Exploiting IA-32 redundancy

Charles Paulet

Instructions encoding (IA-32e)

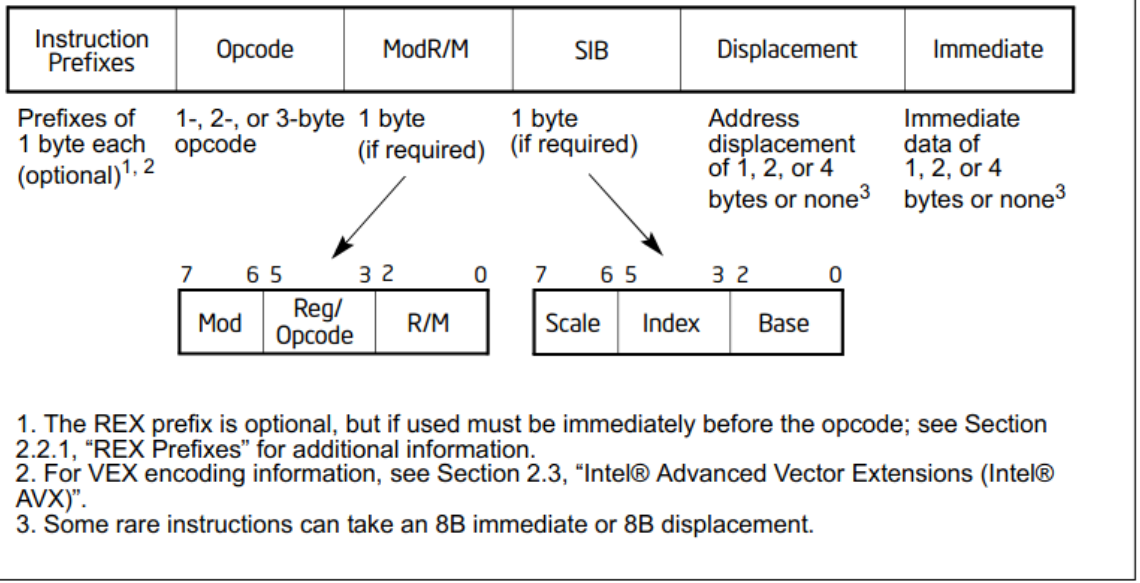
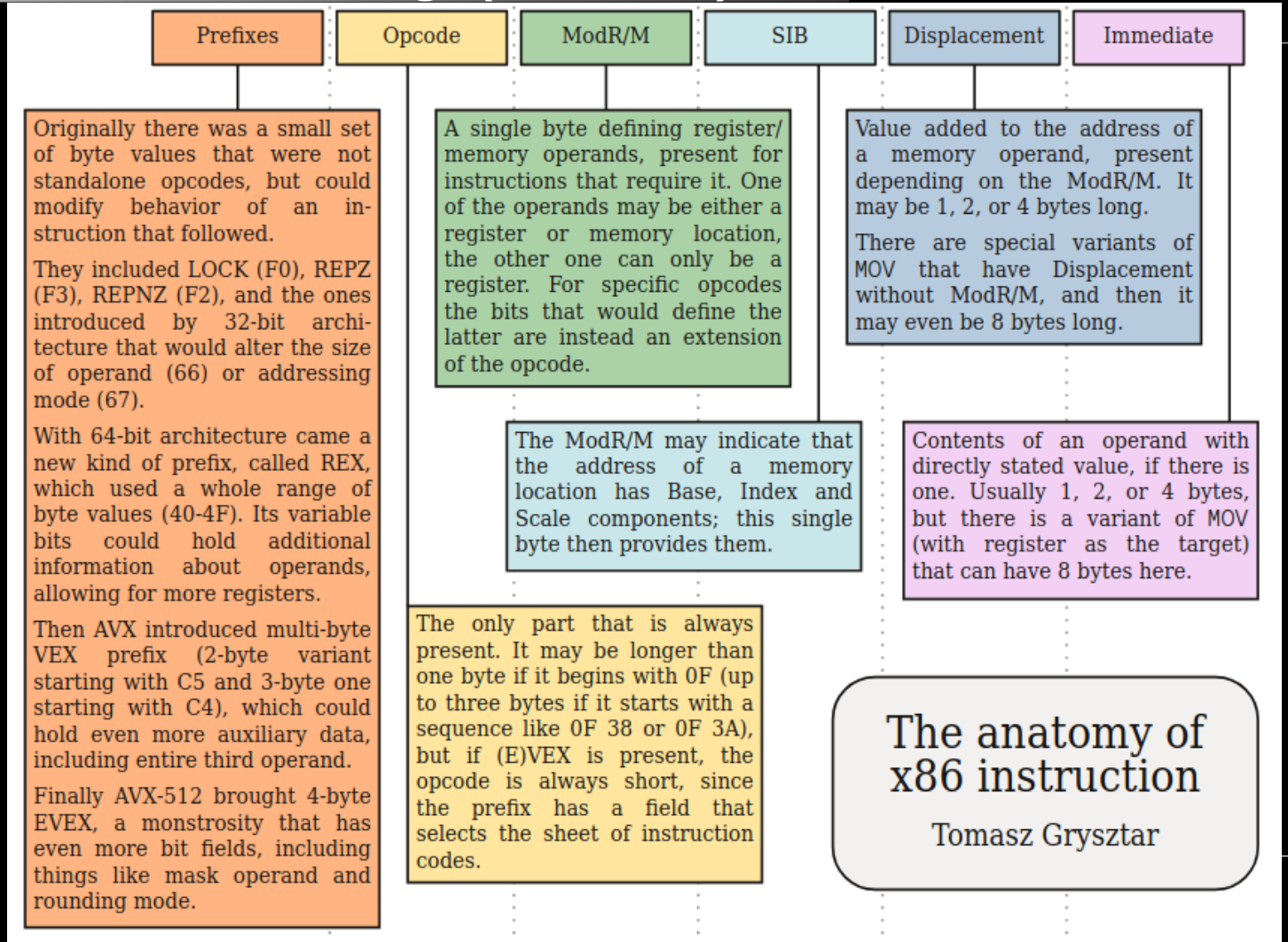
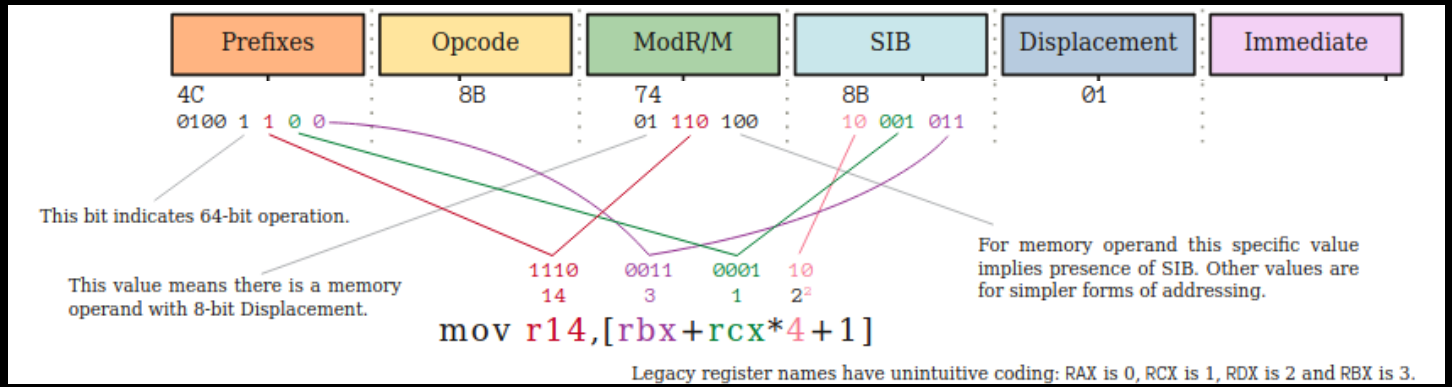


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Instructions encoding (IA-32e)



Example (format)




```
Instruction details:
mnemonic:     mov r14, qword ptr [rbx + rcx*4 + 1]
bytes:        0x4c 0x8b 0x74 0x8b 0x01
prefix:       0x00 0x00 0x00 0x00
opcode:       0x8b 0x00 0x00 0x00
rex:          0x4c
addr size:    0x08
modrm:        0x74 (mod: 0b01) (reg: 0b110) (rm: 0b100)
modrm offset: 0x02
disp:         0x01
sib:          0x8b (scale: 0b10) (index: 0b001) (base: 0b011)
```

Opcodes encoding

- 1, 2 or 3-byte opcodes
- Manual has 3 opcode tables
- Each cell using abbreviations for ModR/M and SIB bytes (for specific addressing modes)

Example (ADD Gv, Ev)

Table A-2. One-byte Opcode Map: (00H — F7H)

	0	1	2	3	4	5
0	ADD					
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX

```
Instruction details:
mnemonic:     add rax, qword ptr [rip + 1]
bytes:        0x48 0x03 0x05 0x01 0x00 0x00 0x00
prefix:       0x00 0x00 0x00 0x00
opcode:       0x03 0x00 0x00 0x00
rex:          0x48
addr size:    0x08
modrm:        0x05 (mod: 0b00) (reg: 0b000) (rm: 0b101)
modrm offset: 0x02
disp:         0x01
sib:          0x00 (base: 0b000) (index: 0b000) (scale: 0b00)
```

Operands encoding

Operand encoding = addressing method + operand type

Manual abbreviations revealing possible mutations:

Abbr	Description
E	A ModR/M byte follows the opcode and specifies the operand
G	The ModR/M reg part selects a general register
I	Immediate data
b	Byte
v	Word, doubleword or quadword depending on operand-size attribute

Register operands and the ModR/M byte

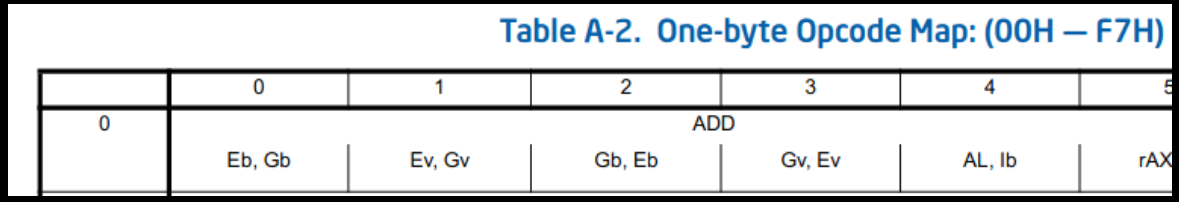


Table A-2. One-byte Opcode Map: (00H — F7H)

	0	1	2	3	4	5
0	ADD					
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX

Instruction	bytes	opcode	ModR/M
add eax, ebx	01 d8	01	d8 (mod: 0b11) (reg: 0b011) (rm: 0b000)
add ebx, eax	03 d8	03	d8 (mod: 0b11) (reg: 0b011) (rm: 0b000)
add eax, ebx	03 c3	03	c3 (mod: 0b11) (reg: 0b000) (rm: 0b011)

Register operands and the SIB byte

When SIB.scale is 0b00 :

Instruction	bytes	SIB
mov rax, qword ptr [rbx + rcx]	48 8b 04 0b	0x0b (index: 0b001, rcx) (base: 0b011, rbx)
mov rax, qword ptr [rcx + rbx]	49 8b 04 19	0x19 (index: 0b011, rbx) (base: 0b001, rcx)

Instruction	bytes	SIB
mov byte ptr [eax + ebx], 5	67 c6 04 18 05	0x18 (index: 0b011, ebx) (base: 0b000, eax)
mov byte ptr [ebx + eax], 5	67 c6 04 03 05	0x03 (index: 0b000, eax) (base: 0b011, ebx)

Duplicate opcode extensions

An opcode extension is used when there is no second register operand.

Group 0xf6

Instruction	bytes	ModR/M reg
test bl, 0x10	f6 c3 10	0b000
test bl, 0x10	f6 cb 10	0b001

Group 0xf7

Instruction	bytes	ModR/M reg
test ebx, 0xaabbccdd	f7 c3 dd cc bb aa	0b000
test ebx, 0xaabbccdd	f7 cb dd cc bb aa	0b001

Duplicate opcodes in 32-bit mode

Group 1 have duplicated opcodes for 0x80 and 0x82 on x86 when working with imm8 (or Ib)

Instruction	bytes
add byte ptr [eax], 0x10	80 00 10
add byte ptr [eax], 0x10	82 00 10

Also works for ADD, OR, ADC, SBB, AND, SUB, XOR, CMP depending on the ModR/M reg

Duplicate opcodes in 32/64-bit mode

Group 1 have duplicated opcodes for 0x81 and 0x83 when working with imm8 (or Ib)

Instruction	bytes
add eax, 0x10	83 c0 10
add eax, 0x10	81 c0 10 00 00 00
add rax, 0x10	48 83 c0 10
add rax, 0x10	48 81 c0 10 00 00 00

Duplicates for the acc register

The acc register can be mutated using the x4 and x5 opcodes

Instruction	bytes
add al, 0x10	04 10
add al, 0x10	80 c0 10
add eax, 0x10	05 10 00 00 00
add eax, 0x10	81 c0 10 00 00 00
add eax, 0x10	83 c0 10
add eax, 0x10	48 05 10 00 00 00 00
add rax, 0x10	48 81 c0 10 00 00 00
add rax, 0x10	48 83 c0 10

And for ADD, OR, ADC, SBB, AND, SUB, XOR, CMP, depending on the ModR/M reg

Duplicates using zero displacements

Instruction	bytes	ModR/M mod
add dword ptr [eax], eax	67 01 00	0b00
add dword ptr [eax + 00], eax	67 01 40 00	0b01
add dword ptr [eax + 00000000], eax	67 01 80 00 00 00 00	0b10

SIB scale/index corner case

When SIB.index is 0b100, SIB.scale is useless

Instruction	bytes	SIB.scale
mov byte ptr [0xaabbccdd], 0xff	c6 04 25 dd cc bb aa ff	0b00
mov byte ptr [0xaabbccdd], 0xff	c6 04 65 dd cc bb aa ff	0b01
mov byte ptr [0xaabbccdd], 0xff	c6 04 a5 dd cc bb aa ff	0b10
mov byte ptr [0xaabbccdd], 0xff	c6 04 e5 dd cc bb aa ff	0b11

Instruction	bytes	SIB.scale
mov byte ptr [ebp + 0x56], 0xff	c6 44 25 56 ff	0b00
mov byte ptr [ebp + 0x56], 0xff	c6 44 65 56 ff	0b01
mov byte ptr [ebp + 0x56], 0xff	c6 44 a5 56 ff	0b10
mov byte ptr [ebp + 0x56], 0xff	c6 44 e5 56 ff	0b11

Legacy prefixes

In 32-bit mode, we can omit some legacy prefixes:

Instruction	bytes
add qword ptr [eax], eax	67 01 00
add qword ptr [eax], eax	01 00

Some instructions might accept one or more prefixes:

Instruction	bytes
nop	90
nop	66 90
nop	66 67 90
nop	66 66 67 90

Logic transformations

Like zeroing registers

Instruction	bytes	ModR/M
xor bx, bx	66 31 db	0xdb (mod: 0b11) (reg: 0b011) (rm: 0b011)
xor ebx, ebx	31 db	0xdb (mod: 0b11) (reg: 0b011) (rm: 0b011)
sub bx, bx	66 29 db	0xdb (mod: 0b11) (reg: 0b011) (rm: 0b011)
sub ebx, ebx	29 db	0xdb (mod: 0b11) (reg: 0b011) (rm: 0b011)
mov ebx, 0	bb 00 00 00 00	0x00 (mod: 0b00) (reg: 0b000) (rm: 0b000)
mov rbx, 0	48 c7 c3 00 00 00 00	0xc3 (mod: 0b11) (reg: 0b000) (rm: 0b011)

Going further

- Signatures/pattern-matching bypasses?
- Cmake / C&C module?
- Steganography?
- LLVM passes?
- post-compilation / LIEF frontend?

Tool 1: transasm

- implementing the techniques discussed
- based on capstone
- **there are still mutations waiting to be found** and to be implemented:
  - explore extensions like VEX, EVEX
  - SIB's S and I arithmetics
- we might use it as a mutation engine
- More :
  - reconstruct offsets (think jmps)
  - 1 insn → n insns

```
test_prime_x86_64_program (transasm.tests.func.te
>> 33 d2 -> xor edx, edx
<< 31 c9 -> xor ecx, ecx
>> 33 c9 -> xor ecx, ecx
<< 83 f9 02 -> cmp ecx, 2
>> 81 f9 02 00 00 00 -> cmp ecx, 2
<< 89 c8 -> mov eax, ecx
>> 8b c1 -> mov eax, ecx
<< 31 db -> xor ebx, ebx
>> 33 db -> xor ebx, ebx
<< 83 e9 01 -> sub ecx, 1
>> 81 e9 01 00 00 00 -> sub ecx, 1
<< 83 f9 01 -> cmp ecx, 1
>> 81 f9 01 00 00 00 -> cmp ecx, 1
<< 01 ca -> add edx, ecx
>> 03 d1 -> add edx, ecx
<< 48 89 d6 -> mov rsi, rdx
>> 48 8b f2 -> mov rsi, rdx
<< 48 31 ff -> xor rdi, rdi
>> 48 33 ff -> xor rdi, rdi
```

Tool 2: asmshell

ASM REPL based on capstone, keystone, unicorn

```
> add eax, ebx
Code:
0000000000000002: 01 d8                    | add eax, ebx
Registers:
rax:    0000000000000010  r8:  0000000000000000  cs: 0000000000000000  cr0: 0000000000000011 [ PE ET ]
rbx:    0000000000000000  r9:  0000000000000000  ss: 0000000000000000  cr1: 0000000000000000
rcx:    0000000000000000  r
rdx:    0000000000000000  r
rdi:    0000000000000000  r
rsi:    0000000000000000  r
rbp:    0000000000000000  r
rsp:    0000000000200000  r
rip:    0000000000000002
eflags: 0000000000000002 [
Stack:
0000000000200000: 00 00 00
0000000000200010: 00 00 00
0000000000200020: 00 00 00
0000000000200030: 00 00 00
```

```
> .wb 0 01d8 ; .dec 0
Instruction details:
mnemonic:      add eax, ebx
bytes:         0x01 0xd8
prefix:        0x00 0x00 0x00 0x00
opcode:        0x01 0x00 0x00 0x00
rex:           0x00
addr size:     0x08
modrm:         0xd8 (mod: 0b11) (reg: 0b011) (rm: 0b000)
modrm offset:  0x01
disp:          0x00
sib:           0x00 (base: 0b000) (index: 0b000) (scale: 0b00)
```

Key takeaways

- It’s easy to get started with capstone, keystone, unicorn
- x86 encoding knowledge can be reused elsewhere (minimal disassembler)