Microprocessor System Design

Omid Fatemi

80x86 Addressing modes

(omid@fatemi.net)

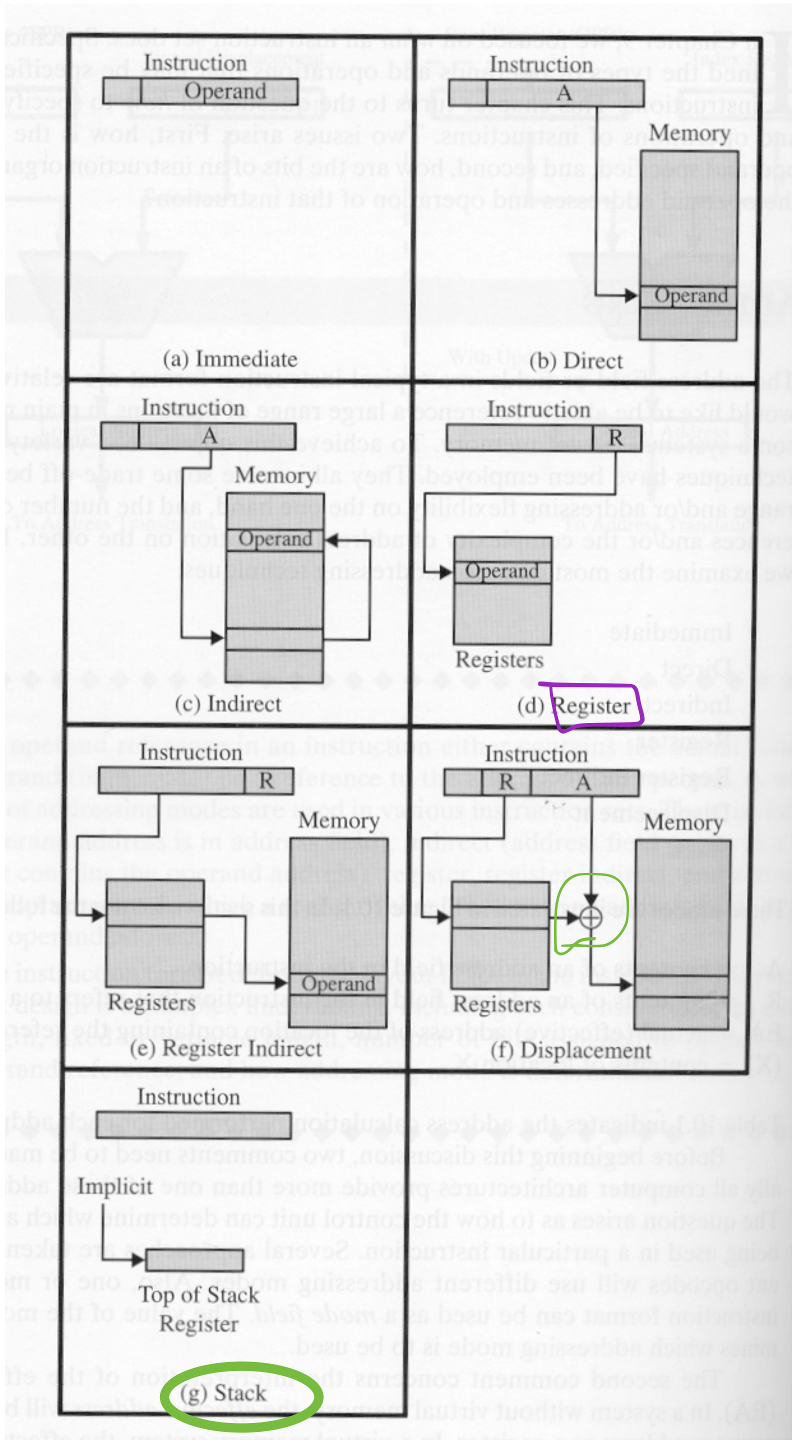
Review

- Programs for 80x86
- Machine language, Assembly, …
- Registers, segments
- Instruction set
- Simple program
- Logical, physical address
- Stack

Addressing Modes

- Operands in an instruction
  - Registers AX
  - Numbers → immediate 12H Means immediate, a number
  - Memory
    - » Direct addressing [3965] [] show address in memory
    - » Register indirect [BX]
    - » Based relative addressing mode [BX+6], [BP]-10
    - » Indexed relative addressing mode [SI+5], [DI]-8
    - » Based indexed addressing mode [BX+SI+5]

برو سراغ رجیستر فایل، محتوای ان ادرس یک حافظه‌ای است مقدار اون رو از حافظه بردار



Operand types

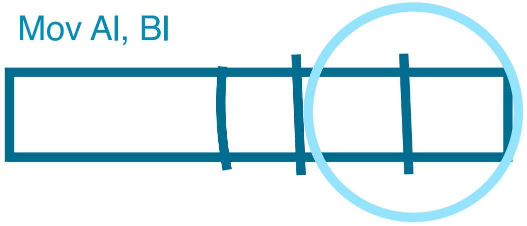
1) Register - Encoded in instruction

because the memory is what slows us down

- Fastest executing
- No bus access (in instr. queue)
- Short instruction length

because we need less bits to show the address of registers because they are fewer in number

Mov Al, Bl



The address is part of the instruction

2) Immediate - Constant encoded in instruction

- 8 or 16 bits
- No bus access (in instr. queue)
- Can only be source operand

because we can't put the result in a number, we need it to be a memory space or a register

3) Memory – in memory, requires bus transfer

- Can require computation of address
- Address of operand *DATA* is Called

EFFECTIVE ADDRESS

Effective Address

- Computed by $\boxed{\text{EU}} \longrightarrow$ ALU is inside this

- In General, Effective address =

  $\underbrace{\texttt{displacement}}_{\text{signed 8 bit number}}$ `+ [base register]+ [index register]`

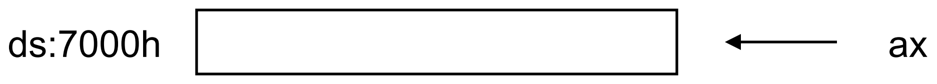
  `(if any)` `(if any)`

- Any Combination of These 3 Values
  - Leads to Several Different Addressing Modes

- Displacement
  - 8 or 16 bit Constant in the Instruction
  - "base register" Must be **`BX`** or **`BP`**
  - "index register" Must be **`SI`** or **`DI`**

Direct Addressing

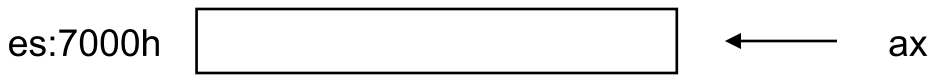
```
mov  [7000h],  ax
```



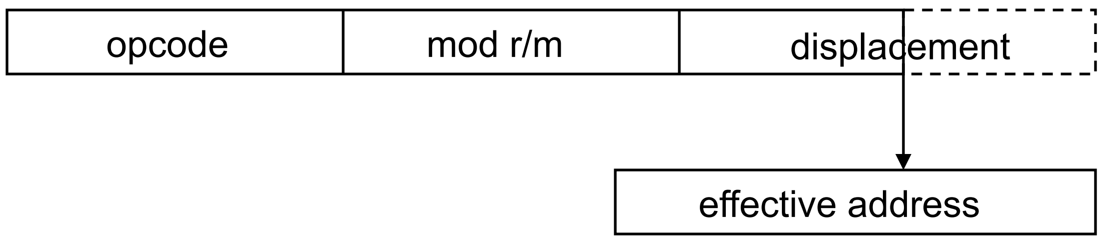
```
A3 00 70
```

```
mov  es:[7000h], ax
```

26 **A3 00 70**

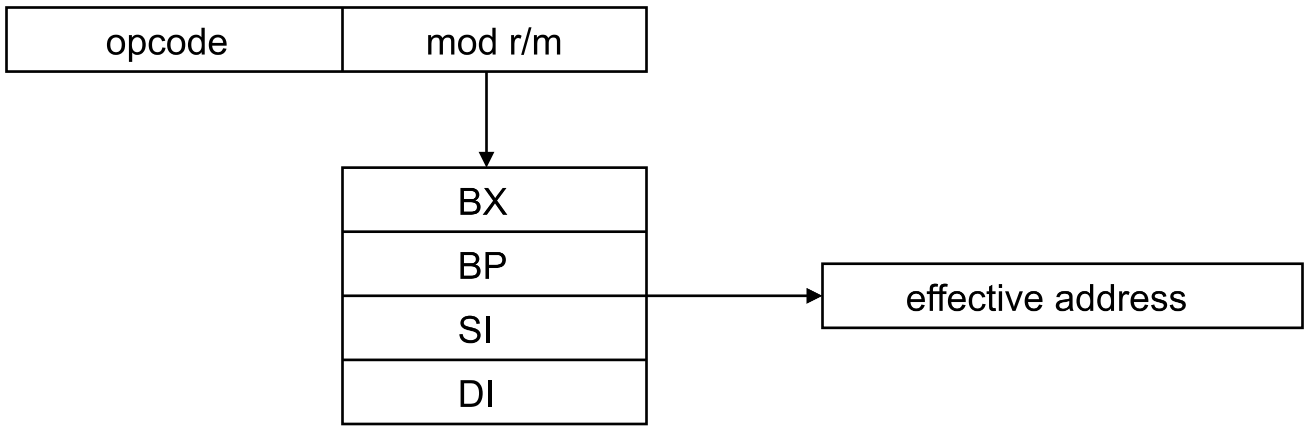


prefix byte
- longer instruction
- more fetch time



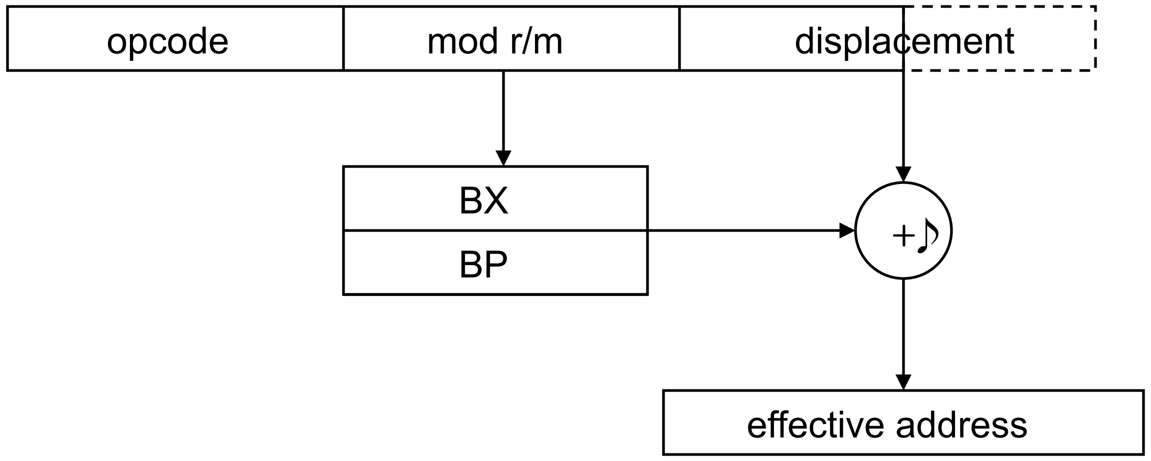
Register Indirect Addressing

```
mov  al,  [bp]       ;al gets 8 bits at SS:BP

mov  ah,  [bx]       ;ah gets 8 bits at DS:BX

mov  ax,  [di]       ;ax gets 16 bits at DS:SI

mov  eax, [si]       ;eax gets 32 bits at DS:SI
```



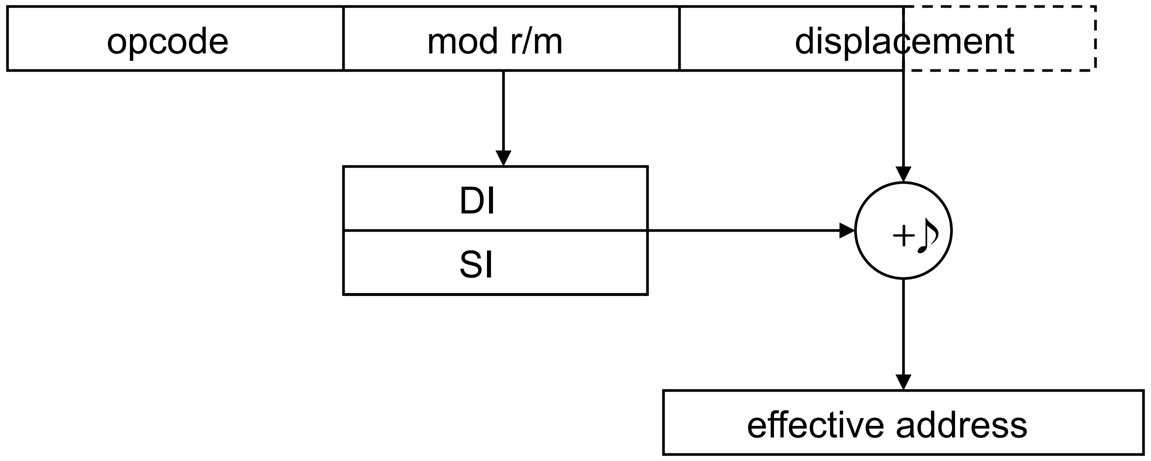
Based Indirect Addressing

```
mov    al,    [bp+2]        ;al gets 8 bits at SS:BP+2

mov    ah,    [bx-4]        ;ah gets 8 bits at DS:BX-4
```



Indexed Indirect Addressing

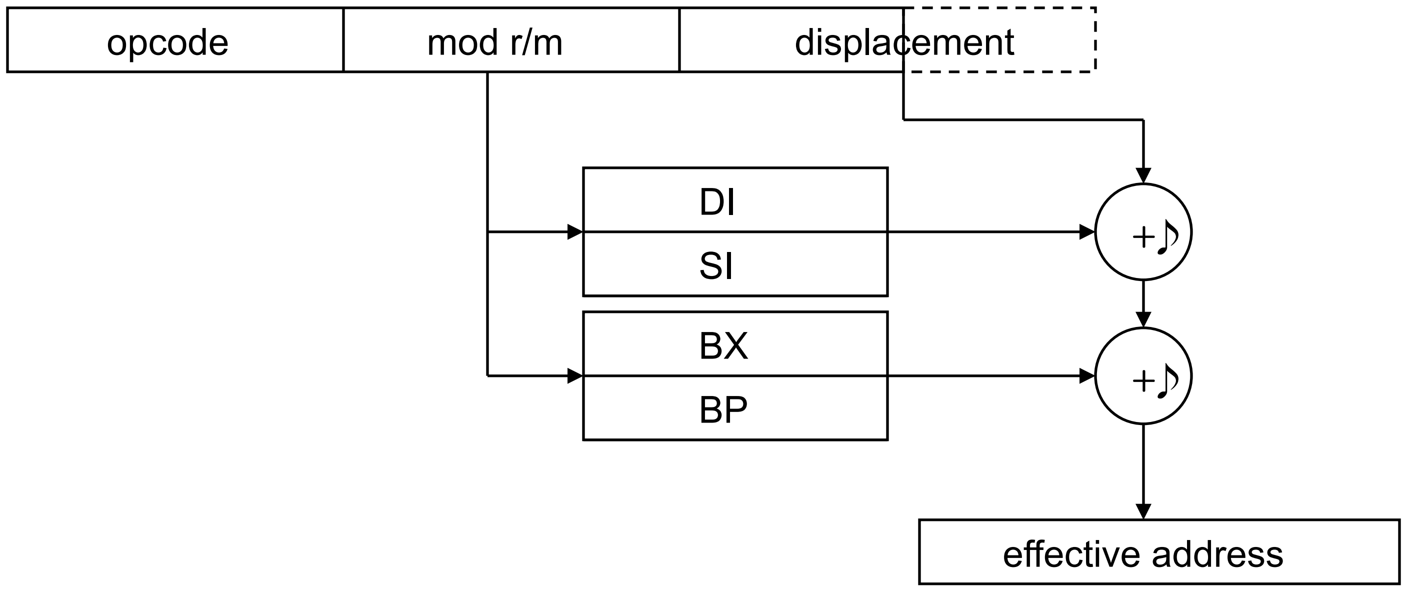
```
mov    ax,    [si+1000h]    ;ax gets 16 bits at
DS:SI+1000h

mov    eax,   [si+300h]     ;eax gets 32 bits at
DS:SI+300h

Mov    [di+100h],   al      ;DS:DI+100h gets 8 bits in al
```



Based Indexed Indirect Addressing

♪

```
mov    ax,    [bp+di]       ;ax gets 16 bits at SS:BP+DI

mov    ax,    [di+bp]       ;ax gets 16 bits at DS:BP+DI

mov    eax,   [bx+si+10h]   ;eax gets 32 bits at DS:BX+SI+10h

mov    cx,    [bp+si-7]     ;cx gets 16 bits at SS:BP+SI-7
```



Addressing Mode Examples

```
mov al,   bl             ;8-bit register addressing
mov di,   bp             ;16-bit register addressing
mov eax,  eax            ;32-bit register addressing
```

Register

```
mov al,   12             ;8-bit immediate,  al<-0ch
mov cx,   faceh          ;16-bit immediate, cx<-64,206
mov ebx,  2h             ;32-bit immediate, ebx<-00000002h
```

Immediate

```
mov al,   LIST           ;al<-8 bits stored at label LIST
mov ch,   DATA           ;ch<-8 bits stored at label DATA
mov ds,   DATA2          ;ds<-16 bits stored at label DATA2
```

Direct

```
mov al,   [bp]           ;al<-8 bits stored at SS:BP
mov ah,   [bx]           ;ah<-8 bits stored at DS:BX
mov ax,   [bp]           ;ax<-16 bits stored at SS:BP
mov eax,  [bx]           ;eax<-32 bits stored at DS:BX
mov al,   [bp+2]         ;al<-8 bits stored at SS:(BP+2)
mov ax,   [bx-4]         ;ax<-16 bits stored at DS:(BX-4)
mov al,   LIST[bp]       ;al<-8 bits stored at SS:(BP+LIST)
mov bx,   LIST[bx]       ;bx<-16 bits stored at DS:(BX+LIST)
mov al,   LIST[bp+2]     ;al<-8 bits stored at SS:(BP+2+LIST)
mov ax,   LIST[bx-12h] ;ax<-16 bits stored at DS:(BX-

18+LIST)
```

Based

More Addressing Mode Examples

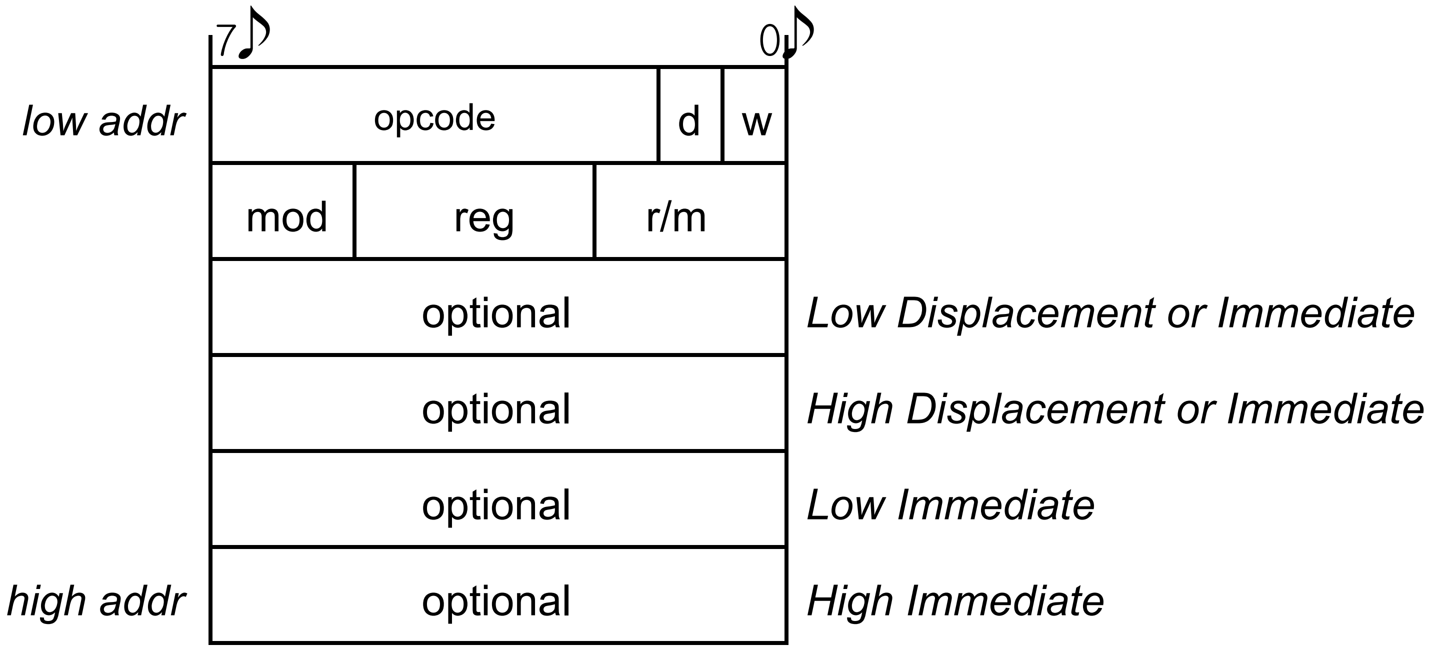
```
mov al,   [si]          ;al<-8 bits stored at DS:SI
mov ah,   [di]          ;ah<-8 bits stored at DS:DI
mov ax,   [si]          ;ax<-16 bits stored at DS:SI
mov eax,  [di]          ;eax<-32 bits stored at DS:DI
mov ax,   es:[di]       ;ax<-16 bits stored at ES:DI
mov al,   [si+2]        ;al<-8 bits stored at DS:(SI+2)
mov ax,   [di-4]        ;ax<-16 bits stored at DS:(DI-4)
mov al,   LIST[si]      ;al<-8 bits stored at DS:(SI+LIST)
mov bx,   LIST[di]      ;bx<-16 bits stored at DS:(DI+LIST)
mov al,   LIST[si+2]    ;al<-8 bits stored at DS:(SI+2+LIST)
mov ax,   LIST[di-12h]  ;ax<-16 bits stored at DS:(DI-18+LIST)
```

Indexed

```
mov al,   [bp+di]       ;al<-8 bits from SS:(BP+DI)
mov ah,   ds:[bp+si]    ;ah<-8 bits from DS:(BP+SI)
mov ax,   [bx+si]       ;ax<-16 bits from DS:(BX+SI)
mov eax,  es:[bx+di]    ;eax<-32 bits from ES:(BX+DI)
mov al,   LIST[bp+di]   ;al<-8 bits from SS:(BP+DI+LIST)
mov ax,   LIST[bx+si]   ;ax<-16 bits from DS:(BX+SI+LIST)
mov al,   LIST[bp+di-10h]      ;al<-8 bits from SS:(BP+DI-16+LIST)
mov ax,   LIST[bx+si+1AFH]     ;ax<-16 bits from DS:(BX+SI+431+LIST)
```

Based Indexed

Instruction Format (opcode, d, w)

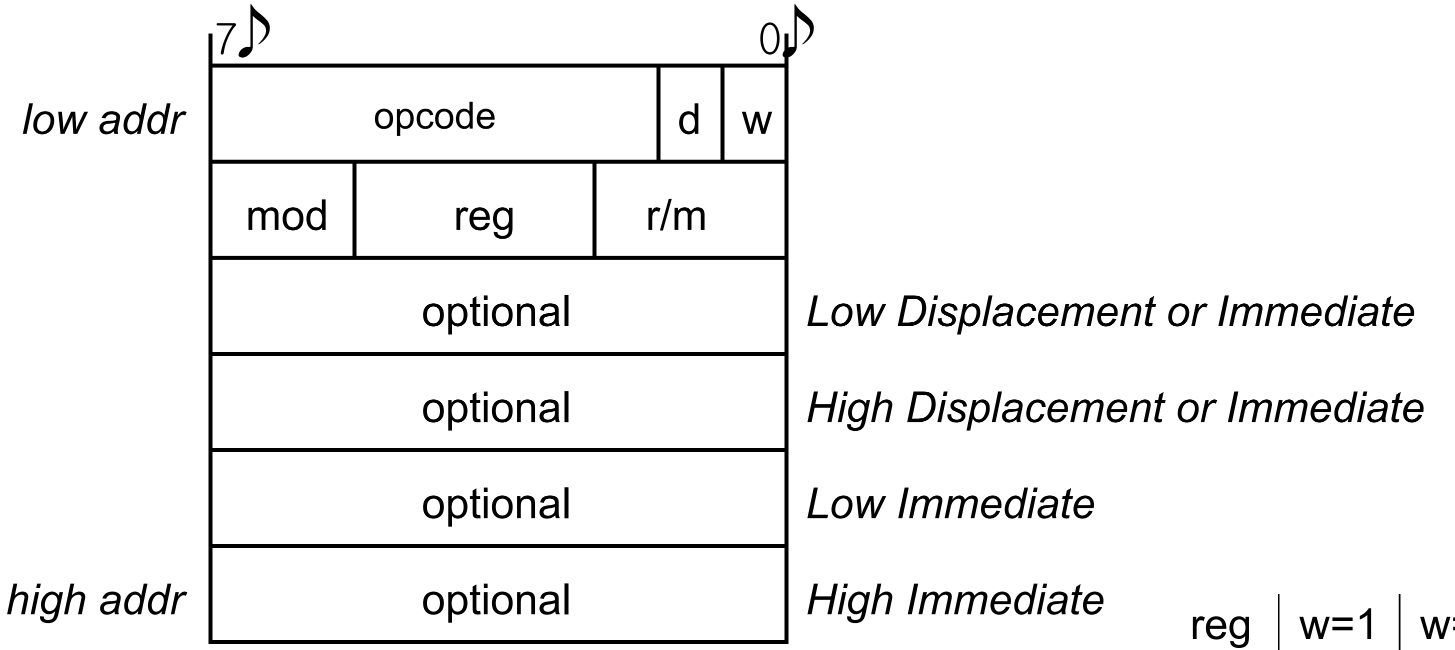


opcode 6-bit value that specifies instruction type

d is 1-bit value that specifies destination
  d=0 for memory and *d*=1 for register

w is 1-bit value that specifies if destination is word or byte
  w=0 for byte and *w*=1 for word

Instruction Format (reg, mod)



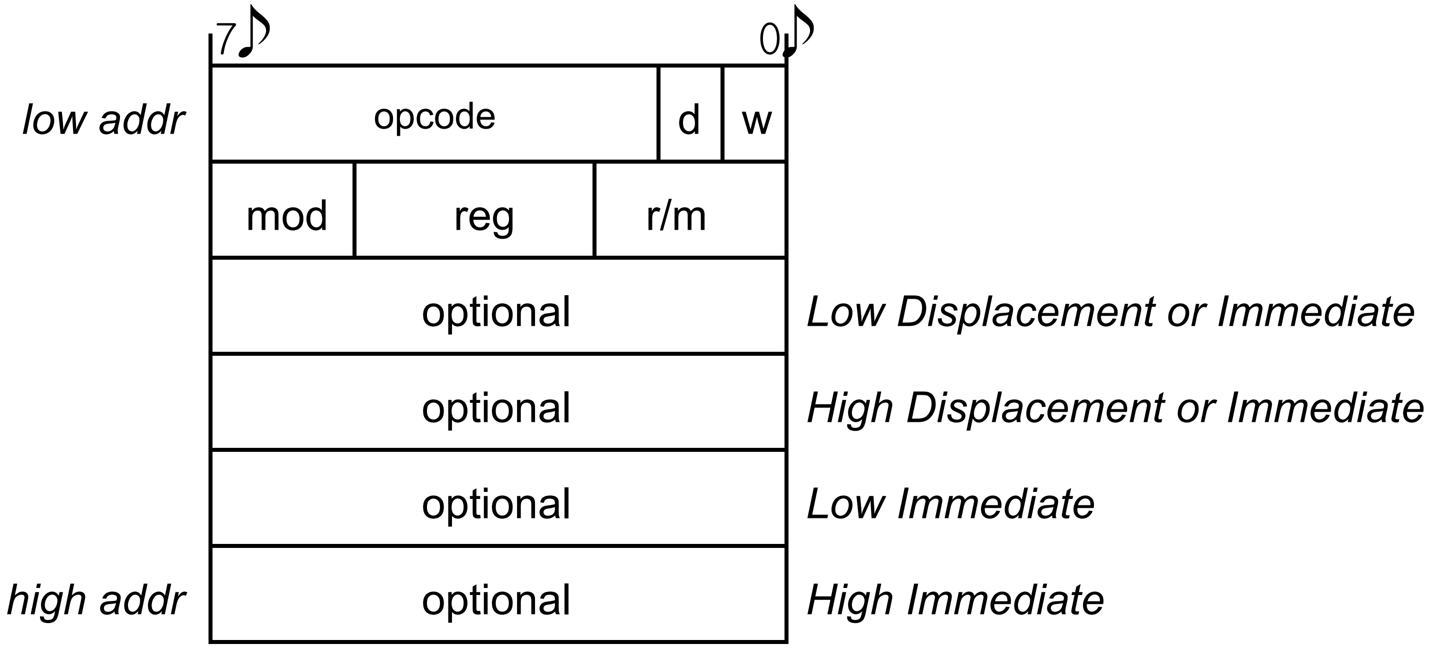
reg	w=1	w=0
000	ax	al
001	cx	cl
010	dx	dl
011	bx	bl
100	sp	ah
101	bp	ch
110	si	dh
111	di	bh

mod is 2-bit value that indicates *addressing mode* (along with *r/m* value)

reg is 3-bit value that specifies a (destination) register (see table to right) or opcode extension♪

r/m is 3-bit value that specifies operand location (r/m means register/memory)

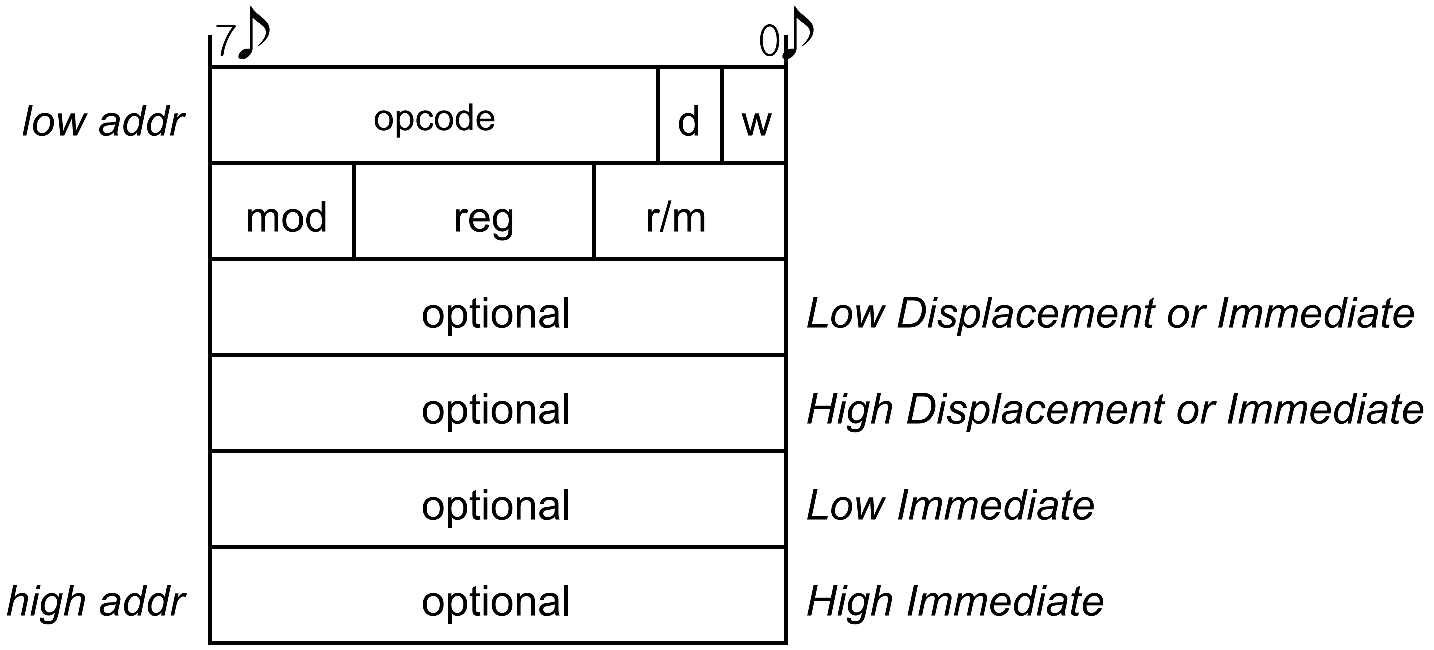
Instruction Format (displacement)



Displacement may be either 8 or 16 bits
- signed integer encoded into instruction
- used in computation of operand address

Immediate may be 8, 16 or 32 bits
- signed integer
- used as an actual operand

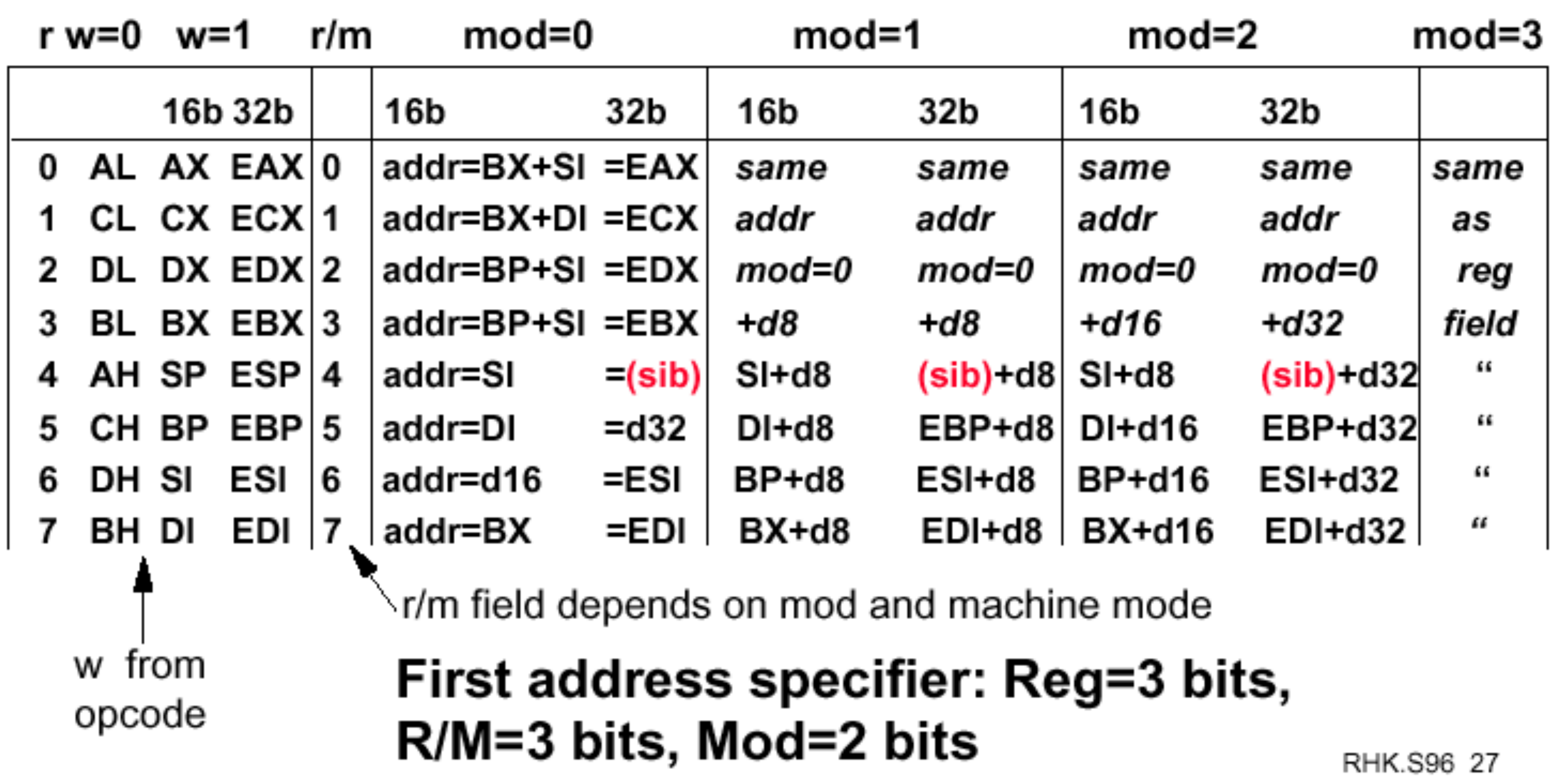
Instruction Format Example



Consider the Instruction: **`mov ax, bx`**
The Assembler translates this into: **`8B C3`**

opcode is:	100010	**`mov`**
d is:	1	destination is register
w is:	1	destination size = 1 word
mod is:	11	this indicates that r/m specifies a register
reg is:	000	destination register is **`ax`**
r/m is:	011	source register is **`bx`**

Instruction Format (mod)



r	w=0	w=1		r/m	mod=0		mod=1		mod=2		mod=3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr*	*addr*	*addr*	*addr*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+d8*	*+d8*	*+d16*	*+d32*	*field*
4	AH	SP	ESP	4	addr=SI	=(sib)	SI+d8	(sib)+d8	SI+d8	(sib)+d32	“
5	CH	BP	EBP	5	addr=DI	=d32	DI+d8	EBP+d8	DI+d16	EBP+d32	“
6	DH	SI	ESI	6	addr=d16	=ESI	BP+d8	ESI+d8	BP+d16	ESI+d32	“
7	BH	DI	EDI	7	addr=BX	=EDI	BX+d8	EDI+d8	BX+d16	EDI+d32	“

w from opcode

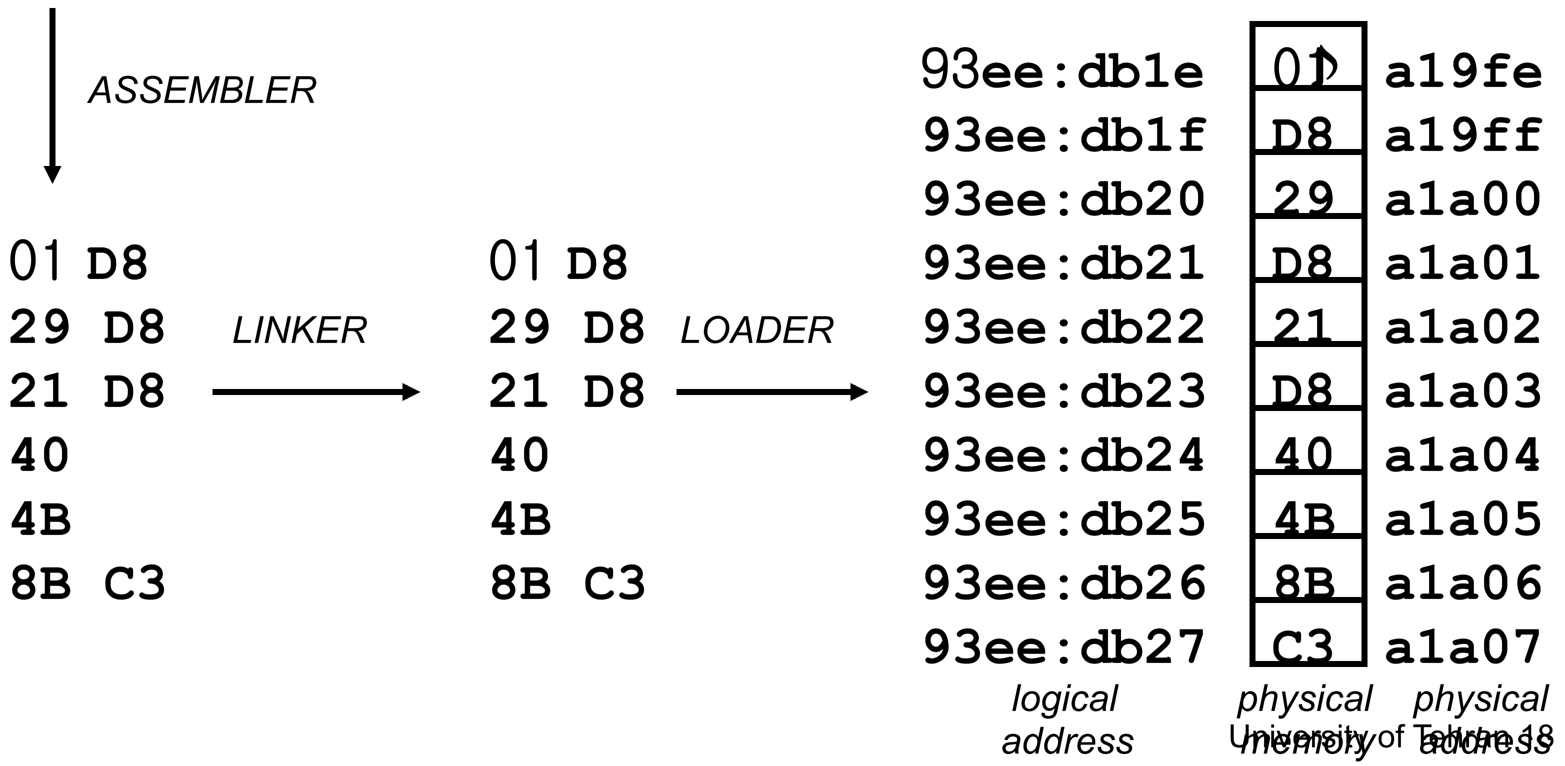
r/m field depends on mod and machine mode

First address specifier: Reg=3 bits, R/M=3 bits, Mod=2 bits

RHK.S96 27

Assembler versus Machine Code

```
ADD   AX, BX      ;AX gets value AX+BX
SUB   AX, BX      ;AX gets value AX-BX
AND   AX, BX      ;AX gets bitwise AND of AX and BX
INC   AX          ;AX gets its original value plus 1
DEC   BX          ;BX gets its original value minus 1
MOV   AX, BX      ;AX gets values in BX
```



```
01 D8           01 D8
29 D8           29 D8
21 D8           21 D8
40              40
4B              4B
8B C3           8B C3
```

logical address	physical memory	physical address
93ee:db1e	01	a19fe
93ee:db1f	D8	a19ff
93ee:db20	29	a1a00
93ee:db21	D8	a1a01
93ee:db22	21	a1a02
93ee:db23	D8	a1a03
93ee:db24	40	a1a04
93ee:db25	4B	a1a05
93ee:db26	8B	a1a06
93ee:db27	C3	a1a07

I/O Port Addressing

- **x86 family has 65,536 I/O ports** 16 bit address
- **Each port has address (like memory)**
  - **Referred to as "I/O memory space"**
- **I/O port is 1 byte or 2 bytes**
  - **with 386+ also 4 bytes**
- **Two addressing modes**

1) Direct port address can only support 0 to 255

- Can only be 1 byte

- Can only be address ports 00h through ffh

2) Port address present in DX

- Can address all ports 0000h through ffffh

- **Can only use DX for port addresses**
- **Can only use AL,AX,EAX for port data**

puts the content of AL in the port number (address) 23H

out 23H, AL

out

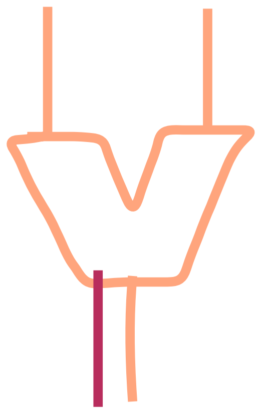
in

16 bit: shows the current state of something

Flag Register

for instance if we do the following:
53 + c7 = 1 1c
then the carry flag (CF) will become one

15															0
x	x	x	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF



CF	*Carry Flag*	Arithmetic Carry/Borrow
OF	*Overflow Flag*	Arithmetic Overflow
ZF	*Zero Flag*	Zero Result; Equal Compare
SF	*Sign Flag*	Negative Result; Non-Equal
PF	*Parity Flag*	Even Number of "1" bits
AF	*Auxiliary Carry*	Used with BCD Arithmetic
DF	*Direction Flag*	Auto-Increment/Decrement (used for "string operations")
IF	*Interrupt Flag*	Enables Interrupts (allows "fetch-execute" to be interrupted)
TF	*Trap Flag*	Allows Single-Step (for debugging; causes interrupt after each op)

80x86 Family

- 16-bit Processors
  - 8088 (8-bit data / 20-bit address)
  - 8086/186 (16-bit data / 20-bit address)
  - 80286 (16-bit data / 24-bit address)
- 32-bit Processors
  - 80386 (16/24 or 32/32 common)
  - 80486 (32/32), Pentium, PII (64/32)
  - Pentium Pro, II, III, IV (64/36)
  - PPC 60x (32 or 64/32)
- All 80x86 processors use a 16-bit address for i/o

8 And 16 bit Organizations

then the connections can only exchange only as much bits as the data bus in every transaction

- 8088 — 8 lines of data bus
  - Data is organized into byte widths
  - The 1MB memory is organized as 1M x 8-bits
- 8086/80186 — 16 bit data bus
  - Data is organized into word widths
  - The 1MB memory is organized as 512k x 16-bits
- 80286/80386SX — ۲۴ خط ادرس داریم پس می‌تونه ۱۶ مگابایت حافظه داشته باشه
  - Data is organized into word widths
  - The 16MB memory is organized as 8M x 16-bits

$2^{24} = 16$ MB

32 and 64 bit Organizations

پ پ 80386DX/80486

ت ت Data is organized into double word widths

ت ت The 4GB memory is organized as 1G x 32-bits

پ پ Pentium Pro/Pentium 1-4

ت ت Data is organized into quad word widths

ت ت The 4GB memory is organized as 512M x64-bits

پ (on P2-4, actual address bus is 36 bits)

Little Endian / Big Endian

Byte addressable

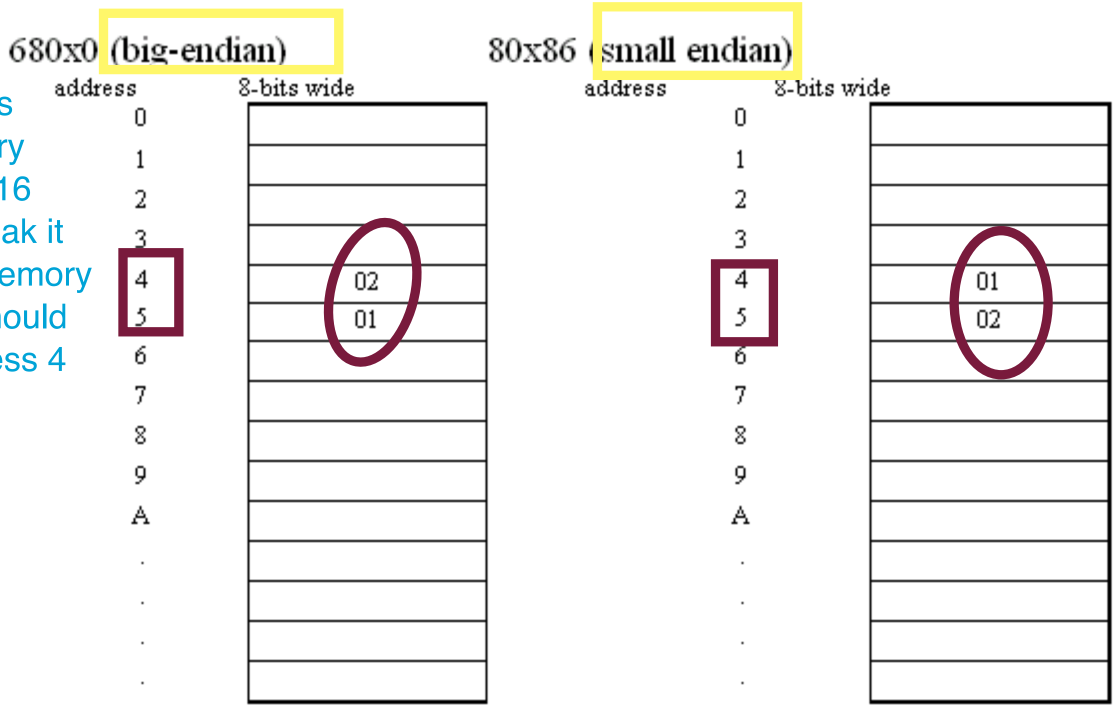
for the 68000:

```
MOVE.W     #513, D0       ; move value 513 into the lower 16 bits of D0
MOVE.W     D0,4           ; store the lower word of D0 into memory 4
```

for the 80x86:

```
MOV   AX,513              ; load AX (16 bits), with the value 513
MOV   [4],AX              ; store AX into memory 4
```

byte addressability has to be true in the memory so if we want to save a 16 bit number we have to break it to 2 parts each 8 bits in 2 memory spaces, now which part should we place in memory address 4 and which in 5?



small endian: if we place the least significant part in 4 (low byte in 4)

big endian: if we place the most significant part in 4 (high byte in 4)

Summary

- Operands
- Addressing modes
- IO ports
- Flag
- Memory alignment