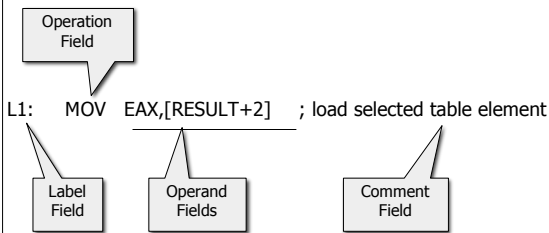


ELEC3730 Embedded Systems Lecture 10: Intel Architecture Assembly Language

- Assembly code format
- Defining Data
- Instruction set essentials

The Four Fields of a Line of Code in Assembly Language



Instruction Format Examples

- No operands
 - STC ; set Carry flag
 - One operand
 - INC EAX ; increment register
 - INC mybyte ; increment memory
 - Two operands
 - ADD EBX, ECX ; add a register to a register
 - SUB mybyte, 25 ; subtract constant from memory
 - ADD EAX, 36 ; add an expression to a register
 - [...] Refers to data at address
- ```
ORG 1234h
xyzy: DWORD 5678h ; the address of this word is 1234 (hex)

MOV EAX,[xyzy] ; loads 5678 (hex) into register EAX
MOV EAX,xyzy ; loads 1234 (hex) into register EAX
```

---

---

---

---

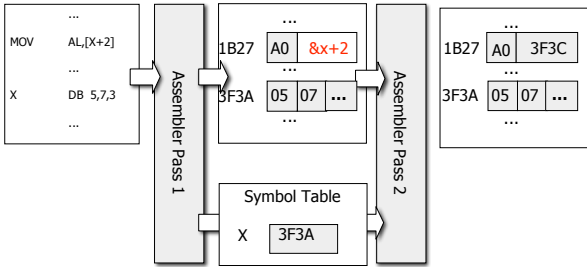
---

---

---

---

## Two Passes of an Assembler




---

---

---

---

---

---

---

---

## Registers - Conventions and Restrictions

- Arithmetic & data movement
  - EAX – accumulator - general purpose
  - EBX – base address of variable or function
  - ECX – counter for repeating or looping instructions
  - EDX – data (eg. 64 bit computations, I/O)
- Data & instruction addresses
  - EBP – base (or frame) pointer for variables passed on stack
  - ESP – stack pointer (address of bottom of stack)
  - EIP – instruction pointer (address of next instruction to be executed)
- Data offsets (for arrays)
  - ESI – source index
  - EDI – destination index
- Address of base locations for all references to memory
  - CS – code segment (base location of all executable instructions)
  - DS – data segment (base location for all variables)
  - SS – stack segment (base location of stack)
- Control flags – specific bits in EFLAGS register
  - DF – direction flag for string operations
  - IF – enable/disable interrupts
  - CF – carry flag (Result of unsigned arithmetic operation too large)
  - SF – sign flag (1 = negative result; 0 = positive result)
  - ZF – zero flag (1 = zero result; 0 = nonzero result)

---

---

---

---

---

---

---

---

## Data Definition Statement

- Sets aside storage in memory for a variable
- Syntax: [name] directive initializer
- BYTE, SBYTE: 8-bit unsigned integer; 8-bit signed integer
- WORD, SWORD: 16-bit unsigned & signed integer
- DWORD, SDWORD: 32-bit unsigned & signed integer

Each of the following defines a single byte of storage:

```
.data
value1 BYTE 'A' ; character constant
value2 BYTE 255 ; largest unsigned byte
value3 SBYTE -128 ; smallest signed byte
value4 BYTE ? ; uninitialized byte
```

A variable name is a data label that implies an offset (an address)

All data in the data segment is continuous in memory, stored one byte after another.

```
.data
list1 BYTE 10,20,30,40
var1 BYTE 20 DUP(0) ; 20 bytes, all equal to zero
```

---

---

---

---

---

---

---

---

## 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

- Variable names are references to offsets within the data segment
  - Suppose *var1* were located at offset 10400h

```
.data
var1 BYTE 10h ; var1 = offset 10400h
.code
mov al, var1 ; AL = 10h
mov al, [00010400h] ; AL = 10h
```

---

---

---

---

---

---

---

---

---

---

## MOV Instruction

- Move from source to destination.  
Syntax: **MOV destination,source**
- No more than one memory operand permitted
- Both operands must be same size

```
.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
```

- Operands from memory - size sometimes unclear
- May be inferred: **MOV AL,[EBX]**
- AL is 8 bits, so register EBX contains the address of an 8-bit memory operand.
- May be explicit: **INC DWORD [EBX]**
- Ambiguous without "DWORD"!

---

---

---

---

---

---

---

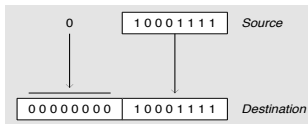
---

---

---

## Zero Extension

- Copy a smaller value into a larger destination ?
- Use MOVZX instruction
- Fills (extends) the upper half of the destination with zeros.
- Destination must be a register



```
mov bl,10001111b
movzx ax,bl ; zero-extension
 ; ax = 0000 0000 1000 1111b
```

---

---

---

---

---

---

---

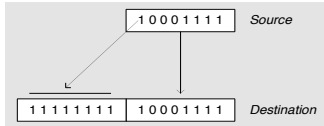
---

---

---

## Sign Extension

The MOVSB 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
; ax = 1111 1111 1000 1111b
mov bl,01111111b
movsx ax,bl ; sign extension
; ax = 0000 0000 0111 1111b
```

---

---

---

---

---

---

---

---

---

---

## 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
```

---

---

---

---

---

---

---

---

---

---

## Direct-Offset Operands

- A constant offset may be 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
```

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

---

---

---

---

---

---

---

---

---

---

## INC and DEC Instructions

- **INC destination** ;destination ← destination + 1
- **DEC destination** ;destination ← destination - 1
- Operand may be register or memory
- Does not affect the carry flag
- May affect the sign, zero, or overflow flags

```
.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
```

---

---

---

---

---

---

---

---

## ADD and SUB Instructions

- **ADD destination, source** ;destination ← destination + source
- **SUB destination, source** ;destination ← destination - source
- Same operand rules as for the MOV instruction

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code
mov eax,var1 ; ---EAX---
add eax,var2 ; 00010000h
add eax,var2 ; 00030000h
add ax,0FFFFh ; 0003FFFFh
add eax,1 ; 00040000h
sub ax,1 ; 0004FFFFh
```

---

---

---

---

---

---

---

---

## NEG (negate) Instruction

- Reverses the sign of an operand (two's complement).
- Operand can be a register or memory operand.

```
.data
valB BYTE -1
valW WORD +32767
.code
mov al,valB ; AL = -1
neg al ; AL = +1
neg valW ; valW = -32767
```

---

---

---

---

---

---

---

---

## Indexed Operands (Pointers)

An indexed operand adds a constant to a register to generate an effective address. There are two notational forms:

`[label + reg]`                      `label[reg]`

```
.data
arrayW WORD 1000h,2000h,3000h
ptrW DWORD arrayW ; Initialise Pointer
.code
mov esi,0
mov ax,[arrayW + esi] ; AX = 1000h
add esi,2
add ax,[arrayW + esi] ; AX = 1000h+2000h
mov esi,ptrW ; Use pointer
mov ax,[esi] ; AX = 1000h
```

---

---

---

---

---

---

---

---

---

---

## 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
```

---

---

---

---

---

---

---

---

---

---

## LOOP Instruction

- The LOOP instruction creates a counting loop
- Syntax: **LOOP target**
- Logic: ECX ← ECX - 1, if ECX > 0, jump to target
- Implementation:
  - Assembler calculates relative offset between the current location and target label.
  - Relative offset: -128 to 127 bytes. (Max approx 42 instructions)
  - The relative offset is added to EIP.

| 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 ; dec ECX and JMP if NZ |
| 0000000E |              |                                 |

-5 (FBh) is added to the current location, causing a jump to location 09:  
 $09 \leftarrow 0E + FB \quad (09 \leftarrow 14 + -5)$

---

---

---

---

---

---

---

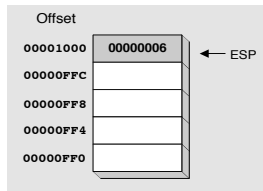
---

---

---

## Runtime Stack

- Managed by the CPU, using two registers
  - SS (stack segment) – holds a segment descriptor
  - ESP (stack pointer) \* - holds 32-bit offset of location on stack



\* SP in Real-address mode

---

---

---

---

---

---

---

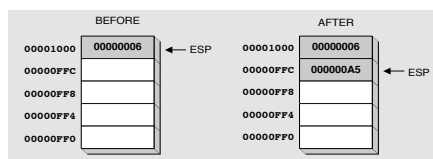
---

---

---

## PUSH Operation (1 of 2)

- A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location pointed to by the stack pointer.
- For example, the instruction "push A5"



---

---

---

---

---

---

---

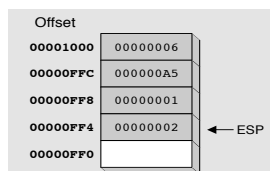
---

---

---

## PUSH Operation (2 of 2)

- This is the same stack, after pushing two more integers:
- Instructions: push 1  
push 2



The stack grows downward. The area below ESP is always available (unless the stack has overflowed).

---

---

---

---

---

---

---

---

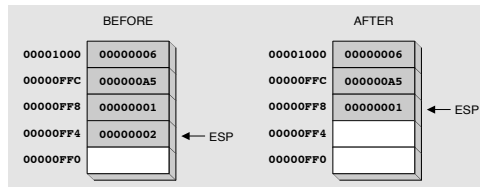
---

---

## POP Operation

- Copies value at stack[ESP] into a register or variable.
- Adds 4 to ESP
  - Example

```
pop eax ; eax = 2
```




---

---

---

---

---

---

---

---

---

---

## Example: Nested Loop

Can use push & pop to save and restore the counter value of an outer loop

```

mov ecx,100 ; set outer loop count
L1: ; begin the outer loop
 push ecx ; save outer loop count

 mov ecx,20 ; set inner loop count
 L2: ; begin the inner loop
 ;
 ;
 loop L2 ; repeat the inner loop

 pop ecx ; restore outer loop count
 loop L1 ; repeat the outer loop

```

Save/restore everything! (enter/exit interrupt service routine)  
 PUSHAD ; Pushes EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI  
 POPAD ; Pops EDI, ESI, EBP, skip, EBX, EDX, ECX, EAX

---

---

---

---

---

---

---

---

---

---

## CALL and RET Instructions

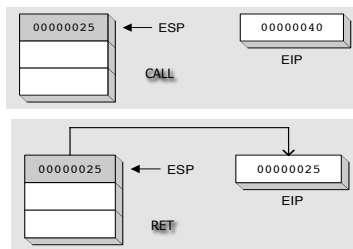
- The CALL instruction calls a procedure
  - pushes offset of next instruction on the stack
  - copies the address of the called procedure into EIP
- The RET instruction returns from a procedure
  - pops top of stack into EIP

```

main PROC
00000020 call MySub
00000025 mov eax,ebx
.
main ENDP

MySub PROC
00000040 mov eax,edx
.
ret
MySub ENDP

```




---

---

---

---

---

---

---

---

---

---

## Boolean Operations

- **Syntax:**

```
AND destination, source
OR destination, source
XOR destination, source
NOT destination
```

Ex: Convert character in AL to upper case (clear bit 5).

```
mov al, 'a' ; AL = 0110 0001b
and al, 11011111b ; AL = 0100 0001b
```

Ex: Convert byte into ASCII character (set bits 4 and 5)

```
mov al, 6 ; AL = 0000 0110b
or al, 00110000b ; AL = 0011 0110b
```

---

---

---

---

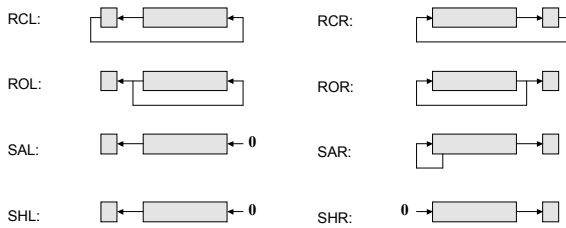
---

---

---

---

## Shift/Rotate Instructions: `opc dst, count`



---

---

---

---

---

---

---

---

## TEST Instruction

- Nondestructive AND operation between operands
  - No operands are modified, but the Zero flag is affected.
- Example: jump to a label if either bit 0 or bit 1 in AL is set.

```
test al, 00000011b
jnz ValueFound
```

- Example: jump to a label if neither bit 0 nor bit 1 in AL is set.

```
test al, 00000011b
jz ValueNotFound
```

---

---

---

---

---

---

---

---

## CMP Instruction

- Nondestructive subtraction of source from destination
  - *No operands are modified, but the Zero, Carry flags affected.*
- Example: destination == source

```
mov al,5
cmp al,5 ; ZF = 1, al = 5
```

- Example: destination < source

```
mov al,4
cmp al,5 ; CF=1, al = 4
```

- Example: destination > source

```
mov al,6
cmp al,5 ; ZF = 0, CF = 0; al = 6
```

## Commonly-Used Conditional Jump Instructions

| Compare  | Mnemonic(s) | Jump if . . .                         | Determined by . . . |
|----------|-------------|---------------------------------------|---------------------|
| equality | JE (JZ)     | Equal (Zero)                          | ZF==1               |
|          | JNE (JNZ)   | Not Equal (Not Zero)                  | ZF==0               |
| unsigned | JB (JNAE)   | Below (Not Above or Equal)            | CF==1               |
|          | JBE (JNA)   | Below or Equal (Not Above)            | CF==1    ZF==1      |
|          | JAE (JNB)   | Above or Equal (Not Below)            | CF==0               |
|          | JA (JNBE)   | Above (Not Below or Equal)            | CF==0 && ZF==0      |
| signed   | JL (JNGE)   | Less than (Not Greater than or Equal) | SF!=OF              |
|          | JLE (JNG)   | Less than or Equal (Not Greater than) | SF!=OF    ZF==1     |
|          | JGE (JNL)   | Greater than or Equal (Not Less than) | SF==OF              |
|          | JG (JNLE)   | Greater than (Not Less than or Equal) | SF==OF && ZF==0     |

## Applications

Ex: Jump to label if unsigned EAX > EBX

```
cmp eax,ebx
ja Larger
```

Ex: Jump to label if signed EAX > EBX

```
cmp eax,ebx
jg Greater
```

Ex: Copy larger of AX and BX into variable named `Large`

```
mov Large,bx
cmp ax,bx
jna Next
mov Large,ax
Next:
```

