Intel 8086 Instructions

CO 2103 Assembly Language

- This is a compilation of slides on common Intel 8086 Instructions.
- The slides are mostly taken from the slides used by my colleague Dr Md Mahmud Hasan, who taught this module before me.

Data Transfer

Data Transfer Instructions

- Operand Types
- Instruction Operand Notation
- Direct Memory Operands
- **`MOV`** Instruction
- Zero & Sign Extension
- **`XCHG`** Instruction
- Direct-Offset Instructions

E.g. MOV Instruction

- Move from source to destination. Syntax:

  MOV *destination,source*

- No more than one memory operand permitted
- **CS**, **EIP**, and **IP** cannot be the destination
- No immediate to segment moves

```
.data
count BYTE 100
wVal  WORD 2
.code
mov bl,count
mov ax,wVal
mov count,al

mov al,wVal          ; error
mov ax,count         ; error
mov eax,count        ; error
```

MOV Instruction

- **`MOV`** ***`target,source`***
  - reg, reg
  - mem, reg
  - reg, mem
  - mem, immed
  - reg, immed
- Sizes of both operands must be the same
- *reg* can be any non-segment register except IP cannot be the target register
- **`MOV`**'s between a segment register and memory or a 16-bit register are possible

Operand Types

- Three basic types of operands:
  - Immediate – a constant integer (8, 16, or 32 bits)
    - value is encoded within the instruction
  - Register – the name of a register
    - register name is converted to a number and encoded within the instruction
  - Memory – reference to a location in memory
    - memory address is encoded within the instruction, or a register holds the address of a memory location

Instruction Operand Notation

Operand	Description
r8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
r16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
r32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
r/m8	8-bit operand which can be an 8-bit general register or memory byte
r/m16	16-bit operand which can be a 16-bit general register or memory word
r/m32	32-bit operand which can be a 32-bit general register or memory doubleword
mem	an 8-, 16-, or 32-bit memory operand

Example MOV Instructions

```
b db 4Fh
w dw 2048

mov bl,dh
mov ax,w
mov ch,b
mov al,255
mov w,-100
mov b,0
```

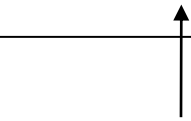
- When a variable is created with a define directive, it is assigned a default size attribute (byte, word, etc)
- You can assign a size attribute using LABEL

```
LoByte LABEL BYTE
aWord DW 97F2h
```

Direct Memory Operands

- A direct memory operand is a named reference to storage in memory
- The named reference (label) is automatically dereferenced by the assembler

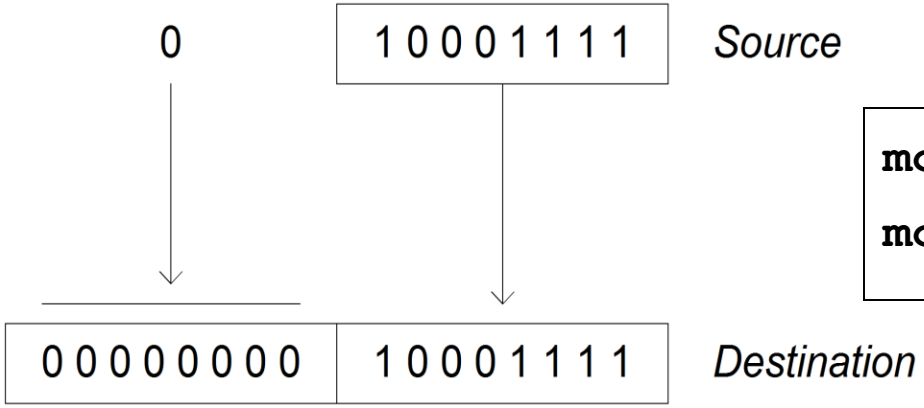
```
.data
var1 BYTE 10h
.code
mov al,var1   ; AL = 10h
mov al,[var1] ; AL = 10h
```



alternate format

Zero Extension

- When you copy a smaller value into a larger destination, the **`MOVZX`** instruction fills (extends) the upper half of the destination with zeros.

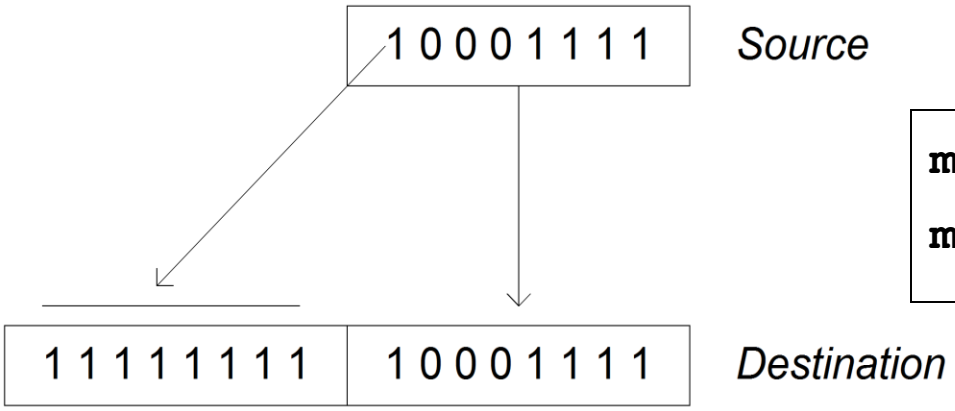



```
mov bl,10001111b
movzx ax,bl   ; zero-extension
```

The destination must be a register

Sign Extension

- The **`MOVSX`** instruction fills the upper half of the destination with a copy of the source operand's sign bit.




```
mov bl,10001111b
movsx ax,bl   ; sign extension
```

The destination must be a register

Your turn …

- Explain why each of the following **MOV** statements are invalid:

```
.data
bVal   BYTE   100
bVal2  BYTE   ?
wVal   WORD   2
dVal   DWORD  5
.code
mov ds,45      ; a.
mov esi,wVal   ; b.
mov eip,dVal   ; c.
mov 25,bVal    ; d.
mov bVal2,bVal       ; e.
```

XCHG Instruction

- **XCHG** exchanges the values of two operands. At least one operand must be a register. No immediate operands are permitted.

```
.data
var1 WORD 1000h
var2 WORD 2000h
.code
xchg ax,bx    ; exchange 16-bit regs
xchg ah,al    ; exchange 8-bit regs
xchg var1,bx  ; exchange mem, reg
xchg eax,ebx  ; exchange 32-bit regs

xchg var1,var2      ; error: two memory operands
```

eXCHanGe

- **`XCHG target, source`**
  - reg, reg
  - reg, mem
  - mem, reg
- **`MOV`** and **`XCHG`** cannot perform memory to memory moves
- This provides an efficient means to swap the operands
  - No temporary storage is needed
  - Sorting often requires this type of operation
  - This works only with the general registers

Direct-Offset Operands

- A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

```
.data
arrayB BYTE 10h,20h,30h,40h
.code
mov al,arrayB+1                ; AL = 20h
mov al,[arrayB+1]              ; alternative notation
```

Q: Why doesn't **arrayB+1** produce **11h**?

Direct-Offset Operands (cont)

- A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

```
.data
arrayW  WORD 1000h,2000h,3000h
arrayD  DWORD 1,2,3,4
.code
mov ax,[arrayW+2]           ; AX = 2000h
mov ax,[arrayW+4]           ; AX = 3000h
mov eax,[arrayD+4]          ; EAX = 00000002h
```

```
; Will the following statements assemble and run?
mov ax,[arrayW-2]           ; ??
mov eax,[arrayD+16]         ; ??
```

Addresses with Displacements

```
b db 4Fh, 20h, 3Ch
w dw 2048, -100, 0

mov bx, w+2
mov b+1, ah
mov ah, b+5
mov dx, w-3
```

- Type checking is still in effect

- The assembler computes an address based on the expression
- *NOTE: These are address computations done at assembly time*

  MOV ax, b-1

  will not subtract 1 from the value stored at b

Your turn ...

- Write a program that rearranges the values of three doubleword values in the following array as: **3,1,2**.

```
.data
arrayD DWORD 1,2,3
```

- Step1: copy the first value into **EAX** and exchange it with the value in the second position.

```
mov eax,arrayD
xchg eax,[arrayD+4]
```

- Step 2: Exchange **EAX** with the third array value and copy the value in **EAX** to the first array position.

```
xchg eax,[arrayD+8]
mov  arrayD,eax
```

Evaluate this …

- We want to write a program that adds the following three bytes:

```
.data
myBytes BYTE 80h,66h,0A5h
```

- What is your evaluation of the following code?

```
mov al,myBytes
add al,[myBytes+1]
add al,[myBytes+2]
```

- What is your evaluation of the following code?

```
mov ax,myBytes
add ax,[myBytes+1]
add ax,[myBytes+2]
```

- Any other possibilities?

Evaluate this … (cont)

```
.data
myBytes BYTE 80h,66h,0A5h
```

- How about the following code. Is anything missing?

```
movzx ax,myBytes
mov   bl,[myBytes+1]
add   ax,bx
mov   bl,[myBytes+2]
add   ax,bx          ; AX = sum
```

Yes: Move zero to **BX** before the **MOVZX** instruction.

Arithmetic and Flags

Arithmetic Instructions

- INC and DEC
- ADD and SUB
- NEG Instruction
- Implementing Arithmetic Expressions
- Flags affected by ADD and SUB

Arithmetic Instructions

ADD *dest, source*

SUB *dest, source*

INC *dest*

DEC *dest*

NEG *dest*

- *Operands must be of the same size*
- *source* can be a general register, memory location, or constant
- *dest* can be a register or memory location
  - except operands cannot both be memory

INC and DEC

- INC
  - adds 1 to destination operand
- DEC
  - subtracts 1 from destination operand
- (both)
  - operand can be either a register or variable
  - Carry flag not affected

INC and DEC Examples

```
.data
myWord  WORD 1000h
myDword DWORD 10000000h

.code
inc myWord    ; 1001h
dec myWord    ; 1000h
inc myDword   ; 10000001h

mov ax,00FFh
inc ax ; AX = 0100h
mov ax,00FFh
inc al ; AX = 0000h
```

Your turn ...

- Show the value of the destination operand after each of the following instructions executes:

```
.data
myByte BYTE 0FFh, 0

.code
mov al,myByte ; AL =
mov ah,[myByte+1]    ; AH =
dec ah ; AH =
inc al ; AL =
dec ax ; AX =
```

ADD and SUB Instructions

- **`ADD`** destination, source
  - Logic: *destination* ← *destination* + source
- **`SUB`** destination, source
  - Logic: *destination* ← *destination* – source
- Same operand rules as for the **`MOV`** instruction

ADD and SUB Examples

```
.data
var1 DWORD 10000h
var2 DWORD 20000h

.code  ; ---EAX---
mov eax,var1  ; 00010000h
add eax,var2  ; 00030000h
add ax,0FFFFh ; 0003FFFFh
add eax,1     ; 00040000h
sub ax,1      ; 0004FFFFh
```

ADD Instruction

```
ADD destination, source
```

- Adds source operand to destination operand
- Affects the Carry, Overflow, Sign and Zero flags
- source operand can be register, immediate value, or memory
- destination operand can be register or memory
- only one operand can be a memory operand

ADD Instruction Examples

```
.data
membyte         db 25
memword         dw 36
doubleVal       dd 12340000h

.code
add   al,5
add   bx,ax
add   eax,edx
add   membyte,al
add   memword,bx
add   doubleVal,edx
```

SUB Instruction

```
SUB destination, source
```

- Subtracts source operand from destination operand
- Affects the Carry, Overflow, Sign and Zero flags
- source operand can be register, immediate value, or memory
- destination operand can be register or memory
- only one operand can be a memory operand

SUB Instruction Examples

```
.data
membyte         db 25
memword         dw 36
doubleVal       dd 12340000h

.code
sub   al,5
sub   bx,ax
sub   eax,edx
sub   membyte,al
sub   memword,bx
sub   doubleVal,edx
```

NEG (negate) Instruction

- Converts the operand to its twos complement.
- It's a good idea to check the Overflow flag after performing a NEG operation.
- Operand can be a register or memory operand.

NEG (negate) Examples

```
.data
valB BYTE -1
valW WORD +32767

.code
mov al,valB   ; AL = -1
neg al        ; AL = +1
neg valW      ; valW = -32767

mov  bl,00000001b
neg  bl       ; bl = 11111111
mov  ah,11001101b
neg  ah       ; ah = 00110011
mov  al,-128
neg  al      ; al = 80h, OF=1
```

```
;Suppose AX contains -32,768 and we apply
NEG to it. Will the result be valid?
```

Implementing Arithmetic Expressions

- HLL compilers translate mathematical expressions into assembly language. You can do it also. For example:

Rval = -Xval + (Yval – Zval)

```
Rval DWORD ?
Xval DWORD 26
Yval DWORD 30
Zval DWORD 40
.code
mov eax,Xval
neg eax          ; EAX = -26
mov ebx,Yval
sub ebx,Zval     ; EBX = -10
add eax,ebx
mov Rval,eax     ; -36
```

Your turn ...

- Translate the following expression into assembly language. Do not permit **`Xval`**, **`Yval`**, or **`Zval`** to be modified:

  `Rval = Xval - (-Yval + Zval)`

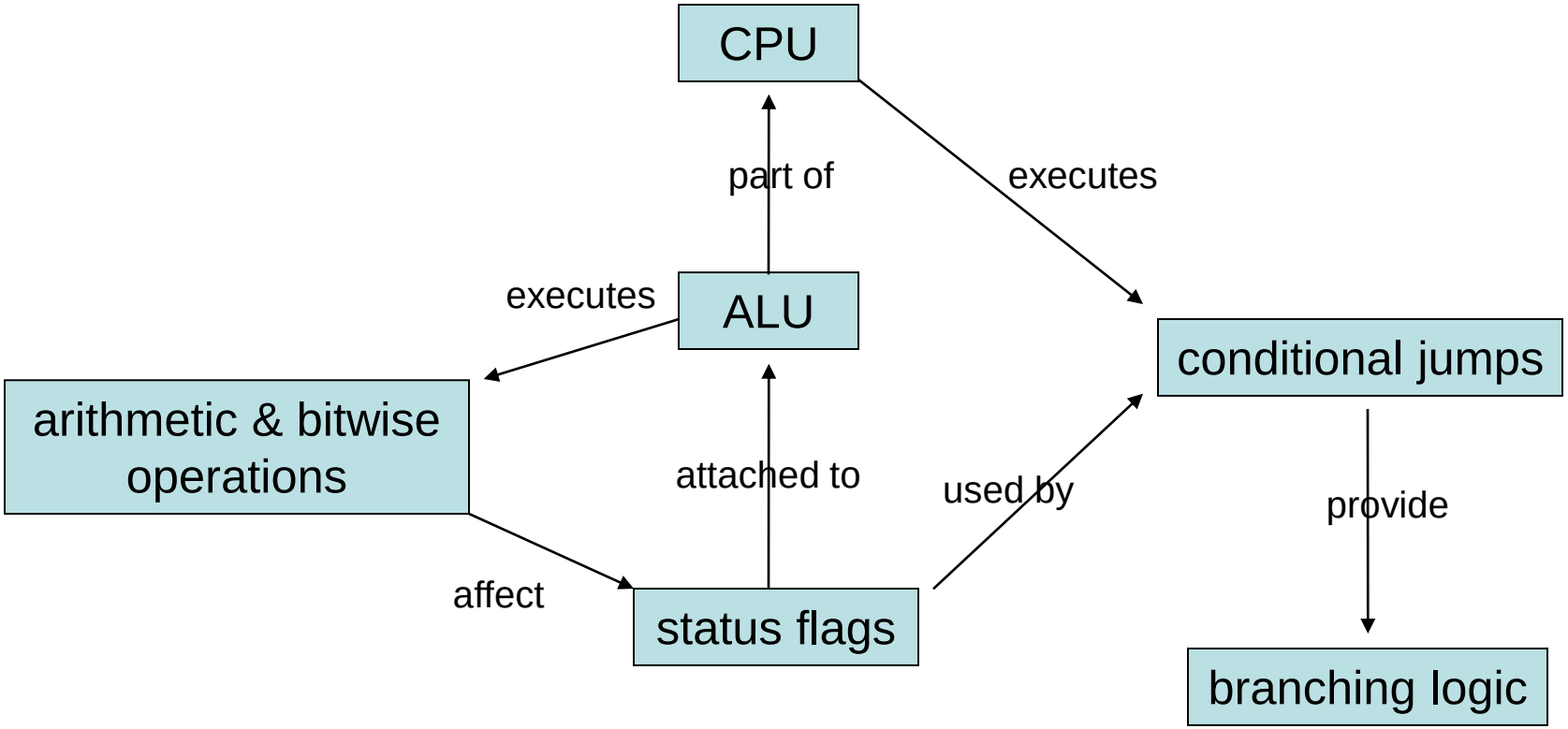
- Assume that all values are signed doublewords.

```
mov ebx,Yval
neg ebx
add ebx,Zval
mov eax,Xval
sub ebx
mov Rval,eax
```

Flags Affected by Arithmetic

- The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations
  - based on the contents of the destination operand
- Essential flags:
  - Zero flag – destination equals zero
  - Sign flag – destination is negative
  - Carry flag – unsigned value out of range
  - Overflow flag – signed value out of range
- The **`MOV`** instruction never affects the flags.

Concept Map



You can use diagrams such as this to express the relationships between assembly language concepts.

Flags Affected by ADD and SUB

After the instruction has been executed ...

- If the destination is zero, the Zero flag is set
- If the destination is negative, the Sign flag is set
- If there was a carry out of the highest bit of the destination, the Carry flag is set
- If the signed result is too small or too large to fit in the destination, the Overflow flag is set

Zero Flag (ZF)

- Whenever the destination operand equals Zero, the Zero flag is set.

```
mov cx,1
sub cx,1        ; CX = 0, ZF = 1
mov ax,0FFFFh
inc ax          ; AX = 0, ZF = 1
inc ax          ; AX = 1, ZF = 0
```

A flag is set when it equals 1.
A flag is clear when it equals 0.

Sign Flag (SF)

- The Sign flag is set when the destination operand is negative. The flag is clear when the destination is positive.

```
mov cx,0
sub cx,1        ; CX = -1, SF = 1
add cx,2        ; CX = 1, SF = 0
```

- The sign flag is a copy of the destination's highest bit:

```
mov al,0
sub al,1                ; AL = 11111111b, SF = 1
add al,2                ; AL = 00000001b, SF = 0
```

Carry Flag (CF)

- The Carry flag is set when the result of an operation generates an unsigned value that is out of range (too big or too small for the destination operand).

```
mov al,0FFh
add al,1      ; CF = 1, AL = 00

; Try to go below zero:
mov al,0
sub al,1      ; CF = 1, AL = FF
```

- In the second example, we tried to generate a negative value. Unsigned values cannot be negative, so the Carry flag signaled an error condition.

Your turn ...

- For each of the following marked entries, show the values of the destination operand and the Sign, Zero, and Carry flags:

```
mov ax,00FFh
add ax,1      ; AX=        SF=  ZF=  CF=
sub ax,1      ; AX=        SF=  ZF=  CF=
add al,1      ; AL=        SF=  ZF=  CF=
mov bh,6Ch
add bh,95h    ; BH=        SF=  ZF=  CF=

mov al,2
sub al,3      ; AL=        SF=  ZF=  CF=
```

Signed Overflow

- The Overflow flag is set when the signed result of an operation is invalid or out of range.
- Signed overflow occurs when adding two signed integers, if and only if…
  - both operands are positive, or both operands are negative
  - and the sign of the sum is opposite to the sign of the values being added

```
mov  al,+127
add  al,+1     ; AL = 80h (-128), Overflow
```

Signed Overflow Examples

```
mov   al,+127
add   al,+1          ; AL = 80h (-128), Overflow

mov   dx,-32768      ; DX = 8000h (-32768)
add   dx,-1          ; DX = 8001h, Overflow

mov   dx,-32768      ; DX = 8000h (-32768)
sub   dx,1           ; DX = 8001h, Overflow
```

Your turn ...

- What will be the values of the Carry and Overflow flags after each operation?

```
mov al,-128
neg al ; CF = OF =

mov ax,8000h
add ax,2       ; CF = OF =

mov ax,0
sub ax,2       ; CF = OF =

mov al,-5
sub al,+125    ; CF = OF =
```

Logic

Boolean Instructions

Operation	Description
AND	Result is 1 only when both input bits are 1.
OR	Result is 1 when either input bit is 1.
XOR	Result is 1 only when the input bits are different (called exclusive-OR).
NOT	Result is the reverse of the input bit (in other words, 1 becomes 0, and 0 becomes 1).
NEG	Convert a number to its twos complement.
TEST	Perform an implied AND operation on the destination operand, setting the flags appropriately.
BT, BTR, BTC, BTS	Copy bit n from the source operand to the Carry flag and toggle/clear/set the same bit in the source operand.
CMP	Compare two operands, setting the flags appropriately.

AND Instruction

- Performs a Boolean **AND** operation between each of the corresponding bits in two operands and places the result in the first operand:

```
and op-1, op-2
function: op-1 ← op-1 and op-2

op-1:      1 1 0 1 0 0 1 1
op-2:      0 1 0 0 1 1 0 1
result:    0 1 0 0 0 0 0 1 (op-1)
```

OR Instruction

- Performs a Boolean **OR** operation between each of the corresponding bits in two operands and places the result in the first operand:

```
or op-1, op-2
function: op-1 ← op-1 or op-2

op-1:      1 1 0 1 0 0 1 1
op-2:      0 1 0 0 1 1 0 1
result:    1 1 0 1 1 1 1 1
```

XOR Instruction

- Performs a Boolean **XOR** operation between each of the corresponding bits in two operands and places the result in the first operand:

```
xor op-1, op-2
function: op-1 ← op-1 xor op-2

op-1:     1 1 0 1 0 0 1 1
op-2:     0 1 0 0 1 1 0 1
result:   1 0 0 1 1 1 1 0
```

Uses of XOR Instruction

- **XOR**ing any bit with **1** toggles (inverts) the bit:

```
mov  al,10110011b
xor  al,11111111b    ; AL = 01001100
```

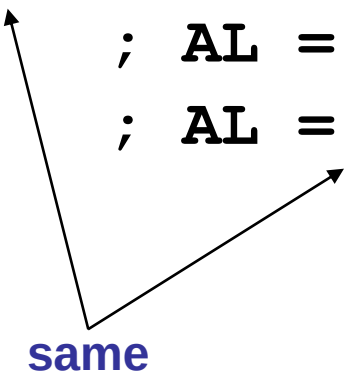
- **XOR**ing any bit with **0** leaves the bit unchanged:

```
mov  al,10110011b
xor  al,00000000b    ; AL = 10110011
```

Uses of XOR Instruction

- The **XOR** operation reverses itself. It is particularly well suited to inverting graphic images.

```
mov   al,10110011b
xor   al,10101100b    ; AL = 00011111
xor   al,10101100b    ; AL = 10110011
```



NOT Instruction

- Performs a Boolean **NOT** (inversion) operation on each of the bits in the operand.

```
mov  bh,11001101b
not  bh                    ; bh = 00110010
```

Program Control

TEST Instruction

- Performs an implied **AND** operation between corresponding bits in the two operands. Only the flags are affected.

```
mov  bl,00000001b
test bl,00000001b ; ZF = 0

mov  ah,11001101b
test ah,00110010b ; ZF = 1
```

TEST Instruction

- Example: the following will jump to the label `PrinterReady` if either bit 0, 1, or 2 is set in the AL register:

```
test al,00000111b
jnz  PrinterReady
.
.
PrinterReady:
```

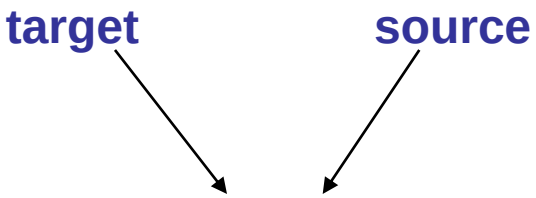
BT, BTC, BTR, and BTS

Table 2. BT, BTC, BTR, and BTS Instructions

Instruction	Description
BT op1,n	Copy bit n from the first operand to the Carry flag.
BTC op1,n	Copy bit n from the first operand to the Carry flag and complement (toggle) the bit in the first operand.
BTR op1,n	Copy bit n from the first operand to the Carry flag and reset (clears) the bit in the first operand.
BTS op1,n	Copy bit n from the first operand to the Carry flag and set the bit in the first operand.

CMP Instruction

- Performs an implied subtraction of the source operand from the target operand. Only the flags are affected.




```
mov   ax,5
cmp   ax,5      ; ax == 5
cmp   ax,7      ; ax < 7
cmp   ax,3          ; ax > 3
```

Flags Set by the CMP Instruction

Unsigned:

CMP Results	CF	ZF
Destination < source	1	0
Destination = source	0	1
Destination > source	0	0

Signed:

CMP Results	ZF	SF, OF
Destination < source	?	SF <> OF
Destination = source	1	?
Destination > source	0	SF = OF

Example 1. Using the CMP Instruction

```
Label1:
    mov  al,5
    cmp  al,10              ; CF=1, ZF=0
Label2:
    mov  ax,1000
    mov  cx,1000
    cmp  cx,ax              ; CF=0, ZF=1
Label3:
    mov  si,105
    cmp  si,0               ; CF=0, ZF=0
```

Table 4. Jumps Based on General Comparisons

Mnemonic	Description	Flags / Registers
JZ	Jump if zero	ZF = 1
JE	Jump if equal	ZF = 1
JNZ	Jump if not zero	ZF = 0
JNE	Jump if not equal	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if no carrry	CF = 0
JCXZ	Jump if CX = 0	CX = 0
JECXZ	Jump if ECX = 0	ECX = 0
JP	Jump if parity even	PF = 1
JNP	Jump if parity odd	PF = 0

Table 5. Jumps Based on Unsigned Comparisons

Mnemonic	Description	Flag(s)
JA	Jump if above (if op1 > op2)	CF = 0 and ZF = 0
JNBE	Jump if not below or equal (if op1 not <= op2)	CF = 0 and ZF = 0
JAE	Jump if above or equal (if op1 >= op2)	CF = 0
JNB	Jump if not below (if op1 not < op2)	CF = 0
JB	Jump if below (if op1 < op2)	CF = 1
JNAE	Jump if not above or equal (if op1 not >= op2)	CF = 1
JBE	Jump if below or equal (if op1 <= op2)	CF = 1 or ZF = 1
JNA	Jump if not above (if op1 not > op2)	CF = 1 or ZF = 1

Table 6. Jumps Based On Signed Comparisons

Mnemonic	Description	Flag(s)
JG	Jump if greater (if op1 > op2)	SF = OF and ZF = 0
JNLE	Jump if not less than or equal (if op1 not <= op2)	SF = OF and ZF = 0
JGE	Jump if greater than or equal (if op1 >= op2)	SF = OF
JNL	Jump if not less (if op1 not < op2)	SF = OF
JL	Jump if less (if op1 < op2)	SF <> OF
JNGE	Jump if not greater than or equal (if op1 not >= op2)	SF <> OF
JLE	Jump if less than or equal (if op1 <= op2)	ZF = 1 or SF <> OF
JNG	Jump if not greater (if op1 not > op2)	ZF = 1 or SF <> OF
JS	Jump if signed (op1 is negative)	SF = 1
JNS	Jump if not signed (op1 is positive)	SF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if no overflow	OF = 0

Finding the Smallest of Three Integers

```
1:          mov   small,al      ; assume AL is the smallest
2:          cmp   small,bl      ; if small <= BL then
3:          jbe   L1            ;   jump to L1
4:          mov   small,bl      ; else move BL to small
5:    L1:   cmp   small,cl      ; if small <= CL then
6:          jbe   L2            ;   jump to L2
7:          mov   small,cl      ; else move CL to small
8:    L2:
```

Ex 3. Single Character Encryption Using XOR

```
XORVAL = 239                ; any value between 0-255
.code
main proc
     mov   ax,@data
     mov   ds,ax
L1:  mov   ah,6            ; direct console input
     mov   dl,0FFh         ; don't wait for character
     int   21h             ; AL = character
     jz    L2              ; quit if ZF = 1 (EOF)
     xor   al,XORVAL
     mov   ah,2            ; write to output
     mov   dl,al
     int   21h
     jmp   L1              ; repeat the loop
L2:  mov   ax,4C00h        ; return to DOS
     int   21h
main endp
end  main
```

Example 4. Testing the Isalpha Procedure (1 of 2)

```
title Test Alphabetic Input                (ISALPHA.ASM)
; This program reads and displays characters
; until a non-alphabetic character is entered.
.model small
.stack 100h
.code
main proc
    mov  ax,@data
    mov  ds,ax
L1: mov  ah,1          ; input a character
    int  21h           ; AL = character
    call Isalpha       ; test value in AL
    jnz  exit          ; exit if not alphabetic
    jmp  L1            ; continue loop
exit:
    mov   ax,4C00h     ; return to DOS
    int   21h
main endp
```

Example 4. Testing the Isalpha Procedure (2 of 2)

```
; Isalpha sets ZF = 1 if the character
; in AL is alphabetic.

Isalpha proc
    push ax                 ; save AX
    and  al,11011111b       ; convert to uppercase
    cmp  al,'A'             ; check 'A'...'Z' range
    jb   B1
    cmp  al,'Z'
    ja   B1
    test ax,0               ; ZF = 1
B1: pop ax                  ; restore AX
    ret
Isalpha endp

end main
```

Table 3. Do-While Loop Example

High Level	Low Level

do while (a < b)
  <statement-1>
  <statement-2>
  .
  .
enddo
	L1: if (a >= b) then
	  jump to L2
	  endif
	  statement-1
	  statement-2
	  .
	  .
	  jump to L1
	L2: (continue here)

Largest and Smallest Signed Array Values

- Pseudocode for a simple loop that finds the largest and smallest values in an array:

```
count = 0
smallest = array[count]
largest = array[count]
CX = 6
do while CX > 0
  AX = array[count]
  if (AX < smallest) then smallest = AX
  if (AX > largest) then largest = AX
  count = count + 1
  CX = CX - 1
end do
```

Ex 5. Largest and Smallest Signed Array Values (1 of 4)

- Here is the algorithm implemented in an assembly language program:

```
.data
largest  dw  ?
smallest dw  ?

array dw  -1,2000,-421,32767,500,0,-26,-
  4000
arrayCount = ($-array) / (type array)
largemsg db "Largest value:  ",0
smallmsg db "Smallest value: ",0
```

Ex 5. Largest and Smallest Signed Array Values (2 of 4)

```
.code
extrn Clrscr:proc, Crlf:proc, Writeint_signed:proc, \
     Writestring:proc

main proc
    mov  ax,@data           ; initialize DS
    mov  ds,ax
    mov  di,offset array
    mov  ax,[di]            ; get first element
    mov  largest,ax         ; initialize largest
    mov  smallest,ax        ; initialize smallest
    mov  cx,arrayCount      ; loop counter

A1: mov  ax,[di]            ; get array value
    cmp  ax,smallest        ; [DI] >= smallest?
    jge  A2                 ; yes: skip
    mov  smallest,ax        ; no: move [DI] to smallest
```

Ex 5. Largest and Smallest Signed Array Values (3 of 4)

```
A2: cmp   ax,largest      ; [DI] <= largest?
    jle   A3              ; yes: skip
    mov   largest,ax      ; no: move [DI] to largest

A3: add   di,2            ; point to next number
    loop A1               ; repeat the loop until CX=0

    call ShowResults
    mov   ax,4C00h        ; return to DOS
    int   21h
main endp
```

Example 5. Largest and Smallest Signed Array Values (4 of 4)

```
; Display the largest and smallest values.
ShowResults proc
    mov   dx,offset largemsg            ; largest
    call  Writestring
    mov   ax,largest
    call  Writeint_signed
    call  CrLf

    mov   dx,offset smallmsg            ; smallest
    call  Writestring
    mov   ax,smallest
    call  Writeint_signed
    call  CrLf
    ret
ShowResults endp
end main
```

SET*condition* Instruction

- The SET*condition*n instruction sets a byte operand to 1 if a given condition is true, or to 0 if the condition is false. Here are examples:

```
setz    al                ; set AL to 1 if ZF = 1
sete    al                ; set AL to 1 if ZF = 1
setnc   myFlag            ; set myFlag to 1 if CF = 0
setae   myFlag            ; set myFlag to 1 if CF = 0
setge   byte ptr [si]     ; set [si] to 1 if SF = OF
setnbe byte ptr [si]      ; set [si] to 1 if SF = OF
```

JMP and LOOP Instructions

- JMP Instruction
- LOOP Instruction
- LOOP Example
- Summing an Integer Array
- Copying a String

JMP Instruction

- JMP is an unconditional jump to a label that is usually within the same procedure.
- Syntax: JMP *target*
- Logic: EIP ← *target*
- Example:

```
top:
.
.
jmp top
```

A jump outside the current procedure must be to a special type of label called a global label (see Section 5.8.3 for details).

LOOP Instruction

- The LOOP instruction creates a counting loop
- Syntax: LOOP *target*
- Logic:
  - ECX ← ECX – 1
  - if ECX > 0, jump to *target*
- Implementation:
  - The assembler calculates the distance, in bytes, between the current location and the offset of the target label. It is called the relative offset.
  - The relative offset is added to EIP.

LOOP Example

The following loop calculates the sum of the integers 5 + 4 + 3 +2 + 1:

```
offset      machine code          source code
00000000    66 B8 0000            mov   ax,0
00000004    B9 00000005           mov   ecx,5

00000009    66 03 C1          L1: add   ax,cx
0000000C    E2 FB                 loop  L1
0000000E
```

When LOOP is assembled, the current location = 0000000E. Looking at the LOOP machine code, we see that –5 (FBh) is added to the current location, causing a jump to location 00000009:

00000009 ← 0000000E + FB

Your turn . . .

If the relative offset is encoded in a single byte,

(a) what is the largest possible backward jump?

(b) what is the largest possible forward jump?

- –128
- +127

Your turn . . .

What will be the final value of AX?

10

```
mov ax,6
mov ecx,4
L1:
inc ax
loop L1
```

How many times will the loop execute?

4,294,967,296

```
mov ecx,0
X2:
inc ax
loop X2
```

Nested Loop

If you need to code a loop within a loop, you must save the outer loop counter's ECX value. In the following example, the outer loop executes 100 times, and the inner loop 20 times.

```
.data
count DWORD ?
.code
mov ecx,100   ; set outer loop count
L1:
mov count,ecx ; save outer loop count
mov ecx,20    ; set inner loop count
L2:    .
    .
    loop L2   ; repeat the inner loop
mov ecx,count ; restore outer loop count
loop L1       ; repeat the outer loop
```

Summing an Integer Array

The following code calculates the sum of an array of 16-bit integers.

```
.data
intarray WORD 100h,200h,300h,400h
.code
    mov edi,OFFSET intarray    ; address of intarray
    mov ecx,LENGTHOF intarray  ; loop counter
    mov ax,0    ; zero the accumulator
L1:
    add ax,[edi]         ; add an integer
    add edi,TYPE intarray      ; point to next integer
loop L1 ; repeat until ECX = 0
```

Your turn . . .

What changes would you make to the program on the previous slide if you were summing a doubleword array?

Copying a String

The following code copies a string from source to target.

```
.data
source  BYTE  "This is the source string",0
target  BYTE  SIZEOF source DUP(0),0

.code
mov  esi,0              ; index register
mov  ecx,SIZEOF source          ; loop counter
L1:
mov  al,source[esi]             ; get char from source
mov  target[esi],al             ; store it in the target
inc  esi                ; move to next character
loop L1          ; repeat for entire string
```

Your turn . . .

Rewrite the program shown in the previous slide, using indirect addressing rather than indexed addressing.

Conditional Loops (LOOPZ and LOOPE)

The LOOPZ and LOOPE instructions allow a loop to continue if ZF =1 and CX > 0. The following example scans an array looking for a nonzero value:

```
.data
intarray dw 0,0,0,0,1,20,35,-12,66,4,0
ArraySize = ($-intarray) / (type intarray)
.code
  mov   bx,offset intarray ; point to the array
  sub   bx,2               ; back up one position
  mov   cx,ArraySize       ; repeat ArraySize times
next:
  add   bx,2               ; point to next entry
  cmp   word ptr [bx],0    ; compare value to zero
  loopz next               ; while ZF=1 and CX > 0
```

Conditional Loop Example

Be careful not to modify the flags prior to to a conditional LOOP instruction:

```
next:
  cmp   [bx],ax   ; compare array value to zero
  add   bx,2      ; point to next entry, ZF = 0
  loopz next      ; error: Zero flag has changed
```

LOOPNZ and LOOPNE

The LOOPNZ and LOOPNE instructions continue the loop as long as ZF=0 and CX > 0. The following loop scans an array until a positive integer is found:

```
.data
bytearray db  -3,-6,-1,-10,10,30,40,4
ArraySize  = ($-array)
.code
   mov   si,offset bytearray-1
   mov   cx,ArraySize
next:
   inc   si
   test  byte ptr [si],10000000b
   loopnz next
```

High-Level Logic Structures

- IF Statement
- Compound IF Using OR
- Compound IF Using AND
- Loops
  - WHILE Loop
  - DO-WHILE Loop
  - REPEAT-UNTIL Loop
- CASE Structure

Simple IF Statement

Pseudocode:

```
if (AX = op2) then
  mov flag,1
  mov count,0
end if
```

Implementation:

```
      cmp   ax,op2
      je    True
      jmp   EndIf
True:
      mov   flag,1
      mov   count,0
EndIf:
```

Simple IF Statement (alternate form)

Reversing the condition enables you to shorten the implementation by one instruction.

Pseudocode:

```
if (AX = op2)
  mov flag,1
  mov count,0
end if
```

Implementation:

```
    cmp   ax,op2
    jne   EndIf
    mov   flag,1
    mov   count,0
EndIf:
```

Compound IF Statement Using OR

Pseudocode:

```
if (AL > op1)
  or (AL >= op2)
  or  (AL = op3)
  mov  flag,1
  mov  count,5
end if
```

Implementation:

```
     cmp   al,op1
     jg    L1
     cmp   al,op2
     jge   L1
     cmp   al,op3
     je    L1
     jmp   L2
L1:  mov   flag,1
     mov   count,5
L2:
```

Compound IF Statement Using AND

Pseudocode:

```
if (AL > op1)
  and (AL >= op2)
  and (AL = op3)
  mov  flag,1
  mov  count,5
end if
```

Implementation:

```
    cmp  al,op1
    jng  L1
    cmp  al,op2
    jnge L1
    cmp  al,op3
    jne  L1
    mov  flag,1
    mov  count,5
L1:
```

Combining AND and OR

Pseudocode:

```
if (AL > op1)
  and (AL >= op2)
  or (AL = BL)
  mov  flag,1
  mov  count,5
end if
```

Implementation:

```
     cmp  al,op1
     jng  L1
     cmp  al,op2
     jnge L1
     jmp  L2
L1:  cmp  al,bl
     jne  L3
L2:  mov  flag,1
     mov count,5
L3:
```

Ex 6. Do-While Loop Example

Assembly Code

```
while:
    cmp  op1,op2
    jnl  L3
    <statement1>
    cmp op2,op3
    jne L1
    <statement2>
    <statement3>
  jmp L2
L1: <statement4>
L2: jmp while
L3:
```

Pseudocode

```
do while (op1 < op2)

  <statement1>
  if op2 = op3 then

    <statement2>
    <statement3>
  else
    <statement4>
  endif
enddo
```

6.4.4. Repeat-Until Structure

Given the following structure...

```
repeat:
  <statement1>
  <statement2>
  <statement3>
until (op1 = op2) or (op1 > op3)
```

Code this loop in assembly language:

Repeat-Until Loop

```
 1:        repeat:
 2:           <statement1>
 3:           <statement2>
 4:           <statement3>
 5:        test1:
 6:           cmp  op1,op2      ; if op1 = op2 then
 7:           je   endif        ;   exit the loop
 8:        test2:               ; else
 9:           cmp  op1,op3      ;   if op1 <= op3 then
10:           jng   repeat      ;     repeat the loop
11:        endif:               ;   end if
12:                             ; end if
```

6.4.5 CASE Structure

The CASE structure allows a multiway branch by comparing a single value to a list of values. This example is taken from Pascal:

```
case input of
  'A'  :Process_A;
  'B'  :Process_B;
  'C'  :Process_C;
  'D'  :Process_D;
end;
```

Implementing the Case in Assembly Language

The most direct implementation involves CMP and conditional jump instructions:

```
mov   al,input          ; case al of
cmp   al,'A'            ;   'A':
je    Process_A         ;      goto Process_A
cmp   al,'B'            ;   'B':
je    Process_B         ;      goto Process_B
cmp   al,'C'            ;   'C':
je    Process_C         ;      goto Process_C
cmp   al,'D'            ;   'D':
```

But you can also call procedures and jump to an ending label:

Ex 9. Case Structure Using Procedure Calls

```
    mov    al,input          ; case al of
    cmp    al,'A'            ;   'A':
    jne    L1
    call   Process_A         ;       call Process_A
    jmp    L4                ;       jump to L4
L1: cmp    al,'B'            ;   'B':
    jne    L2
    call   Process_B         ;       call Process_B
    jmp    L4                ;       jump to L4
L2: cmp    al,'C'            ;   'C':
    jne    L3
    call   Process_C         ;       call Process_C
    jmp    L4                ;       jump to L4
L3: cmp    al,'D'            ;   'D':
    jne    L4
    call   Process_D         ;       call Process_D
L4:                          ;   end
```

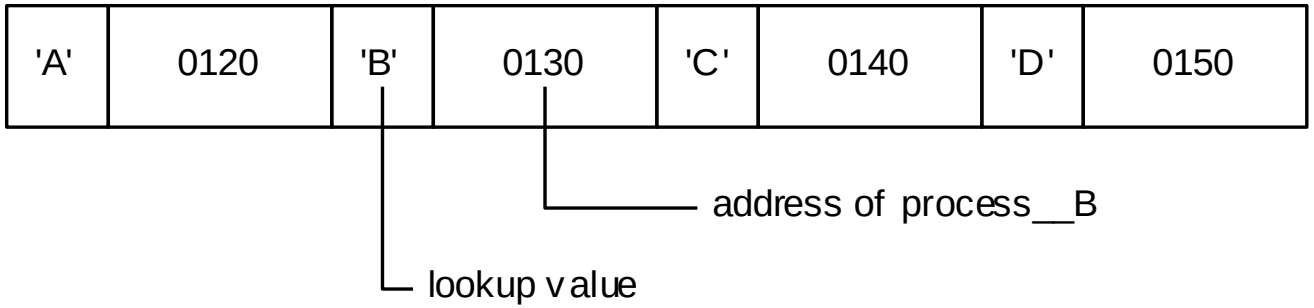
Ex 10. Table Containing Offsets of Procedures

A table may contain lookup values and corresponding procedure offsets. Unlike a case structure, this table can be modified at runtime.

```
CaseTable  db    'A'              ; lookup value
           dw    process_A        ; address of procedure
           db    'B'
           dw    process_B
           db    'C'
           dw    Process_C
           db    'D'
           dw    Process_D
```

Fig 1. Table of Procedure Offsets

Let's assume that four different procedures are located at offsets 0120, 0130, 0140, and 0150, respectively. This is how the table would look in memory:



Ex 11. Case Table Lookup Using a Loop

When AL matches a character in the table, the following code calls the procedure whose address is located in the table immediately following the matching character:

```
NumberOfEntries = 4
input dw ?
.code
      mov   al,input              ; value to be found
      mov   bx,offset CaseTable   ; point BX to the table
      mov   cx,NumberOfEntries    ; loop counter
  L1: cmp   al,[bx]               ; match found?
      jne   L2                    ; no: continue
      call  word ptr [bx+1]       ; yes: call the procedure
      jmp   L3                    ; exit the search
  L2: add   bx,3                  ; point to next entry
      loop  L1                    ; repeat until CX = 0
  L3:
```

Table 7. Conditional Jumps, Flag Settings (Question 9)

	Instruction	Overflow	Sign	Zero	Carry	Jump Taken?
a.	JNZ	0	0	1	0	No
b.	JA	1	0	1	0	
c.	JNB	1	0	1	0	
d.	JBE	0	0	1	0	
e.	JGE	1	1	0	0	
f.	JNLE	0	1	0	1	
g.	JNS	0	0	1	0	
h.	JNG	1	1	0	1	
i.	JE	1	0	1	0	
j.	JNAE	1	0	1	0	