Faculty of Information and Communication Technology

UCEC1044 Basic Microprocessor

Arts of 8088/86 programming

Y.C.See
chark97@hotmail.com

Definition

- **A program**: A sequence of instructions, commands or statements to be executed by a microprocessor. Programs are normally written in a computer language.

- **Machine language**: Is the coded sequences of the 0's and 1', which a CPU understands. Any program written in any computer language must eventually be translated to machine language. Only computer can understand.
- **Assembly language**: Uses alphanumeric **mnemonics**, which when combined with certain character symbols forms the **instructions** that a microprocessor understands. The instructions are combined with other symbols to make **assembly language statements**. The statements obey certain syntax rules that are defined by the assembly language designers. Assembly language statements need another program (called the **assembler**) to translate them into machine code.
- **The instruction set**: The collection of instructions (normally grouped by functionality) that a certain microprocessor understands. Each microprocessor has its own instruction set.

- **High level language (HLL)**: A computer language, which is microprocessor independent, as long as there is an operating system dependent program (called the **compiler**) that translated it into machine code. Eg. Java, C program
- **Source code**: Any program written in assembly language or an HLL is referred to as source code.

Why assembly language?

- **Hardware prospective**
  - Assembly language teaches how a computer works at the machine level (i.e. registers)
  - Assembly language helps understand the limitations of the Von Neumann architecture

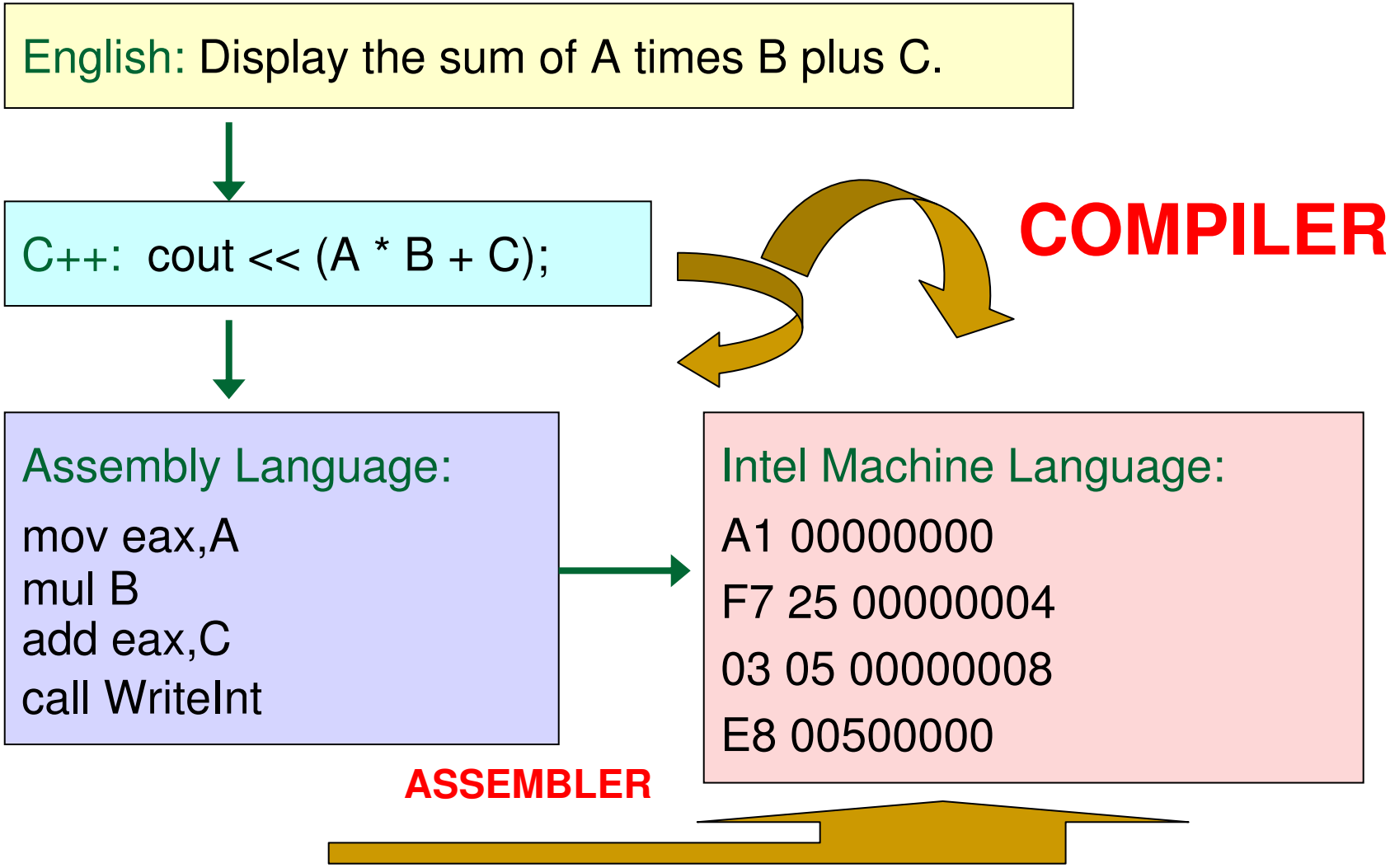
- **Software prospective**
  - The foundation of many abstract issues in software lies in assembly language and computer architecture
  - Data types, addressing modes, stack, input/output
  - Takes up less memory
  - Executes much faster

  - **Assembly language is not used just to illustrate algorithms, but to demonstrate what is actually happening inside the computer!**

Comparison HLL and AL

Type of Application	High-Level Languages	Assembly Language
Business application software, written for single platform, medium to large size.	Formal structures make it easy to organize and maintain large sections of code.	Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.
Hardware device driver.	Language may not provide for direct hardware access. Even if it does, awkward coding techniques must often be used, resulting in maintenance difficulties.	Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.
Business application written for multiple platforms (different operating systems).	Usually very portable. The source code can be recompiled on each target operating system with minimal changes.	Must be recoded separately for each platform, often using an assembler with a different syntax. Difficult to maintain.
Embedded systems and computer games requiring direct hardware access.	Produces too much executable code, and may not run efficiently.	Ideal, because the executable code is small and runs quickly.

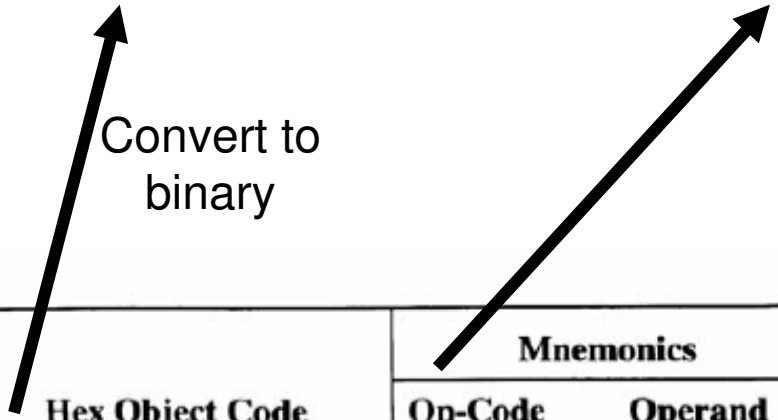
Process of translating



Compiler vs Assembler

- The compiler takes as its source code a C program; this is a file of ASCII characters. It produces either an assembly language, as an intermediate step, or else machine code directly. Either way, the final result is machine code.
- The assembler takes as its source code an assembly language; this is also a file of ASCII characters; it used this to produce machine code.

Machine vs. Assembly Language



Address	Hex Object Code				Mnemonics		Comment
					Op-Code	Operand	
0100	E4	27			IN	AL,27H	Input first number from port 27H and store in AL
0102	88	C3			MOV	BL,AL	Save a copy of register AL in register BL
0104	E4	27			IN	AL,27H	Input second number to AL
0106	00	D8			ADD	AL,BL	Add contents of BL to AL and store the sum in AL
0107	E6	30			OUT	30H,AL	Output AL to port 30H
0109	F4				HLT		Halt the computer

High Level Language

```
#include <stdio.h>

main ()
{
        int             N1;
        int             N2;
        int             Sum;

        printf("\nEnter the first number to add    :");
        scanf("%d,&N1);

        printf("\nEnter the second number to add    :");
        scanf("%d,&N2);

        sum = N1 + N2;

        printf("%d + %d = %d\n", N1, N2, Sum);

}
```

Steps to creating a program

1. Edit or create a file using a text editor
2. Assemble the source code into machine code object files
3. Link the object files into an executable program
4. Convert the executable to a binary file
5. Download the file
6. Run it!

MASM (1)

- Type your assembly code in NOTEPAD and save as abc.asm then,
- C:\MASM c:\see\abc.asm
  - [few lines will appear and you just need to press enter for three times and this process will compile you file for errors and generate abc.obj and abc.lst]
- C:\LINK c:\see\abc
  - Executable file created
- Run the executable file…which is abc.exe
  - Executable file created

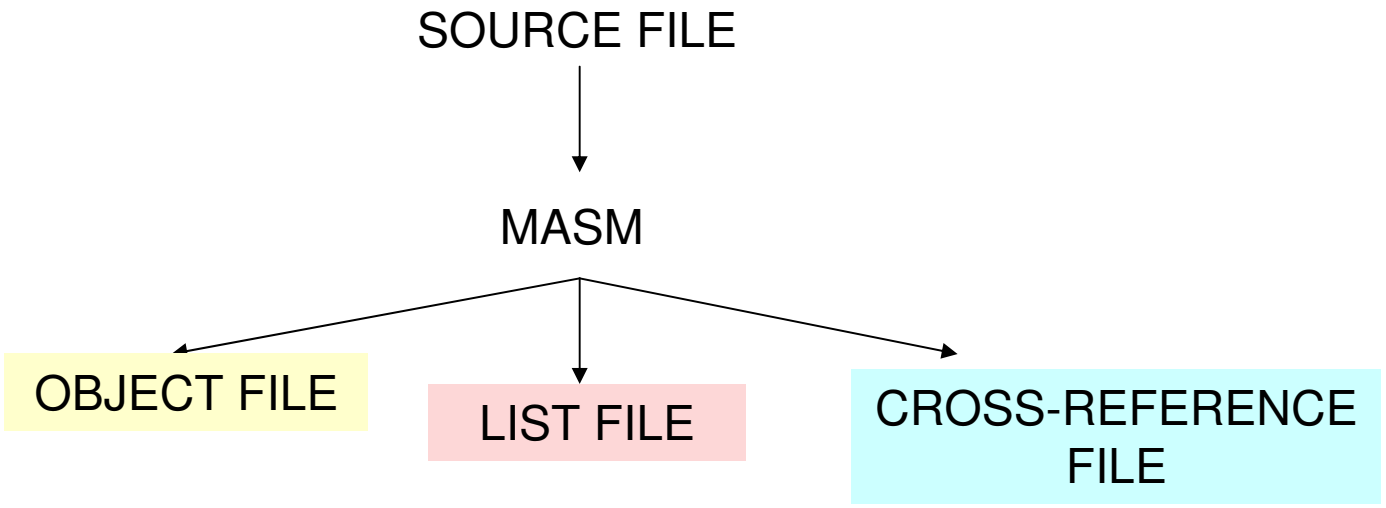
MASM (2)

- The MASM assembler can download it from the lecturer website or from

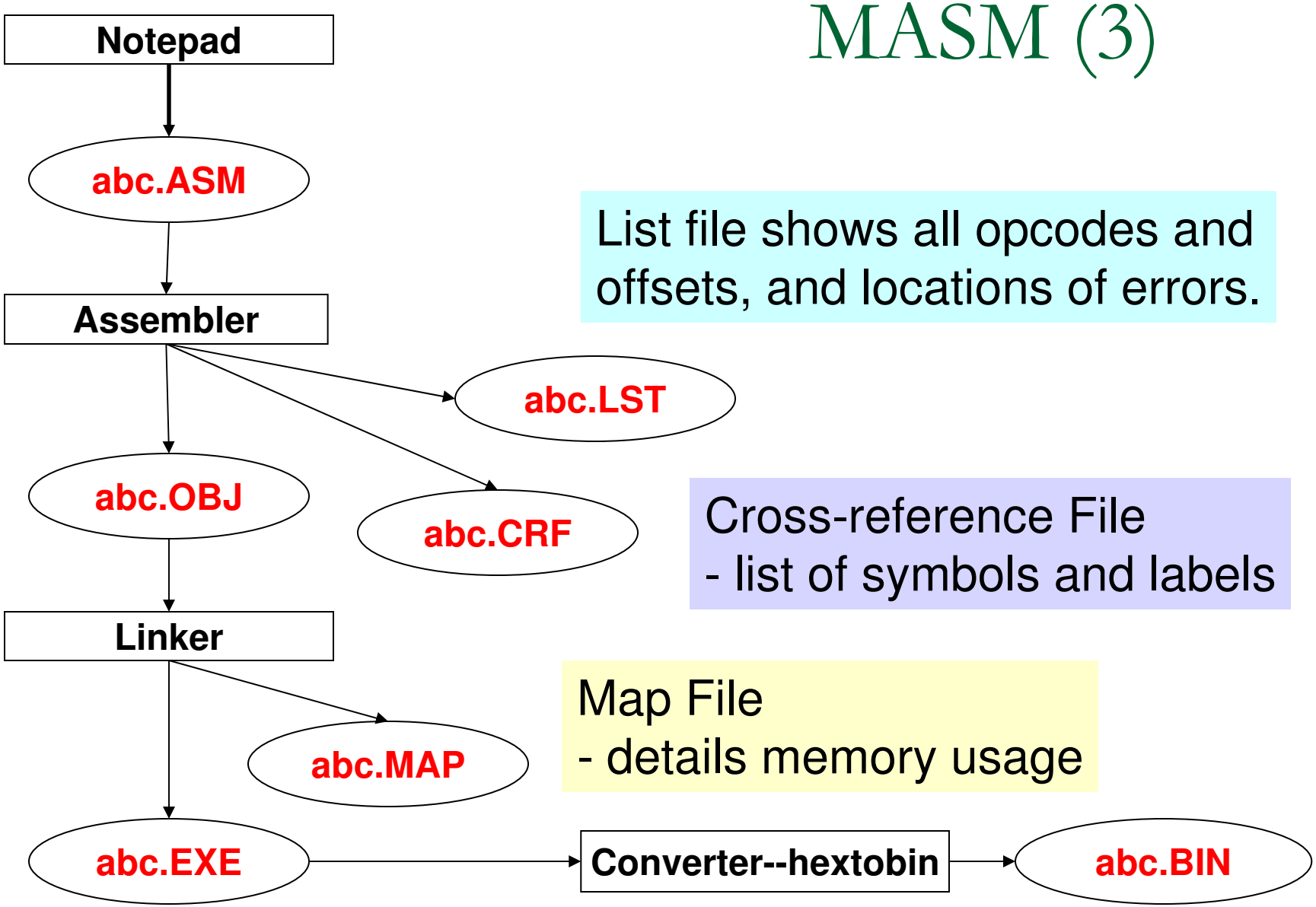
  http://www.masm32.com/masmdl.htm

  It's an FREEWARE

- Translate an assembly code to machine code by generates the following files



MASM (3)



List file shows all opcodes and offsets, and locations of errors.

Cross-reference File
- list of symbols and labels

Map File
- details memory usage

Files created by the assembler and linker

Filename	Description	Step When Created
hello.asm	Source program	Edit
hello.obj	Object program	Assembly
hello.lst	Listing file	Assembly
hello.exe	Executable program	Link
hello.map	Map file	Link

Extension	Meaning
.ASM	Source code created by the editor
.BAK	Source code backup
.BSC	Project database created by BSCMAKE
.COM	Executable program limited to a single segment
.DBG	Debug file created by the linker for .COM files only
.EXE	Executable program with multiple segments
.LIB	Library file
.LST	List file, created by the assembler, containing object and source code
.OBJ	Object code created by the assembler
.MAP	Map file created by the linker showing program segments
.SBR	Database file created by the assembler for the current module

DEBUG

Debug commands may be divided into four categories: program creation/debugging, memory manipulation, miscellaneous, and input-output:

Program Creation and Debugging

A Assemble a program using instruction mnemonics

G Execute the program currently in memory

R Display the contents of registers and flags

- Go to dos prompt or type *cmd* on the run command
- In dos prompt type *"debug"* and then *?*

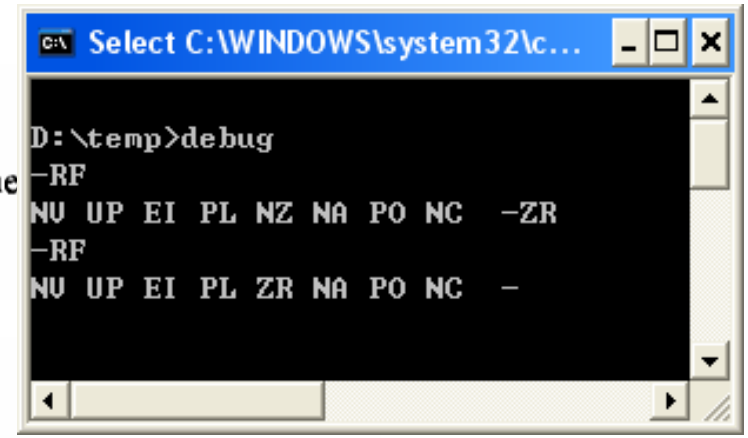
Debug Command Set

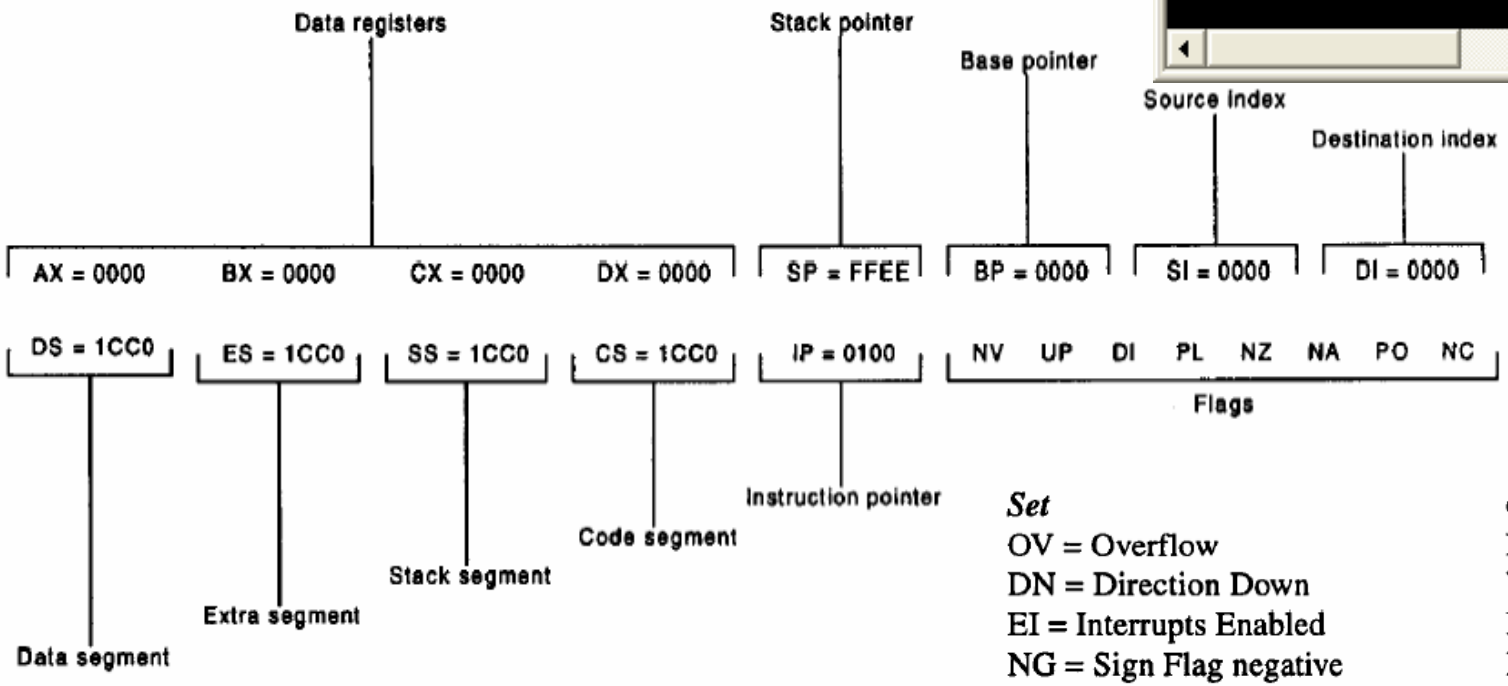
Command	Syntax	Function
Register	R [REGISTER NAME]	Examine or modify the contents of an internal register
Quit	Q	End use of the DEBUG program
Dump	D [ADDRESS]	Dump the contents of memory to the display
Enter	E ADDRESS [LIST]	Examine or modify the contents of memory
Fill	F STARTING ADDRESS ENDING ADDRESS LIST	Fill a block in memory with the data in list
Move	M STARTING ADDRESS ENDING ADDRESS DESTINATION ADDRESS	Move a block of data from a source location in memory to a destination location
Compare	C STARTING ADDRESS ENDING ADDRESS DESTINATION ADDRESS	Compare two blocks of data in memory and display the locations that contain different data
Search	S STARTING ADDRESS ENDING ADDRESS LIST	Search through a block of data in memory and display all locations that match the data in list
Input	I ADDRESS	Read the input port
Output	O ADDRESS, BYTE	Write the byte to the output port
Hex Add/Subtract	H NUM1,NUM2	Generate hexadecimal sum and difference of the two numbers
Unassemble	U [STARTING ADDRESS ENDING ADDRESS]	Unassemble the machine code into its equivalent assembler instructions
Name	N FILE NAME	Assign the filename to the data to be written to the disk
Write	W [STARTING ADDRESS [DRIVE STARTING SECTOR NUMBER OF SECTORS]]	Save the contents of memory in a file on a diskette
Load	L [STARTING ADDRESS [DRIVE STARTING SECTOR NUMBER OF SECTORS]]	Load memory with the contents of a file on a diskette
Assemble	A [STARTING ADDRESS]	Assemble the instruction into machine code and store in memory
Trace	T [=ADDRESS] [NUMBER]	Trace the execution of the specified number of instructions
Go	G [=STARTING ADDRESS [BREAKPOINT ADDRESS....]]	Execute the instructions down through the breakpoint address

Register and Flag Name in Debug

- ***R** register*

Example	Description
R	Display the contents of all registers.
R IP	Display the contents of IP and prompt for a new value
R CX	Same (for the CX register).
R F	Display all flags and prompt for a new flag value.

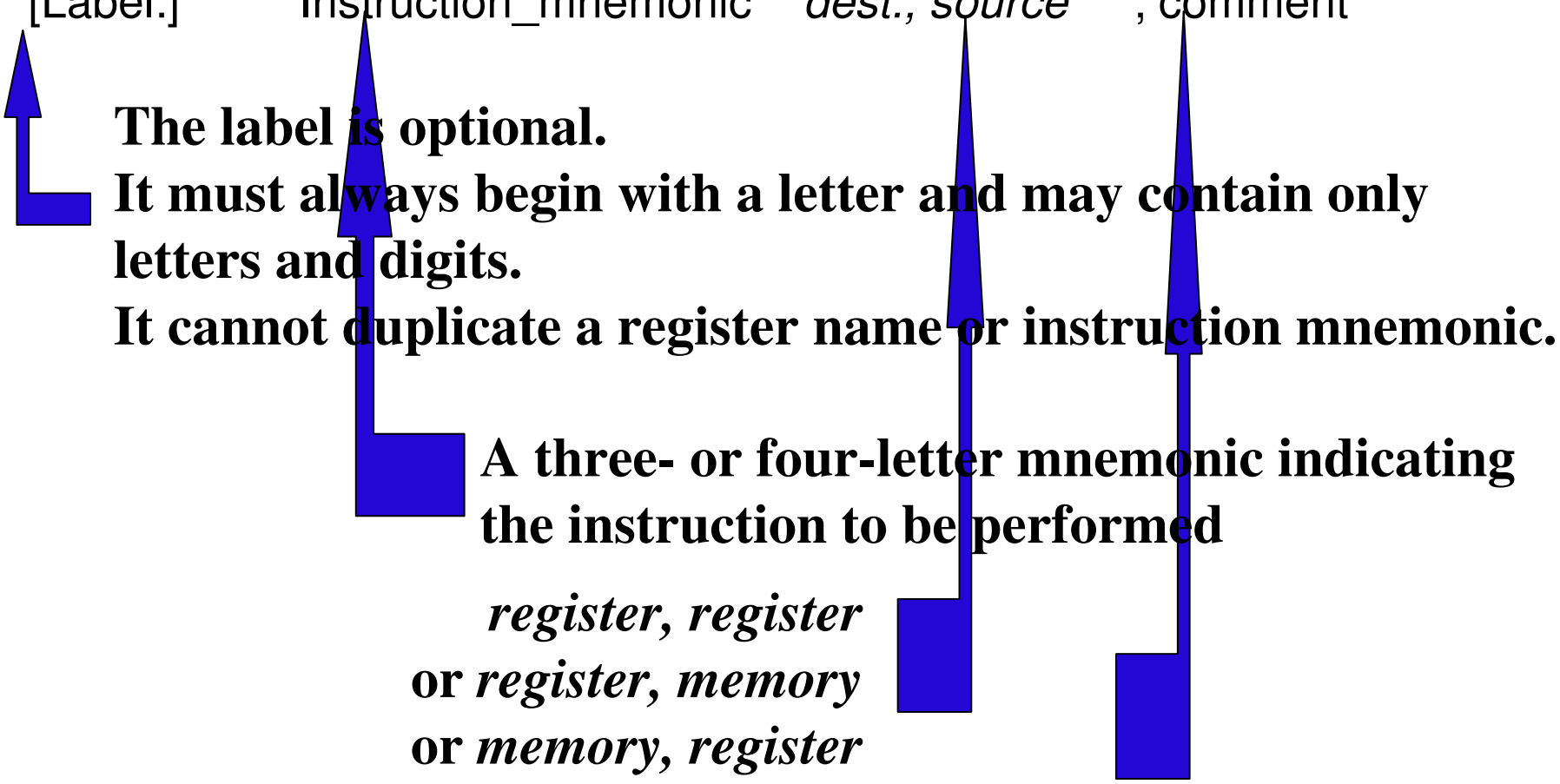




Set	*Clear*
OV = Overflow	NV = No Overflow
DN = Direction Down	UP = Direction Up
EI = Interrupts Enabled	DI = Interrupts Disabled
NG = Sign Flag negative	PL = Sign Flag positive
ZR = Zero	NZ = Not Zero
AC = Auxiliary Carry	NA = No Auxiliary Carry
PO = Odd Parity	PE = Even Parity
CY = Carry	NC = No Carry

8088 instruction

- [Label:] Instruction_mnemonic *dest., source* ; comment



The label is optional.
It must always begin with a letter and may contain only letters and digits.
It cannot duplicate a register name or instruction mnemonic.

A three- or four-letter mnemonic indicating the instruction to be performed

register, register
or *register, memory*
or *memory, register*
NOT *memory, memory*

the assembler ignores everything after the semicolon

Labels

- Act as place markers
  - marks the address (offset) of code and data
- Code label
  - target of jump and loop instructions
  - example: **S1:** (followed by colon)

Mnemonics and Operands

- Instruction Mnemonics

  examples: MOV, ADD, SUB, MUL, INC, DEC

- Operands
  - constant (immediate value)
  - constant expression
  - register
  - memory (data label)

Comments

- Comments are important
  - explain the program's purpose
  - when it was written, and by whom
  - revision information
  - tricky coding techniques
  - application-specific explanations
- Single-line comments
  - begin with semicolon (;)

Example of assembly program

```
;THE FORM OF AN ASSEMBLY Language PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
        .MODEL SMALL
        .STACK 64
        .DATA
DATA1   DB 52H
DATA2   DB 29H
SUM     DB ?
        .CODE
MAIN    PROC FAR                ;this is the program entry point
        MOV AX, @DATA           ;load the data segment address
        MOV DS, AX              ;assign value to DS
        MOV AL, DATA1           ;get the first operand
        MOV BL, DATA2           ;get the second operand
        ADD AL, BL              ;add the operands
        MOV SUM, AL             ;store the result in location
SUM
        MOV AH, 4CH             ;set up to return to DOS
        INT 21 H
        MAIN ENDP
        END MAIN                ;this is the program exit point
```

Example 2

```
title Hello World Program          (hello.asm)

; This program displays "Hello, world!"
.model small
.stack 100h
.data
message db "Hello, world!",0dh,0ah,'$'

.code
main proc
    mov  ax,@data
    mov  ds,ax

    mov  ah,9
    mov  dx,offset message
    int  21h

    mov  ax,4C00h
    int  21h
main endp
end main
```

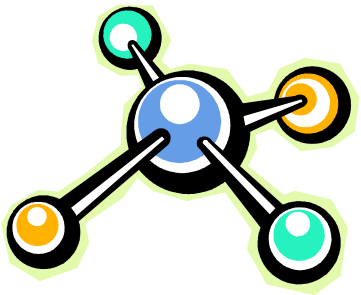
Program usage no more than 64K
data and 64K code
Stack=256bytes
.data=where variables are stored

.code – beginning of the code segment
Beginning of procedure called main
@=address of data

9=msdos function to display a string

Halt the program to return to O/S
End of procedure
End of program

Model Definition (1)



- **choose the size of the memory model.**
- **memory model consists of**
  - **SMALL,**
  - **MEDIUM,**
  - **COMPACT,**
  - **LARGE**
  - **HUGE**
  - **TINY**

Model Definition (2)

- .MODEL SMALL
  - Most widely used memory models for
  - the small model uses a maximum of 64K bytes for code and another 64K bytes for data.
- .MODEL MEDIUM
  - the data must fit into 64K bytes
  - the code can exceed 64K bytes of memory
- .MODEL COMPACT
  - the data can exceed 64K bytes
  - but the code cannot exceed 64K bytes
- .MODEL LARGE
  - both data and code can exceed 64K
  - but no single set of data should exceed 64K
- .MODEL HUGE
  - both code and data can exceed 64K
  - data items (such as arrays) can exceed 64K
- .MODEL TINY
  - used with COM files in which data and code must fit into 64K bytes

Segment Definition

- **Can write an Assembly language program that uses only one segment, but normally a program consists of at least three segments:**
- **.STACK**
  - **marks the beginning of the stack segment**
  - **stack segment defines storage for the stack**
  - **E.g. .STACK 64 ; reserves 64 bytes of memory for the stack**
- **.DATA**
  - **marks the beginning of the data segment**
  - **data segment defines the data that the program will use**
- **.CODE**
  - **marks the beginning of the code segment**
  - **code segment contains the Assembly language instructions**

Assembler Directives (1)

- instructions to the assembler instead of instructions to be executed at run-time.
- The most common used directives are the DB and DW commands to reserve memory.
- E.g.

```
Reply     dB  'Press any key to continue'
```

variable name — assembler directive — ASCII string

Assembler Directives (2)

- **ORG (origin)**
  - indicates the beginning of an offset address.
  - any code and data that follows starts at the new address
- **DB (define byte)**
  - allocates memory in byte-sized units
  - follow the directive with the value you wish to store
  - decimal, binary, hex, ASCII, or undefined
- **DUP (duplicate)**
  - duplicates a given number or character
  - precede with number of duplicates, follow (in brackets) with the value
- **DW (define word)**
  - similar to DB, but allocates memory in word-sized units
  - best to store words at *even* memory addresses (optimal for 8086)
- **EQU (equate)**
  - defines constants
  - does *not* set aside storage for the data

MORE! Refer page 308-319 The 8088 and 8086 Microprocessors by Walter A.Triebel and Avtar Singh

Assembler Directives (3)

- **E.g.**
  - data_1 DB 10 ; store decimal value
    data_2 DB 6CH ; store hex value
    data_3 DB 'M' ; store single character G as ASCII code
    ORG 10h ; set offset to 10 hex
    data_4 DB "Computer" ; stores 8 ASCII characters
    ORG 200h
    data_5 DB 10 DUP(0) ; fill 10 bytes with value 0
    data_6 DB 6 DUP (?) ; set aside 6 bytes with undefined values
    data_7 DW 320 ; store decimal word

- To use the above values:
  - MOV AL, data_1 ; AL is now 14
    MOV DI, offset data_4 ; DI is now 10h (the offset) of "Computer"
    MOV data_5, BL ; the value of BL is saved to memory

Assembler Directives-PROC (*function*)

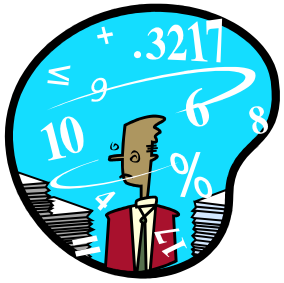
- The first line of the segment after the .CODE directive is the PROC
  - the *procedure* is a group of instructions designed to accomplish a specific function.
- A code segment may consist of only one procedure, but usually is organized into several small procedures in order to make the program more structured.
- Every procedure must have :
  - a name defined by the PROC directive,
  - followed by the assembly language instructions
  - and closed by the ENDP directive.
  - The PROC and ENDP statements must have the same label.
- The PROC directive may have the option FAR (both IP and CS is saved) or NEAR (default- IP is saved).

Summarize of directive

Directive	Description
end	End of program assembly
endp	End of procedure
page	Set a page format for the listing file
proc	Begin procedure
title	Title of the listing file
.code	Mark the beginning of the code segment
.data	Mark the beginning of the data segment
.model	Specify the program's memory model
.stack	Set the size of the stack segment

Mnemonic	Description	Bytes	Attribute
DB	Define byte	1	Byte
DW	Define word	2	Word
DD	Define doubleword	4	Doubleword
DF, DP	Define far pointer	6	Far pointer
DQ	Define quadword	8	Quadword
DT	Define tenbytes	10	Tenbyte

Number Format



- Hexadecimal numbers ---- character h or H
  - E.g. MOV AL, 3Ah
- Must begin with a number else use zero
  - E.g. MOV AL, 0ffh
- Q for octal
- xB for binary

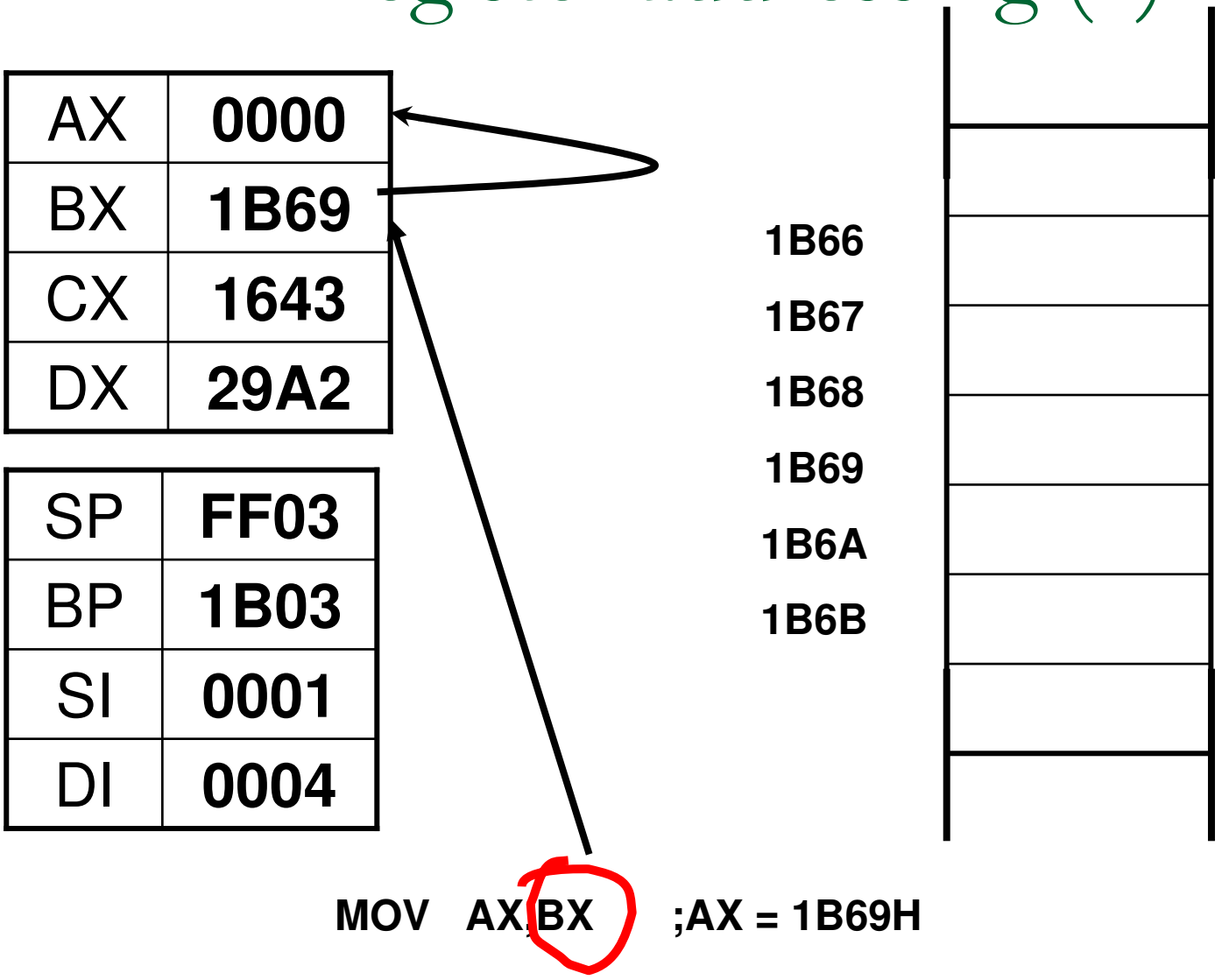
Addressing modes

- **Addressing modes tells how we can determine the exact location of the operand we want.**
- **Implied** - the data value/data address is implicitly associated with the instruction.
- **Register** - references the data in a register or in a register pair.
- **Immediate** - the data is provided in the instruction.
- **Direct** - the instruction operand specifies the memory address where data is located.
- **Register indirect** - instruction specifies a register containing an address, where data is located. This addressing mode works with SI, DI, BX and BP registers. **Based** - 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP), the resulting value is a pointer to location where data resides.
- **Indexed** - 8-bit or 16-bit instruction operand is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.
- **Based Indexed** - the contents of a base register (BX or BP) is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.
- **Based Indexed with displacement** - 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI), the resulting value is a pointer to location where data resides.

Register addressing (1)

- Memory is not accessed when this mode is executed.
- Fast
- Source and destination must have the same size
- e.g. ADD AL,BL

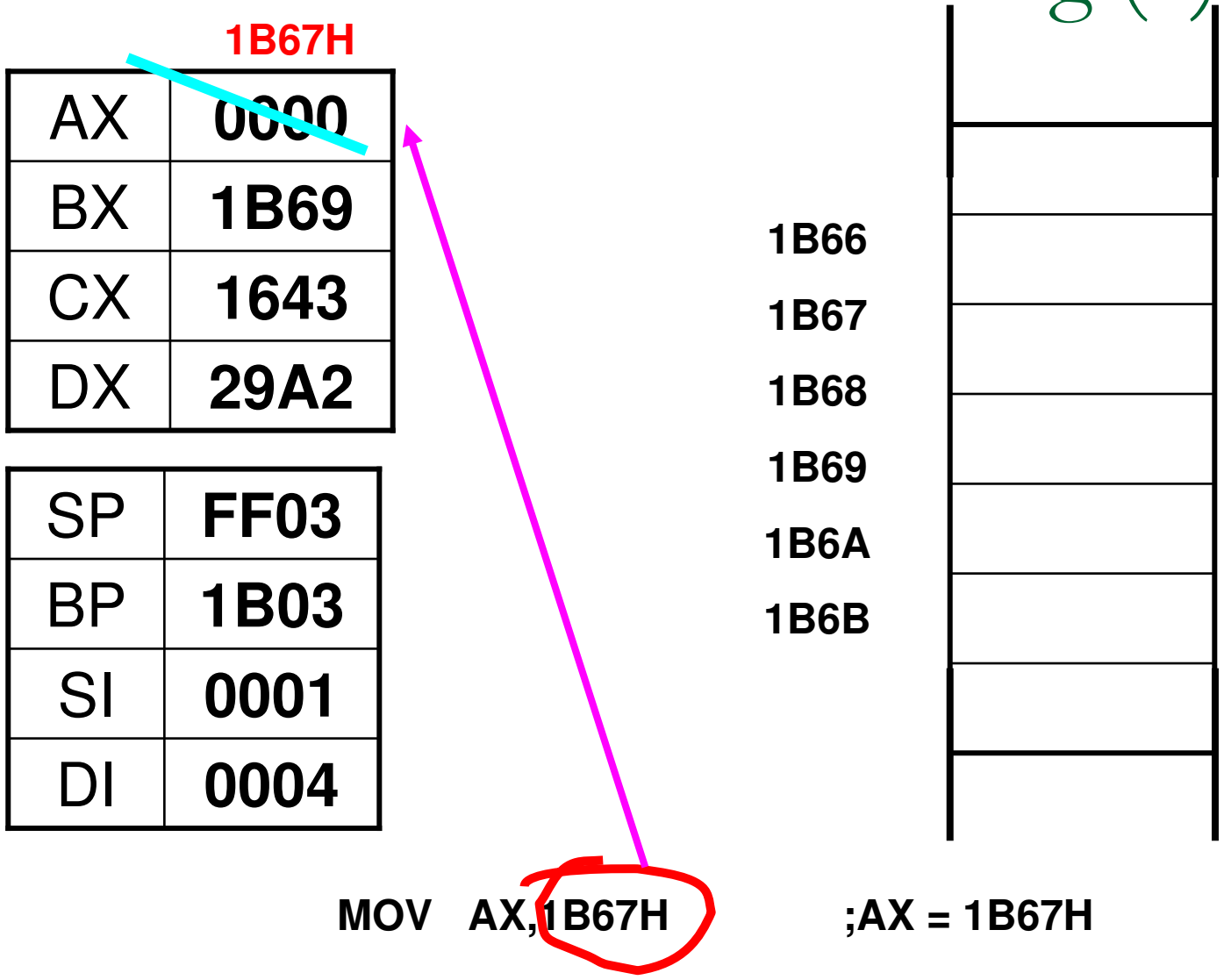
Register addressing (2)



Immediate addressing (1)

- The source operand is a constant
- Can be used to load information into register except segment register and flag register
- E.g.
  - MOV AX, 567 ; load decimal value 567 into AX
- The data must be first moved to a register then to the segment register.
  - MOV AX, 2560H
  - MOV DS,AX
  - MOV DS,2560H ; ------ERROR!

Immediate addressing (2)



Direct addressing (1)

- The effective address is formed by the addition of a segment register and a displacement value that is coded directly into the instruction.
- This address is the offset address
- E.g.
  - MOV DL, [1234H] ;move contents of DS:1234H into DL
- The physical address is calculated by combining the contents of offset location 1234 with DS

How?

Let's said DS=1234H

```
MOV   AL,24H
MOV   [2345H],AL
```

What is the physical address of the memory location and its contents after the above execution?

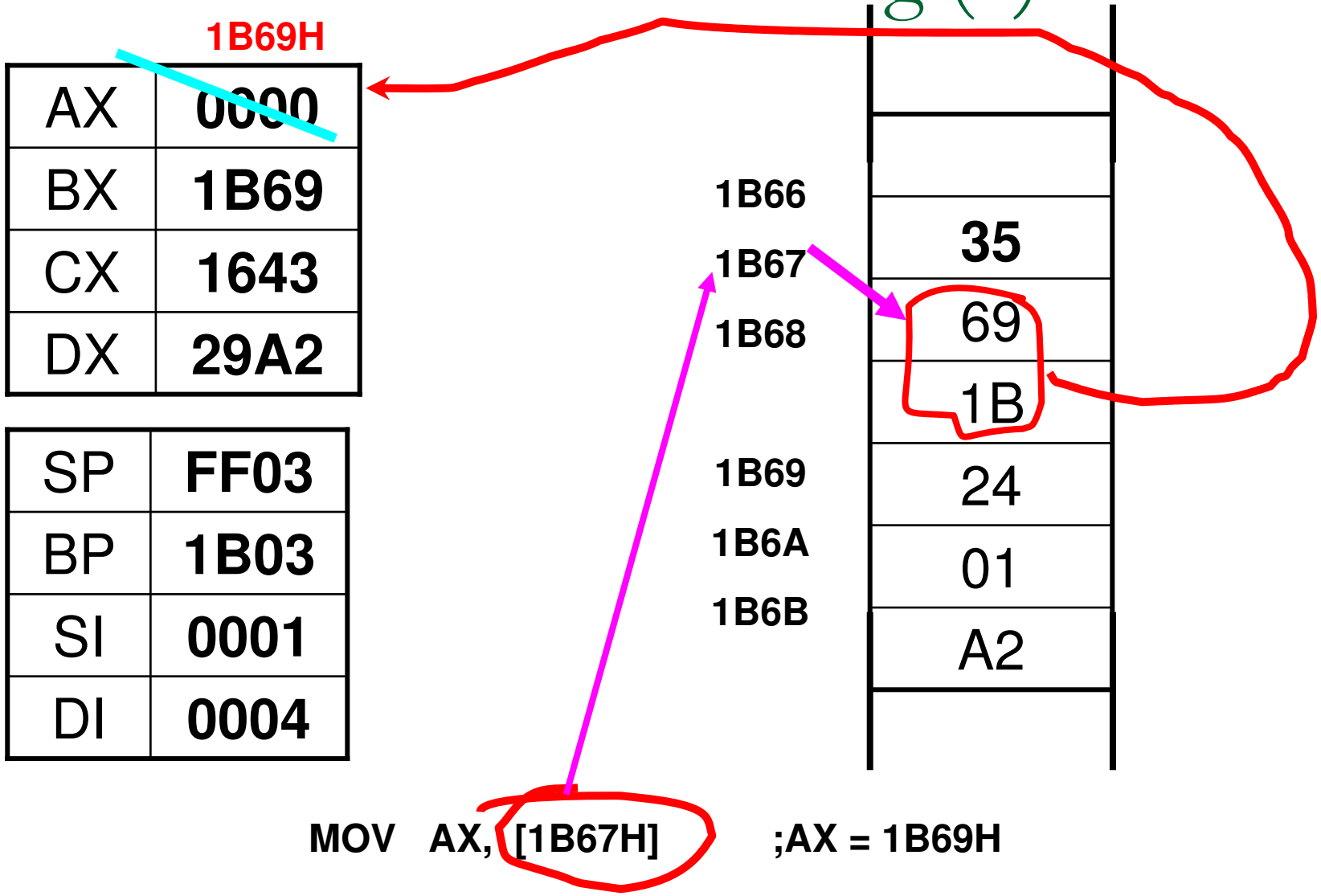
AL=24H

DS:2345 which is 1234H:2345H

12340H+2345H = 14685H = physical address

After the execution the memory location with address 14685H will contain the value 24H

Direct addressing (2)



Register indirect (1)

- A choice of four registers (BX,BP,SI,DI) to use within the square brackets to specify a memory location.
- The operand is held by these register.
- They must combined with DS in order to generate the 20-bit physical address.

- E.g. MOV [DI],AH

```
;move the contents of AH into DS:DI
MOV  DL,[SI]
;the contents of DS:SI into DL
MOV  AL,[BX]
;the contents of the memory location pointed by DS:BX into AL
```

HOW?

- DS=1234H, SI=2345H, and AX=17ABH.

  MOV [SI],AX

  is executed. What is the contents of the memory location?

Contents of AX are moved into memory locations with logical address DS:SI amd DS:SI+1

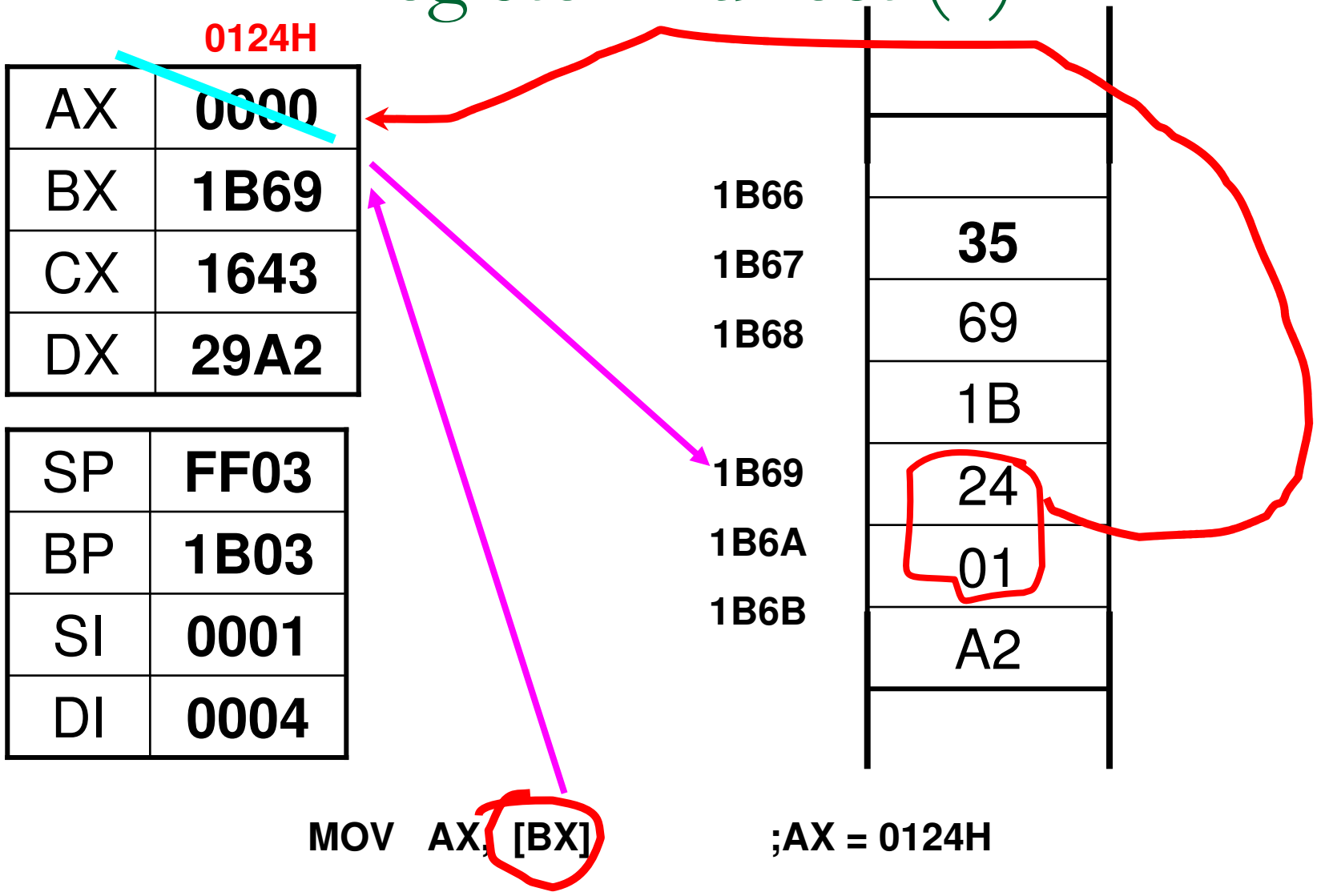
Therefore the physical address starts at DS + SI = 14685H

According to little endian (early lecturers)

Low address 14685H contains ABH, the low byte

High address 14686H contains 17H, the high byte

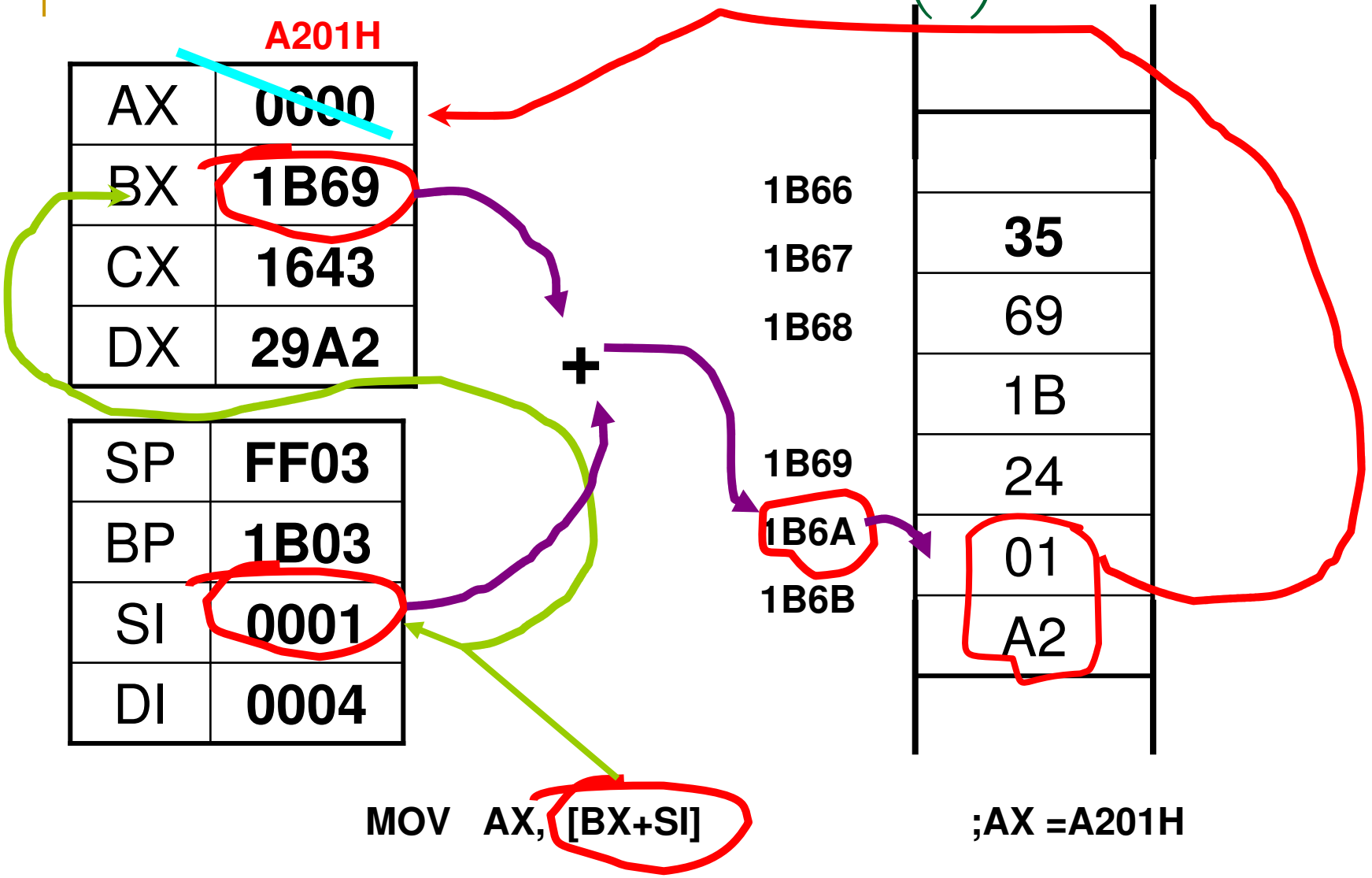
Register indirect (2)



Base relative (1)

- Uses one of the two base registers (BX,BP) as the pointer to the desired memory location.
- Similar to register indirect addressing, the only difference is that an 8 or 16-bits offset may be included
- Physical address are DS for BX ad SS for BP
- Offset is interpreted as a signed 2's complement number 8-bits (-128 -> 127) or 16-bits (-32768 -> +32767)
- So have the ability to point forward or backward in memory.
- E.g. MOV CX,[BX]+10 = MOV CX,[BX+10]
  = MOV CX,10[BX]
  - ;move DS:BX+10 and DS:BX+10+1 into CL and CH
  - ;Physical address = DS*10 + BX+10

Base relative (2)



Index Relative (1)

- Similar to based relative addressing
- Except that registers DI and SI hold the offset address
- E.g. MOV DX,[SI]+3
  - ;Physical address= DS*10 + SI +3

  MOV CL,[DI]+6

  ;Physical address= DS*10 + DI +6

How?

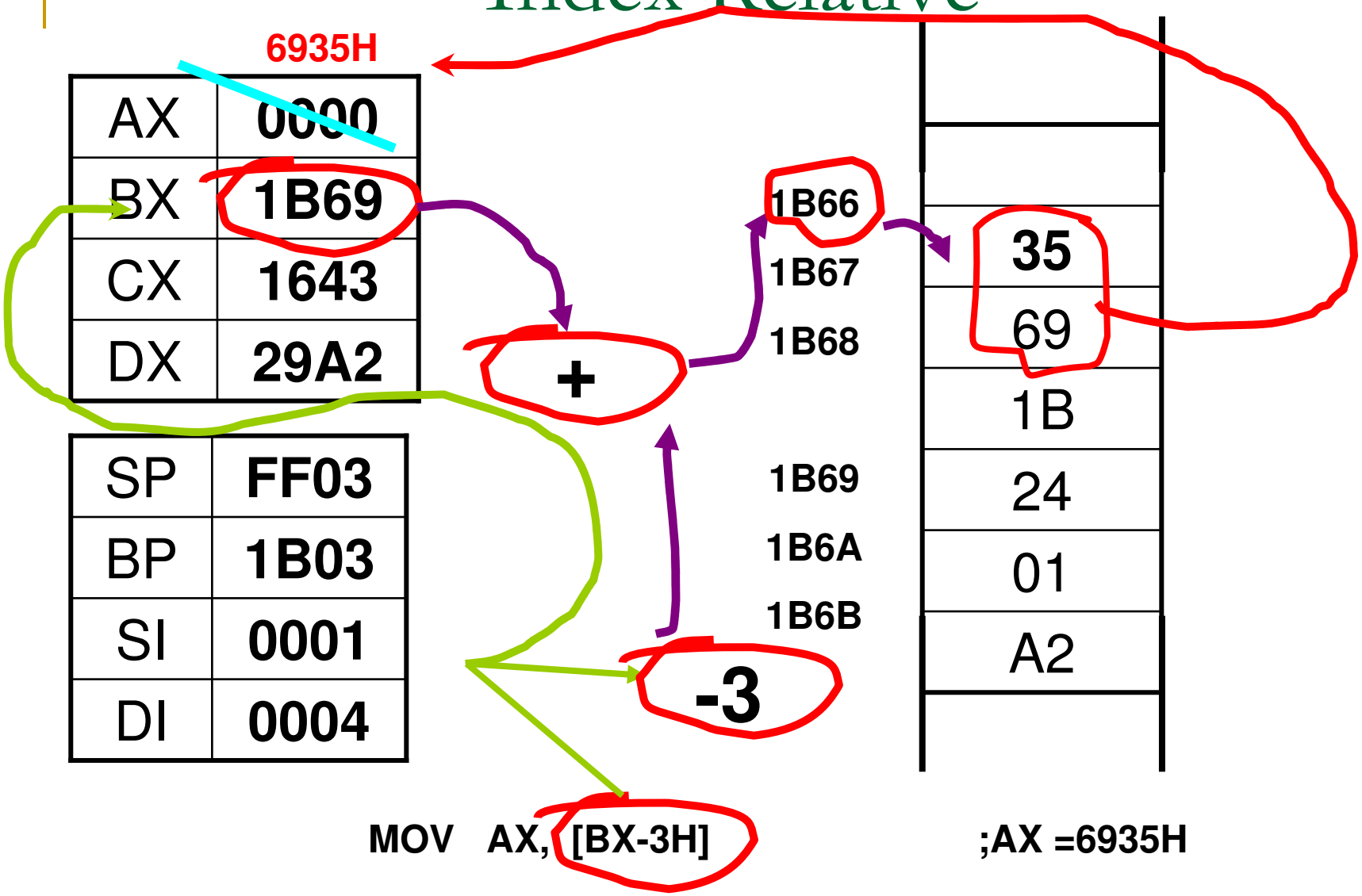
- DS = 4500, SS = 2000, BX = 2100, SI = 1486, DI = 8500, BP= 7814, and AX = 2512. What are their physical memory location for the following instruction to be executed?

1. MOV [BX]+20, AX
2. MOV [SI]+10, AX
3. MOV [DI]+4, AX
4. MOV [BP]+12, AX

PA = segment reg. *10 + (offset reg.) + displacement

1. DS:BX+20 45000 + 2100 + 20 = location 47120 = (12) and 47121 = (25)
2. DS:SI+10 45000 + 1486 + 10 = location 46496 = (12) and 46497 = (25)
3. DS:DI+4 45000 + 8500 + 4 = location 4D504 = (12) and 4D505 = (25)
4. SS:BP+12 20000 + 7814+ 12 = location 27826 = (12) and 27827 = (25)

Index Relative



Base Index Relative (1)

- Combines based and indexed addressing.
- Contents of both registers are not signed numbers (0 – 65535)
- One base register and one index register are used.
- E.g.

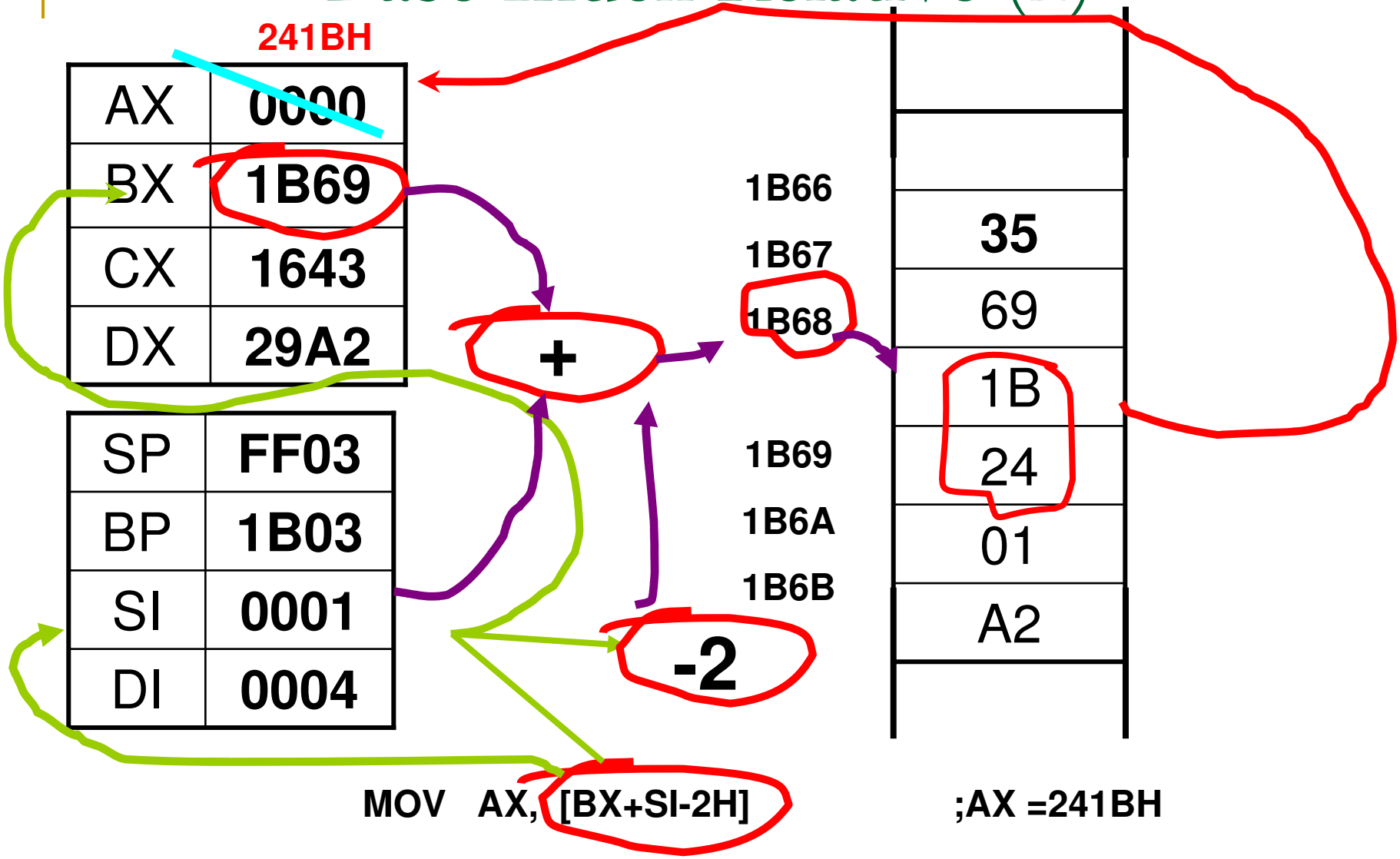
MOV CH,[BX][SI] + 10

- **;Physical address= DS*10 + BX +SI +10**

MOV AH,[BP][SI] + 5

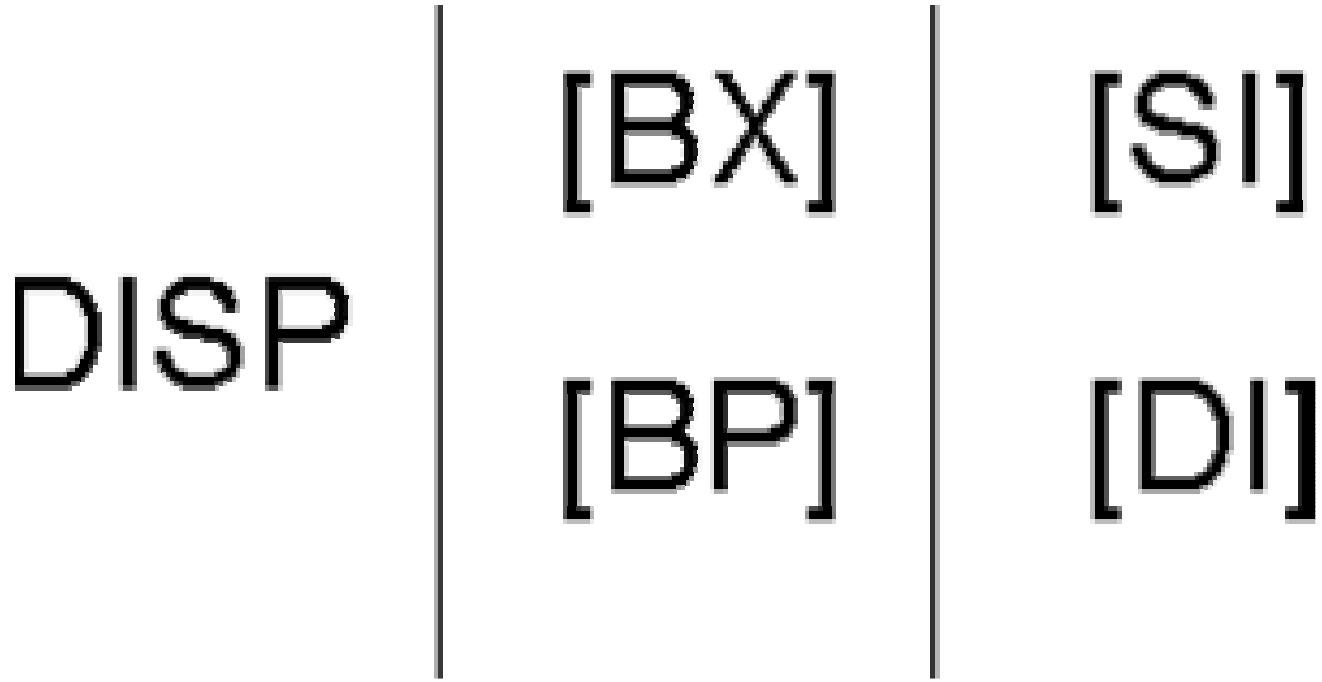
; Physical address= SS*10 + BP +SI + 5

Base Index Relative (2)

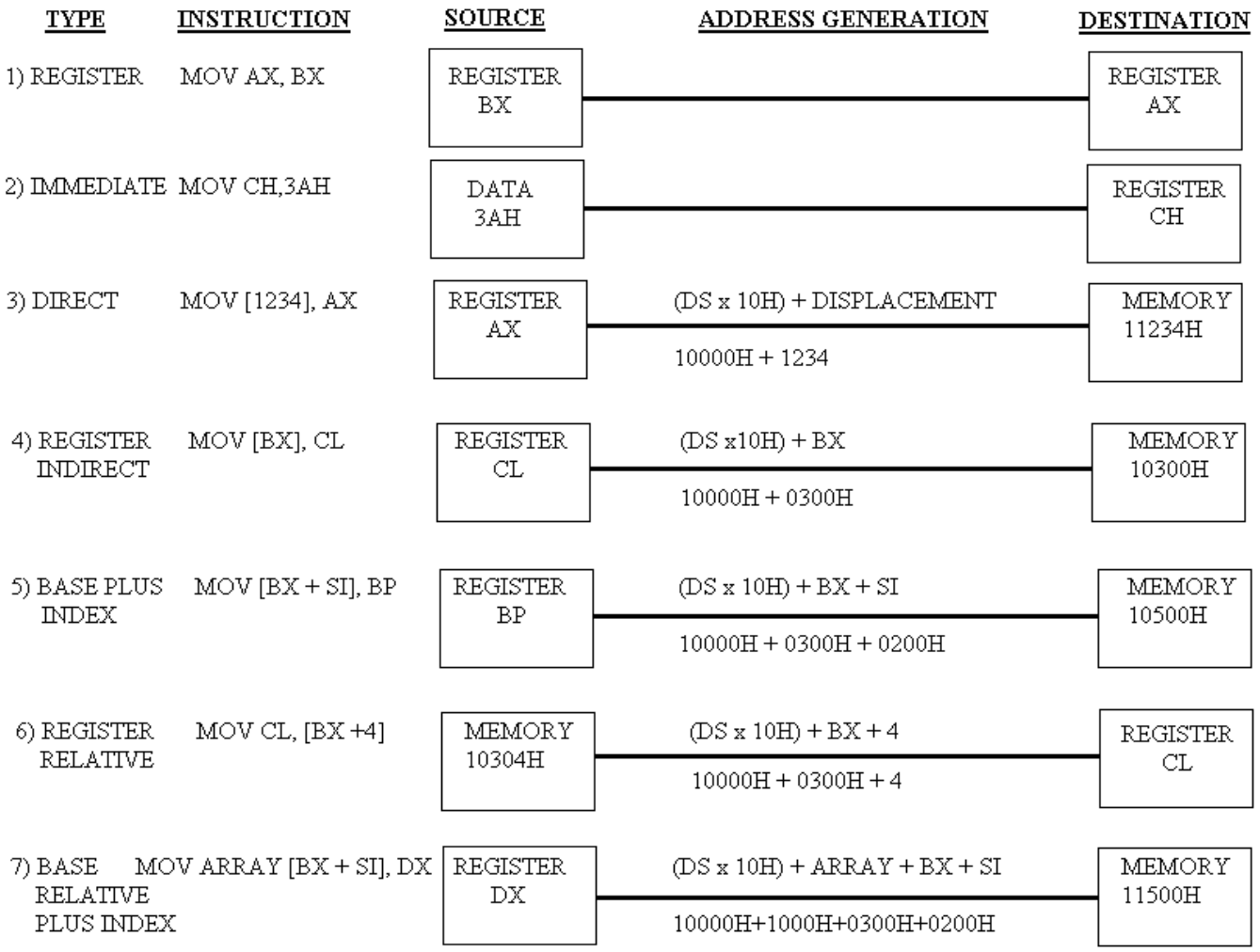


To remember!!

WAKE UP..COPY TIME!



8086 ADDRESS MODES



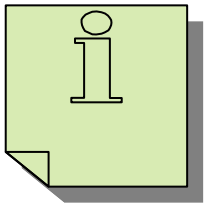
ASSUME: BX = 0300H; SI = 0200H; ARRAY = 1000H; DS = 1000H.

In different name DIFFERENT FORM

WAKE UP! COPY TIME

Port Addressing

- Use of I/O ports for data communication between the CPU and outside world.
- One way to get data is read it from input port
  - IN AL,DX
  - ; 8-bits are input to AL from I/O port DX
- To write data to output port
  - OUT DX,AL
  - ;8-bits are output from AL to I/O port DX



INFO!

- **MOV AH,[BP][SI] + 5**

  ; Physical address= SS*10 + BP +SI + 5

  Equivalent to

- **MOV AH, [BP+SI+5]; or**
- **MOV AH, [SI+BP+5] ;**

- **MOV AX, [SI][DI] + offset is illegal.**

Summary of the Addressing Mode

Addressing Mode	Operand	Default Segment
Register	Reg	None
Immediate	Data	None
Direct	[offset]	DS
Register Indirect	[BX] [SI] [DI]	DS DS DS
Based Relative	[BX]+disp [BP]+disp	DS SS
Indexed Relative	[DI]+disp [SI]+disp	DS DS
Based Indexed Relative	[BX][SI or DI]+disp [BP][SI or DI]+disp	DS SS

Summary of Offset

Type of Memory Reference	Default Segment	Alternate Segment	Offset
Instruction Fetch	CS	none	IP
Stack Operations	SS	none	SP,BP
General Data	DS	CS,ES,SS	BX,address
String Source	DS	CS,ES,SS	SI,DI, addresss
String Destination	ES	None	DI

Segment Override

- The pointer register BX,SI or DI specifies an offset address relative to DS
- For e.g.

MOV BX, [10H] ; uses DS

- It's possible to specify an offset relative to one of the other segment register.

segment_register:[pointer_reg]

- For e.g.

```
MOV BX,ES:[SI]
                ; if SI = 0100H the source address
                  =ES:0100H
```

- Can also be used with based and indexed modes.

Summary of Segment Overrides

Instruction	Segment Used	Default Segment
MOV AX,CS:[BP]	CS:BP	SS:BP
MOV DX,SS:[SI]	SS:SI	DS:SI
MOV AX,DS:[BP]	DS:BP	SS:BP
MOV CX,ES:[BX]+12	ES:BX+12	DS:BX+12
MOV SS,[BX][DI]+32,AX	SS:BX+DI+32	DS:BX+DI+32

Types of 8088/86 instructions

- Can be (roughly) divided into eight different classes:
  - Data moving instructions. ---mov, lea, les , push, pop, pushf, popf
  - Arithmetic - add, subtract, increment, decrement, convert byte/word and compare.---- add, inc sub, dec, cmp, neg, mul, imul, div, idiv
  - Logic - AND, OR, exclusive OR, shift/rotate and test.---and, or, xor, not, shl, shr, rcl, rcr.
  - String manipulation - load, store, move, compare and scan for byte/word. ---movs, stos, lods.
  - Control transfer - conditional, unconditional, call subroutine and return from subroutine. ---jmp, call, ret, conditional jumps
  - Input/Output instructions. ---in, out.
  - Conversions ---cbw, cwd, xlat
  - Other - setting/clearing flag bits, stack operations, software interrupts, etc.

Summary of 8088/86 instructions

Data transfer (14 instructions): MOV, PUSH, POP, XCHG, IN, OUT, XLAT, LEA, LDS, LES, LAHF, SAHF, PUSHF, POPF
Arithmetic (20 instructions): ADD, ADC, INC, AAA, DAA, SUB, SBB, DEC, NEG, CMP, AAS, DAS, MUL, IMUL, AAM, DIV, IDIV, AAD, CBW, CWD
Logic (12 instructions): NOT, SHL/SAL, SHR, SAR, ROL, ROR, RCL, RCR, AND, TEST, OR, XOR
String manipulation (6 instructions): REP, MOVS, CMPS, SCAS, LODS, STOS
Control transfer (26 instructions): CALL, JMP, RET, JE/JZ, JL/JNGE, JLE/JNG, JB/JNAE, JBE/JNA, JP/JPE, JO, JS, JNE/JNZ, JNL/JGE, JNLE/JG, JNB/JAE, JNBE/JA, JNP/JPO, JNO, JNS, LOOP, LOOPZ/LOOPE, LOOPNZ/LOOPNE, JCXZ, INT, INTO, IRET
Process control (12 instructions): CLC, CMC, STC, CLD, STD, CLI, STI, HLT, WAIT, ESC, LOCK, NOP

MOV

- MOV destination,source
  - 8 bit moves
    - MOV CL,55h
    - MOV DL,CL
    - MOV BH,CL
  - 16 bit moves
    - MOV CX,468Fh
    - MOV AX,CX
    - MOV BP,DI
- Data can be moved among all registers but data cannot be moved directly into the segment registers (CS,DS,ES,SS).
- To load as such, first load a value into a non-segment register and then move it to the segment register
  - MOV AX,1234h
    MOV DS,AX

ADD / SUB

- ADD destination,source
- The ADD instruction tells the CPU to add the source and destination operands and put out the results in the destination
  - MOV DH,25H
    ADD DH,34h ; (AL = 59h)
- SUB destination,source
  - MOV DH,60H
    SUB DH,A5h ; (AL =??)

BBH with C=1

ADC/SBB

```
ADC AX, BX      ; AX=1234h, BX=2345h
                ; AX=1234h + 2345h + 1=357Ah,
                ; C=0
```

Analyze the following program:

```
DATA_A    DD  62562FAH
DATA_B    DD  412963BH
RESULT    DD  ?
...       ...  ...
          MOV   AX,WORD PTR DATA_A        ;AX=62FA
          SUB   AX,WORD PTR DATA_B        ;SUB 963B from AX
          MOV   WORD PTR RESULT,AX        ;save the result
          MOV   AX,WORD PTR DATA_A +2     ;AX=0625
          SBB   AX,WORD PTR DATA_B +2     ;SUB 0412 with borrow
          MOV   WORD PTR RESULT+2,AX      ;save the result
```

Solution:

After the SUB, AX = 62FA – 963B = CCBF and the carry flag is set. Since CF = 1, when SBB is executed, AX = 625 – 412 – 1 = 212. Therefore, the value stored in RESULT is 0212CCBF.

AAA/AAS

AAA/AAS is used for adjusting the result of binary addition/subtraction so as to facilitate ASCII arithmetic.

Usage: `ADD AL,source`
`AAA`

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

If numeric data are represented by ASCII codes, using binary arithmetic instruction (e.g. **ADD/SUB**) would not produce desirable results.

For example, if AL = **31H** ("1") and BL = **39H** ("9"),

`ADD AL,BL`

will result in AL = 6AH, which is not the ASCII code of 10.
AAA is used to adjust this result and produce 3130H in AX.

MOV AL,"9"

AL=39H

AL=10

Similarly, **AAS** corrects the subtraction result of two ASCII coded numbers.

DAA/DAS

Addition and subtraction of BCD data cannot be performed by the **ADD/ADC** and **SUB/SBB** instructions.

DAA and **DAS** are used to convert the result to the correct BCD format.

Usage:

```
ADD AL,source
DAA
```

Let AL = 15H and BL = 25H, which correspond to BCD numbers 15 and 25.

```
ADD AL,BL
```

results in AL = 3AH, which is an illegal expression in BCD.

```
DAA
```

will convert 3AH to 40H.

Add 06 or 60 if lower or upper 4-bits greater than 9

MUL/IMUL(signed)/DIV/IDIV(signed)

```
MUL source
DIV source
```

AL=-1 and CL=-2

MUL CL ; 11111111*11111110=1111110100000010=FD02=AX

IMUL CL ; -1*-2=2

AX= -21H= -33(decimal) and BH=5H =answer -6 remainder -3

IDIV BH ; AX=FDFAh AL=FA= -6 remainder AH=FD = -3

DIV BH ;  should BX=05H and DIV BX give you AX=332C and DX=3 as remainder

CBW/CWD

- CBW (Convert singed byte to signed word) will copy D7 (sign flag) to all bits in AH.
- CWD (convert signed word to singed double word) copies D15 of AX to all bits of the DX register.

```
MOV AX,6E2FH  ; 28,207  = 0110 1110 0010 1111
MOV CX,13D4H  ; +5,076  = 0001 0011 1101 0100
ADD AX,CX     ;=33,283 = 1000 0010 0000 0011=-32,523
```

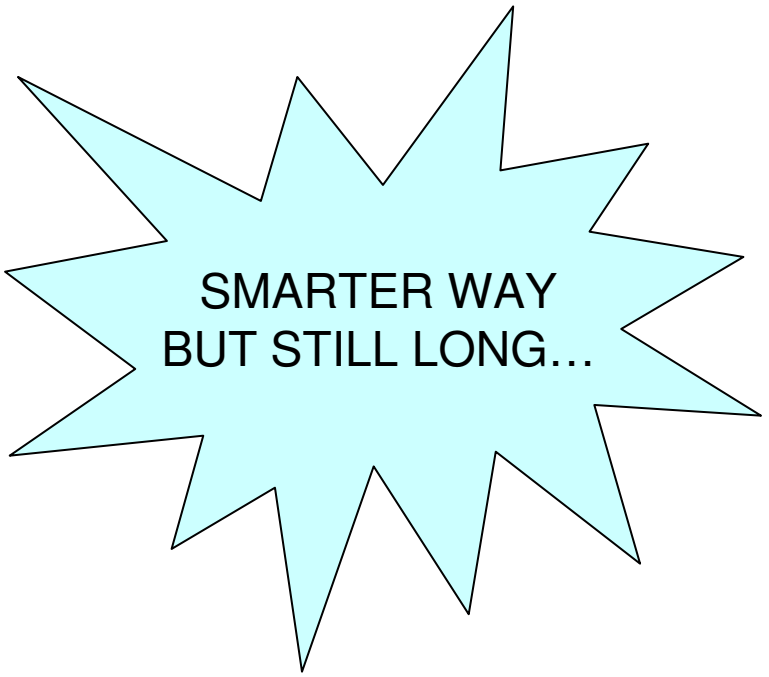
Smart programming (1)

- A program to add 5 bytes of data 25h,12h,15h,1Fh, and 2Bh.
  - ```
    MOV AL,00h
    ADD AL, 25h
    ADD AL, 12h
    ADD AL,15h
    ADD AL,1Fh
    ADD AL,2Bh
    ```
- Data and code are mixed in the instructions here
- The problem with it is if the data changes, the code must be searched for every place the data is included and data retyped.
- It is a good idea then to set aside an area of memory strictly for data
- The data is first placed in the memory locations
  - DS:0200 = 25h
    DS:0201 = 12h
    DS:0202 = 15h
    DS:0203 = 1Fh
    DS:0204 = 2Bh
  - ```
    MOV AL,0
    ADD AL,[0200] ; add the contents of DS:0200 to AL
    ADD AL,[0201]
    ADD AL,[0202]
    ADD AL,[0203]
    ADD AL,[0204]
    ```
- If the data is stored at a different offset address, say 0100 h ???

RETYPE THE WHOLE PROG.

Smart programming

- Use BX as a pointer
  - ```
    MOV AL,0
    MOV BX,0200h
    ADD AL,[BX]
    INC BX
    ADD AL,[BX]
    INC BX
    ADD AL,[BX]
    INC BX
    ADD AL,[BX]
    INC BX
    ADD AL,[BX]
    ```

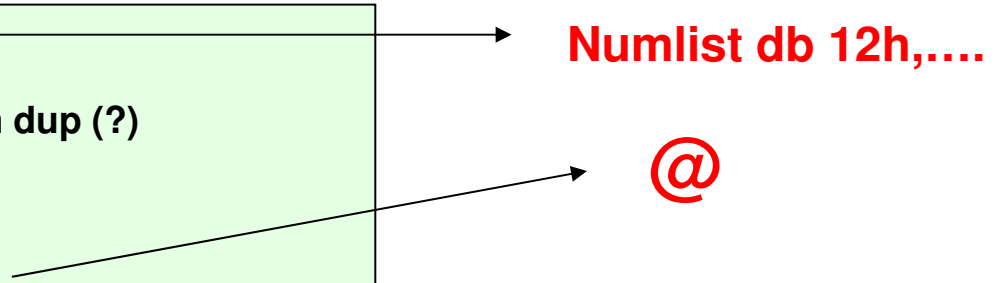


- If the offset address of data is to be changed, only one instructions will need to be modified
    ```

BEST

- Conditional jump instruction will be used to implement the counter checking logic.
  -

```
data segment
count      equ 5
result     dw   01h dup (?)
data ends
code segment
org 100h
start:     mov AX,data
           mov DS,AX
           xor   AX,AX
           xor   BX,BX
           mov SI,offset numlist
again:     mov BL,[SI]
           add AX,BX
           inc  SI
           dec CX
           jnz  again
           mov DI,offset result
           mov [DI],AX
           code ENDS
end        start
```



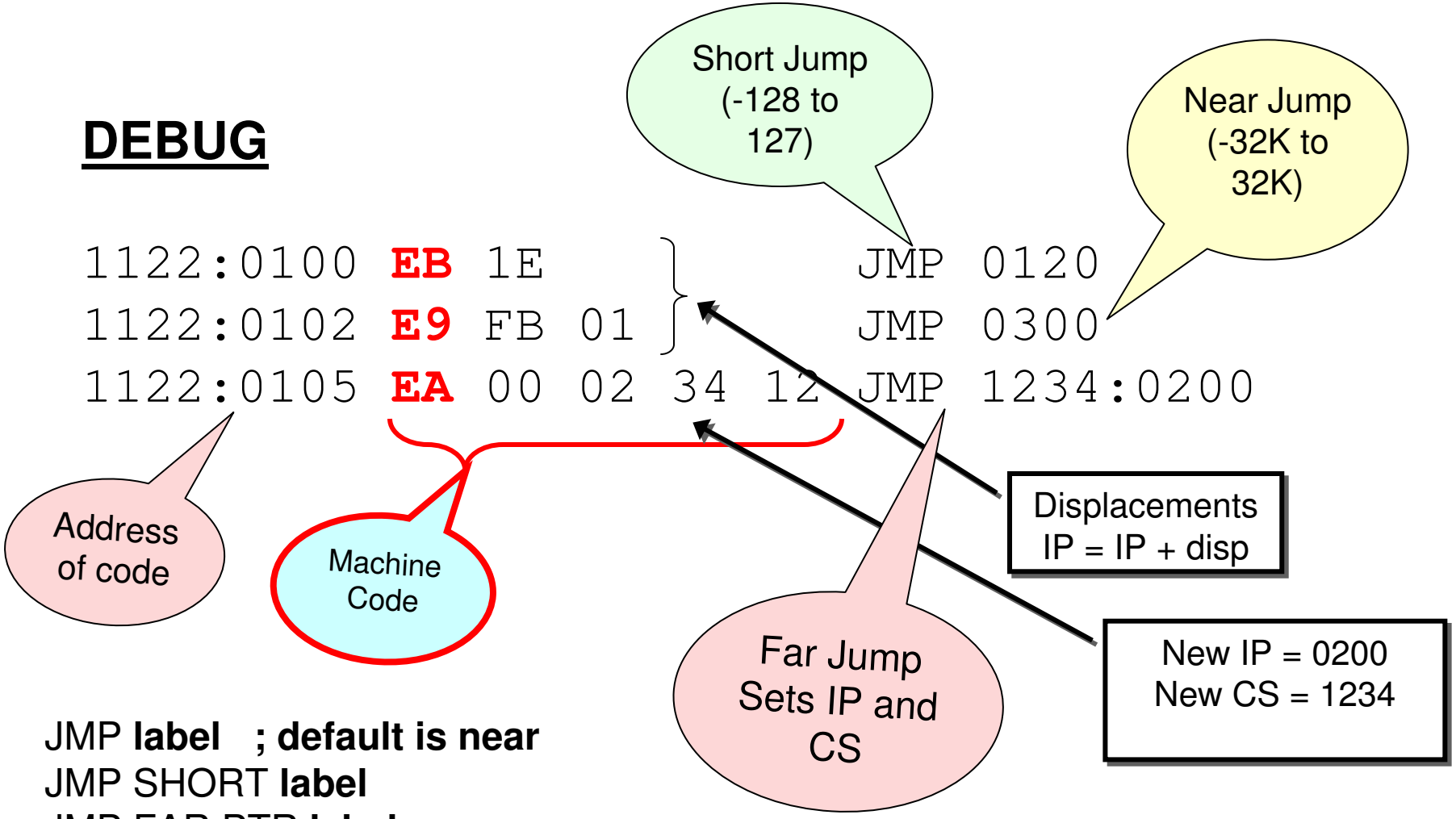
Decision Instructions for Signed and Unsigned Integers

Mnemonic	Condition
	Signed Operations
JG/JNLE	Greater/not less or equal ((SF ⊕ OF) + ZF) = 0
JGE/JNL	Greater or equal/not less (SF ⊕ OF) = 0
JL/JNGE	Less/not greater or equal (SF ⊕ OF) = 1
JLE/JNG	Less or equal/not greater ((SF ⊕ OF) + ZF) = 1
JO	Overflow (OF = 1)
JS	Sign (SF = 1)
JNO	Not overflow (OF = 0)
JNS	Not sign (SF = 0)
	Unsigned Operations
JA/JNBE	Above/not below or equal (CF ⊕ ZF) = 0
JAE/JNB	Above or equal/not below (CF = 0)
JB/JNAE	Below/not above or equal (CF = 1)
JBE/JNA	Below or equal/not above (CF ⊕ ZF) = 1
	Either
JC	Carry (CF = 1)
JE/JZ	Equal/zero (ZF = 1)
JP/JPE	Parity/parity even (PF = 1)
JNC	Not carry (CF = 0)
JNE/JNZ	Not equal/not zero (ZF = 0)
JNP/JPO	Not parity/parity odd (PF = 0)

Jump instruction

- Divided into:
  - **Unconditional jump**
  - **Conditional jump**
- Two cases of jump action:
  - **SHORT** jump --- target location within –128 to +127 from the current location.
  - **Intra-segment** or **NEAR** jump --- only IP is changed; displacement for direct jump is up to 32K.
  - **Inter-segment** or **FAR** jump --- both CS and IP are changed;
- Useful in decision and repetition of a specific portion of the program.

Short/Near/Far



JMP **label** ; **default is near**
JMP SHORT **label**
JMP FAR PTR **label**

Decision instruction

- If the condition by Jxx is *TRUE*,
  then program execution Jumps to the near destination

  If the condition by Jxx is *FALSE*,
  then program execution continues with the next instruction
- Consider

```
        JC   S1
S2:
```

goes to label S1 if the C flag is set, C=1
goes to S2 if the C flag is clear, C=0

E.g.1

CMP *dest, src* ; *dest-src* to set flags
; result is NOT stored

The compare instruction can be used to set flags without modifying the *destination* operand.

DEBUG

```
1234:0100  cmp ax, [800]
1234:0104  jle 109
1234:0106  jmp 200
1234:0109
```

.ASM

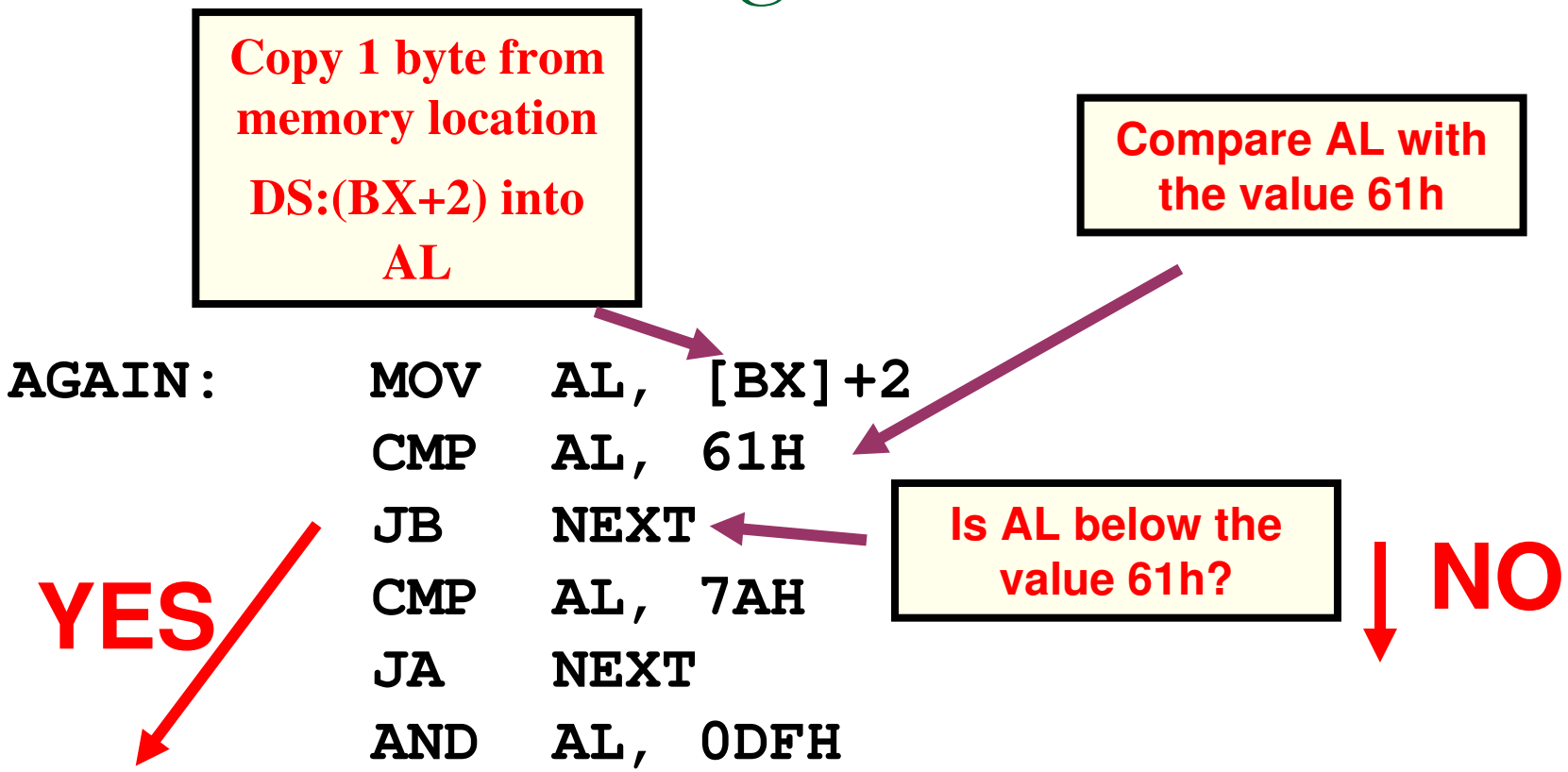
```
        cmp ax, [800h]
        jle loc
        jmp further_loc
loc:
```

Jump to offset 200h only if AX > DS:[800h]

XOR A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

JLE/JNG	Less or equal/not greater	((SF ⊕ OF) + ZF) = 1

E.g. 2



JB/JNAE	Below/not above or equal (CF = 1)
JA/JNBE	Above/not below or equal (CF ⊕ ZF) = 0

E.g.3

```
AGAIN:    MOV   AL, [BX]+2
          CMP   AL, 61H
          JB    NEXT
          CMP   AL, 7AH
          JA    NEXT
          AND   AL, 0DFH
NEXT:     MOV   [SI], AL
```

YES

NO

Compare AL with the value 7Ah

Is AL above the value 7Ah?

JB/JNAE	Below/not above or equal (CF = 1)
JA/JNBE	Above/not below or equal (CF ⊕ ZF) = 0

E.g.4

```
AGAIN:    MOV   AL, [BX]+2
          CMP   AL, 61H
          JB    NEXT
          CMP   AL, 7AH
          JA    NEXT
          AND   AL, 0DFH
NEXT:     MOV   [SI], AL
```

Bitwise AND register AL with 0DFH (1101 1111b)

Move to memory location DS:SI the contents of AL

JB/JNAE	Below/not above or equal (CF = 1)
JA/JNBE	Above/not below or equal (CF ⊕ ZF) = 0

E.g.5

```
al = *(bx + 2);

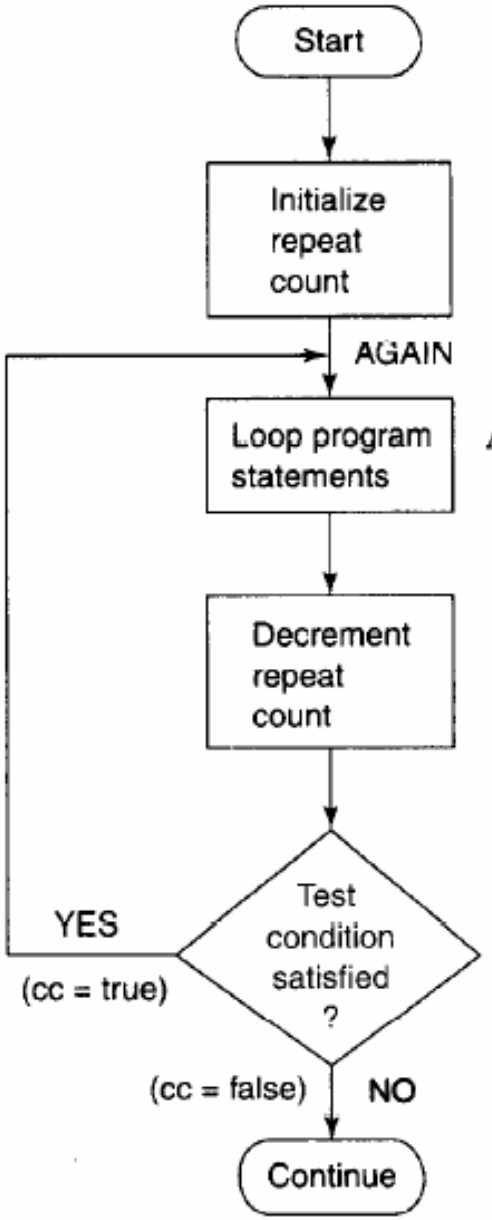
if ( al >= 0x61 && al <= 0x7A ) {
      al = al & 0xDF;
}

*si = al;
```

```
0005  8A 47 02     AGAIN:    MOV  AL, [BX]+2
0008  3C 61                 CMP  AL, 61H
000A  72 06                 JB   NEXT
000C  3C 7A                 CMP  AL, 7AH
000E  77 02                 JA   NEXT
0010  24 DF                 AND  AL, 0DFH
0012  88 04       NEXT:     MOV  [SI], AL
```

JB/JNAE	Below/not above or equal (CF = 1)
JA/JNBE	Above/not below or equal (CF ⊕ ZF) = 0

Repeat-Until program and instruction sequence




```
AGAIN:  MOV   CL,COUNT   ;Set loop repeat count
        ---   ---        ;1st instruction of loop
        ---   ---        ;2nd instruction of loop
        .     .                .
        .     .                .
        .     .                .
        ---   ---        ;nth instruction of loop
        DEC   CL         ;Decrement repeat count by 1
        JNZ   AGAIN      ;Repeat from AGAIN if (CL) ≠ 00H or (ZF) = 0
        ---   ---        ;First instruction executed after the loop is
                         ;complete, (CL) = 00H, (ZF) = 1
```

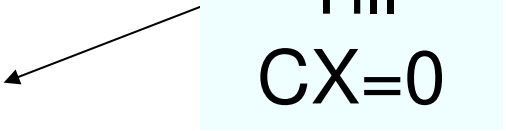
Iteration

Using condition

```
Delay:    PUSH  CX
          MOV   CX, 2000
Here:     DEC   CX
          JNZ   Here
          POP   CX
```

Using LOOP instruction

```
Delay:    PUSH  CX
          MOV   CX, 2000
Here:     LOOP  Here
          POP   CX
```



E.g. using JCXZ

Jump if register CX is zero

```
Delay_1:   PUSH CX
           MOV CX,2000
Here:      DEC CX
           JNZ Here
           POP CX
```

=

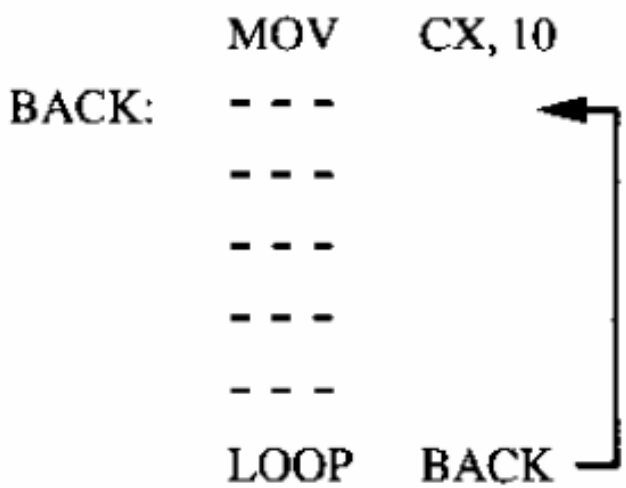
```
Delay_1:     PUSH CX
             MOV CX, 2000
Here:        DEC CX
             JCXZ Done
             JMP Here
Done:        POP CX
```

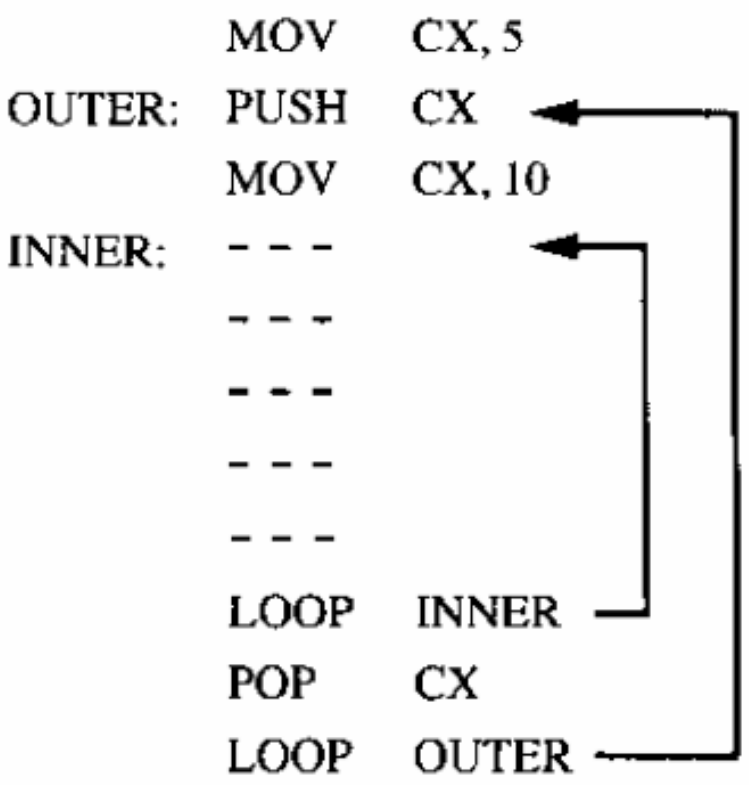
LOOPE / LOOPZ LOOPNE / LOOPNZ

- A variation on LOOP
  - Decrements CX
  - Jumps if CX is not 0 and ZF is set
- Read as:
  - Loop while equal
  - Loop while zero

- Another variation on LOOP
  - Decrements CX
  - Jumps if CX is not 0 and ZF is clear
- Read as:
  - Loop while not equal
  - Loop while not zero

Single and nested loop





Example

Assume that the daily temperatures for the last 30 days have been stored starting at memory location 1200H. Find first day that had a 20-degree temperature.

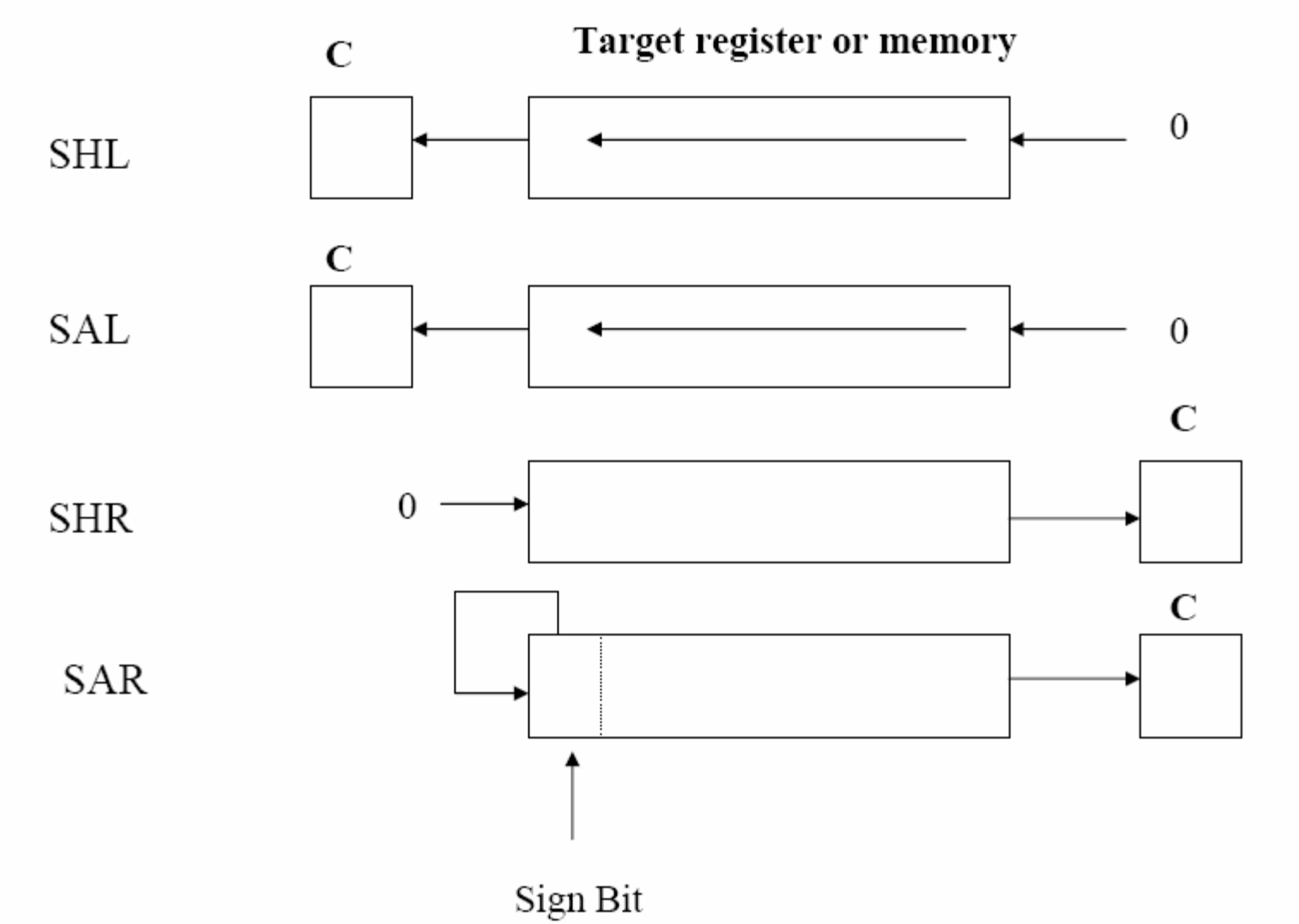
```
        MOV    CX, 30          ; Set up counter
        MOV    DI, 1200H       ; Set up the pointer
AGAIN:  CMP    [DI], 20        ; Check temperature
        INC    DI              ; Does not affect flags
        LOOPNE AGAIN
                               ; If ZF is 0, no day was found
```

Example

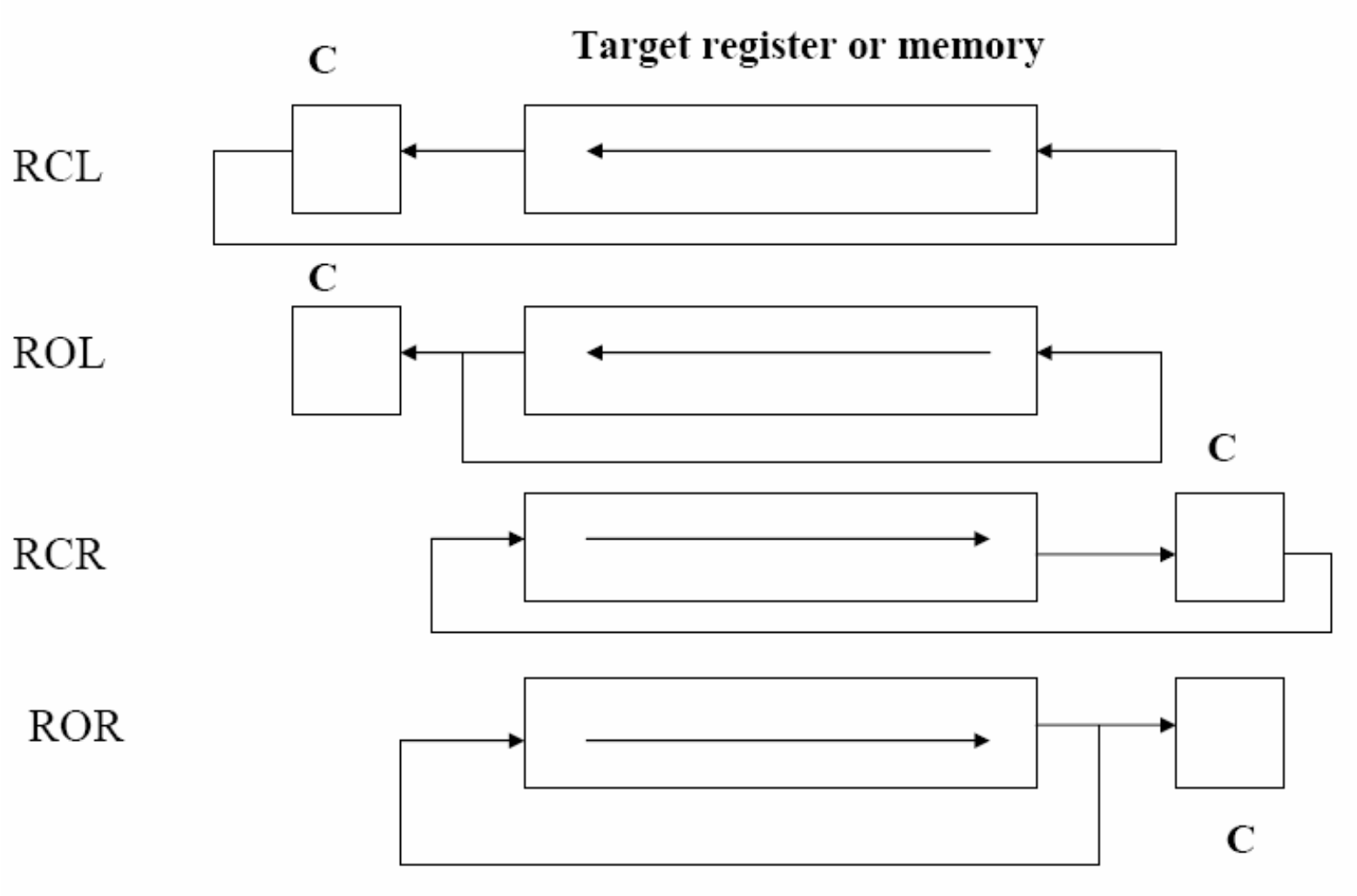
Write a program that calculates the average of five temperatures and writes the result in AX

```
DATA  DB      +13,-10,+19,+14,-18     ;0d,f6,13,0e,ee
      MOV     CX,5                    ;LOAD COUNTER]
      SUB     BX, BX                  ;CLEAR BX,
      MOV     SI, OFFSET DATA         ;SET UP POINTER
BACK: MOV     AL,[SI]                 ;MOVE BYTE INTO AL
      CBW                             ;SIGN EXTEND INTO AX
      ADD     BX, AX                  ;ADD TO BX
      INC     SI                      ;INCREMENT POINTER
      DEC     CX                      ;DECREMENT COUNTER
      JNZ     BACK                    ;LOOP IF NOT FINISHED
      MOV     AL,5                    ;MOVE COUNT TO AL
      CBW                             ;SIGN EXTEND INTO AX
      MOV     CX,AX                   ;SAVE DENOMINATOR IN CX
      MOV     AX,BX                   ;MOVE SUM TO AX
      CWD                             ;SIGN EXTEND THE SUM
      IDIV    CX                      ;FIND THE AVERAGE
```

Shift



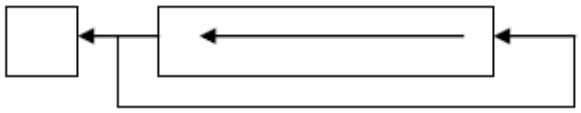
Rotate



Example

Write a program that counts the number of 1's in a byte and writes it into BL

```
DATA1  DB 97              ; 61h
       SUB   BL,BL        ;clear BL to keep the number of 1s
       MOV    DL,8        ;rotate total of 8 times
       MOV    AL,DATA1
AGAIN: ROL    AL,1        ;rotate it once
       JNC    NEXT        ;check for 1
       INC    BL          ;if CF=1 then add one to count
NEXT:  DEC    DL          ;go through this 8 times
       JNZ    AGAIN       ;if not finished go back
       NOP
```



Subroutine handling

- CALL & RET (*return*)
  - Saves information on the stack
  - Procedure must end with a RET
  - Can be NEAR (intra-segment) or FAR (inter-segment)
    - A NEAR CALL must be matched by a NEAR RET

- In a NEAR CALL, the contents of the register IP is pushed onto the stack
- IP is the given a new value, based on the location of the subroutine
- RET pops a value off the stack and into IP
  - Without CALL, RET pops an invalid IP
- In a FAR CALL, both CS and IP are stored on the stack
- A FAR return pops IP and CS off the stack

CALL

These labels do NOT have colons after them.

```
        MOV AL, 200 ;
              X
              X
        CALL SUBR1
        CALL SUBR2

SUBR1:  blah1
        RET

SUBR2:  blah2
        RET
```

```
        CALL   SUBR1

SUBR1 PROC NEAR
        ...             ; your code
        ...
        RET
SUBR1 ENDP
```

```
        CALL   SUBR1

SUBR1 PROC FAR
        ...             ; your code
        ...
        RETF
SUBR1 ENDP
```

DIFFERENCE STYLES

Full Segment Definition

```
;stack segment
Name1 SEGMENT
            db     64 dup (?)
Name1 ENDS
;data segment
Name2 SEGMENT
            value1 db 54
            ;data
Name2 ENDS
;code segment
Name3 SEGMENT
MAIN      PROC     FAR
            ASSUME CS:   ,DS:   ,SS:
            mov      AX,Name2
            mov      DS,AX
MAIN   ENDP
Name3 ENDS
          END   MAIN
```

Simplified Format

```
            .model small
            .stack 64


;data segment
            .data
            value1 db 54

            .code
MAIN      PROC     FAR

            mov AX,@data
            mov DS,AX


MAIN      ENDP
            END      MAIN
```

DIFFERENCE STYLES II

Full Segment Definition

```
CODE SEGMENT
        ASSUME CS:CODE  DS:CODE
main proc far
      .
      .
       call test1
      .
      .
main proc end

Test1 proc near
      .
      .
Test endp

Msg db "…"

CODE ENDS
       END MAIN
```

Simplified Format

```
                .model small
                .stack 64


;data segment
                .data
                value1 db 54

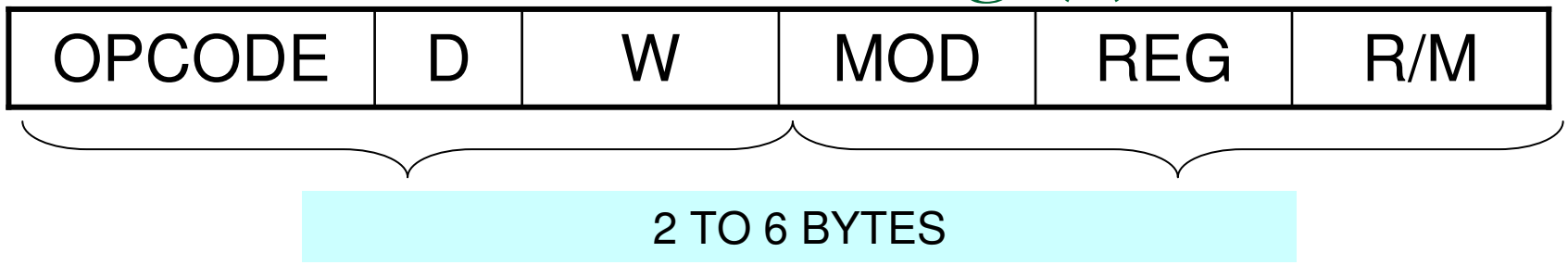
                .code
MAIN        PROC        FAR

                mov AX,@data
                mov DS,AX


MAIN        ENDP
                END         MAIN
```

MANY MORE INSTRUCTIONS…. LEARN IT YOURSELF

Hand Coding (1)



- Opcode field ---8-BITS
- Register Direction Bit (D bit)
  - 1: destination
  - 0: source
- Data Size Bit (W bit)
  - 0: 8 bits 1: 16 bits
- • **Byte 2 has two fields:**
- Mode field (MOD)
- Register field (REG)
- Register/memory field (R/M field)

Encoding of reg Field when w field us present in instruction

Register Specified by reg Field during 16-Bit Data Operations		
	Function of w Field	
reg	When w = 0	When 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

Register Specified by reg Field during 32-Bit Data Operations		
	Function of w Field	
reg	When w = 0	When w = 1
000	AL	EAX
001	CL	ECX
010	DL	EDX
011	BL	EBX
100	AH	ESP
101	CH	EBP
110	DH	ESI
111	BH	EDI

2-bit MOD field and 3-bit R/M field together specify the second operand

mod = 00		mod = 01		mod = 10		mod = 11		
Effective Address Calculation								
r/m		r/m		r/m		r/m	w=0	w=1
000	(BX) + (SI)	000	(BX) + (SI) + D8	000	(BX) + (SI) + D16	000	AL	AX
001	(BX) + (DI)	001	(BX) + (DI) + D8	001	(BX) + (DI) + D16	001	CL	CX
010	(BP) + (SI)	010	(BP) + (SI) + D8	010	(BP) + (SI) + D16	010	DL	DX
011	(BP) + (DI)	011	(BP) + (DI) + D8	011	(BP) + (DI) + D16	011	BL	BX
100	(SI)	100	(SI) + D8	100	(SI) + D16	100	AH	SP
101	(DI)	101	(DI) + D8	101	(DI) + D16	101	CH	BP
110	Direct Address	110	(BP) + D8	110	(BP) + D16	110	DH	SI
111	(BX)	111	(BX) + D8	111	(BX) + D16	111	BH	DI

Mod	Explanation
00	Memory Mode, no displacement follows except when r/m = 110, then 16-bit displacement follows.
01	Memory mode, 8-bit displacement follows
10	Memory mode, 16-bit displacement follows
11	Register Mode (no displacement)

8086/8088 Instruction Set Summary

Mnemonic and Description	Instruction Code			
DATA TRANSFER				
MOV = Move:	76543210	76543210	76543210	76543210
Register/Memory to/from Register	100010dw	mod reg r/m		
Immediate to Register/Memory	1100011w	mod000r/m	data	data if w = 1
Immediate to Register	1011w reg	data	data if w = 1	
Memory to Accumulator	1010000w	addr-low	addr-high	
Accumulator to Memory	1010001w	addr-low	addr-high	
Register/Memory to Segment Register	10001110	mod 0 reg r/m		
Segment Register to Register/Memory	10001100	mod 0 reg r/m		

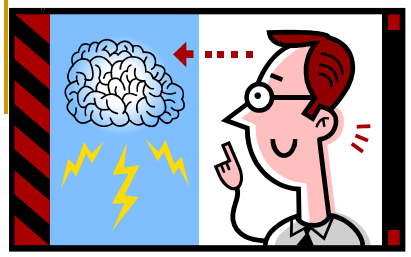
REG is assigned according to the following table:

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

*except if mod = 00 and r/m = then EA = disp-high: disp-low.
if s:w = 01 then 16 bits of immediate data form the operand
if s:w = 11 then an immediate data byte is sign extended to form the 16-bit operand
if v = 0 then "count" = 1; if v = 1 then "count" in (CL) register
x = don't care

Above/below refers to unsigned value
Greater = more positive:
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction
if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16 bits, disp-high is absent
if mod = 10 then DISP = disp-high; disp-low
if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)

Refer to 8088/86 datasheet



HAND CODING

```
        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 SOURCE BLOCK ELEMENT FROM AH TO DEST. BLOCK
        INC SI           ; INCREMENT SOURCE BLOCK POINTER
        INC DI           ; INCREMENT DESTINA. BLOCK POINTER
        DEC CX           ; DECREMENT REPET COUNTER
        JNZ NXTPT        ; JUMP TO NXTPT IF CX NOT EQUAL TO ZERO
        NOP              ; NO OPERATION
```

Identify the type of instruction and hand code the above assembly program!

Refer page 74-79, 113-116 The 8088 and 8086 Microprocessors by Walter A.Triebel and Avtar Singh

HAND CODING (2)

OPCODE	MOD	0	REG	R/M
1000 1110	11	0	11	000 = 8E D8h

- MOV DS,AX
  Opcode = Move reg. to segment
  1000 1110
  MOD=11-reg mode no displacement
  REG=11- to DS
  R/M=000 – from AX

OPCODE	D	W	MOD	REG	R/M
000000	1	1	00	000	100 = 03 04 h

- ADD AX,[SI]
  Opcode=000000
  D = 1 (to register)
  W bit = 1 (16-bits)
  MOD = 00 (displacement absent)
  REG = 000 (to AX)
  R/M = 100 ([SI]+disp)

OPCODE	D	W	MOD	REG	R/M
1000 10	0	0	11	000	011 = 88 C3h

8A D8h

OR

- MOV BL,AL
  Opcode = 100010
  D = 0 (AL source operand - from)
  W bit = 0 (8-bits)
  MOD = 11 (register mode)
  REG = 000 (from AL)
  R/M = 011 (to BL)

OPCODE	D	W	MOD	REG	R/M
000000	0	1	10	000	001 = 01 81 34 12h

- ADD [BX][DI] + 1234h, AX
  Opcode=000000
  D = 0 (from register)
  W bit = 1 (16-bits)
  MOD = 10 (16-bits displacement)
  REG = 000
  R/M = 001 ([BX][DI]+disp)

Jump if Condition is met

8-bit displacement

0111 t t t n	byte offset

Full displacement

0000 1111	1000 t t t n	word offset

Hex		t	t	t	n	Flag Test	unsigned	signed	Other	Condition
0		**0**	**0**	**0**	**0**	OF = 1		JO		Overflow
1		**0**	**0**	**0**	**1**	OF = 0		JNO		No overflow
2		**0**	**0**	**1**	**0**	CF = 1	JB, JNAE			Below, Not above or equal
3		**0**	**0**	**1**	**1**	CF = 0	JNB, JAE			Not below, Above or equal
4		**0**	**1**	**0**	**0**	ZF = 1	JE, JZ	JE, JZ		Equal, Zero
5		**0**	**1**	**0**	**1**	ZF = 0	JNE, JNZ	JNE, JNZ		Not equal, Not zero
6		**0**	**1**	**1**	**0**	CF = 1 or ZF = 1	JBE, JNA			Below or equal, Not above
7		**0**	**1**	**1**	**1**	CF = 0 and ZF = 0	JNBE, JA			Not below or equal, Above
8		**1**	**0**	**0**	**0**	SF = 1		JS		Sign
9		**1**	**0**	**0**	**1**	SF = 0		JNS		Not sign
A		**1**	**0**	**1**	**0**	PF = 1			JP	Parity, Parity Even
B		**1**	**0**	**1**	**1**	PF = 0			JNP	Not parity, Parity Odd
C		**1**	**1**	**0**	**0**	SF ≠ OF		JNGE, JL		Less than, Not greater than or equal to
D		**1**	**1**	**0**	**1**	SF = OF		JGE, JNL		Not less than, Greater than or equal to
E		**1**	**1**	**1**	**0**	ZF = 1 or SF ≠ OF		JNG, JLE		Less than or equal to, Not greater than
F		**1**	**1**	**1**	**1**	ZF = 0 and SF = OF		JG, JNLE		Not less than or equal to, Greater than

JMP-Unconditional Jump (to same segment)

Short	1110 1011: 8-bit displacement
Direct	1110 1001: full displacement
Register indirect	1111 1111: 11 100 reg
Memory indirect	1111 1111: mod 100 r/m

Examples:

JMP unconditional jump (same argument)

1110 1011: byte displacement (EB __)

1110 1001: word displacement (E9 __)

JCXZ jump if CX = 0

1110 0011: byte displacement (E3 __)

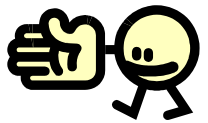
(to work with ECX use address size prefix)

LOOP

1110 0010: byte displacement (E2 __)

Include auto-decrement of the CX register.
Jump if CX is not zero after decrement.

Answer



```
        MOV AX, 2000H   ; IMMEDIATE DATA TO REGISTER      B80020
        MOV DS, AX      ; MOVE REGISTER TO SEGMENT REG    8ED8
        MOV SI, 100H    ; MOV IMMED. TO REG               BE0001
        MOV DI, 120H    ; MOV IMMED. TO REG               BF2001
        MOV CX, 10H     ; MOV IMMED. TO REG               B91000
NXTPT:  MOV AH,[SI]     ; MOV MEMORY DATA TO REG          8A24
        MOV [DI],AH     ; MOV REGISTER DATA TO MEMORY     8825
        INC SI          ; INCREMENT REG.                  46
        INC DI          ; INCREMENT REG.                  47
        DEC CX          ; DECREMENT REG.                  49
        JNZ NXTPT       ; JUMP ON NOT EQUAL TO ZERO       75F7
        NOP             ; NO OPERATION                    90
```

QUIZ

- Hand code the following instructions
  - MOV CX,7
  - MOV AL,BL
  - MOV [6465H],AX
  - MOV DL,[SI]
  - MOV AX,[BX+4]
  - MOV [DL-8],AL
  - MOV CL,[BX+DI+2080H]
  - AND AL,[345H]
  - TEST DX,2003H