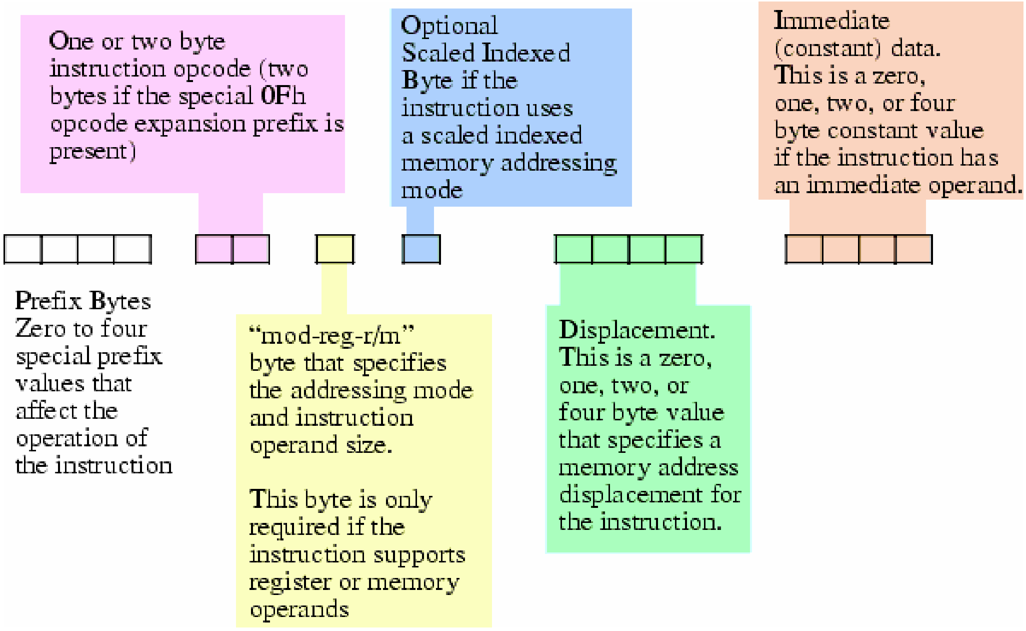
CIS-77 Home http://www.c-jump.com/CIS77/CIS77syllabus.htm

Encoding Real x86 Instructions

1. Encoding Real x86 Instructions

- It is time to take a look that the actual machine instruction format of the x86 CPU family.
- They don't call the x86 CPU a Complex Instruction Set Computer (CISC) for nothing!
- Although more complex instruction encodings exist, no one is going to challenge that the x86 has a complex instruction encoding:

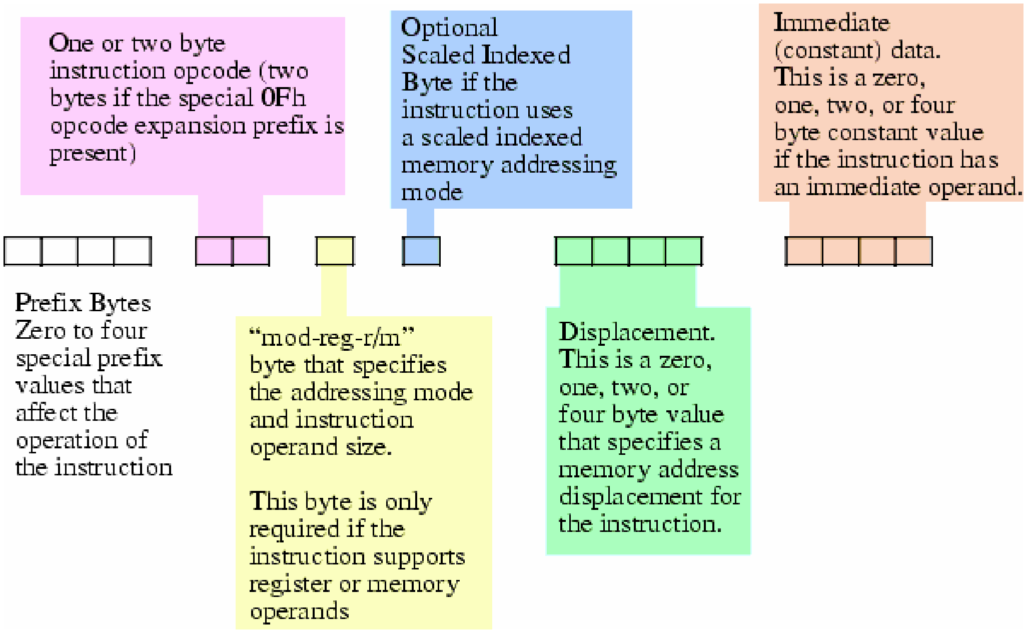

One or two byte instruction opcode (two bytes if the special 0Fh opcode expansion prefix is present)
Optional Scaled Indexed Byte if the instruction uses a scaled indexed memory addressing mode
Immediate (constant) data. This is a zero, one, two, or four byte constant value if the instruction has an immediate operand.
Prefix Bytes Zero to four special prefix values that affect the operation of the instruction
"mod-reg-r/m" byte that specifies the addressing mode and instruction operand size.
This byte is only required if the instruction supports register or memory operands
Displacement. This is a zero, one, two, or four byte value that specifies a memory address displacement for the instruction.

2. x86 Instructions Overview

Although the diagram seems to imply that instructions can be up to 16 bytes long, in actuality the x86 will not allow instructions greater than 15 bytes in length.

The prefix bytes **are not** the *opcode expansion prefix* discussed earlier - they are special bytes to modify the behavior of existing instructions.

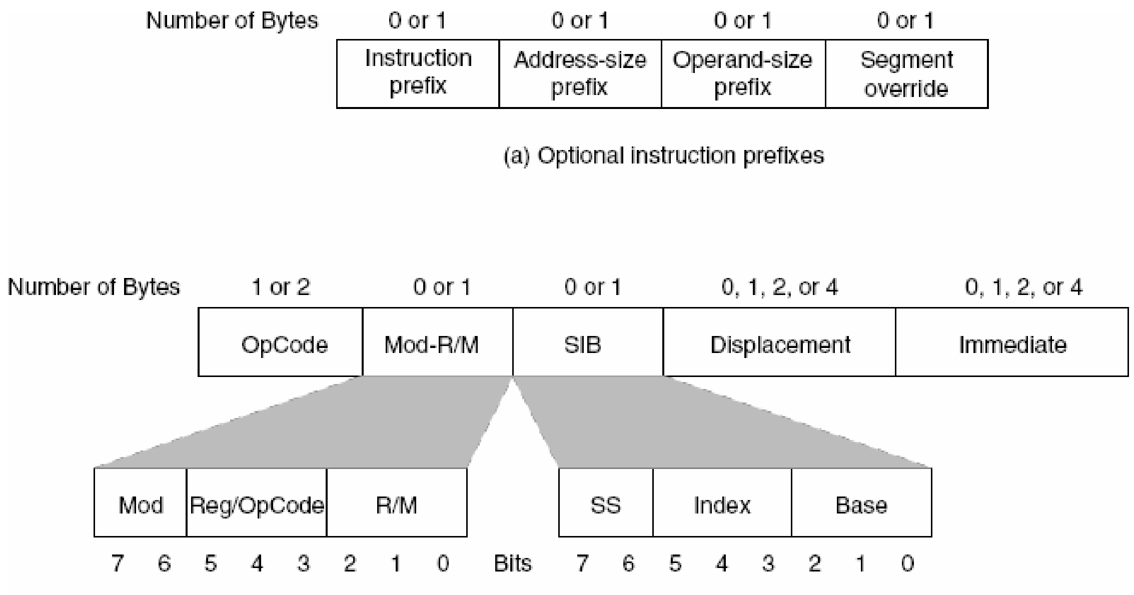
x86 Instruction Encoding:


One or two byte instruction opcode (two bytes if the special 0Fh opcode expansion prefix is present)
Optional Scaled Indexed Byte if the instruction uses a scaled indexed memory addressing mode
Immediate (constant) data. This is a zero, one, two, or four byte constant value if the instruction has an immediate operand.
Prefix Bytes Zero to four special prefix values that affect the operation of the instruction
"mod-reg-r/m" byte that specifies the addressing mode and instruction operand size.
This byte is only required if the instruction supports register or memory operands
Displacement. This is a zero, one, two, or four byte value that specifies a memory address displacement for the instruction.

3. x86 Instruction Format Reference

Another view of the x86 instruction format:

Additional reference:



(a) Optional instruction prefixes

(b) General instruction format

- Intel x86 instructions by opcode
- Intel x86 instructions by mnemonic
- Brief Intel x86 instruction reference.

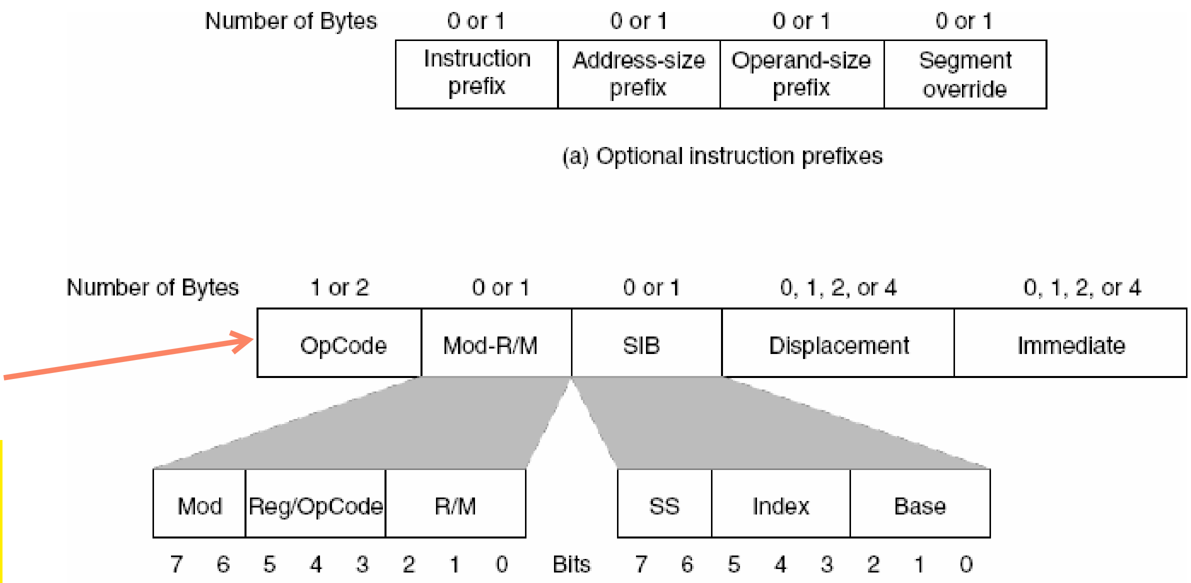
4. x86 Opcode Sizes

The x86 CPU supports two basic opcode sizes:

1. standard **one**-byte opcode
2. **two**-byte opcode consisting of a **0Fh** *opcode expansion prefix byte*.

The second byte then specifies the actual instruction.

x86 instruction format:



(a) Optional instruction prefixes

(b) General instruction format

- The x86 opcode bytes are 8-bit equivalents of **iii** field that we discussed in simplified encoding.
- This provides for up to 512 different instruction classes, although the x86 does not yet use them all.

5. x86 ADD Instruction Opcode

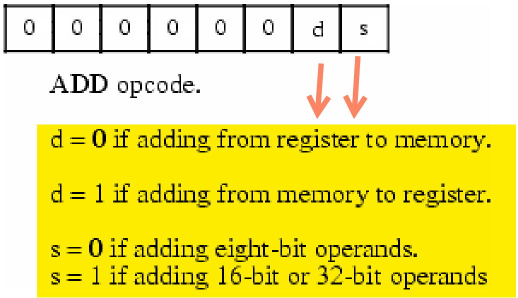
Bit number **zero** marked **s** specifies the **size** of the operands the ADD instruction operates upon:

- If **s = 0** then the operands are 8-bit registers and memory locations.
- If **s = 1** then the operands are either 16-bits or 32-bits:
  - Under 32-bit operating systems the default is 32-bit operands if **s = 1**.
  - To specify a 16-bit operand (under Windows or Linux) you must insert a special *operand-size prefix byte* in front of the instruction (example of this later.)

see page 9 for what problem it causes

You'll soon see that this direction bit **d** creates a problem that results in one instruction have two different possible opcodes.

x86 ADD instruction opcode :



d = 0 if adding from register to memory.

d = 1 if adding from memory to register.

s = 0 if adding eight-bit operands.
s = 1 if adding 16-bit or 32-bit operands

Bit number **one**, marked **d**, specifies the **direction** of the data transfer:

- If **d = 0** then the destination operand is a memory location, e.g.

value in al is stored at the address in the ebx register

```
add [ebx], al
```

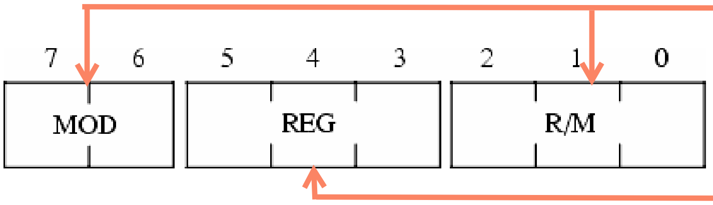
- If **d = 1** then the destination operand is a register, e.g.

value at the address that ebx contains will be copied to al register

```
add al, [ebx]
```

6. Encoding x86 Instruction Operands, MOD-REG-R/M Byte

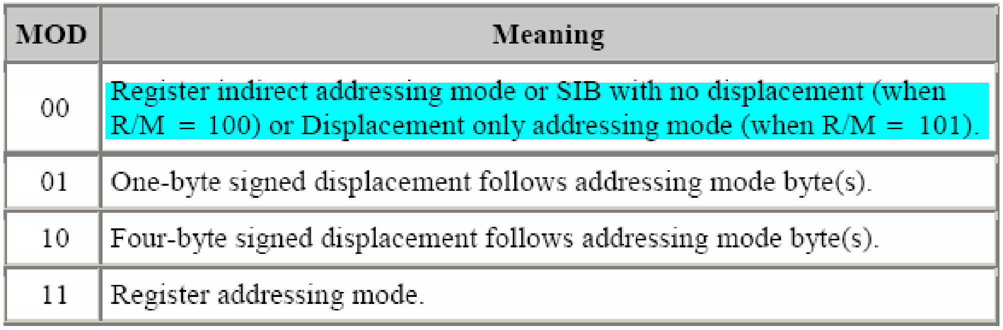
The MOD-REG-R/M byte specifies instruction operands and their addressing mode[(*)]:



both specifies a operand - desitnation is a memory location

it specifies a operand - destination is a register

The **MOD** field specifies x86 addressing mode:



MOD	Meaning
00	Register indirect addressing mode or SIB with no displacement (when R/M = 100) or Displacement only addressing mode (when R/M = 101).
01	One-byte signed displacement follows addressing mode byte(s).
10	Four-byte signed displacement follows addressing mode byte(s).
11	Register addressing mode.

The **REG** field specifies source or destination **register**:

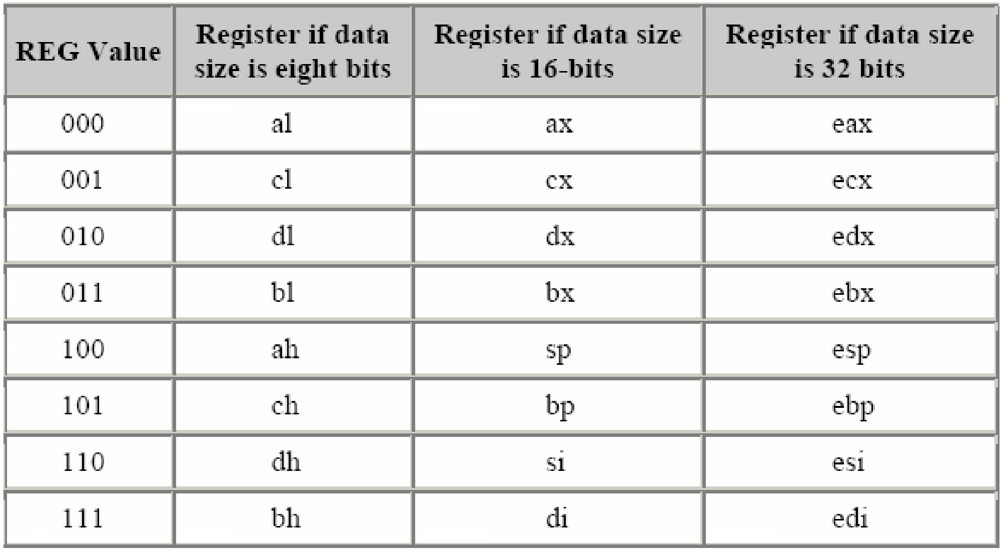
The **R/M** field, combined with **MOD**, specifies either

1. the second operand in a **two**-operand instruction, or
2. the only operand in a **single**-operand instruction like NOT or NEG.

The **d** bit in the opcode determines which operand is the source, and which is the destination:

d=0: MOD R/M <- REG, REG is the source

d=1: REG <- MOD R/M, REG is the destination



REG Value	Register if data size is eight bits	Register if data size is 16-bits	Register if data size is 32 bits
000	al	ax	eax
001	cl	cx	ecx
010	dl	dx	edx
011	bl	bx	ebx
100	ah	sp	esp
101	ch	bp	ebp
110	dh	si	esi
111	bh	di	edi

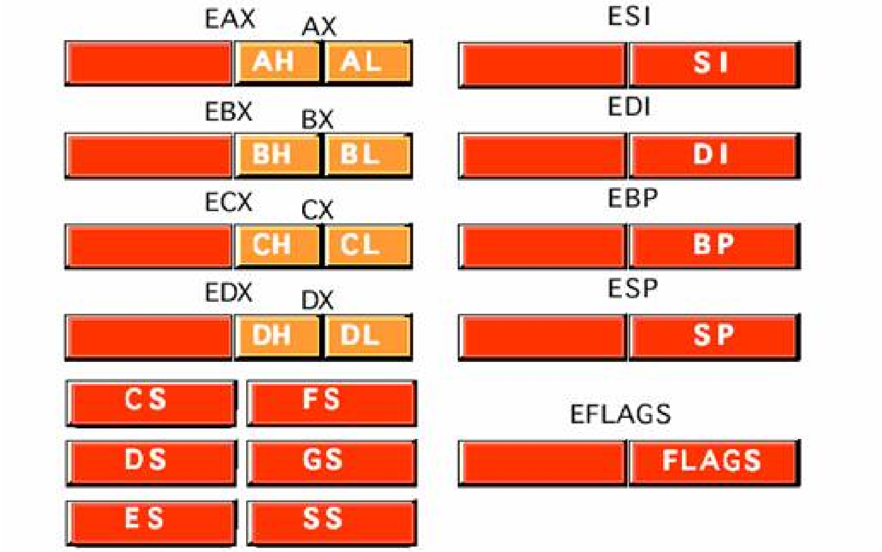
(*) Technically, registers do not have an address, but we apply the term *addressing mode* to registers nonetheless.

7. General-Purpose Registers

- The EAX, EDX, ECX, EBX, EBP, EDI, and ESI registers are 32-bit general-purpose registers, used for temporary data storage and memory access.

- The AX, DX, CX, BX, BP, DI, and SI registers are 16-bit equivalents of the above, they represent the low-order 16 bits of 32-bit registers.

- The AH, DH, CH, and BH registers represent the high-order 8 bits of the corresponding registers.

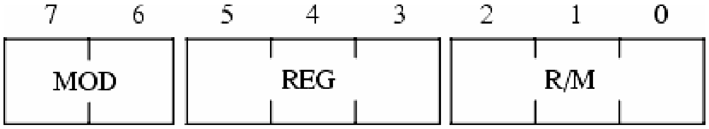


Since the processor accesses registers more quickly than it accesses memory, you can make your programs run faster by keeping the most-frequently used data in registers.

- Similarly, AL, DL, CL, and BL represent the low-order 8 bits of the registers.

8. REG Field of the MOD-REG-R/M Byte

Depending on the instruction , this can be



The **REG** field specifies an x86 register[(*)]:

REG Value	Register if data size is eight bits	Register if data size is 16-bits	Register if data size is 32 bits
000	al	ax	eax
001	cl	cx	ecx
010	dl	dx	edx
011	bl	bx	ebx
100	ah	sp	esp
101	ch	bp	ebp
110	dh	si	esi
111	bh	di	edi

either the *source* or the destination operand.

Many instructions have the **d** (direction) field in their opcode to choose **REG** operand role:

1. If **d=0**, **REG** is the source,
   MOD R/M <- REG.
2. If **d=1**, **REG** is the destination,
   REG <- MOD R/M.

[(*)] For certain (often single-operand or immediate-operand) instructions, the **REG** field may contain an *opcode extension* rather than the register bits. The **R/M** field will specify the operand in such case.

9. MOD R/M Byte and Addressing Modes

```
MOD R/M Addressing Mode
=== === ================================
 00 000 [ eax ]
 01 000 [ eax + disp8 ]               (1)
 10 000 [ eax + disp32 ]
 11 000 register  ( al / ax / eax )   (2)
 00 001 [ ecx ]
 01 001 [ ecx + disp8 ]
 10 001 [ ecx + disp32 ]
 11 001 register  ( cl / cx / ecx )
 00 010 [ edx ]
 01 010 [ edx + disp8 ]
 10 010 [ edx + disp32 ]
 11 010 register  ( dl / dx / edx )
 00 011 [ ebx ]
 01 011 [ ebx + disp8 ]
 10 011 [ ebx + disp32 ]
 11 011 register  ( bl / bx / ebx )
 00 100 SIB  Mode                     (3)
 01 100 SIB  +  disp8  Mode
 10 100 SIB  +  disp32  Mode
 11 100 register  ( ah / sp / esp )
 00 101 32-bit Displacement-Only Mode (4)
 01 101 [ ebp + disp8 ]
 10 101 [ ebp + disp32 ]
 11 101 register  ( ch / bp / ebp )
 00 110 [ esi ]
 01 110 [ esi + disp8 ]
 10 110 [ esi + disp32 ]
```

1. Addressing modes with 8-bit displacement fall in the range **-128..+127** and require only a single byte displacement after the opcode (Faster!)
2. The size bit in the opcode specifies 8 or 32-bit register size. To select a 16-bit register requires a prefix byte.
3. The so-called scaled indexed addressing modes, **SIB** = scaled index byte mode.
4. Note that there is no [ebp] addressing. It's slot is occupied by the 32-bit *displacement only* addressing mode. Intel decided that programmers can use [ebp+ **disp8**] addressing mode instead, with its 8-bit displacement set equal to zero (instruction is a little longer, though.)

it is a trick , note it carefully

```
11 110 register  ( dh / si / esi )
00 111 [ edi ]
01 111 [ edi + disp8 ]
10 111 [ edi + disp32 ]
11 111 register  ( bh / di / edi )
```

10. SIB (Scaled Index Byte) Layout

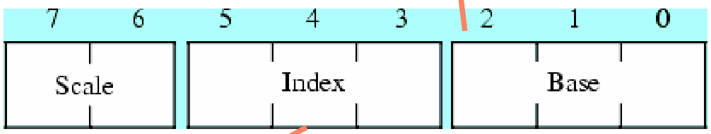
- *Scaled indexed addressing mode* uses the second byte (namely, **SIB** byte) that follows the **MOD-REG-R/M** byte in the instruction format.
- The **MOD** field still specifies the displacement size of **zero**, **one**, or **four** bytes.

The **MOD-REG-R/M** and SIB bytes are complex, because Intel reused 16-bit addressing circuitry in the 32-bit mode, rather than simply abandoning the 16-bit format in the 32-bit mode.

There are good hardware reasons for this, but the end result is a complex scheme for specifying addressing modes in the opcodes.

when base is 101 and mode 00 only displacement and no base

Scaled index byte layout:



no 100 --> so no ESP

Index	Register
000	EAX
001	ECX
010	EDX
011	EBX
100	Illegal
101	EBP
110	ESI
111	EDI

intel reserved ESP register for future devpmnt

Scale Value	Index*Scale Value
00	Index*1
01	Index*2
10	Index*4
11	Index*8

Base	Register
000	EAX
001	ECX
010	EDX
011	EBX
100	ESP
101	Displacement-only if MOD = 00, EBP if MOD = 01 or 10
110	ESI
111	EDI

there will be no BASE register if MOD=00 and BASE=101

11. Scaled Indexed Addressing Mode

any GP register can be used here

```
[ reg32 + eax*n ] MOD = 00
[ reg32 + ebx*n ]
[ reg32 + ecx*n ]
[ reg32 + edx*n ]
[ reg32 + ebp*n ]
[ reg32 + esi*n ]
[ reg32 + edi*n ]

[ disp + reg8 + eax*n ] MOD = 01
[ disp + reg8 + ebx*n ]
[ disp + reg8 + ecx*n ]
[ disp + reg8 + edx*n ]
[ disp + reg8 + ebp*n ]
[ disp + reg8 + esi*n ]
[ disp + reg8 + edi*n ]

[ disp + reg32 + eax*n ] MOD = 10
[ disp + reg32 + ebx*n ]
[ disp + reg32 + ecx*n ]
[ disp + reg32 + edx*n ]
[ disp + reg32 + ebp*n ]
[ disp + reg32 + esi*n ]
[ disp + reg32 + edi*n ]

[ disp + eax*n ] MOD = 00, and
[ disp + ebx*n ] BASE field = 101
[ disp + ecx*n ]
[ disp + edx*n ]
[ disp + ebp*n ]
[ disp + esi*n ]
[ disp + edi*n ]
```

displacement only

Note: **n** = **1**, **2**, **4**, or **8**.

In each **scaled indexed addressing mode** the **MOD** field in **MOD-REG-R/M** byte specifies the *size of the displacement*. It can be zero, one, or four bytes:

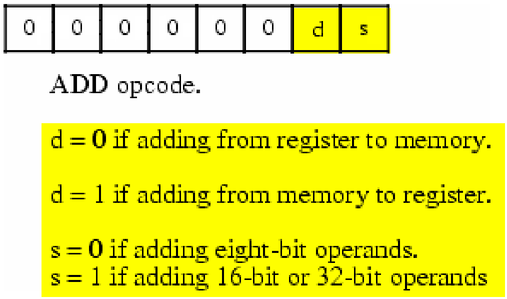
```
MOD R/M  Addressing Mode
--- ---  ---------------------------
 00 100  SIB
 01 100  SIB + disp8
 10 100  SIB + disp32
```

The **Base** and **Index** fields of the SIB byte select the base and index registers, respectively.

Note that this addressing mode does not allow the use of the **ESP** register as an index register. Presumably, Intel left this particular mode undefined to provide the ability to extend the addressing modes in a future version of the CPU.

12. Encoding ADD Instruction Example

The ADD opcode can be decimal 0, 1, 2, or 3, depending on the direction and size bits in the opcode:



0	0	0	0	0	0	d	s

ADD opcode.

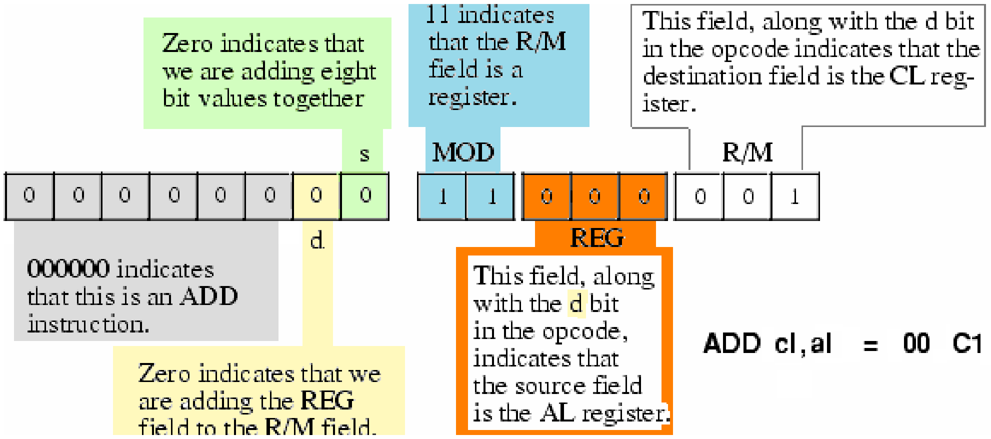
d = 0 if adding from register to memory.

d = 1 if adding from memory to register.

s = 0 if adding eight-bit operands.
s = 1 if adding 16-bit or 32-bit operands

How could we encode various forms of the ADD instruction using different addressing modes?

13. Encoding ADD CL, AL Instruction


Zero indicates that we are adding eight bit values together
11 indicates that the R/M field is a register.
This field, along with the d bit in the opcode indicates that the destination field is the CL register.
s
MOD
R/M
0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 1
d
REG
000000 indicates that this is an ADD instruction.
This field, along with the d bit in the opcode, indicates that the source field is the AL register.
ADD cl,al = 00 C1
Zero indicates that we are adding the REG field to the R/M field.

- Interesting side effect of the direction bit and the **MOD-REG-R/M byte** organization: some instructions can have two different opcodes, and both are legal!

- For example, encoding of

both opcodes does the same job

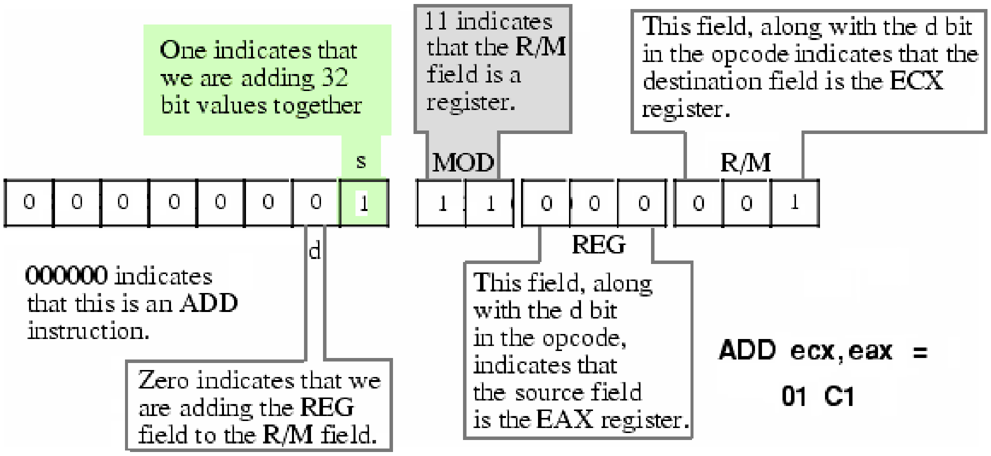
```
add cl, al
```

could be **00 C1** (if **d=0**), or **02 C8**, if **d** bit is set to **1**.

- The possibility of opcode duality issue here applies to all instructions with two register operands.

14. Encoding ADD ECX, EAX Instruction

```
add ecx, eax
```

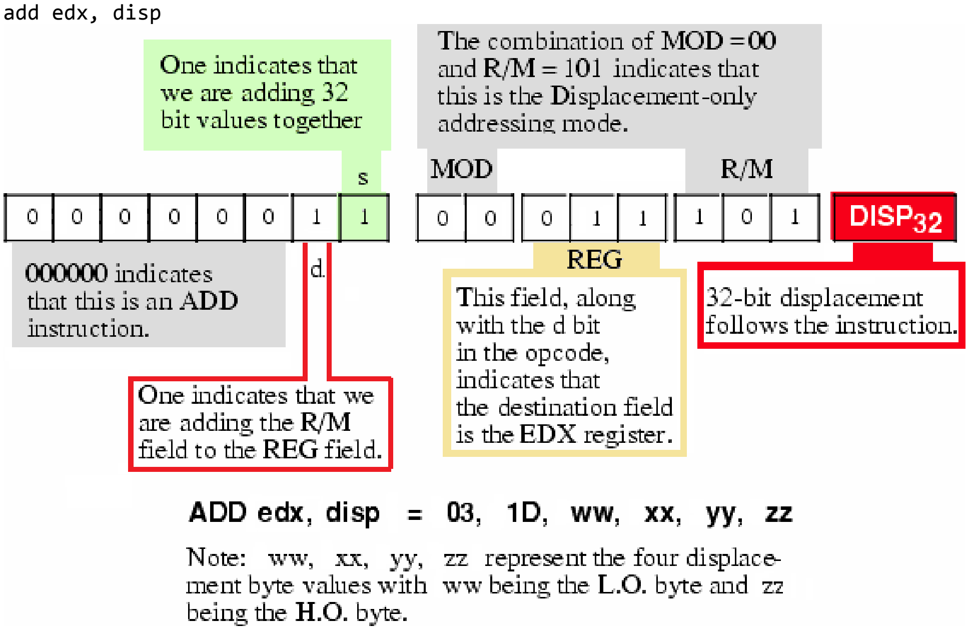

One indicates that we are adding 32 bit values together
11 indicates that the R/M field is a register.
This field, along with the d bit in the opcode indicates that the destination field is the ECX register.
s
MOD
R/M
0 0 0 0 0 0 0 1
1 1 0 0 0 0 0 1
d
REG
000000 indicates that this is an ADD instruction.
This field, along with the d bit in the opcode, indicates that the source field is the EAX register.
ADD ecx,eax = 01 C1
Zero indicates that we are adding the REG field to the R/M field.

Note that we could also encode ADD **ECX**, **EAX** using the bytes **03 C8**.

15. Encoding ADD EDX, DISPLACEMENT Instruction

Encoding the ADD **EDX**, DISP Instruction:

```
add edx, disp
```



One indicates that we are adding 32 bit values together

The combination of MOD = 00 and R/M = 101 indicates that this is the Displacement-only addressing mode.

000000 indicates that this is an ADD instruction.

One indicates that we are adding the R/M field to the REG field.

This field, along with the d bit in the opcode, indicates that the destination field is the EDX register.

32-bit displacement follows the instruction.

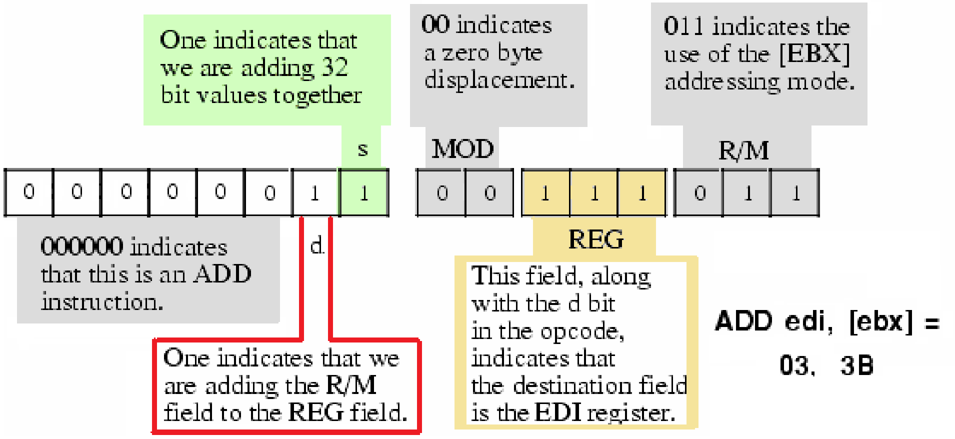
ADD edx, disp = 03, 1D, ww, xx, yy, zz

Note: ww, xx, yy, zz represent the four displacement byte values with ww being the L.O. byte and zz being the H.O. byte.

16. Encoding ADD EDI, [EBX] Instruction

Encoding the ADD **EDI**, [**EBX**] instruction:

```
add edi, [ebx]
```



One indicates that we are adding 32 bit values together

00 indicates a zero byte displacement.

011 indicates the use of the [EBX] addressing mode.

000000 indicates that this is an ADD instruction.

One indicates that we are adding the R/M field to the REG field.

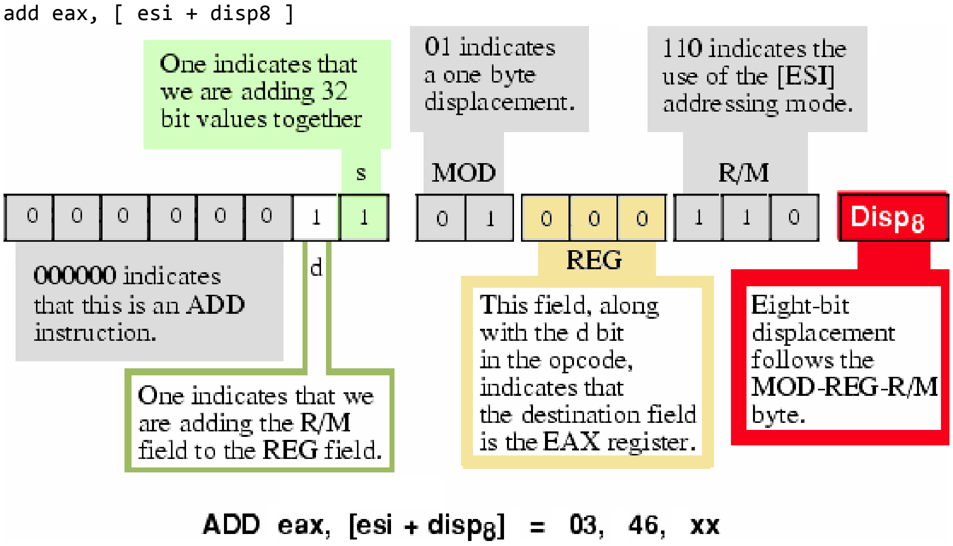
This field, along with the d bit in the opcode, indicates that the destination field is the EDI register.

ADD edi, [ebx] = 03, 3B

17. Encoding ADD EAX, [ESI + disp8] Instruction

Encoding the ADD **EAX**, [**ESI** + **disp8**] instruction:

```
add eax, [ esi + disp8 ]
```

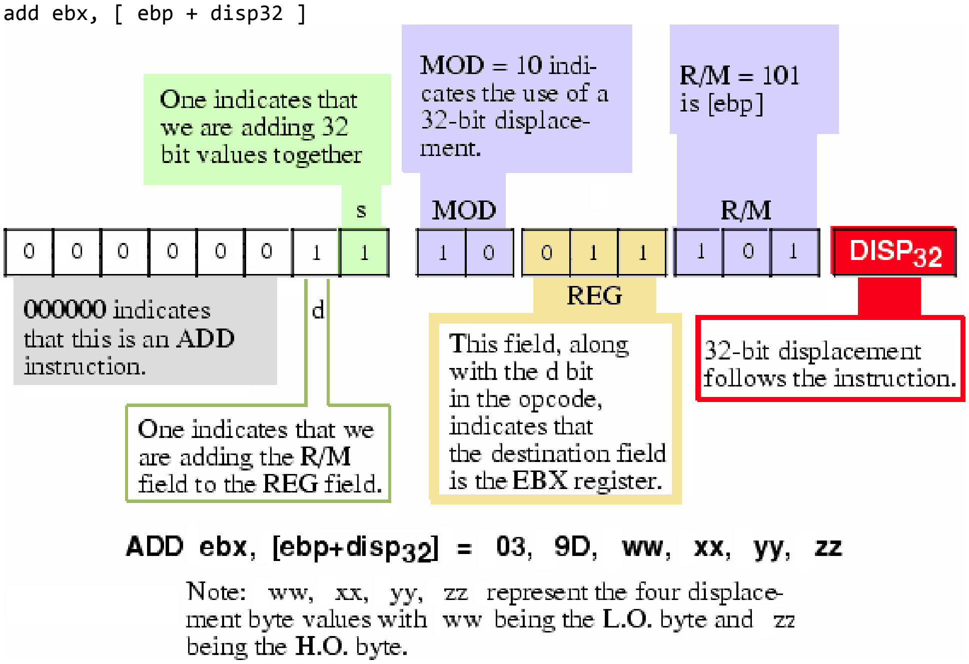

One indicates that we are adding 32 bit values together
01 indicates a one byte displacement.
110 indicates the use of the [ESI] addressing mode.
s
MOD
R/M
0 0 0 0 0 0 1 1
0 1
0 0 0
1 1 0
Disp$_8$
d
REG
000000 indicates that this is an ADD instruction.
One indicates that we are adding the R/M field to the REG field.
This field, along with the d bit in the opcode, indicates that the destination field is the EAX register.
Eight-bit displacement follows the MOD-REG-R/M byte.

ADD eax, [esi + $disp_8$] = 03, 46, xx

18. Encoding ADD EBX, [EBP + disp32] Instruction

Encoding the ADD **EBX**, [**EBP** + **disp32**] instruction:

```
add ebx, [ ebp + disp32 ]
```


One indicates that we are adding 32 bit values together
MOD = 10 indicates the use of a 32-bit displacement.
R/M = 101 is [ebp]
s
MOD
R/M
0 0 0 0 0 0 1 1
1 0
0 1 1
1 0 1
DISP$_{32}$
d
REG
000000 indicates that this is an ADD instruction.
One indicates that we are adding the R/M field to the REG field.
This field, along with the d bit in the opcode, indicates that the destination field is the EBX register.
32-bit displacement follows the instruction.

ADD ebx, [ebp+$disp_{32}$] = 03, 9D, ww, xx, yy, zz

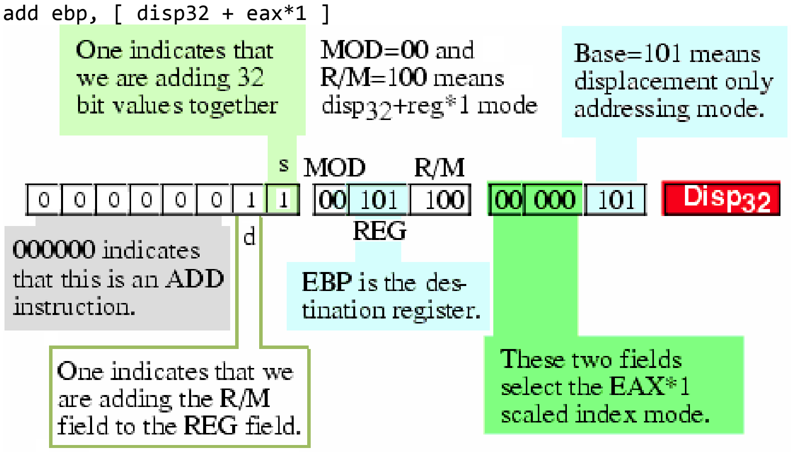
Note: ww, xx, yy, zz represent the four displacement byte values with ww being the L.O. byte and zz being the H.O. byte.

base + index*scacle

19. Encoding ADD EBP, [disp32 + EAX*1] Instruction

Encoding the ADD **EBP**, [**disp32** + **EAX***1] Instruction

```
add ebp, [ disp32 + eax*1 ]
```


One indicates that we are adding 32 bit values together
MOD=00 and R/M=100 means disp32+reg*1 mode
Base=101 means displacement only addressing mode.
s
MOD
R/M
0 0 0 0 0 0 1 1
00 101 100
00 000 101
Disp32
d
REG
000000 indicates that this is an ADD instruction.
EBP is the destination register.
One indicates that we are adding the R/M field to the REG field.
These two fields select the EAX*1 scaled index mode.

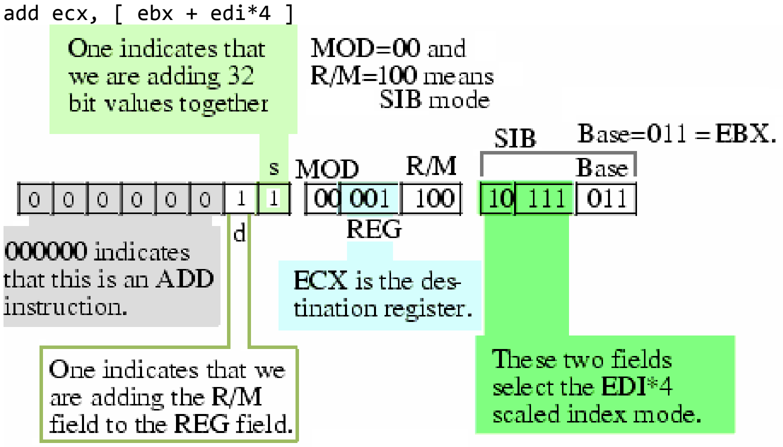
ADD ebp, [disp_{32} + eax*1] = 03, 2C, 05, ww, xx, yy, zz

Note: ww, xx, yy, zz represent the four displacement byte values with ww being the L.O. byte and zz being the H.O. byte.

20. Encoding ADD ECX, [EBX + EDI*4] Instruction

Encoding the ADD **ECX**, [**EBX** + **EDI***4] Instruction

```
add ecx, [ ebx + edi*4 ]
```


One indicates that we are adding 32 bit values together
MOD=00 and R/M=100 means SIB mode
SIB
Base=011 = EBX.
Base
s
MOD
R/M
0 0 0 0 0 0 1 1
00 001 100
10 111 011
d
REG
000000 indicates that this is an ADD instruction.
ECX is the destination register.
One indicates that we are adding the R/M field to the REG field.
These two fields select the EDI*4 scaled index mode.

ADD ecx, [ebx+ edi*4] = 03, 0C, BB

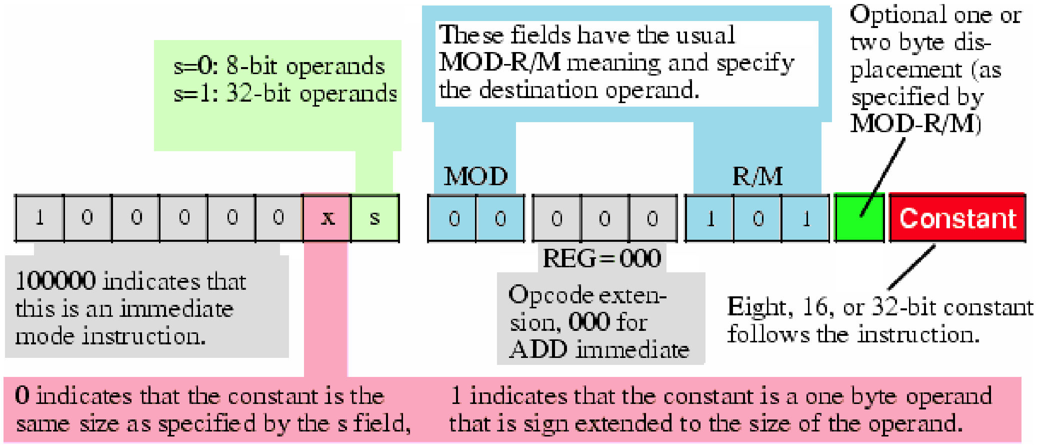
21. Encoding ADD Immediate Instruction

MOD-REG-R/M and **SIB** bytes have no bit combinations to specify an immediate operand.

Encoding x86 immediate operands:

Instead, x86 uses a entirely different instruction format to specify instruction with an immediate operand.

There are three rules that apply:


s=0: 8-bit operands
s=1: 32-bit operands
These fields have the usual MOD-R/M meaning and specify the destination operand.
Optional one or two byte dis-placement (as specified by MOD-R/M)
MOD
R/M
1 0 0 0 0 0 x s
0 0
0 0 0
1 0 1
Constant
REG=000
100000 indicates that this is an immediate mode instruction.
Opcode exten-sion, 000 for ADD immediate
Eight, 16, or 32-bit constant follows the instruction.
0 indicates that the constant is the same size as specified by the s field,
1 indicates that the constant is a one byte operand that is sign extended to the size of the operand.

1. If opcode high-order bit set to **1**, then *instruction has an immediate constant*.

2. There is no direction bit in the opcode:

    : indeed, you cannot specify a constant as a destination operand!

    Therefore, destination operand is always the location encoded in the **MOD-R/M** bits of the the **MOD-REG-R/M** byte.

    In place of the direction bit **d**, the opcode has a sign extension **x** bit instead:

    - For 8-bit operands, the CPU ignores **x** bit.
    - For 16-bit and 32-bit operands, **x** bit specifies the size of the Constant following at the end of the instruction:
        - If **x** bit contains **zero**, the Constant is the same size as the operand (i.e., 16 or 32 bits).
        - If **x** bit contains **one**, the Constant is a signed 8-bit value, and the CPU sign-extends this value to the appropriate size before adding it to the operand.
    - This little **x** trick often makes programs shorter, because adding small-value constants to 16 or 32 bit operands is very common.

3. The third difference between the ADD-immediate and the standard ADD instruction is the meaning of the **REG** field in the **MOD-REG-R/M** byte:

    - Since the instruction implies that
        - the source operand is a constant, and
        - **MOD-R/M** fields specify the destination operand,

      the instruction does not need to use the **REG** field to specify an operand.
    - Instead, the x86 CPU uses these three bits as an **opcode extension**.
    - For the ADD-immediate instruction the **REG** bits must contain zero.
    - Other bit patterns would correspond to a different instruction.

    Note that when adding a constant to a memory location, the displacement (if any) immediately precedes the immediate (constant) value in the opcode sequence.

22. Encoding Eight, Sixteen, and Thirty-Two Bit Operands

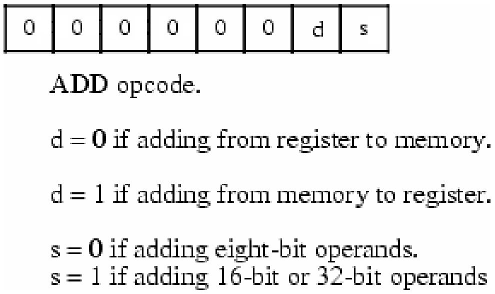
- When Intel designed the 8086, one bit in the opcode, **s**, selected between 8 and 16 bit integer operand sizes.

- Later, when CPU added 32-bit integers to its architecture on 80386 chip, there was a problem:

  three encodings were needed to support 8, 16, and 32 bit sizes.

- Solution was an *operand size prefix byte*.

x86 ADD Opcode:



ADD opcode.

d = 0 if adding from register to memory.

d = 1 if adding from memory to register.

s = 0 if adding eight-bit operands.
s = 1 if adding 16-bit or 32-bit operands

- Intel studied x86 instruction set and came to the conclusion:

  in a 32-bit environment, programs were more likely to use 8-bit and 32-bit operands far more often than 16-bit operands.

- So Intel decided to let the size bit **s** in the opcode select between 8- and 32-bit operands.

why s bit determines 8 bit and 32 bit operands

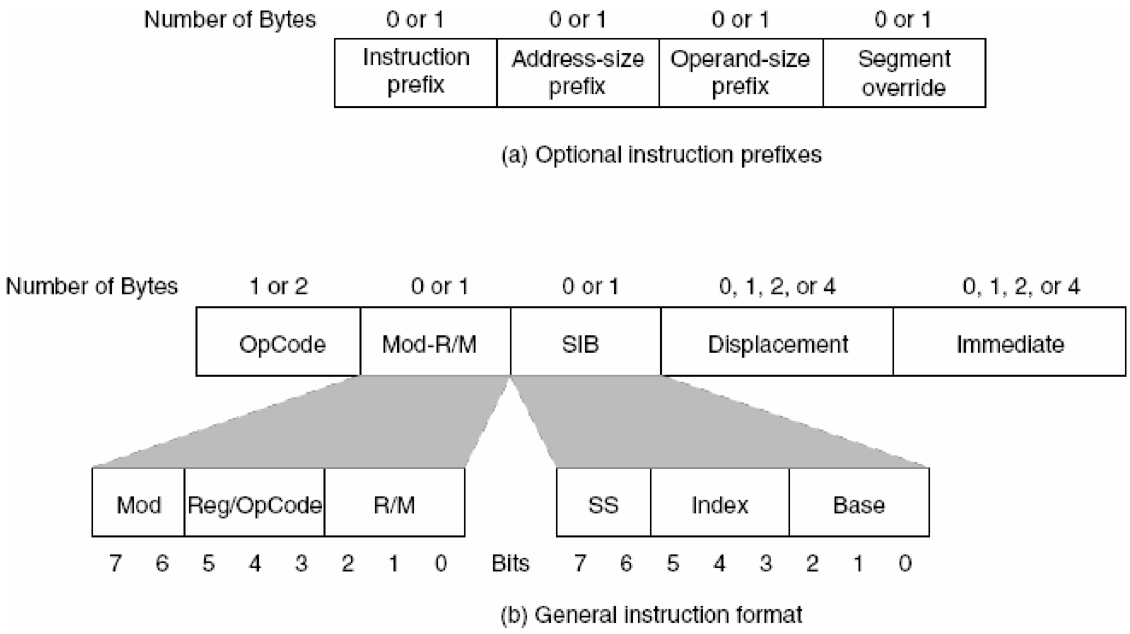
23. Encoding Sixteen Bit Operands

32-bit programs don't use 16-bit operands that often, but they do need them now and then.

To allow for 16-bit operands, Intel added prefix a 32-bit mode instruction with the *operand size prefix byte* with value **66h**.

This prefix byte tells the CPU to operand on 16-bit data rather than 32-bit data.

x86 instruction format:



(a) Optional instruction prefixes

(b) General instruction format

- There is nothing programmer has to do explicitly to put an operand size prefix byte in front of a 16-bit instruction:

  the assembler does this automatically as soon as 16-bit operand is found in the instruction.

- However, keep in mind that whenever you use a 16-bit operand in a 32-bit program, the instruction is longer by one byte:

```
Opcode     Instruction
--------   ------------
41h        INC ECX
66h 41h    INC CX
```

- Be careful about using 16-bit instructions if size (and to a lesser extent, speed) are important, because

1. instructions are longer, and
2. slower because of their effect on the instruction cache.

24. x86 Instruction Prefix Bytes

- x86 instruction can have up to 4 prefixes.
- Each prefix adjusts interpretation of the opcode:

1. **Repeat/lock** prefix byte guarantees that instruction will have exclusive use of all shared memory, until the instruction completes execution:

```
        F0h = LOCK
```

2. **String manipulation** instruction prefixes

```
        F3h = REP, REPE
        F2h = REPNE
```

   where

   REP repeats instruction the number of times specified by *iteration count* ECX.

   REPE and REPNE prefixes allow to terminate loop on the value of **ZF** CPU flag.

   Related string manipulation instructions are:

   - MOVS, move string
   - STOS, store string
   - SCAS, scan string
   - CMPS, compare string, etc.

   See also string manipulation sample program: rep_movsb.asm

3. **Segment override** prefix causes memory access to use *specified segment* instead of *default segment* designated for instruction operand.

```
        2Eh = CS
        36h = SS
        3Eh = DS
        26h = ES
        64h = FS
        65h = GS
```

4. **Operand override**, **66h**. Changes size of data expected by default mode of the instruction e.g. 16-bit to 32-bit and vice versa.

5. **Address override**, **67h**. Changes size of address expected by the instruction. 32-bit address could switch to 16-bit and vice versa.

25. Alternate Encodings for Instructions

- To shorten program code, Intel created alternate (shorter) encodings of some very commonly used instructions.
- For example, x86 provides a single byte opcode for

  ```
  add al, constant    ; one-byte opcode and no MOD-REG-R/M byte
  add eax, constant   ; one-byte opcode and no MOD-REG-R/M byte
  ```

  the opcodes are **04h** and **05h**, respectively. Also,
- These instructions are one byte shorter than their standard ADD immediate counterparts.
- Note that

  ```
  add ax, constant   ; operand size prefix byte + one-byte opcode, no MOD-REG-R/M byte
  ```

  requires an operand size prefix just as a standard ADD **AX**, **constant** instruction, yet is still one byte shorter than the corresponding standard version of ADD immediate.
- Any decent assembler will *automatically* choose the shortest possible instruction when translating program into machine code.
- Intel only provides alternate encodings only for the accumulator registers **AL**, **AX**, **EAX**.
- This is a good reason to use accumulator registers if you have a choice

  (also a good reason to take some time and study encodings of the x86 instructions.)

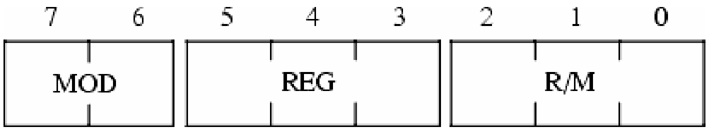
26. x86 Opcode Summary

- x86 opcodes are represented by one or two bytes.
- Opcode could extend into unused bits of **MOD-REG-R/M** byte.
- Opcode encodes information about
  - operation type,
  - operands,
  - size of each operand, including the size of an immediate operand.

27. MOD-REG-R/M Byte Summary

- **MOD-REG-R/M** byte follows one or two opcode bytes of the instruction
- It provides *addressing mode* information for one or two operands.

MOD-REG-R/M Byte:



- If operand is in memory, or operand is a register:

- **MOD** field (bits [7:6]), combined with the **R/M** field (bits [2:0]), specify memory/register operand, as well as its *addressing mode*.
- **REG** field (bits [5:3]) specifies another register operand in of the two-operand instruction.

28. ISA Design Considerations

- Instruction set architecture design that can stand the test of time is a true intellectual challenge.
- It takes several compromises between space and efficiency to assign opcodes and encode instruction formats.
- Today people are using Intel x86 instruction set for purposes never intended by original designers.
- Extending the CPU is a very difficult task.
- The instruction set can become *extremely complex*.
- If x86 CPU was designed *from scratch* today, it would have a totally different ISA!
- Software developers usually don't have a problem adapting to a new architecture when writing new software...

  ...but they are very resistant to moving existing software from one platform to another.
- This is the primary reason the Intel x86 platform remains so popular to this day.

29. ISA Design Challenges

- Allowing for future expansion of the chip requires some *undefined opcodes*.
- From the beginning there should be a balance between the number of undefined opcodes and
  1. the number of initial instructions, and
  2. the size of your opcodes (including special assignments.)
- Hard decisions:
  - Reduce the number of instructions in the initial instruction set?
  - Increase the size of the opcode?
  - Rely on an opcode prefix byte(s), which makes later added instructions longer?
- There are no easy answers to these challenges for CPU designers!

30. Intel Architecture Software Developer's Manual

Classic Intel Pentium II Architecture Software Developer's Manual contains three parts:

1. Volume 1 , *Intel Basic Architecture*: Order Number 243190 , PDF, 2.6 MB.

2. Volume 2 , *Instruction Set Reference*: Order Number 243191 , PDF, 6.6 MB.
3. Volume 3 , *System Programing Guide*: Order Number 243192 , PDF, 5.1 MB.

It is highly recommended that you download the above manuals and use them as a reference.

31. Intel Instruction Set Reference (Volume2)

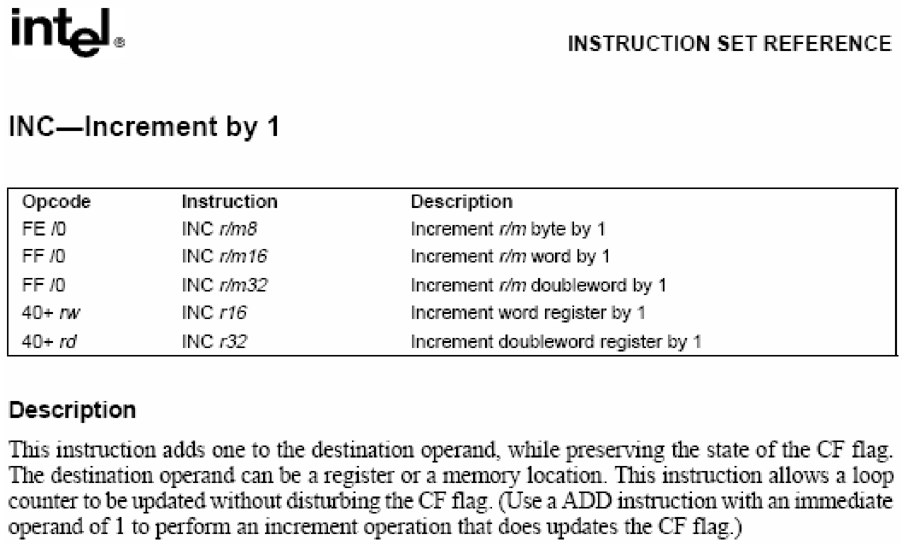
- **Chapter 3** of the Instruction Set Reference describes
  - each Intel instruction in detail
  - algorithmic description of each operation
  - effect on flags
  - operand(s), their sizes and attributes
  - CPU exceptions that may be generated.
- The instructions are arranged in alphabetical order.
- **Appendix A** provides *opcode map* for the entire Intel Architecture instruction set.

32. Chapter 3 of Intel Instruction Set Reference

- **Chapter 3** begins with instruction format example and explains the **Opcode** column encoding.
- The **Opcode** column gives the *complete machine codes* as it is understood by the CPU.
- When possible, the actual *machine code bytes* are given as exact hexadecimal bytes, in the same order in which they appear in memory.
- However, there are opcode definitions other than hexadecimal bytes...

33. Intel Reference Opcode Bytes

Fow example,



INSTRUCTION SET REFERENCE

INC—Increment by 1

Opcode	Instruction	Description
FE /0	INC *r/m8*	Increment *r/m* byte by 1
FF /0	INC *r/m16*	Increment *r/m* word by 1
FF /0	INC *r/m32*	Increment *r/m* doubleword by 1
40+ *rw*	INC *r16*	Increment word register by 1
40+ *rd*	INC *r32*	Increment doubleword register by 1

Description

This instruction adds one to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

34. Intel Reference Opcode Bytes, Cont.

- `/digit` - A digit between 0 and 7 indicates that
  - The **reg** field of **Mod R/M** byte contains the instruction opcode extension.
  - The **r/m** (register or memory) operand of **Mod R/M** byte indicates

```
R/M Addressing Mode
=== ===========================
000 register ( al / ax / eax )
001 register ( cl / cx / ecx )
010 register ( dl / dx / edx )
011 register ( bl / bx / ebx )
100 register ( ah / sp / esp )
101 register ( ch / bp / ebp )
110 register ( dh / si / esi )
111 register ( bh / di / edi )
```

- The size bit in the opcode specifies 8 or 32-bit register size.
- A 16-bit register requires a prefix byte:

```
Opcode      Instruction
--------    ------------
41h         INC ECX
66h 41h     INC CX
```

35. Intel Reference Opcode Bytes, Cont.

- `/r` - Indicates that the instruction uses the **Mod R/M** byte of the instruction.
- **Mod R/M** byte contains both
  - a register operand **reg** and
  - an **r/m** (register or memory) operand.

36. Intel Reference Opcode Bytes, Cont.

- `cb, cw, cd, cp` - A 1-byte (cb), 2-byte (cw), 4-byte (cd), or 6-byte (cp) value, following the opcode, is used to specify
  - a *code offset*,
  - and possibly a new value for the code segment register CS.

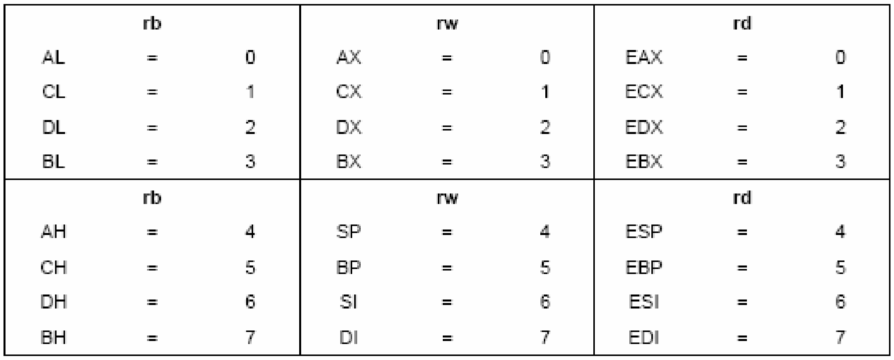
37. Intel Reference Opcode Bytes, Cont.

- `ib, iw, id` - A 1-byte (ib), 2-byte (iw), or 4-byte (id) indicates presence of the **immediate operand** in the instruction.
- Typical order of opcode bytes is
  - **opcode**
  - **Mod R/M** byte (optional)
  - **SIB** scale-indexing byte (optional)
  - **immediate operand**.
- The opcode determines if the operand is a signed value.
- All words and doublewords are given with the low-order byte first (little endian).

38. Intel Reference Opcode Bytes, Cont.

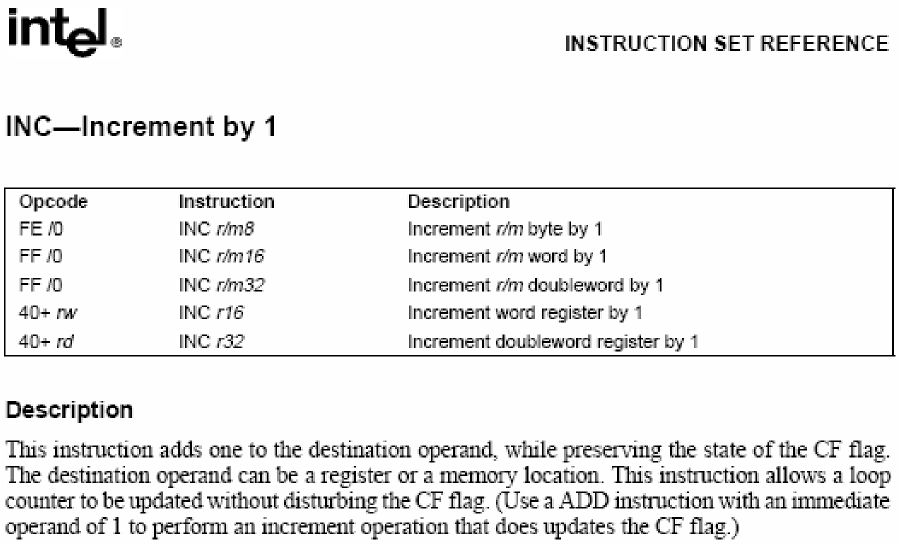
- `+rb, +rw, +rd` - A register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.
- Register Encodings Associated with the **+rb**, **+rw**, and **+rd**:

Table 3-1. Register Encodings Associated with the +rb, +rw, and +rd Nomenclature



rb			rw			rd		
AL	=	0	AX	=	0	EAX	=	0
CL	=	1	CX	=	1	ECX	=	1
DL	=	2	DX	=	2	EDX	=	2
BL	=	3	BX	=	3	EBX	=	3
rb			rw			rd		
AH	=	4	SP	=	4	ESP	=	4
CH	=	5	BP	=	5	EBP	=	5
DH	=	6	SI	=	6	ESI	=	6
BH	=	7	DI	=	7	EDI	=	7

For example,



INSTRUCTION SET REFERENCE

INC—Increment by 1

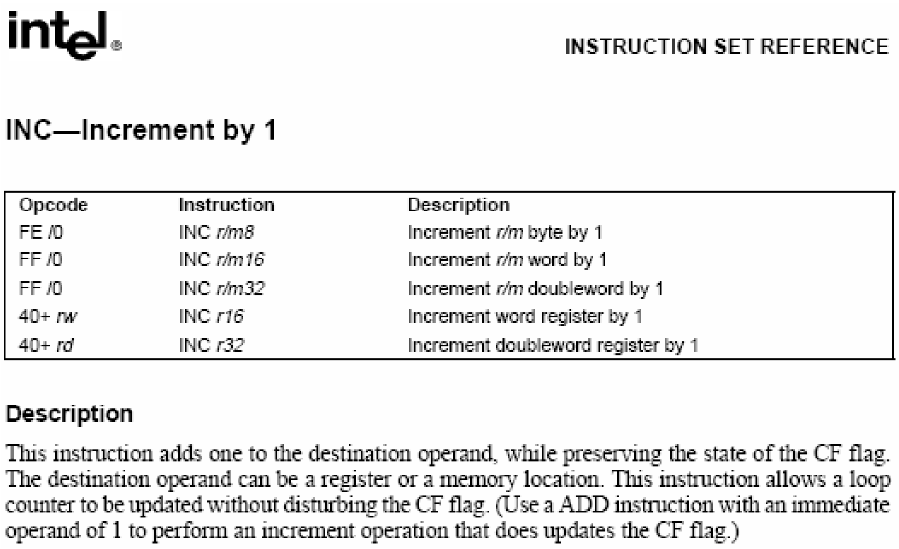
Opcode	Instruction	Description
FE /0	INC *r/m8*	Increment *r/m* byte by 1
FF /0	INC *r/m16*	Increment *r/m* word by 1
FF /0	INC *r/m32*	Increment *r/m* doubleword by 1
40+ *rw*	INC *r16*	Increment word register by 1
40+ *rd*	INC *r32*	Increment doubleword register by 1

Description

This instruction adds one to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

39. Intel Reference Instruction Column

- The **Instruction** column gives the syntax of the instruction statement as it would appear in a 386 Assembly program.
- For example,



INSTRUCTION SET REFERENCE

INC—Increment by 1

Opcode	Instruction	Description
FE /0	INC *r/m8*	Increment *r/m* byte by 1
FF /0	INC *r/m16*	Increment *r/m* word by 1
FF /0	INC *r/m32*	Increment *r/m* doubleword by 1
40+ *rw*	INC *r16*	Increment word register by 1
40+ *rd*	INC *r32*	Increment doubleword register by 1

Description

This instruction adds one to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)