

## 80x86 Instruction Encoding

Machine Language

## 8086 Instructions

- Like other attributes of x86 processors, the machines through x86-64 are backwardly compatible with the 8086
- We will look at 8086 encoding in detail
- Extension to Pentium instruction is straightforward

## Encoding of 8086 Instructions

- 8086 instructions are encoded as binary numbers
- Instructions vary in length from 1 to 6 bytes  
Note that many RISC architectures have fixed length instructions
- Below is the general 2-operand instruction format

| byte | 7          | 6 | 5   | 4 | 3 | 2   | 1 | 0 |  |
|------|------------|---|-----|---|---|-----|---|---|--|
| 1    | opcode     |   |     |   |   | d   | w |   | Opcode byte  |
| 2    | mod        |   | reg |   |   | r/m |   |   | Addressing mode byte<br>low disp, addr, or data<br>high disp, addr, or data<br>low data<br>high data |
| 3    | [optional] |   |     |   |   |     |   |   |  |
| 4    | [optional] |   |     |   |   |     |   |   |  |
| 5    | [optional] |   |     |   |   |     |   |   |  |
| 6    | [optional] |   |     |   |   |     |   |   |  |

## Instruction Formats

- There are many variations in Intel instruction format
- Some instructions are optimized to be small
  - Increment and decrement
  - Addition, subtraction, logical operations on accumulator
  - Add immediate to register
  - Push/pop register

## Opcode and Addressing Mode

- The first two bytes are called the opcode byte and the addressing mode byte
- The opcode byte specifies the operation, the size of operands and the direction of data movement between operands
- The addressing mode byte specifies two operands for the instruction  
For some instructions, such as any immediate mode instruction the addressing mode byte also serves as an "opcode extension"
- Some instructions are one-byte instructions and lack the addressing mode byte

## Prefix Bytes

- An instruction may also be optionally preceded by one or more prefix bytes for repeat, segment override, or lock prefixes
- In 32-bit machines we also have an address size override prefix and an operand size override prefix

## Byte Order

- Note the order of bytes in an assembled instruction
- 16-bit values are stored in little-endian order

| [prefix] | opcode | [addr mode] | [Low Addr] | [High Addr] | [Low Data] | [High Data] |
|----------|--------|-------------|------------|-------------|------------|-------------|
|----------|--------|-------------|------------|-------------|------------|-------------|

## Prefix Bytes

- There are four types of prefix instructions:
  - Repetition
  - Segment Overrides
  - Lock
  - Address/Operand size overrides (for 32-bit machines)
- Encoded as follows (Each in a single byte)
- Repetition
 

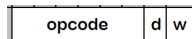
|                 |     |
|-----------------|-----|
| REP, REPE, REPZ | F3H |
| REPNE, REPNZ    | F2H |
- Note that REP and REPE are not distinct  
Machine (microcode) interpretation of REP and REPE code depends on instruction currently being executed
- Segment override
 

|    |     |
|----|-----|
| CS | 2EH |
| DS | 3EH |
| ES | 26H |
| SS | 36H |
- Lock
 

|  |     |
|--|-----|
|  | F0H |
|--|-----|

## The Opcode Byte

- The opcode field specifies the operation performed (mov, xchg, etc). Usually (but not always) 6 bits



- The d (direction) field specifies the direction of data movement:
  - d = 1 destination is operand specified by REG field
  - d = 0 destination is operand specified by R/M field
- The d position MAY be interpreted as the "s" bit
  - s = 1 one byte of immediate data is present which must be sign-extended to produce a 16-bit operand
  - s = 0 two bytes of immediate are present

## The Opcode Byte

- The d position is interpreted as the "c" bit in Shift and Rotate instructions
  - C = 1 CL is used for shift count
  - C = 0 Shift/Rotate by 1 or by immediate value
- The w (word/byte) bit specifies operand size
  - W = 1 data is word (16 bits)
  - W = 0 data is byte
- In 32-bit instructions
  - W = 1 data is dword (32 bits)
  - W = 0 data is byte
- What if we have a 16-bit operand in 32-bit code?

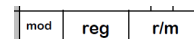
## Operand and Address Size Overrides

- We only have one bit (the w bit) for operand size so only two operand sizes can be directly specified
- Operand and Address size override prefixes are used to specify 32-bit registers in 16-bit code and 16-bit registers in 32-bit code
  - 66h = operand size override
  - 67h = address size override
- Interpretation of an instruction depends on whether it is executed in a 16-bit code segment or a 32-bit code segment
 

| Instruction    | 16-bit code | 32-bit code |
|----------------|-------------|-------------|
| mov ax, [bx]   | 8B 07       | 67 66 8B 07 |
| mov eax, [bx]  | 66 8B 07    | 67 8B 07    |
| mov ax, [ebx]  | 67 8B 03    | 66 8B 03    |
| mov eax, [ebx] | 67 66 8B 03 | 8B 03       |

## Addressing Mode Byte

- Contains three fields that specify operands



Mod Bits 6-7 (mode; determines how R/M field is interpreted)  
 Reg Bits 3-5 (register) or SREG (Seg register)  
 R/M Bits 0-2 (register/memory)

- MOD
 

|    |  |
|----|--|
| 00 | Use R/M Table 1 for R/M operand          |
| 01 | Use R/M Table 2 with 8-bit displacement  |
| 10 | Use R/M Table 2 with 16-bit displacement |
| 11 | Use REG table for R/M operand            |

### REG table

| REG | w=0 | w=1 | REG | w=0 | w=1 |
|-----|-----|-----|-----|-----|-----|
| 000 | AL  | AX  | 100 | AH  | SP  |
| 001 | CL  | CX  | 101 | CH  | BP  |
| 010 | DL  | DX  | 110 | DH  | SI  |
| 011 | BL  | BX  | 111 | BH  | DI  |

- For 32 bit code

| REG | w=0 | w=1 | REG | w=0 | w=1 |
|-----|-----|-----|-----|-----|-----|
| 000 | AL  | eax | 100 | AH  | esp |
| 001 | CL  | ecx | 101 | CH  | ebp |
| 010 | DL  | edx | 110 | DH  | esi |
| 011 | BL  | ebx | 111 | BH  | edi |

#### SREG

|     |    |     |    |     |    |     |    |
|-----|----|-----|----|-----|----|-----|----|
| 000 | ES | 001 | CS | 010 | SS | 110 | DS |
|-----|----|-----|----|-----|----|-----|----|

### R/M Tables

#### R/M Table 1 (Mod = 00)

|     |         |     |         |     |      |     |           |
|-----|---------|-----|---------|-----|------|-----|-----------|
| 000 | [BX+SI] | 010 | [BP+SI] | 100 | [SI] | 110 | Drc't Add |
| 001 | [BX+DI] | 011 | [BP+DI] | 101 | [DI] | 111 | [BX]      |

#### R/M Table 2 (Mod = 01 or 10)

##### Add DISP to register specified:

|     |         |     |         |     |      |     |      |
|-----|---------|-----|---------|-----|------|-----|------|
| 000 | [BX+SI] | 010 | [BP+SI] | 100 | [SI] | 110 | [BP] |
| 001 | [BX+DI] | 011 | [BP+DI] | 101 | [DI] | 111 | [BX] |

### Addressing Mode Byte

- Not present if instruction has zero explicit operands  
For one-operand instructions the R/M field indicates where the operand is to be found  
For two-operand instructions (except those with an immediate operand) one is a register determined by REG (SREG) field and the other may be register or memory and is determined by R/M field.
- The Direction bit has meaning only in two-operand instructions  
Indicates whether "destination" is specified by REG or by R/M
- Note that this allows many instructions to be encoded in two different ways  
Swap R/M and REG operands and flip d bit

### Addressing Mode 00

- Specifies R/M Table 1 (with NO displacement)
- |     |         |     |         |     |      |     |           |
|-----|---------|-----|---------|-----|------|-----|-----------|
| 000 | [BX+SI] | 010 | [BP+SI] | 100 | [SI] | 110 | Drc't Add |
| 001 | [BX+DI] | 011 | [BP+DI] | 101 | [DI] | 111 | [BX]      |
- Note that the 110 case (direct addressing) requires that the instruction be followed by two address bytes
  - There are then two possibilities:
- | Opcode | AddrMode | Offset-Low | Offset-High |
|--------|----------|------------|-------------|
|--------|----------|------------|-------------|
- Examples:
- ```
MOV AX,[2A45]
MOV AX,[DI]
```

### Addressing Mode 01

- Specifies R/M Table 2 with 8-bit signed displacement
- R/M Table 2 (Mod = 01 or 10)
- Add DISP to register specified:
- |     |         |     |         |     |      |     |      |
|-----|---------|-----|---------|-----|------|-----|------|
| 000 | [BX+SI] | 010 | [BP+SI] | 100 | [SI] | 110 | [BP] |
| 001 | [BX+DI] | 011 | [BP+DI] | 101 | [DI] | 111 | [BX] |
- All instructions have the form:
- | Opcode | AddrMode | Disp8 |
|--------|----------|-------|
|--------|----------|-------|
- Examples
- ```
ADD AX,[BX+1000h]
MOV DX,[BX+DI+130]
```

### Addressing Mode 11

- Specifies that R/M bits refer to REG table
  - All two operand register-to-register instructions use addressing mode 11
- ```
MOV AX,BX
MOV DX,CX
MOV AH,BL
```
- Addressing Mode 11 is also used by immediate mode instructions where dest is a register
- ```
ADD BX,1
ADC CH,0
OR SI,0F0Fh
```

### Addressing Mode 10

- Specifies R/M Table 2 with 16-bit unsigned displacement

R/M Table 2 (Mod = 01 or 10)

Add DISP to register specified:

```
000 [BX+SI]    010 [BP+SI]  100 [SI]  110  [BP]
001 [BX+DI]    011 [BP+DI]  101 [DI]  111  [BX]
```

- All instructions have the form:

Opcode AddrMode Disp-Low Disp-High

- Examples

```
MOV AX,[BP+2]
MOV DX,[BX+DI+4]
MOV [BX-4],AX
```

### MOV reg/mem to/from reg/mem

- This instruction has the structure:

100010dw modreg/m Disp-lo Disp-hi

- Where 0, 1 or 2 displacement bytes are present depending on the MOD bits

- **MOV AX,BX**

w = 1 because we are dealing with words

MOD = 11 because it is register-register

- if d = 0 then REG = source (BX) and R/M = dest (AX)  
= 1000 1001 1101 1000 (89 D8)

- if d = 1 then REG = source (AX) and R/M = dest (BX)  
= 1000 1011 1010 0011 (8B C3)

### MOV reg/mem to/from reg/mem

- **MOV [BX+10h],CL**

w = 0 because we are dealing with a byte

d = 0 because we need R/M Table 2 to encode [BX+10h]

- therefore first byte is 10001000 = 88H

- since 10H can be encoded as an 8-bit displacement, we can use

MOD=01 REG=001 and R/M=111 = 0100 1111 = 4FH

- and the last byte is 10H

result: 88 4F 10

Note: MOV [BX+10H],CX = 89 4F 10

- since 10H can also be encoded as a 16-bit displacement, we can use

MOD=10 REG=001 and R/M=111 = 1000 1111 = 8FH

- and the last bytes are 00 10

result: 88 8F 00 10

### MOV reg/mem, imm

- This instruction has the structure:

1100 011w MOD 000 R/M displ disp2

- Where 0, 1 or 2 displacement bytes are present depending on value of MOD

- **MOV BYTE PTR [100h],10h**

w = 0 because we have byte operand

MOD = 00 (R/M Table 1) R/M = 110 (Direct Addr)

bytes 3 and 4 are address; byte 5 immediate data

- Result

C6 06 00 01 10

- **MOV WORD PTR [BX+SI],10h**

w = 1 because we have word operand

MOD = 00 (R/M Table 1) R/M = 000 ([BX+SI])

bytes 3 and 4 are immediate data

- Result

C7 00 10 00

### MOV imm to reg

- This instruction is optimized as a 4-bit opcode, with register encoded into the instruction

1011wreg

- Examples

```
MOV bx, 3      1011 w=1 reg=011=BX
10111011 imm  BB 03 00
MOV bh, 3      1011 w=0 reg=111=BH
10110111 imm  B7 03
MOV bl, 3      1011 w=0 reg=011=BL
10110011 imm  B3 03
```

### MOV direct mem to/from accumulator

- Another optimized instruction

101000dw addr

- Example **mov al, [34F4]**

d = 0 because dest is REG

w = 0 because AL is 8 bits

10100000 addr = A0 F4 C4

- Example **mov [34F4], ax**

d = 1 because dest is MEM

w = 1 because AX is 16 bits

10100011 addr = A3 F4 C4

### POP Reg/Mem

- POP memory/register has the structure:  
`8F MOD00R/M`
- Note that  $w = 1$  always for POP (cannot pop bytes)  
 Note: The middle 3 bits of the R/M byte are specified as 000 but actually can be any value
- To POP into AX:  
`MOD = 11 (Use REG table) R/M = 000 -> 11 000 000`  
`Encoding: 8F C0`
- To POP into BP:  
`MOD = 11 (Use REG table) R/M = 101 -> 11 000 101`  
`Encoding: 8F C5`

### POP Reg/Mem

- To POP into memory location DS:1200H  
`MOD = 00 R/M = 110 00 000 110`  
`Encoding 8F 06 00 12`
- To POP into memory location CS:1200H add a prefix byte  
`CS = 2Eh`  
`Encoding = 2E 8F 06 00 12`

### POP General Register

- This one-byte opcode has the structure:  
`01011 REG`
- So  
`POP AX = 01011000 = 58`  
`POP BX = 01001011 = 5B`
- Note that there are two legal encodings of POP REG
- Shorter form exists because POPs are so common
- All assemblers and compilers will use the shorter form

### POP Segment Register

- This one-byte opcode has the structure:  
`00seg111`  
`POP ES = 0000 0111 = 07H`  
`POP DS = 0001 1111 = 1FH`  
`POP SS = 0001 0111 = 17H`
- Note that both forms of POP REG do not follow the general rules outlined above--registers are coded into the opcode byte
- Note also that even though POP CS is illegal, DEBUG will correctly assemble it as 0F -- but will not unassemble it.

### Immediate Mode Instructions

- Immediate mode instructions have only one register or memory operand; the other is encoded in the instruction itself  
 The Reg field is used an "opcode extension"  
 The addressing mode byte has to be examined to determine which operation is specified
- |                                 |                        |                        |
|---------------------------------|------------------------|------------------------|
| <code>add imm to reg/mem</code> | <code>1000 00sw</code> | <code>mod000r/m</code> |
| <code>or imm to reg/mem</code>  | <code>1000 00sw</code> | <code>mod001r/m</code> |
- In instructions with immediate operands the "d" bit is interpreted as the "s" bit
  - When the s bit is set it means that the single byte operand should be sign-extended to 16 bits

### ADD imm to reg/mem

- `add dx, 3 ;Add imm to reg16`  
`1000 00sw mod000r/m`  
 $w=1$  (DX is 16 bits)  
`mod = 11 (use REG table) r/m = 010 =DX`
- With s bit set we have  
`1000 0011 11 000 010 operand = 83 C2 03`
- With s bit clear we have  
`1000 0001 11 000 010 operand = 81 C2 03 00`

## Examples of Equivalent Encodings

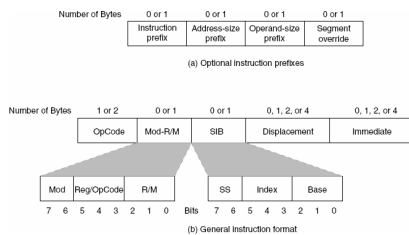
- The short instructions were assembled with debug's A command
  - The longer instructions were entered with the E command
- |           |          |     |              |
|-----------|----------|-----|--------------|
| 1822:0100 | 58       | POP | AX           |
| 1822:0101 | 8FC0     | POP | AX           |
| 1822:0103 | 894F10   | MOV | [BX+10],CX   |
| 1822:0106 | 898F1000 | MOV | [BX+0010],CX |
| 1822:010A | A10001   | MOV | AX,[0100]    |
| 1822:010D | 8B060001 | MOV | AX,[0100]    |
- The above examples show inefficient machine language equivalences.
  - There are also plenty of "efficient" equivalences where the instructions are the same length
  - It is possible to create signature in machine language for a particular assembler or compiler by picking specific encodings

## Extending 16-bit encoding to 32 bits

- There are only a few changes needed
- Add some opcodes for new instruction
  - Treat w bit as 0=8 bits, 1=32 bits, so in REG field interpret w=1 and REG=000 as eax, not ax
  - Add operand size prefix byte to handle 16-bit operands
  - Likewise 8/16 bit displacement, imm data, or address values are treated as 8/32 bit
  - The MAJOR change is the interpretation of addressing mode byte
- The R/M byte can specify the presence of a SIB byte  
SIB has fields scale, index, base  
(see instruction format reference)

## 32-bit General Format

- This is the general form for common 2 operand instructions



## 32-Bit SIB Example

- ADD ebx,[eax + 4 \* ecx]
- Opcode 000000dw = 00000011 because dest is REG operand and w=1 indicates 32 bits
- Addressing mode byte mod reg r/m = 00 001 100  
Mod 00 with R/M 100 means SIB mode, no displacement
- SIB byte scale index base = 10 001 000  
10 = 4 001=ecx 000 = eax
- Complete Instruction =  
00000011 00001100 10001000  
03 0C 88