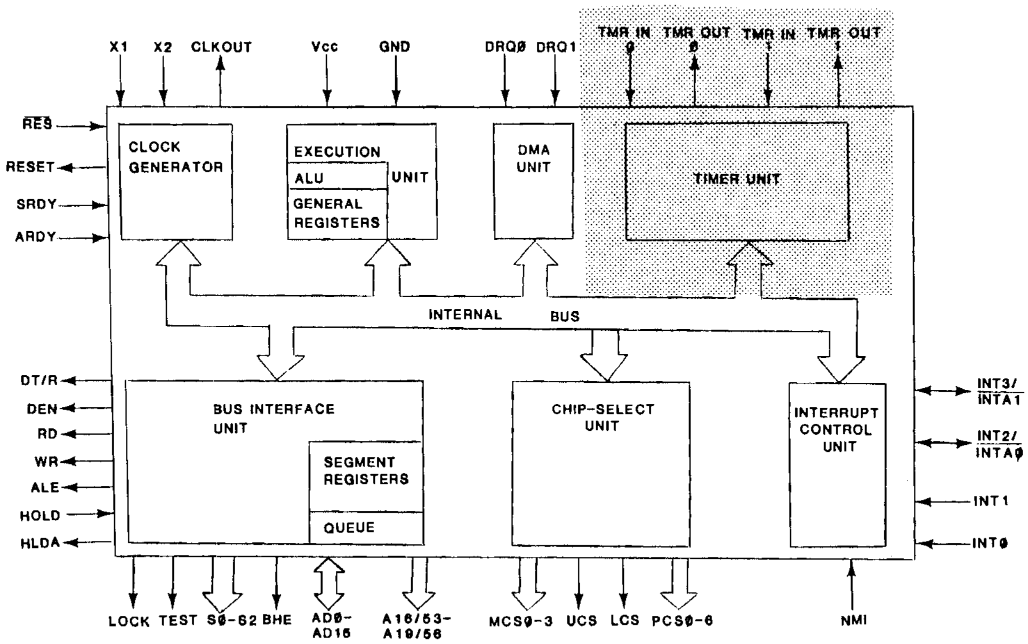
CHIP SELECTS



USING 80186 CHIP SELECTS

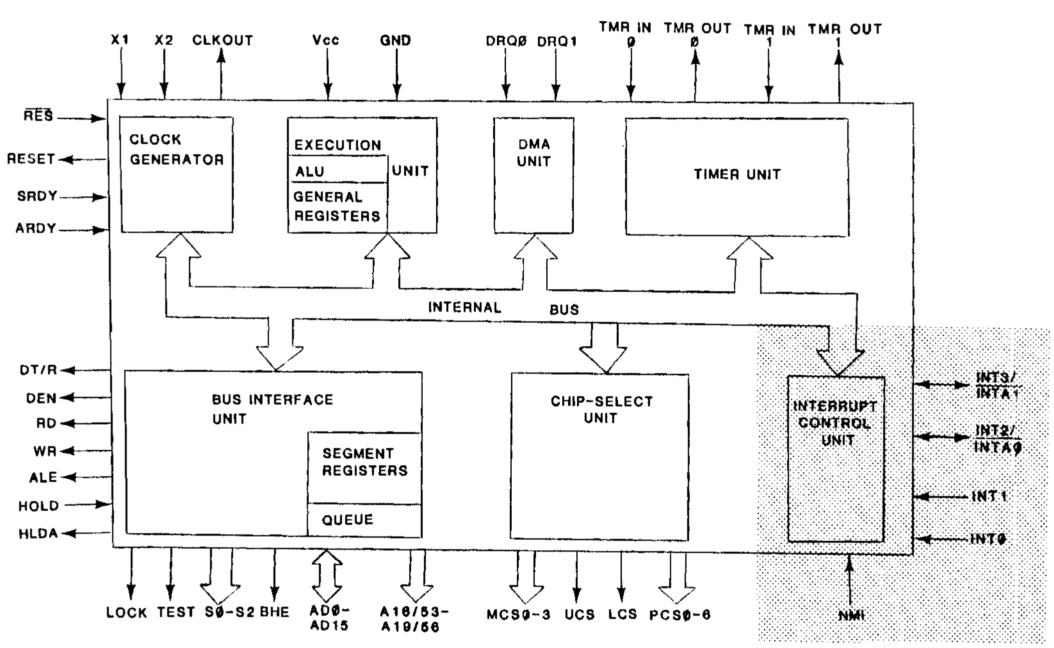


TIMER UNIT



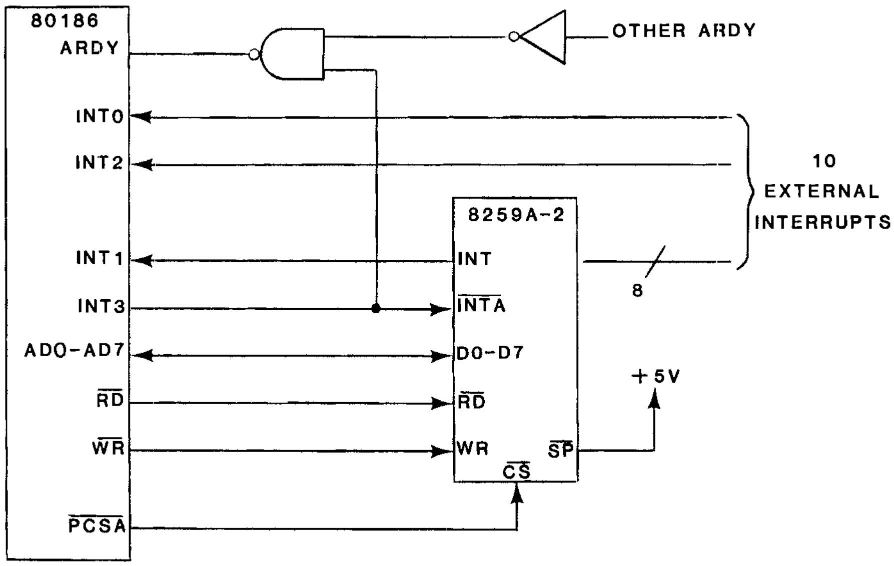
20-21

INTERRUPT CONTROLLER



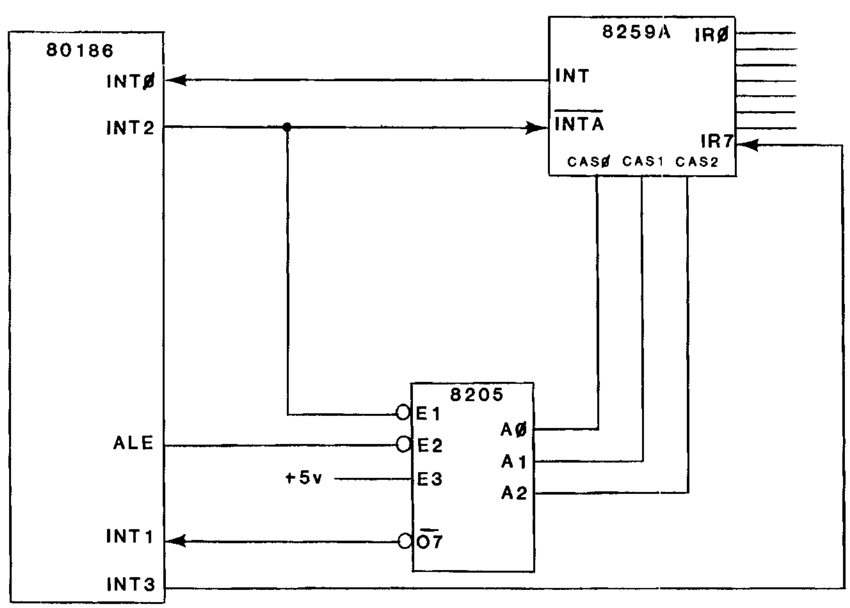
20-22

NON-iRMX86 DIRECT INPUT MODE AND CASCADE MODE

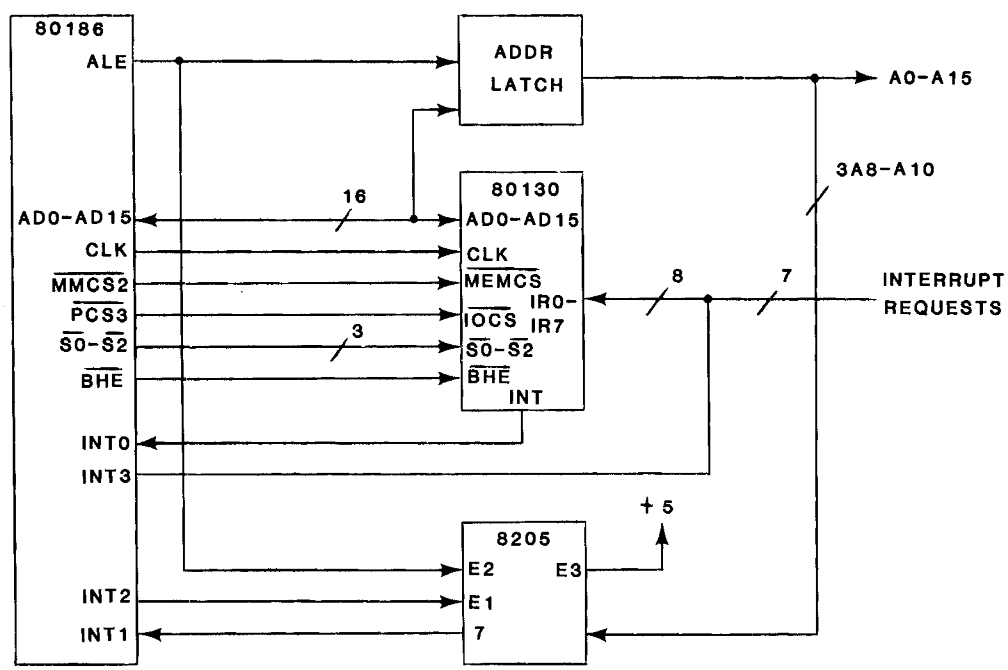


20-23

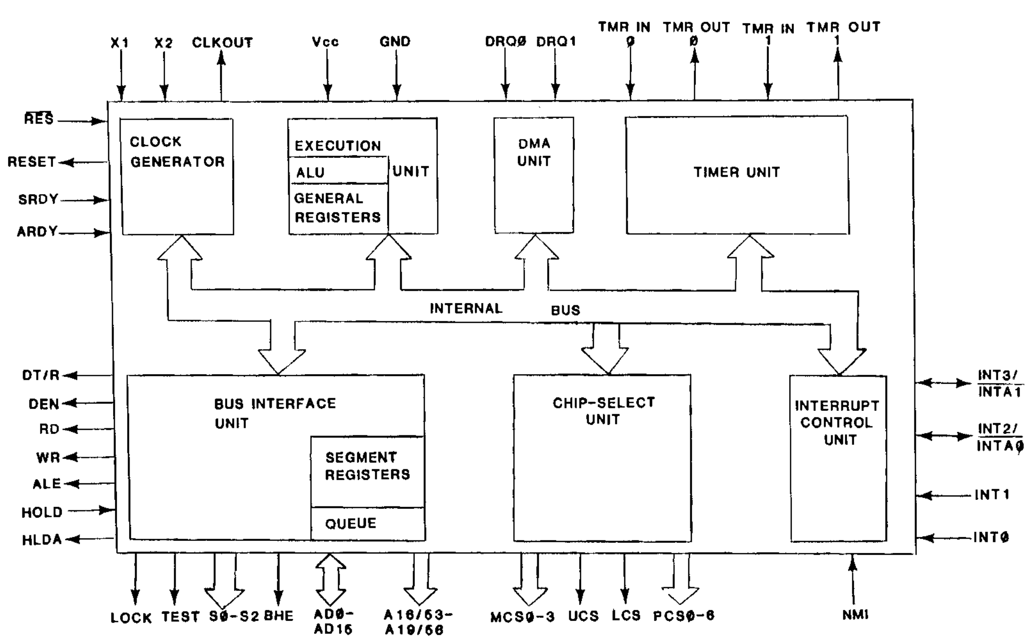
iRMX86 MODE



iRMX86 MODE INTERFACE TO 80130



80186/80188 BLOCK DIAGRAM



80186/80188 DIFFERENCES

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

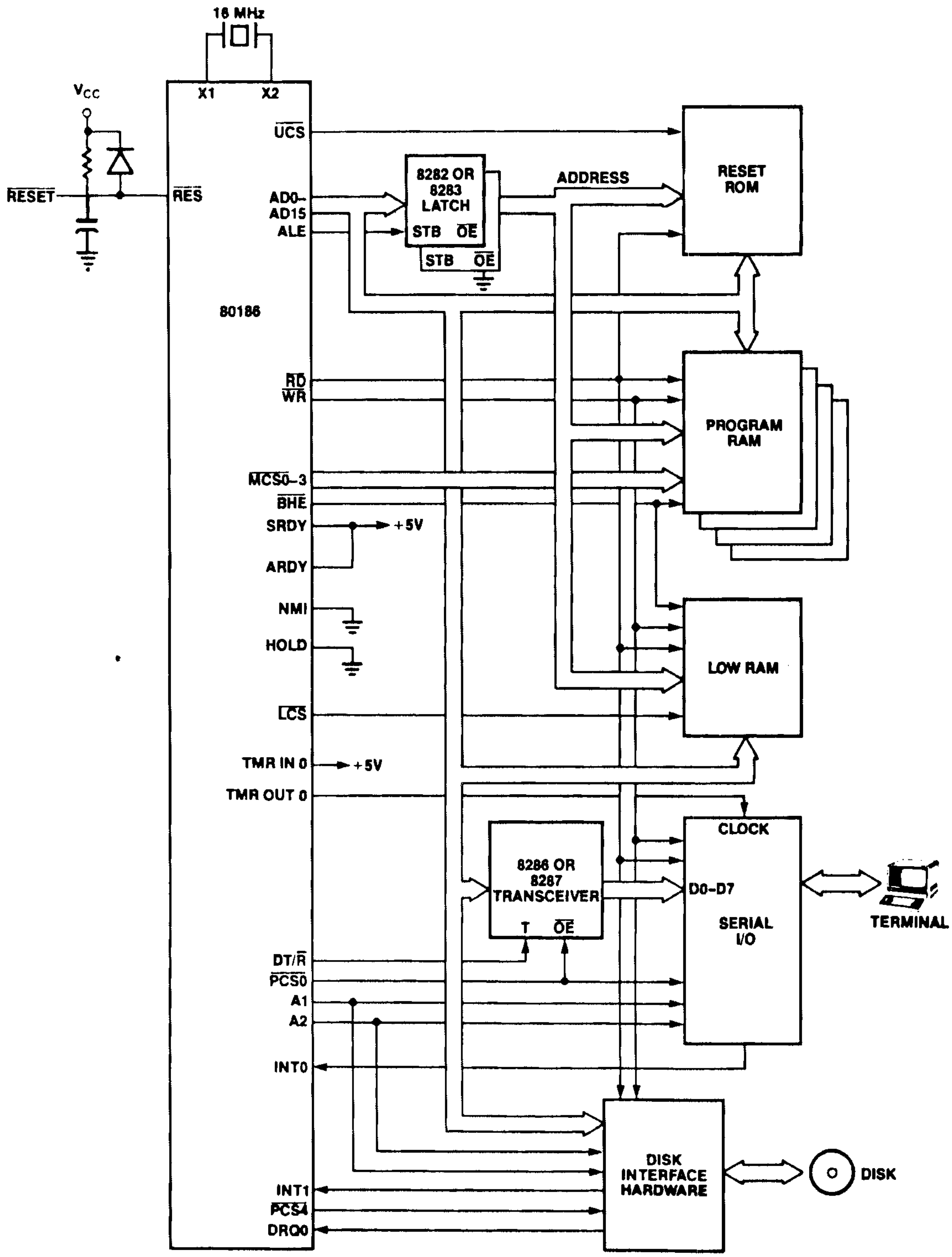
AD8 - AD15 ON THE 80186 ARE TRANSFORMED TO A8 - A15 ON THE 80188 AND ARE VALID THROUGHOUT THE BUS CYCLE.

$\overline{BHE}$/S7 IS ALWAYS DEFINED HIGH BY THE 80188.

THE DMA CONTROLLER ONLY PERFORMS BYTE TRANSFERS.

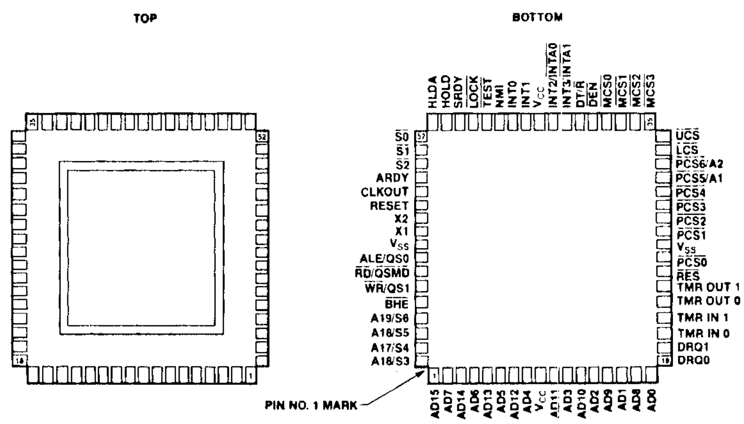
EXECUTION TIMES FOR MEMORY ACCESSES ON THE 80188 ARE INCREASED BECAUSE OF 8-BIT EXTERNAL DATA BUS. INTERNAL DATA BUS IS STILL 16-BITS.

TYPICAL iAPX 186, 188 COMPUTER SYSTEM



• $\overline{BHE}$ NOT IMPLEMENTED ON iAPX 188

iAPX 186,188 PINOUT



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

21-1

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

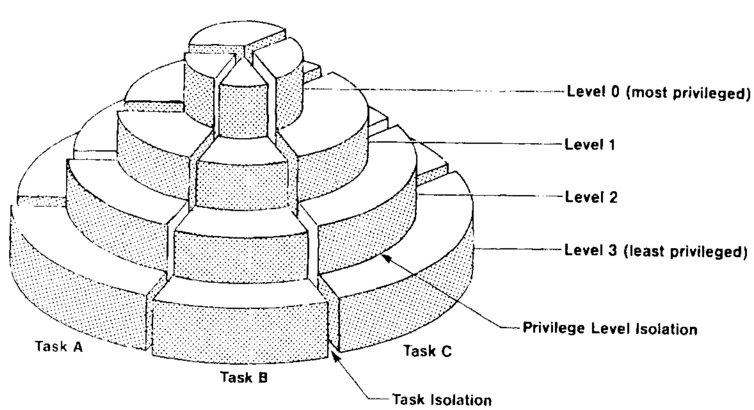
21-2

iAPX 286 PROTECTED VIRTUAL MODE SATISFIES SYSTEM REQUIREMENTS

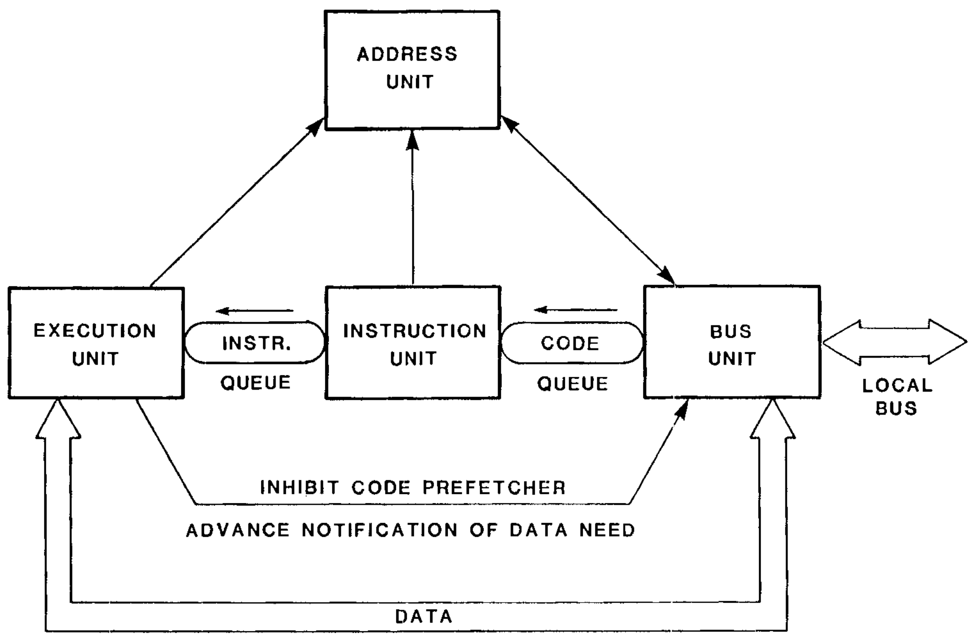
- ADVANCED MEMORY MANAGEMENT WITH NO PERFORMANCE PENALTY
  —16 MBYTE PHYSICAL ADDRESS
  —1 BILLION BYTE VIRTUAL ADDRESS/TASK
  —VIRTUAL MEMORY SUPPORT—INSTRUCTION RESTART
- ADVANCED PROTECTION MECHANISM
  —AUTOMATIC INTEGRITY CHECKS (CODE AND DATA TYPING, SIZE, AND PRIVILEGE)
  —TASK ISOLATION CONTROL (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

21-3

MEMORY PROTECTION

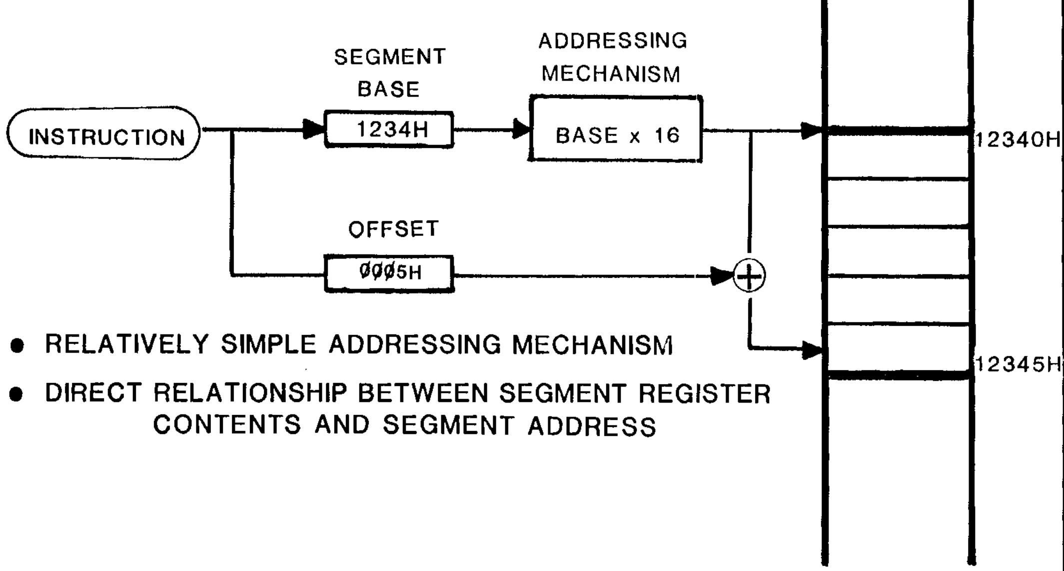


PIPELINED ARCHITECTURE



ACCESSING MEMORY

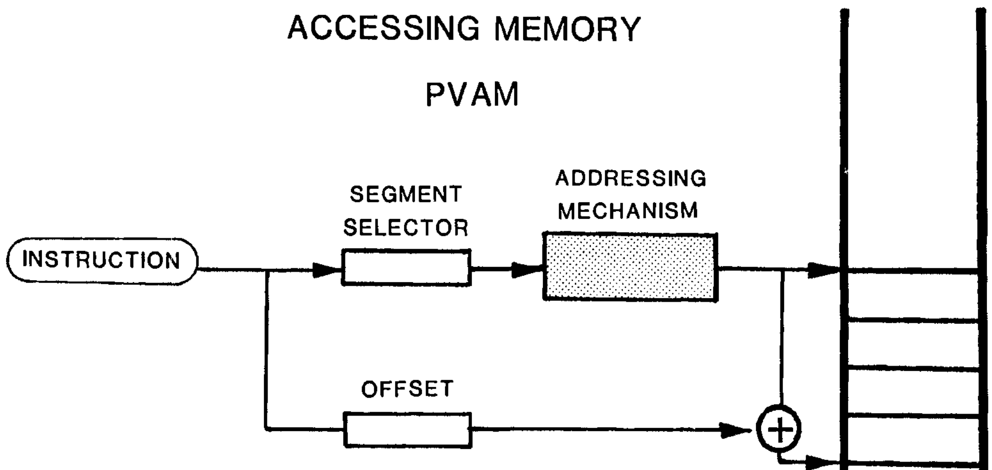
REAL ADDRESS MODE



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

ACCESSING MEMORY

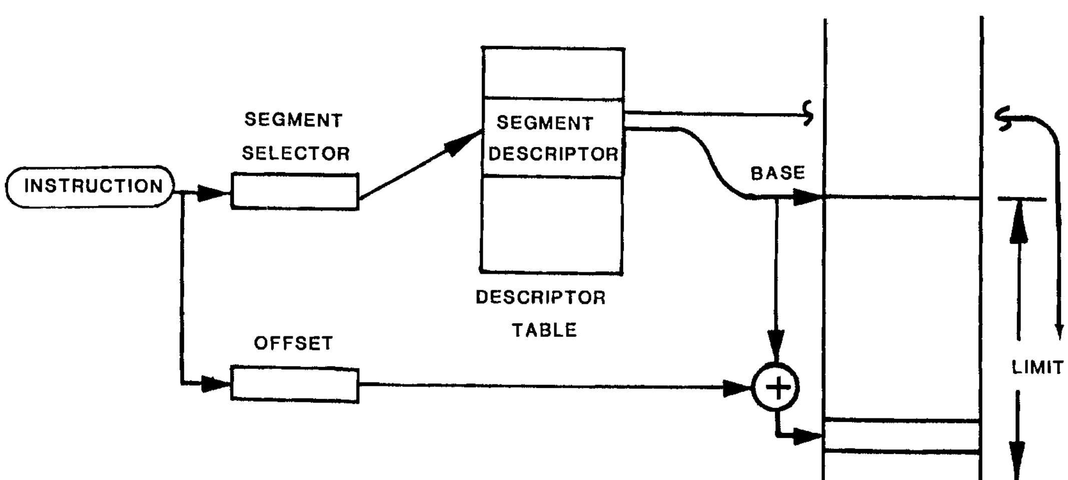
PVAM



- 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

PVAM ADDRESSING MECHANISM



- 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	:2.5 USEC
	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

21-9

BEYOND

286

PERFORMANCE

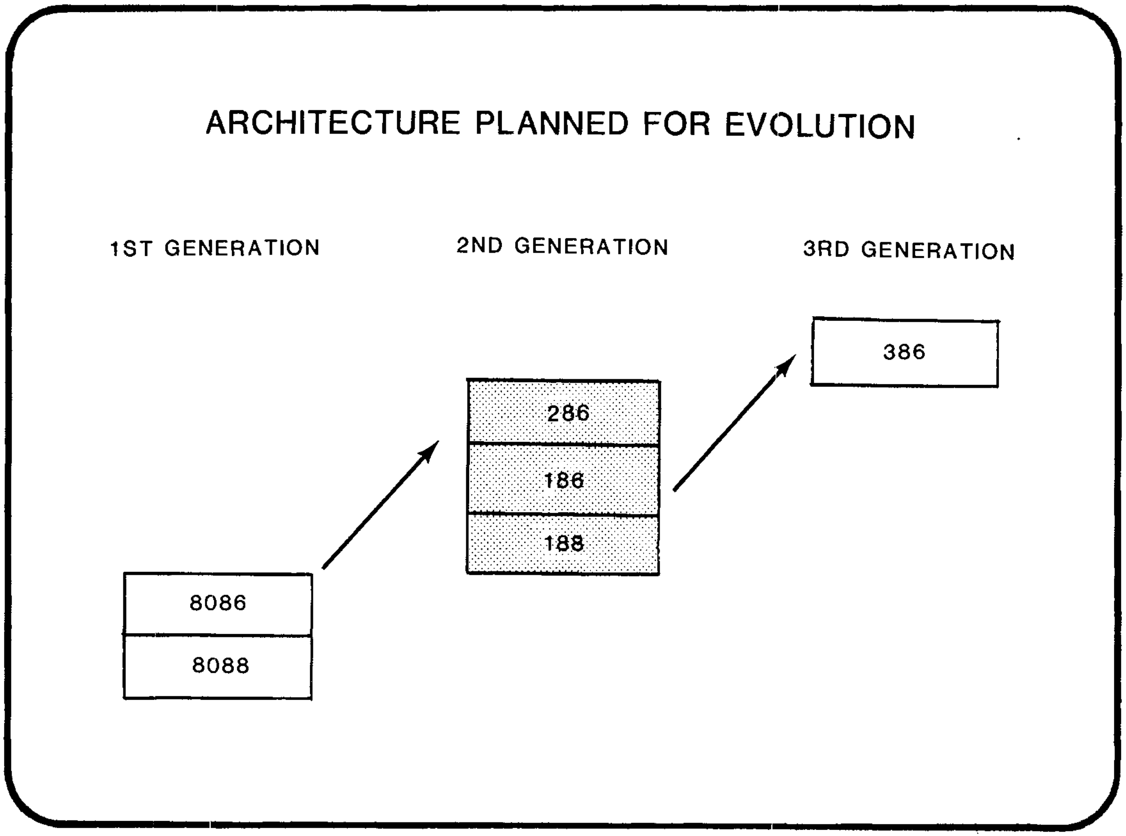
iAPX 386

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

21-11

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
2ND GENERATION
3RD GENERATION
386
286
186
188
8086
8088

APPENDIX A

LAB EXERCISES

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 Ø. 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 Ø 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 8Ø86 processor in the development system (ISIS uses the 8Ø85 processor).

ASM86 — is the program that you want the 8Ø86 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.

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 8Ø86 object code to be executed on an 8Ø86 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 8Ø86 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).

LAB 1

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 8Ø86 processor every 5Ø msec. When interrupted, the 8Ø86 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 8Ø86 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 Ø. 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.
```

LAB 2

When you finish this lab you will be able to:

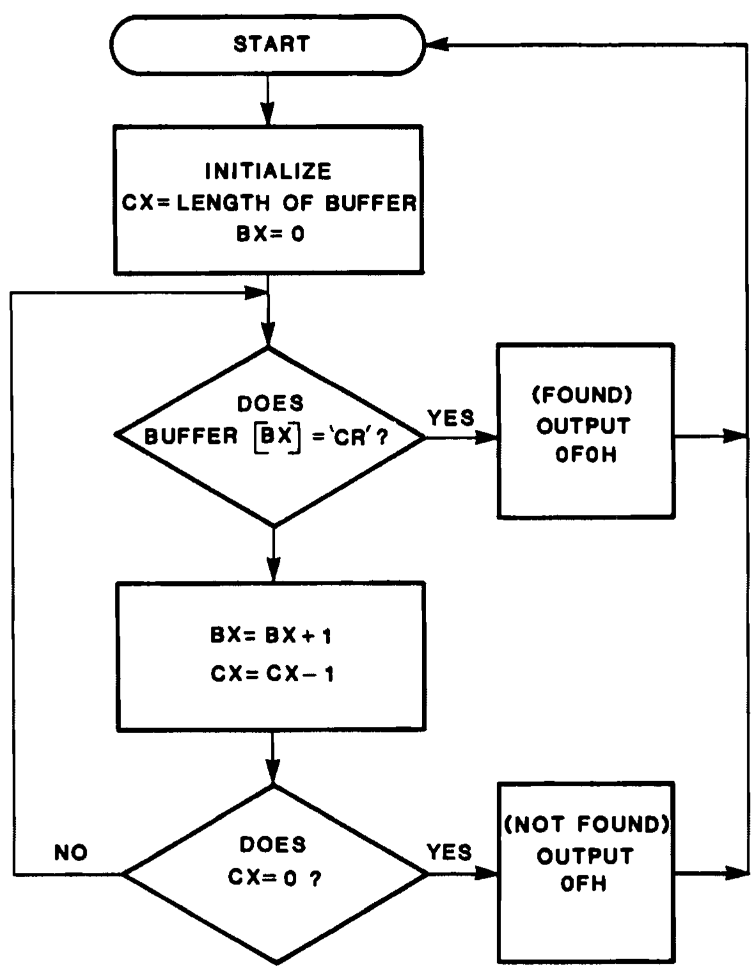
* Define and access a data array
* Debug using DEBUG-86 symbolic debugger

PROBLEM (part 1)

Using the flow chart in the following text, write a program that will continuously search a 5Ø byte array called BUFFER for the ASCII code for return (ØDH). If a return is found, the program should output FØH (for FOund) to port Ø LEDs and continue looking from the beginning of the buffer.

If a return is not found, the program should output ØFH 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.



LAB 2

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 8Ø86 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 <u>Intellec Series III Microcomputer Development System Pocket Reference Card</u> with a full explanation in the <u>Intellec Series III Microcomputer Development System Console Operating Instructions</u> manual Chapter 6.

To load the program into memory type:

```
LOAD :F1:LAB2
```

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:Ø TO CS:2Ø
```

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:Ø TO CS:2Ø
```

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
```

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 5ØT
```

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 5ØT indicates base ten. The default base is hex.

Fill the buffer with all zeroes by typing:

```
BYT .BUFFER LEN 5ØT = Ø
```

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 (ØDH) in the buffer and see if your program finds it, type:

```
BYT .BUFFER+1ØT = ØDH
```

This will place a ØDH 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.

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
```

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:Ø). 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 Ø, type:

```
PORT Ø
```

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 Ø. 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 = Ø, COUNT = Ø
2) IF BUFFER[BX] = ØDH 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 = Ø THEN OUTPUT ØFFH 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 8Ø86 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 8Ø86 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).

LAB 3

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

LAB 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 8Ø86 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 "¦" (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.

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 Ø by that set on port 1. The result (port Ø/port 1) should be displayed on the port Ø LEDs. This should be done in a continuous loop.

A divide error may occur. For example, if the port 1 switches were Ø then the answer of infinity cannot be represented. You will have to write an interrupt service routine for the type Ø 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 Ø = .(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 Ø 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 mistoke that you have to correct.
AEDIT will allow you to correct the 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.

AEDIT Basics Lab

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 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:

Arrow keys	Moves cursor right, left, up, or down.
Right arrow-HOME	Move cursor to end of line.
Left arrow-HOME	Move cursor to the beginning of the line.
Down arrow-HOME	Move cursor to the next page.
Up arrow-HOME	Move cursor to previous page.

Delete commands:

Ctrl-X	Delete all characters left of the cursor to the beginning of the line.
Ctrl-A	Delete all characters right of the cursor to the end of the line.
Ctrl-Z	Delete line.
Ctrl-U	Undo a Ctrl-A, X, or Z.
Ctrl-F	Delete character under cursor.
RUBOUT	Delete the preceeding character.

Menu commands:

Insert	Insert text before cursor.
Xchange	Type over characters under cursor.
Block	Allows you to delimit a block of characters with the following subcommands:
Buffer	Store delimitted block in buffer.
Delete	Delete delimitted block and store it in the buffer.
Get	If responded to with a return, gets the contents of the block buffer.
Quit	Ends the editing session with the following subcommands:
Abort	Quit with all changes lost.
Exit	Write new file to disk and quit.
Init	Edit a new file without returning to the operating system.
Update	Update your file without returning to the operating system.
Write	Writes contents of file to the named file without returning to the operating system.
Esc	Takes you back to the command mode.
Ctrl-C	Aborts the command and returns you to the command mode.

APPENDIX B

LAB SOLUTIONS

```
SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB1A
OBJECT MODULE PLACED IN :F2:LAB1A.OBJ
ASSEMBLER INVOKED BY:  :F3:ASM86.86 :F2:LAB1A.ASM


LOC  OBJ                  LINE     SOURCE

                                           NAME    LAB1A

  0000                             LEDS    EQU     0       ;LED PORT
  0001                             SWITCH  EQU     1       ;SWITCH PORT

----                               CODE    SEGMENT
                                   ASSUME  CS:CODE

0000 E401                          START:  IN      AL,SWITCH
0002 E600                                  OUT     LEDS,AL
0004 EBFA                                  JMP     START

----                               CODE    ENDS
                                           END     START

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

```
SERIES-III 8086/87/88/186 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 SYMBOLS DEBUG


LOC  OBJ                LINE     SOURCE

                                 NAME      LAB1_PART2

 0000                            LEDS     EQU      0
 0001                            SWITCH   EQU      1
 0001                            PATTERN  EQU      01H          ;LED PATTERN

----                             CODE     SEGMENT
                                          ASSUME CS:CODE
0000 B001                        START:   MOV      AL,PATTERN
0002 E600                        AGAIN:   OUT      LEDS,AL      ;OUTPUT PATTERN

0004 B90500                               MOV      CX,5         ;5 TIMES FOR 1 SEC
0007 8BD1                        OUTER:   MOV      DX,CX        ;SAVE IT FOR LATER
0009 B9FFFF                               MOV      CX,0FFFFH    ;.2 SEC DELAY
000C E2FE                        INNER:   LOOP     INNER
000E 8BCA                                 MOV      CX,DX        ;GET IT BACK
0010 E2F5                                 LOOP     OUTER        ;TO DO IT 5 TIMES

0012 D0C8                                 ROR      AL,1         ;ROTATE PATTERN
0014 EBEC                                 JMP      AGAIN        ;REPEAT
----                             CODE     ENDS
                                 END      START
```

```
SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB1_PART3
OBJECT MODULE PLACED IN :F2:LAB1C.OBJ
ASSEMBLER INVOKED BY:  :F3:ASM86.86 :F2:LAB1C.ASM SYMBOLS DEBUG


LOC  OBJ              SOURCE

                      NAME    LAB1_PART3

 0000                 LEDS    EQU     0
 0001                 SWITCH  EQU     1
 0001                 PATTERN EQU     01H             ;LED PATTERN

----                  CODE    SEGMENT
                              ASSUME CS:CODE
0000 B001             START:  MOV     AL,PATTERN
0002 E600             AGAIN:  OUT     LEDS,AL         ;OUTPUT PATTERN
0004 8AD8                     MOV     BL,AL           ;SAVE PATTERN

0006 E401                     IN      AL,SWITCH       ;DELAY TIME IS SET BY
0008 B400                     MOV     AH,0            ; SWITCHES
000A 8BC8                     MOV     CX,AX
000C E30B                     JCXZ    CONTIN          ;IF CX IS ZERO, THEN
                                                      ;SKIP DELAY. OTHERWISE
                                                      ;DELAY WOULD BE TOO LONG
000E 8BD1             OUTER:  MOV     DX,CX           ;SAVE IT FOR LATER
0010 B9FFFF                   MOV     CX,0FFFFH       ;.2 SEC DELAY
0013 E2FE             INNER:  LOOP    INNER
0015 8BCA                     MOV     CX,DX           ;GET IT BACK
0017 E2F5                     LOOP    OUTER           ;TO DO IT 5 TIMES

0019 8AC3             CONTIN: MOV     AL,BL           ;PUT PATTERN BACK
001B D0C8                     ROR     AL,1            ;ROTATE PATTERN
001D EBE3                     JMP     AGAIN           ;REPEAT
----                  CODE    ENDS
                      END     START
```

SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB2
OBJECT MODULE PLACED IN :F2:LAB2.OBJ
ASSEMBLER INVOKED BY: :F3:ASM86.86 :F2:LAB2.ASM SYMBOLS DEBUG

```
LOC  OBJ                LINE     SOURCE

                                 ;THIS PROGRAM IMPLEMENTS THE FLOWCHART GIVEN IN LAB 2

                                         NAME    LAB2

  000D                           CR      EQU     0DH              ;CARRIAGE RETURN
  00F0                           FOUND   EQU     0F0H             ;LED PATTERN IF CR IS FOUND
  000F                           NFOUND  EQU     0FH              ;LED PATTERN IF CR IS NOT FOUND
  0000                           LED     EQU     0                ;LED PORT

----                             DATA    SEGMENT
0000 (50                         BUFFER  DB      50 DUP (?)
     ??
     )
----                             DATA    ENDS

----                             CODE    SEGMENT
                                         ASSUME  CS:CODE,DS:DATA
0000 B8----          R           START:  MOV     AX,DATA                 ;INITIALIZE DS SEGMENT
0003 8ED8                                MOV     DS,AX
0005 B93200                      AGAIN:  MOV     CX,LENGTH BUFFER        ;LOAD CX FOR LOOP COUNT
0008 33DB                                XOR     BX,BX                   ;INITIALIZE INDEX
000A 803F0D                      CHECK:  CMP     BUFFER[BX],CR           ;CHECK CONTENTS OF BUFFER FOR 0DH
000D 7409                                JE      FNDIT                   ;JMP IF CR WAS FOUND
000F 43                                  INC     BX                      ;BUMP INDEX
0010 E2F8                                LOOP    CHECK                   ;DO IT AGAIN

                                 ;IF THE CPU FALLS OUT OF THE LOOP TO THIS LOCATION THEN
                                 ; A CR WAS NOT FOUND
0012 B00F                        NFD:    MOV     AL,NFOUND               ;SIGNAL OPERATOR THAT CR
0014 E600                                OUT     LED,AL                  ; WAS NOT FOUND
0016 EBED                                JMP     AGAIN

                                 ;IF THE CPU JUMPS HERE THEN A CR WAS FOUND
0018 B0F0                        FNDIT:  MOV     AL,FOUND                ;SIGNAL OPERATOR THAT CR
001A E600                                OUT     LED,AL                  ; WAS FOUND
001C EBE7                                JMP     AGAIN
----                             CODE    ENDS
                                         END     START
```

```
SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB2_PART2
OBJECT MODULE PLACED IN :F2:LAB2B.OBJ
ASSEMBLER INVOKED BY:  :F3:ASM86.86 :F2:LAB2B.ASM SYMBOLS DEBUG


LOC  OBJ               LINE     SOURCE

                          1             NAME    LAB2_PART2
                          2
 000D                     3     CR      EQU     0DH                     ;CARRIAGE RETURN
 0000                     4     LEDS    EQU     0                       ;PORT FOR LEDS
 00FF                     5     NO_CR   EQU     0FFH                    ;LED PATTERN IF CR NOT FOUND
                          6
----                      7     DATA    SEGMENT
0000 ??                   8     COUNT   DB      ?
0001 (50                  9     BUFFER  DB      50 DUP(?)
     ??
     )
----                     10     DATA    ENDS
                         11
----                     12     CODE    SEGMENT
                         13             ASSUME  CS:CODE,DS:DATA
                         14
0000 B8----        R     15     START:  MOV     AX,DATA
0003 8ED8                16             MOV     DS,AX                   ;INITIALIZE DS
0005 B93200              17     AGAIN:  MOV     CX,LENGTH BUFFER        ;SET CX WITH LOOP COUNT
0008 33DB                18             XOR     BX,BX                   ;INITIALIZE INDEX
000A C606000000          19             MOV     COUNT,0                 ;INITIALIZE COUNT
                         20
000F 807F010D            21     CHECK:  CMP     BUFFER[BX],CR           ;LOOK FOR CR
0013 7504                22             JNE     NFIND                   ;IF NO CR THEN DON'T COUNT IT
0015 FE060000            23             INC     COUNT                   ;ELSE COUNT IT
0019 43                  24     NFIND:  INC     BX                      ;BUMP INDEX
001A E2F3                25             LOOP    CHECK
                         26
001C 803E000000          27             CMP     COUNT,0                 ;IF COUNT IS ZERO
0021 7407                28             JE      NONFND                  ;THEN PUT OUT NONE FOUND CODE
                         29
0023 A00000              30             MOV     AL,COUNT                ;ELSE PUT OUT NUMBER OF CR
0026 E600                31             OUT     LEDS,AL
0028 EBDB                32             JMP     AGAIN
                         33
002A B0FF                34     NONFND: MOV     AL,NO_CR                ;THIS IS WHERE WE PUT OUT
002C E600                35             OUT     LEDS,AL                 ; NONE FOUND CODE
002E EBD5                36             JMP     AGAIN
                         37
----                     38     CODE    ENDS
                         39             END     START
```

```
SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB3_PART_1
OBJECT MODULE PLACED IN :F2:LAB3A.OBJ
ASSEMBLER INVOKED BY:  :F3:ASM86.86 :F2:LAB3A.ASM SYMBOLS DEBUG


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      STACK   SEGMENT
0000 (100                  14              DW      100 DUP(?)
     ????
     )
  00C8                     15      TOP     EQU     THIS WORD
----                       16      STACK   ENDS
                           17
----                       18      CODE    SEGMENT
                           19              ASSUME CS:CODE,SS:STACK
0000 B8----          R     20      START:  MOV     AX,STACK         ;INITIALIZE THE
0003 8ED0                  21              MOV     SS,AX            ; STACK SEGMENT AND
0005 8D26C800              22              LEA     SP,TOP           ; STACK POINTER REGISTERS.
                           23
0009 9A0000----      E     24      AGAIN:  CALL    CI               ;GET CHARACTER FROM THE KEYBOARD
000E 50                    25              PUSH    AX               ;PLACE CHARACTER ONN THE STACK
000F 9A0000----      E     26              CALL    CO               ;OUTPUT IT TO THE CRT
0014 EBF3                  27              JMP     AGAIN
----                       28      CODE    ENDS
                           29      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

                                   ; THIS PROGRAM IS THE SOLUTION TO LAB3 PART 2 OF THE WORKSHOP.
                                   ; IT INPUTS CHARACTERS FROM THE KEYBOARD, ENCRYPTS THEM (ADD
                                   ; ONE TO THE ASCII VALUE) AND THEN OUTPUTS THE RESULT TO THE
                                   ; CRT.  THE PROGRAM ALSO DETECTS WHEN A CR IS INPUT, AND INSERT A LF.

                                           EXTRN   CO:FAR,CI:FAR

                                           NAME    LAB3_PART_2

  000D                             CR      EQU     0DH
  000A                             LF      EQU     0AH

----                               STACK   SEGMENT
0000 (100                                  DW      100 DUP(?)
     ????
     )
00C8                               T_O_S   LABEL   WORD
----                               STACK   ENDS

----                               CODE    SEGMENT
                                           ASSUME CS:CODE,SS:STACK

0000                               ENCRYPT PROC
                                   ; THIS IS A SIMPLE ENCRYPTOR PROCEDURE.  ENCRYPT EXPECTS
                                   ; TO RECEIVE AN ASCII CHARACTER AS A PARAMETER ON THE STACK.
                                   ; IT INCREMENTS THE ASCII VALUE BY ONE AND RETURNS THE
                                   ; ENCRYPTED CHARACTER IN THE AL REGISTER.

0000 55                                    PUSH    BP              ;SAVE BP
0001 8BEC                                  MOV     BP,SP           ;USE AS REFERENCE IN STACK
0003 8B4604                                MOV     AX,[BP+4]       ;GET CHARACTER
0006 FEC0                                  INC     AL              ;INCREMENT IT AND LEAVE IT
0008 5D                                    POP     BP              ; IN AL
0009 C20200                                RET     2               ;DELETES PARAMETER FROM STACK
                                   ENCRYPT ENDP

000C B8----           R            START:  MOV     AX,STACK        ;INITIALIZE STACK
000F 8ED0                                  MOV     SS,AX
0011 8D26C800                              LEA     SP,T_O_S
0015 9A0000----       E            AGAIN:  CALL    CI              ;GET CHARACTER FROM KEYBOARD
001A 3C0D                                  CMP     AL,CR           ;IS IS CARRIAGE RETURN?
001C 740C                                  JE      CRLF            ;IF YES THEN OUTPUT CR/LF
001E 50                                    PUSH    AX              ;PASS CHAR. ON STACK
001F E8DEFF                                CALL    ENCRYPT         ;TRANSFORM IT
0022 50                                    PUSH    AX
0023 9A0000----       E                    CALL    CO              ;OUTPUT CHAR ON SCREEN
0028 EBEB                                  JMP     AGAIN

                                   ;WE SHOULD ONLY BE EXECUTING CRLF IF A CARRIAGE RETURN WAS INPUT
                                   ; CRLF OUTPUTS A CARRIAGE RETURN AND LINE FEED
```

```
LOC  OBJ                  LINE     SOURCE

002A B00D                   49     CRLF:   MOV     AL,CR
002C 50                     50             PUSH    AX
002D 9A0000----      E      51             CALL    CO              ;OUTPUT A CARRIAGE RETURN
0032 B00A                   52             MOV     AL,LF
0034 50                     53             PUSH    AX
0035 9A0000----      E      54             CALL    CO              ;OUTPUT A LINE FEED
003A EBD9                   55             JMP     AGAIN           ;GO BACK TO GET NEXT CHAR.
----                        56     CODE    ENDS
                            57             END     START
```

```
SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB3_PART_3
OBJECT MODULE PLACED IN :F2:LAB3C.OBJ
ASSEMBLER INVOKED BY:  :F3:ASM86.86 :F2:LAB3C.ASM SYMBOLS DEBUG


LOC  OBJ                SOURCE

                        ; THIS PROGRAM IS THE SOLUTION TO LAB3 PART 3 OF THE WORKSHOP.
                        ; IT DOES THE SAME AS PART 2 EXCEPT THE PROCEDURE IS IN
                        ; ANOTHER SEGMENT

                                EXTRN   CO:FAR,CI:FAR

                                NAME    LAB3_PART_3

 000D                   CR      EQU     0DH
 000A                   LF      EQU     0AH

----                    STACK   SEGMENT
0000 (100                       DW      100 DUP(?)
     ????
     )
00C8                    T_O_S   LABEL   WORD
----                    STACK   ENDS

----                    PRO     SEGMENT
                                ASSUME CS:CODE,SS:STACK

0000                    ENCRYPT PROC    FAR
                        ; THIS IS THE SAME PROCEDURE AS PART 2 EXCEPT THE PROCEDURE
                        ; IS IN ANOTHER SEGMENT AND IS FAR AND THE PARAMETER IS NOW
                        ; SIX BYTES FROM THE TOP OF THE STACK

0000 55                         PUSH    BP              ;SAVE BP
0001 8BEC                       MOV     BP,SP           ;USE AS REFERENCE IN STACK
0003 8B4606                     MOV     AX,[BP+6]       ;GET CHARACTER
0006 FEC0                       INC     AL              ;INCREMENT IT AND LEAVE IT
0008 5D                         POP     BP              ; IN AL
0009 CA0200                     RET     2               ;DELETES PARAMETER FROM STACK
                        ENCRYPT ENDP
----                    PRO     ENDS

----                    CODE    SEGMENT
                                ASSUME CS:CODE,SS:STACK

0000 B8----        R    START:  MOV     AX,STACK        ;INITIALIZE STACK
0003 8ED0                       MOV     SS,AX
0005 8D26C800                   LEA     SP,T_O_S
0009 9A0000----    E    AGAIN:  CALL    CI              ;GET CHARACTER FROM KEYBOARD
000E 3C0D                       CMP     AL,CR           ;IS IS CARRIAGE RETURN?
0010 740E                       JE      CRLF            ;IF YES THEN OUTPUT CR/LF
0012 50                         PUSH    AX              ;PASS CHAR. ON STACK
0013 9A0000----    R            CALL    ENCRYPT         ;TRANSFORM IT
0018 50                         PUSH    AX
0019 9A0000----    E            CALL    CO              ;OUTPUT CHAR ON SCREEN
001E EBE9                       JMP     AGAIN
```

```
LOC  OBJ                  LINE    SOURCE

                            49    ;WE SHOULD ONLY BE EXECUTING CRLF IF A CARRIAGE RETURN WAS INPUT
                            50    ; CRLF OUTPUTS A CARRIAGE RETURN AND LINE FEED
0020 B00D                   51    CRLF:   MOV     AL,CR
0022 50                     52            PUSH    AX
0023 9A0000----      E      53            CALL    CO              ;OUTPUT A CARRIAGE RETURN
0028 B00A                   54            MOV     AL,LF
002A 50                     55            PUSH    AX
002B 9A0000----      E      56            CALL    CO              ;OUTPUT A LINE FEED
0030 EBD7                   57            JMP     AGAIN           ;GO BACK TO GET NEXT CHAR.
----                        58    CODE    ENDS
                            59            END     START
```

```
SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB3_PART_3
OBJECT MODULE PLACED IN :F2:LAB3D.OBJ
ASSEMBLER INVOKED BY:  :F3:ASM86.86 :F2:LAB3D.ASM SYMBOLS DEBUG


LOC  OBJ                  LINE     SOURCE

                             1     ; THIS PROGRAM IS THE SOLUTION TO LAB3 PART 4 OF THE WORKSHOP.
                             2     ; IT INPUTS CHARACTERS FROM THE KEYBOARD, AND OUTPUTS THEM TO
                             3     ; THE CRT.  IT ALSO IMPLEMENTS A SHIFT LOCK FEATURE.  BY TYPING
                             4     ; AN UPPER CASE BACK SLASH "\" ALL SUBSEQUENT LOWER CASE ALPHA CHARACTERS
                             5     ; WILL BE CONVERTED TO UPPER CASE.  TYPING THE UPPER CASE BACK SLASH
                             6     ; AGAIN RETURNS THE OUTPUT TO UPPER AND LOWER CASE AGAIN.
                             7
                             8             EXTRN   CO:FAR,CI:FAR
                             9
                            10             NAME    LAB3_PART_3
                            11
  000D                      12     CR              EQU     0DH
  000A                      13     LF              EQU     0AH
  007C                      14     LOCK_KEY        EQU     7CH     ;SHIFT LOCK KEY (ASCII)
  0000                      15     NULL            EQU     00H     ;NULL ASCII CHARACTER
                            16
----                        17     STACK   SEGMENT
0000 (100                   18             DW      100 DUP(?)
     ????
     )
00C8                        19     T_O_S   LABEL   WORD
----                        20     STACK   ENDS
                            21
----                        22     CODE    SEGMENT
                            23             ASSUME CS:CODE,SS:STACK
0000 00                     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     ;TOGGLING A MEMORY BASED FLAG WHICH INDICATES IF THE SHIFT IS
                            32     ;CURRENTLY LOCKED OR NOT.  NOTE: THIS LOCK ONLY AFFECTS ALPHA
                            33     ;CHARACTERS AND S 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 8B4604                 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 2E80360000080          42             XOR     SHFTFLG,80H     ;TOGGLE SHIFT FLAG
0011 B000                   43             MOV     AL,NULL         ;AND DON'T OUTPUT ANYTHING
0013 EB1390                 44             JMP     DONE
0016 2EF606000080           45     TST:    TEST    SHFTFLG,80H     ;LOOK AT SHIFT FLAG STATUS
001C 740A                   46             JZ      DONE            ;IF CLEAR, RETURN THE UNALTERED CHAR.
001E 3C60                   47             CMP     AL,60H          ;IF SET, LOOK
0020 7206                   48             JB      DONE            ;FOR LOWER CASE
```

```
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
                          54     SHIFT  ENDP
                          55
002C B8----         R     56     START: MOV    AX,STACK      ;INITIALIZE STACK
002F 8ED0                 57            MOV    SS,AX
0031 8D26C800             58            LEA    SP,T_O_S
0035 9A0000----     E     59     AGAIN: CALL   CI            ;GET CHARACTER FROM KEYBOARD
003A 3C0D                 60            CMP    AL,CR         ;IS IS CARRIAGE RETURN?
003C 740C                 61            JE     CRLF          ;IF YES THEN OUTPUT CR/LF
003E 50                   62            PUSH   AX            ;PASS CHAR. ON STACK
003F E8BFFF               63            CALL   SHIFT         ;CONVERT TO UPPER CASE IF SHIFT LOCKED
0042 50                   64            PUSH   AX
0043 9A0000----     E     65            CALL   CO            ;OUTPUT CHAR ON SCREEN
0048 EBEB                 66            JMP    AGAIN
                          67
                          68     ;WE SHOULD ONLY BE EXECUTING CRLF IF A CARRIAGE RETURN WAS INPUT
                          69     ; CRLF OUTPUTS A CARRIAGE RETURN AND LINE FEED
004A B00D                 70     CRLF:  MOV    AL,CR
004C 50                   71            PUSH   AX
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
005A EBD9                 76            JMP    AGAIN         ;GO BACK TO GET NEXT CHAR.
----                      77     CODE   ENDS
                          78            END    START
```

SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE INTERRUPT_HANDLER
OBJECT MODULE PLACED IN :F2:LAB3E.OBJ
ASSEMBLER INVOKED BY: :F3:ASM86.86 :F2:LAB3E.ASM SYMBOLS DEBUG

```
LOC  OBJ                 SOURCE

                         ;THIS IS THE OPTIONAL EXERCISE TO WRITE AN INTERRUPT HANDLING ROUTINE
                         ;THIS WILL HANDLE THE INTERRUPT FOR DIVIDE ERROR

                                         NAME    INTERRUPT_HANDLER

 0000                    DIVIDEND        EQU     0                ;PORT FOR DIVIDEND
 0001                    DIVISOR         EQU     1                ;PORT FOR DIVISOR
 0000                    QUOTIENT        EQU     0                ;ANSWER OUTPUT HERE
 0001                    ERROR           EQU     1                ;OR IF ERROR THESE WILL FLASH

----                     INTERRUPT       SEGMENT AT 0
0000 ????                DIV_ERR_IP      DW      ?                ;OFFSET TO BE LOADED
0002 ????                DIV_ERR_CS      DW      ?                ;SEGMENT TO BE LOADED
----                     INTERRUPT       ENDS

----                     STACK           SEGMENT
0000 (100                                DW      100 DUP (?)
     ????
     )
00C8                     TOP             LABEL   WORD
----                     STACK           ENDS

----                     DIVIDE          SEGMENT
                                         ASSUME  CS:DIVIDE

0000 00                  ALARM           DB      0                ;HOLDS PATTERN TO LEDS

0001 50                  DIVIDE_ERROR:   PUSH    AX               ;SAVE REGISTERS USED
0002 51                                  PUSH    CX
0003 2EF6160000                          NOT     ALARM            ;COMPLEMENT LED PATTERN
0008 2EA00000                            MOV     AL,ALARM         ;GET THE FLASH VALUE
000C E601                                OUT     ERROR,AL         ;AND SEND IT OUT

000E B90300                              MOV     CX,3             ;DELAY ABOUT .6 SEC
0011 8BC1                OUTER:          MOV     AX,CX
0013 B9FFFF                              MOV     CX,0FFFFH
0016 E2FE                INNER:          LOOP    INNER
0018 8BC8                                MOV     CX,AX
001A E2F5                                LOOP    OUTER

001C 59                                  POP     CX               ;GET BACK REGISTERS
001D 58                                  POP     AX
001E CF                                  IRET                     ;AND RETURN

----                     DIVIDE          ENDS

----                     MAIN            SEGMENT
                                         ASSUME  CS:MAIN,DS:INTERRUPT,SS:STACK

0000 B8----         R    START:          MOV     AX,STACK         ;INITIALIZE STACK
```

```
LOC  OBJ              SOURCE

0003 8ED0                                MOV     SS,AX
0005 8D26C800                            LEA     SP,TOP
0009 B80000                              MOV     AX,INTERRUPT
000C 8ED8                                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                        MOV     DIV_ERR_IP,OFFSET DIVIDE_ERROR
0014 C7060200----   R                    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             AGAIN:             IN      AL,DIVISOR       ;GET VALUE TO DIVIDE BY
001C 8AD8                                MOV     BL,AL            ;AND SAVE IT
001E E400                                IN      AL,DIVIDEND      ;GET WHAT TO DIVIDE BY
0020 32E4                                XOR     AH,AH            ;AND CONVERT IT TO A WORD
0022 F6F3                                DIV     BL
0024 E600                                OUT     QUOTIENT,AL      ;OUTPUT DIVISION RESULT TO LEDS
0026 EBF2                                JMP     AGAIN            ;DO THIS CONTINUOUSLY
----                  MAIN               ENDS
                                         END     START
```

```
iNDX-S41 (V2.1) 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB4_PART_1_MAIN
OBJECT MODULE PLACED IN :F1:LAB4A1.OBJ
ASSEMBLER INVOKED BY:  /SW/ASM86 :F1:LAB4A1.ASM SB DB


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
  000D                       9     CR      EQU     0DH
  000A                      10     LF      EQU     0AH
                            11
----                        12     STACK   SEGMENT
0000 (100                   13             DW      100 DUP(?)
     ????
     )
00C8                        14     T_O_S   LABEL   WORD
----                        15     STACK   ENDS
                            16
                            17
----                        18     CODE    SEGMENT
                            19             ASSUME CS:CODE,SS:STACK
                            20
0000 B8----          R      21     START:  MOV     AX,STACK          ;INITIALIZE STACK
0003 8ED0                   22             MOV     SS,AX
0005 8D26C800               23             LEA     SP,T_O_S
0009 9A0000----      E      24     AGAIN:  CALL    CI                ;GET CHARACTER FROM KEYBOARD
000E 3C0D                   25             CMP     AL,CR             ;IS IS CARRIAGE RETURN?
0010 740E                   26             JE      CRLF              ;IF YES THEN OUTPUT CR/LF
0012 50                     27             PUSH    AX                ;PASS CHAR. ON STACK
0013 9A0000----      E      28             CALL    ENCRYPT           ;TRANSFORM IT
0018 50                     29             PUSH    AX
0019 9A0000----      E      30             CALL    CO                ;OUTPUT CHAR ON SCREEN
001E EBE9                   31             JMP     AGAIN
                            32
                            33     ;WE SHOULD ONLY BE EXECUTING CRLF IF A CARRIAGE RETURN WAS INPUT
                            34     ; CRLF OUTPUTS A CARRIAGE RETURN AND LINE FEED
0020 B00D                   35     CRLF:   MOV     AL,CR
0022 50                     36             PUSH    AX
0023 9A0000----      E      37             CALL    CO                ;OUTPUT A CARRIAGE RETURN
0028 B00A                   38             MOV     AL,LF
002A 50                     39             PUSH    AX
002B 9A0000----      E      40             CALL    CO                ;OUTPUT A LINE FEED
0030 EBD7                   41             JMP     AGAIN             ;GO BACK TO GET NEXT CHAR.
----                        42     CODE    ENDS
                            43             END     START
```

```
SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB4_PART_1_SUB
OBJECT MODULE PLACED IN :F2:LAB4A2.OBJ
ASSEMBLER INVOKED BY:  :F3:ASM86.86 :F2:LAB4A2.ASM SYMBOLS DEBUG


LOC  OBJ            SOURCE

                            NAME    LAB4_PART_1_SUB

  0000              SWITCHES        EQU     0

                            PUBLIC  ENCRYPT

----                PRO     SEGMENT
                            ASSUME CS:PRO

0000 1F00           TABLE   DW      PLUS_1,MINUS_1,PLUS_2   ;JUMP TABLE
0002 2300
0004 2700

0006                ENCRYPT PROC    FAR
                    ; THIS PROCEDURE WILL ENCRYPT THE CHARACTERS ACCORDING TO THE
                    ; VALUE READ FROM PORT 0.

0006 55                     PUSH    BP              ;SAVE BP
0007 8BEC                   MOV     BP,SP           ;USE AS REFERENCE IN STACK
0009 E400                   IN      AL,SWITCHES     ;FIND OUT WHICH ONE
000B 3C02                   CMP     AL,2            ;SEE IF OUT OF RANGE
000D 770A                   JA      ERROR           ;YES THEN EXIT
000F 32E4                   XOR     AH,AH           ;OTHERWISE CONVERT TO WORD
0011 8BF0                   MOV     SI,AX           ;PUT IT IN AN INDEX REGISTER
0013 8B4606                 MOV     AX,[BP+6]       ;GET CHARACTER
0016 2EFF24                 JMP     TABLE[SI]       ;AND ENCRYPT IT

0019 B02A           ERROR:  MOV     AL,'*'          ;ILLEGAL CHARACTER
001B 5D             EXIT:   POP     BP              ; IN AL
001C CA0200                 RET     2               ;DELETES PARAMETER FROM STACK

001F FEC0           PLUS_1:         INC     AL              ;INCREMENT CHARACTER
0021 EBF8                           JMP     EXIT

0023 FEC8           MINUS_1:        DEC     AL              ;DECREMENT CHARACTER
0025 EBF4                           JMP     EXIT

0027 0402           PLUS_2:         ADD     AL,2            ;ADD 2 TO CHARACTER
0029 EBF0                           JMP     EXIT

                    ENCRYPT ENDP
----                PRO     ENDS
                            END
```

```
SERIES-III 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE LAB4_PART_2_SUB
OBJECT MODULE PLACED IN :F2:LAB4B.OBJ
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
0000 (65                     6     TABLE   DB      41H DUP ('*')          ;ONLY LETTERS ENCRYPTED
     2A
     )
0041 5A595857565554          7             DB      'ZYXWVUTSRQPONMLKJIHGFEDCBA'
     535251504F4E4D
     4C4B4A49484746
     4544434241
005B (6                      8             DB      6 DUP ('*')
     2A
     )
0061 5A595857565554          9             DB      'ZYXWVUTSRQPONMLKJIHGFEDCBA'
     535251504F4E4D
     4C4B4A49484746
     4544434241
007B (5                     10             DB      5 DUP ('*')
     2A
     )
----                        11     TRANS   ENDS
                            12
----                        13     PRO     SEGMENT
                            14             ASSUME CS:PRO,DS:TRANS
                            15
0000                        16     ENCRYPT PROC    FAR
                            17     ; THIS PROCEDURE WILL ENCRYPT THE CHARACTERS ACCORDING TO THE
                            18     ; VALUE READ FROM PORT 0.
                            19
0000 55                     20             PUSH    BP              ;SAVE BP
0001 8BEC                   21             MOV     BP,SP           ;USE AS REFERENCE IN STACK
0003 1E                     22             PUSH    DS              ;SAVE DS AND BX SINCE WE ARE USING THEM
0004 53                     23             PUSH    BX
0005 B8----          R      24             MOV     AX,TRANS
0008 8ED8                   25             MOV     DS,AX
000A 8D1E0000               26             LEA     BX,TABLE
000E 8B4606                 27             MOV     AX,[BP+6]       ;GET CHARACTER
0011 D7                     28             XLATB                   ;CONVERT THE CHARACTER AND LEAVE IT IN AL
0012 5B                     29             POP     BX              ;GET BACK THE REGISTERS
0013 1F                     30             POP     DS
0014 5D                     31             POP     BP
0015 CA0200                 32             RET     2               ;DELETES PARAMETER FROM STACK
                            33
                            34     ENCRYPT ENDP
----                        35     PRO     ENDS
                            36             END
```

CO and CI

```
/*                                                                    */


/* 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;
                   CONN=DQ$CREATE ( @(4,':CO:'), @ERR);
                   CALL DQ$OPEN (CONN, 2, 0,@ERR);
                END;
        CALL DQ$WRITE (CONN, @CHAR,1,@ERR);
END CO;
END COMOD;
```

CO and CI

```
/*                                                                       */


/*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;
```

APPENDIX C

CLASS EXERCISE SOLUTIONS

CLASS EXERCISE SOLUTIONS

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.

```
        CS          XXXX
        IP        + YYYY
                   05820
```

3. DS, and BX, BP, SI, or DI

4. 00230H

5. 0002H

REVIEW (FILL IN REGISTER NAMES)

```
 ----------
 |   SS   |-------------------------------------->---------------   0
 ----------                |                     |              |
                           +                     |              |
            -------------------------------      |              |
            |             SP              |      |    STACK     |
            -------------------------------      |              |
                           |                     |              |
                           --------------------->|              |
                                                 ----------------
 ----------                                      |              |
 |   DS   |-------------------------------------->---------------
 ----------                |                     |              |
                           +                     |              |
            -------------------------------      |              |
            |   BX, BP, SI, DI            |      |    DATA      |
            -------------------------------      |              |
                           |                     |              |
                           --------------------->|              |
                                                 ----------------
 ----------                                      |              |
 |   CS   |-------------------------------------->---------------
 ----------                |                     |              |
                           +                     |              |
            -------------------------------      |              |
            |             IP              |      |    CODE      |
            -------------------------------      |              |
                           |                     |              |
                           --------------------->|              |
                                                 ----------------  FFFFF
```

CLASS EXERCISE SOLUTIONS

4.1

```
        MOV     DX,0FFFF8H
        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
```

CLASS EXERCISE SOLUTIONS

7.2

```
        NAME            CLASS_EXERCISE_7_2

        STATUS_PORT     EQU         10
        DATA_PORT       EQU         11
        RDY             EQU         00000001B

        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
        POLL            ENDS
                        END         HANDSHAKE
```

8.1

```
        1.    WAREA     DW          2000H
        2.    BAREA     DB          ?
        3.    MOV       BAREA,10
        4.    AND       WAREA,40H
        5.    TEST      WAREA,8000H
```

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
```

CLASS EXERCISE SOLUTIONS

9.1 RUN ASM86 :F1:PROB.LEM SB DB PR(:F1:LISTIN.G)
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
```

CLASS EXERCISE SOLUTIONS

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 = ?
      −15 = ?
      Yes one wait state

CLASS EXERCISE SOLUTIONS

15.1

```
        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

```
------------------------------------------------------------------------------
            NAME MODA               | NAME    MODB
PUBLIC      USEFUL_DATA,HANDY       | EXTRN   USEFUL_DATA:BYTE
                                    | EXTRN   HANDY:FAR
DATA        SEGMENT                 | B_CODE  SEGMENT
USEFUL_DATA     DB    ?             | ASSUME  CS:B_CODE
DATA        ENDS                    | &       DS:SEG USEFUL_DATA
                                    |
A_CODE      SEGMENT                 |         MOV AX,SEG USEFUL_DATA
            ASSUME CS:A_CODE        |         MOV DS,AX
                                    |         MOV AL,USEFUL_DATA
HANDY       PROC FAR                |         ---
            MOV  AX,0               |         CALL HANDY
            RET                     | B_CODE  ENDS
HANDY       ENDP                    |         END
A_CODE      ENDS                    |
            END                     |
------------------------------------------------------------------------------
```

CLASS EXERCISE SOLUTIONS

19.1

5	BM3 DRIVES BUSY HIGH
2	BM2 ISSUES CBRQ HIGH
1	BM2 DRIVES BPRO HIGH
6	BM2 TAKES OVER BUSY, DRIVES BUSY LOW
3	BM3 SEES CBRQ LOW
4	BM3 SEES BPRN HIGH

APPENDIX D

DAILY QUIZZES

Quiz #1

1. Match the pointer with the appropriate memory area:

CPU
IP
SP
DI

MEMORY
STACK RAM
INSTRUCTIONS ROM/PROM/EPROM/RAM
VARIABLES RAM

2. What is the state (1,Ø) of the zero flag after the CPU executes the following arithmetic operations?

5FH	5FH	5FH
-5FH	-4FH	-6FH

3. Which SEG REG and offset REG would the CPU use to generate an address for the following types of memory access?

	SEG	OFFSET
Op code fetch	________	________
Stack access	________	________
Data access	________	________

4. Where does the CPU get immediate data?

5. What is wrong with the following 8Ø86 instructions and what can be done to make them work?

```
IN   AL,ØFFFFH
```

```
SAR  AX,5
```

Quiz #2

1. Match the following:

TEST ____		a. 2's complement
		b. Used for multi-word addition
CMP ____		c. "Non-destructive" AND
		d. Used when dividing one signed word by another
NOT ____		e. 1's complement
NEG ____		f. "Non-destructive" subtract
		g. Used when dividing one signed byte by another
ADC ____		
CBW ____		
CWD ____		

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 8Ø86 CPU after being reset.

    FLAGS ________

    CS ________

    IP,DS,SS,ES ________

    AX,BX,CX,DX ________

4. What address will the 8Ø86 CPU begin execution after being reset

TRUE - FALSE (circle one)

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	________

Quiz #3

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.

____	IF and TF are cleared
__1__	CPU completes execution of current instruction
____	CS and IP loaded from Interrupt Vector Table
____	Flags pushed onto stack
____	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 $AL_{LOW} > 9$ OR IF A.C. = 1 THEN ADD 6

  RULE 2 : IF $AL_{HI} > 9$ OR IF C = 1 THEN ADD 60

EXAMPLES:

DECIMAL		BCD	
29		0010 1001	
+ 1		1	
30		0010 1010	
		0110	(RULE 1)
		0011 0000	
18		0001 1000	
+18		0001 1000	
36		0011 0000	
		0110	(RULE 1)
		0011 0110	
72		0111 0010	
+93		1001 0011	
165	1	0000 0101	
		0110 0000	(RULE 2)
	1	0110 0101	

(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

Decimal	Binary
Z 5	XXXX 0101
+ Z 6	+ XXXX 0110
X B	XXXX 1011
+ 6	0110

AH	AL
+ 1	0 1

AH	AL	
+1	0000 0001	AAA

ASCII ARITHMETIC - ADDITION

OPERATION: C = A + B ; WHERE A AND B ARE STRINGS OF ASCII DIGITS, AND C IS TO BE A STRING OF UNPACKED BCD DIGITS.

```
        MOV     BX, STRING_LENGTH - 1
        CLC
NEXT:   MOV     AL, A[BX]
        ADC     AL, B[BX]
        AAA
        MOV     C[BX], AL
        DEC     BX
        JNS     NEXT
```

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 = 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:  MOV     DL, B           ;GET B
        MOV     SI, OFFSET A    ;POINTER TO A
        MOV     DI, OFFSET C    ;POINTER TO C
        MOV     CX, LENGTH A    ;# OF TIMES TO LOOP
        CLD                     ;AUTO INCREMENT

        AND     DL, 0FH         ;RID B OF ZONE
        XOR     AH, AH          ;SEED LOOP

NEXT:   LODS    A               ;GET BYTE
        AND     AL, 0FH         ;ZERO ZONE
        AAD                     ;ADJUST FOR DIVIDE
        DIV     DL
        STOS    C               ;SAVE QUOTENT BYTE
        LOOP    NEXT
```

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:

DECIMAL	BCD	
9	1001	BCD = BINARY
x 9	* x 1001	BCD = BINARY
81	1010001	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:

81	÷	10	=	8	REMAINDER 1
1010001	÷	1010	=	1000	REMAINDER 0001

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 * 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:  MOV   DL, B                 ;GET SINGLE ASCII DIGIT
        MOV   CX, LENGTH A          ;NUMBER OF TIMES TO LOOP
        STD                         ;SET UP FOR AUTO DECREMENT
        MOV   SI, OFFSET A + LENGTH A -1
        MOV   DI, OFFSET C + LENGTH A -1

        MOV   BYTE PTR [DI], 0      ;CLEAR C(1)
        AND   DL, 0FH               ;CLEAR ZONE OF B

NEXT:   LODS  A                     ;LOAD BYTE FROM A
        AND   AL, 0FH               ;CLEAR ZONE
        MUL   DL                    ;MULTIPLY BY B
        AAM                         ;ADJUSTED RESULT IN AX
        ADD   AL, [DI]              ;ACCUMULATE INTO C
        AAA                         ;IN UNPACKED FORMAT
        STOS  WORD PTR C            ;PROPOGATE UPPER DIGIT
        INC   DI                    ;POINT TO PROPER DIGIT
        LOOP  NEXT
```

NOTE: AAM PLACES THE UPPER DIGIT IN AH. AAA PROPIGATES THE CARRY FROM THE LOWER NIBBLE BY ADDING THE CARRY TO AH. THE C STRING IS ONE BYTE LONGER THAN THE A STRING.

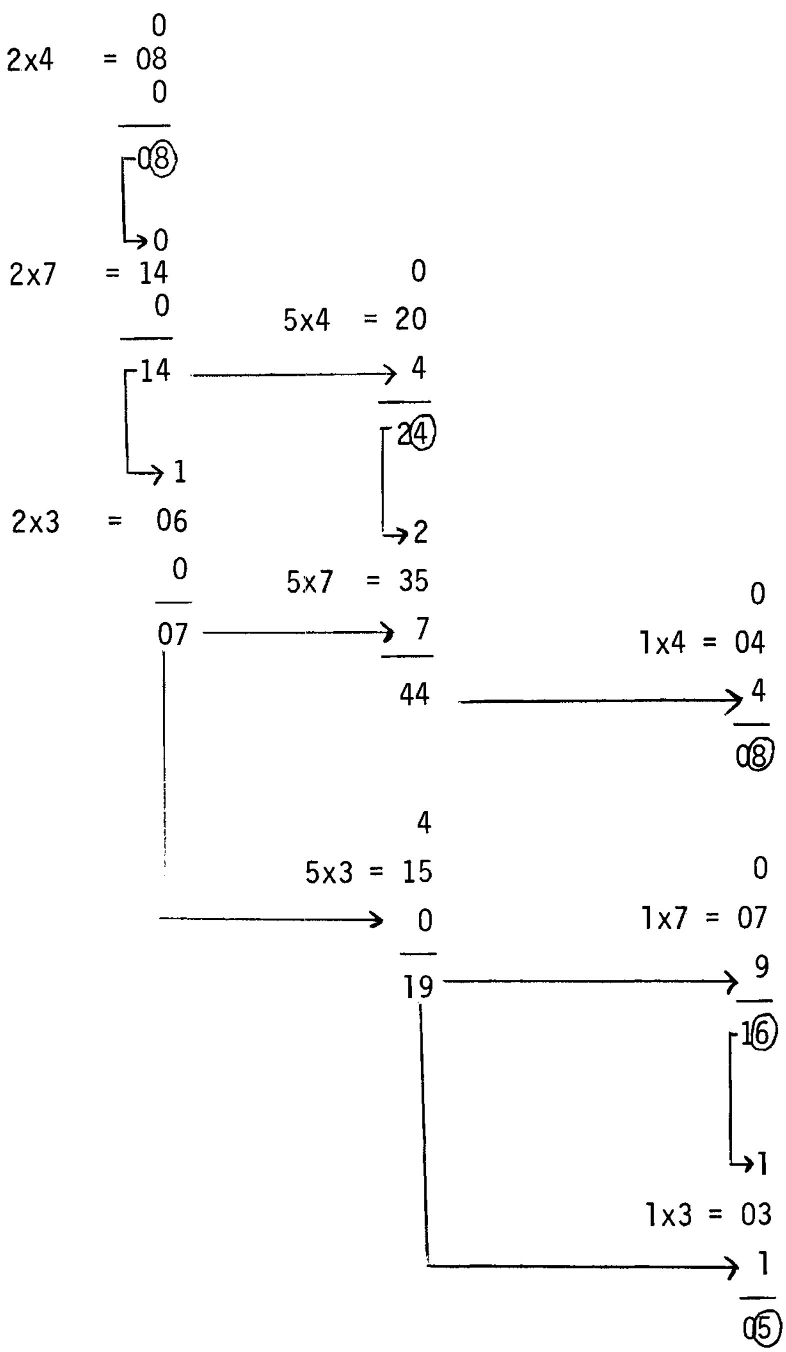
MULTIPLICATION LOOP

UNPACKED BCD

MULTIPLICAND	INDEX	SI
PARTIAL PRODUCT	INDEX	DI
MULTIPLIER	INDEX	BX
MULTIPLIER LENGTH		B
MULTIPLICAND LENGTH		C

```
        ZERO PARTIAL PRODUCT
        MULTIPLIER INDEX    BX = 1
LOOP1:  DL = 0                                                    <--+
        INITIALIZE MULTIPLICAND INDEX SI = 1                          |
        INITIALIZE PARTIAL PRODUCT INDEX: DI = BX (MULTIPLIER INDEX)  |
->LOOP2: FETCH MULTIPLICAND [SI]  TO AL                               |
|                                                                     |
|       MULTIPLY  MULTIPLIER [BX] * AL ---> AL                        |
|       ASCII  MULTIPLY ADJUST      AX                                |
|       ADD DL TO AL                                                  |
|       ASCII  ADD ADJUST  AL                                         |
|       ADD PARTIAL PRODUCT [DI]  TO AL                               |
|       ASCII  ADD ADJUST   AL                                        |
|       STORE AL TO PARTIAL PRODUCT [DI]                              |
|       SAVE  DL = AH                                                 |
|                                                                     |
|       DI = DI + 1                                                   |
|       SI = SI + 1                                                   |
|                                                                     |
+------ IF SI ≤ C  (MULTIPLICAND LENGTH) TO TO LOOP 2                 |
        STORE DL TO PARTIAL PRODUCT  [DI]                             |
        BX = BX + 1                                                   |
        IF BX ≤ B  (MULTIPLIER COUNT) GO TO LOOP 1 -------------------+
```

```
   374
 x 152
 -----
   748
 1870
 374
 -----
 56848
```



APPENDIX F

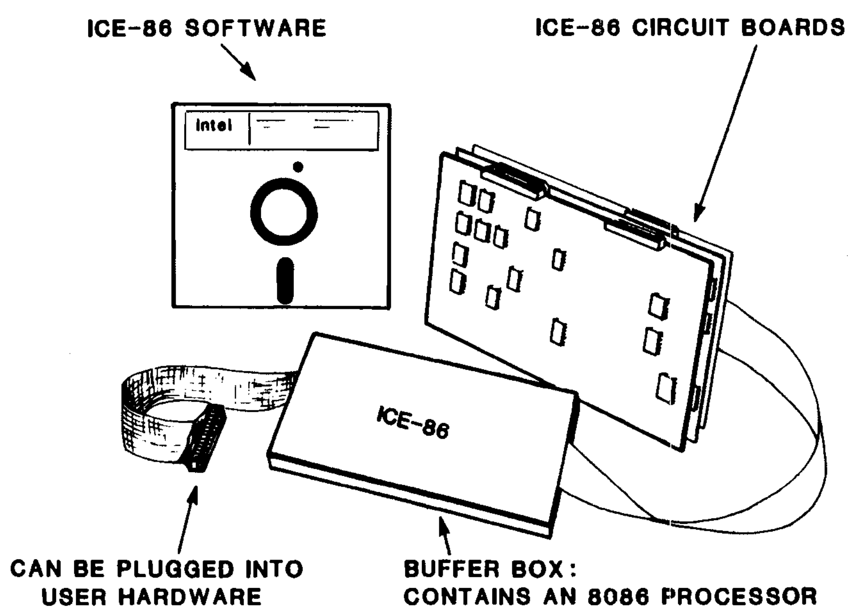
ICE-86,88 IN-CIRCUIT EMULATOR

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 COMPONENTS

FM CONTROLLER PCB - 8080 ICEµP, 12KB FIRMWARE ROM, 3KB SCRATCHPAD RAM
86 CONTROLLER PCB - 2KB ICE RAM, 1K x6 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 AUXILLIARY CONNECTOR
"T" CABLE
GROUND CABLE

ICE-86 DISKETTE -

ICE86	ICE86,OV5
ICE86,OV0	ICE86,OV6
ICE86,OV1	ICE86,OV7
ICE86,OV2	ICE86,OV8
ICE86,OV3	ICE86,OVE
ICE86,OV4	

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 OR64) (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.

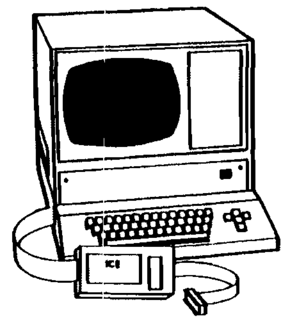
NOTE:

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:

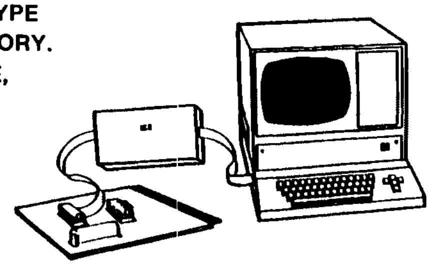
NO PROTOTYPE HARDWARE AVAILABLE- USE ICE-86 STANDALONE, DEBUG SOME OR ALL PROGRAM MODULES. PROGRAMS RESIDE IN ICE AND/OR MDS AND/OR DISK MEMORY.



PRODUCT DEVELOPMENT PHASES USING ICE-86

PHASE 2:

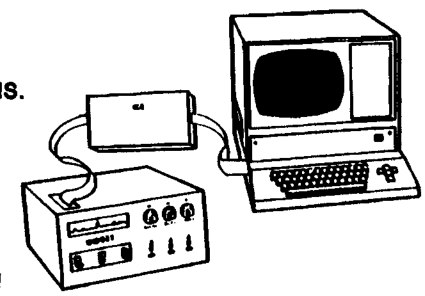
SKELETON PROTOTYPE HARDWARE AVAILABLE- DEBUG HARDWARE BY EXECUTING TEST SOFTWARE. DEBUG SYSTEM WITH PROTOTYPE HARDWARE AND SOFTWARE. PROGRAMS RESIDE IN PROTOTYPE AND/OR ICE AND/OR MDS AND/OR DISK MEMORY. DOWN LOADING OF PROGRAMS DONE BY ICE, NO NEED TO BURN PROMS.



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

INIT IO:   MOV        DX,USART_CMD_PORT
             .
             .
             .
           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.

- WHEN USED IN CONJUNCTION WITH THE OPTIONAL INITCODE CONTROL ON THE LOC86 INVOCATION LINE. THE LOCATOR USES THIS INFORMATION TO CREATE A SEGMENT CALLED ?? LOC86_INITCODE WHICH INITIALIZES ALL SPECIFIED REGISTERS.

INVOKING ICE-86

THE ICE-86 SOFTWARE DRIVER IS INVOKED FROM ISIS-II.

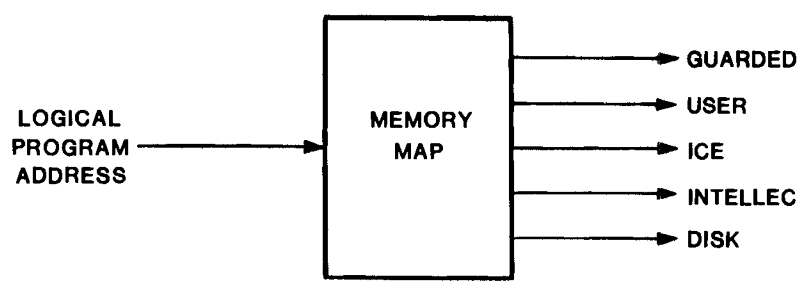
-ICE86

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

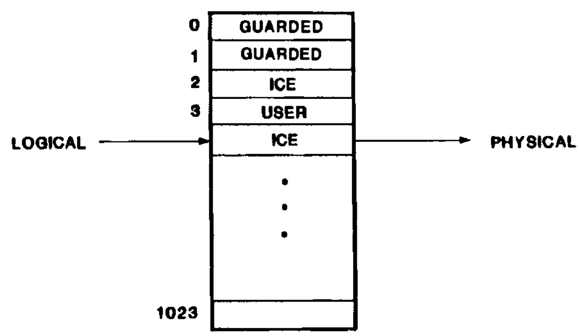
PREPARATION OF THE ENVIRONMENT
MEMORY MAPPING



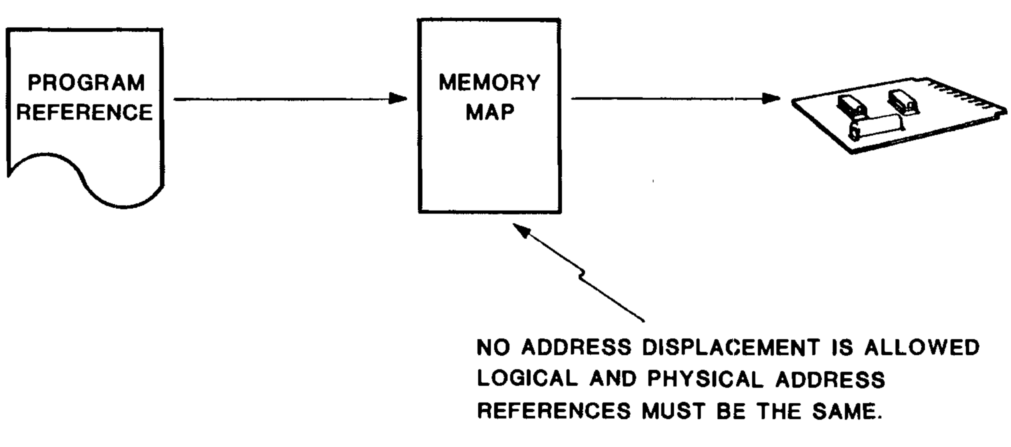

```
                 ( GUARDED                                                  )
                 ( USER [NOVERIFY]                                          )
MAP partition = < ICE [physical-segment-number] [NOVERIFY]                  >
                 ( INTELLEC [physical-segment-number] [NOVERIFY]            )
                 ( DISK [physical-segment-number] [NOVERIFY]                )
where

partition =  ( logical-segment-number   [TO     logical-segment-number]   )
             ( logical-segment-number LENGTH logical-segment-length      )
```

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

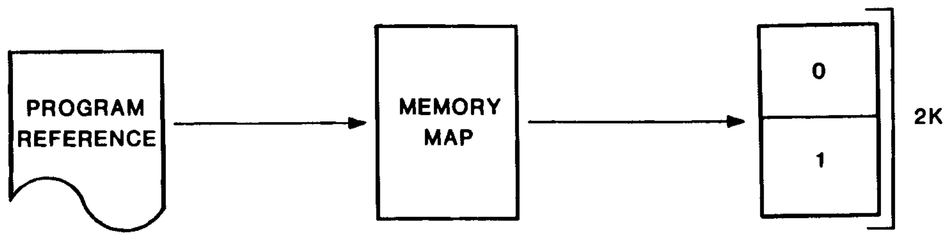


MAPPING TO USER MEMORY



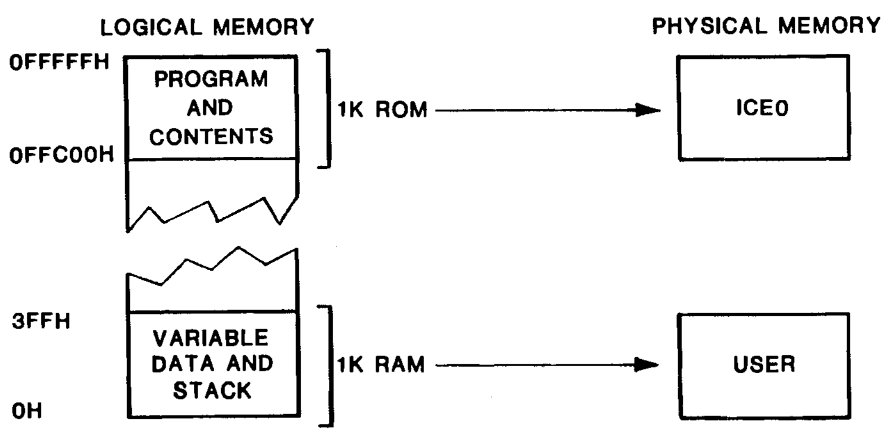
* MAP 0 LEN 32=USER
* MAP 1000=USER

MAPPING TO ICE-86 MEMORY



* MAP 0=ICE 0
* MAP 1023=ICE 1

MEMORY MAPPING EXAMPLE



* MAP 0=USER
* MAP 1023=ICE 0

* DISPLAY MAP STATUS COMMAND

Example 1.

MAP 0 TO 3

Display:

```
0000T = USE      0001T = ICE 0000T 0002T = INT 0004T 0003 = DIS 0001T
```

Example 2.

MAP

Display:

```
0000T = USE           0001T = ICE   0000T   0002T = INT   0004T   0003 = DIS   0000T
0004T = DIS   0001T   0004T = DIS   0002T   0000T = USE           0007 = USE
  .      .      .      .      .      .      .      .               .      .      .
  .      .      .      .      .      .      .      .               .      .      .
  .      .      .      .      .      .      .      .               .      .      .
  .      .      .      .      .      .      .      .               .      .      .
1023T = DIS
```

* RESET MAP COMMAND

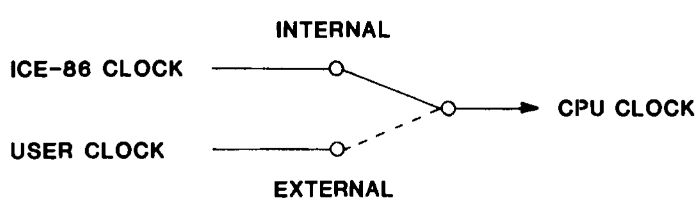
RESET MAP

PREPARATION OF THE ENVIRONMENT
CLOCK SELECTION

* CLOCK= INTERNAL ;DEFAULT

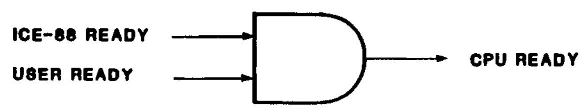
OR

* CLOCK= EXTERNAL

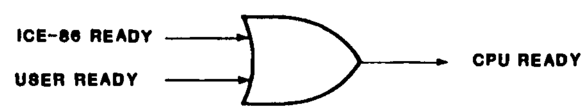


PREPARATION OF THE ENVIRONMENT
ENABLE/DISABLE READY COMMAND

ENABLE RDY **Default**



DISABLE RDY



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

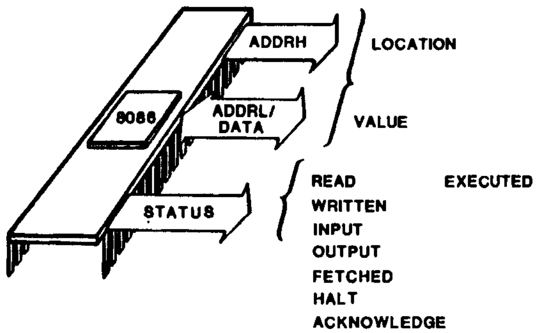
STEP EMULATION -

* 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.

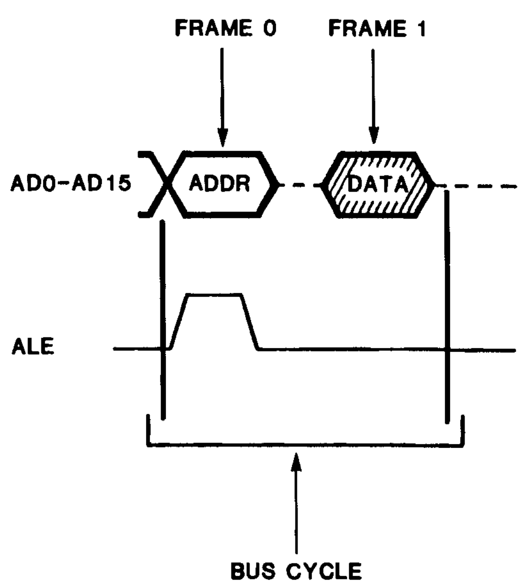
```
*STEP FROM .START
```

ICE-86 OPERATION

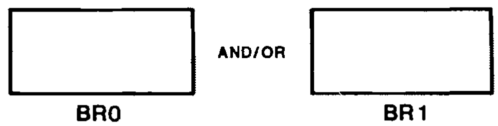


ICE-86 MONITORS THE BUSSES, (ADDRESS AND DATA CONTROL); EACH FRAME OF A BUS CYCLE IS MONITORED AND CAN BE SAVED.

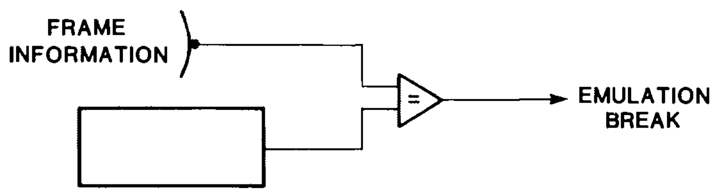
8086 BUS CYCLE TRACING



ICE-86 BREAKPOINTS



ICE-86 HAS TWO BREAKPOINT REGISTERS THAT MAY BE GIVEN VALUES THROUGH SOFTWARE COMMANDS.



BREAKPOINT REGISTER CONTENTS

ICE-86 BREAKPOINTS

ICE-86 BREAKPOINTS ARE OF TWO TYPES:

EXECUTION

- TAKES INTO ACCOUNT THE QUEUE
- TRACKS INSTRUCTION THROUGH QUEUE

SYNTAX:

EXECUTED

NON-EXECUTION

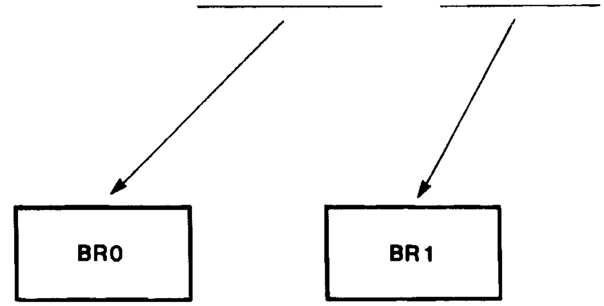
- BASED ON BUS ACTIVITY ONLY

SYNTAX:

READ
WRITTEN
INPUT
OUTPUT
FETCHED
HALT
ACKNOWLEDGED

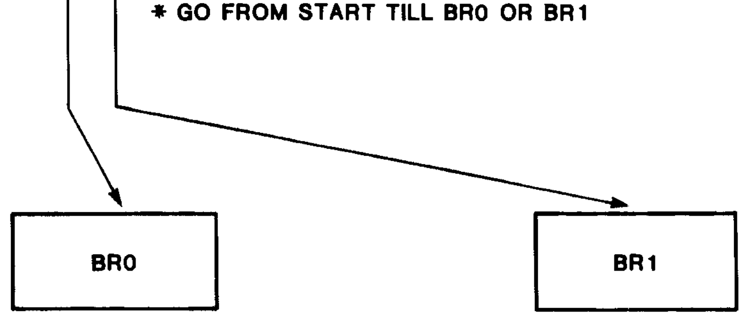
LOADING THE BREAKPOINT REGISTERS

GO FROM .START TILL .PORT 2 OUTPUT OR .PARM1 READ



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
```

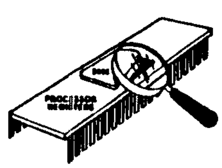
OR

```
* BRO=.PROC1 EXEC
* GR=TILL BRO
* GO FROM .START
```

OR

```
* GR=TILL .PROC1 EXEC
* GO FROM .START
```

INTERROGATION MODE
DISPLAY/CHANGE



REGISTERS	FLAGS	PINS (READ ONLY)
REG	RF	HOLD
RBX	AFL	NMI
RAL	TFL	IR
SP	IFL	RDY

```
*REG
RAX=0000H RBX=0000H RCX=0000H RDX=0000H SP=0000H BP=0000H SI=0000H DI=0000H
CS=0000H DS=0000H SS=0000H ES=0000H RF=0000H IP=0000H
*
*RAX=5555
*
*RCH=FF
*
*REG
RAX=5555H RBX=0000H RCX=FF00H RDX=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=0200H
*
*HOLD
HOL=0
```

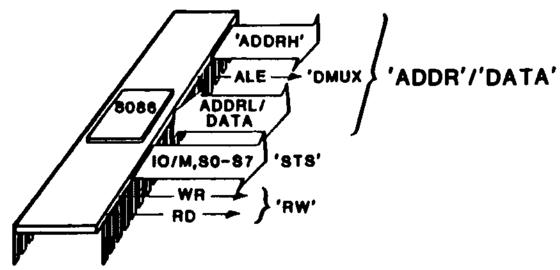
INTERROGATION MODE (CONT.)
ACCESSING MEMORY AND I/O

```
*BYTE .BUFFER LEN 16T = 77
*
*BYTE .BUFFER LEN 16T
BYT 0020:0000H=77H 77H 77H 77H 77H 77H 77H 77H 77H 77H 77H 77H 77H 77H 77H 77H
*
*INTEGER .SUM = -9
*
*!SUM
INT 0022:0000H=-0009H
*
*WORD .XYZ
WOR 0023:0004H=0261H
*
*!XYZ = 0
*
*!XYZ
WOR 0023:0004H=0000H
*
*WPORT .CONTROL = 9090
*
*PORT FFF9
POR FFF9H=AAH
*
*PORT FFFB = FF
*WPORT .LIGHTS = 0
*
*WPORT .SWITCHES
WPO FFF8H=AADFH
```

INTERROGATION MODE (CON'T.)
CODE DISASSEMBLY

```
*ASM .START LEN 20
ADDR            PREFIX     MNEMONIC   OPERANDS                          COMMENTS
0020:0010H                 MOV        DX,FFEAH
0020:0013H                 MOV        AL,00H
0020:0015H                 OUT        DX,AL
0020:0016H                 MOV        AL,39H
0020:0018H                 OUT        DX,AL
0020:0019H                 CALL       $+008EH                           ; SHORT
0020:001CH                 CALL       $+007CH                           ; SHORT
0020:001FH                 MOV        WORD PTR [0024H],0000H
0020:0025H                 PUSH       WORD PTR [0024H]
0020:0029H                 MOV        AL,00H
0020:002BH                 PUSH       AX
0020:002CH                 MOV        AL,01H
0020:002EH                 PUSH       AX
0020:002FH                 CALL       $+0087H                           ; SHORT
```

TRACE DATA COLLECTION



BUS DATA

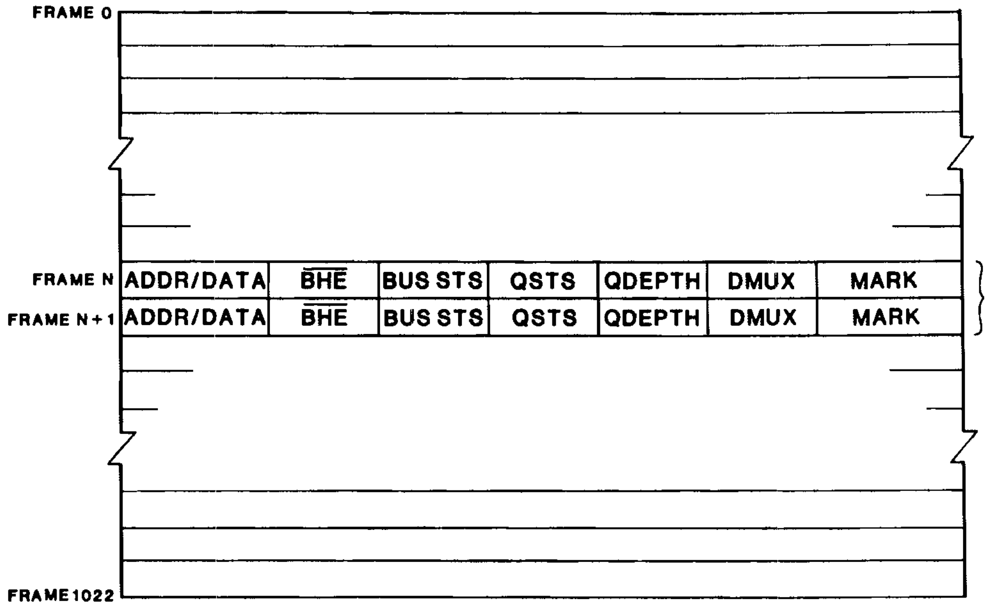
TRACE DATA

ADDR/DATA	BHE	BUS STS	QSTS	QDEPTH	DMUX	MARK
20	1	3	2	3	2	1

• EACH FRAME OF TRACE DATA CONTAINS 32 BITS OF INFORMATION.

TRACE DATA BUFFER

2 FRAMES/MACHINE CYCLE - 511 CYCLE CAPACITY



CONTROLLING TRACE DATA COLLECTION

* ENABLE TRACE

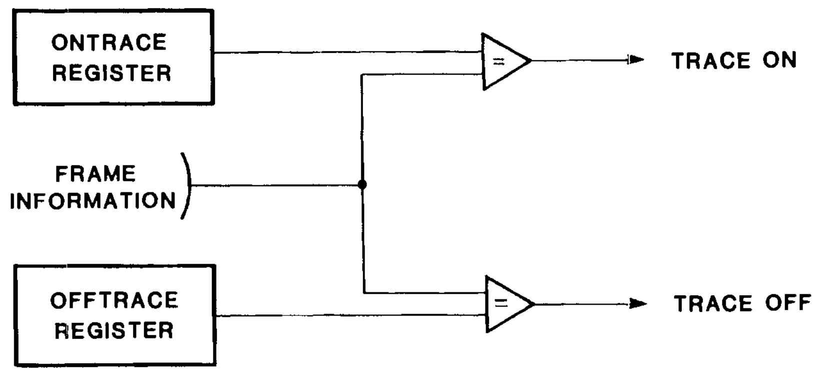
  NOTE: BY DEFAULT THE TRACE IS INITIALLY TURNED ON.

* DISABLE TRACE

TRACE DATA CAN ALSO BE COLLECTED CONDITIONALLY

CONDITIONAL TRACE DATA COLLECTION

ICE-86 HAS TWO TRACE CONTROL REGISTERS THAT MAY BE LOADED BY SOFTWARE COMMANDS.



USING THE TRACE CONTROL REGISTERS

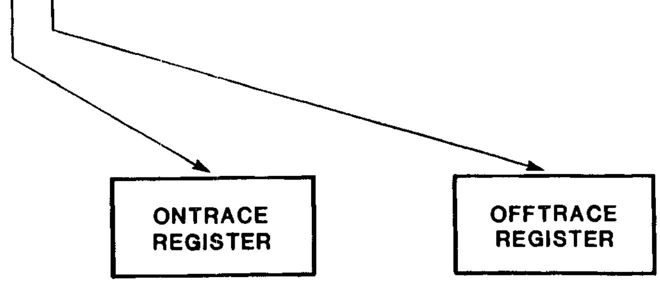
* ONTRACE =.DISPLAY_DATA FETCHED

;TRACE CONTROL REGISTERS CAN ONLY
;BE LOADED WITH NON-EXECUTION
MATCH CONDITIONS.

* OFFTRACE =.LIGHT_PORT OUTPUT
* ENABLE TRACE CONDITIONALLY NOW OFF

OR

* ENABLE TRACE CONDITIONALLY NOW ON



DISPLAYING TRACE DATA

Set TRACE Display Mode Command

```
TRACE = {FRAME      }
        {INSTRUCTION}
```

Examples:

```
TRACE = FRAME
TRACE = INSTRUCTION
```

PRINT Command

1. PRINT ALL
2. PRINT [[+ :: -]*decimal*]

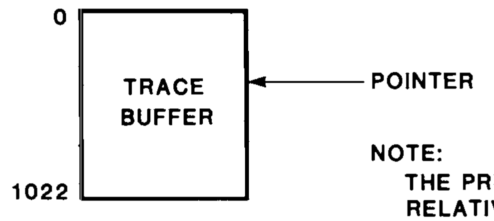
Example:

```
PRINT
PRINT ALL
PRINT +5
PRINT 5
PRINT -10
```

EXAMPLES

```
*TRACE
TRA=INS
*
*
*PRINT -5
FRAME ADDR   PREFIX     MNEMONIC   OPERANDS                          COMMENTS
0997: 00217H              MOV      DX,FFF8H
1003: 0021AH              IN       AX,DX
      FFF8H-I-0220H
1007: 0021BH              NOT      AX
1010: 0021DH              MOV      DX,FFFAH
1015: 00220H              OUT      DX,AX
      FFFAH-O-FDDFH
*
*
*TRACE = FRAME
*
*
*PRINT -5
FRAME  ADDR   BHE/ STS QSTS QDEPTH DMUX MARK
1016: 2FFF3H   0    F   N     3     D    0
1017: 0FFFAH   0    O   N     3     A    0
1018: 2FDDFH   0    O   N     3     D    0
1019: 00224H   0    F   N     3     A    0
1020: 2F4FBH   0    F   N     5     D    0
```

TRACE BUFFER POINTER



NOTE:
THE PRINT COMMAND FUNCTIONS RELATIVE TO THE POINTER.

MOVE, OLDEST, and NEWEST Commands

MOVE [[+::-]*decimal*]
OLDEST
NEWEST

Example:

MOVE
MOVE +6
MOVE -11
OLDEST
NEWEST

MISCELLANEOUS FEATURES AND COMMANDS

Set or Display Console Input Radix Commands

SUFFIX

SUFFIX = Y::Q::O::T::H

Examples:

SUFFIX

SUFFIX = Y

Set or Display Console Output Radix Commands

BASE

BASE = Y::Q::O::T::H::ASCII

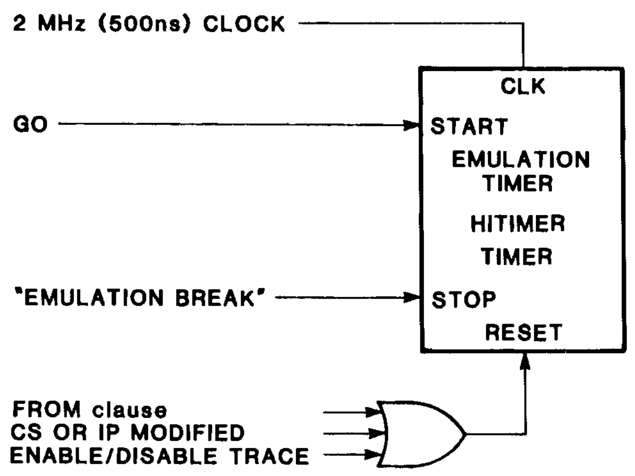
Examples:

BASE

BASE = Q

- INITIAL RADIX IS HEX FOR BOTH INPUT AND OUTPUT.

EMULATION TIMER



LOAD Command

LOAD[:*drive*:]*filename* { NOCODE / NOSYMBOL / NOLINE }

Examples:

```
LOAD:F0:TEST.VR1
LOAD:F1:MYPROG NOLINE
LOAD:F2:COUNT. ONE NOCODE NOLINE
LOAD:F3:NEWCOD NOSYMBOL
```

SAVE Command

SAVE [:*drive*:]*filename* NOCODE:*partition* [,*partition*]*
NOSYMBOL
NOLINE

Examples:

```
SAVE:F1:TEST
SAVE:F0:MYPROG 0000 TO 0FFF NOLINE
SAVE:F2:COUNT.TWO NOLINE NOSYMBOL
SAVE:F3:NEWSYM NOCODE NOLINE
SAVE:F1:TEST #1 TO #50,..SUBR #1 TO ..SUBR #20
```

LIST Command

(a) LIST :*device:*

(b) LIST [:*drive*:] *filename*

Examples:

```
LIST :LP:
LIST :CO:
LIST :F1:ICEFIL
```

• TO RETURN TO ISIS-II

* EXIT

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

   *________________________________

CLASS EXERCISE 6.1 (CON'T.)

6. EXAMINE THE REGISTERS

   * ______________________________

7. EXAMINETHE 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

   * ______________________________

   * ______________________________

CLASS EXERCISE 6.1 (CON'T.)

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 :FØ:DEMO.A86. DEMO.A86 is a source file for a program which is written in 8Ø86 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.

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 <u>requires</u> 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 Ø LEN 2 = USER

This command identifies the first two 1K blocks in the 8Ø86's logical address space as being located in the user system (ØØØØØH - ØØ7FFH).

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:0

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.

<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 ØFFF9H 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 -2Ø

you can see what the last 2Ø 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 BRØ. How can you determine what BRØ contains? You guessed it...type:

```
*BRØ
```

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.

```
 1 * 65536 = 65536
 2 * 65536 = 131072
 3 * 65536 = 196608
 4 * 65536 = 262144
 5 * 65536 = 327680
 6 * 65536 = 393216
 7 * 65536 = 458752
 8 * 65536 = 524288
 9 * 65536 = 589824
10 * 65536 = 655360
```

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 2Ø: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

intel® customer training

Intel Corporation · 3065 Bowers Avenue · Santa Clara, California 95051 · (408) 987-8080