

CPE 408330

Assembly Language and Microprocessors

Chapter 3: Assembly Language Programming

[Computer Engineering Department,
Hashemite University]

Lecture Outline

- ▶ 3.1 Software: The Microcomputer Program
- ▶ 3.2 Assembly Language Programming Development on the PC
- ▶ 3.3 The Instruction Set
- ▶ 3.4 The MOV Instruction
- ▶ 3.5 Addressing Modes

3.1 Software : The Microcomputer Program

- ▶ A **program** is a sequence of commands that tell the microprocessor what to do.
- ▶ Each command is called “**instruction**”.
- ▶ An instruction can be divided into **two** parts:
 - Operation code (**opcode**) – one- to five-letter *mnemonic*
- ▶ **Operands**: Identify whether the elements of data to be processed are in registers or memory.

Opcode Destination Operand Source Operand

ADD AX, BX



- ▶ Source operand– location of one operand to be processed
- ▶ Destination operand—location of the other operand to be processed and the location of the result
- ▶ Format of an assembly statement:

LABEL: INSTRUCTION ; COMMENT

3.1 Software : The Microcomputer Program

- ▶ Label—address identifier for the statement
- ▶ Instruction—the operation to be performed
- ▶ Comment—documents the purpose of the statement
- ▶ Example:

START: MOV AX, BX ; COPY BX into AX

- ▶ Other examples:

INC SI ;Update pointer

ADD AX, BX

- Few instructions have a label—usually marks a point to jump
- Not all instructions need a comment
- ▶ What is the “MOV part of the instruction called?
- ▶ What is the BX part of the instruction called?
- ▶ What is the AX part of the instruction called?

3.1 Software : The Microcomputer Program

- ▶ Assembly language program
 - **Assembly language** program (.asm) file—known as “source code”
 - Converted to **machine code** by a process called “assembling”
 - Assembling performed by a software program — an “8088/8086 assembler”
 - “Machine (object) code” that can be run on a PC is output in the executable (.exe) file
 - “Source listing” output in (.lst) file—printed and used during execution and debugging of program
- ▶ **DEBUG**—part of “disk operating system (DOS)” of the PC
 - Permits programs to be assembled and disassembled
 - Line-by-line assembler
 - Also permits program to be run and tested
- ▶ **MASM**—Microsoft 80x86 macroassembler
 - Allows a complete program to be assembled in one step

3.1 Software : The Microcomputer Program

▶ Assembly source program

```
TITLE      BLOCK-MOVE PROGRAM
PAGE ,132
```

COMMENT *This program moves a block of specified number of bytes
from one place to another place*

;Define constants used in this program

```
      N = 16 ;Bytes to be moved
      BLK1ADDR= 100H ;Source block offset address
      BLK2ADDR= 120H ;Destination block offset addr
      DATASEGADDR= 2000H ;Data segment start address
```

```
STACK_SEG      SEGMENT      STACK 'STACK'
      DB
      STACK_SEG      64 DUP(?)
      ENDS
```

```
CODE_SEG      SEGMENT      'CODE'
      BLOCK      PROC      FAR
      ASSUME CS:CODE_SEG,SS:STACK_SEG
```

3.1 Software : The Microcomputer Program

▶ Assembly source program (continued)

;To return to DEBUG program put return address on the stack

```
PUSH    DS
MOV     AX, 0
PUSH    AX
```

;Set up the data segment address

```
MOV     AX, DATASEGADDR
MOV     DS, AX
```

;Set up the source and destination offset addresses

```
MOV     SI, BLK1ADDR
MOV     DI, BLK2ADDR
```

;Set up the count of bytes to be moved

```
MOV     CX, N
```

;Copy source block to destination block

```
NXTPT:  MOV     AH, [SI] ;Move a byte
        MOV     [DI], AH
        INC     SI ;Update pointers
        INC     DI
        DEC     CX ;Update byte counter
        JNZ     NXTPT ;Repeat for next byte
        RET     ;Return to DEBUG program
```

```
BLOCK   ENDP
```

```
CODE_SEG ENDS
```

```
END     BLOCK ;End of program
```

3.1 Software : The Microcomputer Program

- ▶ Assembly language must be converted by an ***assembler to an equivalent machine language*** program for execution by the 8088.
- ▶ A **directive** is a statement that is used to control the translation process of the assembler.

e.g. DB 64 DUP(?)

- Defines and leaves un-initialized a block of 64 bytes in memory for use as a stack
- ▶ The machine language output produced by the assembler is called ***object code***.

3.1 Software : The Microcomputer Program

► Listing of an assembled program

Microsoft (R) Macro Assembler Version 5.10
BLOCK-MOVE PROGRAM

5/17/92 18:10:04
Page 1-1

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62

TITLE BLOCK-MOVE PROGRAM

PAGE          ,132
COMMENT *This program moves a block of specified number of bytes
        from one place to another place*

;Define constants used in this program

N=           16           ;Bytes to be moved
BLK1ADDR=    100H         ;Source block offset address
BLK2ADDR=    120H         ;Destination block offset address
DATASEGADDR=1020H        ;Data segment start address

STACK_SEG    SEGMENT      STACK 'STACK'
              DB          64 DUP(?)

STACK_SEG    ENDS

CODE_SEG     SEGMENT      'CODE'
BLOCK        PROC         FAR
ASSUME       CS:CODE_SEG,SS:STACK_SEG

;To return to DEBUG program put return address on the stack
PUSH        DS
MOV         AX, 0
PUSH        AX

;Setup the data segment address
MOV         AX, DATASEGADDR
MOV         DS, AX

;Setup the source and destination offset addresses
MOV         SI, BLK1ADDR
MOV         DI, BLK2ADDR

;Setup the count of bytes to be moved
MOV         CX, N

;Copy source block to destination block
NXTPT:MOV     AH, [SI]           ;Move a byte
MOV         [DI], AH
INC         SI                 ;Update pointers
INC         DI
DEC         CX                 ;Update byte counter
JNZ         NXTPT              ;Repeat for next byte
RET                             ;Return to DEBUG program

BLOCK        ENDP
CODE_SEG     ENDS
END          BLOCK              ;End of program

```

3.1 Software : The Microcomputer Program

- ▶ Listing of the assembled program

e.g. 0013 8A 24 NXTPT: MOV AH, [SI] ; Move a byte

- ▶ Where:

- 0013 = offset address (IP) of first byte of code in the CS
- 8A24 = machine code of the instruction
- NXTPT: = Label
- MOV = instruction mnemonic
- AH = destination operand
- [SI] = source operand in memory

3.1 Software : The Microcomputer Program

- ▶ Listing of an assembled program

```

Segments and Groups:
Name      Length  Align  Combine Class
CODE_SEG  001D  PARA  NONE  'CODE'
STACK_SEG 0040  PARA  STACK  'STACK'

Symbols:
Name      Type  Value  Attr
BLK1ADDR  NUMBER  0100
BLK2ADDR  NUMBER  0120
BLOCK     F PROC  0000  CODE_SEG  Length = 001D
DATASEGADDR  NUMBER  1020
N         NUMBER  0010
NXTPT     L NEAR  0013  CODE_SEG
@CPU      TEXT  0101h
@FILENAME  TEXT  block
@VERSION  TEXT  510

59 Source Lines
59 Total Lines
15 Symbols

4722 + 347542 Bytes symbol space free

0 Warning Errors
0 Severe Errors

```

- Other information provided in the listing
 - **Size** of code segment and stack
 - What is the size of the code segment?
 - At what offset address does it begin? End?
- Names, types, and values of **constants** and variables
 - At what line of the program is the symbol “N” define?
 - What value is it assigned?
 - What is the offset address of the instruction that uses N?
- # lines and symbols used in the program
 - Why is the value of N given as 0010?
- # errors that occurred during assembly

3.1 Software : The Microcomputer Program

- ▶ Assembly language versus high-level language
- ▶ It is **easier** to write program with high-level language.
- ▶ Program written in assembly language usually takes up **less memory** space and executes much **faster**.
- ▶ Device **service routines** are usually written in assembly language.
- ▶ Assembly language is used to write those parts of the application that must perform **real-time** operations, and high-level language is used to write those parts that are **not time critical**.

Memory Models (new slide)

Model	Data	Code	Definition
Tiny*	near		CS=DS=SS
Small	near**	near	DS=SS
Medium	near**	far	DS=SS, multiple code segments
Compact	far	near	single code segment, multiple data segments
Large	far	far	multiple code and data segments
Huge	huge	huge	multiple code and data segments; single array may be >64 KB

* In the Tiny model, all four segment registers point to the same segment.

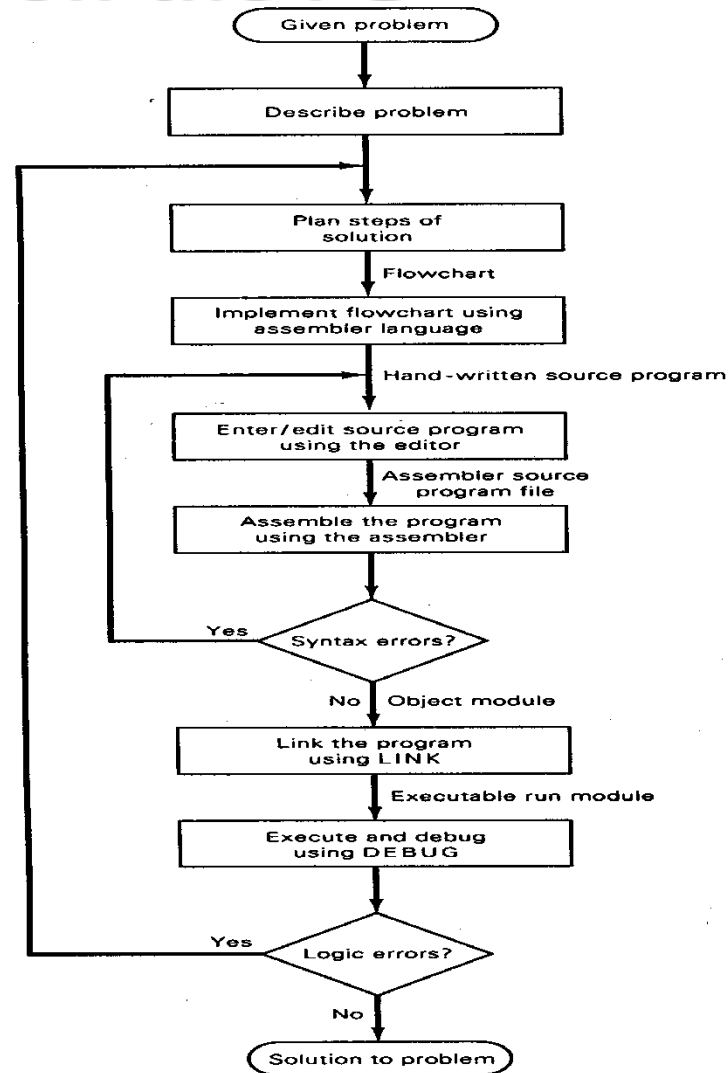
** In all models with *near* data pointers, **SS** equals **DS**.

3.2 Assembly Language Programming Development on the PC

- ▶ Describing the problem
- ▶ Planning the solution
- ▶ Coding the solution with assembly language
- ▶ Creating the source program
- ▶ Assembling the source program into an object module
- ▶ Producing a run module
- ▶ Verifying the solution
- ▶ Programs and files involved in the program development cycle

3.2 Assembly Language Programming Development on the PC

Program development cycle

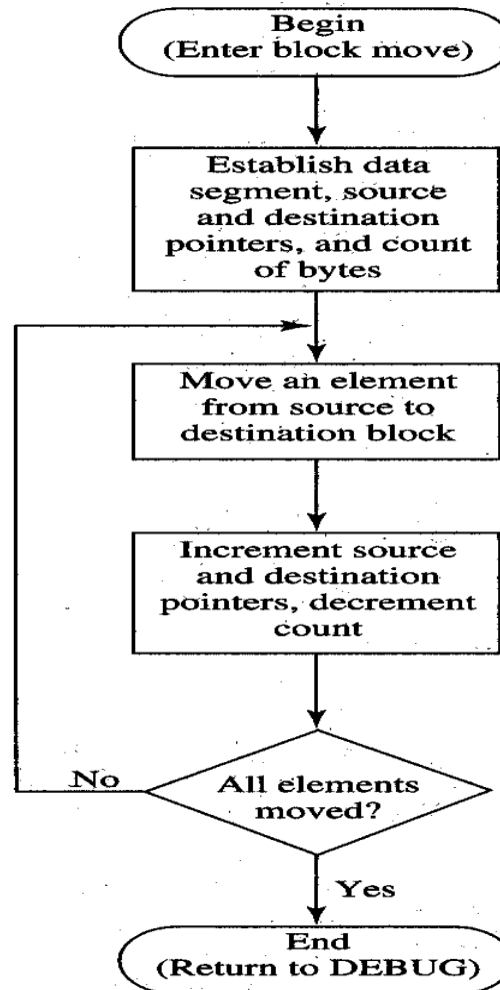


3.2 Assembly Language Programming Development on the PC

- ▶ Describing the problem
 - Most applications are described with a written document called an ***application specification***.
- ▶ Planning the solution
 - A ***flowchart*** is an outline that both documents ***the*** operations that must be performed by software to implement the planned solution and shows the sequence in which they are performed.

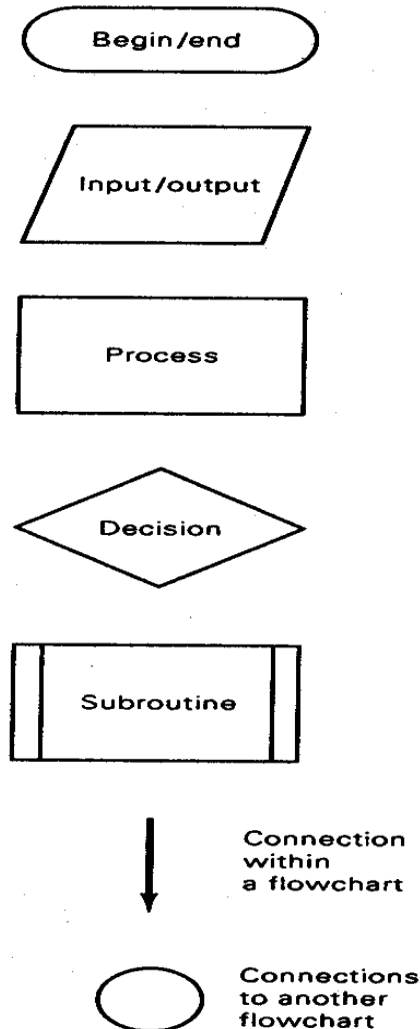
3.2 Assembly Language Programming Development on the PC

Flow chart of a block-move program



3.2 Assembly Language Programming Development on the PC

Commonly used flowchart symbols



3.2 Assembly Language Programming Development on the PC

- ▶ Coding the solution with assembly language
 - Two types of statements are used in the source program:
 - The *assembly language instructions* are used to tell the microprocessor what operations are to be performed to implement the application.
 - A *directive* is the instruction to the assembler program used to convert the assembly language program into machine code.

3.2 Assembly Language Programming Development on the PC

- ▶ Coding the solution with assembly language
 - The *assembly language instructions*

[Example]

```
MOV    AX, DATASEGMENT
MOV    DS, AX
MOV    SI, BLK1ADDR
MOV    DI, BLK2ADDR
MOV    CX, N
```

- The *directive*

[Example]

```
BLOCK      PROC FAR
or
BLOCK      ENDP
```

3.2 Assembly Language Programming Development on the PC

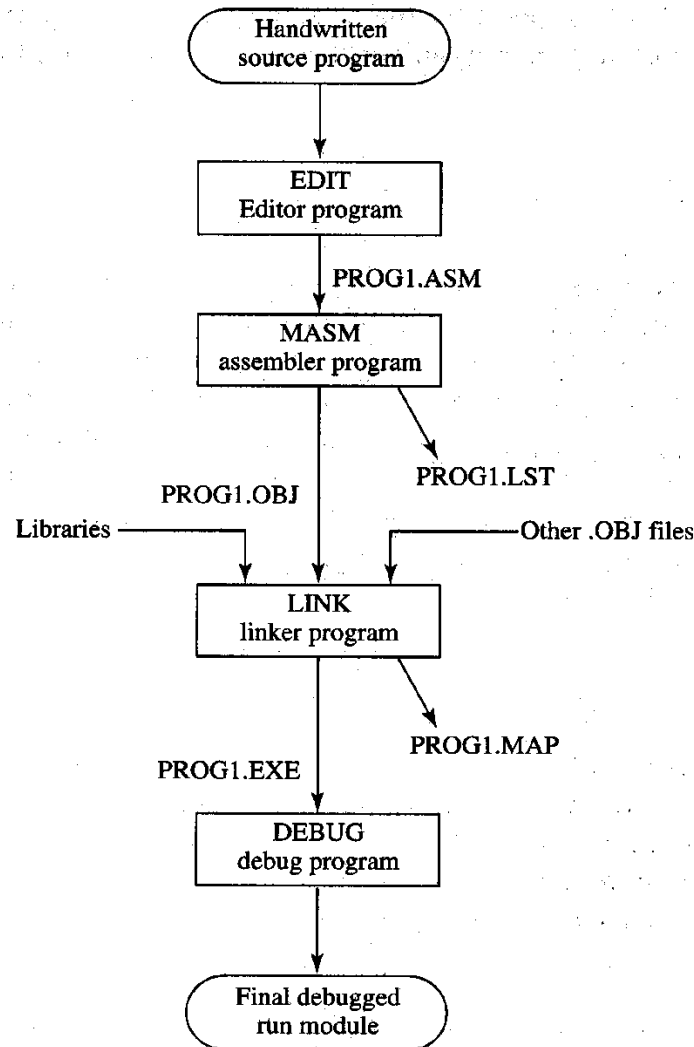
- ▶ Creating the source program
 - The EDIT editor
 - The Notepad editor in Windows
 - The Microsoft PWB (Programmer's Work Bench)
- ▶ Assembling the source program into an object module
 - The Microsoft MASM assembler
 - The Microsoft PWB (Programmer's Work Bench)
 - Used for writing and compiling code
 - The *assembler source file* and the *object module*

3.2 Assembly Language Programming Development on the PC

- ▶ Producing a run module
 - The object module must be processed by the LINK program to produce an executable *run module*.
- ▶ Verifying the solution
- ▶ Programs and files involved in the program development cycle
 - PROG1.ASM (Editor)
 - PROG1.OBJ (Assembler)
 - PROG1.LST (Assembler)
 - PROG1.EXE (Linker)
 - PROG1.MAP (Linker)

3.2 Assembly Language Programming Development on the PC

The development programs and users files



3.3 The Instruction Set

- ▶ The instruction set of a microprocessor defines the **basic operations** that a programmer can specify to the device to perform
- ▶ Instruction set groups
 - Data transfer instructions (moving data between registers and/or memory locations).
 - Arithmetic instructions (ADD,SUB,DIV,MUL,INC,DEC)
 - Logic instructions (AND,OR,XOR,ROL,NOT)
 - String manipulation instructions (REP,MOVS,LODS,STDS)
 - Control transfer instructions (JMP,RET,LOOP)
 - Processor control instructions (CLC,CMC,STC,HLT,)

3.3 The Instruction Set

- ▶ In assembly language each instruction is represented by a “**mnemonic**” that describes its operation and is called its “operation code (**opcode**)”
 - MOV = move → data transfer
 - ADD = add → arithmetic
 - JMP = unconditional jump → control transfer
- ▶ **Operands**: Identify whether the elements of data to be processed are in **registers** or **memory**
 - Source operand– location of one operand to be processed
 - Destination operand—location of the other operand to be processed and the location of the result

3.3 The Instruction Set

► Data transfer instructions

Mnemonic and Description	Instruction Code			
DATA TRANSFER				
MOV – Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1	
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		
PUSH – Push:				
Register/Memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m		
Register	0 1 0 1 0 reg			
Segment Register	0 0 0 reg 1 1 0			
POP – Pop:				
Register/Memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m		
Register	0 1 0 1 1 reg			
Segment Register	0 0 0 reg 1 1 1			
XCHG – Exchange:				
Register/Memory with Register	1 0 0 0 0 1 1 w	mod reg r/m		
Register with Accumulator	1 0 0 1 0 reg			

3.3 The Instruction Set

► Data transfer instructions

Mnemonic and Description	Instruction Code	
DATA TRANSFER		
IN – Input from:		
Fixed Port	1 1 1 0 0 1 0 w	port
Variable Port	1 1 1 0 1 1 0 w	
OUT – Output to:		
Fixed Port	1 1 1 0 0 1 1 w	port
Variable Port	1 1 1 0 1 1 1 w	
XLAT – Translate Byte to AL	1 1 0 1 0 1 1 1	
LEA – Load EA to Register	1 0 0 0 1 1 0 1	mod reg r/m
LDS – Load Pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m
LES – Load Pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m
LAHF – Load AH with Flags	1 0 0 1 1 1 1 1	
SAHF – Store AH into Flags	1 0 0 1 1 1 1 0	
PUSHF – Push Flags	1 0 0 1 1 1 0 0	
POPF – Pop Flags	1 0 0 1 1 1 0 1	

3.3 The Instruction Set

► Arithmetic instructions

Mnemonic and Description	Instruction Code			
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
ARITHMETIC				
ADD – Add:				
Reg./Memory with Register to Either	000000dw	mod reg r/m		
Immediate to Register/Memory	100000sw	mod 000r/m	data	data if s: w = 01
Immediate to Accumulator	0000010w	data	data if w = 1	
ADC – Add with Carry:				
Reg./Memory with Register to Either	000100dw	mod reg r/m		
Immediate to Register/Memory	100000sw	mod 010r/m	data	data if s: w = 01
Immediate to Accumulator	0001010w	data	data if w = 1	
INC – Increment:				
Register/Memory	1111111w	mod 000r/m		
Register	01000reg			
AAA – ASCII Adjust for Add	00110111			
DAA – Decimal Adjust for Add	00100111			
SUB – Subtract:				
Reg./Memory and Register to Either	001010dw	mod reg r/m		
Immediate from Register/Memory	100000sw	mod 101r/m	data	data if s: w = 01
Immediate from Accumulator	0010110w	data	data if w = 1	
SSB – Subtract with Borrow				
Reg./Memory and Register to Either	000110dw	mod reg r/m		
Immediate from Register/Memory	100000sw	mod 011r/m	data	data if s: w = 01
Immediate from Accumulator	000111w	data	data if w = 1	

3.3 The Instruction Set

► Arithmetic instructions

Mnemonic and Description	Instruction Code			
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
ARITHMETIC				
DEC – Decrement:				
Register/memory	1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
NEG – Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		
CMP – Compare:				
Register/Memory and Register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with Register/Memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s w = 01
Immediate with Accumulator	0 0 1 1 1 1 0 w	data	data if w = 1	
AAS – ASCII Adjust for Subtract	0 0 1 1 1 1 1			
DAS – Decimal Adjust for Subtract	0 0 1 0 1 1 1			
MUL – Multiply (Unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
IMUL – Integer Multiply (Signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
AAM – ASCII Adjust for Multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
DIV – Divide (Unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
IDIV – Integer Divide (Signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
AAD – ASCII Adjust for Divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
CBW – Convert Byte to Word	1 0 0 1 1 0 0 0			
CWD – Convert Word to Double Word	1 0 0 1 1 0 0 1			

3.3 The Instruction Set

► Logic instructions

Mnemonic and Description	Instruction Code			
LOGIC	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
NOT – Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m		
SHL/SAL – Shift Logical/Arithmetic Left	1 1 0 1 0 0 v w	mod 1 0 0 r/m		
SHR – Shift Logical Right	1 1 0 1 0 0 v w	mod 1 0 1 r/m		
SAR – Shift Arithmetic Right	1 1 0 1 0 0 v w	mod 1 1 1 r/m		
ROL – Rotate Left	1 1 0 1 0 0 v w	mod 0 0 0 r/m		
ROR – Rotate Right	1 1 0 1 0 0 v w	mod 0 0 1 r/m		
RCL – Rotate Through Carry Flag Left	1 1 0 1 0 0 v w	mod 0 1 0 r/m		
RCR – Rotate Through Carry Right	1 1 0 1 0 0 v w	mod 0 1 1 r/m		
AND – And:				
Reg./Memory and Register to Either	0 0 1 0 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w = 1
Immediate to Accumulator	0 0 1 0 0 1 0 w	data	data if w = 1	
TEST – And Function to Flags, No Result:				
Register/Memory and Register	1 0 0 0 0 1 0 w	mod reg r/m		
Immediate Data and Register/Memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate Data and Accumulator	1 0 1 0 1 0 0 w	data	data if w = 1	
OR – Or:				
Reg./Memory and Register to Either	0 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w = 1
Immediate to Accumulator	0 0 0 0 1 1 0 w	data	data if w = 1	
XOR – Exclusive or:				
Reg./Memory and Register to Either	0 0 1 1 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w = 1
Immediate to Accumulator	0 0 1 1 0 1 0 w	data	data if w = 1	

3.3 The Instruction Set

► String manipulation instructions

Mnemonic and Description	Instruction Code
STRING MANIPULATION	
REP – Repeat	1111001z
MOVS – Move Byte/Word	1010010w
CMPS – Compare Byte/Word	1010011w
SCAS – Scan Byte/Word	1010111w
LODS – Load Byte/Wd to AL/AX	1010110w
STOS – Stor Byte/Wd from AL/A	1010101w

3.3 The Instruction Set

► Control transfer instructions

Mnemonic and Description	Instruction Code		
CONTROL TRANSFER			
CALL – Call:			
Direct within Segment	1 1 1 0 1 0 0 0	disp-low	disp-high
Indirect within Segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	
Direct Intersegment	1 0 0 1 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect Intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	
JMP – Unconditional Jump:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within Segment	1 1 1 0 1 0 0 1	disp-low	disp-high
Direct within Segment-Short	1 1 1 0 1 0 1 1	disp	
Indirect within Segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
Direct Intersegment	1 1 1 0 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect Intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	

3.3 The Instruction Set

► Control transfer instructions

Mnemonic and Description	Instruction Code		
RET – Return from CALL:			
Within Segment	1 1 0 0 0 0 1 1		
Within Seg Adding Immed to SP	1 1 0 0 0 0 1 0	data-low	data-high
Intersegment	1 1 0 0 1 0 1 1		
Intersegment Adding Immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high
JE/JZ – Jump on Equal/Zero	0 1 1 1 0 1 0 0	disp	
JL/JNGE – Jump on Less/Not Greater or Equal	0 1 1 1 1 1 0 0	disp	
JLE/JNG – Jump on Less or Equal/Not Greater	0 1 1 1 1 1 1 0	disp	
JB/JNAE – Jump on Below/Not Above or Equal	0 1 1 1 0 0 1 0	disp	
JBE/JNA – Jump on Below or Equal/Not Above	0 1 1 1 0 1 1 0	disp	
JP/JPE – Jump on Parity/Parity Even	0 1 1 1 1 0 1 0	disp	
JO – Jump on Overflow	0 1 1 1 0 0 0 0	disp	
JS – Jump on Sign	0 1 1 1 1 0 0 0	disp	
JNE/JNZ – Jump on Not Equal/Not Zero	0 1 1 1 0 1 0 1	disp	
JNL/JGE – Jump on Not Less/Greater or Equal	0 1 1 1 1 1 0 1	disp	
JNLE/JG – Jump on Not Less or Equal/Greater	0 1 1 1 1 1 1 1	disp	

3.3 The Instruction Set

► Control transfer instructions

Mnemonic and Description	Instruction Code	
JNB/JAE – Jump on Not Below/Above or Equal	0 1 1 1 0 0 1 1	disp
JNBE/JA – Jump on Not Below or Equal/Above	0 1 1 1 0 1 1 1	disp
JNP/JPO – Jump on Not Par/Par Odd	0 1 1 1 1 0 1 1	disp
JNO – Jump on Not Overflow	0 1 1 1 0 0 0 1	disp
JNS – Jump on Not Sign	0 1 1 1 1 0 0 1	disp
LOOP – Loop CX Times	1 1 1 0 0 0 1 0	disp
LOOPZ/LOOPE – Loop While Zero/Equal	1 1 1 0 0 0 0 1	disp
LOOPNZ/LOOPNE – Loop While Not Zero/Equal	1 1 1 0 0 0 0 0	disp
JCXZ – Jump on CX Zero	1 1 1 0 0 0 1 1	disp
INT – Interrupt		
Type Specified	1 1 0 0 1 1 0 1	type
Type 3	1 1 0 0 1 1 0 0	
INTO – Interrupt on Overflow	1 1 0 0 1 1 1 0	
IRET – Interrupt Return	1 1 0 0 1 1 1 1	

3.3 The Instruction Set

► Process control instructions

Mnemonic and Description	Instruction Code	
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
PROCESSOR CONTROL		
CLC – Clear Carry	1 1 1 1 1 0 0 0	
CMC – Complement Carry	1 1 1 1 0 1 0 1	
STC – Set Carry	1 1 1 1 1 0 0 1	
CLD – Clear Direction	1 1 1 1 1 1 0 0	
STD – Set Direction	1 1 1 1 1 1 0 1	
CLI – Clear Interrupt	1 1 1 1 1 0 1 0	
STI – Set Interrupt	1 1 1 1 1 0 1 1	
HLT – Halt	1 1 1 1 0 1 0 0	
WAIT – Wait	1 0 0 1 1 0 1 1	
ESC – Escape (to External Device)	1 1 0 1 1 x x x	mod x x x r/m
LOCK – Bus Lock Prefix	1 1 1 1 0 0 0 0	

3.4 The MOV Instruction

- ▶ The move (MOV) instruction is used to transfer a byte or a word of data from a source operand to a destination operand.

- ▶ e.g. `MOV DX, CS`
`MOV [SUM], AX`

- Note that the MOV instruction cannot transfer data directly between external memory.

Mnemonic	Meaning	Format	Operation	Flags affected
MOV	Move	MOV D,S	(S) → (D)	None

(a)

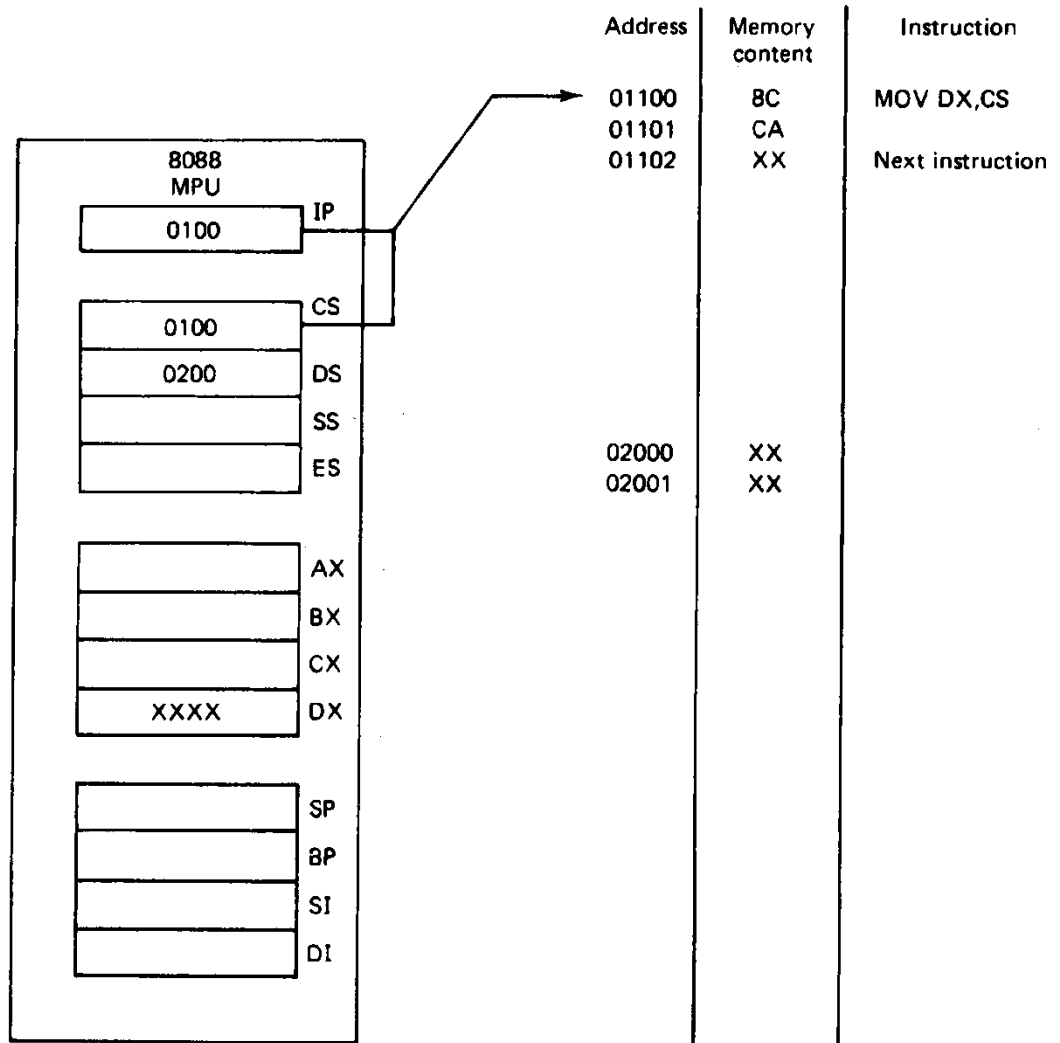
Destination	Source
Memory	Accumulator
Accumulator	Memory
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Seg-reg	Reg16
Seg-reg	Mem16
Reg16	Seg-reg
Memory	Seg-reg

(b)

Allowed operands for MOV instruction

3.4 The MOV Instruction

► MOV DX, CS

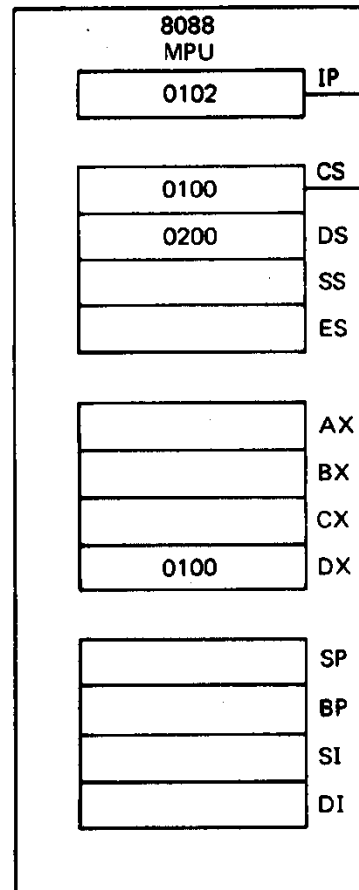


Before execution

(c)

3.4 The MOV Instruction

► MOV DX, CS



Address	Memory content	Instruction
01100	8C	MOV DX,CS
01101	CA	
01102	XX	Next instruction
02000	XX	
02001	XX	

(d)

After execution

3.5 Addressing Modes

- ▶ Addressing mode is a method of specifying an **operand** & categorized into three types:
 - **Register** operand addressing mode
 - **Immediate** operand addressing mode
 - **Memory** operand addressing mode
 - **Direct** addressing mode
 - **Register indirect** addressing mode
 - **Based** addressing mode
 - **Indexed** addressing mode
 - **Based-indexed** addressing mode

3.5 Addressing Modes

- ▶ **Register** operand addressing mode:

The operand to be accessed is specified as residing in an internal register of 8088.

e.g. **MOV AX, BX**

- ▶ Only the **data registers** can be accessed as bytes or words
 - Ex. AL,AH → bytes
 - AX → word
- ▶ **Index** and **pointer** registers as words
 - Ex. SI → word pointer
- ▶ **Segment** registers only as words
 - Ex. DS → word pointer

Register	Operand sizes	
	Byte (Reg 8)	Word (Reg 16)
Accumulator	AL, AH	AX
Base	BL, BH	BX
Count	CL, CH	CX
Data	DL, DH	DX
Stack pointer	—	SP
Base pointer	—	BP
Source index	—	SI
Destination index	—	DI
Code segment	—	CS
Data segment	—	DS
Stack segment	—	SS
Extra segment	—	ES

3.5 Addressing Modes

▶ Register operand addressing mode

• Example

MOV AX,BX

Source = BX → word data

Destination = AX → word data

Operation: (BX) → (AX)

• State before fetch and execution

CS:IP = 0100:0000 = 01000H

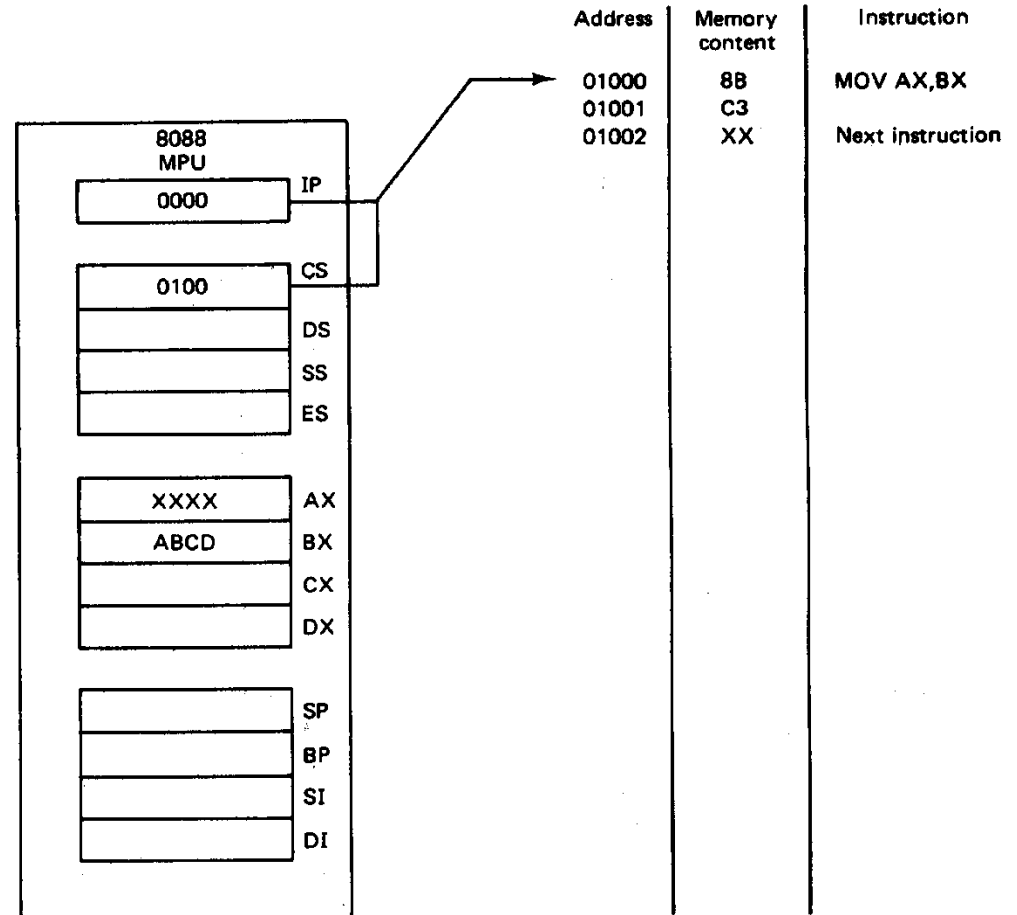
Move instruction code = 8BC3H

(01000H) = 8BH

(01001H) = C3H

(BX) = ABCDH

(AX) = XXXX → don't care state



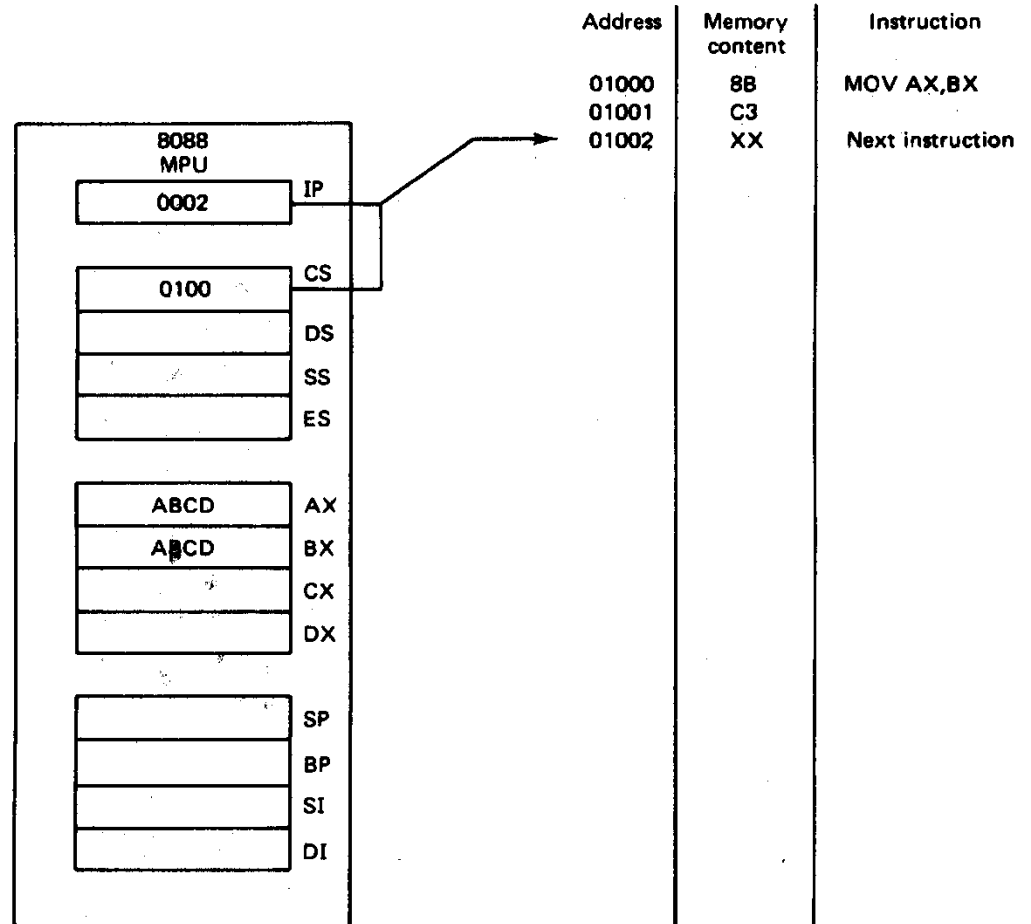
Before execution

(a)

3.5 Addressing Modes

▶ Register operand addressing mode

- Example (continued)
- State after execution
 $CS:IP = 0100:0002 = 01002H$
 $01002H \rightarrow$ points to next sequential instruction
 $(BX) = ABCDH$
 $(AX) = ABCDH \rightarrow$ Value in BX copied into AX



(b)

After execution

3.5 Addressing Modes

- ▶ **Immediate** operand addressing mode
 - Operand is coded as **part of the instruction**
 - Applies only to the **source** operand
 - **Destination** operand uses **register** addressing mode

- ▶ **Types**

- **Imm8** = 8-bit immediate operand
- **Imm16** = 16-bit immediate operand
- General instruction structure and operation

MOV Rx,ImmX

ImmX → (Rx)

Before execution

3.5 Addressing Modes

► Immediate operand addressing mode

• Example

MOV AL,15H

Source = Imm8 → immediate byte data

Destination = AL → Byte of data

• Operation: (Imm8) → (AL)

• State before fetch and execution

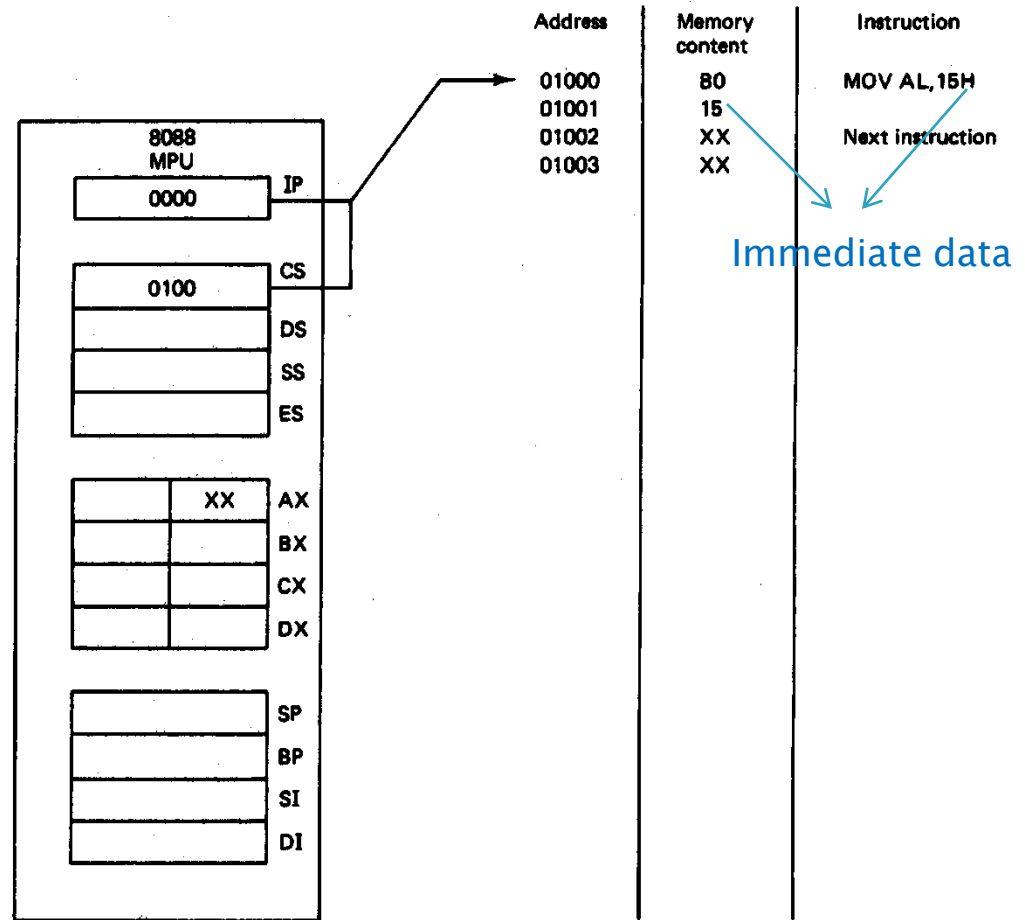
CS:IP = 0100:0000 = 01000H

Move instruction code = B015H

(01000H) = B0H

(01001H) = 15H → Immediate data

What about MOV AL,1515H ?



(a)

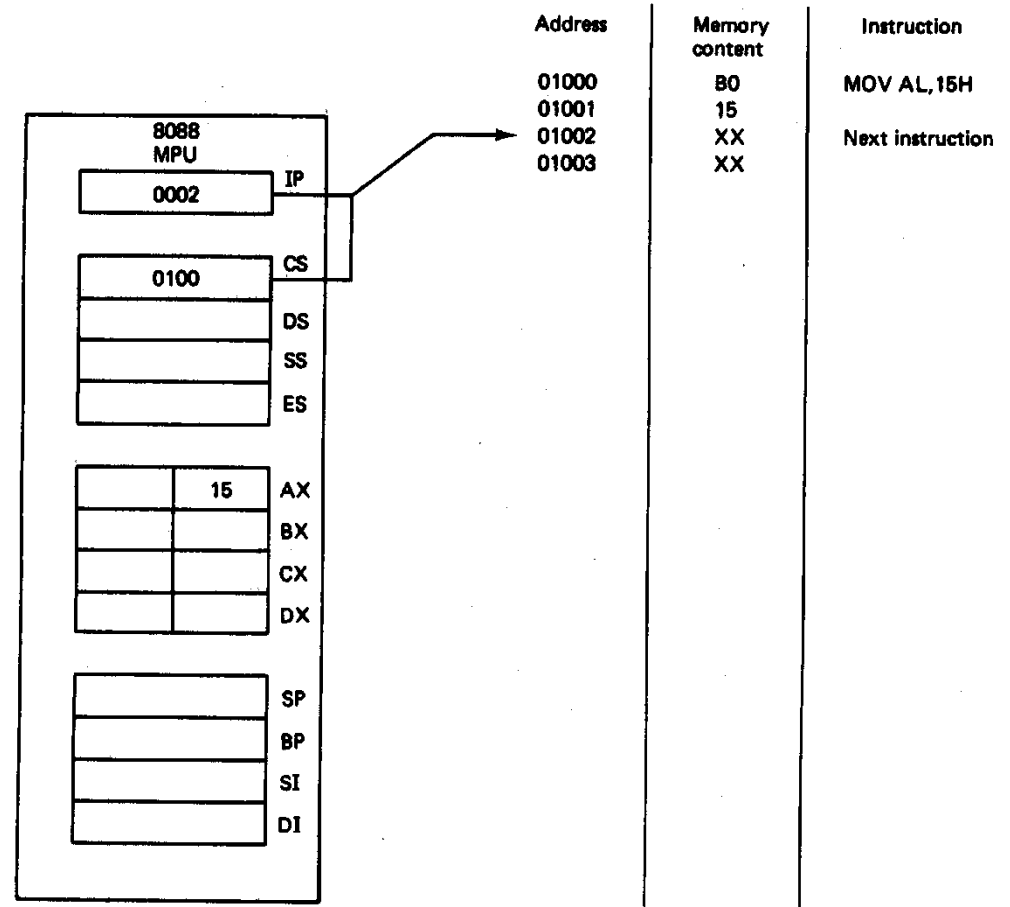
Before execution

3.5 Addressing Modes

► Immediate operand addressing mode

- Example (continued)
- State after execution
(AH) = XX → don't care state

What about MOV AL,1515H ?



(b)

After execution

3.5 Addressing Modes

► Memory addressing modes

To reference an operand in memory, the 8088 must calculate the physical address (PA) of the operand and then initiate a read or write operation to this storage location.

Physical Address (PA) = Segment Base Address (SBA) + Effective Address (EA)

PA = SBA : EA

PA = Segment base : Base + Index + Displacement

$$PA = \left\{ \begin{array}{c} CS \\ SS \\ DS \\ ES \end{array} \right\} : \left\{ \begin{array}{c} BX \\ BP \end{array} \right\} + \left\{ \begin{array}{c} SI \\ DI \end{array} \right\} + \left\{ \begin{array}{c} \text{8-bit displacement} \\ \text{16-bit displacement} \end{array} \right\}$$

EA = Base + Index + Displacement

Where:

SBA = Segment base address

EA = Effective address (offset)

• Components of a effective address

- Base → base registers BX or BP
- Index → index register SI or DI
- Displacement → 8 or 16-bit displacement
- Not all elements are used in all computations—results in a variety of addressing modes

Physical and effective address computation for memory operands

Memory addressing modes

Effective address (EA)

Memory addressing mode	Example	Base	Index	Disp.
Direct	MOV CX,[1234]			✓
Indirect	MOV AX,[SI]	✓		
Based	MOV [BX]+ 1234H,AL	✓		✓
Indexed	MOV AL,[SI]+1234H		✓	✓
Based-Index	MOV AH, [BX][SI]+1234H	✓	✓	✓

$$PA = SBA + EA$$

$$EA = \text{Base} + \text{Index} + \text{Displacement}$$

3.5 Addressing Modes

► Memory addressing modes – Direct addressing mode

PA = Segment base: Direct address

$$PA = \left\{ \begin{array}{c} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ \text{Direct address} \right\}$$

The default segment register is DS

Computation of a direct
memory address

e.g. MOV AX, [1234H]

- Similar to immediate addressing in that information coded directly into the instruction

- Immediate information is the effective address called the direct address

- Physical address computation

PA = SBA:EA → 20-bit address

PA = SBA:[DA] → immediate 8-bit or 16 bit displacement

[DA]: Displacement Address

- Segment base address is DS by default

PA = DS:[DA]

- Segment override prefix (SEG) is required to enable use of another segment register

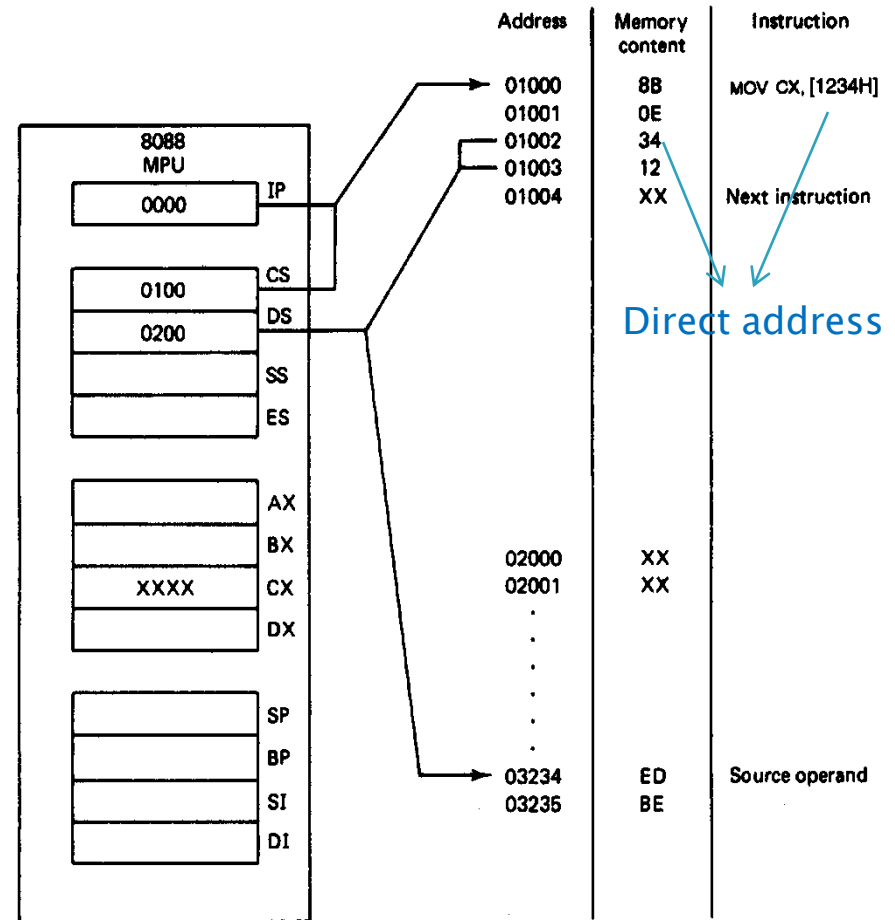
PA = SEG:ES:[DA]

3.5 Addressing Modes

- ▶ Memory addressing modes – Direct addressing mode
- Example

MOV CX,[1234H]

- State before fetch and execution
 - Instruction
- CS = 0100H
 IP = 0000H
 CS:IP = 0100:0000H = 01000H
 (01000H,01001H) = Opcode = 8B0E
 (01003H,01002) = DA = 1234H
- Source operand—direct addressing
- DS = 0200H
 DA = 1234H
 PA = DS:DA = 0200H:1234H
 = 02000H+1234H = 03234H
 (03235H,03234H) = BEEDH
- Destination operand--register addressing
- (CX) = XXXX → don't care state



Before execution (a)

3.5 Addressing Modes

► Memory addressing modes – Direct addressing mode

- Example (continued)
- State after execution

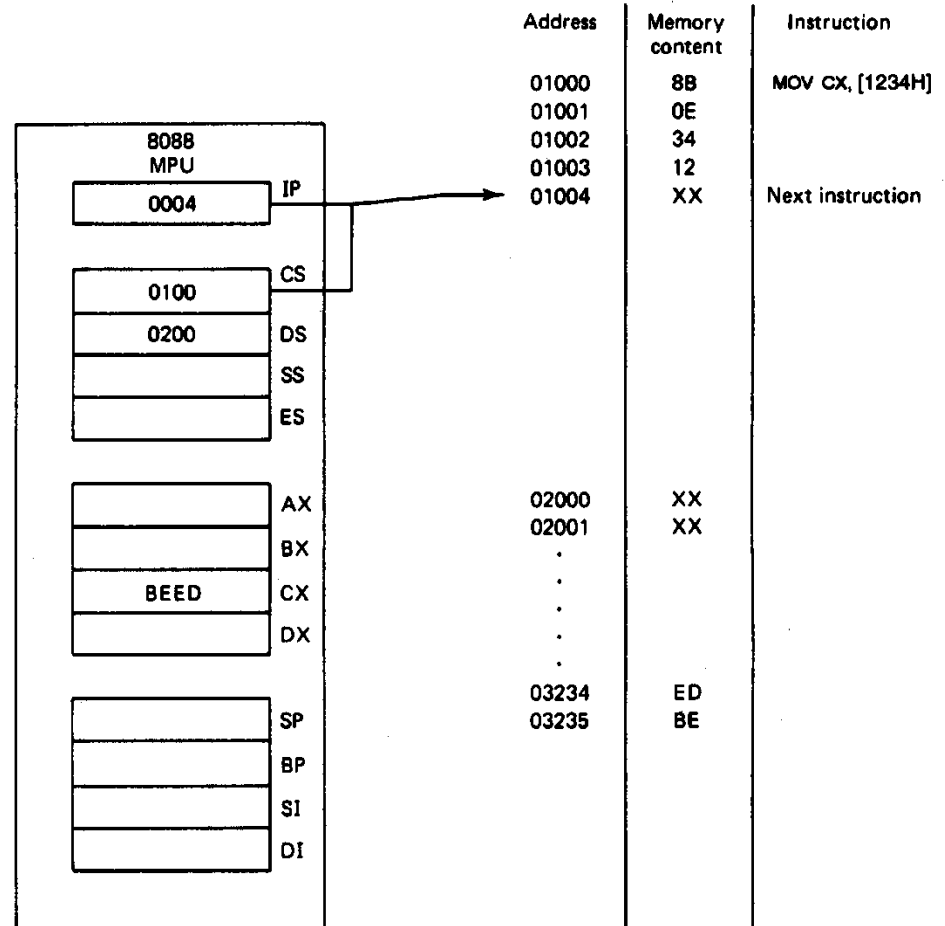
Instruction

CS:IP = 0100:0004 = 01004H

01004H → points to next

Sequential instruction

- Source operand
(03235H,03234H) = BEEDH → unchanged
- Destination operand
(CX) = BEEDH



(b)

After execution

3.5 Addressing Modes

- ▶ Memory addressing modes – **Register indirect addressing mode**

PA = Segment Base : Direct Address

$$PA = \left\{ \begin{array}{c} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ \begin{array}{c} BX \\ BP \\ SI \\ DI \end{array} \right\}$$

The default segment register is DS
Computation of an indirect memory address

e.g. **MOV AX, [SI]**

- Similar to direct addressing in that the effective address is combined with the contents of DS to obtain the physical address

- Effective address resides in either a **base** or **index** register

- Physical address computation

PA = SBA:EA → 20-bit address

PA = SBA:[Rx] → 16-bit offset

- Segment base address is **DS** by default for BX, SI, and DI

- Segment base address is **SS** by default for BP

PA = DS:[Rx]

- Segment override prefix (SEG) is required to enable use of another segment register

PA = SEG:ES:[Rx]

3.5 Addressing Modes

▶ Memory addressing modes – Register indirect addressing mode

$$PA = 02000_{16} + 1234_{16} = 03234_{16}$$

• Example

MOV AX,[SI]

• State before fetch and execution Instruction

CS = 0100H

IP = 0000H

CS:IP = 0100:0000H = 01000H

(01000H,01001H) = Opcode = 8B04H

• Source operand–register indirect addressing

DS = 0200H

SI = 1234H

PA = DS:SI = 0200H:1234H

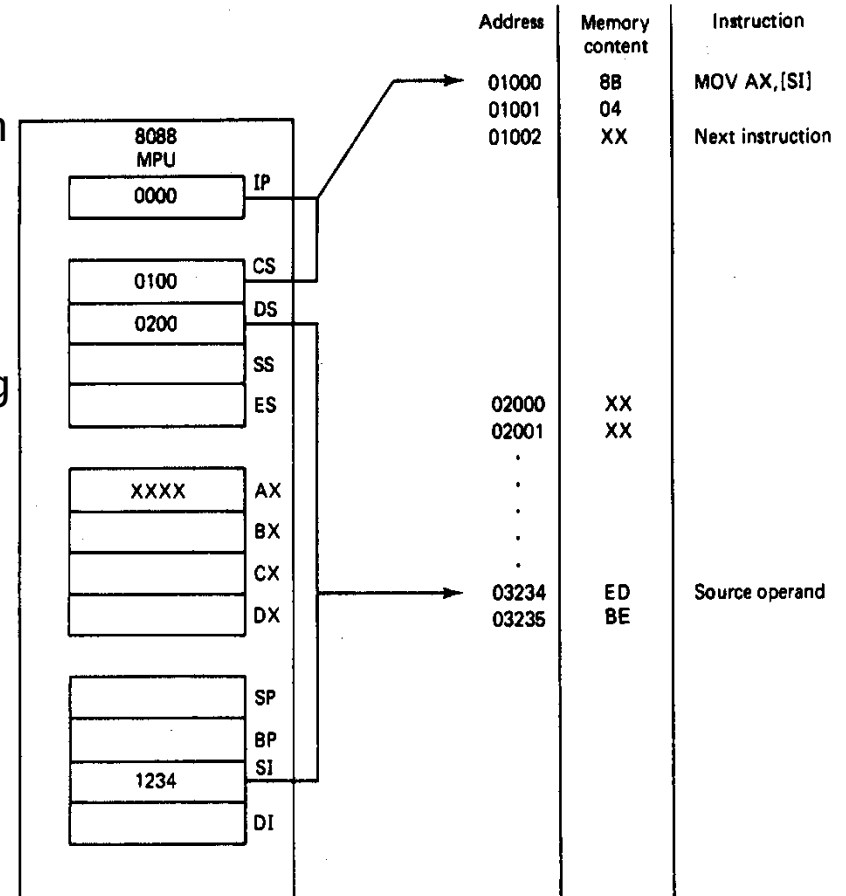
= 02000H + 1234H

= 03234H

(03235H,03234H) = BEEDH

• Destination operand–register operand addressing

(AX) = XXXX → don't care state



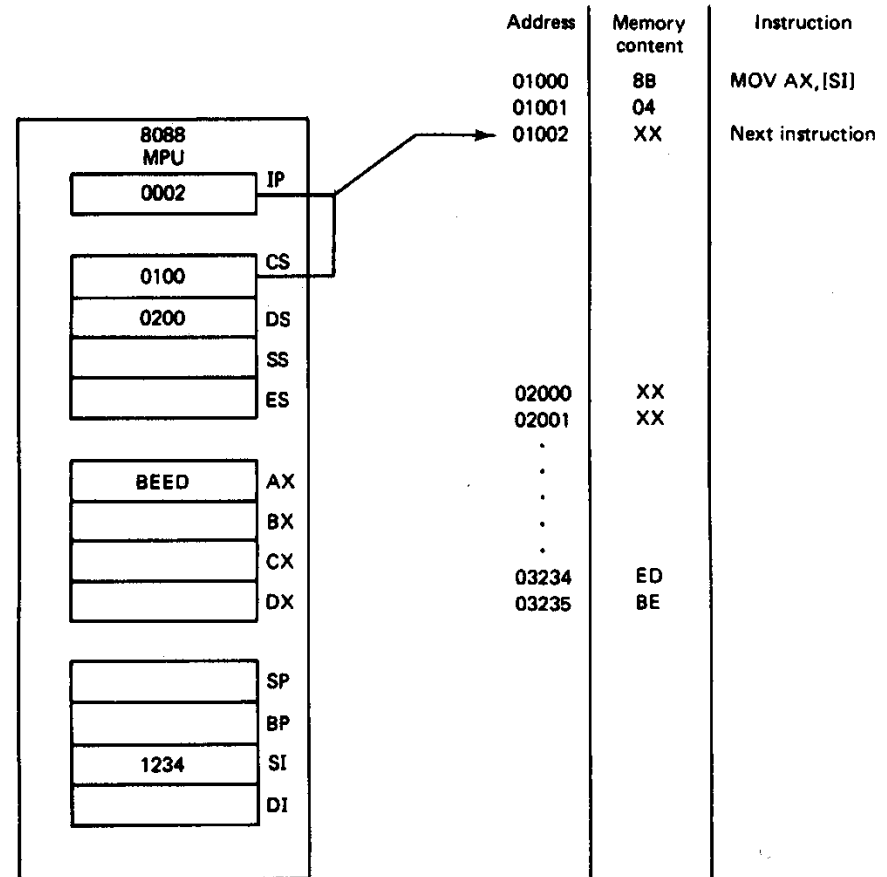
Before execution^(a)
CPE 0408330

3.5 Addressing Modes

- Memory addressing modes – Register indirect addressing mode

$$PA = 02000_{16} + 1234_{16} = 03234_{16}$$

- Example (continued)
 - State after execution Instruction
 $CS:IP = 0100:0002 = 01002H$
 $01002H \rightarrow$ points to next sequential instruction
 - Source operand
 $(03235H, 03234H) = BEEDH \rightarrow$ unchanged
 - Destination operand
 $(AX) = BEEDH$



(b)

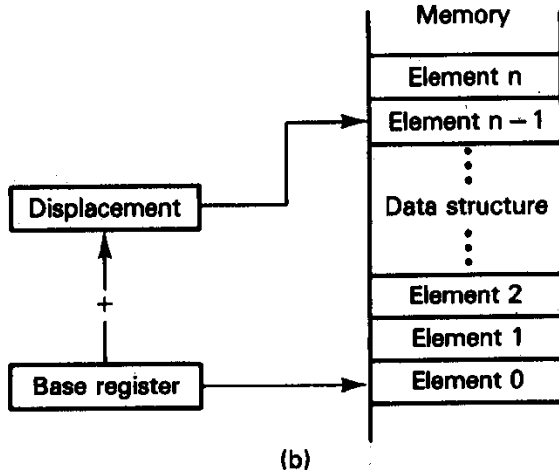
After execution

3.5 Addressing Modes

► Memory addressing modes – Based addressing mode

$$PA = \left\{ \begin{matrix} CS \\ DS \\ SS \\ ES \end{matrix} \right\} : \left\{ \begin{matrix} BX \\ BP \end{matrix} \right\} + \left\{ \begin{matrix} 8\text{-bit displacement} \\ 16\text{-bit displacement} \end{matrix} \right\}$$

(a)



- Effective address formed from contents of a base register and a displacement
- Base register is either BX or BP (stack)
 - Direct/indirect displacement is 8-bit or 16bit
- Physical address computation
 $PA = SBA:EA \rightarrow 20\text{-bit address}$
 $PA = SBA:[BX \text{ or } BP] + DA$
- Accessing a data structure
 - Based addressing makes it easy to access elements of data in an **array**
 - Address in base register points to **start** of the array
 - Displacement selects the element **within** the array
 - Value of the displacement is simply changed to access **another element** in the array
 - Program changes value in base register to select **another array**

e.g. **MOV [BX]+1234H, AL**

3.5 Addressing Modes

Memory addressing modes – Based addressing mode

$$PA = 02000_{16} + 1000_{16} + 1234_{16} = 04234_{16}$$

Example

MOV [BX] +1234H,AL

State before fetch and execution Instruction

CS = 0100H, IP = 0000H

CS:IP = 0100:0000H = 01000H

(01000H,01001H) = Opcode = 8887H

(01002H,01003H) = Direct displacement = 1234H

Destination operand—based addressing

DS = 0200H, BX = 1000H, DA = 1234H

PA = DS:DS+DA = 0200H:1000H+1234H

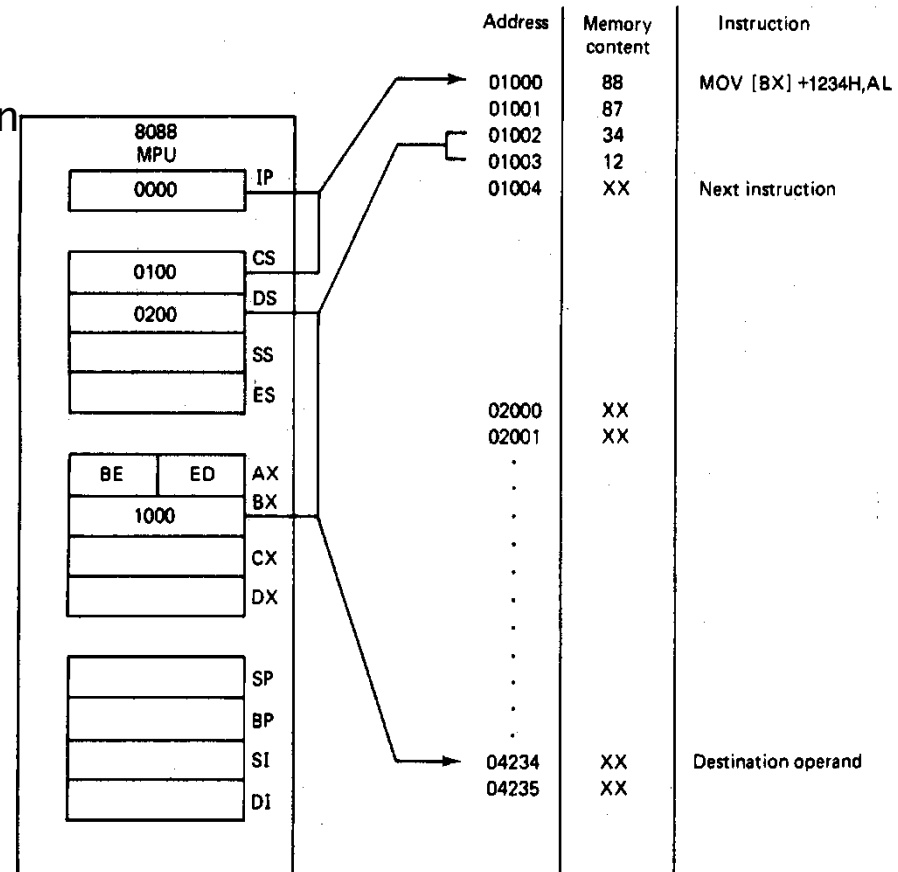
= 02000H+1000H+1234H

= 04234H

(04234H) = XXH

Source operand—register operand addressing

(AL) = ED



Before execution

(a)

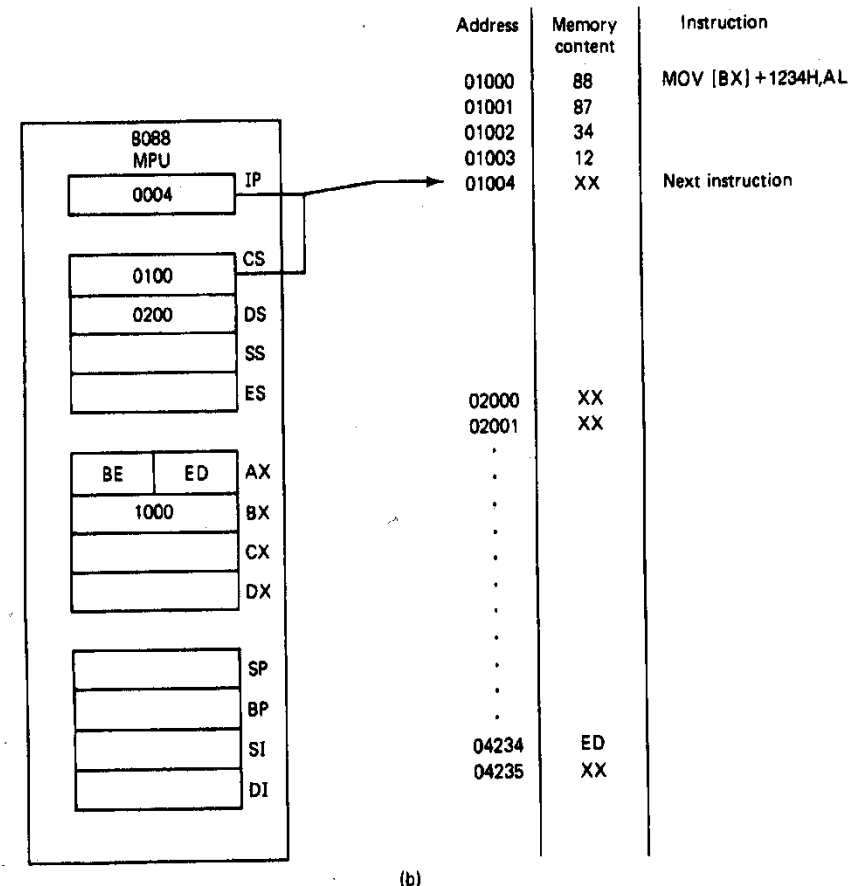
3.5 Addressing Modes

- Memory addressing modes – Based addressing mode

$$PA = 02000_{16} + 1000_{16} + 1234_{16} = 04234_{16}$$

- Example (continued)
- State after execution Instruction
CS:IP = 0100:0004 = 01004H
01004H → points to next sequential instruction
- Destination operand
(04234H) = EDH
- Source operand
(AL) = EDH → **unchanged**

Note: if BP is used instead of BX, the calculation of PA is performed using SS instead of DS.

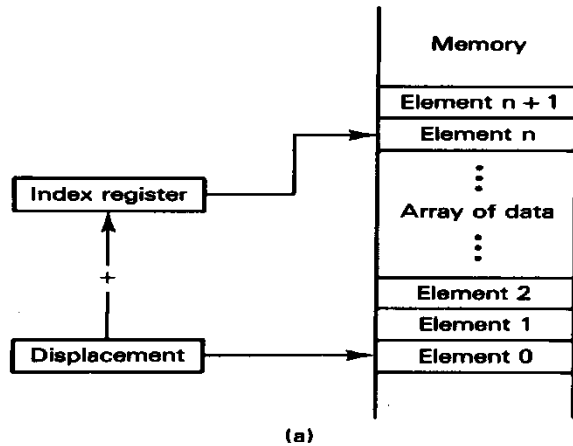


After execution

3.5 Addressing Modes

▶ Memory addressing modes – Indexed addressing mode

PA = Segment Base : Index + Displacement



PA = Segment base: Index + Displacement

$$PA = \left\{ \begin{matrix} CS \\ DS \\ SS \\ ES \end{matrix} \right\} : \left\{ \begin{matrix} SI \\ DI \end{matrix} \right\} + \left\{ \begin{matrix} 8\text{-bit displacement} \\ 16\text{-bit displacement} \end{matrix} \right\}$$

(b)

Computation of an indexed address

e.g. **MOV AL, [SI]+1234H**

- Similar to based addressing, it makes accessing elements of data in an array easy
 - Displacement points to the **beginning** of array in memory
 - Index register selects element **within** the array
 - Program simply changes the value of the displacement to access another array
 - Program changes (re-computes) value in index register to select another element in the array
 - Effective address formed from direct displacement and contents of an index register
 - Direct displacement is 8-bit or 16-bit
 - Index register is either SI → source operand or DI → destination operand
 - Physical address computation
- PA = SBA:EA → 20-bit address
 PA = SBA: DA + [SI or DI]

3.5 Addressing Modes

- Memory addressing modes – Indexed addressing mode

$$PA = 02000_{16} + 2000_{16} + 1234_{16} = 05234_{16}$$

• Example

MOV AL,[SI] +1234H

- State before fetch and execution Instruction

CS = 0100H

IP = 0000H

CS:IP = 0100:0000H = 01000H

(01000H,01001H) = Opcode = 8A84H

(01002H,01003H) = Direct displacement = 1234H

- Source operand—indexed addressing

DS = 0200H

SI = 2000H

DA = 1234H

PA = DS:SI+DA = 0200H:2000H+1234H

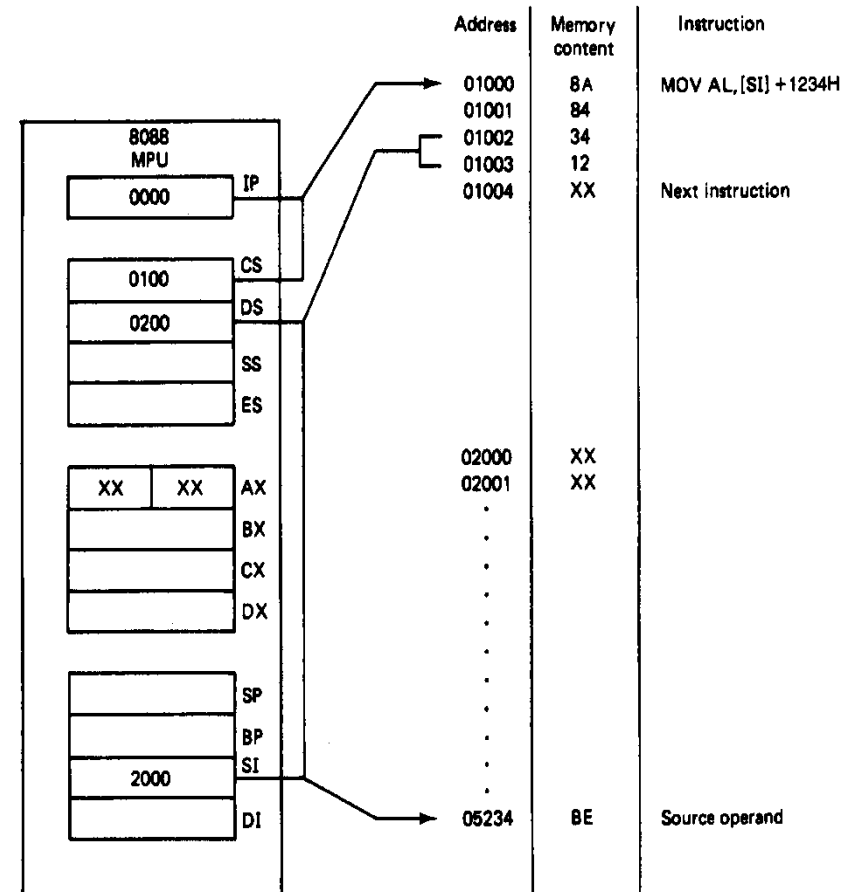
= 02000H+2000H+1234H

= 05234H

(05234H) = BEH

- Destination operand—register operand addressing

(AL) = XX → don't care state



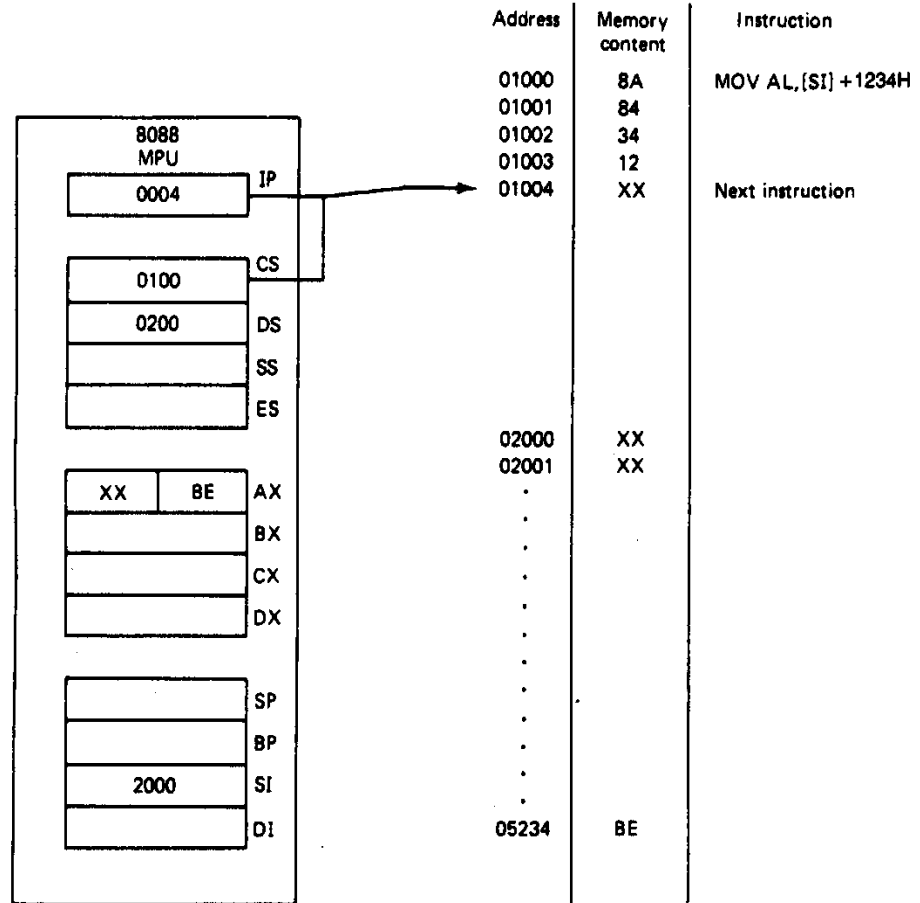
(a)
Before execution

3.5 Addressing Modes

- Memory addressing modes – Indexed addressing mode

$$PA = 02000_{16} + 2000_{16} + 1234_{16} = 05234_{16}$$

- Example (continued)
- State after execution Instruction
 $CS:IP = 0100:0004 = 01004H$
 $01004H \rightarrow$ points to next sequential instruction
- Source operand
 $(05234H) = BEH \rightarrow$ unchanged
- Destination operand
 $(AL) = BEH$

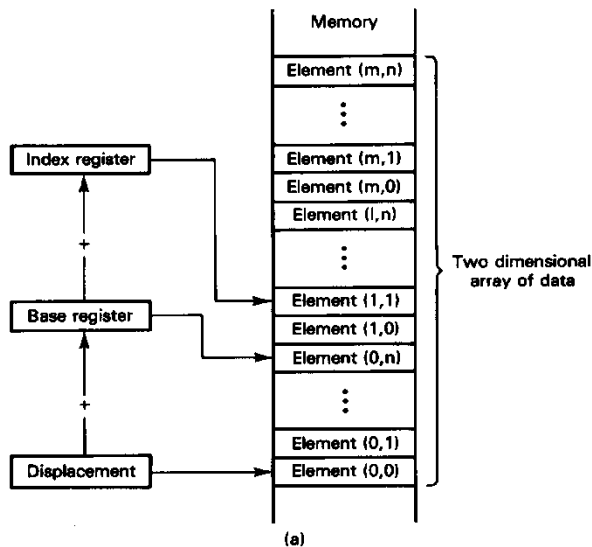


(b)

After execution

3.5 Addressing Modes

► Memory addressing modes – Based-indexed addressing mode



PA = Segment base: Base + Index + Displacement

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \begin{Bmatrix} 8\text{-bit displacement} \\ 16\text{-bit displacement} \end{Bmatrix}$$

(b)

Computation of an indexed address

e.g. `MOV AH, [BX][SI]+1234H`

- Combines the functions of based and indexed addressing modes
 - Enables easy access to **two-dimensional arrays** of data
 - **Displacement** points to the **beginning** of array in memory
 - **Base** register selects the **row (m)** of elements
 - **Index** register selects element in a **column (n)**
 - Program simply changes the value of the displacement to access another array
 - Program changes (re-computes) value in base register to select another row of elements
 - Program changes (re-computes) the value of the index register to select the element in another column
 - Effective address formed from direct displacement and contents of a base register and an index register
 - Direct displacement is 8-bit or 16bit
 - Base register either BX or BP (stack)
 - Index register is either SI → source operand or DI → destination operand
 - Physical address computation
- PA = SBA:EA → 20-bit address
 PA = SBA:DA + [BX or BP] + [SI or DI]

3.5 Addressing Modes

- ▶ Memory addressing modes – Based-indexed addressing mode

$$PA = 02000_{16} + 1000_{16} + 2000_{16} + 1234_{16} = 06234_{16}$$

- **Example**

MOV AH,[BX][SI] +1234H

- State before fetch and execution Instruction

CS = 0100H, IP = 0000H

CS:IP = 01 00:0000H = 01 000H

(01 000H, 01 001H) = Opcode = 8AA0H

(01002H,01003H) = Direct displacement = 1234H

- Source operand-based-indexed addressing

DA = 1234H, DS = 0200H, BX = 1000H,

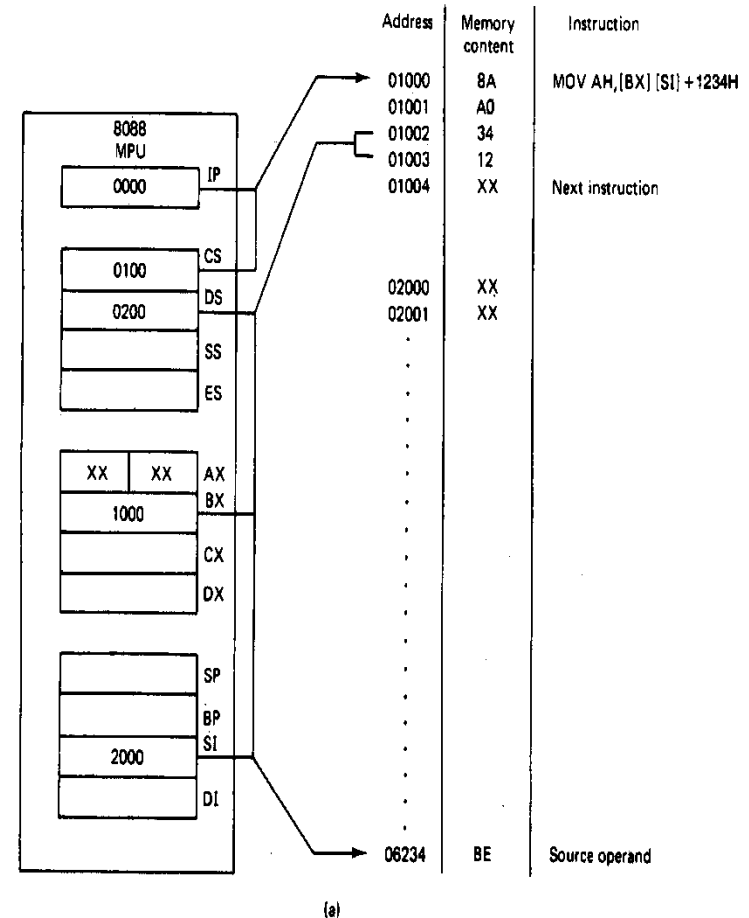
$$SI = 2000H$$
$$PA = DS:DA + BX + SI$$
$$= 0200\text{H}:1234\text{H} + 1000\text{H} + 2000\text{H}$$
$$= 02000H + 1234H + 1000H + 2000H$$

= 06234H

(06234H) = BEH

- Destination operand—register operand addressing

(AH) = XX \rightarrow don't care state



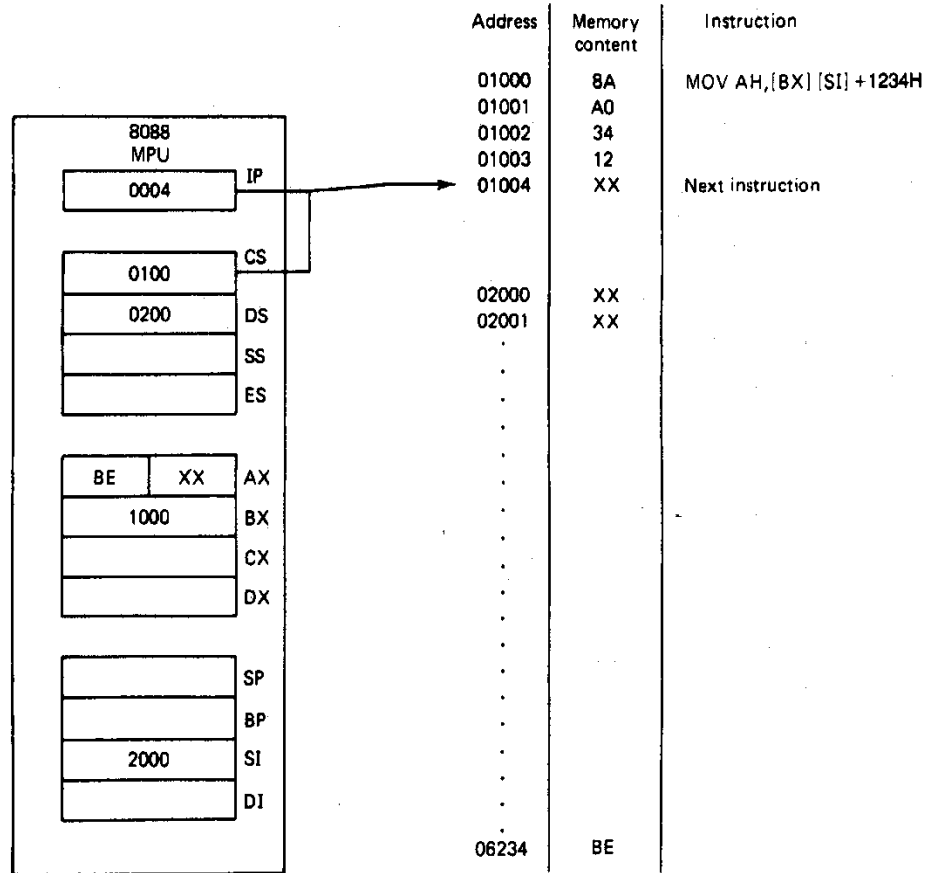
Before execution

3.5 Addressing Modes

- Memory addressing modes – Based-indexed addressing mode

$$PA = 02000_{16} + 1000_{16} + 2000_{16} + 1234_{16} = 06234_{16}$$

- Example (continued)
- State after execution Instruction
 $CS:IP = 0100:0004 = 01004H$
 $01004H \rightarrow$ points to next sequential instruction
- Source operand
 $(06234H) = BEH \rightarrow$ unchanged
- Destination operand
 $(AH) = BEH$



(b)

After execution

H.W. #3

- ❑ Solve the following problems from Chapter 3 from the course textbook:

5, 10, 23, 25, 26, 29, 31