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

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

<table>
<thead>
<tr>
<th>prefix</th>
<th>opcode</th>
<th>addr mode</th>
<th>[Low Addr]</th>
<th>[High Addr]</th>
<th>[Low Data]</th>
<th>[High Data]</th>
</tr>
</thead>
</table>

Prefix Bytes

- There are four types of prefix instructions:
  - Repetition
  - Segment Overides
  - Lock
  - Address/Operand size overrides (for 32-bit machines)
- Encoded as follows (Each in a single byte)
- Repetition
  - REP, REPPE, REPZ
  - F3H
  - REPNE, REPNZ
  - F2H
- Note that REP and REPE and 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

Opcode Byte

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

<table>
<thead>
<tr>
<th>opcode</th>
<th>d</th>
<th>w</th>
</tr>
</thead>
</table>
- 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

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

<table>
<thead>
<tr>
<th>instruction</th>
<th>16-bit code</th>
<th>32-bit code</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ax, [bx]</td>
<td>8B 07</td>
<td>67 66 8B 07</td>
</tr>
<tr>
<td>mov eax, [bx]</td>
<td>66 8B 07</td>
<td>67 8B 07</td>
</tr>
<tr>
<td>mov ax, [bx]</td>
<td>67 8B 03</td>
<td>66 8B 03</td>
</tr>
<tr>
<td>mov eax, [bx]</td>
<td>67 66 8B 03</td>
<td>8B 03</td>
</tr>
</tbody>
</table>

Addressing Mode Byte

- Contains three fields that specify operands

<table>
<thead>
<tr>
<th>mod</th>
<th>reg</th>
<th>r/m</th>
</tr>
</thead>
</table>
- 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 R/M Table for R/M operand
### REG table

<table>
<thead>
<tr>
<th>REG w=0</th>
<th>REG w=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 AL</td>
<td>000 AH</td>
</tr>
<tr>
<td>001 CL</td>
<td>001 CH</td>
</tr>
<tr>
<td>010 DL</td>
<td>010 DH</td>
</tr>
<tr>
<td>011 BL</td>
<td>011 BH</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>For 32 bit code</td>
<td></td>
</tr>
</tbody>
</table>

### R/M Tables

<table>
<thead>
<tr>
<th>R/M Table 1 (Mod = 00)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 [BX+SI]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R/M Table 2 (Mod = 01 or 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 [BX+SI]</td>
</tr>
</tbody>
</table>

### 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
- Indicate 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)
- All instructions have the form:
  - Opcode AddrMode Disp8
- Examples:
  - MOV AX, [2A45]
  - MOV AX, [DI]

### Addressing Mode 01

- Specifies R/M Table 2 with 8-bit signed displacement
- Add DISP to register specified:
- 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
- Examples:
  - 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
  - ADD CX, 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:
- All instructions have the form:
  Opcode AddMode 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 = 0 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 1001 1000 (89 D8)
- If d = 1 then REG = source (AX) and R/M = dest (BX)
  = 1000 1011 1010 0011 (8B C3)

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
  10110011 imm B7 03 00
- MOV bx, 3 1011 w=1 reg=111=BH
  10110011 imm B7 03
- MOV bl, 3 1011 w=1 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 REG
  w = 1 because AX is 8 bits
  10100011 addr = A3 F4 C4
POP Reg/Mem

- POP memory/register has the structure:
  \[ 8F \text{ MOD} X X \text{ R/M} \]
- 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:
  \[ \text{MOD} = 11 \text{ (Use REG table)} \quad \text{R/M} = 000 \rightarrow 11 000 000 \]
  Encoding: 8F C0
- To POP into BP:
  \[ \text{MOD} = 11 \text{ (Use REG table)} \quad \text{R/M} = 101 \rightarrow 11 000 101 \]
  Encoding: 8F C5

POP General Register

- This one-byte opcode has the structure:
  \[ 01011 \text{ REG} \]
- 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:
  \[ 00 \text{seg} 111 \]
- 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
- \[ \text{add imm to reg/mem} \quad 1000 \text{ 00sw mod} 000x \text{r/m} \]
- \[ \text{or imm to reg/mem} \quad 1000 \text{ 00sw mod} 001x \text{r/m} \]
- 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

- \[ \text{add} \quad d \quad \text{imm to reg/mem} \]
- \[ 1000 \text{ 00sw mod} 000x \text{r/m} \]
- \[ w=1 (DX is 16 bits) \quad \text{mod} = 11 \text{ (use REG table)} \quad r/m = 010 = DX \]
- With s bit set we have
  \[ 1000 \text{ 0011 11 000 010} \quad \text{operand} = 83 \text{ C2 03} \]
- With s bit clear we have
  \[ 1000 \text{ 0001 11 000 010} \quad \text{operand} = 81 \text{ C2 03 00} \]
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
1. Add some opcodes for new instruction
2. Treat w bit as 0=8 bits, 1=32 bits, so in REG field interpret w=1 and REG=000 as eax, not ax
3. Add operand size prefix byte to handle 16-bit operands
4. Likewise 8/16 bit displacement, imm data, or address values are treated as 8/32 bit
5. 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 = 00000111 00011100 10001000 03 0C 88