

# x8664-encoding

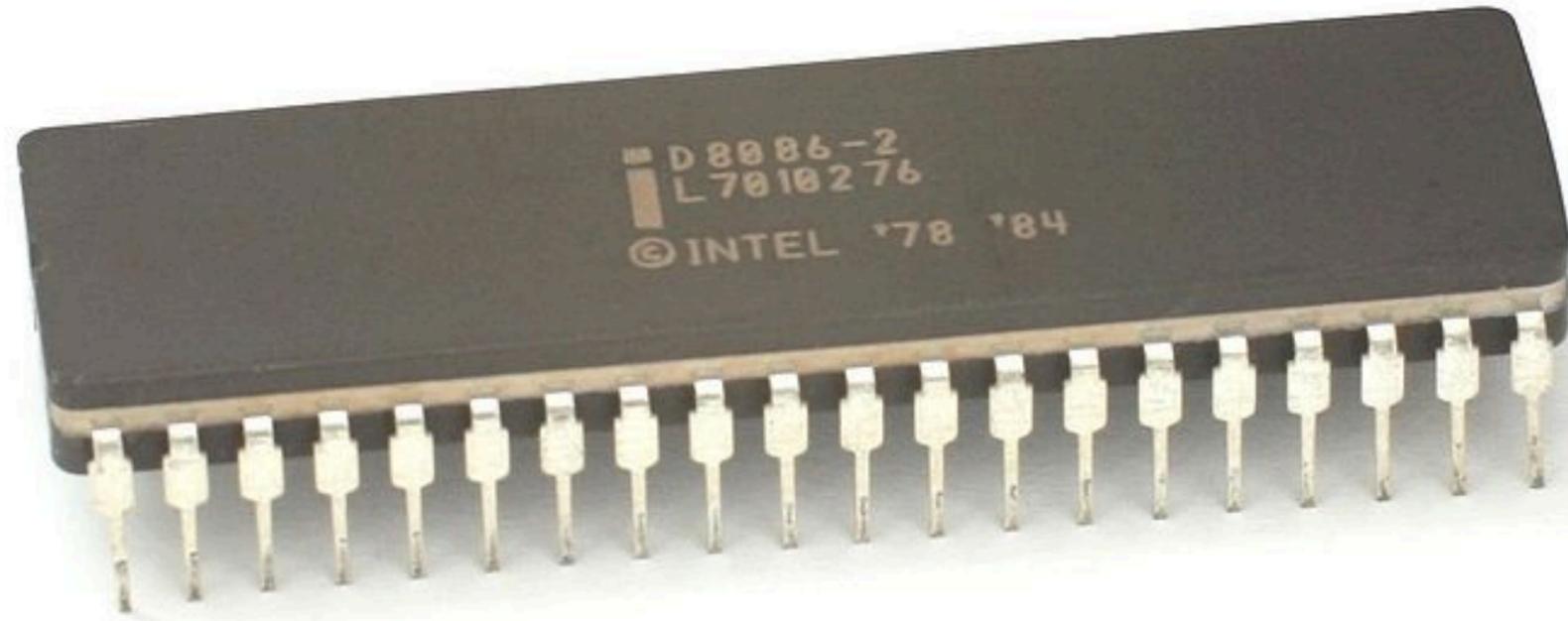
# x86 instruction encoding

x86-64 encoding is quite complicated

reason we don't teach it in CS 2130

mostly complicated because of history

# the 8086



1979 Intel processor

4 general purpose 16-bit registers: AX, BX, CX, DX

4 special 16-bit registers: SI, DI, BP, SP

# 8086 instruction encoding: simple

special cases: 1-byte instructions:

- anything with no arguments

- push ax, push bx, push cx, ... (dedicated opcodes)

- pop ax, ...

# 8086 instruction encoding: two-arg

1-byte opcode

sometimes ModRM byte:

- 2-bit “mod” and

- 3-bit register number (source or dest, depends on opcode) and

- 3-bit “r/m” (register or memory)

“mod” + “r/m” specify one of:

- `%reg` (mod = 11)

- `(%bx/%bp, %si/%di)`

- `(%bx/%si/%di)`

- `offset(%bx/%bp/, %si/%di)` (8- or 16-byte offset)

non-intuitive table

# 16-bit ModRM table

Effective Address	Mod	R/M
[BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] disp 16 <sup>2</sup> [BX]	00	000 001 010 011 100 101 110 111
[BX+SI]+disp8 <sup>3</sup> [BX+DI]+disp8 [BP+SI]+disp8 [BP+DI]+disp8 [SI]+disp8 [DI]+disp8 [BP]+disp8 [BX]+disp8	01	000 001 010 011 100 101 110 111
[BX+SI]+disp16 [BX+DI]+disp16 [BP+SI]+disp16 [BP+DI]+disp16 [SI]+disp16 [DI]+disp16 [BP]+disp16 [BX]+disp16	10	000 001 010 011 100 101 110 111
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AHMM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111

# 16-bit ModRM table

Effective Address	Mod	R/M
[BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] disp16 <sup>2</sup> [BX]	00	000 001 010 011 100 101 110 111
[BX+SI]+disp8 <sup>3</sup> [BX+DI]+disp8 [BP+SI]+disp8 [BP+DI]+disp8 [SI]+disp8 [DI]+disp8 [BP]+disp8 [BX]+disp8	01	000 001 010 011 100 101 110 111
[BX+SI]+disp16 [BX+DI]+disp16 [BP+SI]+disp16 [BP+DI]+disp16 [SI]+disp16 [DI]+disp16 [BP]+disp16 [BX]+disp16	10	000 001 010 011 100 101 110 111
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AHMM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111

e.g. `add %bl, %cl`

(Intel syntax: `add CL, BL`)

Intel manual:

02 /r: `ADD r8 (dest), r/m8`

/r means ModRm byte with reg set to reg#

opcode = 0x02 ModRM byte =

11 (mod) / 001 (reg: %cl) / 011 (r/m: %bl)

or 1100 1011

final encoding: 02 cb

# 8086 evolution

Intel 8086 – 1979, 16-bit registers

Intel (80)386 – 1986, 32-bit registers

AMD K8 – 2003, 64-bit registers

# x86 modes

x86 has multiple *modes*

maintains compatibility

e.g.: modern x86 processor can work like 8086

called “real mode”

different mode for 32-bit/64-bit

same basic encoding; some sizes change

# 32-bit ModRM table

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =		
Effective Address	Mod	R/M
[EAX] [ECX] [EDX] [EBX] [--][--] <sup>1</sup> disp32 <sup>2</sup> [ESI] [EDI]	00	000 001 010 011 100 101 110 111
[EAX]+disp8 <sup>3</sup> [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111

# 32-bit ModRM table

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =		
Effective Address	Mod	R/M
[EAX] [ECX] [EDX] [EBX] [--][--] <sup>1</sup> disp32 <sup>2</sup> [ESI] [EDI]	00	000 001 010 011 100 101 110 111
[EAX]+disp8 <sup>3</sup> [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111

general pattern for 32-bit x86 register numbering:  
 AX = 0, CX, DX, BX, SP, BP, SI, DI = 7

not all registers treated equally to make space for special types of addressing:  
 (base + index \* scale, constant address)

# 32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in ModRM byte

solution: if “r/m” bits = 100 (4, normally ESP), extra “SIB” byte:

- 2 bit scale: 00 is 1, 01 is 2, 10 is 4, 11 is 8

- 3 bit index: index register number

- 3 bit base: base register number

(%baseReg,%indexReg,scale)

# 32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in ModRM byte

solution: if “r/m” bits = 100 (4, normally ESP), extra “SIB” byte:

2 bit *scale*: 00 is 1, 01 is 2, 10 is 4, 11 is 8

3 bit index: index register number

3 bit base: base register number

(%baseReg,%indexReg,*scale*)

# 32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in ModRM byte

solution: if “r/m” bits = 100 (4, normally ESP), extra “SIB” byte:

2 bit scale: 00 is 1, 01 is 2, 10 is 4, 11 is 8

3 bit *index*: index register number

3 bit base: base register number

(%baseReg, %*indexReg*, scale)

# 32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in ModRM byte

solution: if “r/m” bits = 100 (4, normally ESP), extra “SIB” byte:

2 bit scale: 00 is 1, 01 is 2, 10 is 4, 11 is 8

3 bit index: index register number

3 bit *base*: base register number

(*%baseReg*, %indexReg, scale)

# intel manual: SIB table

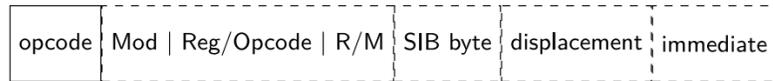
Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX] [ECX] [EDX] [EBX] none [EBP] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 08 10 18 20 28 30 38	01 09 11 19 21 29 31 39	02 0A 12 1A 22 2A 32 3A	03 0B 13 1B 23 2B 33 3B	04 0C 14 1C 24 2C 34 3C	05 0D 15 1D 25 2D 35 3D	06 0E 16 1E 26 2E 36 3E	07 0F 17 1F 27 2F 37 3F
[EAX*2] [ECX*2] [EDX*2] [EBX*2] none [EBP*2] [ESI*2] [EDI*2]	01	000 001 010 011 100 101 110 111	40 48 50 58 60 68 70 78	41 49 51 59 61 69 71 79	42 4A 52 5A 62 6A 72 7A	43 4B 53 5B 63 6B 73 7B	44 4C 54 5C 64 6C 74 7C	45 4D 55 5D 65 6D 75 7D	46 4E 56 5E 66 6E 76 7E	47 4F 57 5F 67 6F 77 7F
[EAX*4] [ECX*4] [EDX*4] [EBX*4] none [EBP*4] [ESI*4] [EDI*4]	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 99 A1 A9 B1 B9	82 8A 92 9A A2 AA B2 BA	83 8B 93 9B A3 AB B3 BB	84 8C 94 9C A4 AC B4 BC	85 8D 95 9D A5 AD B5 BD	86 8E 96 9E A6 AE B6 BE	87 8F 97 9F A7 AF B7 BF
[EAX*8] [ECX*8] [EDX*8] [EBX*8] none [EBP*8] [ESI*8] [EDI*8]	11	000 001 010 011 100 101 110 111	C0 C8 D0 D8 E0 E8 F0 F8	C1 C9 D1 D9 E1 E9 F1 F9	C2 CA D2 DA E2 EA F2 FA	C3 CB D3 DB E3 EB F3 FB	C4 CC D4 DC E4 EC F4 FC	C5 CD D5 DD E5 ED F5 FD	C6 CE D6 DE E6 EE F6 FE	C7 CF D7 DF E7 EF F7 FF

**NOTES:**

1. The [\*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [\*] means disp8 or disp32 + [EBP]. This provides the

# basic 32-bit encoding



dashed: not always present

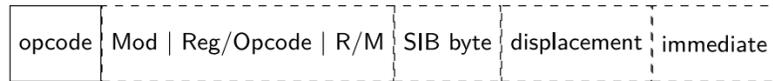
opcodes: 1-3 bytes

some 5-bit opcodes, with 3-bit register field  
(alternate view: 8-bit opcode with fixed register)

sometimes Reg part of ModRM used as add'tl part of opcode  
in Intel manual: `/1` == ModRM byte with reg=1

displacement, immediate: 1, 2, or 4 bytes  
or, rarely, 8 bytes

# basic 32-bit encoding



dashed: not always present

opcodes: 1-3 bytes

*some 5-bit opcodes, with 3-bit register field*

(alternate view: 8-bit opcode with fixed register)

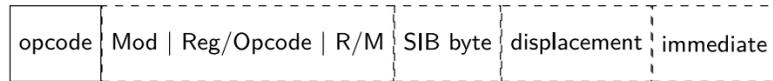
sometimes Reg part of ModRM used as add'tl part of opcode

in Intel manual: `/1 == ModRM byte with reg=1`

displacement, immediate: 1, 2, or 4 bytes

or, rarely, 8 bytes

# basic 32-bit encoding



dashed: not always present

opcodes: 1-3 bytes

some 5-bit opcodes, with 3-bit register field  
(alternate view: 8-bit opcode with fixed register)

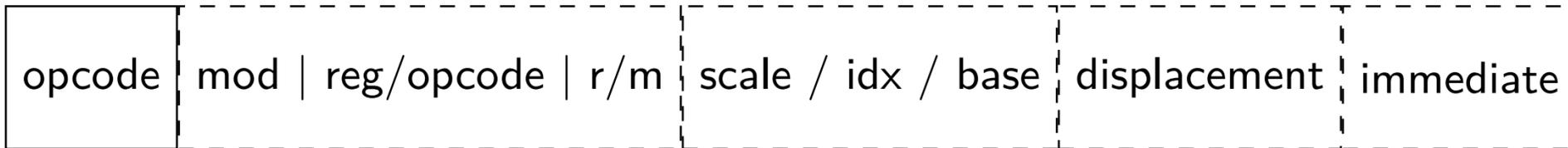
*sometimes Reg part of ModRM used as add'l part of opcode*

in Intel manual: `/1` == ModRM byte with reg=1

displacement, immediate: 1, 2, or 4 bytes

or, rarely, 8 bytes

# exercise 1



r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =		
Effective Address	Mod	R/M
[EAX] [ECX] [EDX] [EBX] [--][--] <sup>1</sup> disp32 <sup>2</sup> [ESI] [EDI]	00	000 001 010 011 100 101 110 111
[EAX]+disp8 <sup>3</sup> [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM/XMM1 EDX/DX/DI/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111

## BTS—Bit Test and Set

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF AB /r	BTS <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag and set.
OF AB /r	BTS <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF AB /r	BTS <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag and set.
OF BA /5 <i>ib</i>	BTS <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and set.
OF BA /5 <i>ib</i>	BTS <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF BA /5 <i>ib</i>	BTS <i>r/m64, imm8</i>	MI	Valid	N.E.	Store selected bit in CF flag and set.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

exercise: encode `btsl $7, 4(%rax)`  
 (Intel syntax: `BTS DWORD PTR [RAX+4], 7`)

# exercise 1 solution

```
btsl $7, 4(%rax) / BTS DWORD PTR [RAX + 4], 7
```

from ISA reference entry:

0F BA /5 ib - BTS r/m32, imm8 - MI

0F BA + ModRM byte with reg=5 + immediate byte

MI: operand 1 in ModRM byte r/m field; operand 2 in immediate byte

from table:

[EAX]+disp8: mod 01, R/M 000; [EAX]+disp32: mod 10 R/M 000;

0F BA 0b01\_101\_000 04 07 = 0F BA 68 04 07

0F BA 0b10\_101\_000 04 00 00 00 07 = 0F BA A8 04 00 00 00 07

# what about 64-bit?

adds 8 more registers – more bits for reg #?

didn't change encoding for existing instructions, so...

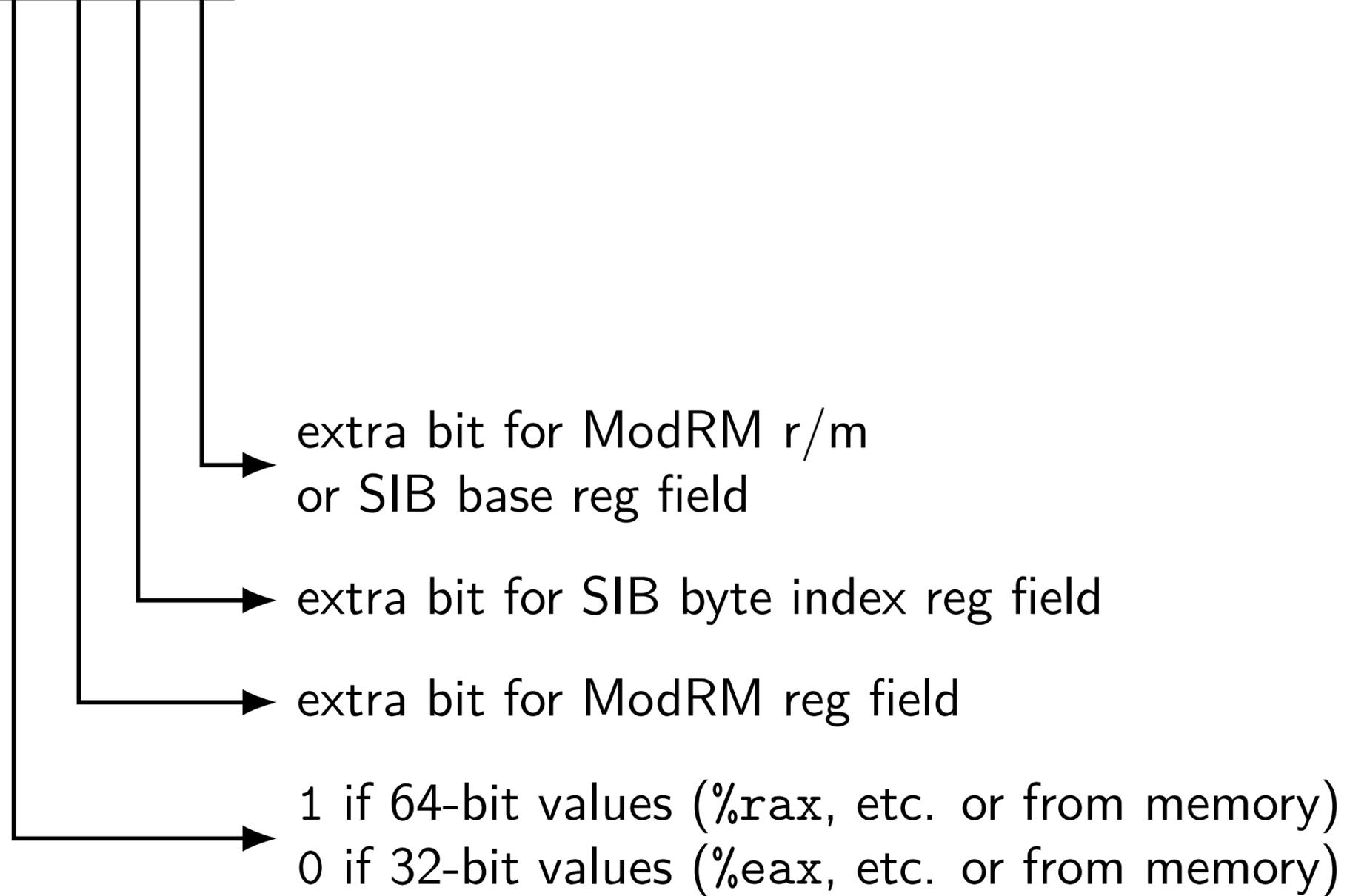
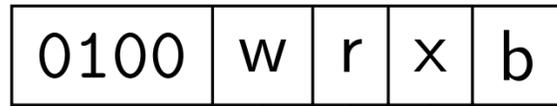
*instruction prefix* “REX”

32-bit x86 already had many prefixes

also selects 64-bit version of instruction

# REX prefix

REX prefix byte



# 64-bit REX exercise (1)

add %eax, %ecx (Intel: ADD ecx, eax)

01 (opcode) c1 (MOD: 11 / REG: 000 (eax) / R/M: 001 (ecx))

exercise 2a: add %eax, %r10d (Intel: ADD r10d, eax) = ???

REX prefix + 01 + MOD-REG-R/M byte

REX prefix:

0100

w (is 64-bit values?)

r (extra bit for Reg field)

s (extra bit for SIB index reg)

b (extra bit for R/M or SIB base field)

# 64-bit REX exercise (2)

add %eax, %ecx (Intel: ADD ecx, eax)

01 (opcode) c1 (MOD: 11 / REG: 000 (eax) / R/M: 001 (ecx))

exercise 2b: add %rax, %rcx (Intel: ADD rcx, rax) = ???

REX prefix + 01 + MOD-REG-R/M byte

REX prefix:

0100

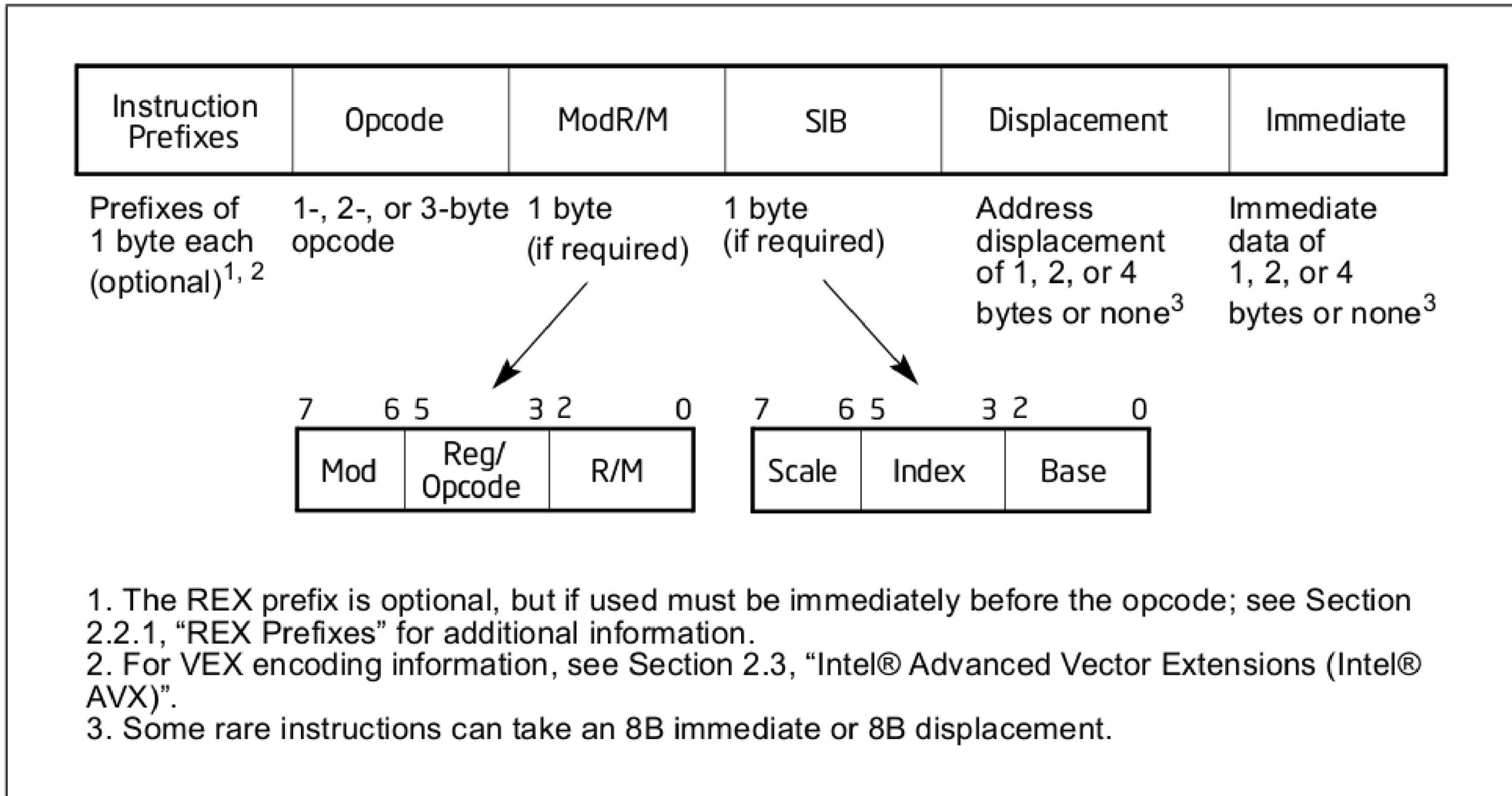
w (is 64-bit values?)

r (extra bit for Reg field)

s (extra bit for SIB index reg)

b (extra bit for R/M or SIB base field)

# overall encoding



**Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format**

# instruction prefixes

REX (64-bit and/or extra register bits)

VEX (SSE/AVX instructions; other new instrs.)

operand/address-size change (64/32 to 16 or vice-versa)

LOCK — synchronization between processors

REPNE/REPZ/REP/REPE/REPZ — turns instruction into loop

segment overrides

# instruction prefixes

REX (64-bit and/or extra register bits)

VEX (SSE/AVX instructions; other new instrs.)

operand/address-size change (64/32 to 16 or vice-versa)

LOCK – synchronization between processors

*REPNE/REPZ/REP/REPE/REPZ – turns instruction into loop*

segment overrides

# x86 encoding example (1)

pushq %rax encoded as 50

5-bit opcode 01010 plus 3-bit register number 000

50+rd	PUSH r64
-------	----------

pushq %r13 encoded as 41 55

41: REX prefix 0010 (constant), w:0, r:0, s:0, b:1

w = 0 because push is never 32-bit in 64-bit mode

55: 5-bit opcode 01010; 3-bit reg # 101

4-bit reg # 1101 = 13

# x86 encoding example (2)

addl 0x12345678(%rax,%rbx,2), %ecx

03: opcode – add r/m32 into r32

03 /r	ADD r32, r/m32	RM
-------	----------------	----

RM	ModRM:reg (r, w)	ModRM:r/m (r)
----	------------------	---------------

8c: ModRM: mod = 10; reg = 001, r/m: 100

reg = 001 = %ecx (table)

SIB byte + 32-bit displacement (table)

58: SIB: scale = 01, index = 011, base = 000

index 011 = %rbx; base 000 = %rax;

78 56 32 12: 32-bit constant 0x12345678

# x86 encoding example (3)

addq 0x12345678(%r10,%r11,2), %rax

4b: REX prefix 0100+w:1, r:0, s:1, b:1

03: opcode — add r/m64 to r64 (with REX.w)

84: ModRM: mod = 10; reg = 000, r/m: 100

reg = 0000 = %rax

SIB byte + 32-bit displacement (table)

5a: SIB: scale = 01, index = 011, base = 010

with REX: index = 1011 (11), base = 1010 (10)

78 56 32 12: 32-bit constant 0x12345678

# x86 encoding example (3)

addq 0x12345678(%r10,%r11,2), %rax

4b: REX prefix 0100+w:1, r:0, s:1, b:1

03: opcode — add r/m64 to r64 (with *REX.w*)

84: ModRM: mod = 10; reg = 000, r/m: 100

reg = 0000 = %rax

SIB byte + 32-bit displacement (table)

5a: SIB: scale = 01, index = 011, base = 010

with REX: index = 1011 (11), base = 1010 (10)

78 56 32 12: 32-bit constant 0x12345678

# x86 encoding example (3)

addq 0x12345678(%r10,%r11,2), %rax

4b: REX prefix 0100+w:1, r:0, s:1, b:1

03: opcode — add r/m64 to r64 (with REX.w)

84: ModRM: mod = 10; reg = 000, r/m: 100

reg = 0000 = %rax

SIB byte + 32-bit displacement (table)

5a: SIB: scale = 01, index = 011, base = 010

with REX: index = 1011 (11), base = 1010 (10)

78 56 32 12: 32-bit constant 0x12345678

# x86 encoding example (3)

addq 0x12345678(%r10,%r11,2), %rax

4b: REX prefix 0100+w:1, r:0, s:1, b:1

03: opcode — add r/m64 to r64 (with REX.w)

84: ModRM: mod = 10; reg = 000, r/m: 100

reg = 0000 = %rax

SIB byte + 32-bit displacement (table)

5a: SIB: scale = 01, index = 011, base = 010

with REX: index = 1011 (11), base = 1010 (10)

78 56 32 12: 32-bit constant 0x12345678

# x86 encoding example (3)

addq 0x12345678(%r10,%r11,2), %rax

4b: REX prefix 0100+w:1, r:0, s:1, b:1

03: opcode — add r/m64 to r64 (with REX.w)

84: ModRM: mod = 10; reg = 000, r/m: 100

reg = 0000 = %rax

SIB byte + 32-bit displacement (table)

5a: SIB: scale = 01, index = 011, base = 010

with REX: index = 1011 (11), base = 1010 (10)

78 56 32 12: 32-bit constant 0x12345678

# x86 encoding example (4)

`movq %fs:0x10,%r13`

64: FS segment override

48: REX: w: 1 (64-bit), r: 1, s: 0, b: 0

8b: opcode for MOV memory to register

2c: ModRM: mod = 00, reg = 101, r/m: 100

with REX: reg = 1101 [%r13]; r/m = 100 (SIB follows)

25: SIB: scale = 00; index = (0)100; base = (0)101

no register/no register in table

10 00 00 00: 4-byte constant 0x10

# x86 encoding example (4)

`movq %fs:0x10,%r13`

64: FS segment override

48: REX: w: 1 (64-bit), r: 1, s: 0, b: 0

8b: opcode for MOV memory to register

2c: ModRM: mod = 00, reg = 101, r/m: 100

with REX: reg = 1101 [%r13]; r/m = 100 (SIB follows)

25: SIB: scale = 00; index = (0)100; base = (0)101

no register/no register in table

10 00 00 00: 4-byte constant 0x10

# x86 encoding example (4)

`movq %fs:0x10,%r13`

64: FS segment override

48: REX: w: 1 (64-bit), r: 1, s: 0, b: 0

8b: opcode for MOV memory to register

2c: ModRM: mod = 00, reg = 101, r/m: 100

with REX: reg = 1101 [%r13]; r/m = 100 (SIB follows)

25: SIB: scale = 00; index = (0)100; base = (0)101

no register/no register in table

10 00 00 00: 4-byte constant 0x10

# x86 encoding example (4)

`movq %fs:0x10,%r13`

64: FS segment override

48: REX: w: 1 (64-bit), r: 1, s: 0, b: 0

8b: opcode for MOV memory to register

2c: ModRM: mod = 00, reg = 101, r/m: 100

with REX: reg = 1101 [%r13]; r/m = 100 (SIB follows)

25: SIB: scale = 00; index = (0)100; base = (0)101

no register/no register in table

10 00 00 00: 4-byte constant 0x10

# x86-64 impossibilities

*illegal*: `movq 0x12345678ab(%rax), %rax`

maximum 32-bit displacement

`movq 0x12345678ab, %rax` okay

extra `mov` opcode for `%rax` only

*illegal*: `movq $0x12345678ab, %rbx`

maximum 32-bit constant

`movq $0x12345678ab, %rax` okay

*illegal*: `pushl %eax`

no 32-bit push/pop in 64-bit mode

but 16-bit allowed (operand size prefix byte 66)

*illegal*: `movq (%rax, %rsp), %rax`

cannot use `%rsp` as index register

`movq (%rsp, %rax), %rax` okay

# position dependence

two ways of encoding addresses in x86-64 assembly:

address in little endian (typically 32-bits – limit on executable size)

(usually 32-bit) difference between address and %rip (next instruction address)

---

## assembly

## encoding

`movq label, %al`

`8a 04 25 label addr`

or Intel: `mov AL, [label]`

`jmp *label`

`ff 24 25 label addr`

or Intel: `jmp [label]`

`mov label(%rip), %al`

`8a 05 %rip - label addr`

or Intel NASM: `mov AL, [REL label]`

`jmp *label(%rip), %al`

`e9 %rip - label addr`

or Intel NASM: `jmp [REL label]`

# position-independence: which to use?

suppose we're inserting "evil" code  
at changing addresses in executable's memory

which of the following do we want absolute encoding for?  
(i.e. which would absolute encoding be easier than relative)

- A. address of a jump from evil code to function at fixed loc in executable
- B. address of a jump in a loop in the "evil" code
- C. address of a string in the "evil" code
- D. address of a string in the executable