

8086 Programming

Compiled by: Chandra Thapa

October 23, 2012

UNIT I

Concept (not important for exam and not in syllabus)

Instruction Encoding

How to encode instructions as binary values?

Instructions consist of:

- **operation** (opcode) e.g. MOV
- **operands** (number depends on operation)
 - operands specified using addressing modes
 - addressing mode may include **addressing information**
 - e.g. registers, constant values

Encoding of instruction must include opcode, operands & addressing information.

Encoding:

- represent entire instruction as a **binary value**
 - **number of bytes** needed depends on how much information must be encoded
- instructions are **encoded by assembler**:
 - **.OBJ file !** (link, then loaded by loader)
- instructions are decoded by processor during execution cycle

We will consider a subset of interesting cases

Instructions with No Operands (easy)

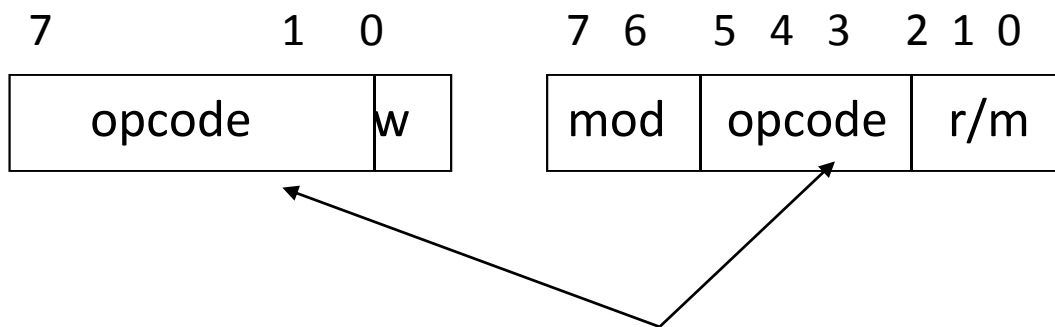
- encode operation only in a single byte
- examples:

RET C3 H NOP 90 H

- Are consistent – never change

Instructions with One Operand

- operand is a register (reg8/16) or a memory operand (mem8/16)
 - always 2 bytes for opcode and addressing info
 - may have up to 2 more bytes of immediate data



- **opcode** bits: some in both bytes! **10 bits** total
- w = width of operand
 - 0 = 8-bit
 - 1 = 16-bit
- mod & r/m encode addressing info

mod = 01 is **not** used by the assembler!

MOD / R/M TABLE

<i>r/m</i>	<i>mod</i>			register	
	00	01	10	11	
		d is 8-bit signed value	n is 16-bit unsigned value	<i>w</i> = 0 <i>w</i> = 1	
000	[BX + SI]	[BX + SI + d]	[BX + SI + n]	AL	AX
001	[BX + DI]	[BX + DI + d]	[BX + DI + n]	CL	CX
010	[BP + SI]	[BP + SI + d]	[BP + SI + n]	DL	DX
011	[BP + DI]	[BP + DI + d]	[BP + DI + n]	BL	BX
100	[SI]	[SI + d]	[SI + n]	AH	SP
101	[DI]	[DI + d]	[DI + n]	CH	BP
110	direct address	[BP + d]	[BP + n]	DH	SI
111	[BX]	[BX + d]	[BX + n]	BH	DI

Example:

INC DH

opcode: 1st byte: 1111111 2nd byte: 000

w = 0 (8-bit operand)

operand = DH register: **mod = 11 r/m = 110**

from table!

opcode w

1st byte: 1111111 0 = **FE H**

mod opcode r/m

2nd byte: 11 000 110 = **C6 H**

What does following encoding represent?

11111111 11000111 = **FF C7 H**

opcode = INC 1st byte: 1111111 2nd byte: 000

w = 1 16-bit operand

mod = 11 register operand

r/m = 111 DI register

encoding for **INC DI !!!**

Another Example: **INC BYTE PTR [SI - 4]**

- indexed addressing to an 8-bit memory operand
- will need extra byte(s) to encode the immediate value (**-4 = FFFC H**)

opcode – same as last example: 111111 000

w = 0 8-bit destination (memory) operand

r/m = 100 (from table)

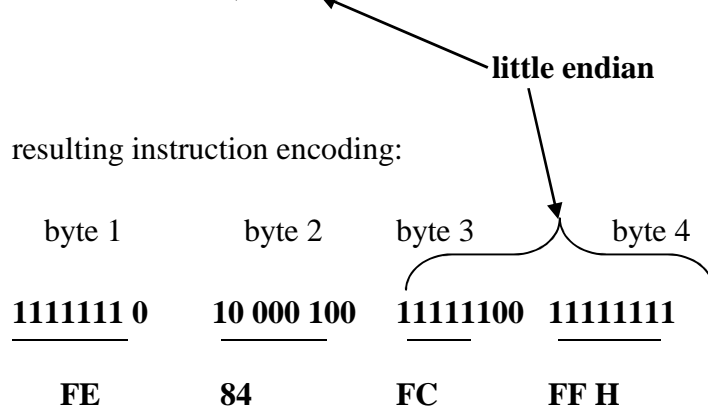
mod could be 01 or 10 depends on constant

can use whichever mod value works

can shorten encodings!

the assembler will use **mod = 10**

16-bit constant (FFFCH) encoded into instruction

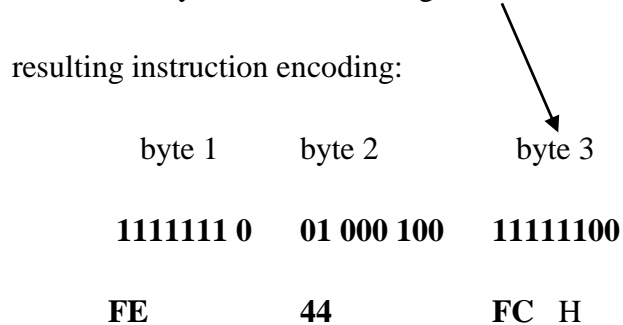


Could also encode same instruction:

mod = 01 constant encoded as signed 8-bit value

therefore instruction encoding includes only

one byte for the encoding of -4



N.B. the 8-bit value ($-4 = \text{FC H}$) is **sign extended** to 16-bits (FFFC H) before adding SI value

why?

value of most signif. bit of byte is
copied to all bits in extension byte

Another Example:

INC BYTE PTR [SI + 128]

- indexed addressing to an 8-bit memory operand
- everything the same as last example, except:
can't encode $+128$ as 8-bit signed value!

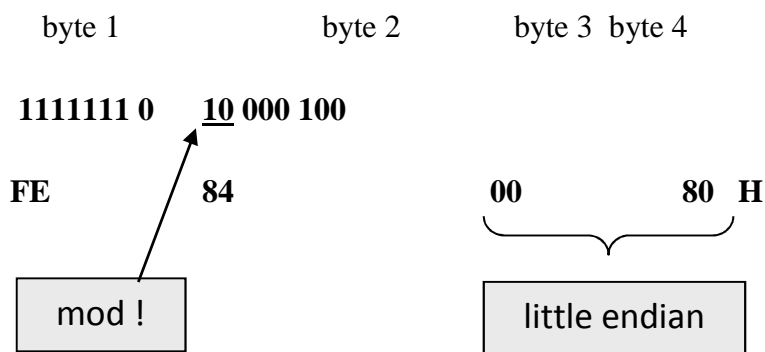
need 16-bits to encode 128

then must have **mod = 10 !!**

instruction encoding would include

two extra bytes encoding 128 = $00\ 80\ \text{H}$

resulting instruction encoding:



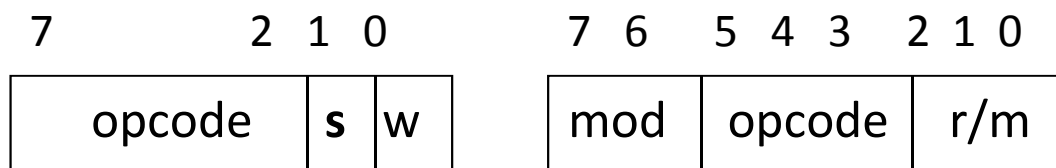
Instructions with Two Operands (2 Forms)

- at most, can have **only one** memory operand
 - can have 0 or 1 memory operands, but not 2
 - limits max. instruction size to 6 bytes

- e.g. **MOV WORD PTR [BX+ 500], 0F0F0H**
 - 2 bytes opcode + addressing info
 - 2 bytes destination addressing constant **500**
 - 2 bytes source constant **F0F0 H**

FORM 1: Two Operands And Source Uses Immediate Mode

- destination either register or memory
 - encode dest using mod & r/m – as before



w (as before) = size of operand (8- or 16-bit)

if **w = 1** (16-bit) then **s is significant**

s → indicates size of immediate value

= **0** → all 16-bits encoded in instruction

assembler always used s = 0

= **1** → 8-bits encoded – sign extend to 16-bits!

Example: **SUB My_Var, 31H**

- My_Var** is a word (DW) stored at address 0200H

opcode bits: 1st byte: **100000** 2nd byte: **101**

w = 1 (16-bit memory operand)

s = 1 – can encode 31H in one byte

sign extend to 0031H

$\text{mod} = 00$
 $\text{r/m} = 110$

} destination: direct addressing

resulting encoding:

opcode

100000 1 1 00 101 110

s w mod r/m

83

2E

2-bytes dest 1-byte
address imm

assembler uses

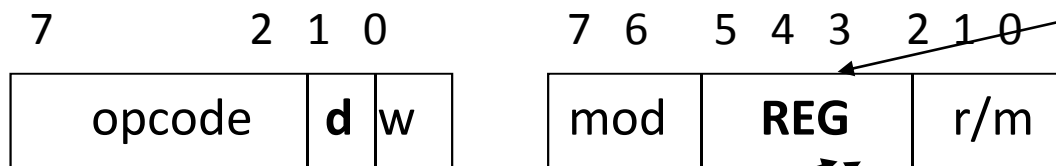
$s = 0$ & 16-bit immediate value
 $= 31\ 00$ (little endian)

02 00 31

stored little
endian

FORM 2: Two Operands And Source Does Not Use Immediate Mode

- at least one of destination or source is register!
- encode register operand
- encode other using mod & r/m – as before



d = destination

= 0 source is encoded in REG

= 1 destination is encoded in REG

Example: **SUB BX, CX**

Case 1: Source (CX) is encoded in **REG**

opcode: **0010 10**

d = 0 – source is encoded in REG

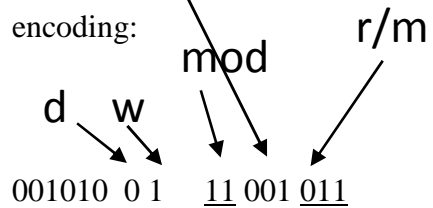
w = 1 – 16-bit operand

mod = 11 destination is register

r/m = 011 BX register is destination register

REG = 001 CX register is source register

encoding:



001010 0 1 11 001 011

29 CB

Case 2: Destination (BX) is encoded in **REG**

opcode: **0010 10**

d = 1 – destination is encoded in REG

w = 1 – 16-bit operand

mod = 11 source is register

r/m = 001 CX register (source)

REG = 011 BX register (destination)

encoding:

001010 0 1 11 011 001

29 D9

- **cases 1 & 2: two encodings for same instruction!**

Some Special-Case Encodings:

- single-operand instructions & operand is 16-bit register – can encode in one byte
- instructions involving the accumulator:

AL or AX

- shorter encoded forms – often one byte

Why might this be important? EXAM !!!!
--

Instruction Encoding (human perspective)

1. **given instruction** – how to encode ?
2. **given binary** – how to decode ?

Given instruction – how to encode ?

- decide on form & number of bytes
- find opcode bits from table
- decide on remaining bits
 - individual bit values
 - look up **mod & r/m** values if needed
 - look up **register** encoding if needed
- fill opcode byte(s)
- add immediate operand data byte(s)
 - words → little endian
 - dest precedes source

Given binary – how to decode ?

- use first 6 bits of first byte to decide on form & number of bytes
- use opcode bits to find operation from table
- identify operands from remaining bits
 - individual bits

- look up **mod & r/m** values if present
- look up **register** encoding if present
- add immediate operand data byte(s) if present
 - words → little endian
 - dest precedes source

Could you hand-assemble a simple program now?

YES! recall previous control flow

encoding discussions !!

What about an operation / opcode look-up table?

- many forms – some give:
 - opcode bits only
 - entire first instruction byte – including operand info encoded in first byte!
- list of info for each instruction will be posted!
 - opcode bits
 - forms

UNIT II

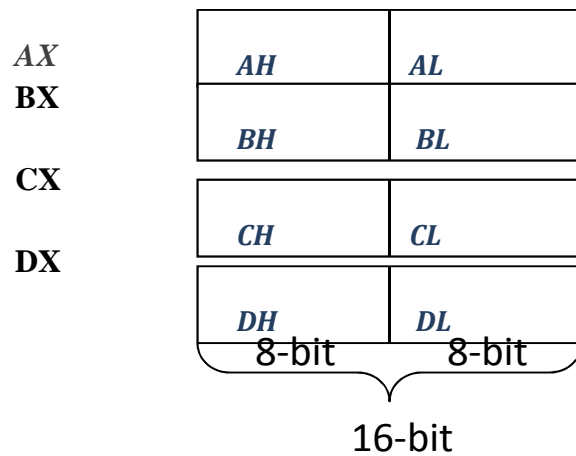
Overview of the 80x86

80x86 register model,
segmented memory model
instruction execution.

8086 Register Set

16-Bit **General Purpose** Registers

- can access all 16-bits at once
- can access just high (H) byte, or low (L) byte



only the General Purpose registers allow access as 8-bit High/Low sub-registers

16-Bit **Segment Addressing** Registers

CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment

16-Bit **Offset Addressing** Registers

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Destination Index

16-Bit **Control/Status** Registers

IP Instruction Pointer (execution control)

FLAGS Status Flags – one bit/flag

- 16-bit reg, but only 9 bits have meaning
 - text: page 27
- ignore unused bits
- usually interested in individual flags
 - not 16-bit value

Quick Overview of 80x86 Flag

Flag	Name	Description
C	Carry	
A	Auxiliary Carry	
O	Overflow	
S	Sign	
Z	Zero	
D	Direction	
I	Interrupt	
T	Trap	

- Flags are set and cleared as “side-effects” of an instruction
 - Part of learning an instruction is learning what flags it writes
- There are instructions that “read” a flag and indicate whether or not that flag is set or cleared.

Other Registers in Programmer's Model

- support the execution of instructions
- **cannot** be accessed directly by programmers
- may be larger than 16-bits:
temporary reg's (scratchpad values)

IR Instruction Register

Segmented Memory Model:

Processor Design Problem: How can 16-bit registers and values be used to specify 20-bit addresses?

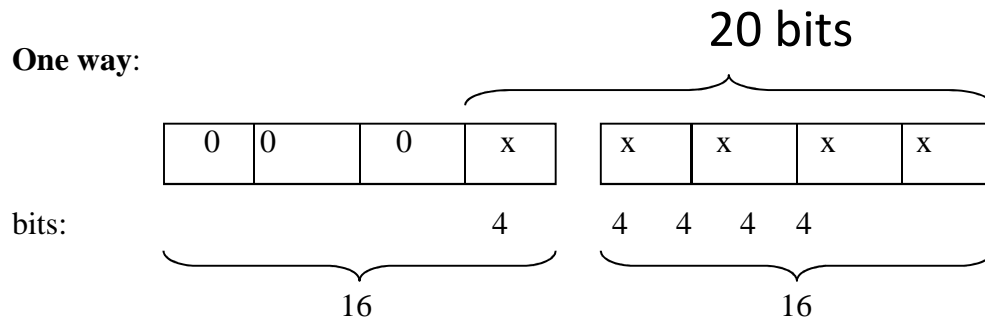
Want to use 16-bit registers to refer to memory addresses (e.g. for indirect addressing modes), then

- need more than one 16-bit register!
 - need at least 2

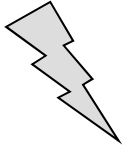
could get by with one 16-bit register and one 8-bit register, BUT ... if only 16-bit registers are used, then need two!

How could/should/might two 16-bit values be combined to form a 20-bit value?

One way:



not used (to easy to understand ☺ , but has some other porblems ☹)



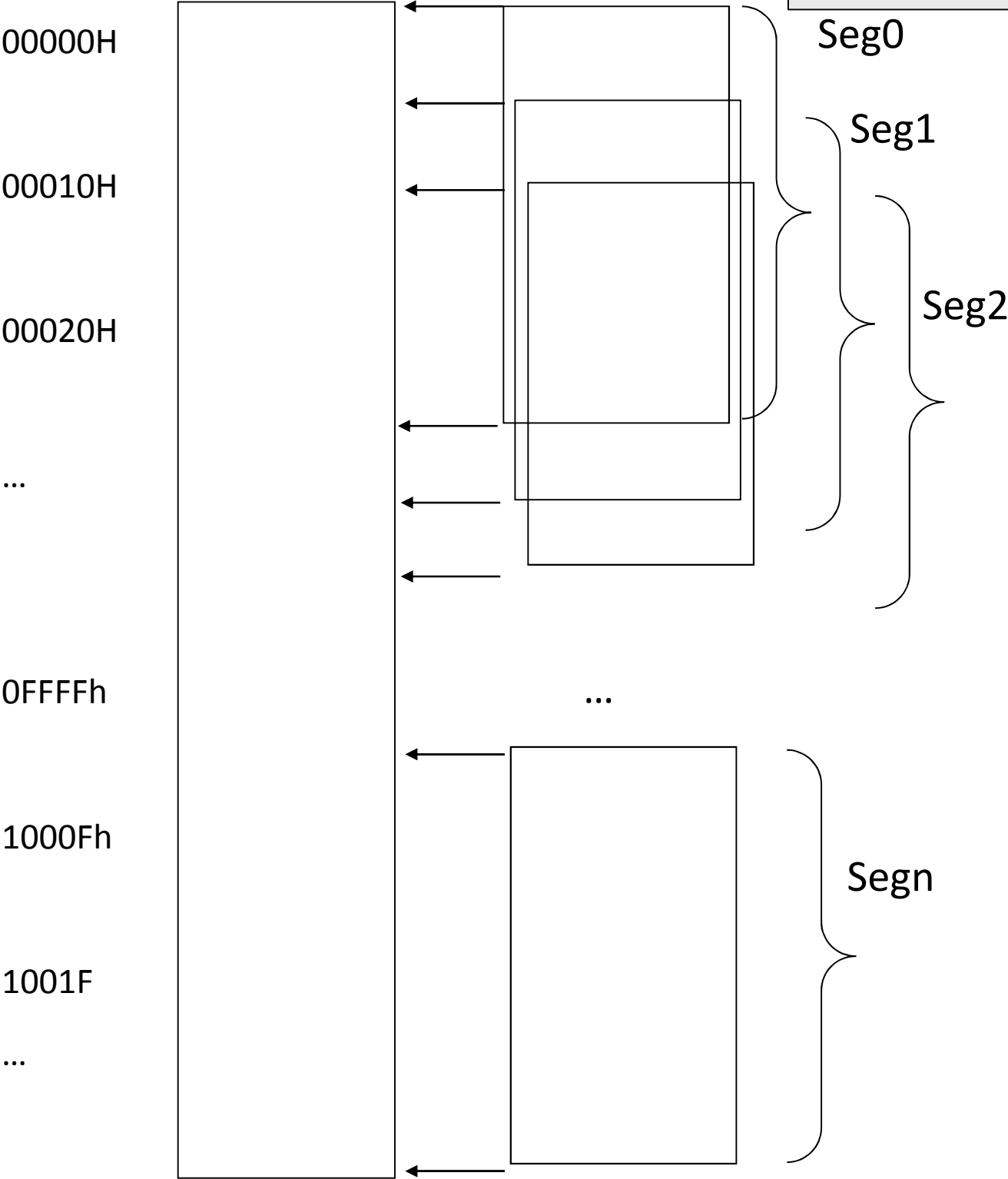
Intel's Solution:

Segmented Memory for a 20-bit Address Space as Viewed from the 8086 Processor's Perspective

- view **memory** as a set of overlapping “segments”
- each segment starts on an address that ends in 0 (hex)
 - **segment 0** starts at address 00000H
 - **segment 1** starts at address 00010H
 - **segment 2** starts at address 00020H
 - etc.
- each segment consists of 64K consecutive locations
 - **segment 0** goes from 00000H to 0FFFFH
 - **segment 1** goes from 00010H to 1000FH
 - **segment 2** goes from 00020H to 1001FH
 - etc.
- a new segment starts every 16 bytes, and each segment is 64K bytes long!
 - **segment i overlaps with segment i + 1**

16-bit “offset”: $2^{16} = 64K$

**Visualization
of Segmented
Memory**



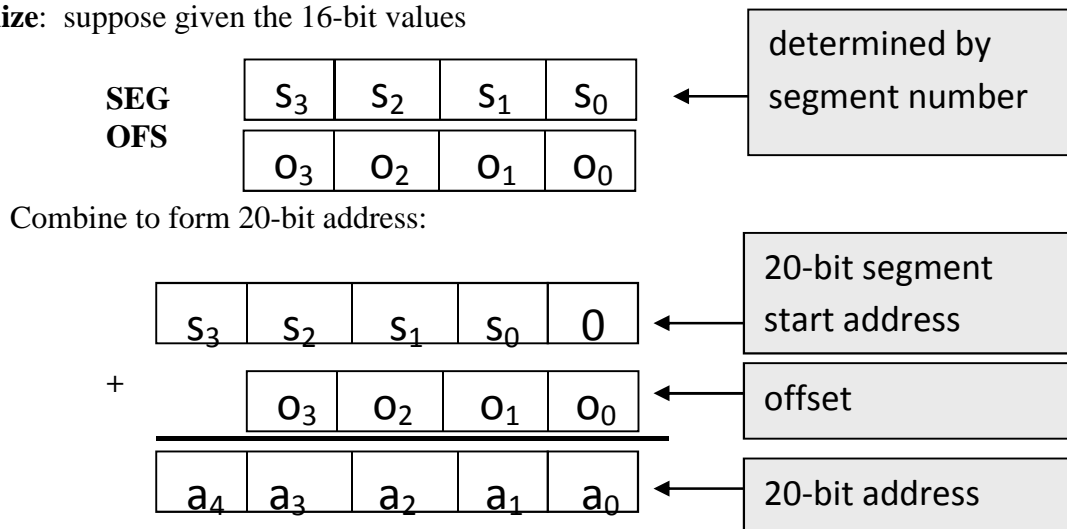
Software (Programmer's) Perspective of Segmented Memory:

- addresses are **NEVER** specified as 20-bit values
- addresses are **ALWAYS** specified as two 16-bit values
 - segment number (16-bits): **SEG**
 - offset (16-bits): **OFS**
 - denote address as “**SEG : OFS**”

Notation!

- **segment number** is converted into the **20-bit start address** of a segment:
 - start address of segment = segment number * 16_{10}
 - recall $16_{10} = 10H$
 - multiplying a hex value by $10H$ is the same as shifting the value left one hex digit (4 bits)
 - e.g. if 16-bit segment number = $0002H$
 - segment start address = $0002 * 10H$
 $= 00020H$
- unique 20-bit address is formed by adding 16-bit offset to 20-bit start address of segment

Visualize: suppose given the 16-bit values



Example: suppose

segment number = $6020H$

offset = $4267H$

seg * 10H → 60200 H

+ ofs → 4267 H

64467 H ←

20-bit address

An Ugly **Side Effect of Segmented Memory**:

each memory byte can be referred to by many different

SEG : OFS pairs ☹ ☹

Example: the (unique) byte at address **00300 H**

can be referred to by:

0 H : 300 H

1 H : 2F0 H

30 H : 0 H

(more too !)

Questions:

At most, how many **different** SEG : OFS pairs can refer to the same memory byte ?

At most, how many **different** segments can one memory byte be contained in ?

(good midterm questions ! ☺)

How is segmented memory managed by the 8086 ?

- 8086 includes **four 16-bit SEGMENT registers**:

CS : Code Segment Register

DS : Data Segment Register

SS : Stack Segment Register

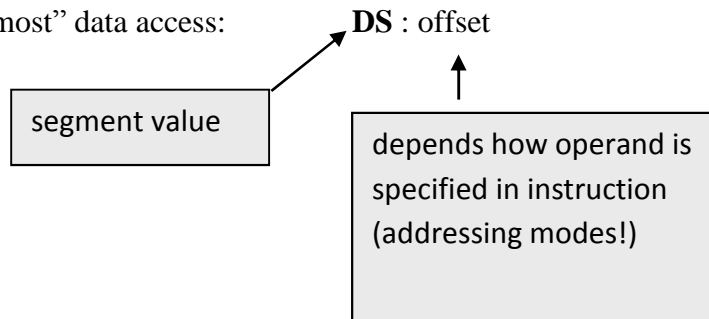
ES : Extra Segment Register

- Segment registers are used by default as the segment values during certain memory access operations

- all instruction fetch:



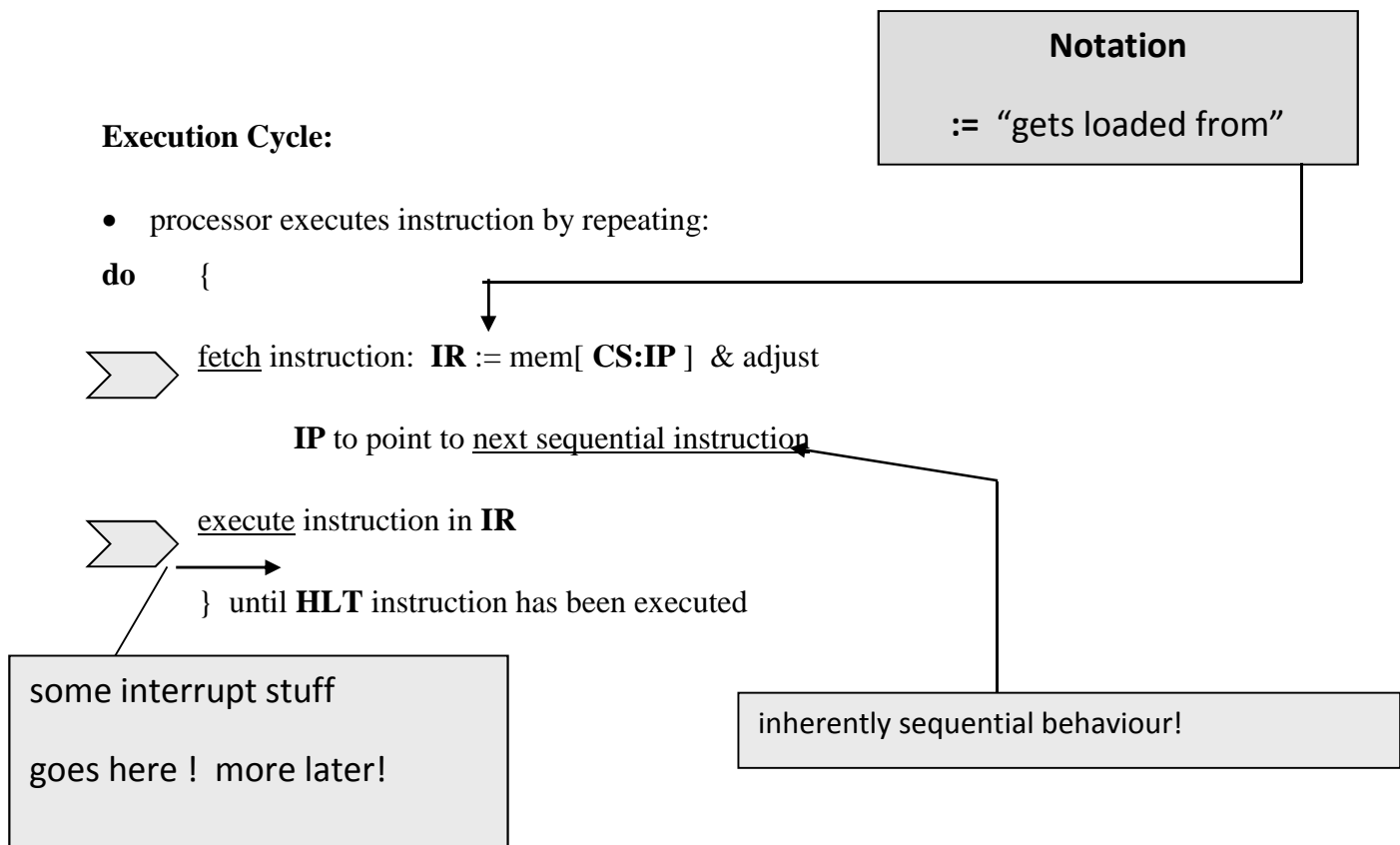
- “most” data access:



(Since the processor uses contents of DS as the 16-bit **segment** value when fetching operands, the programmer only needs to supply the 16-bit **offset** in instructions)

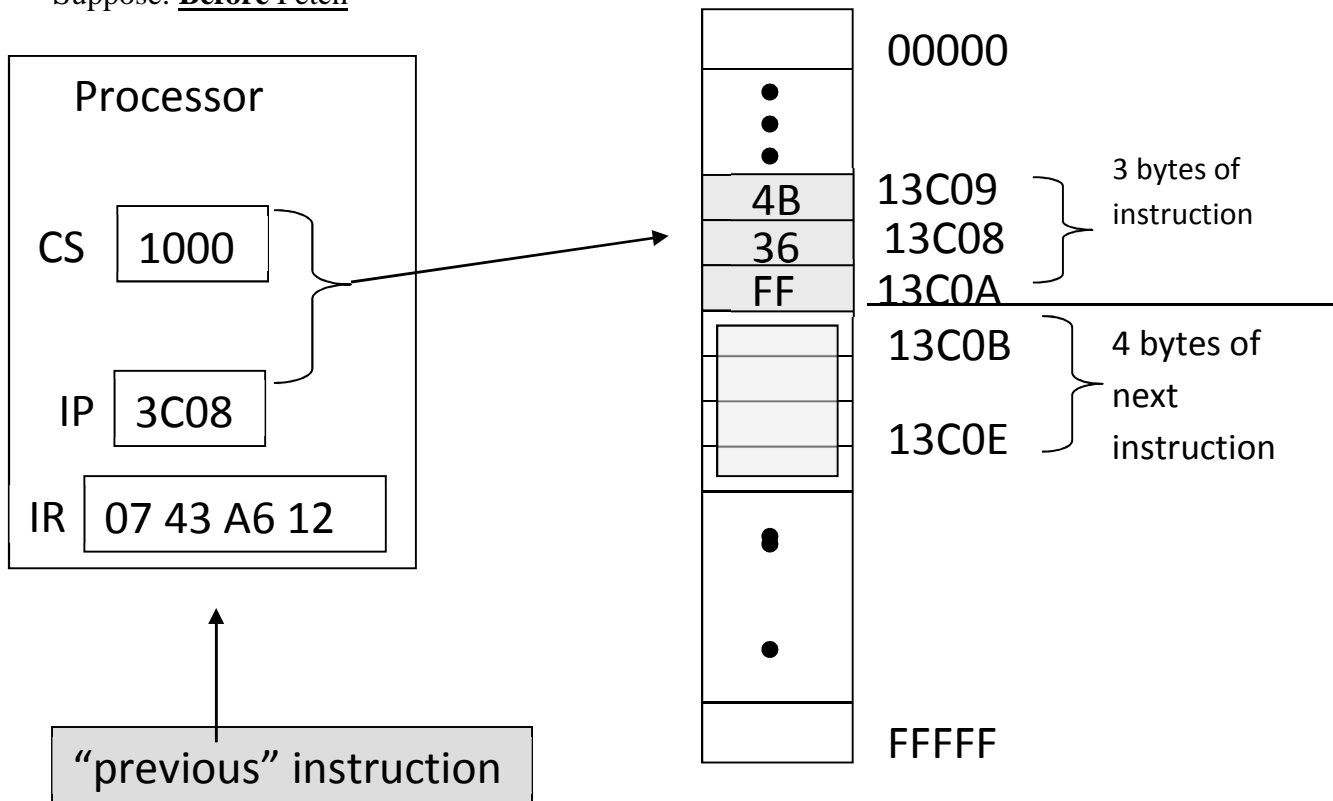
- BUT segments must be initialized before use (More on this later!!)

Segment registers **seem straightforward**, BUT . . . common source of **confusion** for students !

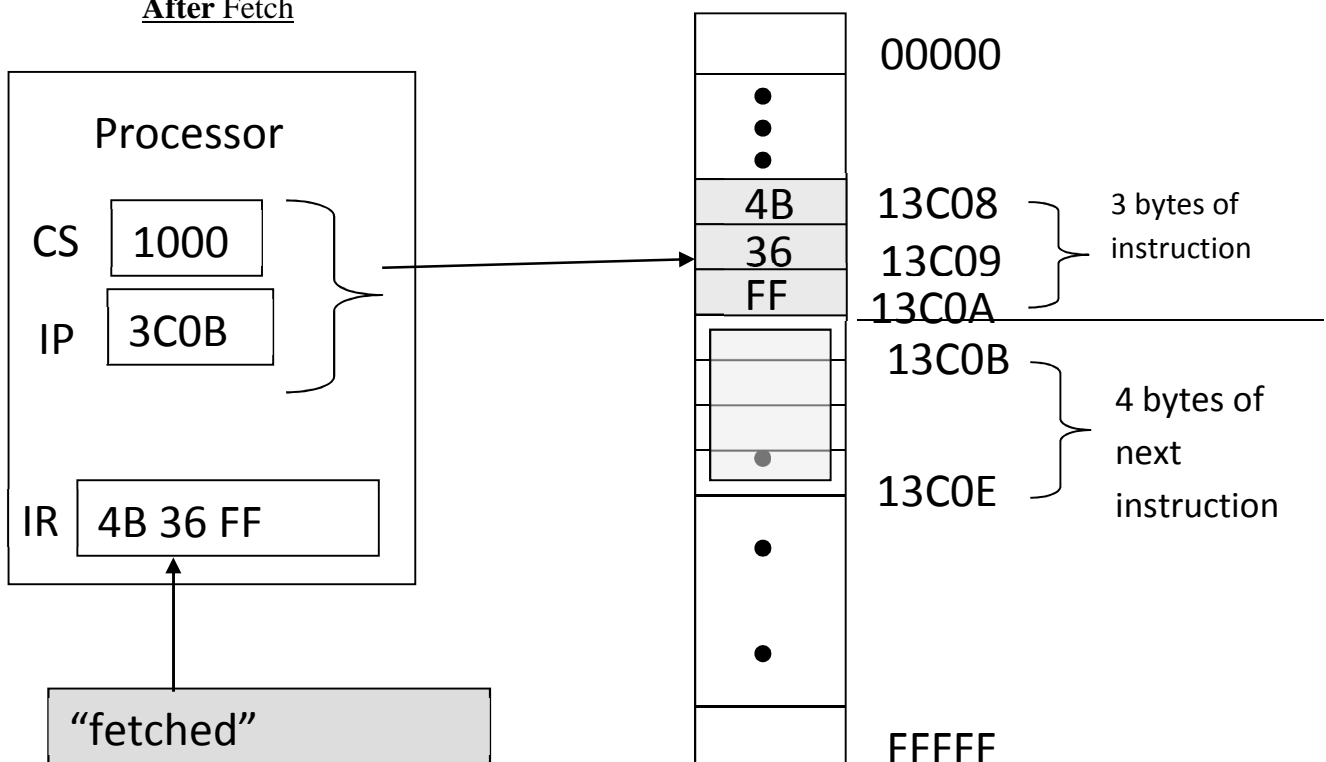


- **CS:IP** contains address of **next** instruction to execute
- 16-bit segment value (**CS**) is combined with 16-bit offset (**IP**) to create a 20-bit address
- **IR** holds instruction in processor
- instruction fetch from memory
- load 1st byte
 - from encoding: decide how many more bytes are needed
 - may need to consider 2nd byte to decide how many bytes in total
(more on encoding later)
- **IP** is adjusted as bytes are loaded into **IR**

Suppose: **Before Fetch**



After Fetch



Program Development

Problem: must convert ideas (human thoughts) into an executing program (binary image in memory)

Need: DEVELOPMENT PROCESS

- people-friendly way to write programs
- **tools** to support conversion to binary image
- **assembly language:** used by people to describe programs
 - **syntax:** set of **symbols** + **grammar rules** for constructing **statements** using symbols
 - **semantics:** what is meant by statements → ultimately: the binary image
- **assembler:** program – converts programs from assembly language to object format

tool
- **object format:** an intermediate format
 - mostly binary, but may include other info
- **linker:** program that combines object files to create an “executable” file

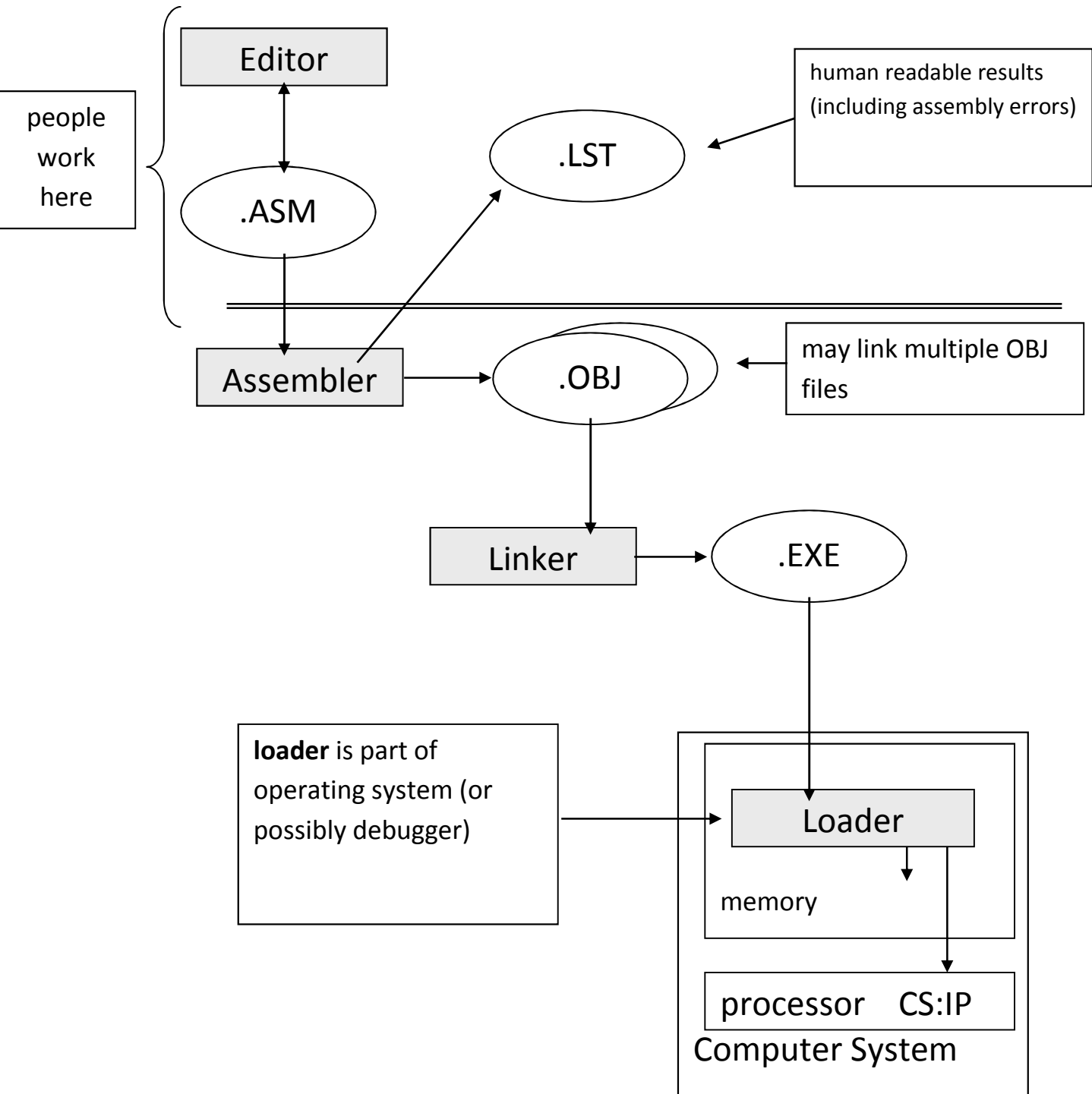
tool
- **loader:** loads executable files into memory, and may initialize some registers (e.g. IP)

tool

Another Useful Tool: DEBUGGER

- allows:
 - program to be loaded and executed
 - control execution (e.g. start/stop)
 - view state variables values
 - modify state variable values

Development Process



People create .ASM files using assembly language

MASM Assembly Language

- Microsoft product
- free with copy of textbook
- **syntax** must account for all aspects of a program and development process:
 - constant values
 - reserve memory to use for variables
 - write instructions: operations & operands
 - specify addressing modes
 - directives to tools in development process

Constants

- **binary** values: consist of only 0's and 1's
 - ends with 'B' or 'b'
 - e.g. 10101110b
- **hexadecimal** value: starts with 0 .. 9
 - may include 0 .. 9, A .. F (a .. f)
 - ends with 'H' or 'h'
 - e.g. 0FFH (8-bit hex value)
- **decimal** value:
 - default format – no “qualifier” extension
 - consists of digits in 0 .. 9
 - e.g. 12345
- **string**: sequence of characters encoded as ASCII bytes:
 - enclose characters in single quotes
 - e.g. 'Hi Mom' – 6 bytes
 - character: string with length = 1

Labels

- user-defined **names** – represent addresses
- lets programmer **refer to addresses** using logical names – no need for concern with exact hexadecimal values
- leave assembler to:
 - decide exact addresses to use
 - deal with hexadecimal addresses
- labels are used to identify addresses for:
 - **control flow** – identify address of target
 - **memory variables** – identify address where data is stored
- labels serve in **2 roles: definition & reference**
- **label definition:**
 - used by assembler to decide exact address
 - must be first non-blank text on a line
 - name must start with alpha A .. Z a .. z
 - then contain: alpha, numeric, ‘_’

If the label is a control flow target (other than procedure name – later) then must append “:”

- some control flow label examples:

Continue:

L8R:

Out_2_Lunch:

- cannot redefine reserved words

- e.g.

MOV:

illegal ! ☹️

- label **represents address** of first allocated byte that follows definition

- e.g. **DoThis:** MOV AX, BX

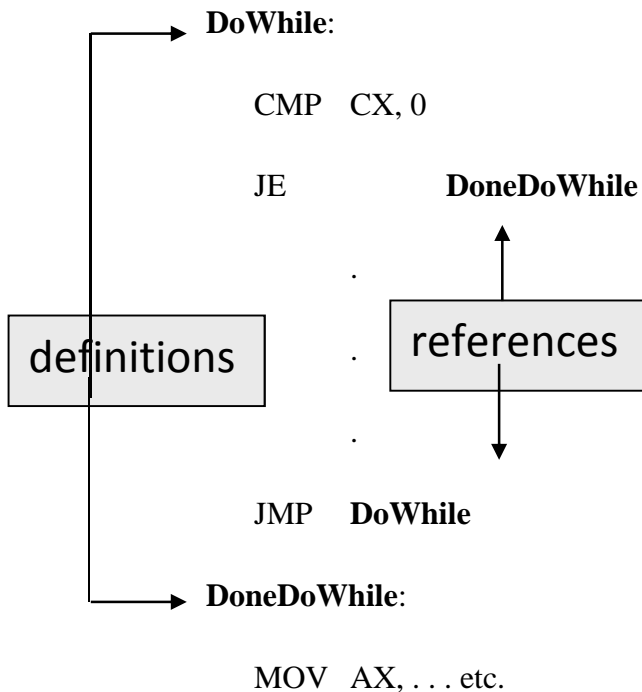
DoThis represents address of first byte of the MOV instruction

- label reference:** use of label in an operand

- refers to address assigned by assembler
 - does not include “:”

control flow example:

; assume CX contains loop counter



- specify target using label
- assembler assigns addresses **AND** calculates relative offsets

Memory Declarations

- reserve memory for variables
- 2 common sizes:

- **DB** reserves a byte of memory
- **DW** reserves a word (2 consecutive bytes) of memory
- may also provide an (optional) initialization value as an operand

Examples:

comments start with ";" and run to the end of the line

X **DB** ; reserves one byte

 ; reserves one byte – label X

 ; X is defined to represent

 ; the address of the byte

label definition

no ":" on variable name definitions!

Y **DB** **3** ; reserve one byte – label Y etc.

 ; and initialize the byte to 3

DW ; reserve 2 consecutive bytes

Z **DW** ; reserves 2 bytes – label Z is

 ; defined to represent the

 ; address of the first byte

W **DW 256** ; reserve 2 bytes – label W etc.

 ; and initialize the bytes to

 ; 256 (little endian !!!)

HUH DW W ; reserve 2 bytes – label etc.

 ; and initialize the bytes to

label definition

label reference

; contain the address of the

; variable W above

DB 'C' ; reserves 1 byte – initializes

; the byte to 43H

Tool Directives

- statements that are intended for other tools
- are not assembled directly into instructions or memory declarations

END Directive:

- directive to 2 tools: assembler & loader
- **assembler:** stop reading from .ASM file
 - any subsequent statements are ignored
- optional operand
 - if present, must be a label reference
 - interpreted as specifying the address of the first instruction to be executed
 - **loader:** load specified address into CS:IP after loading .EXE file

format:

END

label-reference



optional

Some “.” Directives:

.8086

- limits assembler 8086 processor instruction set
- NOTE: a program only requires a fixed amount of memory

- the tools (assembler, linker) can organize the program to fit memory as the tools see fit
- let the tools deal with segments as much as possible ! 😊

.model

- allows tools to make simplifying assumptions
- **.model small**
 - at most: program will use one code and one data segment
 - no intersegment control flow needed
 - never need to modify DS once initialized

.code

- identifies the start of the code segment
- tools will ensure that enough memory is reserved for the encodings of the instructions

.data

- identifies the start of the data segment

.stack *size*

- reserves *size* bytes of memory for the run-time stack
- more on stack later

In this course:

- the actual amount of memory reserved for code will be less than 64K
- the actual amount reserved for data will be less than 64K
- the amount of memory used for the run-time stack will (not likely?) exceed 1K bytes (?)

Example:

- suppose that a program requires:
 - 20 bytes for data
 - 137 bytes for instructions (code)

- 100 bytes for stack
- the tools can decide on segment use
- only use 120 bytes in the “data” segment (data plus stack) and 137 in the “code” – actual segments used could overlap!

Loading:

- program must be loaded into memory
- loading done by loader (tool)
- loader decides which actual segments to be used
- loader initializes SS:SP (for stack use – later!) and CS:IP (to point to first instruction to be executed)
- what about DS?
 - loader “knows” which segment it has loaded as the data segment
 - as the program is loaded, loader replaces every occurrence of “@data” with the data segment number

What does this mean for our program ?

For instruction operand access: **DS** must be **initialized**

- Recall : The processor uses contents of DS as the 16-bit **segment** value when fetching operands, so the programmer only needs to supply the 16-bit **offset** in instructions
- Initialization is typically done dynamically
 - It must be first thing program does !
 - Specifically, no variable defined in the data segment can be referenced until DS is initialized.

How do we initialize DS ?

`MOV DS, @data` ☹

- NO, we cannot load an immediate value directly into DS (limited addressing modes ☹)

Correct initialization:

`MOV AX, @data`

`MOV DS, AX`

First Program:

; 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



proc – like a function definition

UNIT III
Instructions:

- 2 aspects: operation & operands
 - operation: **how** to use state variable values
 - operands: **which** state variables to use

e.g. $C = A + B$

operations: addition (+) and assignment (=)

operands: state variables A , B & C

- **source** operands: provide values to use (inputs)
 - A & B in: $C = A + B$
- **destination** operands: receive results (outputs)
 - C in: $C = A + B$

- same state variable can play multiple roles?

$A = A + A$

- **mnemonics**: specify operations
 - human-oriented short-forms for operations e.g.:
 - MOV (move)
 - SUB (subtract)
 - JMP (jump)

- operands can be specified in a variety of ways
 - **addressing modes!!!!**
 - simple modes: register, immediate, direct
 - more powerful: indirect

- instruction encoding (as binary value) must account for both aspects of instructions:
operation and **operand** information

Types of Instructions

- **data transfer**: copy data among state variables
 - do not modify FLAGS
- **data manipulation**: modify state variable values – including FLAGS
- **control-flow**: determine “next” instruction to execute – allow non-sequential execution

Data Transfer

MOV (Move) Instruction

syntax: **MOV dest , src**

semantics: **dest := src**

- copy src value to dest state variable
- **register and memory operands only** (I/O ??)

Register Addressing Mode:

- allows a **register** to be specified as an operand
 - as **source**: copy register value
 - as **destination**: write value to register

E.G. **MOV AX, DX**

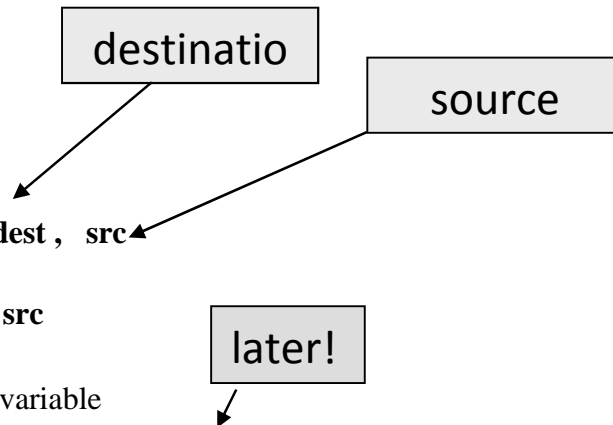
AX := DX

- contents of DX is copied to AX
- register addressing mode for **both** dest and src

- dest and src must be compatible (same size)

MOV AH, CL 8-bit src and dest ☺

MOV AL, CX ????? ☹



Immediate Addressing Mode:

- allows a **constant** to be specified as **source**
 - source value assembled into the instruction
 - loaded into IR as part of instruction
 - value obtained from IR as instruction executed

E.G **MOV AL, 5**

- AL is 8-bit dest
 - instruction encoding includes 8-bit value 05h
-
- what about: **MOV AX, 5**
 - 16-bit dest
 - encoding includes 16-bit value 0005h

- what about **MOV 4, BH** ????

dest as
immediate
value ? ☹

Direct Addressing Mode:

- specify the address of a **memory operand**
- **direct**: specify address as a constant value
 - address gets encoded as part of instruction

static: must be known at assembly-time and remains constant through execution

Example :

MOV AL, [5]

- *Implicitly uses DS*
- Reads contents of byte at address DS:5

BEWARE :

Compare To Immediate Mode

MOV AL, 5

MOV X, AX

- Assumes a variable is declared in data segment
- Write contents of word at address DS:X

What about Operand Compatibility for memory operands (eg. direct), consider

MOV [BC], AX

MOV [1234h], AL

Clear and unambiguous. REGISTER operand determines size

MOV [BC], 1

MOV [1234h], 0

Ambiguous : 8 or 16 bit moves?
Default ?

- Need syntax to remove ambiguity
- Qualify off-processor access using
WORD PTR word pointer – 16-bit operand

BYTE PTR byte pointer – 8-bit operand

MOV BYTE PTR [0FF3E], 1

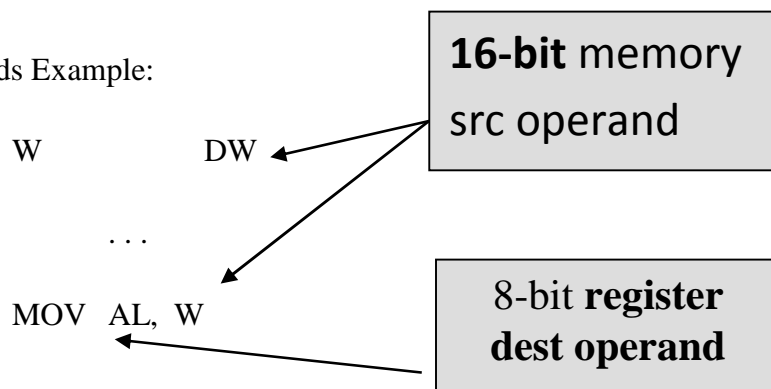
8 bit destination, no ambiguity

MOV WORD PTR [1234h], 0

16-bit destination, no ambiguity

Assembler Tip :

Incompatible Operands Example:



Will give assembly error!

Summary of addressing modes so far :

Register MOV AL, DL

Immediate MOV AL, 5

Memory

Direct MOV AL, VARIABLE

MOV AL, [1234]

Indirect ... Later

Data Manipulation Instructions:

- use state variable values to **compute** new values
- modify state variables to hold results
 - including FLAGS
 - common flag results:
 - **ZF** = zero flag set iff result = 0
 - **CF** = carry flag reflect carry value
 - **SF** = sign flag set iff result < 0
 - assumes 2's complement encoding!
 - **OF** = overflow flag
 - set iff signed overflow
 - what about unsigned overflow?

set = 1, clear = 0

if-and-only-if

specific use of "**overflow**" – not the same as the general concept!

ADD dest, src

dest := dest + src (bitwise add)



- **dest** is both a source and destination operand
- also modifies **FLAGS** as part of instruction execution:

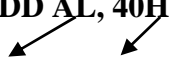
ZF := 1 iff result = 0

SF := 1 iff msbit of result = 1 (sign = negative)

CF := 1 iff carry out of msbit

OF := 1 iff result overflowed signed capacity

Example: Suppose that AL contains 73H, when

execute: **ADD AL, 40H**

 73 H + 40 H
 = B3H carry?

results: **AL** := **B3H** (= 1011 0011 B)

ZF := 0 result ≠ 0

SF := 1 result is negative (signed)

CF := 0 (no carry out of msbit)

OF := 1 +ve + +ve = -ve

SUB dest, src

dest := dest – src

- like ADD, but bitwise subtract
- modifies flags as in ADD, except:
 CF := 1 iff borrow into msbit

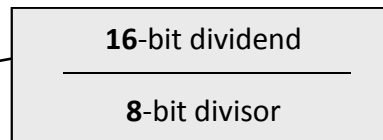
CMP dest, src (Compare)

- like SUB, except dest is not modified
- **modifies FLAGS ONLY !** (to reflect dest – src)

DIV src (Unsigned Integer Divide)

- **src** may be specified using:
 - register, direct or indirect mode
 - **NOT** immediate mode!
- size of **divisor** (8-bit or 16-bit) is determined by size of **src**

DIV src for **8-bit src**:



divide **src** into 16-bit value in **AX**

AL := $AX \div src$ (unsigned divide)

two **8-bit**
results

integer result

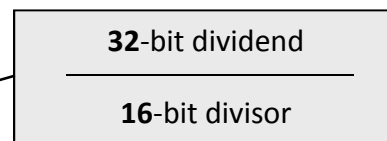
AH := $AX \bmod src$ (unsigned modulus)

integer remainder

flags are undefined after **DIV**

(values may have changed, no meaning)

DIV src for **16-bit src**:



divide **src** into **32-bit** value obtained by concatenating **DX** and **AX** (written **DX:AX**)

AX := $DX:AX \div src$ (unsigned divide)

DX := $DX:AX \bmod src$ (unsigned modulus)

flags are undefined after **DIV**

- what if result is too big to fit in destination?
 - e.g. $AX \div 1$?? $AL = ??$
 - overflow trap – more later!

- in **assignment 2**: use 16-bit source form why?

Learning how to read a reference manual on assembly instructions

- We've seen that instructions often have restrictions – registers, addressing mode

For each instruction – whether in textbook or in processor's programming manual - the permitted operands and the side-effects are given

ADD

O D I S Z A P C

Instruction Formats :

*		*	*	*	*			
---	--	---	---	---	---	--	--	--

ADD reg, reg

ADD reg, immedi

ADD mem, reg

ADD mem, immedi

ADD reg, mem

ADD accum, immedi

Is this permitted : ADD X, Y ?

O D I S Z A P C

MOV

--	--	--	--	--	--	--	--

Instruction Formats :

MOV reg, reg MOV reg, immedi

MOV mem, reg

MOV mem, immedi

MOV reg, mem

MOV mem16, segreg

MOV reg16, segreg MOV segreg, mem16

MOV segreg, reg16

Can you explain why we always initialise our data segment with the following sequence ?

MOV AX, @data

MOV DS, AX

Multiple data declarations on one line:

- separate by a comma
- allocated to successive locations

Examples:

```
                DB          3, 6, 9

Array1         DW   -1, -1, -1, 0

Array2         DB    dup(0)
Array3         DW    dup(?)
```

Strings

- enclose in quotes
- ASCII chars stored in consecutive bytes

```
Message        DB          'Hi Mom!'

MessageNullT   DB          'Hi Mom!', 0
```

Any string to be printed out by DOS functions must be terminated by '\$'

```
DOSMessage     DB          'Hi Mom!', '$'
```

Some Instruction Syntax

- complete instruction on one line
- instruction mnemonic & operands
- **immediate:** state the constant

```
MOV  BX, -1
```

which one is
easier to read?

```
MOV  BX, 0FFFFH
```

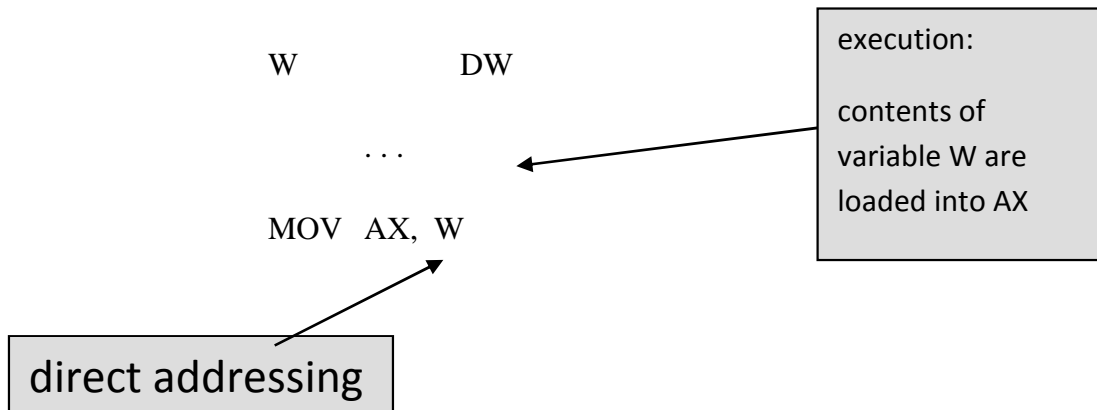
```
MOV  BX, 1111111111111111B
```

- **register:** register name

```
MOV  BX, AX
```

- **direct:**
 - state address of variable as a label reference
 - assembled to include offset to variable (in data segment) as a constant value
 - assembler worries about exact address that will be used **later!**
 - assembler worries about compatibility issues (size of operands)
 - When required, you may explicitly state the type of access required (WORD PTR and BYTE PTR)
 - language issues:
 - how does the assembler know where the data segment is located? **later!**

Example:



A question of style !

EQU directive

VAL EQU 0FFFFh

MOV BX, VAL

CMP AX, VAL

What is the
advantage of
defining a symbol
with EQU ?

EQU versus DW/DB

EQU and Direct Memory look the same but they are NOT !!

```
VAL EQU 0FFFFh
```

```
VAR dw 0FFFFh
```

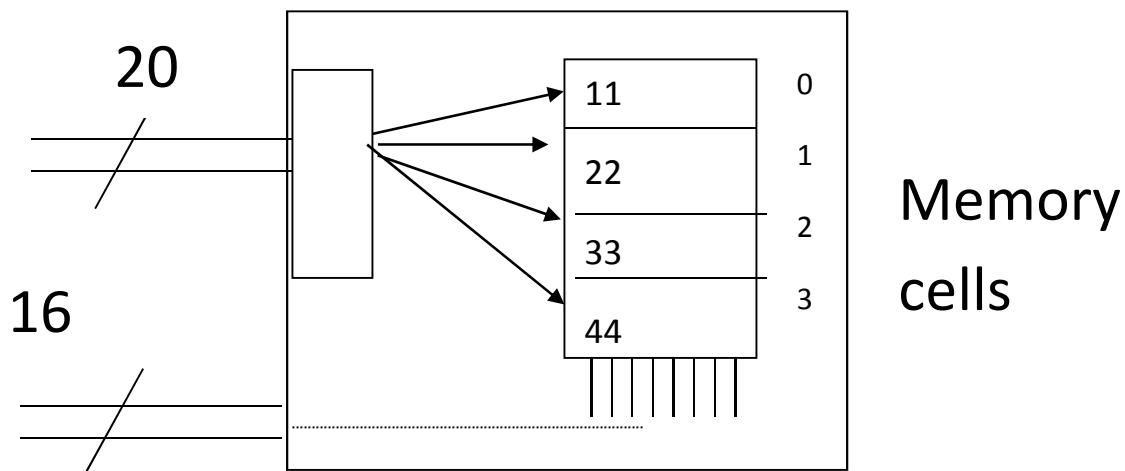
```
MOV BX, VAL
```

```
MOV BX, VAR
```

What's the
difference ??

A question of performance

Just a thought about Data Alignment



80x86 has a 16+ data bus that allows the transfer of bytes, words, dwords, but ...

- word+ transfers can only occur on even-address boundaries

```
MOV AH, [0000]  
MOV AH, [0001]  
MOV AX, [0000]
```

Incurs 1 data transfer
cycle

```
MOV AX, [0001]
```

Incurs 2 data transfer cycles

As a programmer, you don't have to change your program for even or odd locations

- Your program will work regardless of the even/odd alignment of your data variables.

But ...

Why do we use assembly ? Efficiency !

In time-critical applications, you should consider the impact of data alignment on the performance of your code.

1. Organise your data with words first, then bytes so that all words are automatically word-aligned

```
status    db ?  
value     dw ?
```



```
value dw ?  
status db ?
```

2. Use the ALIGN or EVEN directives to force the placement of your word variables at even locations

```
status    db ?  
ALIGN 2  
value     dw      ?
```

Forces assembler to advance its location counter to next address evenly divisible by 2

A question of style !

Basic Coding Conventions

As in high-level languages, coding conventions make programs easier to read, debug and maintain.

```
varName    DB ?  
MAX_LIMIT  EQU 5
```

```
label:  
    MOV AX, BX  
    CMP AX, varName
```

Naming Convention :

1. Labels are lower case, with upper case for joined words
2. EQU symbols are UPPER_CASE
3. Keywords (mnemonics and registers) are upper-case

Indentation :

4. Label is left-justified
5. Instructions are lined up one tab in.
6. Next instruction follows label.


```
varName          DB ?          ; Counter
MAX_LIMIT EQU 5          ; Limit for counter
```

```
; Test for equality
```

```
label:
```

```
    MOV  AX, BX          ; AX is current
    CMP  AX,varName      ; If current < varName
```

Comments

1. Use them
2. Comments on the side to explain instruction
3. Comments on the left for highlight major sections of code.

Control Flow Instructions:

- execution may change value of CS:IP
 - changes address for fetch of next instruction

e.g.: C++ control flow

```
if ( condition )
```

```
{ block T: do this if condition true; }
```

```
else { block F: do this if condition false;
```

```
next_statement ;
```

Why is C++ called a
structured language?

- use data manipulation to decide condition
- if condition is true → continue sequentially into **block T**, at end of block T, must skip to **next_statement**
- if condition false → skip past block T to **block F**, then continue sequentially through block F and on to **next_statement**

need control flow instructions to "skip"

Control Flow Implications of Segmented Memory Model

- instruction execution determined by **CS:IP**
- if control stays in current code segment:
 - **intra**segment control flow
 - only need to modify **IP**
 - must supply (up to) 16-bits of info
- if control passed to address outside of current code segment:
 - **inter**segment control flow
 - must modify **both CS and IP!**
 - must supply 32-bits of information

FOR NOW: we will only be concerned with intrasegment control flow (**only modify IP**)

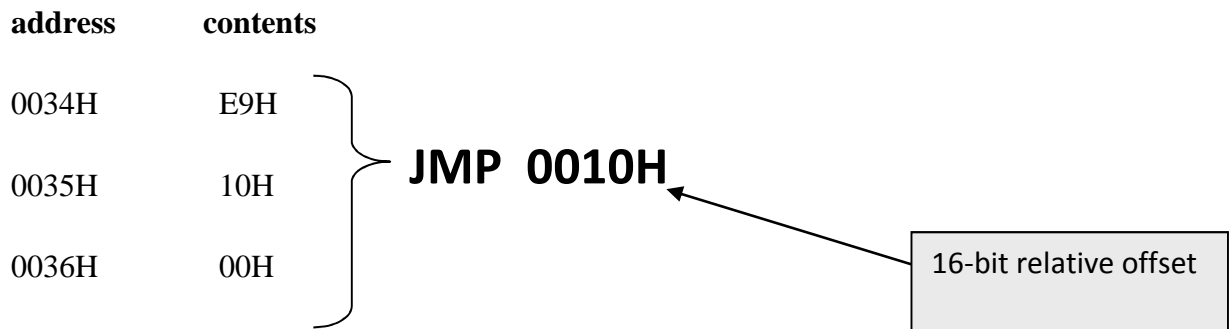
Specifying Control Flow Targets

- must supply operand value used to modify IP
- **absolute** addressing:
 - give 16-bit constant value to replace IP
 - $IP := \text{new value}$
- **relative** addressing:
 - give value to be added to IP (after fetch!)
 - $IP := IP + \text{value}$
 - positive value: jump “forward”
 - negative value: jump “back”
- register/memory **indirect** addressing:
 - specify a register or memory location that contains the value to be used to replace IP
 - $IP := \text{mem}[\text{addrs}]$
 - $IP := \text{register}$

JMP target Unconditional JUMP

- control is **always transferred** to specified target

Relative Addressing Example:



start of fetch: **IP** = 0034H **IR** = ????????

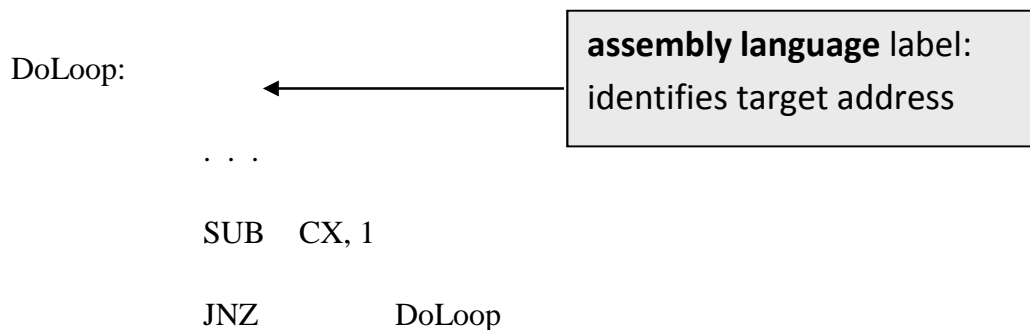
after fetch: **IP** = 0037H **IR** = E9 0010

after execute: **IP** = 0047H **IR** = E9 0010

Conditional Jump:

- specify **condition** in terms of **FLAG values**
 - e.g. **JZ** Jump Zero: Jump iff **ZF** = 1
- if specified condition is **true**: then **jump**!
- if specified condition is **false**: then **continue**!

e.g.: looping example



- many possible conditions
- in many cases: condition and “not” condition are valid instructions

- e.g. JZ Jump Zero
 JNZ Jump Not Zero
- JC Jump Carry (JNC)
- More too!

Conditional Jump Following CMP

- used frequently! Example:

CMP AL, 10

JL **LessThanTen**

. . . ; some code here

LessThanTen:

- CMP dest, src
 - performs dest – src and sets FLAGS
- often useful to think of combination as:

CMP dest, src

J*

jump is taken if “dest * src” condition holds

- **in above example**, jump is taken if $AL < 10$

* is <

Some conditions for *:

JE Jump Equal (JNE)

JL Jump Less Than (JNL)

JLE Jump Less Than or Equal (JNLE)

JG Jump Greater Than

same !?

- processor provides FLAGS to reflect results of (binary) manipulation under both signed and unsigned interpretations
- instructions for different interpretations!
(tests different flags!)

Unsigned		Signed	
JA	Above	JG	Greater
JAЕ	Above or Equal	JGE	Greater or Equal
JB	Below	JL	Less
JBE	Below or Equal	JLE	Less or Equal

(instructions for Not conditions too!)

Is this an issue in Assignment 2 ? (☺)

Suppose AX contains 7FFFH:

Unsigned	
<u>Scenario</u>	
CMP	AX, 8000H
JA	Bigger

Signed	
<u>Scenario</u>	
CMP	AX, 8000H
JG	Bigger

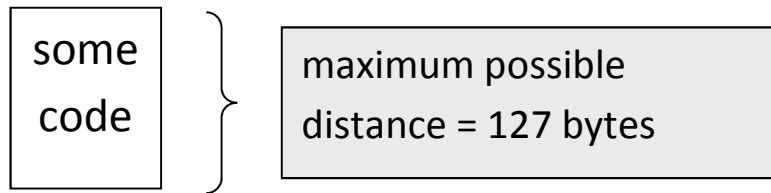
In each scenario, is the jump taken? Why?

Programmer MUST know how binary values are to be interpreted! (e.g. value in AX above)

Conditional jump limitation: uses **8-bit** signed relative offset!

- $IP := IP + (\text{offset } \mathbf{sign\text{-}extended} \text{ to 16-bits})$
- can't jump very far! $-128 \leftrightarrow +127$ bytes

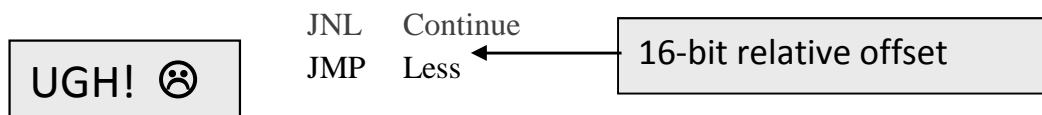
example: JL Less



Less:

MOV ...

One **possible workaround** if distance is greater than 127 bytes (but not the only one!):



Continue:



Less:

MOV ...

LOOP instructions – A special case using a dedicated register

- Action repeated a given number of times
- C++ analogy
for (int =max; i > 0; i++)

MOV CX, max

DoLoop: . . .

SUB CX, 1

JNZ DoLoop

Functionally
equivlanet



Different performance
& code size

```

MOV CX, max

DoLoop:  . .

LOOP    DoLoop

```

LOOP
automatically
decrements CX
 - only works
 with CX

Memory Addressing Modes:

- specify the address of a **memory operand**
- **Two types :**
 1. **direct:** specify address as a constant value
 - address gets encoded as part of instruction
 - **static:** must be known at assembly-time and remains constant through execution !
 2. **indirect:** specify that a register holds the address
 - **dynamic!** address depends on contents of register when instruction is executed
 - may specify a constant value that gets added to address prior to accessing operand (later!)

(Register) Indirect Memory Addressing

- must specify register that holds operand offset
- may only use: BX, SI, DI, BP
- **DS** is default segment for: **BX, SI, DI**
- **SS** is default segment for **BP** (later!)
- syntax: [register]

Indirect Example:

```

W      DW
      ...
MOV BX, OFFSET W

```

“[” & “]” differentiate from
 register mode !

offset of variable is
 treated as a constant
 (text: page 78)

...

MOV AX, [BX]

indirect addressing: value in BX is used as offset to memory operand

- loads AX with contents of W

Why are the [] – brackets needed?

What is the difference?

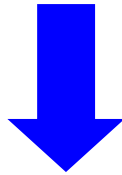
MOV AX, BX

MOV AX, [BX]

Example : LEA

Calculates and load the 16-bit effective address of a memory operand.

MOV BX, **OFFSET W**



Functionally equivalent!

LEA BX, W

Example : Segment Override

Recall :

- **DS** is default segment for: **BX, SI, DI**
- **SS** is default segment for **BP** (later!)

MOV [BX], 3  MOV DS:[BX], 3

MOV [BP], 3  MOV SS:[BP], 3

At times, you may run out of registers and need to use either the index registers or the segment registers outside of their assigned default roles

(eg. duplicating data structures),

MOV SS:[BX], 3

MOV ES:[BX], 3

MOV DS:[BP], 3

An Indirect Addressing Ambiguity :

Indirect addressing : A register holds the offset to the operand

- The offset points to a single memory location

In some cases : No ambiguity in the operand size

eg. MOV AL, [BX]

- AL – 8-bit register
- Interpret [BX] as offset to byte of memory

In some cases, there is ambiguity :

eg . MOV [BX], 1

- Source is immediate value – 8 bit or 16 bit ?
- Move 01 to byte or 0001 to word ?
-

Use syntax to clarify :

MOV BYTE PTR [BX], 1

MOV WORD PTR [BX], 1

The Power of Indirect Addressing:

- **direct mode:** OK for static addresses
- **indirect register mode:** OK for dynamic addresses to access byte or word
 - must have exact address in register
- need more powerful modes for **data structures**

what is a **data structure**?

- **composite structures:** collections of elements

in high-level
language

- **array:** elements are all of the same type
- access using [] ← selector, e.g. X[i]
- **record (struct):** may include elements of different types
 - e.g.


```
struct student {
    string Name;
    int ID; }
```

in high-level
language

- access using “.” selector, e.g. X.Name
- can have:
 - arrays of arrays: multidimensional array
 - arrays of structs e.g. a306Class[i].Name
 - arrays in structs
 - structs in structs
- want **dynamic access** to elements of data structures
 - typically know start address
 - need dynamic specification of offset into structure
 - direct addressing isn’t good enough ☹

Arrays:

- elements are stored sequentially
- all elements are of same type
 - fixed memory requirement for each element
 - constant offset (# or bytes) to start of next element
- 2 relevant cases in programming:
 1. address of array is static
 - writing program for one specific array
 2. address of array is dynamic
 - writing program for “any” array

- e.g. a function that processes an array and accepts the array as an argument
 - different invocations of the function may process different arrays
- 8086 addressing modes exist to support both cases !

What we've looked at so far :

MOV [BX], AX

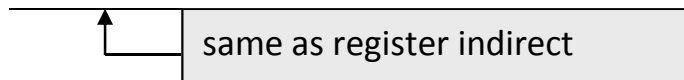
Is the most basic kind of indirect addressing.

Will now be formally called “**register indirect**”

Let's now look at other variations of indirect addressing ...

(Indirect) Indexed or Based Addressing Mode:

- use when accessing array using **static** address
- like register indirect, except also specify a constant
 - e.g. [**BX + constant**]
- during execution, processor uses a temporary register to calculate BX + constant
- accesses memory addressed by BX + constant
- restriction: may only use **BX, SI or DI**



- typical use in accessing an array:
 - start address of array is the constant
 - use register to hold index
 - index = offset (in bytes) to specific element

e.g.

suppose have array of integers declared:

X	DW	; 1 st element of array
	DW	; 2 nd element of array
	...	etc.
SizeOfX:	DW	; number of elements in X

- each element is 2 bytes long!

Code Fragment Example:

; sum the contents of array X into AX

```
MOV AX, 0 ; initialize sum
MOV BX, 0 ; initialize array index
MOV CX, SizeOfX ; get # of elements
```

CheckForDone:

```
CMP CX, 0 ; any elements left to sum?
```

```
JE
```

```
Done
```

address X is static

dynamic! BX
holds offset to
element

```
ADD AX, [ BX + X ] ; sum ith element
```

why "2"?

```
ADD BX, 2 ; adjust index (offset)
```

```
SUB CX, 1 ; one less element
```

```
JMP CheckForDone
```

Done: ; at this point:

When Done:

AX = sum of elements

BX = address of byte that follows

array X in memory

CX = 0

Some **issues** in example:

- **overflow?**

- what if sum exceeds capacity of AX?
- what conditions should be tested? why?

JO vs. **JC** ???

could the control flow be more efficient ?

; set up AX, BX, CX as before

CMP CX, 0 ; any elements left to sum?

JE **Done**

SumElement:

ADD AX, [BX + X] ; sum ith element

ADD BX, 2 ; adjust index (offset)

SUB CX, 1 ; one less element

JNZ **SumElement**

don't do **CMP** – flags
already set after **SUB**!

Done:

- the revised control flow:
 - eliminates 2 instructions from the execution of each loop iteration (CMP/JE)
 - uses one byte less memory (JNZ vs. JMP)
- could the control flow be even more efficient ?
 - adjust CX before executing loop?

; set up AX, BX, CX as before

only
adjust/test CX
at start of loop
execution

CheckForDone:

SUB CX, 1 ; any elements left to sum?

JC **Done**

ADD AX, [BX + X] ; sum i^{th} element

ADD BX, 2 ; adjust index (offset)

JMP CheckForDone

Done:

- or what about:
; set up AX, BX, CX as before

JMP CheckForDone

SumElement:

ADD AX, [BX + X] ; sum i^{th} element

ADD BX, 2 ; adjust index (offset)

CheckForDone:

SUB CX, 1 ; one less element

JAE SumElement

only adjust/test CX **at**
end of loop execution

Done:

(Indirect) Base-Indexed Addressing Mode:

- to access array using **dynamic** address
- like indexed, except use a register instead of a constant
 - e.g. [**BX + SI**]
- during execution, processor uses a temporary register to calculate sum of register values
- accesses memory addressed by sum
- restrictions:
 - one must be **base register**: **BX** (or **BP** \leftarrow later!)
 - one must be **index register**: **SI** or **DI**

- the only legal forms:

Default DS

[BX + SI]

[BX + DI]

base = BX

Default SS

[BP + SI]

[BP + DI]

base = BP

- often, the start address of an array is supplied as an argument to a function
 - put this value in one register
- use other register to hold offset (index) into array

Code Fragment Example:

```
; assume BX = start address of array
;    and CX = number of array elements
; now ... sum the contents of array into AX
```

```
MOV  AX, 0                ; initialize sum
MOV  SI, 0                ; initialize array index
```

CheckForDone:

```
CMP  CX, 0    ; any elements left to sum?

JE   Done

ADD  AX, [ BX + SI ]    ; sum ith element

ADD  SI, 2            ; adjust index (offset)

SUB  CX, 1            ; one less element

JMP  CheckForDone
```

Done: ; at this point:

AX = sum of elements

(Indirect) Based-Indexed with Displacement Addressing Mode:

- like based, except include a constant too
 - e.g. [**BX** + **SI** + **constant**]
- during execution, processor uses a temporary register to calculate sum of values
- accesses memory addressed by sum
- restrictions: same as based mode
- if start address of **array of arrays** is known:
 - use start address as constant
 - use one register as offset to start of sub-array
 - use other register as index
- if start address is not known ??? wing it! ☺
- if **array of structs**:
 - use one register for start address
 - use one register as offset to start of struct
 - use constant to select element
- that's all for addressing modes! ☺

In summary :

Immediate Mode

Register

Memory Direct

Register Indirect

(Indirect) Indexed

(Indirect) Base-Indexed

(Indirect) Base-Indexed with Displacement

How “mechanics” of processor work is one thing – using them in programs is another ☺

Exercises:

Identify the addressing modes of each operand in each of the following instructions

MOV AX, [DX]

MOV list[SI], 00

MOV [BX], [BP][DI]+name

MOV variable, AX

Exercise : (mistake1.asm)

; This program is full of mistakes and/or timebombs.

.model small

.stack 100h

.data

x db ?

y dw ?

.code

main PROC

MOV AX, @data

MOV DS, AX

;1

MOV AX, x

;2

MOV [BX], 01

;3

MOV [BX][BP], CX

;4

```

        CMP     [BP][SI], [BX][SI]

main ENDP

END main

```

Exercise : Find the mistakes in Traversing an Array

```

.model small

.stack 100h

.data

students db 10*82 dup(?)

.code

main PROC

    MOV     BP, students

    MOV     CX, 10

next:

    MOV     AX, [BP]

    INC     BP

    LOOP    next

main ENDP

END main

```

Exercise : Traversing an Array in different ways

Below, an array of student numbers is defined. Show how to traverse the array, ending when the “magic” student number of 0FFFFh is reached.

```

.model small

.stack 100h

.data

```

```

NUM_STUDENTS EQU 2

students dw NUM_STUDENTS dup(0)

terminator dw 0FFFFh

.code

main PROC

    MOV    AX, @data

    MOV    DS, AX

    ; Traversal code

    MOV    AX, 4C00h
    INT    21h

main ENDP

END main

```

Exercise : Accessing an array of structures

Suppose that we now have an array of student records where each record contains the student's name (up to 16 characters) and number (a word).

```

.data

s1 db "Albert$$$$$$$$$$"

    dw 1234

s2 db "Fengji$$$$$$$$$$"

    dw 2345

s3 db "Carol$$$$$$$$$$"

    dw 3432

s4 db "anything$$$$$$$$"

    dw 0FFFFh

```

Write a program to traverse the list, printing out any student whose name begins with 'C'. The list ends with a student number = 0FFFFh

```
MOV    BX, offset s1
```

```
MOV    SI, 0
```

```
next:
```

```
    MOV    AL, [BX][SI]+0
```

```
    CMP    AL, 'C'
```

```
    JNE    testEnd
```

```
    MOV    AH, 9
```

```
    MOV    DX, BX
```

```
    ADD    DX, SI
```

```
    INT    21h
```

```
testEnd:
```

```
    MOV    AX, [BX][SI]+NAME_LEN
```

```
    ADD    SI, NAME_LEN+2
```

```
    CMP    AX, 0FFFFh
```

```
    JNE    next    ....
```

ASIDE :

MASM does offer higher-level support of structures with the **STRUCT** *pseudo-op*

- Pseudo means that it is not part of the processor's instruction set
- Supports programming but ultimately will be translated by assembler into the indirect addressing modes shown previously

student struct

```
stname db 80 dup('$')
```

```
stnum  dw ?
```

student ends

.data

s1 student <"Albert\$", 1234>

s2 student <"Fengji\$", 2345>

s3 student <"Carol\$", 3432>

s4 student <"anything\$", 0FFFF>

.code

main PROC

MOV AX, @data

MOV DS, AX

MOV SI, OFFSET s1

next:

CMP (student PTR [SI]).stname, 'C'

JNE testEnd

MOV AH, 9

MOV DX, SI

INT 21h

testEnd:

MOV AX, (student PTR [SI]).stnum

ADD SI, SIZE student

CMP AX, TERMINATOR

JNE next

MOV AX, 4C00h

INT 21h

main ENDP

END main

Example: Stack

- a data structure
- often used to hold values temporarily

Concept:

recall from 202?

- a “**stack**” is a pile of “*things*”
 - e.g. a stack of papers
- one “thing” is on **top**
 - the rest follow beneath sequentially
- can **add** or **remove** “things” from top of pile
- if **add** a new “thing”, it is now on top
- if **remove** a “thing” from the top, then “thing” that was below it in the pile is now on top
- can look at a thing in the stack if you know the position of the thing relative to the top
 - e.g. 2nd thing is the one below the top

change
state !

read
state !

Stack implementation in a computer:

- stack holds “values”
- reserve a **block of memory** to hold values
- use a **pointer** to point to the value on **top**

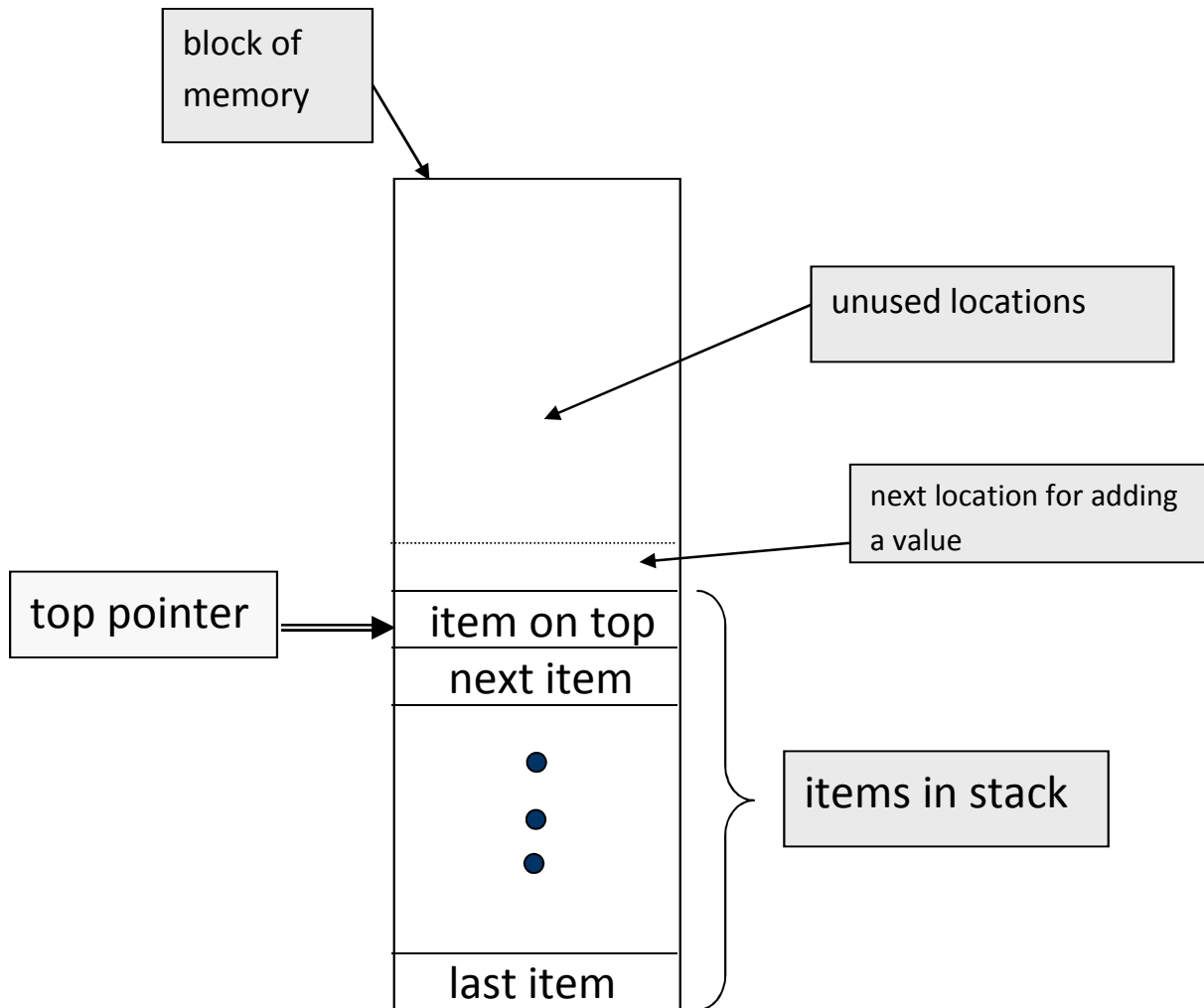
General case of operation:

- pointer points to value on top of stack
- **add:**
 1. adjust pointer to next free (sequential) memory location
 2. copy value to selected memory location (becomes new top)
- **remove:**
 1. copy value of current top

2. adjust pointer to point to value “beneath”
current top (becomes new top)

- **read:**

index from top pointer to i^{th} item



Special Cases:

- **empty stack** (no items in stack)
 - what should top pointer point to?
 - **implementation:** usually points just outside the reserved block – next add will adjust pointer before copying value – will copy into location in the block

- **full stack** (no space to add more items)
 - what should happen if an item added?
 - implementation could:
 - **check** for stack overflow (?)
 - exception handling! ☺ ☹
 - happily **overwrite** memory outside of reserved block (?) ☹ ☺

Issue: Should stack grow from high-to-low addresses (as drawn in picture), or vice versa ?

- **conceptually:** no difference
- **implementation:** typically grows high-to-low
(reasons later!)

Processor has **built-in stack support:**

- called **hardware** or **run-time** stack
- dedicated pointer register: **SS : SP**
 - some instructions use run-time stack **implicitly**
- stack holds 16-bit values
- grows “down” in memory (high-to-low addresses)

(Built-In) Stack Operations:

PUSH adds a new item at top of stack


- must specify 16-bit source operand
 - operand may be register or memory
- effective execution:

$SP := SP - 2$

// adjust pointer

$mem[SP] := operand$ // copy value to top

grows “down”



POP removes item from top of stack

- must specify 16-bit destination operand

- operand may be register or memory
- effective execution:
 $\text{operand} := \text{mem}[\text{SP}]$ // copy value to top

$\text{SP} := \text{SP} + 2$ // adjust pointer

Read an item:

- must index from top
- $[\text{SP} + \text{constant}]$ ☹ illegal!!!
- common solution uses **BP**
- **SS** is default segment register for indirect memory access using **BP** !!

MOV BP, SP ; copy top pointer

access using $[\text{BP} + \text{constant}]$ ☺ legal!

could use any
of BP, BX, SI,
DI

Must **initialize SP** before using stack op's:

.stack size

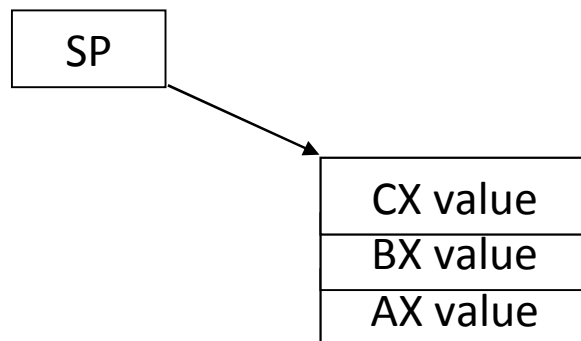
- assembler reserved specified number of bytes as block of memory to use for stack
- directive is translated into instruction to loader to initialize **SS** and **SP** !!
- **SP** points at byte just above (high address!) last byte reserved for stack

Example: suppose need to save registers:

PUSH AX

PUSH BX

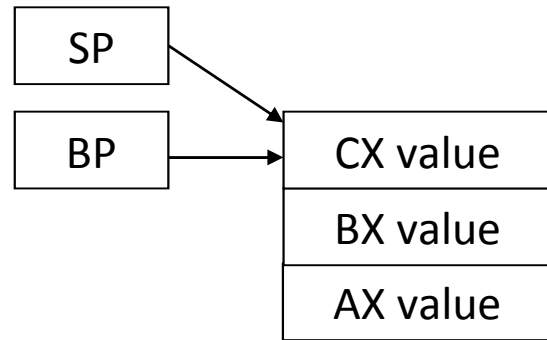
PUSH CX



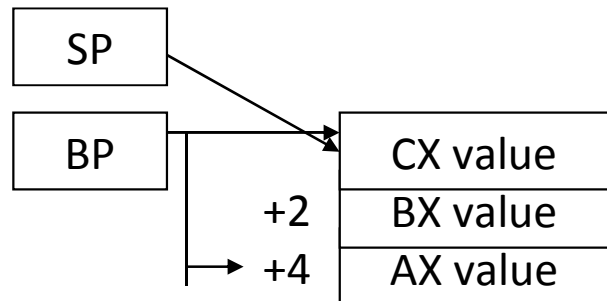
now ... to access saved AX value

could POP values off until AX value reached, OR:

MOV BP, SP



MOV ..., [BP + 4] ; read saved AX

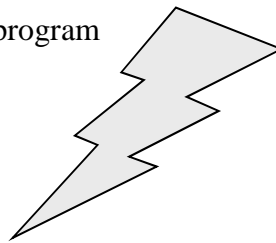


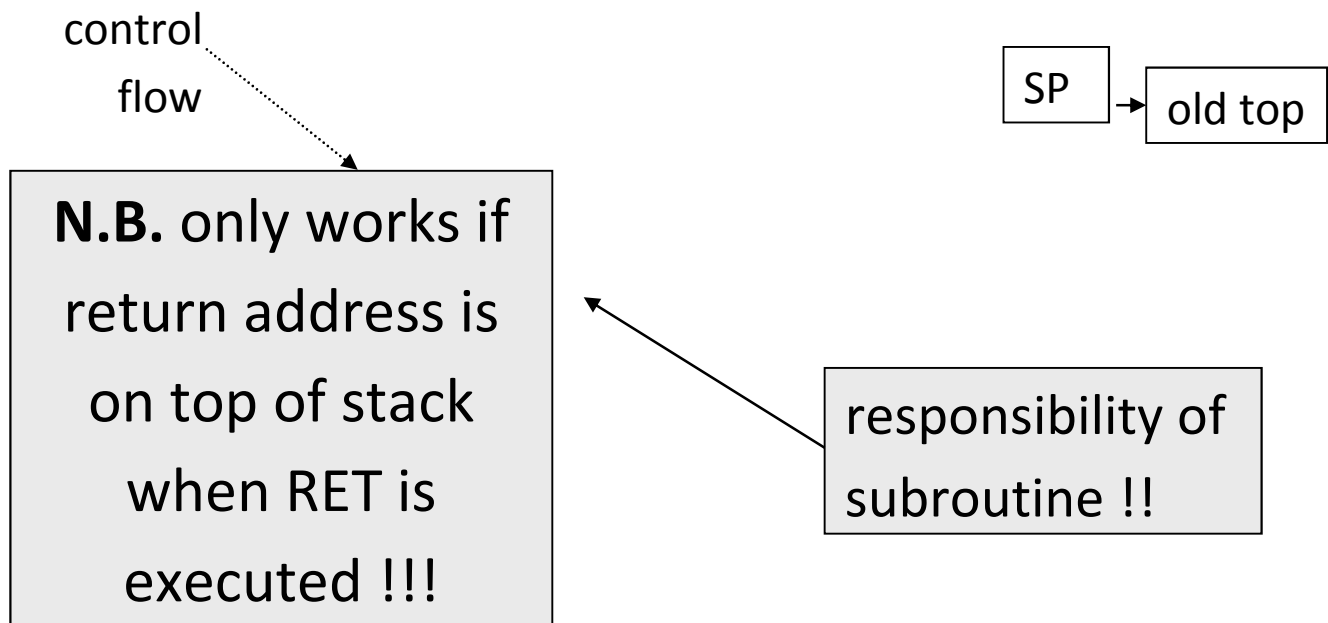
- **Subroutines**

- often need to perform the same program activity many times
- high-level language:
function, procedure, method

- assembly language:
subroutine

- want ability to:
 - **encapsulate** program activity in program text
 - **invoke** the activity from elsewhere in program
 - **control flow!**
 - pass control to the activity
 - execute the activity
 - return control to the invocation point





Parameters: increase **generality** of an activity

- allow invocations to behave differently
- allows **caller** (the code that invokes) and **callee** (the invoked activity) to communicate information

Example:

- could have a dedicated subroutine that displays the value 245

void Display245(); nice, but

- could generalize by adding a parameter
 - allow a 16-bit value to be displayed to be communicated at invocation

void DisplaySigned(word Value);

more general than
Display245

- could further generalize by adding a parameter to communicate the base for displaying Value

**Note: other notes continue in next sections,
students can download from links mentioned in
website.**