# 16.480/552 Microprocessor II and Embedded Systems Design

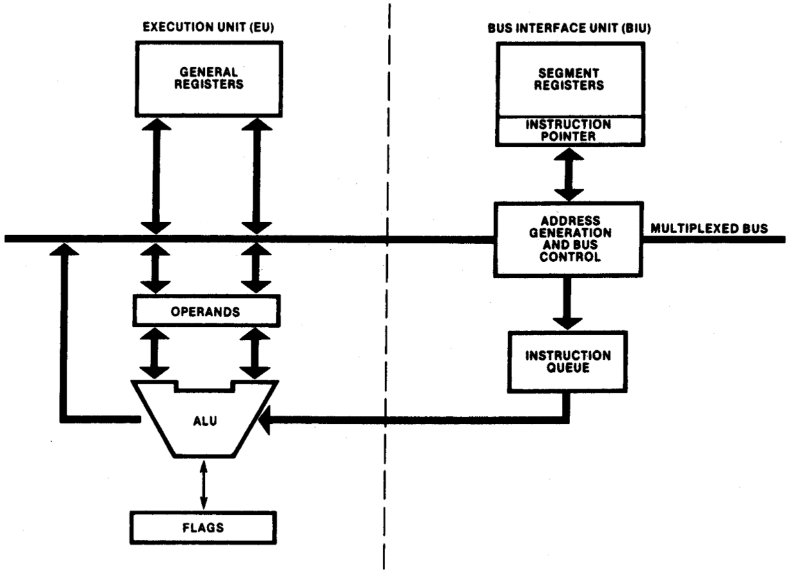
# Lecture 2: 8088/8086 Assembly Language Programming

*Revised based on* "*The 8088 and 8086 Microprocessors*" by Triebel and Singh

# Outline

- Embedded systems overview
  - What are they?
- Design challenge – optimizing design metrics
- Technologies
  - Processor technologies
  - IC technologies
  - Design technologies
- Introduction to 8088/8086

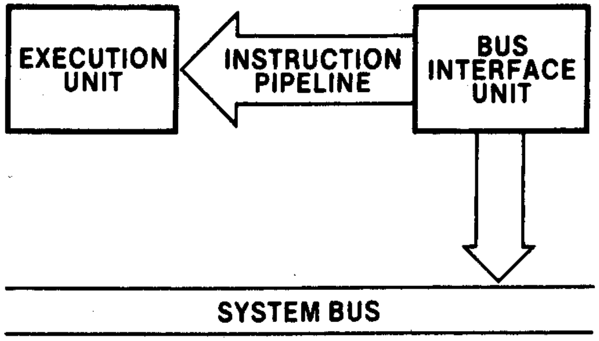
# Internal Architecture of the 8088/8086 Microprocessor- Parallel Processing



- Employs a multiprocessing architecture- parallel processing
- Two processing units:
  - Bus interface unit
  - Execution unit
- Each unit has dedicated functions and they both operate at the same time
- Parallel processing results in higher performance

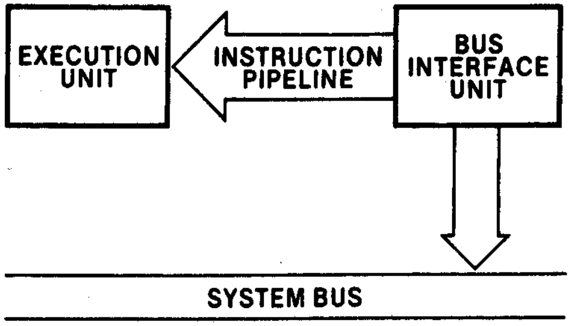
# Bus Interface Unit

- Interface to the outside world
- Key elements
  - Segment registers
    - Hold address information for accessing data
  - Instruction pointer
    - Holds address information for accessing code
  - Address generation/control logic
    - Creates address and external control signals
  - Instruction queue
    - Holds next instructions to be executed
- Responsibilities
  - Performs address generation and bus control
  - Fetching of instruction
  - Reading and writing of data for memory
  - Inputting and outputting of data for input/output peripherals
  - Prioritizes bus accesses—data operands highest priority



# System Bus

- System Bus
  - Interface between MPU and the memory and I/O subsystems
  - All code and data transfers take place over the "system bus"
    - Multiplexed address/data bus—address and data carried over same lines but at different times
    - 8088—8-bit wide data bus, 20 bit address bus, 1 byte/memory cycle
    - 8086—16-bit wide data bus, 20 bit address bus, 2 bytes/memory cycle
    - 1M-byte physical memory address space

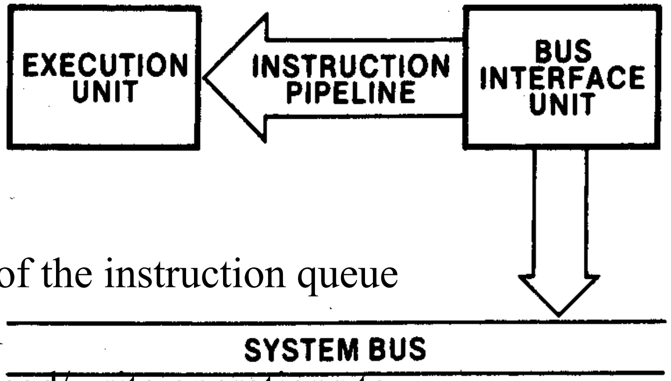


# Instruction Queue

- Instruction Queuing
  - BIU implements a mechanism known as the “instruction queue”
    - 8088 queue- 4 bytes
    - 8086 queue- 6 bytes
  - Whenever the queue is not full the BIU looks ahead in the program and performs bus cycles to pre-fetch the next sequential instruction code
    - FIFO instruction queue-  Bytes loaded at the input end of the queue automatically shift up to the empty location nearest the output
    - Bytes of code are held until the execution unit is ready to accept them
    - Code passed to the EU via instruction pipeline
  - Result is that the time needed to fetch many of the instructions in a microcomputer program is eliminated.
  - If queue is full and the EU is not requesting access to data in memory, BIU does not perform bus cycles (Idle states).

# Execution Unit

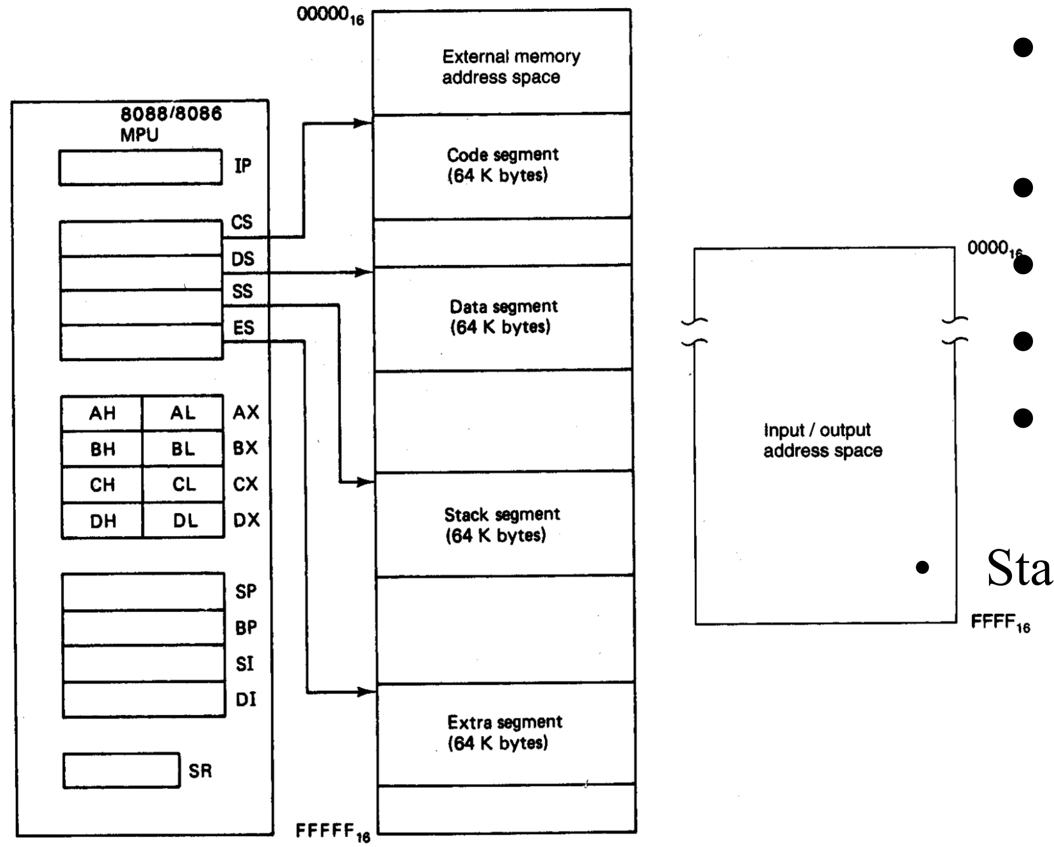
- Key elements of the EU
  - Arithmetic/logic unit (ALU)
    - Performs the operation identified by the instruction: ADD, SUB, AND, etc.
  - Flags register
    - Holds status and control information
  - General-purpose registers
    - Holds address or data information
- Responsible for decoding and execution of instructions
  - Reads machine code instructions from the output side of the instruction queue
  - Decodes the instructions to prepare them for execution
  - Generates addresses and requests the BIU to perform read/write operations to memory or I/O
  - Performs the operation identified by the instruction on the operands
  - Accesses data from the general purpose registers if necessary
  - Tests the state of flags if necessary
  - Updates the state of the flags based on the result produced by executing the instruction.



# The Software Model

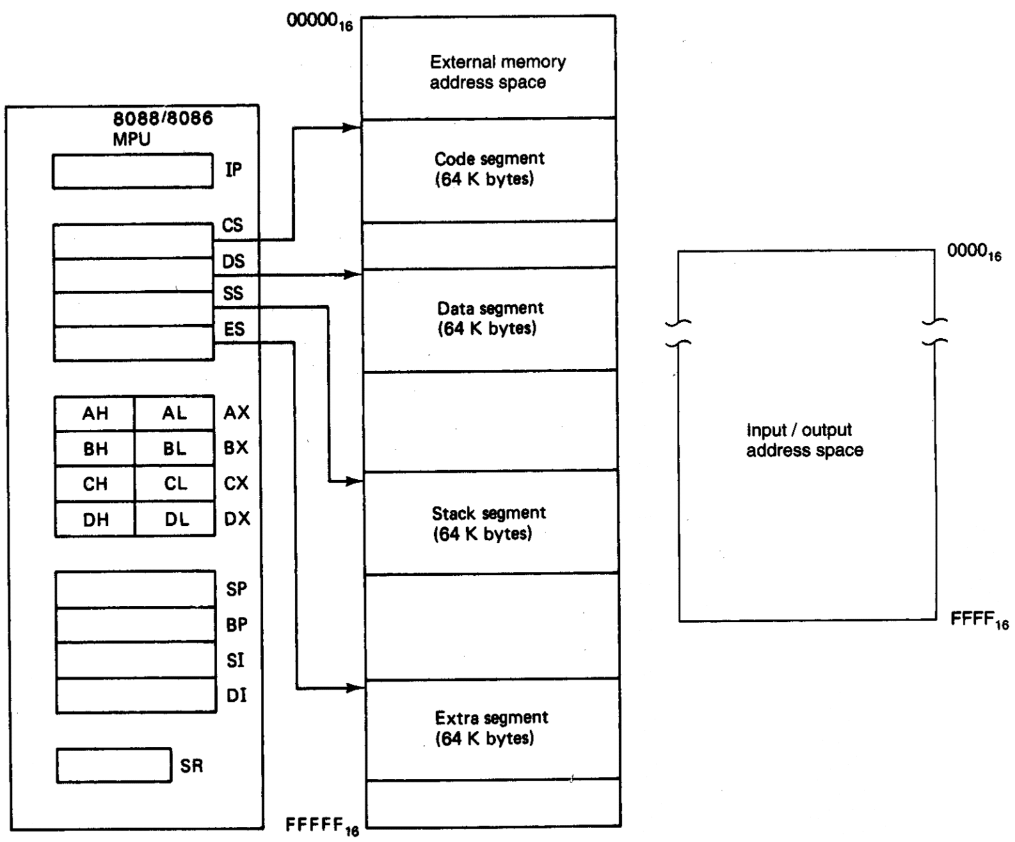
- Aid to the programmer in understanding the operation of the microcomputer from a software point of view
- Elements of the software model
  - Register set
  - Memory address space
  - Input/output address space
- What the programmer must know about the microprocessor
  - Registers available within the device
  - Purpose of each register
  - Function of each register
  - Operating capabilities of each register
  - Limitations of each register
  - Size of the memory and input/output address spaces
  - Organization of the memory and input/output address spaces
  - Types of data

# Register Set


00000₁₆
External memory address space
8088/8086 MPU
IP
CS
DS
SS
ES
Code segment (64 K bytes)
Data segment (64 K bytes)
Stack segment (64 K bytes)
Extra segment (64 K bytes)
AH
AL
AX
BH
BL
BX
CH
CL
CX
DH
DL
DX
SP
BP
SI
DI
SR
FFFFF₁₆
0000₁₆
Input / output address space
FFFF₁₆

- 13- 16-bit registers
  - (4) Data registers- AX, BX, CX, DX
  - (2) Pointer registers- BP, SP
  - (2) Index registers- SI, DI
  - (1) Instruction pointer- IP
  - (4) Segment registers- CS, DS, SS, ES
- Status register (SR)-FLAG

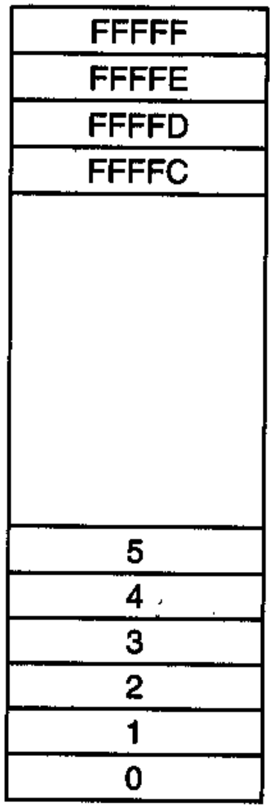
# Memory and Input/Output



00000₁₆
External memory address space
8088/8086 MPU
IP
CS
DS
SS
ES
Code segment (64 K bytes)
Data segment (64 K bytes)
AH
AL
AX
BH
BL
BX
CH
CL
CX
DH
DL
DX
Stack segment (64 K bytes)
SP
BP
SI
DI
Extra segment (64 K bytes)
SR
FFFFF₁₆
0000₁₆
Input / output address space
FFFF₁₆

- Architecture implements independent memory and input/output address spaces
- Memory address space- 1,048,576 bytes long (1M-byte)
- Input/output address space- 65,536 bytes long (64K-bytes)

# Address Space



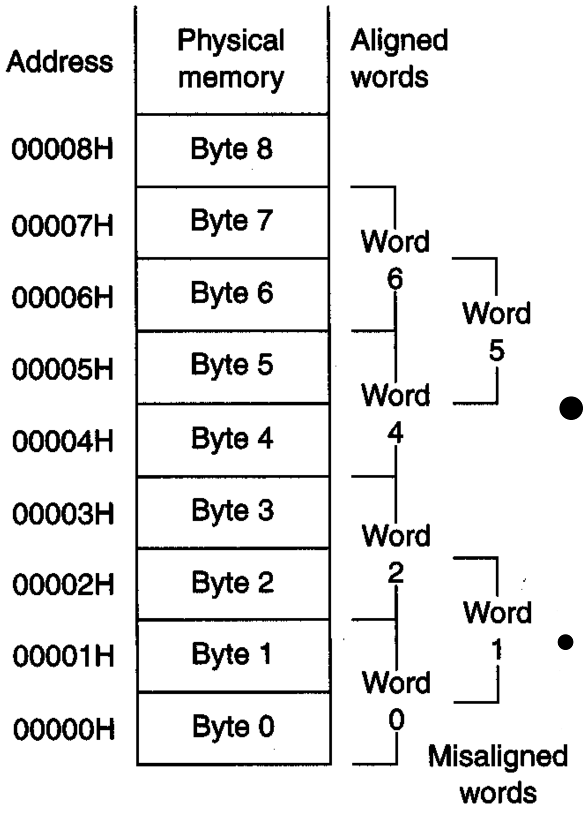
- Memory in the 8088/8086 microcomputer is organized as individual bytes
- Memory address space corresponds to the 1M addresses in the range 00000H to FFFFFH

  $00000H = 00000000000000000000_2$

  $FFFFFH = 11111111111111111111_2$

  $2^{20} = 1,048,576 = 1M$
- Data organization:
  - Double-word: contents of 4 contiguous byte addresses
  - Word: contents of two contiguous byte addresses
  - Byte: content of any individual byte address

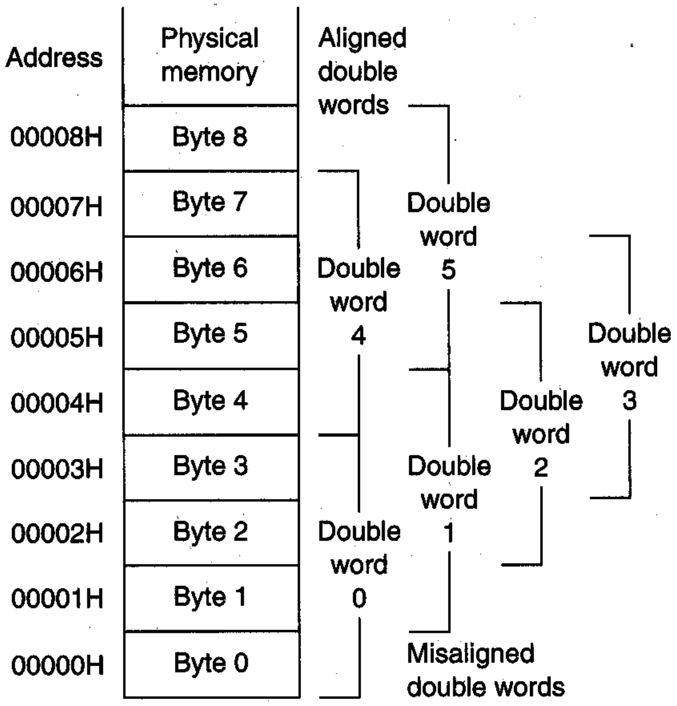
# Aligned and Misaligned Words



| Address | Physical memory |
|---|---|
| 00008H | Byte 8 |
| 00007H | Byte 7 |
| 00006H | Byte 6 |
| 00005H | Byte 5 |
| 00004H | Byte 4 |
| 00003H | Byte 3 |
| 00002H | Byte 2 |
| 00001H | Byte 1 |
| 00000H | Byte 0 |

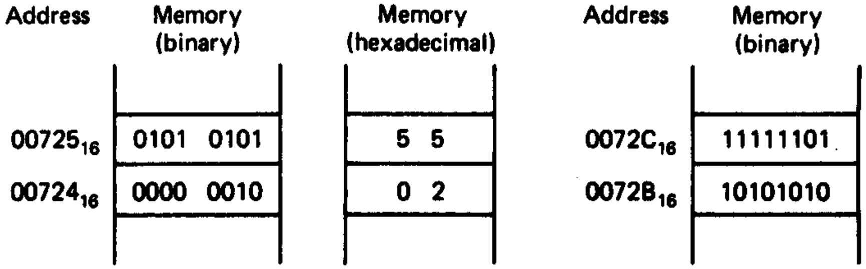
- Words and double words of data can be stored in memory at either an even or odd address boundary
  - Examples of even address boundaries: $00000_{16}$, $00002_{16}$, $00004_{16}$
  - Examples of odd address boundaries: $00001_{16}$, $00003_{16}$, $00005_{16}$
- Words stored at an even address boundary are said to be ***aligned words***
  - Examples are words 0, 2, 4, and 6
- Words stored at an odd address boundary are said to be ***misaligned or unaligned words***
  - Examples are words 1 and 5

# Aligned and Misaligned Double-Words



- Aligned double-words are stored at even addresses that are a multiple of 4
  - Examples are double-words 0 and 4
- Misaligned double-words are stored at addresses that are not a multiple of 4
  - Examples are double words 1, 2, 3, and 5
- There is a performance impact for accessing unaligned data in memory

# Examples of Words of Data



Example [Fig. 2.4 (a)]

($00725_{16}$) = $0101\ 0101_2$=55H= MS-byte

($00724_{16}$) = $0000\ 0010_2$=02H= LS-byte

as a word they give

$0101010100000010_2$=5502H

Address in binary form

$00724_{16}$ = $00000000011100100100_2$

Even address → Aligned word

Example 2.1 [Fig. 2.4 (b)]

($0072C_{16}$) = $1111\ 1101_2$=FDH= MS-byte

($0072B_{16}$) = $1010\ 1010_2$= AAH= LS-byte

as a word they give

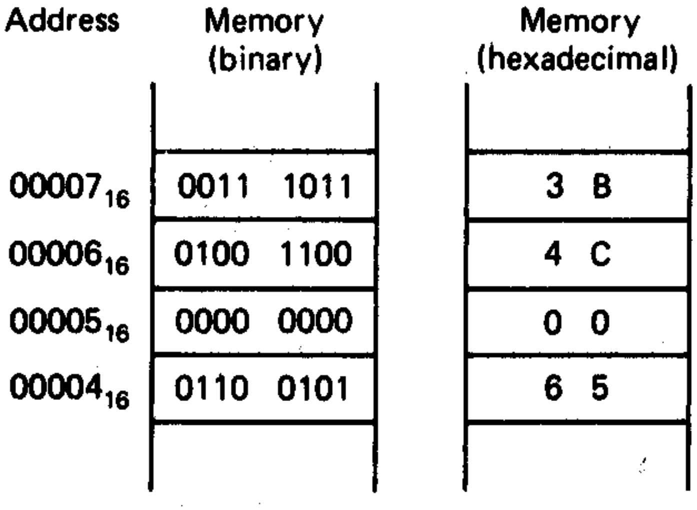
$1111110110101010_2$= FDAAH

Address in binary form

$0072B_{16}$= $00000000011100101011_2$

Odd address → misaligned word

# Example of Double Word Pointer



| Address | Memory (binary) | Memory (hexadecimal) |
|---|---|---|
| | | |
| $00007_{16}$ | 0011 1011 | 3 B |
| $00006_{16}$ | 0100 1100 | 4 C |
| $00005_{16}$ | 0000 0000 | 0 0 |
| $00004_{16}$ | 0110 0101 | 6 5 |
| | | |

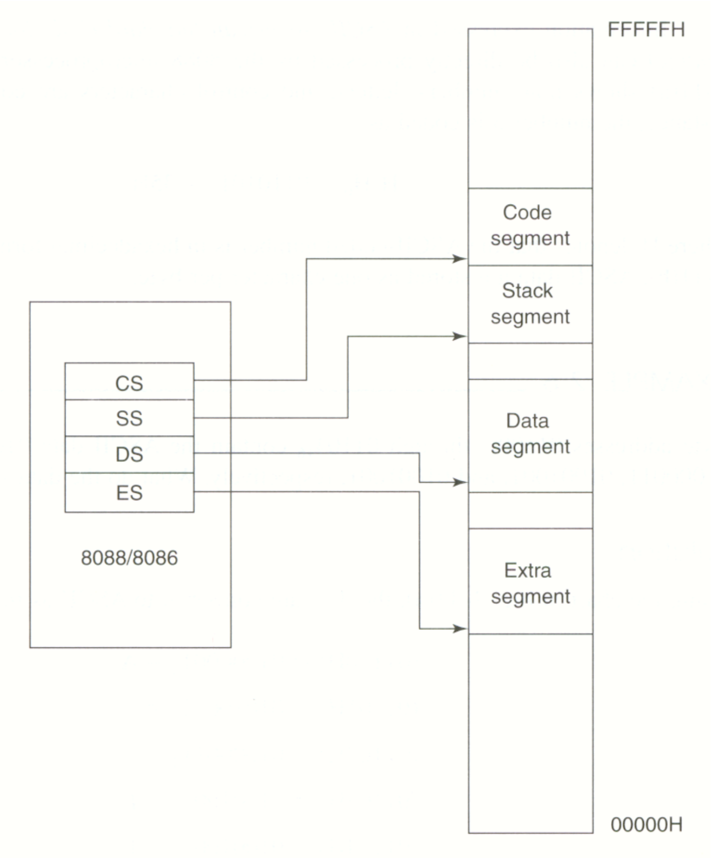
- Pointer consists of two 16 bit address elements: Segment base address and offset address
- LS-Byte:

  Address 00004H = 65H
- MS-Byte:

  Address 00007H = 3BH
- Arranging as double word gives the pointer

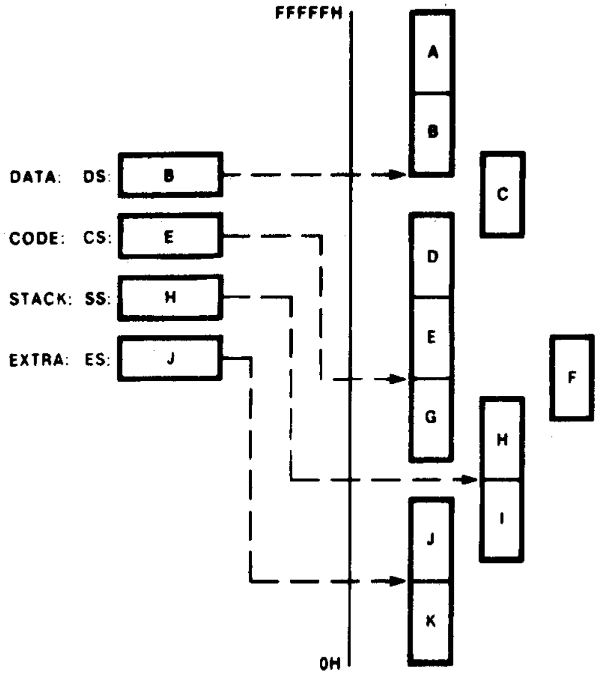
  Address= 00004H = 3B4C0065H
  - Since address is a multiple of 4 → aligned double word
  - Offset address = lower addressed word = 0065H
  - Segment base address = higher addressed word = 3B4CH

# Active Segments of Memory



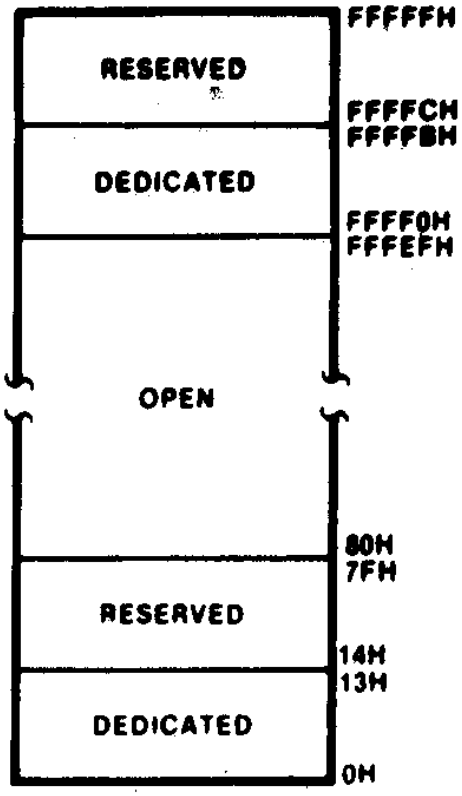
- Memory Segmentation
  - Not all of the 8088/8086 address space is active at one time
  - Address value in a segment register points to the lowest addressed byte in an active segment
  - Size of each segment is 64K contiguous bytes
  - Total active memory is 256k bytes
    - 64K-bytes for code
    - 64K-bytes for stack
    - 128K-bytes for data
- Four Segment Registers
  - Code segment (CS) register- Code storage
  - Stack segment (SS) register- Stack storage
  - Data segment (DS) register- Data storage
  - Extra segment (ES) register- Data storage

# User access, Restrictions, and Orientation



- Segment registers are user accessible
  - Programmer can change values under software control
  - Permits access to other parts of memory
  - Example: a new data space can be activated by replace the values in DS and ES
- Restriction on the address of a segment in memory
  - Must reside on a 16 byte address boundary
  - Examples: 00000H, 00010H, 00020H
- Orientation of segments:
  - Contiguous—A&B or D,E&G or JK
  - Adjacent
  - Disjointed—C&F
  - Overlapping—B&C or C&D

# Memory Map



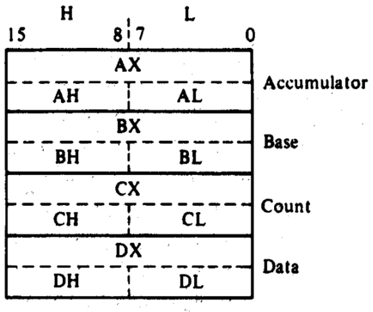
- Memory address space is partitioned into general use and dedicated use areas
- Dedicated/Reserved:
  - 0H → 7FH interrupt vector table
    - 1st 128 bytes
    - 32 4-byte pointers
      - 16-bit segment base address—2 MSBytes
      - 16-bit offset—2 LSBytes
  - 0H → 13H dedicated to internal interrupts and exceptions
  - 14H → 7FH reserved for external user-defined interrupts
  - FFFF0H → FFFFBH dedicated to hardware reset
  - FFFFCH → FFFFFH reserved for future products
- General use:
  - 80H → FFFEFH
  - Available for stack, code, and data

# Accessing Code Storage Space

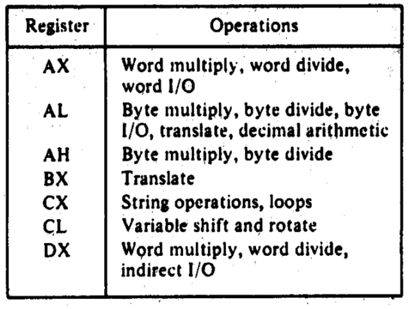
- Instruction pointer (IP): identifies the location of the next word of instruction code to be fetched from the current code segment
  - 16-bit offset—address pointer
  - CS:IP forms 20-bit physical address of next word of instruction code
- Instruction fetch sequence
  - 8088/8086 fetches a word of instruction code from code segment in memory
    - Increments value in IP by 2
    - Word placed in the instruction queue to await execution
    - 8088 prefetches up to 4 bytes of code
- Instruction execution sequence
  - Instruction is read from output of instruction queue and executed
    - Operands read from data memory, internal registers, or the instruction queue
    - Operation specified by the instruction performed on operands
    - Result written to data memory or internal register

# Internal Storage of Data and Addresses

- Four general purpose data registers
  - Accumulator (A) register
  - Base (B) register
  - Count (C) register
  - Data (D) register
- Can hold 8-bit or 16-bit data
  - AH/AL = high and low byte value
  - AX = word value
- Uses:
  - Hold data such as source or destination operands for most operations—ADD, AND, SHL
  - Hold address pointer for accessing memory
- Some also have dedicated special uses
  - C—count for loop, repeat string, shift, and rotate operations
  - B—Table look-up translations, base address
  - D—indirect I/O and string I/O



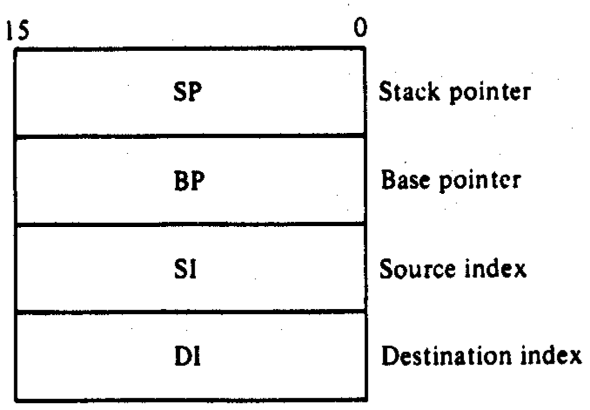
(a)



| Register | Operations |
|---|---|
| AX | Word multiply, word divide, word I/O |
| AL | Byte multiply, byte divide, byte I/O, translate, decimal arithmetic |
| AH | Byte multiply, byte divide |
| BX | Translate |
| CX | String operations, loops |
| CL | Variable shift and rotate |
| DX | Word multiply, word divide, indirect I/O |

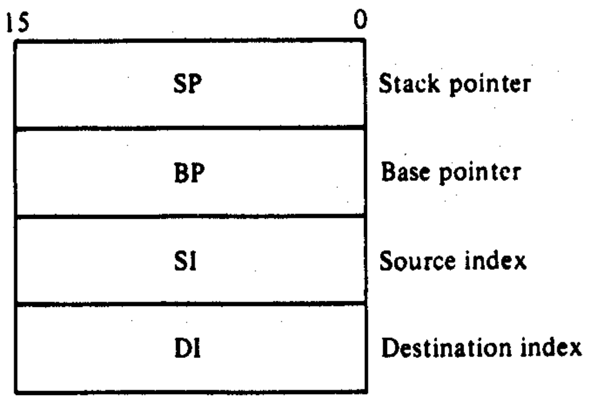
(b)

# Pointer and Index Registers- Accessing Information in Memory



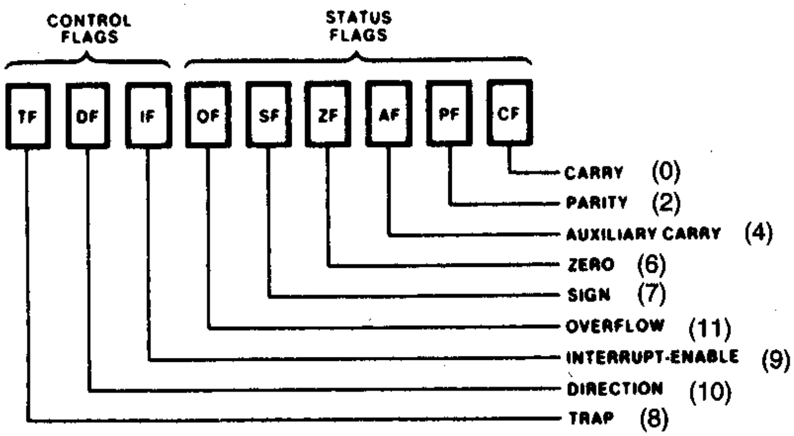
- Pointers are offset addresses used to access information in a segment of memory
- Two pointer registers
  - Stack pointer register
    - SP = 16-bit stack pointer
  - Base pointer register
    - BP = 16-bit base pointer
  - Access information in “stack segment” of memory
    - SP and BP are offsets from the current value of the stack segment base address
    - Select a specific storage location in the current 64K-byte stack segment
    - SS:SP—points to top of stack (TOS)
    - SS:BP—points to an element of data in stack

# Pointer and Index Registers- Accessing Information in Memory



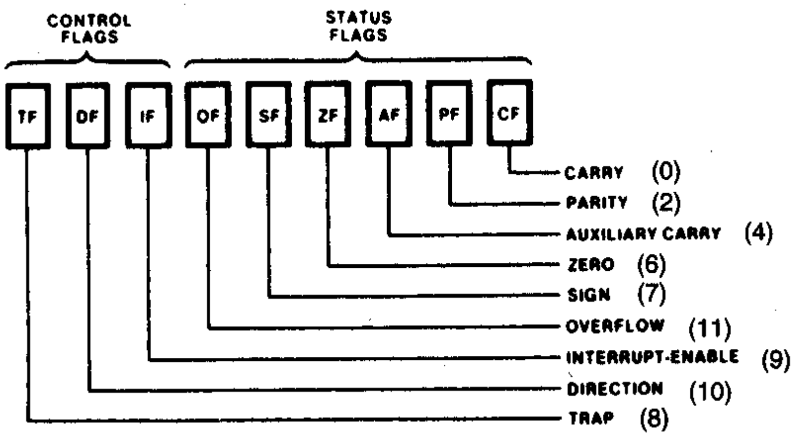
- Value in an index register is also an address pointer
- Two index registers
  - Source index register
    - SI = 16-bit source index register
  - Destination index register
    - DI = 16-bit destination index register
  - Access source and destination operands in data segment of memory
    - DS:SI—points to source operand in data segment
    - DS:DI—points to destination operand in data segment
    - Also used to access information in the extra segment (ES)

# Status Register- Status and Control Flags



- FLAGS register: 16-bit register used to hold single bit status and control information called flags
  - 9 active flags in real mode
  - Two categories
    - Status Flags—indicate conditions that are the result of executing an instruction
      - Execution of most instructions update status
      - Used by control flow instructions as test conditions
    - Control Flags—control operating functions of the processor
      - Used by software to turn on/off operating capabilities

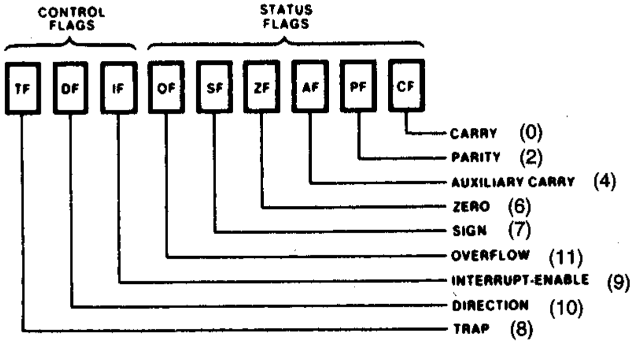
# Flags Register- Status Flags



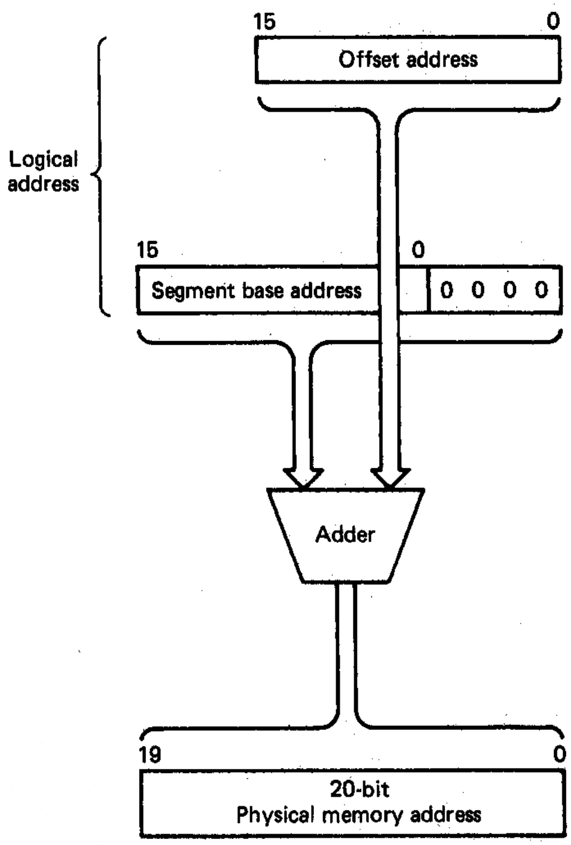
- Examples of Status Flags—CF, PF, ZF, SF, OF, AF
  - Carry flag (CF)
    - 1 = carry-out or borrow-in from MSB of the result during the execution of an arithmetic instruction
    - 0 = no carry has occurred
  - Parity flag (PF)
    - 1 = result produced has even parity
    - 0 = result produced has odd parity
  - Zero flag (ZF)
    - 1 = result produced is zero
    - 0 = result produced is not zero
  - Sign bit (SF)
    - 1 = result is negative
    - 0 = result is positive
  - Others
    - Overflow flag (OF)
    - Auxiliary carry flag (AF)

# Flags Register- Control Flags

- Examples of Control Flags—TF, IF, DF
  - Interrupt flag (IF)
    - Used to enable/disable external maskable interrupt requests
    - 1 = enable external interrupts
    - 0 = disable external interrupts
  - Trap flag (TF)
    - 1 = turns on single-step mode
    - 0 = turns off single step mode
    - Mode useful for debugging
    - Employed by monitor program to execute one instruction at at time (single step execution)
  - Direction flag (DF)
    - Used to determine the direction in which string operations occur
    - 1 = automatically decrement string address—proceed from high address to low address
    - 0 = Automatically increment string address—proceed from low address to high address



# Generating a Memory Address- Logical and Physical Addresses

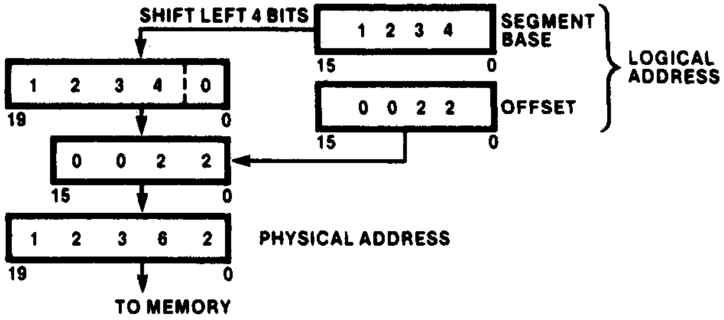


- Logical address: real-mode architecture described by a segment address and an offset
  - Segment base address (CS, DS, ES, SS) are 16 bit quantities
  - Offsets (IP, SI, DI, BX, DX, SP, BP, etc.) are 16 bit quantities
  - Examples:

| | | |
|---|---|---|
| CS:IP | 100H:100H | Code access |
| DS:SI | 2000H:1EFH | Data access |
| ES:DI | 3000H:0H | Data access |
| SS:SP | F000H:FFH | Stack access |

- Physical Address: actual address used for accessing memory
  - 20-bits in length
  - Formed by:
    - Shifting the value of the 16-bit segment base address left 4 bit positions
    - Filling the vacated four LSBs with 0s
    - Adding the 16-bit offset

# Generating a Memory Address- Example



- **Example:**

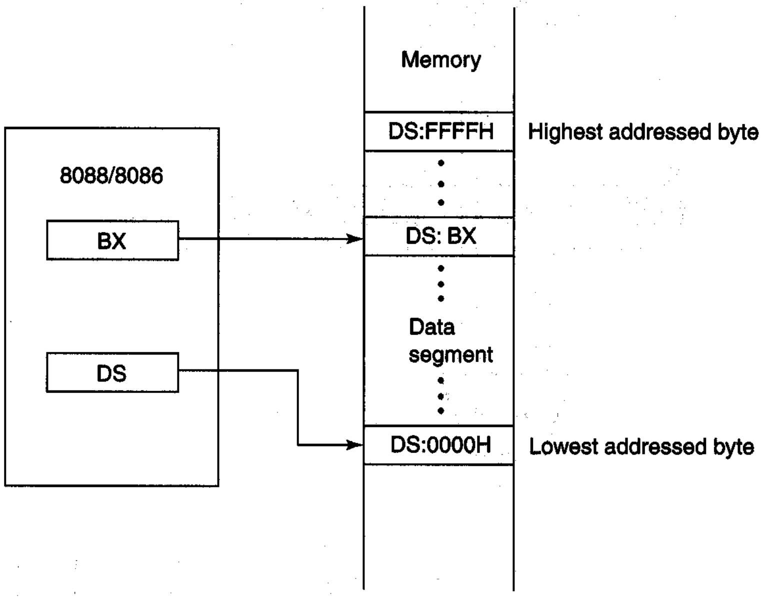
  Segment base address = 1234H
  Offset = 0022H

  1234H = 0001 0010 0011 $0100_2$
  0022H = 0000 0000 0010 $0010_2$

  Shifting base address,
  $0001001000110100\mathbf{0000}_2$ = 12340H

  Adding segment address and offset
  $00010010001101000000_2$ + $0000000000100010_2$ =
  = $00010010001101100010_2$
  = 12362H

# Generating a Real-Mode Memory Address-Boundaries of a Segment



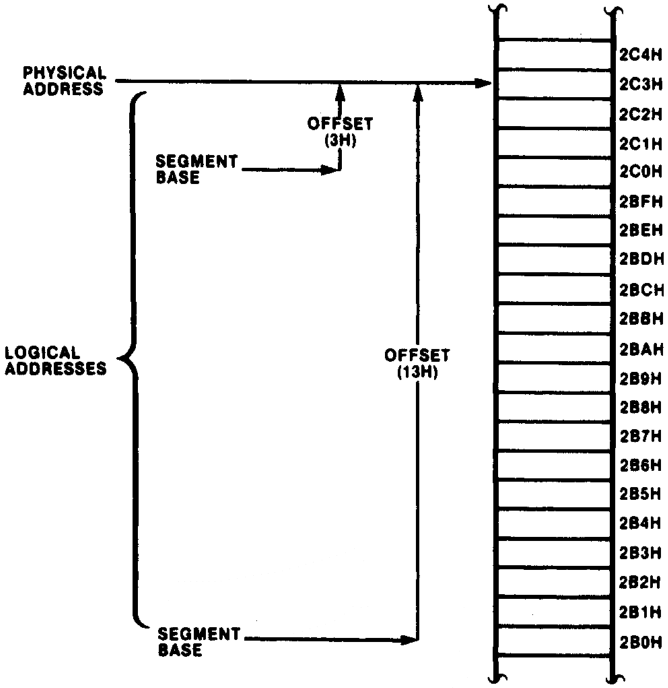
- Four active segments CS, DS, ES, and SS
  - Each 64-k bytes in size → maximum of 256K-bytes of active memory
    - 64K-bytes for code
    - 64K-bytes for stack
    - 128K-bytes for data
- Starting address of a data segment

  DS:0H → lowest addressed byte
- Ending address of a data segment

  DS:FFFFH → highest addressed byte
- Address of an element of data in a data segment

  DS:BX → address of byte, word, or double word element of data in the data segment

# Relationship between Logical and Physical Addresses



- Many different logical addresses map to the same physical address

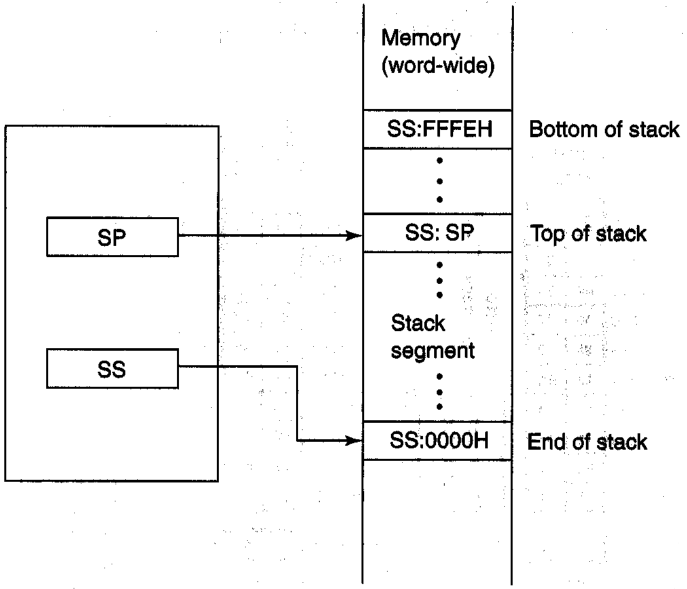
  - Examples:

  2BH:13H = 002B0H+0013H = 002C3H

  2CH:3H = 002C0H + 0003H = 002C3H

  - These logical addresses are called “aliases”

# The Stack

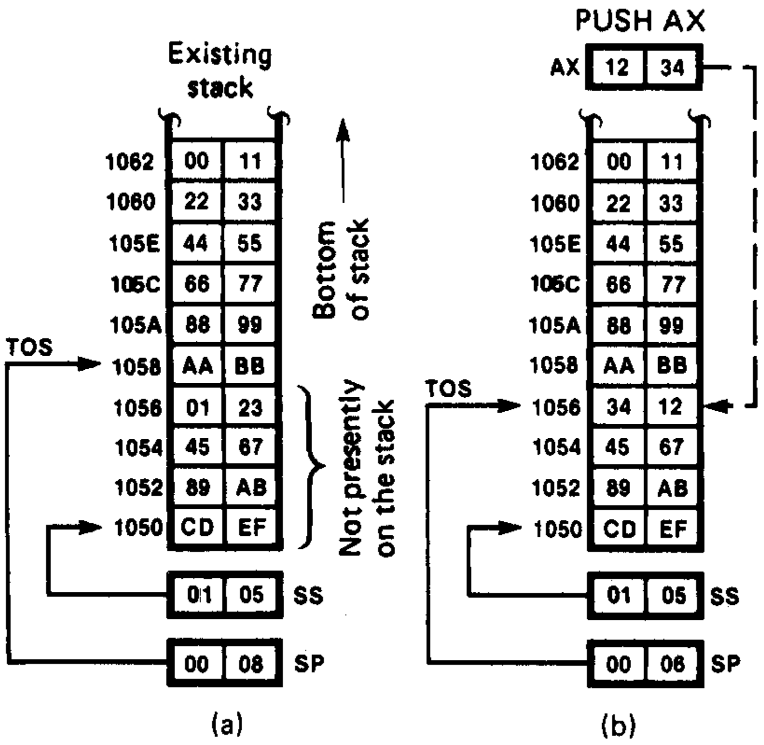


- Stack—temporary storage area for information such as data and addresses
  - Located in stack segment of memory
    - Real mode—64K bytes long
    - Organized as 32k words
    - Information saved as words, not bytes
- Organization of stack
  - SS:0000H→ end of stack (lowest addressed word)
  - SS:FFFEH→ bottom of stack (highest addressed word)
  - SS:SP→ top of stack (last stack location to which data was pushed
  - Stack grows down from higher to lower address
- Used by call, push, pop, and return operations
  - Examples

    PUSH SI → causes the current content of the SI register to be pushed onto the "top of the stack"

    POP SI → causes the value at the "top of the stack" to be popped back into the SI register

# Push Stack Operation



- Status of the stack prior to execution of the instruction
  - PUSH AX
  - AX = 1234H
  - SS = 0105H
  - $A_{EOS}$ = SS:00 → 01050H = end of stack
  - SP = 0008H
  - $A_{BOS}$ = SS:FFFEH → 1104EH
  - $A_{TOS}$ = SS:SP → 01058H = current top of stack
  - BBAAH = Last value pushed to stack
  - Addresses < 01058H = invalid stack data
  - Addresses >= 01058H = valid stack data
- In response to the execution of PUSH AX instruction
  1. SP→ 0006H decremented by 2
     - $A_{TOP}$ → 01056H
  2. Memory write to stack segment
     - AL = 34H → 01056H
     - AH = 12H → 01057H

# Pop Stack Operation

- Status of the stack prior to execution of the instruction POP AX:

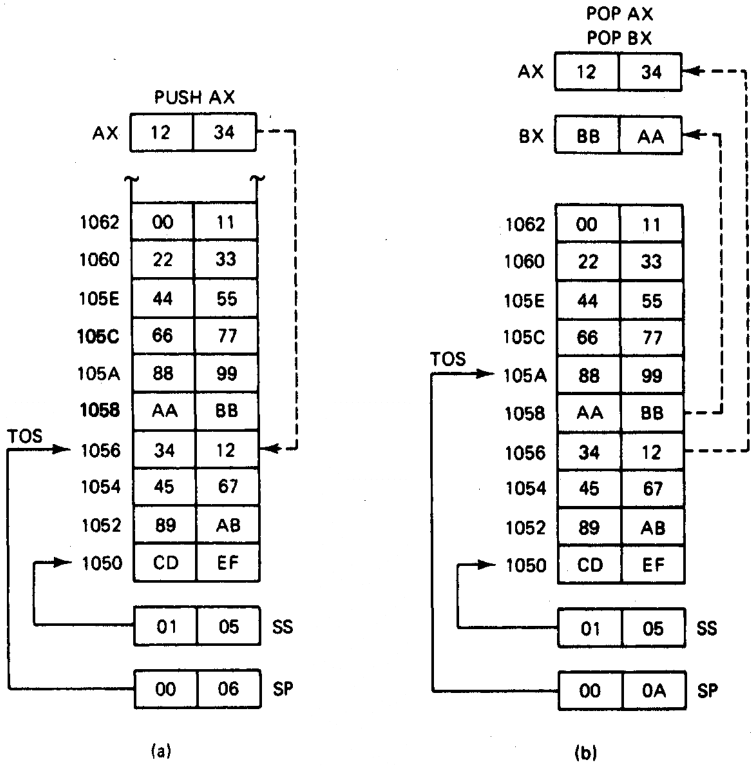
  AX = XXXXH  
  SS = 0105H  
  SP = 0006H  
  $A_{TOS}$ = SS:SP → 01056H = current top of stack  
  1234H = Last value pushed to stack  
  Addresses < 01056H = invalid stack data  
  Addresses >= 01056H = valid stack data

- In response to the execution of POP AX instruction

  1. Memory read to AX  
     01056H = 34H → AL  
     01057H = 12H → AH  
  2. SP→ 0008H incremented by 2  
     $A_{TOP}$ → 01058H

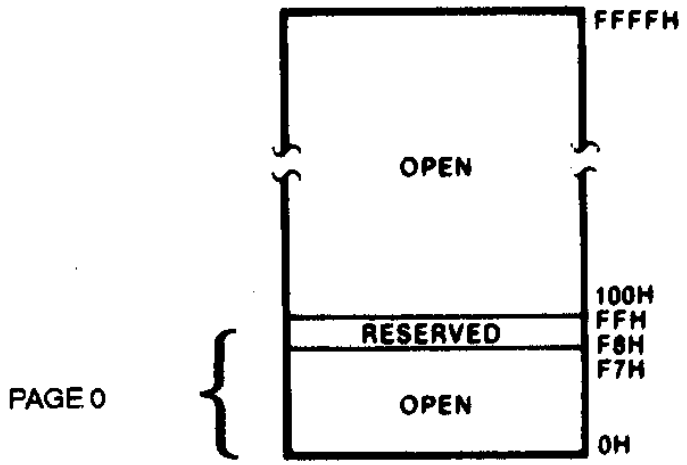
- In response to the execution of POP BX instruction

  1. Memory read to BX  
     01058H = AAH → BL  
     01059H = BBH → BH  
  2. SP→ 000AH incremented by 2: $A_{TOP}$ → 0105AH



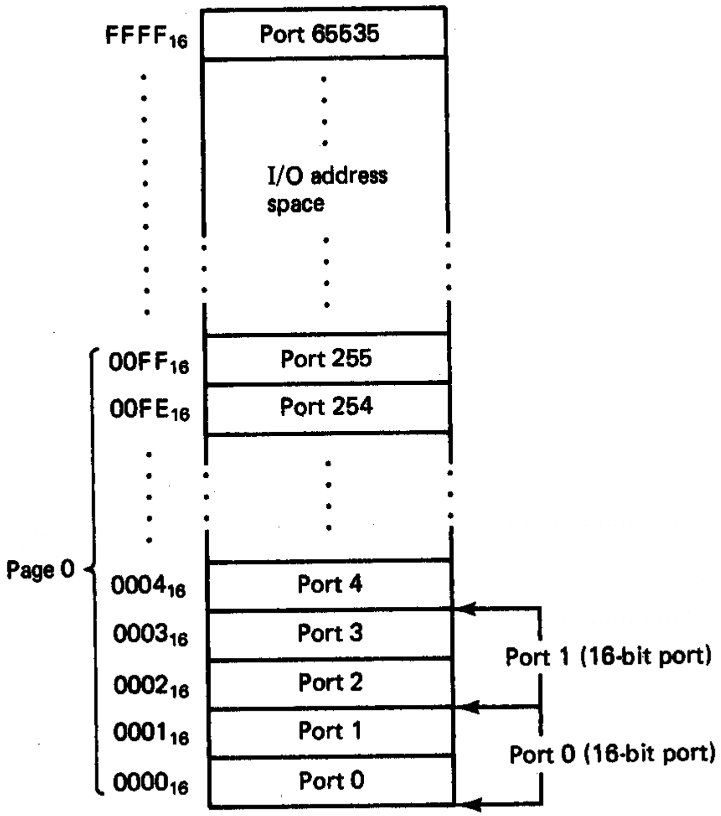
(a) (b)

# Organization of the I/O Address Space



- Input/output address space
  - Place where I/O devices are normally implemented
  - I/O addresses are only 16-bits in length
  - Independent 64K-byte address space
  - Address range 0000H through FFFFH
- Page 0
  - First 256 byte addresses→ 0000H - 00FFH
  - Can be accessed with direct or variable I/O instructions
  - Ports F8H through FF reserved

# Organization of the I/O Data



- Input/output data organization
  - Supports byte or word I/O ports
    - 64K independent byte-wide I/O ports
    - 32K independent aligned word-wide I/O ports
- Examples:

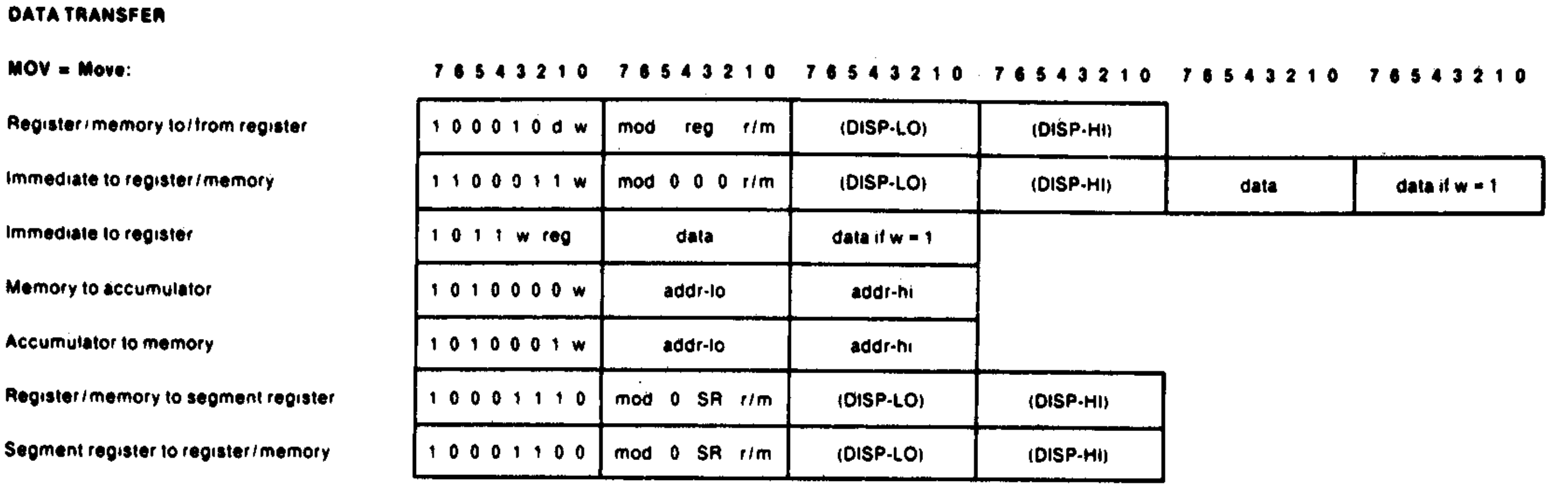
  Byte ports 0,1, 2 → addresses 0000H, 0001H, and 0002H

  Aligned word ports 0,1, 2 → addresses 0000H, 0002H, 0004H
- Advantages of Isolated I/O
  - Complete memory address space available for use by memory devices
  - I/O instructions tailored to maximize performance
- Disadvantage of Isolated I/O
  - All inputs/output must take place between I/O port and accumulator register

# 8088/8086 Instruction Groups and Assembly Notation

- Instructions are organized into groups of functionally related instructions
  - Data Transfer instructions
  - Input/output instructions
  - Arithmetic instructions
  - Logic instructions
  - String Instructions
  - Control transfer instructions
- 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

# 8088/8086 Machine Language



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 | 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 | (DISP-LO) | (DISP-HI) | | |
| Immediate to register/memory | 1 1 0 0 0 1 1 w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) | 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-lo | addr-hi | | | |
| Accumulator to memory | 1 0 1 0 0 0 1 w | addr-lo | addr-hi | | | |
| Register/memory to segment register | 1 0 0 0 1 1 1 0 | mod 0 SR r/m | (DISP-LO) | (DISP-HI) | | |
| Segment register to register/memory | 1 0 0 0 1 1 0 0 | mod 0 SR r/m | (DISP-LO) | (DISP-HI) | | |

- Native language of the 8088/8086 (PC) is “machine language (code)”
  - One to one correspondence to assembly language statements
  - Instructions are encoded with 0’s and 1’s
  - Machine instructions can take up from 1 to 6 bytes
  - Example: Move=MOV
    - The wide choice of register operands, memory operands, and addressing mode available to access operands in memory expands the move instruction to 28 different forms
    - Ranges in size from 3 to 6 bytes

# Structure of an Assembly Language Statement

- General structure of an assembly language statement

  **LABEL: INSTRUCTION ;COMMENT**

  - 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**

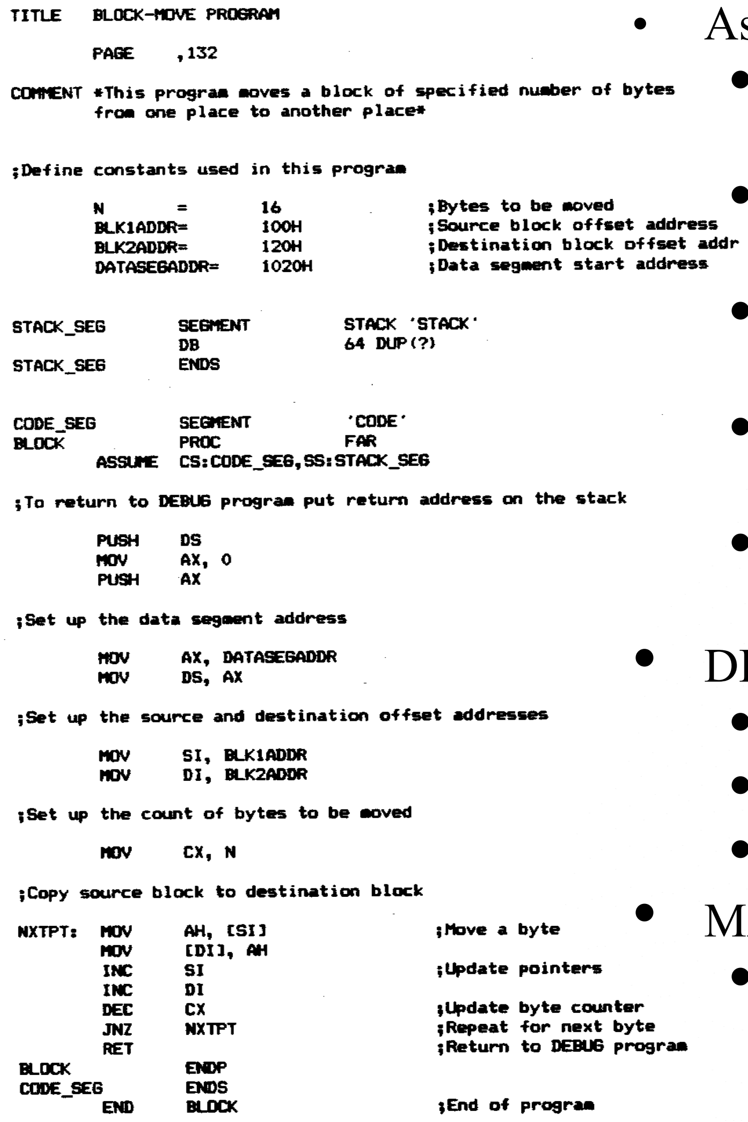
  - Few instructions have a label—usually marks a jump to point
  - Not all instructions need a comment

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?

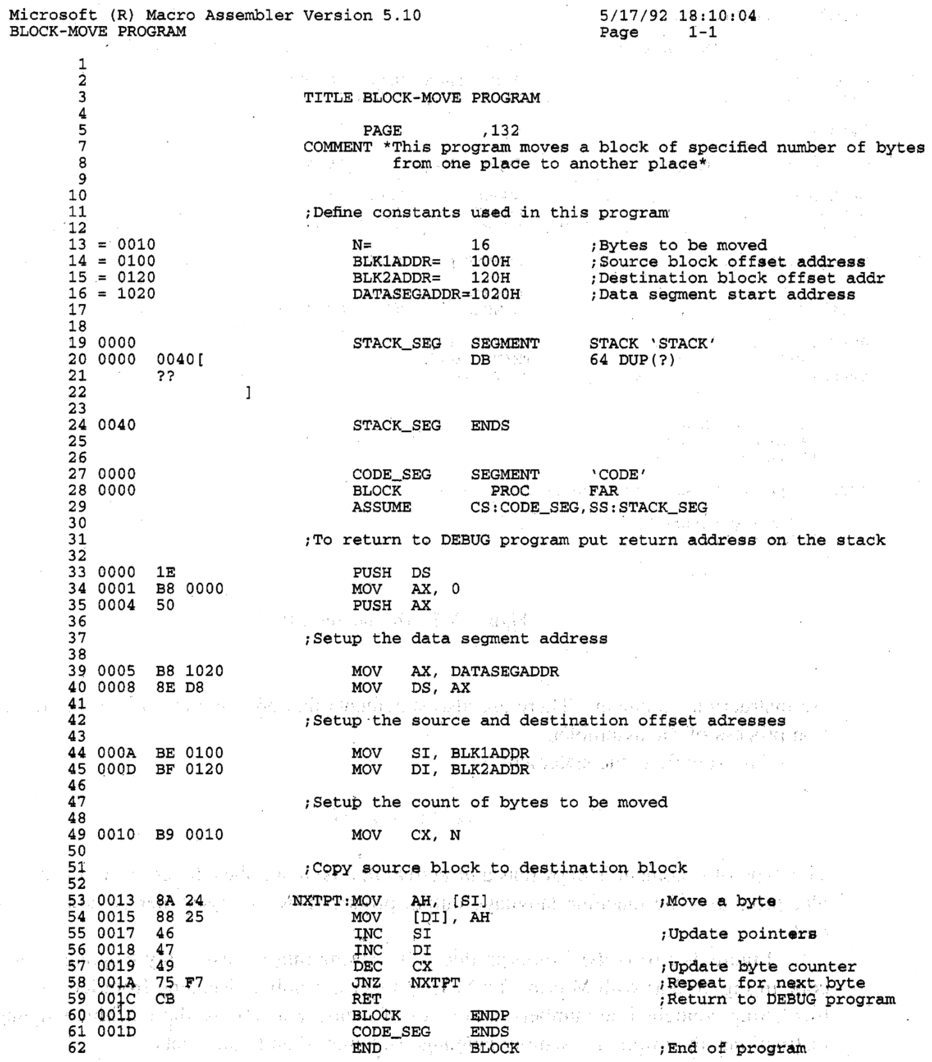
# Assembler and the Source Program



```
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=        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

;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
```

- 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

# Reading the Listing File



```
Microsoft (R) Macro Assembler Version 5.10                5/17/92 18:10:04
BLOCK-MOVE PROGRAM                                        Page    1-1

      1
      2
      3                        TITLE BLOCK-MOVE PROGRAM
      4
      5                              PAGE        ,132
      7                        COMMENT *This program moves a block of specified number of bytes
      8                                 from one place to another place*
      9
     10
     11                        ;Define constants used in this program
     12
     13 = 0010                      N=           16          ;Bytes to be moved
     14 = 0100                      BLK1ADDR=    100H        ;Source block offset address
     15 = 0120                      BLK2ADDR=    120H        ;Destination block offset addr
     16 = 1020                      DATASEGADDR=1020H        ;Data segment start address
     17
     18
     19 0000                        STACK_SEG    SEGMENT     STACK 'STACK'
     20 0000   0040[                             DB          64 DUP(?)
     21         ??
     22                 ]
     23
     24 0040                        STACK_SEG    ENDS
     25
     26
     27 0000                        CODE_SEG     SEGMENT     'CODE'
     28 0000                        BLOCK          PROC      FAR
     29                             ASSUME       CS:CODE_SEG,SS:STACK_SEG
     30
     31                        ;To return to DEBUG program put return address on the stack
     32
     33 0000   1E                   PUSH  DS
     34 0001   B8 0000              MOV   AX, 0
     35 0004   50                   PUSH  AX
     36
     37                        ;Setup the data segment address
     38
     39 0005   B8 1020              MOV   AX, DATASEGADDR
     40 0008   8E D8                MOV   DS, AX
     41
     42                        ;Setup the source and destination offset adresses
     43
     44 000A   BE 0100              MOV   SI, BLK1ADDR
     45 000D   BF 0120              MOV   DI, BLK2ADDR
     46
     47                        ;Setup the count of bytes to be moved
     48
     49 0010   B9 0010              MOV   CX, N
     50
     51                        ;Copy source block to destination block
     52
     53 0013   8A 24           NXTPT:MOV   AH, [SI]                 ;Move a byte
     54 0015   88 25                MOV   [DI], AH
     55 0017   46                   INC   SI                       ;Update pointers
     56 0018   47                   INC   DI
     57 0019   49                   DEC   CX                       ;Update byte counter
     58 001A   75 F7                JNZ   NXTPT                    ;Repeat for next byte
     59 001C   CB                   RET                            ;Return to DEBUG program
     60 001D                        BLOCK        ENDP
     61 001D                        CODE_SEG     ENDS
     62                             END          BLOCK             ;End of program
```

- Instruction statements—operations to be performed by the program
  - Example—line 53

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

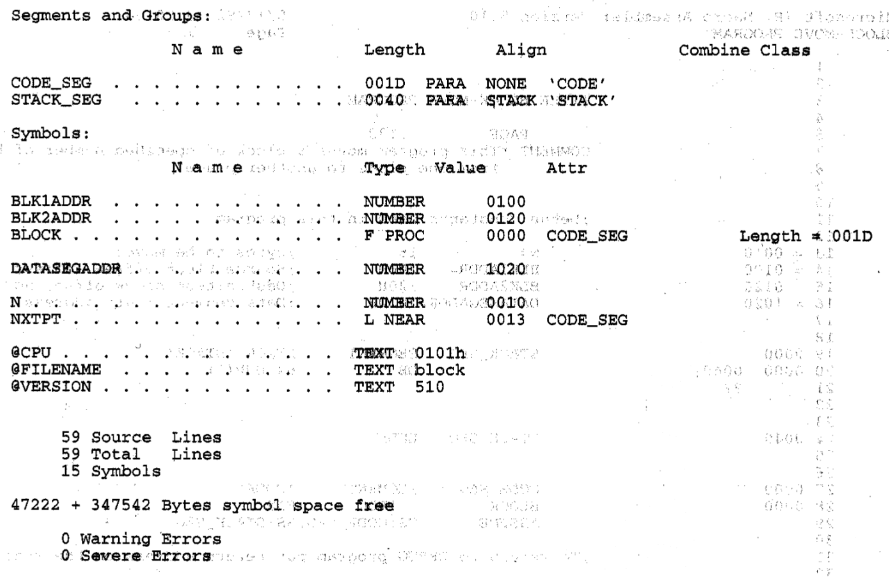
;Move xxxxx = comment

- Directives—provides directions to the assembler program
  - Example—line 20

0000 0040 DB 64 DUP(?)

Defines and leaves un-initialized a block of 64 bytes in memory for use as a stack

# More Information in the Listing



```
Segments and Groups:

                N a m e          Length      Align          Combine Class

CODE_SEG . . . . . . . . . . . . . 001D  PARA  NONE  'CODE'
STACK_SEG  . . . . . . . . . . . . 0040  PARA  STACK 'STACK'

Symbols:

                N a m e          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

47222 + 347542 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

(b)

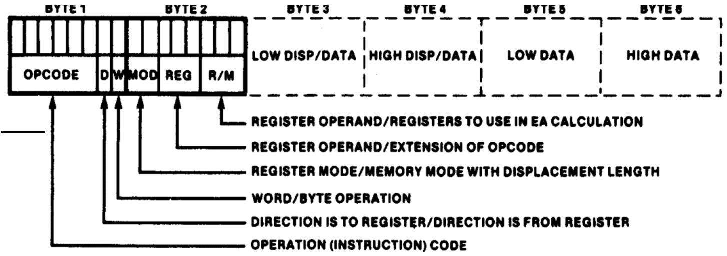
- Other information provided in the listing
    - Size of code segment and stack
        - What is the size of the code segment?
        - At what offset address does it begin? End?

    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?

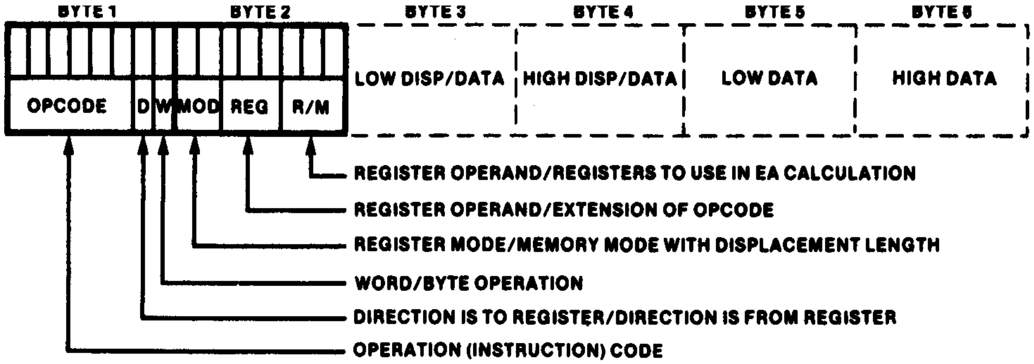
    # lines and symbols used in the program
        - Why is the value of N given as 0010?
    - # errors that occurred during assembly

# Converting Assembly Language to Machine Code

- Part of the 80x86 instruction set architecture (ISA)
  - What is the machine instruction length (fixed, variable, hybrid)?
  - What are the sizes of the fields—varying sizes?
  - What are the functions of the fields?
- 80x86's register-memory architectures is hybrid length
  - Multiple instruction sizes, but all have byte wide lengths—
    - 1 to 6 bytes in length for 8088/8086
    - Up to 17 bytes for 80386, 80486, and Pentium
  - Advantages of hybrid length
    - Allows for many addressing modes
    - Allows full size (32-bit) immediate data and addresses
  - Disadvantage of variable length
    - Requires more complicated decoding hardware—speed of decoding is critical in modern uP
- Load-store architectures normally fixed length—PowerPC (32-bit), SPARC (32-bit), MIP (32-bit), Itanium (128-bits, 3 instructions)

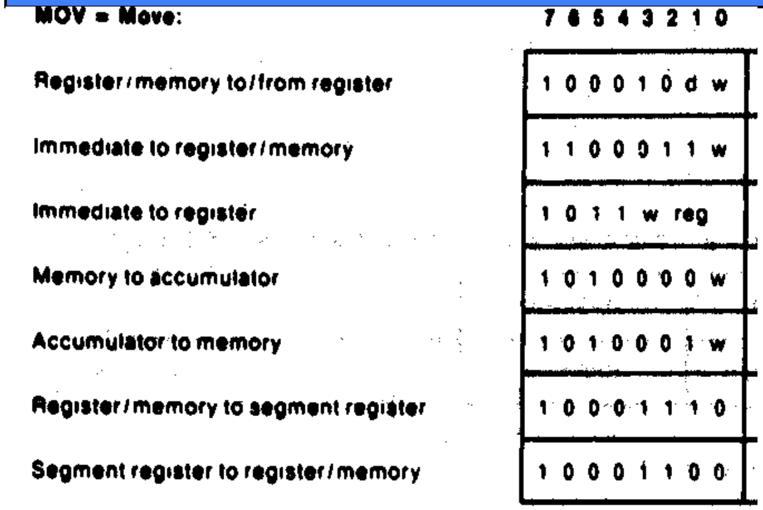

BYTE 1
BYTE 2
BYTE 3
BYTE 4
BYTE 5
BYTE 6
OPCODE
D
W
MOD
REG
R/M
LOW DISP/DATA
HIGH DISP/DATA
LOW DATA
HIGH DATA
REGISTER OPERAND/REGISTERS TO USE IN EA CALCULATION
REGISTER OPERAND/EXTENSION OF OPCODE
REGISTER MODE/MEMORY MODE WITH DISPLACEMENT LENGTH
WORD/BYTE OPERATION
DIRECTION IS TO REGISTER/DIRECTION IS FROM REGISTER
OPERATION (INSTRUCTION) CODE

# General Instruction Format

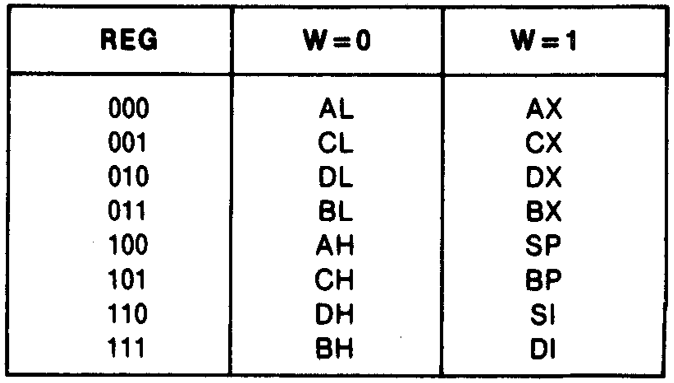

BYTE 1
BYTE 2
BYTE 3
BYTE 4
BYTE 5
BYTE 6
OPCODE
D
W
MOD
REG
R/M
LOW DISP/DATA
HIGH DISP/DATA
LOW DATA
HIGH DATA
REGISTER OPERAND/REGISTERS TO USE IN EA CALCULATION
REGISTER OPERAND/EXTENSION OF OPCODE
REGISTER MODE/MEMORY MODE WITH DISPLACEMENT LENGTH
WORD/BYTE OPERATION
DIRECTION IS TO REGISTER/DIRECTION IS FROM REGISTER
OPERATION (INSTRUCTION) CODE

- Information that must be coded into the instruction
  - Operation code--opcode
  - Source(s) and destination registers
  - Size of data-W
  - Addressing mode for the source or destination
  - Registers used in address computation
  - Immediate address displacement: **How many bytes?**
  - Immediate data: **How many bytes?**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



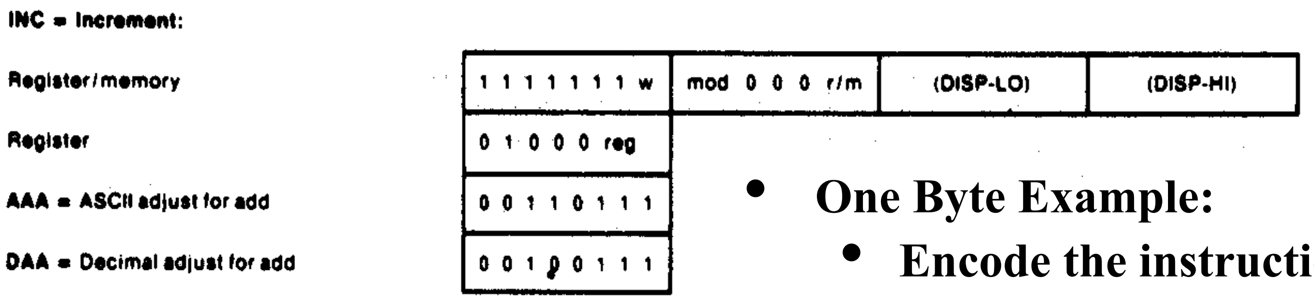
| MOV = Move: | 7 6 5 4 3 2 1 0 |
|---|---|
| Register/memory to/from register | 1 0 0 0 1 0 d w |
| Immediate to register/memory | 1 1 0 0 0 1 1 w |
| Immediate to register | 1 0 1 1 w reg |
| Memory to accumulator | 1 0 1 0 0 0 0 w |
| Accumulator to memory | 1 0 1 0 0 0 1 w |
| Register/memory to segment register | 1 0 0 0 1 1 1 0 |
| Segment register to register/memory | 1 0 0 0 1 1 0 0 |



| REG | W=0 | W=1 |
|---|---|---|
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

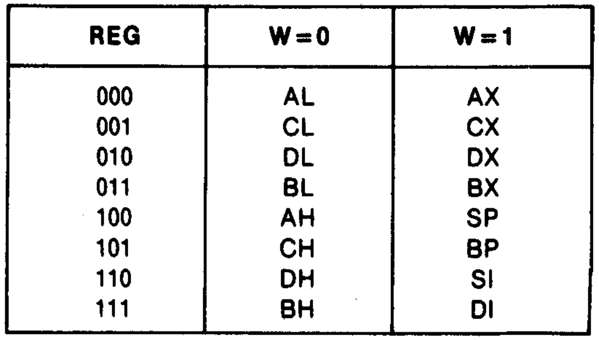
- **Byte 1 information:**
  - **Opcode field (6-bits)—specifies the operation to be performed by the instruction**
    - **Move immediate to registers/memory = 1100011**
    - **Move memory to accumulator = 1010000**
    - **Move segment register to register/memory = 10001100**
  - **REG (3-bit)—selects a first operand as a register**
    - **Move immediate to register = 1011(w)(reg)—only requires one register which is the destination**
      - **Accumulator register= 000**
      - **Count register = 001**
      - **Data Register = 010**
  - **W (1-bit)—data size word/byte for all registers**
    - **Byte = 0**
    - **Word =1**
  - **D (1-bit)—register direction: tells whether the register which is selected by the REG field in the second byte is the source or destination**
    - **Add register to register = 000000(d)(w)**
    - **D = 0 → source operand**
    - **D= 1 → destination operand**

# Binary Instruction Format: Example



**INC = Increment:**

| | | | | |
|---|---|---|---|---|
| Register/memory | 1 1 1 1 1 1 1 w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) |
| Register | 0 1 0 0 0 reg | | | |
| AAA = ASCII adjust for add | 0 0 1 1 0 1 1 1 | | | |
| DAA = Decimal adjust for add | 0 0 1 0 0 1 1 1 | | | |



| REG | W=0 | W=1 |
|---|---|---|
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

- **One Byte Example:**
  - **Encode the instruction in machine code**

    **`INC CX`**
  - **Solution:**
    - **Use "INC register" instruction format—special short form for 16-bit register**

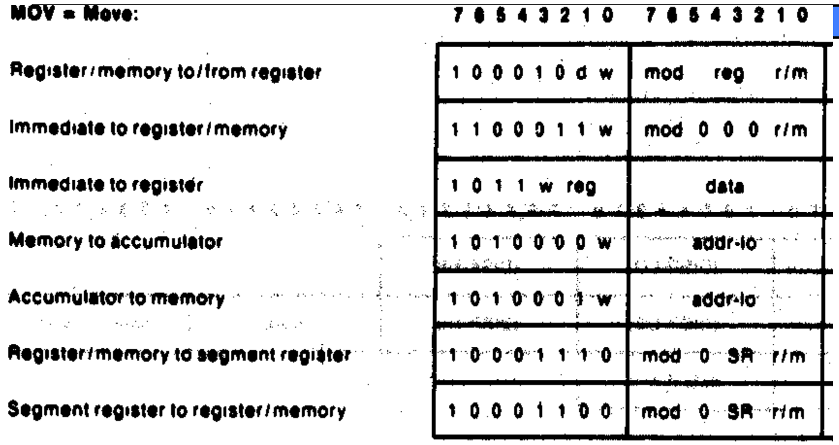
      **01000 (REG)**
    - **CX is destination register**

      **CX = 001**
    - **Machine code is**

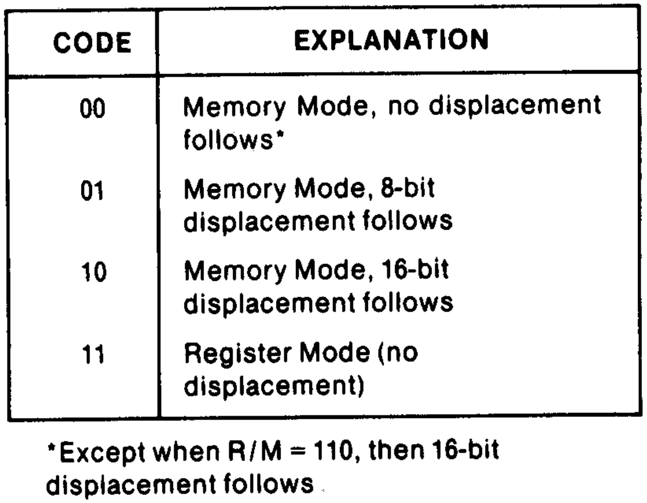
      **01000 (001) = 01000001 = 41H → one byte instruction**

      **`INC CX` = 41H**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



| MOV = Move: | 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 |
| Immediate to register | 1 0 1 1 w reg | data |
| Memory to accumulator | 1 0 1 0 0 0 0 w | addr-lo |
| Accumulator to memory | 1 0 1 0 0 0 1 w | addr-lo |
| Register/memory to segment register | 1 0 0 0 1 1 1 0 | mod 0 SR r/m |
| Segment register to register/memory | 1 0 0 0 1 1 0 0 | mod 0 SR r/m |

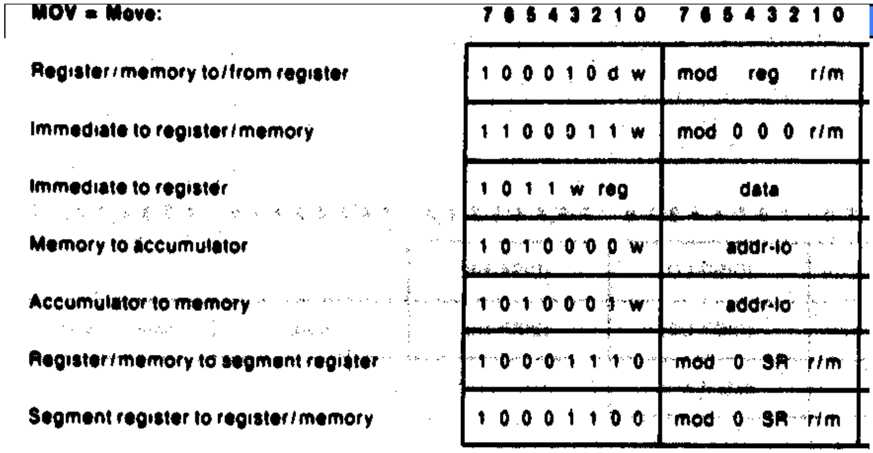


| CODE | EXPLANATION |
|---|---|
| 00 | Memory Mode, no displacement follows* |
| 01 | Memory Mode, 8-bit displacement follows |
| 10 | Memory Mode, 16-bit displacement follows |
| 11 | Register Mode (no displacement) |

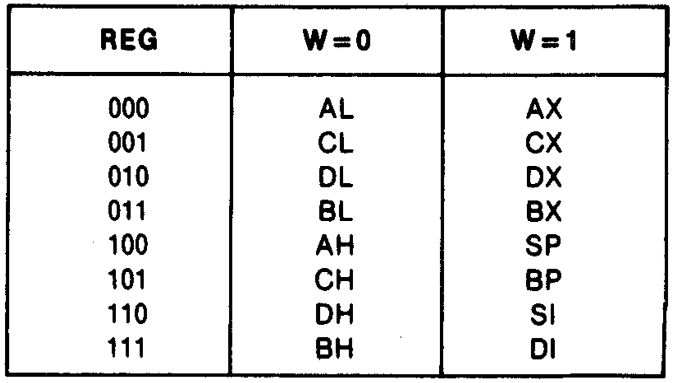
*Except when R/M = 110, then 16-bit displacement follows

- **Byte 2 information:**
  - **MOD (2-bit mode field)—specifies the type of the second operand**
    - **Memory mode: 00, 01,10—Register to memory move operation**
      - **00 = no immediate displacement (register used for addressing)**
      - **01 = 8-bit displacement (imm8) follows (8-bit offset address)**
      - **10 = 16-bit displacement (imm16) follows (16-bit offset address)**
    - **Register mode: 11—register to register move operation**
      - **11 = register specified as the second operand**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



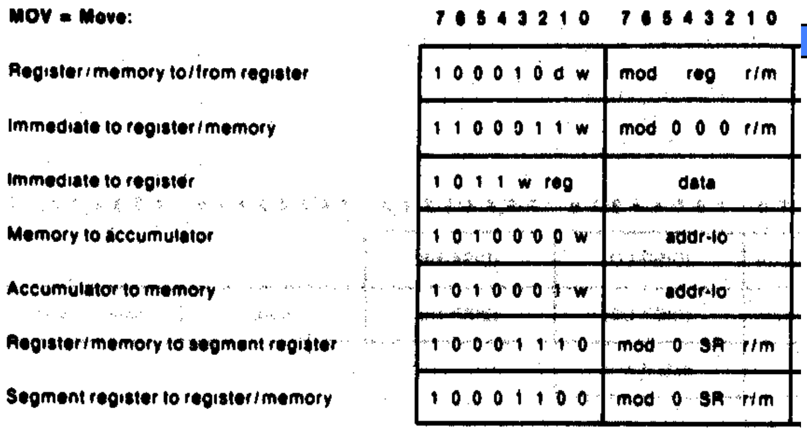
| MOV = Move: | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|
| Register/memory to/from register | 1 0 0 0 1 0 d w | mod reg r/m |
| Immediate to register/memory | 1 1 0 0 0 1 1 w | mod 0 0 0 r/m |
| Immediate to register | 1 0 1 1 w reg | data |
| Memory to accumulator | 1 0 1 0 0 0 0 w | addr-lo |
| Accumulator to memory | 1 0 1 0 0 0 1 w | addr-lo |
| Register/memory to segment register | 1 0 0 0 1 1 1 0 | mod 0 SR r/m |
| Segment register to register/memory | 1 0 0 0 1 1 0 0 | mod 0 SR r/m |



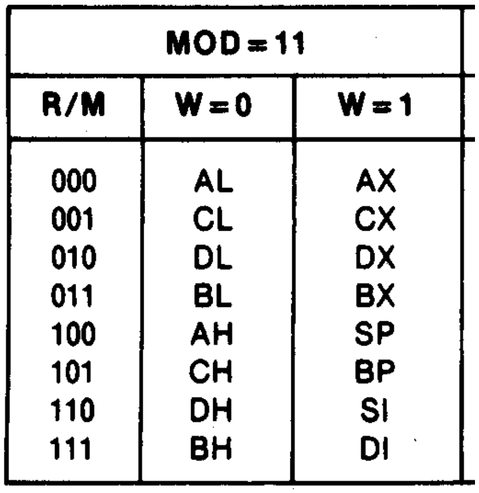
| REG | W=0 | W=1 |
|---|---|---|
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

- **Byte 2 information (continued):**
  - **REG (3-bit register field)—selects the register for a first operand, which may be the source or destination**
    - **Accumulator register= 000**
    - **Count register = 001**
    - **Data Register = 010**
    - **Move register/memory to/from register**
      - **Byte 1= 100010(d)(w)**
      - **Byte 2 = (mod) (reg) (r/m)**
- **Affected by byte 1 information:**
  - **W (1-bit)—data size word/byte for all registers**
    - **Byte = 0**
    - **Word =1**

## 3.3 Converting Assembly Language to Machine Code- General Instruction Format



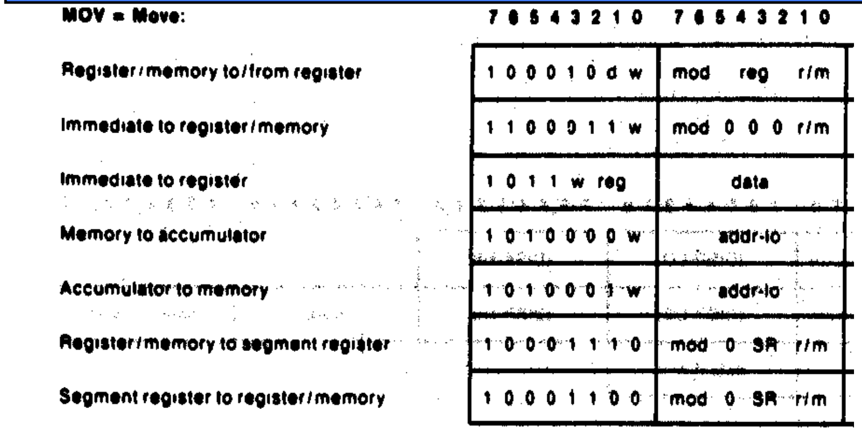
| MOV = Move: | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|
| Register/memory to/from register | 1 0 0 0 1 0 d w | mod reg r/m |
| Immediate to register/memory | 1 1 0 0 0 1 1 w | mod 0 0 0 r/m |
| Immediate to register | 1 0 1 1 w reg | data |
| Memory to accumulator | 1 0 1 0 0 0 0 w | addr-lo |
| Accumulator to memory | 1 0 1 0 0 0 1 w | addr-lo |
| Register/memory to segment register | 1 0 0 0 1 1 1 0 | mod 0 SR r/m |
| Segment register to register/memory | 1 0 0 0 1 1 0 0 | mod 0 SR r/m |



| MOD = 11 | | |
|---|---|---|
| R/M | W = 0 | W = 1 |
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

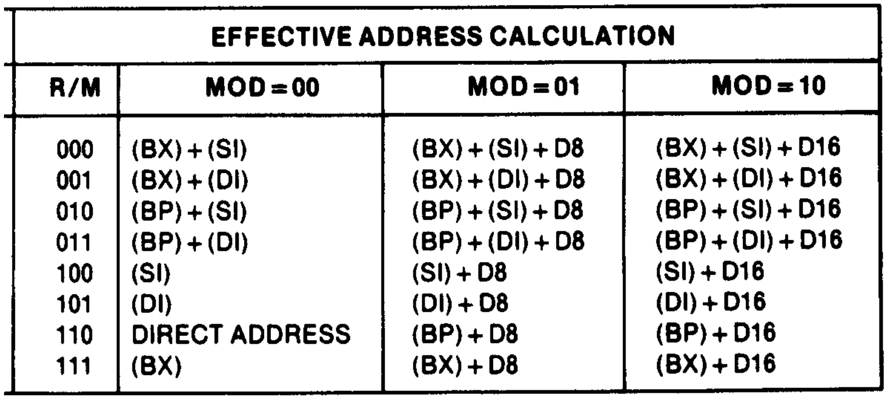
- **Byte 2 information (continued):**
  - **R/M (3-bit register/memory field)—specifies the second operand as a register or a storage location in memory**
    - **Dependent on MOD field**
      - **Mod = 11 R/M selects a register**
        - **R/M = 000 Accumulator register**
        - **R/M= 001 = Count register**
        - **R/M = 010 = Data Register**
    - **Move register/memory to/from register**
      - **Byte 1= 100010(d)(w)**
      - **Byte 2 = (mod) (reg) (r/m)**
- **Affected by byte 1 information:**
  - **W (1-bit)—data size word/byte for all registers**
    - **Byte = 0**
    - **Word =1**
  - **D (1-bit)—register direction for first operand in byte 2 (reg)**
    - **D = 0 → source operand**
    - **D= 1 → destination operand**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



MOV = Move:

| | 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 |
| Immediate to register | 1 0 1 1 w reg | data |
| Memory to accumulator | 1 0 1 0 0 0 0 w | addr-lo |
| Accumulator to memory | 1 0 1 0 0 0 1 w | addr-lo |
| Register/memory to segment register | 1 0 0 0 1 1 1 0 | mod 0 SR r/m |
| Segment register to register/memory | 1 0 0 0 1 1 0 0 | mod 0 SR r/m |



EFFECTIVE ADDRESS CALCULATION

| R/M | MOD = 00 | MOD = 01 | MOD = 10 |
|---|---|---|---|
| 000 | (BX) + (SI) | (BX) + (SI) + D8 | (BX) + (SI) + D16 |
| 001 | (BX) + (DI) | (BX) + (DI) + D8 | (BX) + (DI) + D16 |
| 010 | (BP) + (SI) | (BP) + (SI) + D8 | (BP) + (SI) + D16 |
| 011 | (BP) + (DI) | (BP) + (DI) + D8 | (BP) + (DI) + D16 |
| 100 | (SI) | (SI) + D8 | (SI) + D16 |
| 101 | (DI) | (DI) + D8 | (DI) + D16 |
| 110 | DIRECT ADDRESS | (BP) + D8 | (BP) + D16 |
| 111 | (BX) | (BX) + D8 | (BX) + D16 |

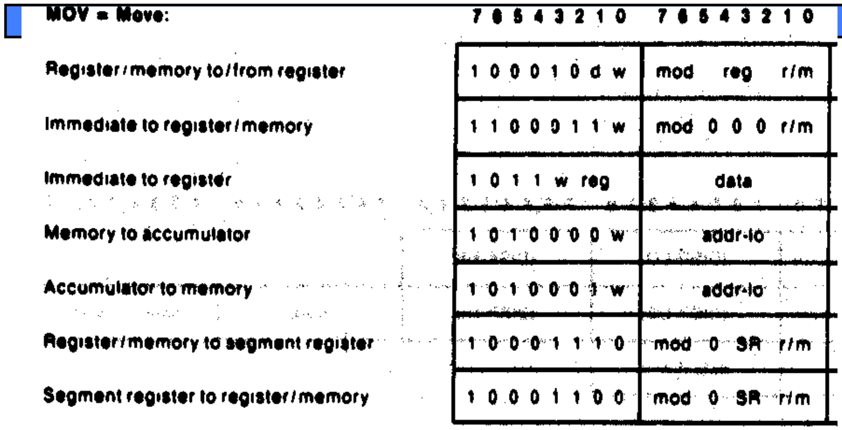
- **Byte 2 information (continued):**
  - **MOD = 00,10, or 10 selects an addressing mode for the second operand that is a storage location in memory, which may be the source or destination**
    - **Dependent on MOD field**
      - **Mod = 00 R/M**
        - **R/M = 100 → effective address computed as**

          **EA = (SI)**
        - **R/M= 000 = → effective address computed as**

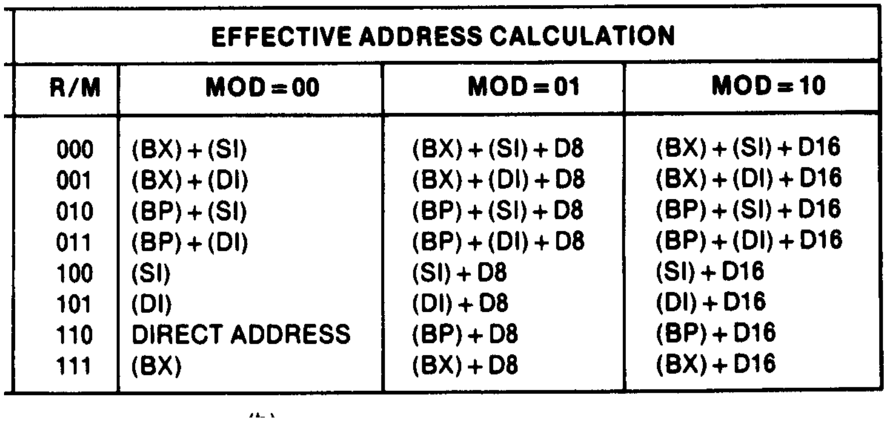
          **EA = (BX)+(SI)**
        - **R/M = 110 = → effective address is coded in the instruction as a direct address**

        **EA = direct address = imm8 or imm16**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



| MOV = Move: | 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 |
| Immediate to register | 1 0 1 1 w reg | data |
| Memory to accumulator | 1 0 1 0 0 0 0 w | addr-lo |
| Accumulator to memory | 1 0 1 0 0 0 1 w | addr-lo |
| Register/memory to segment register | 1 0 0 0 1 1 1 0 | mod 0 SR r/m |
| Segment register to register/memory | 1 0 0 0 1 1 0 0 | mod 0 SR r/m |

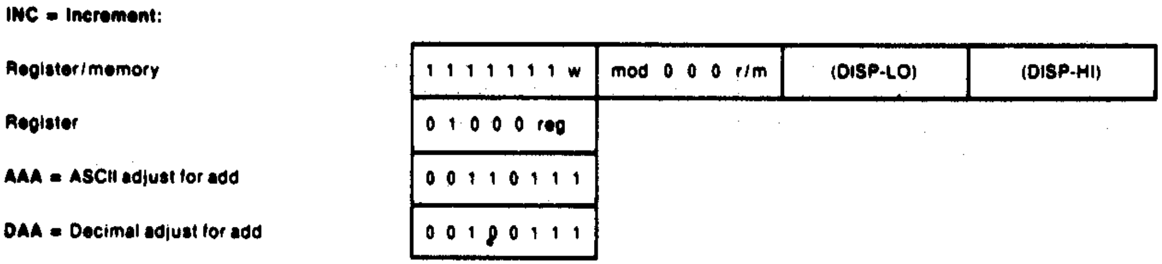


EFFECTIVE ADDRESS CALCULATION

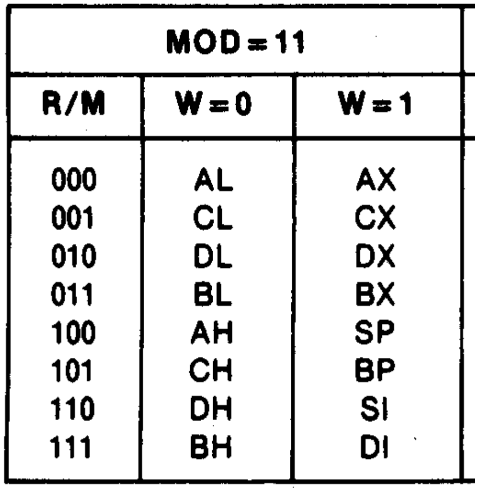
| R/M | MOD=00 | MOD=01 | MOD=10 |
|---|---|---|---|
| 000 | (BX)+(SI) | (BX)+(SI)+D8 | (BX)+(SI)+D16 |
| 001 | (BX)+(DI) | (BX)+(DI)+D8 | (BX)+(DI)+D16 |
| 010 | (BP)+(SI) | (BP)+(SI)+D8 | (BP)+(SI)+D16 |
| 011 | (BP)+(DI) | (BP)+(DI)+D8 | (BP)+(DI)+D16 |
| 100 | (SI) | (SI)+D8 | (SI)+D16 |
| 101 | (DI) | (DI)+D8 | (DI)+D16 |
| 110 | DIRECT ADDRESS | (BP)+D8 | (BP)+D16 |
| 111 | (BX) | (BX)+D8 | (BX)+D16 |

- **<u>Move register/memory to/from register</u>**
  - **Byte 1= 100010(d)(w)**
  - **Byte 2 = (mod) (reg) (r/m)**
- **Affected of byte 1 information:**
  - **W (1-bit)—data size word/byte for all registers**
    - **Byte = 0**
    - **Word =1**
  - **D (1-bit)—register direction for first operand in byte 2 (reg)**
    - **D = 0 → source operand**
    - **D= 1 → destination operand**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



| | | | | |
|---|---|---|---|---|
| **INC = Increment:** | | | | |
| Register/memory | 1 1 1 1 1 1 1 w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) |
| Register | 0 1 0 0 0 reg | | | |
| AAA = ASCII adjust for add | 0 0 1 1 0 1 1 1 | | | |
| DAA = Decimal adjust for add | 0 0 1 0 0 1 1 1 | | | |



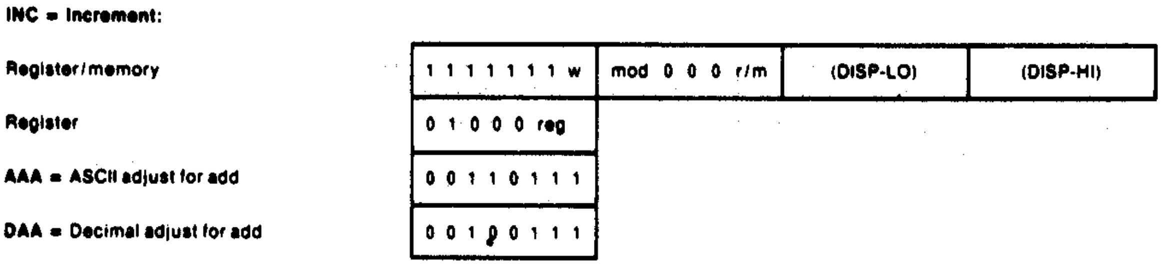
| MOD = 11 | | |
|---|---|---|
| R/M | W = 0 | W = 1 |
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

- **Two Byte example using R/M field for a register:**
  - **Encode the instruction in machine code**

    **`INC CL`**
  - **Solution:**
    - **Use "INC register/memory" instruction format—general form for 8-bit or 16-bit register/memory**
    - **Byte 1**

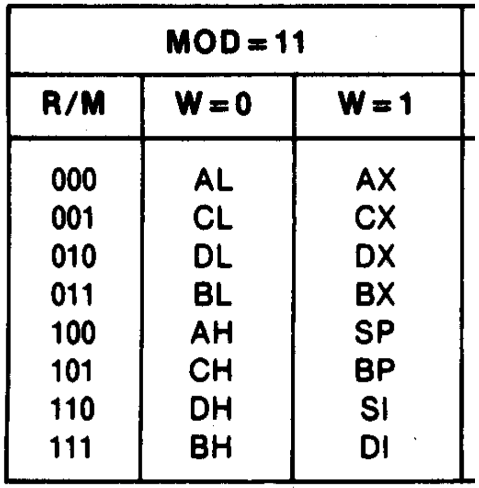
      **1111111(W)**
      - **CL= byte wide register → W = 0**
        **11111110 =FEH**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



**INC = Increment:**

|  |  |  |  |  |
|---|---|---|---|---|
| Register/memory | 1 1 1 1 1 1 1 w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) |
| Register | 0 1 0 0 0 reg |  |  |  |
| AAA = ASCII adjust for add | 0 0 1 1 0 1 1 1 |  |  |  |
| DAA = Decimal adjust for add | 0 0 1 0 0 1 1 1 |  |  |  |



| MOD = 11 |  |  |
|---|---|---|
| R/M | W = 0 | W = 1 |
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

- **Two Byte example using R/M field for a register (continued):**
  - **Byte 2**

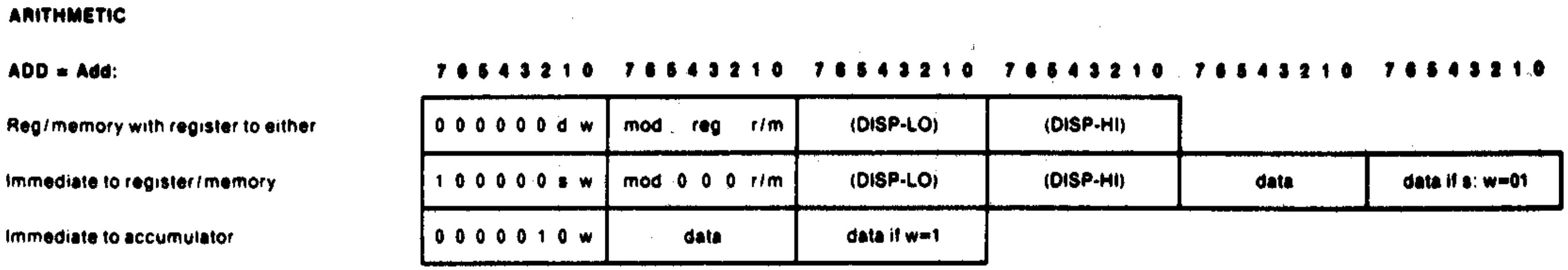
    **(MOD) 000(R/M)**
    - **Destination is register register CL**
      - **MOD = 11**
      - **R/M = 001**

      **(11)000(001) = 11000001 =C1H**
  - **Machine code is**

    **(Byte 1)(Byte 2) = 11111110 11000001 = FEC1H → two byte instruction**

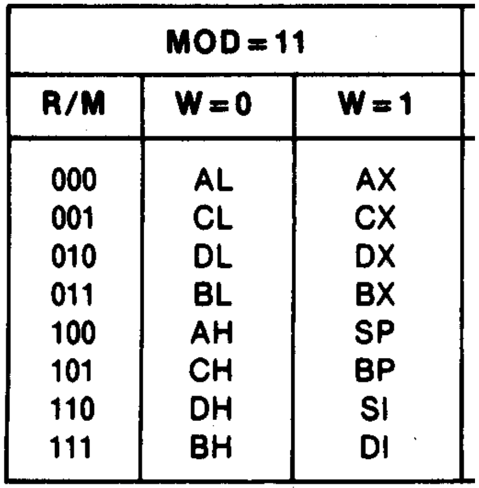
    **`INC CL` = FEC1H**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



ARITHMETIC

| ADD = Add: | 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 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| Reg/memory with register to either | 0 0 0 0 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate to register/memory | 1 0 0 0 0 0 s w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) | data | data if s: w=01 |
| Immediate to accumulator | 0 0 0 0 0 1 0 w | data | data if w=1 | | | |



| MOD = 11 | | |
|---|---|---|
| R/M | W = 0 | W = 1 |
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

- **Two Byte example using R/M field for a register:**
  - **Encode the instruction in machine code**

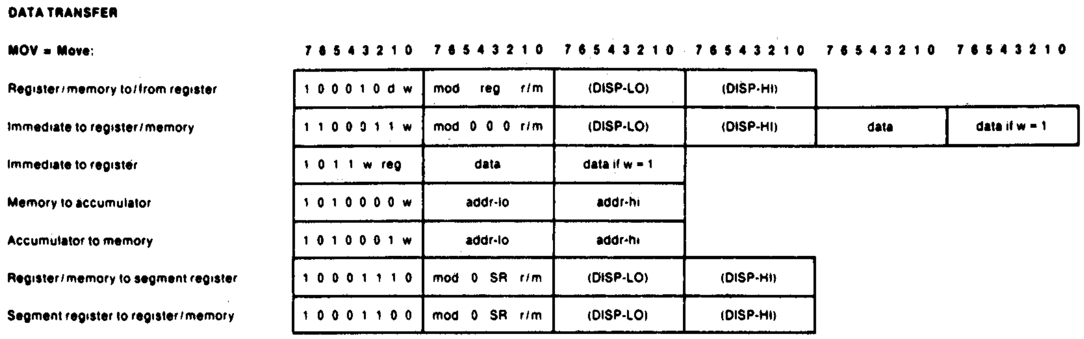
    **MOV BL,AL**

  - **Solution:**
    - **Use "register/memory to/from register" instruction format—most general form of move instruction**
    - **Byte 1**

      **100010(D)(W)**

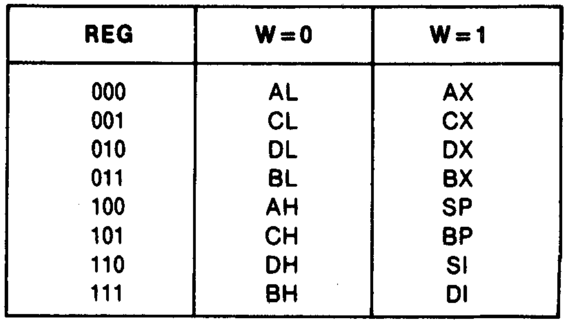
      - **Assuming AL (source operand) is the register encoded in the REG field of byte 2 (1st register)**
        - **D = 0 = source**
      - **Both registers are byte wide**
        - **W = 0 = byte wide**
      - **Byte 1 = 100010(0)(0) = 10001000 =88H**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format

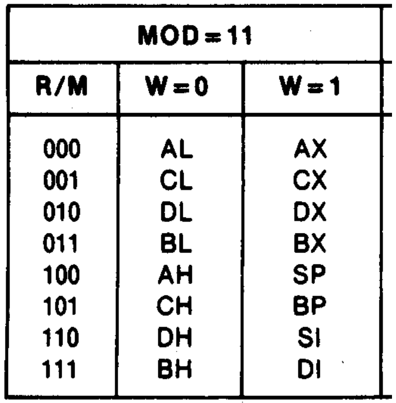


DATA TRANSFER

| 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 | 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 | (DISP-LO) | (DISP-HI) | | |
| Immediate to register/memory | 1 1 0 0 0 1 1 w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) | 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-lo | addr-hi | | | |
| Accumulator to memory | 1 0 1 0 0 0 1 w | addr-lo | addr-hi | | | |
| Register/memory to segment register | 1 0 0 0 1 1 1 0 | mod 0 SR r/m | (DISP-LO) | (DISP-HI) | | |
| Segment register to register/memory | 1 0 0 0 1 1 0 0 | mod 0 SR r/m | (DISP-LO) | (DISP-HI) | | |



| REG | W=0 | W=1 |
|---|---|---|
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |



| MOD=11 | | |
|---|---|---|
| R/M | W=0 | W=1 |
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

- **Two Byte Example (continued):**
  - **Byte 2**

    **(MOD)(REG)(R/M)**

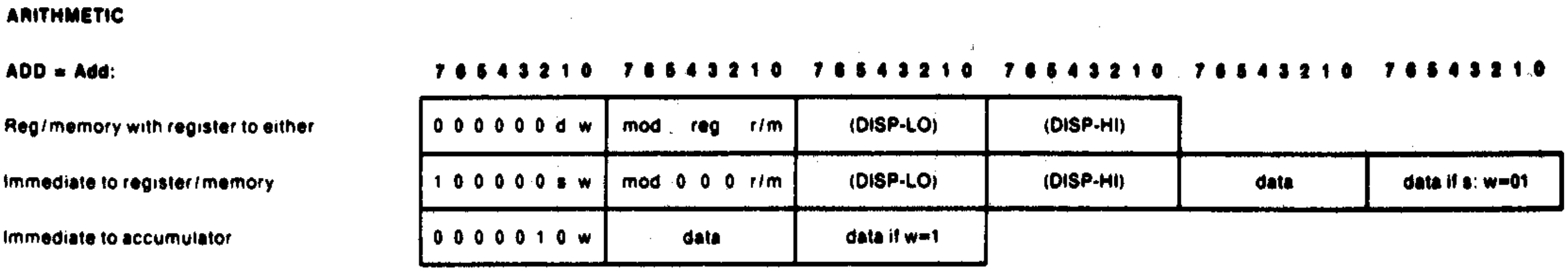
    - **Both operands are registers**
      - **MOD = 11**
    - **2nd register is destination register BL**
      - **R/M = 011**
    - **1st register is source register AL**
      - **REG = 000**

      **(11)000(011) = 11000011 = C3H**
  - **Machine code is**

    **(Byte 1)(Byte 2) = 10001000 11000011 = 88C3H → two byte instruction**

    **`MOV BL,AL` = 88C3H**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



**ARITHMETIC**

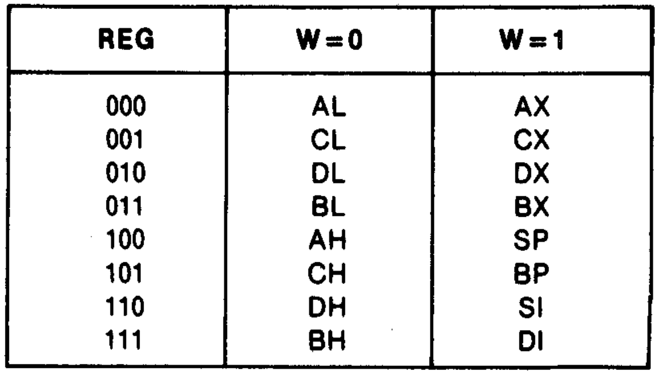
| ADD = Add: | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| Reg/memory with register to either | 0 0 0 0 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate to register/memory | 1 0 0 0 0 0 s w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) | data | data if s: w=01 |
| Immediate to accumulator | 0 0 0 0 0 1 0 w | data | data if w=1 | | | |

- **Two Byte example using R/M field for memory:**
  - **Encode the instruction in machine code**

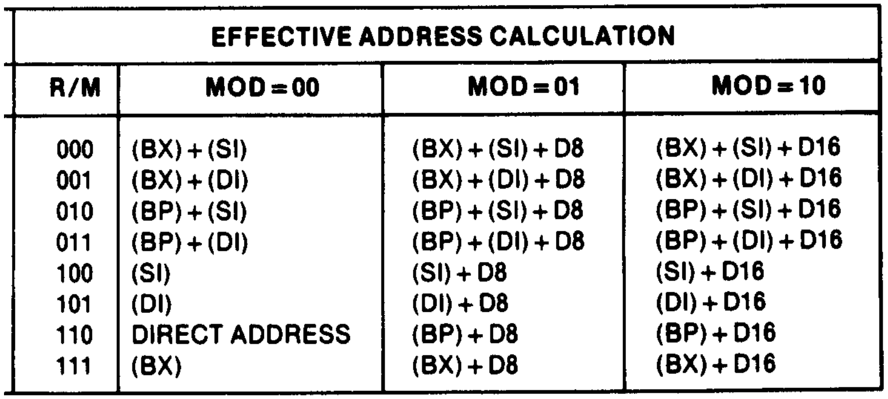
    **`ADD AX,[SI]`**
  - **Solution:**
    - **Use “register/memory with register to either” instruction format**
      - **Most general form of add instruction**
      - **No displacement needed—register indirect addressing**
    - **Byte 1**

      **000000(D)(W)**
      - **AX (destination operand) is the register encoded in the REG field of byte 2 (1st register)**
        - **D = 1 = destination**
      - **Addition is of word wide data**
        - **W = 1 = word wide**
      - **Byte 1 = 000000(1)(1) = 00000011 =03H**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



| REG | W=0 | W=1 |
|---|---|---|
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |



EFFECTIVE ADDRESS CALCULATION

| R/M | MOD=00 | MOD=01 | MOD=10 |
|---|---|---|---|
| 000 | (BX)+(SI) | (BX)+(SI)+D8 | (BX)+(SI)+D16 |
| 001 | (BX)+(DI) | (BX)+(DI)+D8 | (BX)+(DI)+D16 |
| 010 | (BP)+(SI) | (BP)+(SI)+D8 | (BP)+(SI)+D16 |
| 011 | (BP)+(DI) | (BP)+(DI)+D8 | (BP)+(DI)+D16 |
| 100 | (SI) | (SI)+D8 | (SI)+D16 |
| 101 | (DI) | (DI)+D8 | (DI)+D16 |
| 110 | DIRECT ADDRESS | (BP)+D8 | (BP)+D16 |
| 111 | (BX) | (BX)+D8 | (BX)+D16 |

- **Two Byte example using R/M field for memory (continued):**
  - **Byte 2**

    **(MOD)(REG)(R/M)**
    - **Second operand is in memory and pointed to by address is SI**
      - **MOD = 00 → [SI]**
    - **R/M specifies the addressing mode**
      - **R/M = 100 → [SI]**
    - **1st register is destination register AX**
      - **REG = 000**

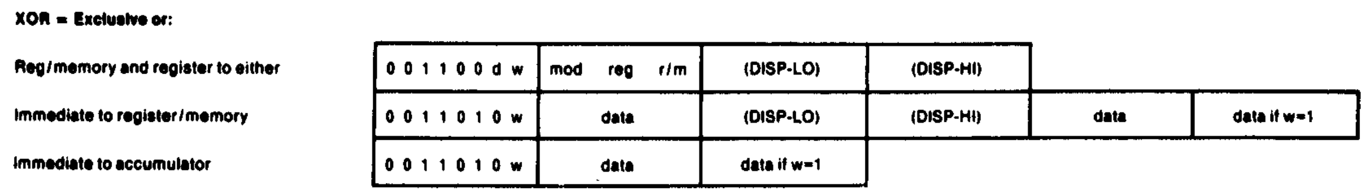
      **(00)000(100) = 00000100 = 04H**
  - **Machine code is**

    **(Byte 1)(Byte 2) = 00000011 00000100**

    **= 0304H → two byte instruction**

    **`ADD AX,[SI] = 0304H`**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



XOR = Exclusive or:

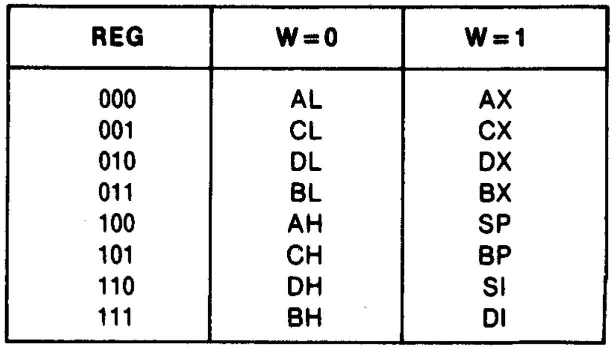
| | | | | | | |
|---|---|---|---|---|---|---|
| Reg/memory and register to either | 0 0 1 1 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate to register/memory | 0 0 1 1 0 1 0 w | data | (DISP-LO) | (DISP-HI) | data | data if w=1 |
| Immediate to accumulator | 0 0 1 1 0 1 0 w | data | data if w=1 | | | |

- **Multi-Byte Example using R/M field with memory displacement:**
  - **Encode the instruction in machine code**

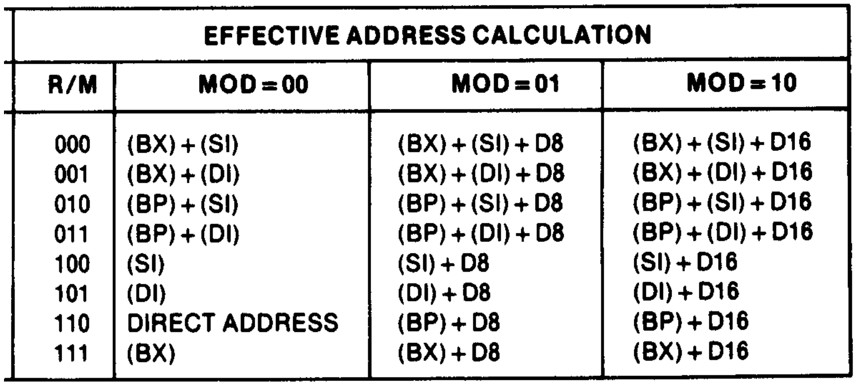
    **XOR CL,[1234H]**
  - **Solution:**
    - **Use "register/memory and register to either" instruction format**
      - **Most general form of XOR instruction**
      - **Displacement needed—direct addressing**
    - **Byte 1**

      **001100(D)(W)**
      - **CL (destination operand) is the register encoded in the REG field of byte 2 (1st register)**
        - **D = 1 = destination**
      - **XOR is of byte wide data**
        - **W = 0 = byte wide**
      - **Byte 1 = 001100(1)(0) = 00110010 =32H**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



| REG | W=0 | W=1 |
|---|---|---|
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |



EFFECTIVE ADDRESS CALCULATION

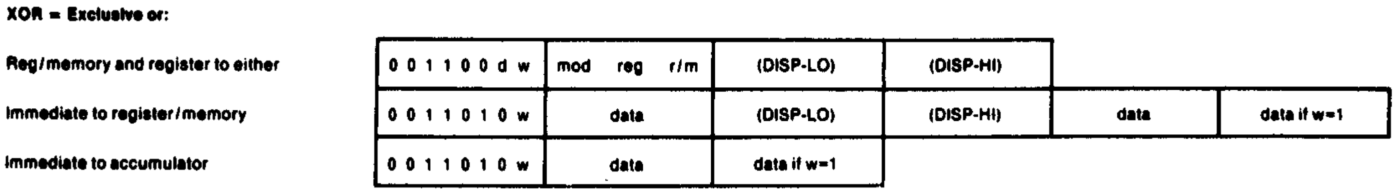
| R/M | MOD=00 | MOD=01 | MOD=10 |
|---|---|---|---|
| 000 | (BX)+(SI) | (BX)+(SI)+D8 | (BX)+(SI)+D16 |
| 001 | (BX)+(DI) | (BX)+(DI)+D8 | (BX)+(DI)+D16 |
| 010 | (BP)+(SI) | (BP)+(SI)+D8 | (BP)+(SI)+D16 |
| 011 | (BP)+(DI) | (BP)+(DI)+D8 | (BP)+(DI)+D16 |
| 100 | (SI) | (SI)+D8 | (SI)+D16 |
| 101 | (DI) | (DI)+D8 | (DI)+D16 |
| 110 | DIRECT ADDRESS | (BP)+D8 | (BP)+D16 |
| 111 | (BX) | (BX)+D8 | (BX)+D16 |

- **Multi-Byte Example using R/M field with memory displacement (continued):**
  - **Byte 2**

    **(MOD)(REG)(R/M)**
    - **Second operand is in memory and pointed to by a direct address**
      - **MOD = 00 → direct address**
    - **R/M specifies the addressing mode**
      - **R/M = 110 → direct address**
    - **1st register is destination register CL**
      - **REG = 001**

      **(00)001(110) = 00001110 = 0EH**

# 3.3 Converting Assembly Language to Machine Code- General Instruction Format



| | | | | | | |
|---|---|---|---|---|---|---|
| XOR = Exclusive or: | | | | | | |
| Reg/memory and register to either | 0 0 1 1 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| Immediate to register/memory | 0 0 1 1 0 1 0 w | data | (DISP-LO) | (DISP-HI) | data | data if w=1 |
| Immediate to accumulator | 0 0 1 1 0 1 0 w | data | data if w=1 | | | |

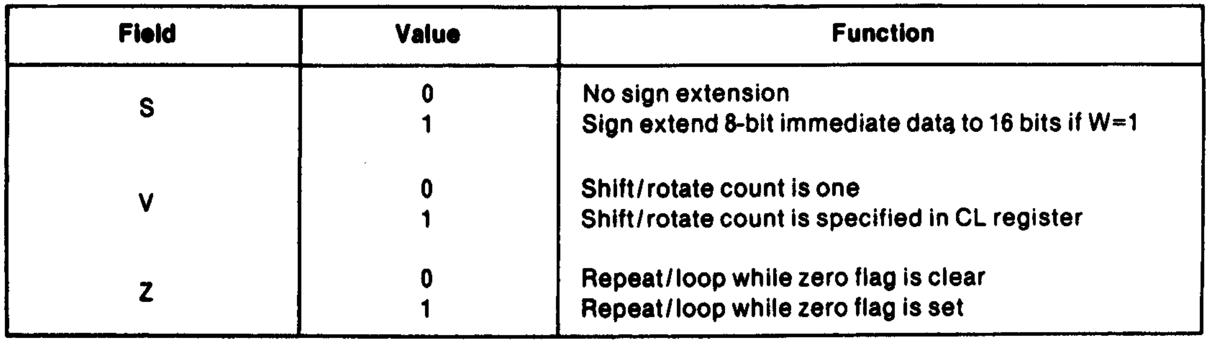
- **Multi-Byte Example using R/M field with memory displacement (continued):**
  - **Bytes 3 & 4**

    **(LOW DISP) (HIGH DISP)**
    - **Indirect address is the displacement from the current data segment address (DS)**
      - **[1234H] = [12 34]**
      - **Byte 3 = LOW DISP = 34H =**
      - **Byte 4 = HIGH DISP =12H**
      -
  - **Machine code is:**

    **(Byte 1)(Byte 2)(Byte 3(Byte 4) = 320E3412H → two byte instruction**

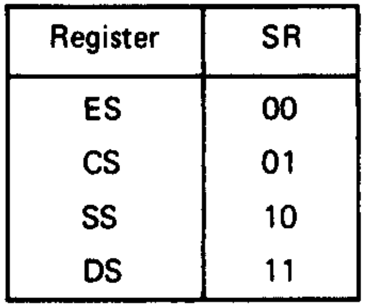
    **XOR CL,[1234H] = 320E3412H**

## 3.3 Converting Assembly Language to Machine Code- General Instruction Format



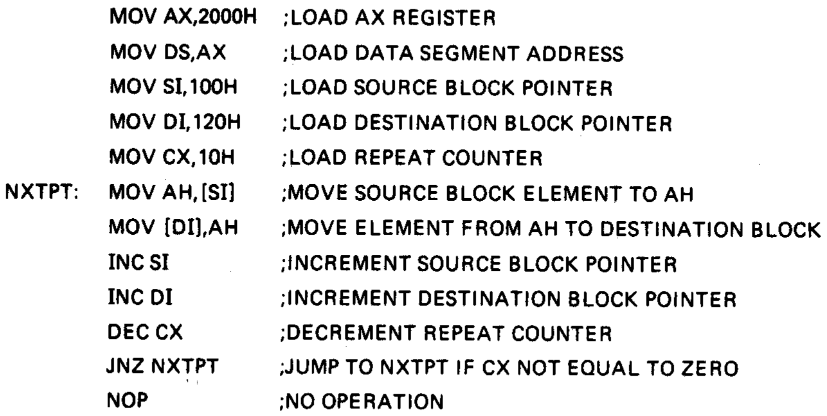
| Field | Value | Function |
|---|---|---|
| S | 0<br>1 | No sign extension<br>Sign extend 8-bit immediate data to 16 bits if W=1 |
| V | 0<br>1 | Shift/rotate count is one<br>Shift/rotate count is specified in CL register |
| Z | 0<br>1 | Repeat/loop while zero flag is clear<br>Repeat/loop while zero flag is set |

- **SR (2-bit segment register field)—used in formats of instructions to specify a segment register**
  - **SR = 11 → DS = data segment register**
  - **ES = extra segment register**
- **rpose fields**
  - **unt for shift and rotate instructions**
  - **) = shift count is 1**
  - **= shift count is in CL register**
  - **condition for REP string instruction**
  - **) → repeat while ZF =0**
  - **Z = 1 → repeat while ZF =1**

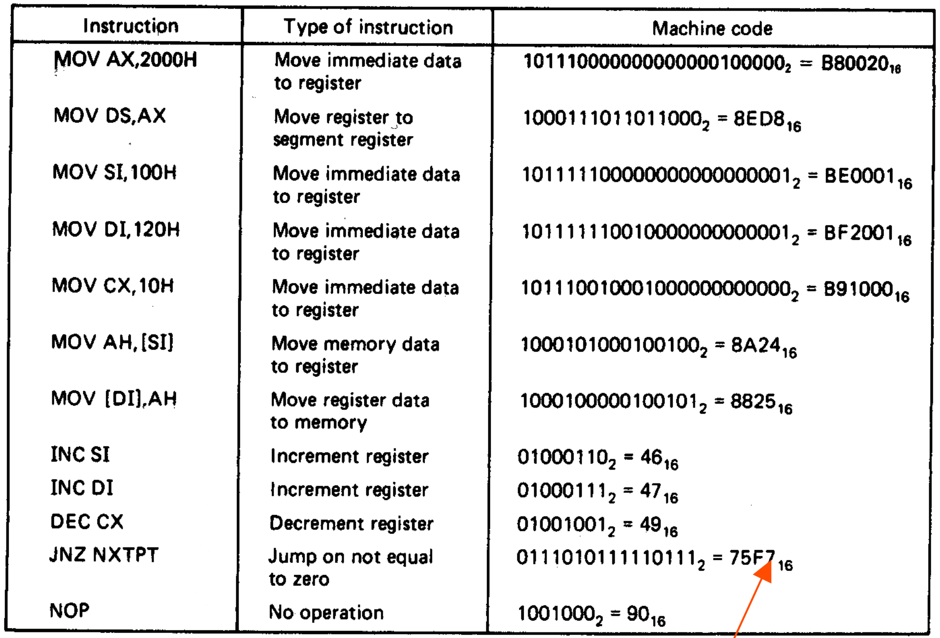


| Register | SR |
|---|---|
| ES | 00 |
| CS | 01 |
| SS | 10 |
| DS | 11 |

# Translating Assembly Langauge to Machine Code



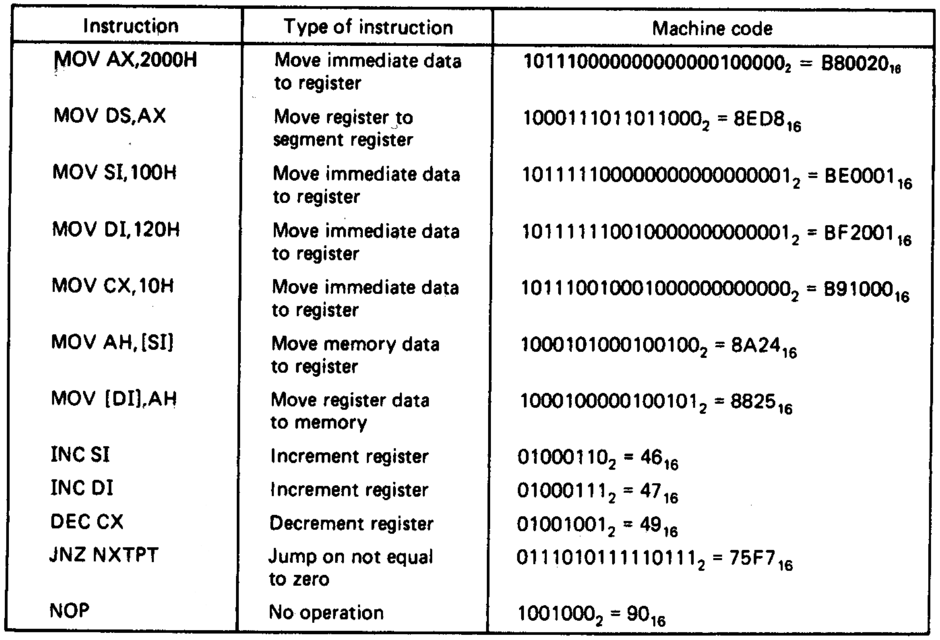
```
         MOV AX,2000H    ;LOAD AX REGISTER
         MOV DS,AX       ;LOAD DATA SEGMENT ADDRESS
         MOV SI,100H     ;LOAD SOURCE BLOCK POINTER
         MOV DI,120H     ;LOAD DESTINATION BLOCK POINTER
         MOV CX,10H      ;LOAD REPEAT COUNTER
NXTPT:   MOV AH,[SI]     ;MOVE SOURCE BLOCK ELEMENT TO AH
         MOV [DI],AH     ;MOVE ELEMENT FROM AH TO DESTINATION BLOCK
         INC SI          ;INCREMENT SOURCE BLOCK POINTER
         INC DI          ;INCREMENT DESTINATION BLOCK POINTER
         DEC CX          ;DECREMENT REPEAT COUNTER
         JNZ NXTPT       ;JUMP TO NXTPT IF CX NOT EQUAL TO ZERO
         NOP             ;NO OPERATION
```



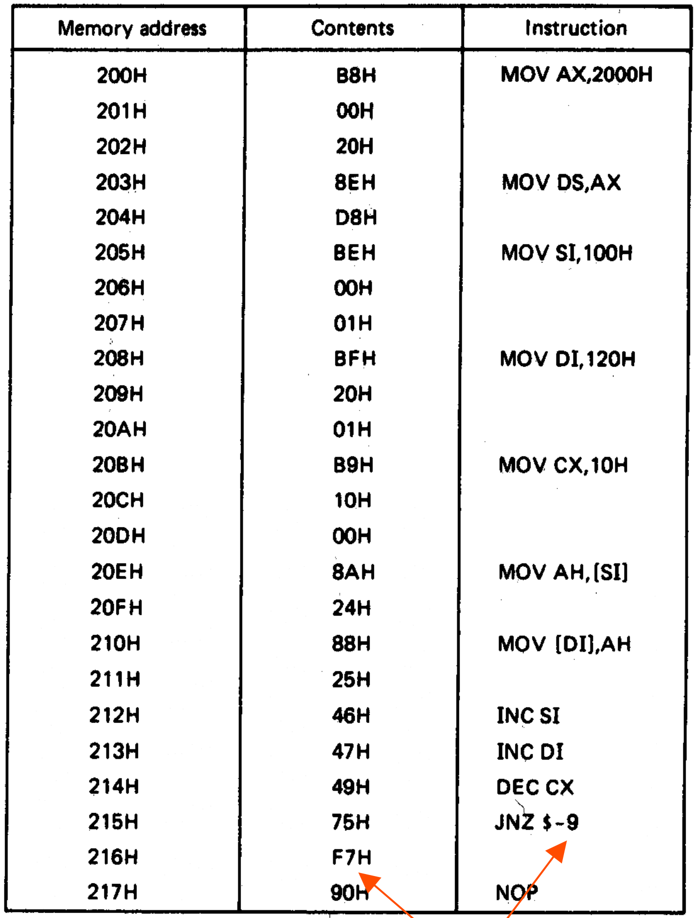
| Instruction | Type of instruction | Machine code |
|---|---|---|
| MOV AX,2000H | Move immediate data to register | $101110000000000000100000_2$ = $B80020_{16}$ |
| MOV DS,AX | Move register to segment register | $1000111011011000_2$ = $8ED8_{16}$ |
| MOV SI,100H | Move immediate data to register | $101111100000000000000001_2$ = $BE0001_{16}$ |
| MOV DI,120H | Move immediate data to register | $101111110010000000000001_2$ = $BF2001_{16}$ |
| MOV CX,10H | Move immediate data to register | $101110010001000000000000_2$ = $B91000_{16}$ |
| MOV AH,[SI] | Move memory data to register | $1000101000100100_2$ = $8A24_{16}$ |
| MOV [DI],AH | Move register data to memory | $1000100000100101_2$ = $8825_{16}$ |
| INC SI | Increment register | $01000110_2$ = $46_{16}$ |
| INC DI | Increment register | $01000111_2$ = $47_{16}$ |
| DEC CX | Decrement register | $01001001_2$ = $49_{16}$ |
| JNZ NXTPT | Jump on not equal to zero | $0111010111110111_2$ = $75F7_{16}$ |
| NOP | No operation | $1001000_2$ = $90_{16}$ |

Displacement for jump to NXTPT:

# Storing The Machine Code Program in Memory



| Instruction | Type of instruction | Machine code |
|---|---|---|
| MOV AX,2000H | Move immediate data to register | $101110000000000000100000_2$ = $B80020_{16}$ |
| MOV DS,AX | Move register to segment register | $1000111011011000_2$ = $8ED8_{16}$ |
| MOV SI,100H | Move immediate data to register | $101111100000000000000001_2$ = $BE0001_{16}$ |
| MOV DI,120H | Move immediate data to register | $101111110010000000000001_2$ = $BF2001_{16}$ |
| MOV CX,10H | Move immediate data to register | $101110010001000000000000_2$ = $B91000_{16}$ |
| MOV AH,[SI] | Move memory data to register | $1000101000100100_2$ = $8A24_{16}$ |
| MOV [DI],AH | Move register data to memory | $1000100000100101_2$ = $8825_{16}$ |
| INC SI | Increment register | $01000110_2$ = $46_{16}$ |
| INC DI | Increment register | $01000111_2$ = $47_{16}$ |
| DEC CX | Decrement register | $01001001_2$ = $49_{16}$ |
| JNZ NXTPT | Jump on not equal to zero | $0111010111110111_2$ = $75F7_{16}$ |
| NOP | No operation | $1001000_2$ = $90_{16}$ |



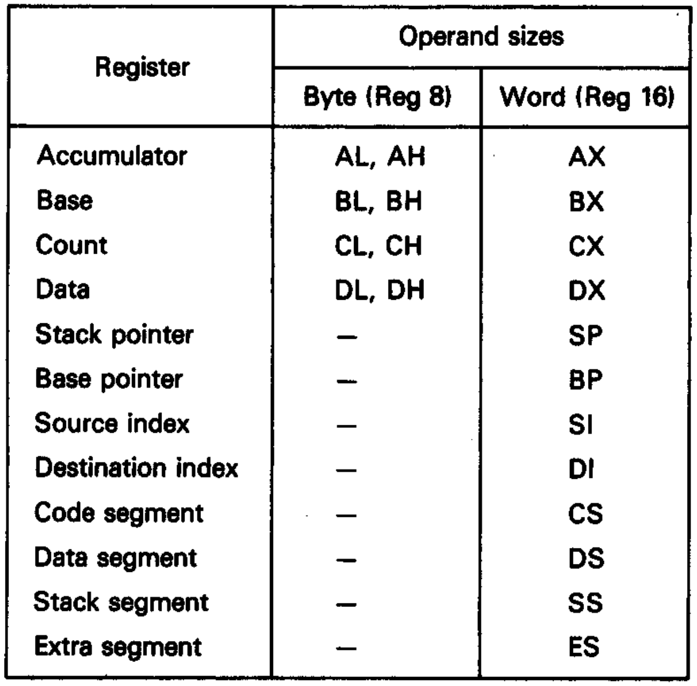
| Memory address | Contents | Instruction |
|---|---|---|
| 200H | B8H | MOV AX,2000H |
| 201H | 00H | |
| 202H | 20H | |
| 203H | 8EH | MOV DS,AX |
| 204H | D8H | |
| 205H | BEH | MOV SI,100H |
| 206H | 00H | |
| 207H | 01H | |
| 208H | BFH | MOV DI,120H |
| 209H | 20H | |
| 20AH | 01H | |
| 20BH | B9H | MOV CX,10H |
| 20CH | 10H | |
| 20DH | 00H | |
| 20EH | 8AH | MOV AH,[SI] |
| 20FH | 24H | |
| 210H | 88H | MOV [DI],AH |
| 211H | 25H | |
| 212H | 46H | INC SI |
| 213H | 47H | INC DI |
| 214H | 49H | DEC CX |
| 215H | 75H | JNZ $-9 |
| 216H | F7H | |
| 217H | 90H | NOP |

Displacement

# Addressing Modes of the 8088/808 Microprocessor-Addressing Modes

- Addressing mode
  - Instructions perform their specified operation on elements of data that are called its operand
  - Types of operands
    - Source operand
    - Destination operand
    - Content of source operand combined with content of destination operand → Result saved in destination operand location
  - Operands may be
    - Part of the instruction—source operand only
    - Held in one of the internal registers—both source and destination operands
    - Stored at an address in memory—either the source or destination operand
    - Held in an input/output port—either the source or destination operand
- Types of addressing modes
  - Register addressing modes
  - Immediate operand addressing
  - Memory operand addressing
  - Each operand can use a different addressing mode

# Register Operand Addressing Mode



| 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 |

- Register addressing mode operands
  - Source operand and destination operands are both held in internal registers of the 8088/8086
  - Only the data registers can be accessed as bytes or words

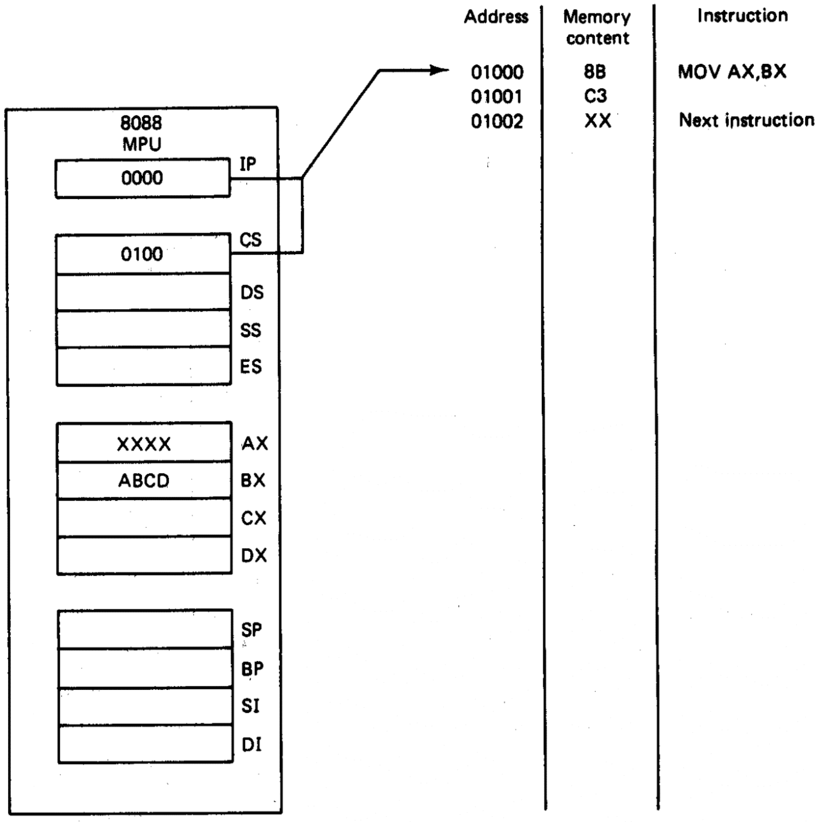
    Ex. AL,AH → bytes

    AX → word
  - Index and pointer registers as words

    Ex. SI → word pointer
  - Segment registers only as words

    Ex. DS → word pointer

# Register Operand Addressing Mode



| Address | Memory content | Instruction |
|---|---|---|
| 01000 | 8B | MOV AX,BX |
| 01001 | C3 | |
| 01002 | XX | Next instruction |

(a)

- 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

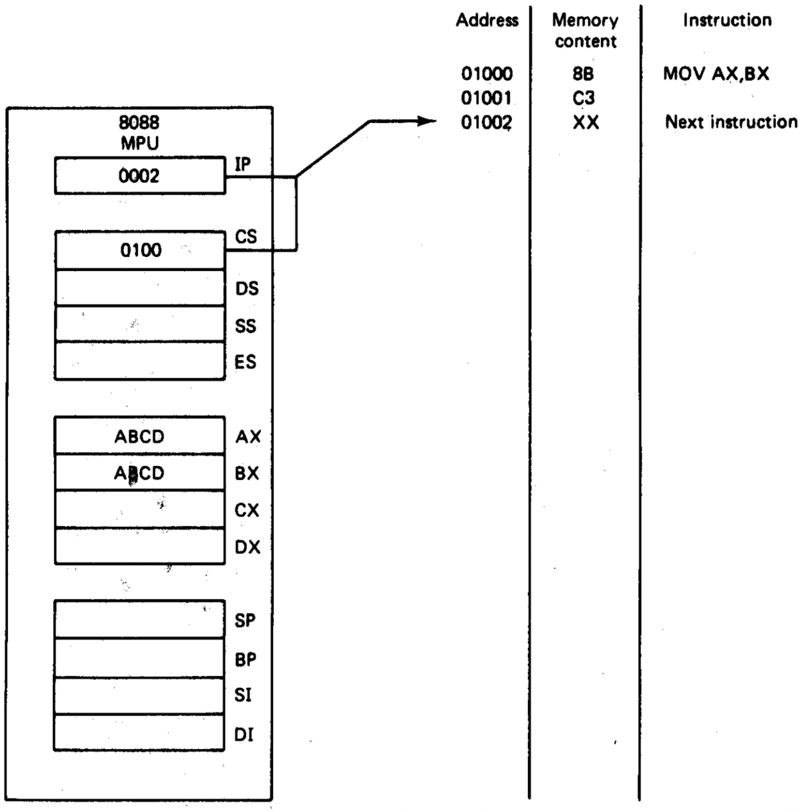
  (01000H) = 8BH

  (01001H) = C3H

  (BX) = ABCDH

  (AX) = XXXX → don’t care state

# Register Operand Addressing Mode



| Address | Memory content | Instruction |
|---|---|---|
| 01000 | 8B | MOV AX,BX |
| 01001 | C3 | |
| 01002 | XX | Next instruction |

(b)

- Example (continued)
- State after execution

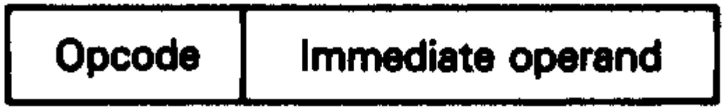
  CS:IP = 0100:0002 = 01002H

  01002H → points to next sequential instruction

  (BX) = ABCDH

  (AX) = ABCDH → Value in BX copied into AX

# Immediate Operand Addressing Mode



Opcode

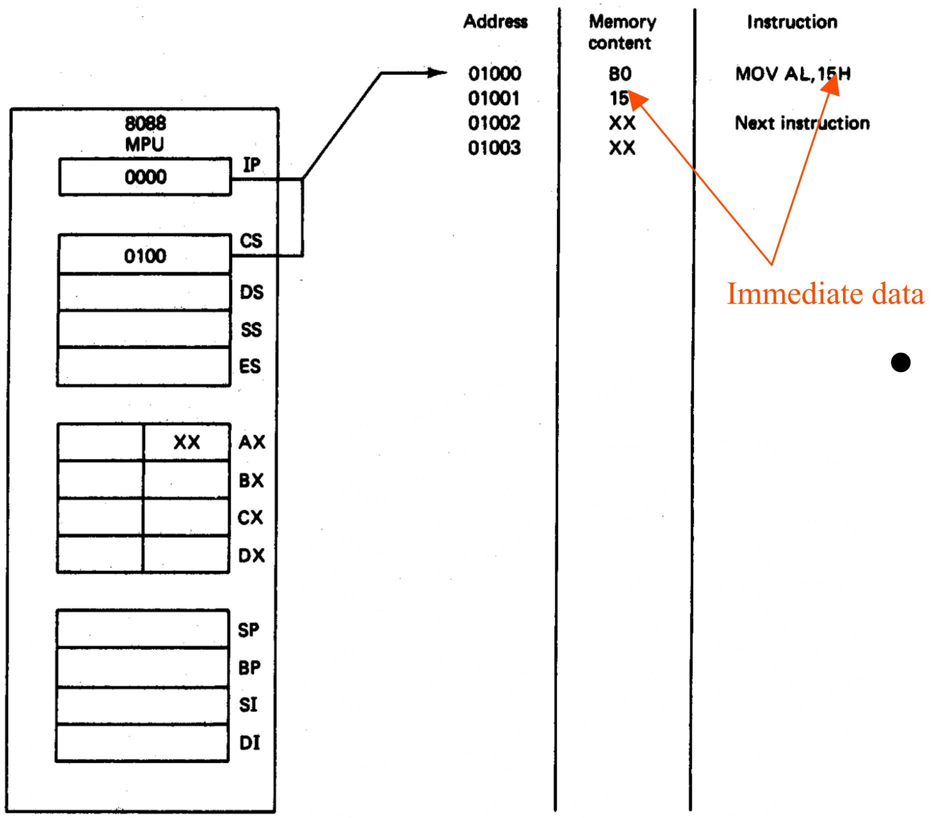
Immediate operand

- Immediate operand
  - 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)

# Immediate Operand Addressing Mode Example



| Address | Memory content | Instruction |
|---|---|---|
| 01000 | B0 | MOV AL,15H |
| 01001 | 15 | |
| 01002 | XX | Next instruction |
| 01003 | XX | |

(a)

- 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

  (AL) = XX → don't care state

# Memory Operand Addressing Mode

$$PA = SBA : EA$$

$$PA = \text{Segment base} : \text{Base} + \text{Index} + \text{Displacement}$$

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

- **Accessing operands in memory**
  - **Only one operand can reside in memory—either the source or destination**
  - **Calculate the 20-bit physical address (PA) at which the operand in  stored in memory**
  - **Perform a read or write to this memory location**
- **Physical address computation**
  - **Given in general as**

    **PA = SBA:EA**

    **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**

# Direct Addressing Mode

PA = Segment base: Direct address

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

- Direct addressing mode
  - 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

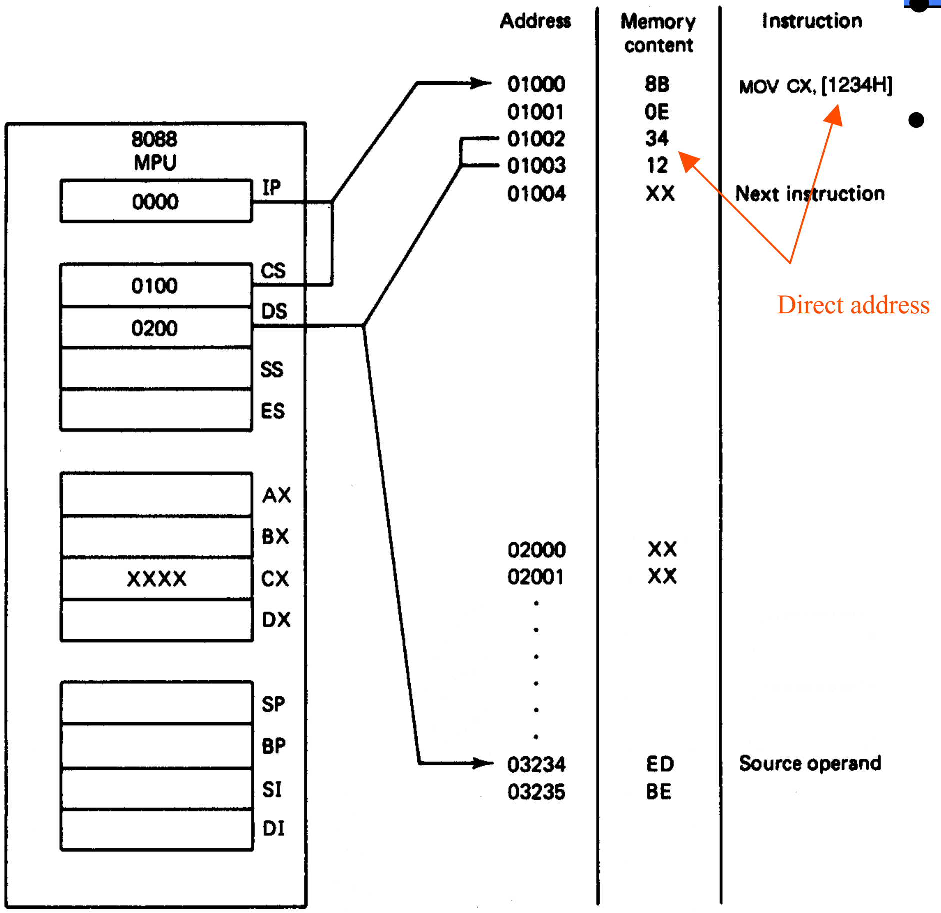
  - Segment base address is DS by default

    PA = DS:[DA]

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

    PA = SEG:ES:[DA]

# Direct Addressing Mode Example

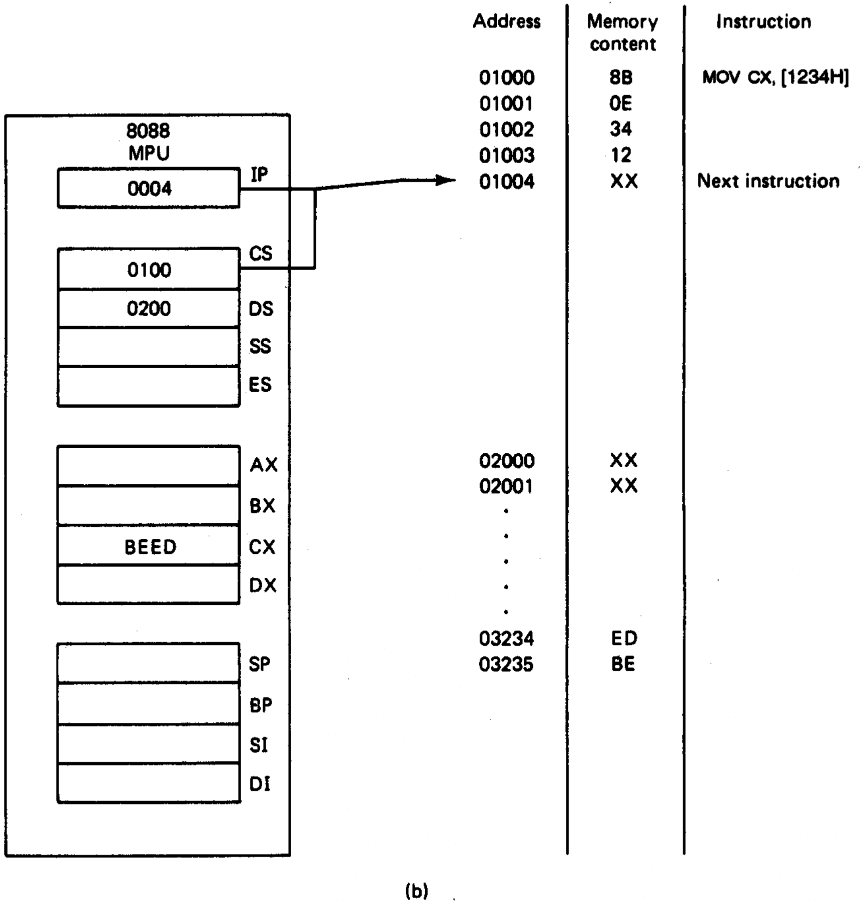


| Address | Memory content | Instruction |
|---|---|---|
| 01000 | 8B | MOV CX, [1234H] |
| 01001 | 0E | |
| 01002 | 34 | |
| 01003 | 12 | |
| 01004 | XX | Next instruction |
| 02000 | XX | |
| 02001 | XX | |
| ⋮ | | |
| 03234 | ED | Source operand |
| 03235 | BE | |

(a)

- 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

# Direct Addressing Mode Example



| Address | Memory content | Instruction |
|---|---|---|
| 01000 | 8B | MOV CX, [1234H] |
| 01001 | 0E | |
| 01002 | 34 | |
| 01003 | 12 | |
| 01004 | XX | Next instruction |
| 02000 | XX | |
| 02001 | XX | |
| . | | |
| . | | |
| . | | |
| . | | |
| . | | |
| 03234 | ED | |
| 03235 | BE | |

- 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) = BEED

# Register Indirect Addressing Mode

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ BP \\ SI \\ DI \end{Bmatrix}$$

- Register indirect addressing mode
  - Similar to direct addressing in that the affective 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

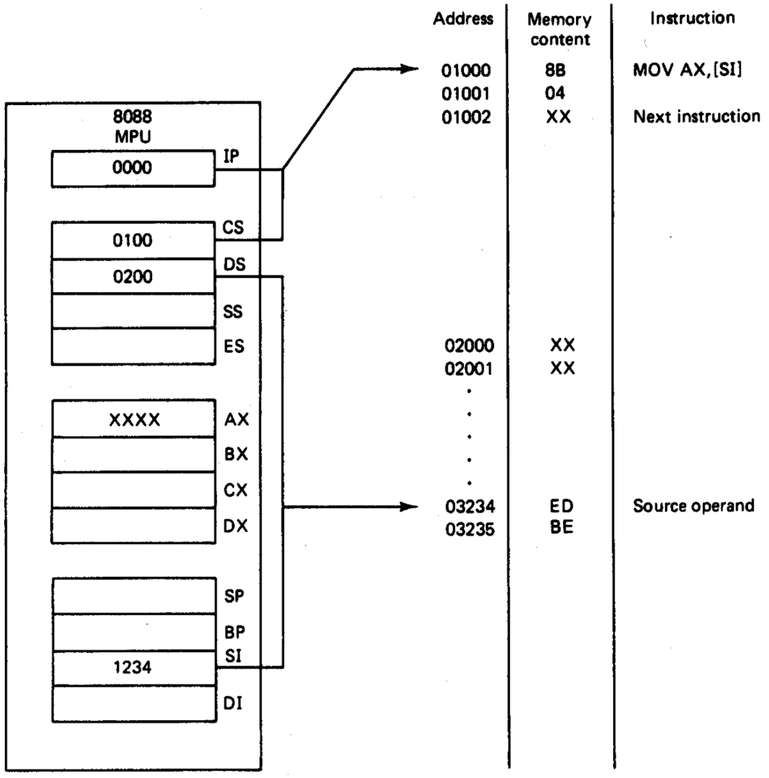
  PA = SBA:[Rx] → 16-bit offset

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

    PA = DS:[Rx]
  - Segment override prefix (SEG) is required to enable use of another segment register

    PA = SEG:ES:[Rx]
  - What about BP?

# Register Indirect Addressing Mode

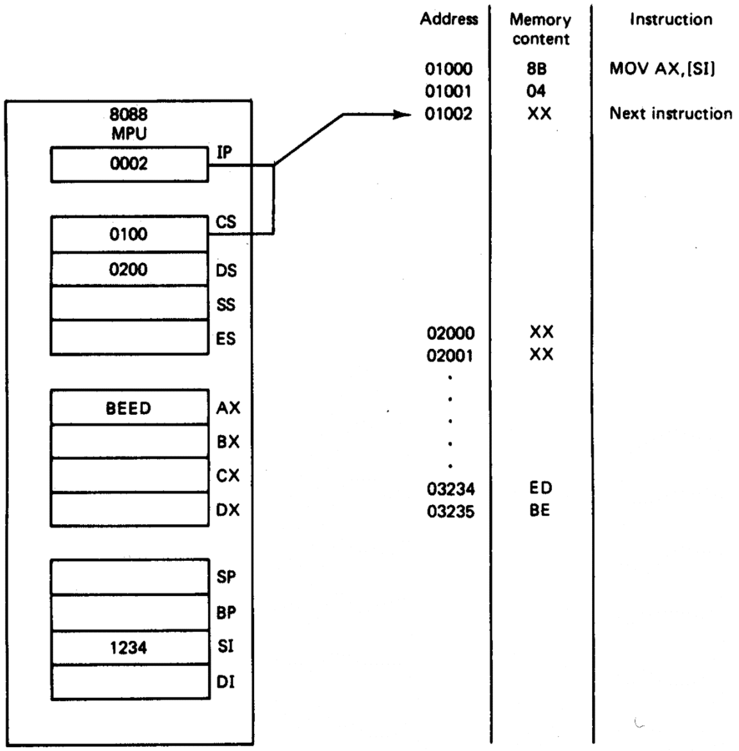


| Address | Memory content | Instruction |
|---|---|---|
| 01000 | 8B | MOV AX,[SI] |
| 01001 | 04 | |
| 01002 | XX | Next instruction |
| 02000 | XX | |
| 02001 | XX | |
| . | | |
| . | | |
| . | | |
| . | | |
| 03234 | ED | Source operand |
| 03235 | BE | |

(a)

- 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

# Register Indirect Addressing Mode



| Address | Memory content | Instruction |
|---|---|---|
| 01000 | 8B | MOV AX,[SI] |
| 01001 | 04 | |
| 01002 | XX | Next instruction |
| 02000 | XX | |
| 02001 | XX | |
| . | | |
| . | | |
| . | | |
| . | | |
| 03234 | ED | |
| 03235 | BE | |

(b)

- Example (continued)
- State after execution
  - Instruction

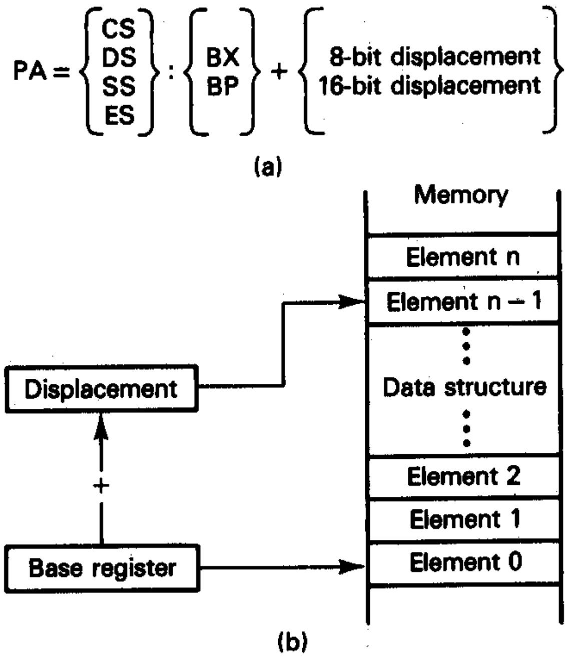
    CS:IP = 0100:0002 = 01002H

    01002H → points to next sequential instruction
  - Source operand

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

    (AX) = BEED

# Base Addressing Mode

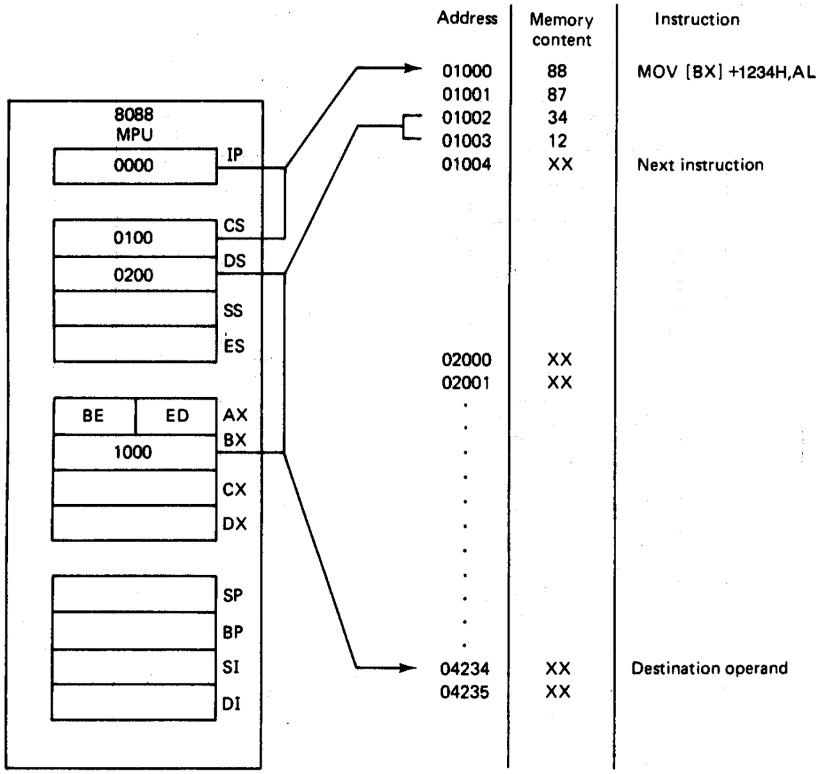


- Based addressing mode
  - Effective address formed from contents of a base register and a displacement
  - Base register is either BX or BP (stack)
    - Direct/indirect displacement is 8-bit or 16bit
- Physical address computation

  PA = SBA:EA → 20-bit address

  PA = SBA:[BX 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

# Base Addressing Mode

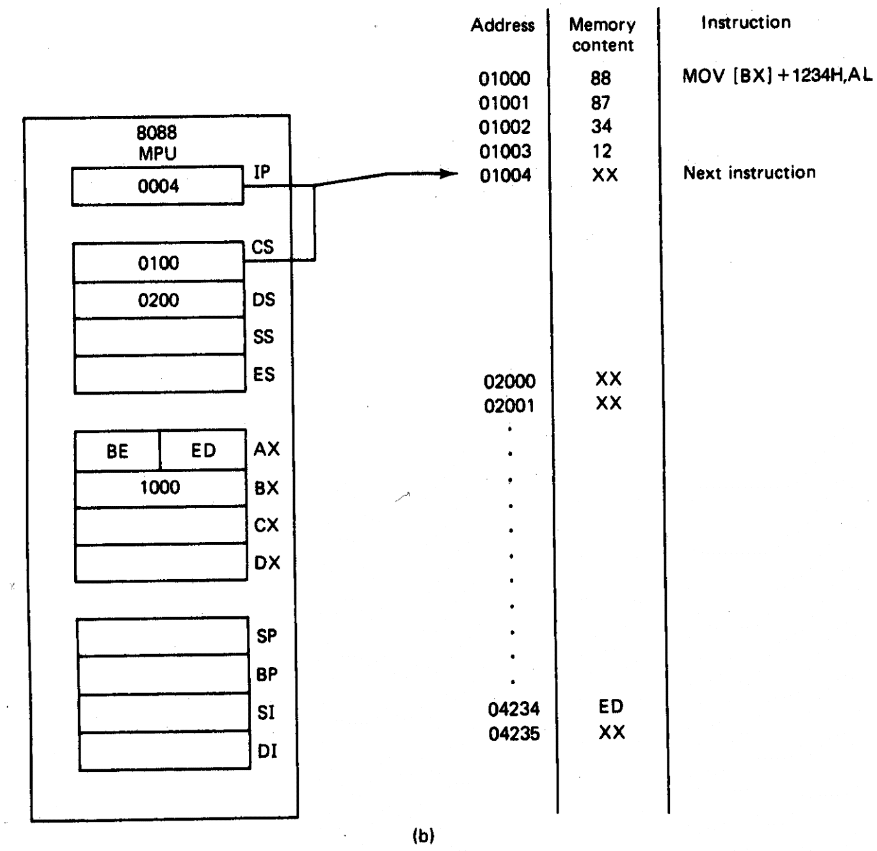


| Address | Memory content | Instruction |
|---|---|---|
| 01000 | 88 | MOV [BX] +1234H,AL |
| 01001 | 87 | |
| 01002 | 34 | |
| 01003 | 12 | |
| 01004 | XX | Next instruction |
| 02000 | XX | |
| 02001 | XX | |
| 04234 | XX | Destination operand |
| 04235 | XX | |

(a)

- 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

# Base Addressing Mode



| Address | Memory content | Instruction |
|---|---|---|
| 01000 | 88 | MOV [BX]+1234H,AL |
| 01001 | 87 | |
| 01002 | 34 | |
| 01003 | 12 | |
| 01004 | XX | Next instruction |
| 02000 | XX | |
| 02001 | XX | |
| ⋮ | | |
| 04234 | ED | |
| 04235 | XX | |

- Example (continued)
- State after execution
  - Instruction

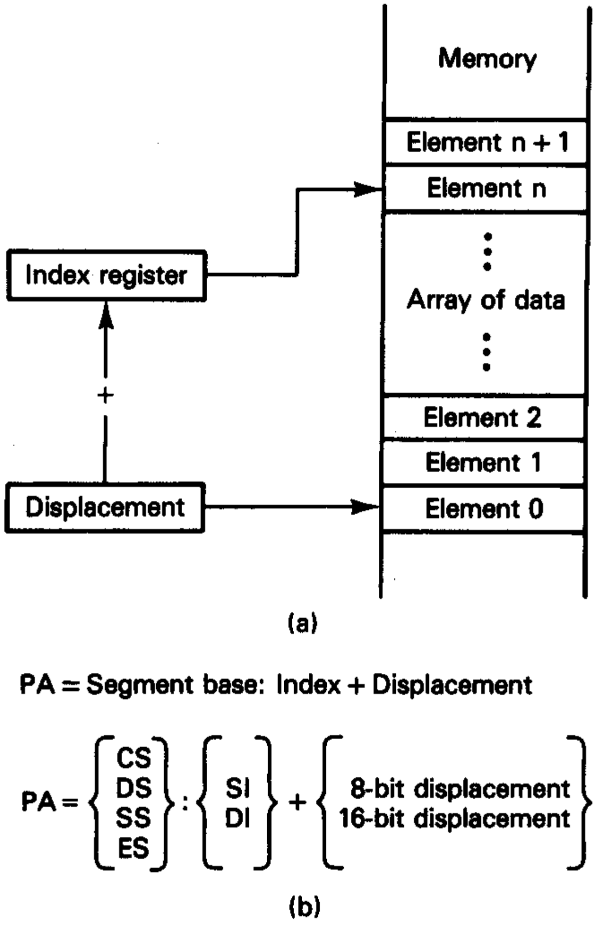
    CS:IP = 0100:0004 = 01004H

    01004H → points to next sequential instruction
  - Destination operand

    (04234H) = EDH
  - Source operand

    (AL) = EDH → unchanged

# Indexed Addressing Mode



(a)

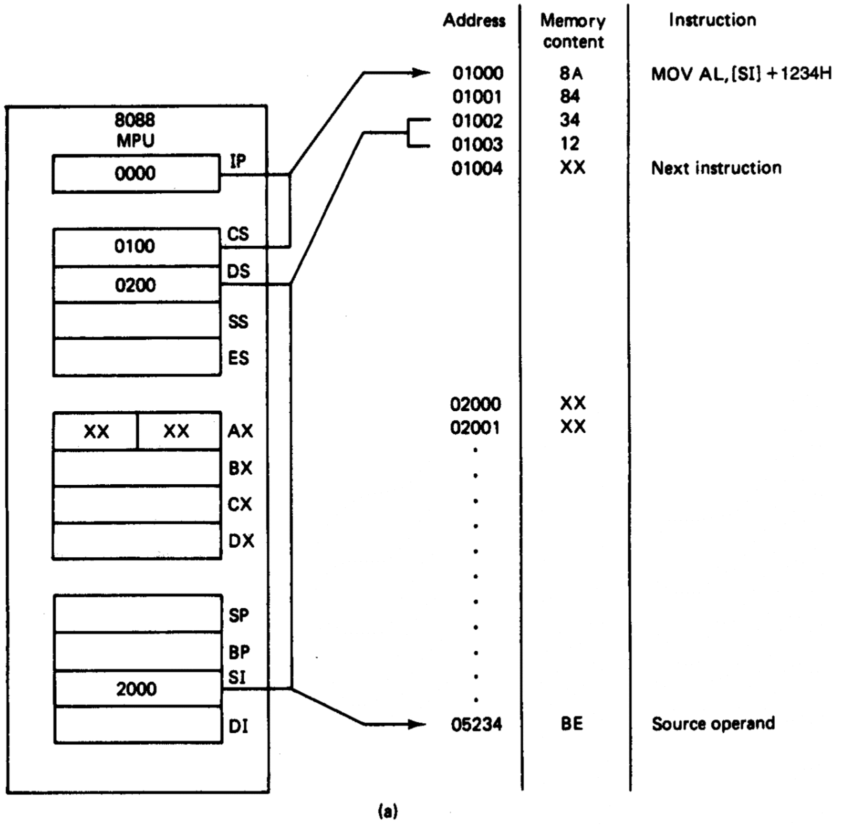
(b)

- Indexed addressing mode
  - 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 in the array
  - Program simply changes the value of the displacement to access another array
  - Program changes (recomputes) 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]

# Indexed Addressing Mode



| Address | Memory content | Instruction |
|---|---|---|
| 01000 | 8A | MOV AL, [SI] +1234H |
| 01001 | 84 | |
| 01002 | 34 | |
| 01003 | 12 | |
| 01004 | XX | Next instruction |
| 02000 | XX | |
| 02001 | XX | |
| 05234 | BE | Source operand |

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

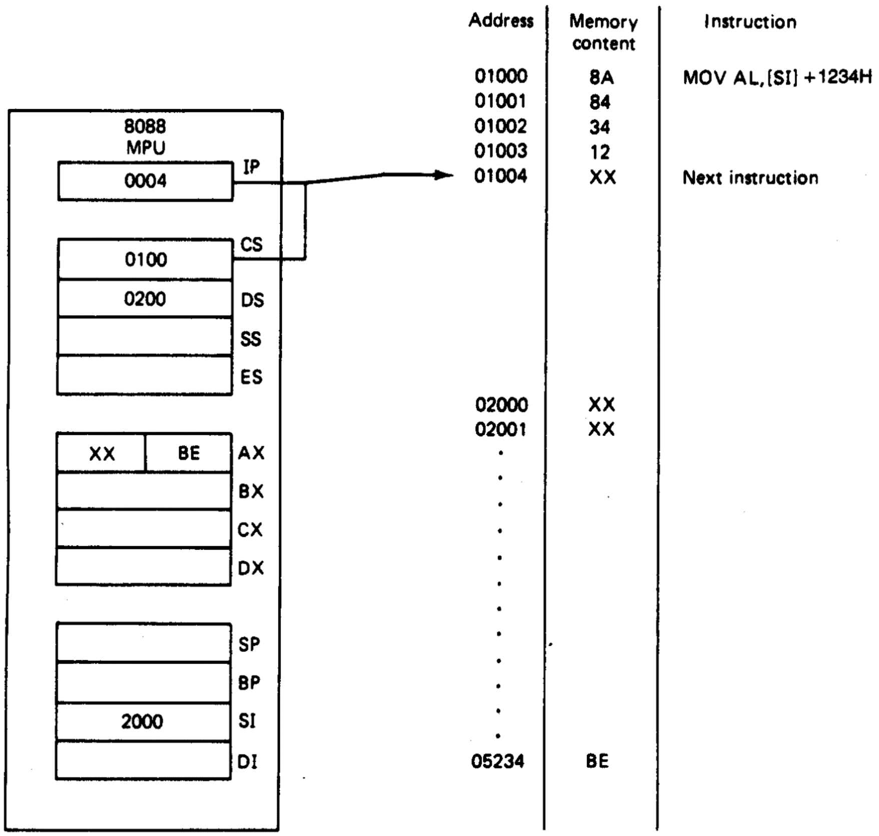
  = 02000H+2000H+1234H

  = 05234H

  (05234H) = BEH
- Destination operand—register operand addressing

  (AL) = XX → don't care state

# Indexed Addressing Mode



| Address | Memory content | Instruction |
|---|---|---|
| 01000 | 8A | MOV AL,[SI] +1234H |
| 01001 | 84 | |
| 01002 | 34 | |
| 01003 | 12 | |
| 01004 | XX | Next instruction |
| 02000 | XX | |
| 02001 | XX | |
| ⋮ | | |
| 05234 | BE | |

(b)

- Example (continued)
- State after execution
  - Instruction

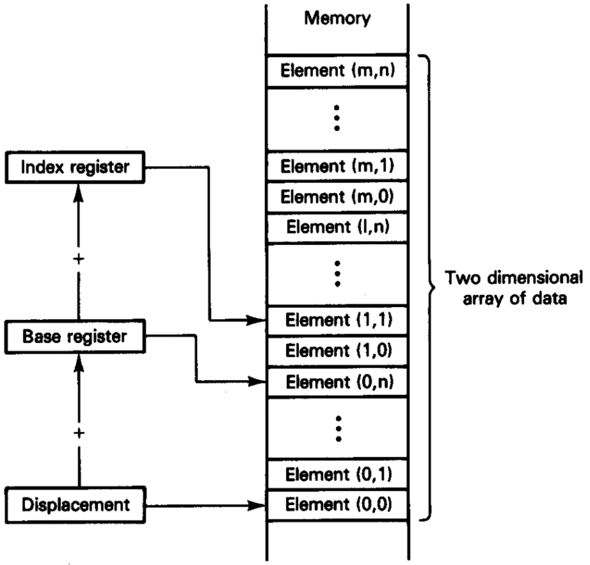
    CS:IP = 0100:0004 = 01004H

    01004H → points to next sequential instruction
  - Source operand

    (05234H) = BEH → unchanged
  - Destination operand

    (AL) = BEH

# Based-Indexed Addressing Mode



(a)

PA = Segment base: Base + Index + Displacement

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

(b)

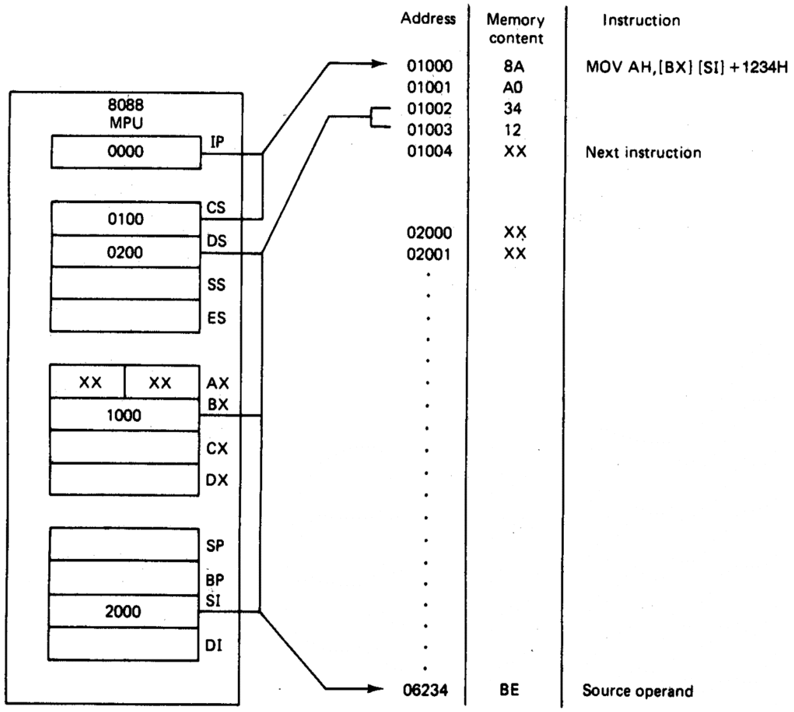
- Based-indexed addressing mode
  - Combines the functions of based and indexed addressing modes
  - Enables easy access to two-dimensional arrays of data
  - Displacement points to the beginning of array in memory
  - Base register selects the row (*m*) of elements
  - Index register selects element 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]

# Based- Indexed Addressing Mode: Example

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

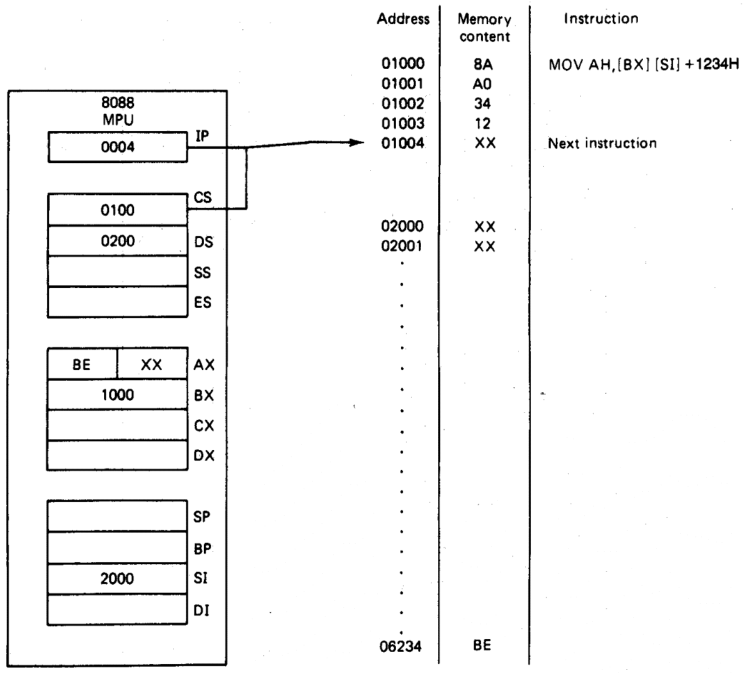


(a)

State before fetch and execution

- Instruction
  CS = 0100H, IP = 0000H
  CS:IP = 0100:0000H = 01000H
  (01000H,01001H) = 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
  = 0200H:1234H + 1000H + 2000H
  = 02000H+1234H +1000H + 2000H
  = 06234H
  (06234H) = BEH
- Destination operand—register operand addressing
  (AH) = XX → don’t care state

# Based- Indexed Addressing Mode



| Address | Memory content | Instruction |
|---|---|---|
| 01000 | 8A | MOV AH,[BX] [SI] +1234H |
| 01001 | A0 | |
| 01002 | 34 | |
| 01003 | 12 | |
| 01004 | XX | Next instruction |
| 02000 | XX | |
| 02001 | XX | |
| 06234 | BE | |

(b)

- Example (continued)
- State after execution
  - Instruction

    CS:IP = 0100:0004 = 01004H

    01004H → points to next sequential instruction
  - Source operand

    (06234H) = BEH → unchanged
  - Destination operand

    (AH) = BEH