Decoding Raw Hex

This is where we get the complex in Complex Instruction Set Computer.

Required Reading: Section 3.6.4 Jump Instruction Encodings

Helpful Sources

- http://ref.x86asm.net/coder64.html#x48
- https://www.systutorials.com/72643/beginners-guide-x86-64-instruction-encoding/

Sidebar: Intel register numbering and naming

▸ The numbering of intel registers does NOT follow ABCD order. The letters come from their original names. The first 4 can be addressed down to the byte level (the high and low bytes of the lower 16 bits). Registers 8-15 have no names and are addressed by number.

Name	#
Accumulate (rax, eax, ax, ah, al)	0
Counter (rcx, ecx, cx, ch, cl)	1
Data (rdx, edx, dx, dh, dl)	2
Base (rbx, ebx, bx, bh, bl)	3
Stack pointer (rsp , esp, sp)	4
Base pointer (rbp , ebp, bp)	5
Source index (rsi ,esi, si)	6
Destination index(rdi ,edi, di)	7

Register Numbering

▸ Do not think that because register A is coded as register zero that register B is coded as register one. It is register three, not one. This is the price we pay for basing a 64 bit architecture on a 32 bit extension to a16 bit foundation that itself had strong ties to a similar 8 bit predecessor chip.

▸ To access the higher numbered registers, we make changes in the prefix bits (explained later).

Archeology – intel x86 chips

- 8086 has 8 registers all are 16 bit integer/pointer (1978)
- 80286 adds memory management (Systems 2 topic)(1982)
- 80386 expands the registers to 32 bits, but we still only have 8 of them (1985)
- Itanium: a clean 64 bit design no one wanted to buy (2001)
- AMD64 architecture (2003)
  - 16 integer/pointer registers
  - 64 bit
  - Powers on in real mode fully backward compatible with 8086

Things changed with 64 bits

- 8 registers
- 3 bits to encode a register number
- One byte (8 bits) gives enough bits to specify 2 registers and a 2-bit mode (3+3+2 = 8)

- 16 registers
- 4 bits to encode a register number
- Runs IA32 instructions unchanged
- We need 2 more bits so that we can deal with a pair of 4-bit register numbers!

IA32 and earlier

X86-64

Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
    pushq    %rbx
    movq     %rdx, %rbx
    call     plus
    movq     %rax, (%rbx)
    popq     %rbx
    ret
```

Obtain with command

```
gcc –Og –S sum.c
```

Produces file `sum.s`

Warning: Will get very different results on different machines (Linux, Mac OS-X, …) due to different versions of gcc and different compiler settings.

Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:
  400595:   53                  push   %rbx
  400596:   48 89 d3            mov    %rdx,%rbx
  400599:   e8 f2 ff ff ff      callq  400590 <plus>
  40059e:   48 89 03            mov    %rax,(%rbx)
  4005a1:   5b                  pop    %rbx
  4005a2:   c3                  retq
```

▶ Disassembler

objdump –d sum
- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either `a.out` (complete executable) or `.o` file

Object Code

```
00000000000400595 <sumstore>:
  400595:  53                    push   %rbx
  400596:  48 89 d3              mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff   callq  400590 <plus>
  40059e:  48 89 03              mov    %rax,(%rbx)
  4005a1:  5b                    pop    %rbx
  4005a2:  c3                    retq
```

Let's do a deep dive on that highlighted mov

movq %rax, (rbx)

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:   48 89 03
```

- C Code
  - Store value **t** where designated by **dest**
- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:

    t: Register **%rax**

    dest: Register **%rbx**

    *dest: Memory **M[%rbx]**
- Object Code
  - 3-byte instruction
  - Stored at address **0x40059e**

Example expanded 1 of 2

movq %rax, (%rbx) [AT&T syntax as opposed to intel syntax] is hex <u>48</u> 89 03

48 – is a prefix denoting 64 bit operations and provides modifiers to the operands. This is the REX prefix. The bits are 0100 WRXB.

First nibble: The first nibble is 0100b or 4. It is always a 4 for a REX prefix

The second nibble is 1000b or 8 and the bits are named **WRXB**

- **W** bit set, W bit set promotes a 32 bit operation to 64 bits. This is what turns MOV into MOVQ. It turns the IA32 instruction into x64.

- **R** bit is zero because RAX is a low numbered register (0). If R was 1 the source register would be R8

- **X** is zero. X modifies the SIB index field, not used in this instruction. (needed for expanded register

- **B** is zero because RBX is a low numbered register (3). If B was 1 the destination register would be R11

Example expanded 2 of 2

movq %rax, (%rbx) [AT&T syntax as opposed to intel syntax]

hex **48 89 03**

89 – means MOV. This is the opcode. It means move register to register or memory. This is an IA32 opcode.

03 – (ModR/M byte) encodes mode and 2 registers: The two mode bits are zero (00). This mode is register to memory. RAX encodes to (000) and RBX encodes to 011. Altogether we have 00 followed by 000 and 011 gives us 0000 0011 or hex 03. This is an IA32 addressing mode byte.

Others ops are simpler
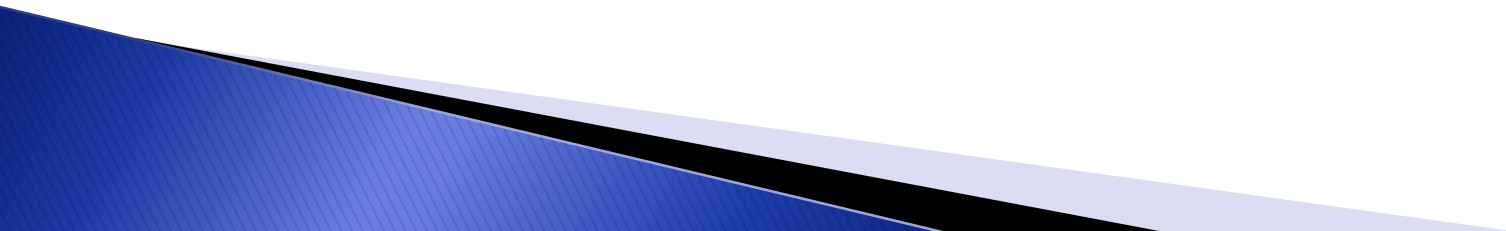
400595: 53   push   %rbx

53: Opcodes 50-57 are PUSH (just add the register number). Register is 3 and we saw before that is RBX

This IA32 operation when done in 64 bit mode pushes the 64 bit register. Since RBX isn't a higher numbered register, there is no need for a fourth bit for the register number. So we don't need a prefix. You can't push a 32 bit register in 64 bit mode, because the opcode means something different in 64 bit mode than it does in 32 bit mode.

Note that this means that pushing RBX is cheaper than pushing R12-15 because it takes one less byte *in the instruction stream.* (All low number registers are cheaper to push or pop)

400596: 48 89 d3   mov   %rdx,%rbx

- 48 – REX prefix we saw before
- 89 – MOVQ as before
- d3 – encodes the mode (11) and registers, RDX (010) and RBX (011). This gives 11 010 011 (mode,reg,reg). Breaking the bits into groups of four we get 1101 0011 or d3 in hex. This instruction has a different mode (register to register) rather than register to memory.

Again, the basics are IA32, but extended with the REX prefix

Some operations look more complex than they are…
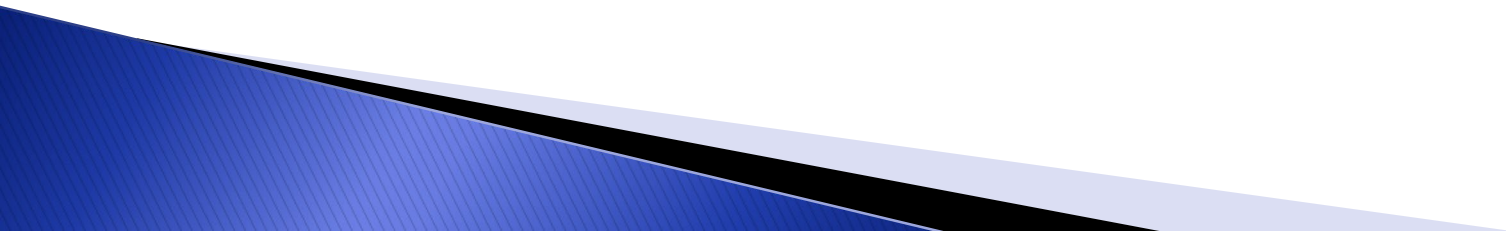
400599: e8 f2 ff ff ff callq 400590 <plus>

- e8 – is CALL (relative).
- f2 ff ff ff – is the offset. *Little endian byte order !* This is 0xfffffff2, which is a small negative number in 2's complement notation (–14). Go up 14 bytes to find the jump target. That is 9 bytes worth of instructions in this function and then 5 more above that. Note that this code started at 400595 and our jump target is 400590, or five bytes above where we started.
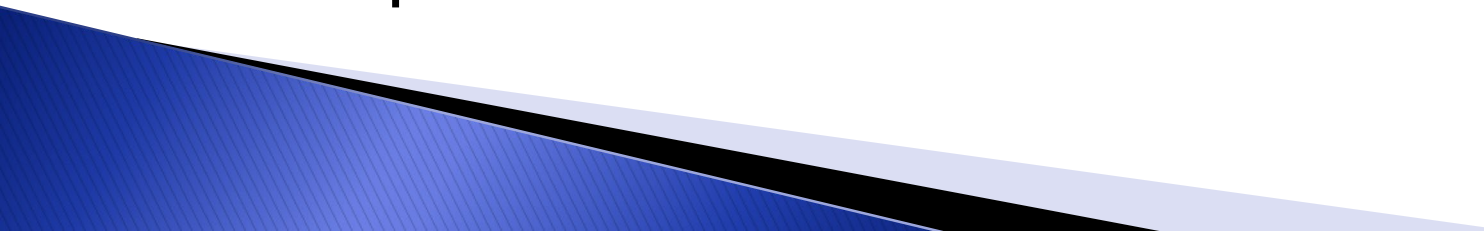- We didn't need a full 8-byte address as the jump target

We did the other move in detail.
Pop for low numbered registers is simple

4005a1: 5b   pop   %rbx

- 5b: POP is opcodes 58+Register. 5B−58 = 3, once again RBX

- Pushing or popping a high-numbered register (R8-R15) requires a REX prefix, which is one more byte in the instruction stream.
- In 64 bit mode, this pop instruction pops a 64 bit register.

Final Note

- This is a ton of work!
- This is why we have assemblers – so we don't have to look up opcodes and modes and prefix bytes out of tables and websites and try to get it right doing it by hand!
- This is why we have disassemblers – so we don't have to turn raw bytes into prefixes, opcodes, modes and registers!
- Maxed out these instructions can be many bytes, but the chip won't allow more than 15 bytes in one instruction.
- This is the "complex" part of "Complex Instruction Set Computer!"

Hidden Lesson

- Refactoring with backwards compatibility is hard!
  - We need the REX prefix in front of an IA32 operation to transform it to x86-64 and to encode the new registers *without altering the underlying IA32 instruction*
  - We can't code the new registers directly as before, there are not enough bits in the byte of the old encoding
    - Mode is 2 bits
    - Source Register is 3 bits when you have 8 of them
    - Destination Register is another 3 bits when you have 8 of them

Another hidden lesson

- 32 bit operations take less code (no REX prefix).
- Consider: **xorl %eax, %eax**
- <u>32 bit operations clear the upper 4 bytes of any destination register</u>

- Lesson: resorting to 32 bit operations can *sometimes* give you the same 64 bit effect using one less byte in the instruction stream providing a tiny speed increase for free.

Using the higher numbered registers:
49 89 c4   mov   %rax,%r12

- 49 – (0100 1001) REX prefix went from 48 to 49 because r12 needs to have the high bit set in its register number and the REX prefix stores the high bits of the register numbers. In WRXB, W is set (64 bit op), R is not set (upper bit of rax), and B is set (upper bit of r12).
- 89 – MOVQ as before
- c4 – encodes the mode (11) and registers, lower 3 bits of register number for RAX (000) and the lower 3 bits of the register number for R12 (100). This gives 11 000 100 (mode,reg,reg). (The uppermost bit of the register numbers are in the REX prefix.) Breaking the bits into groups of four we get 1100 0100 or c4 in hex.

Using the higher numbered registers:
4c 89 f8   mov   %r15, %rax

- 4c – (0100 1100) REX prefix went from 48 to 4c because r15 needs to have the high bit set in its register number and the REX prefix stores the high bits of the register numbers. In WRXB, W is set (64 bit op), R is set (upper bit of r15), and B is clear(upper bit of rax) giving hex c in the lower nibble.

- 89 – MOVQ as before

- f8 – encodes the mode (11) and registers, lower 3 bits of register number for R15 (111) and the lower 3 bits of the register number for RAX (000). This gives 11 111 000 (mode,reg,reg). (The uppermost bit of the register numbers are in the REX prefix.) Breaking the bits into groups of four we get 1111 1000 or f8 in hex.

More Resources

- https://software.intel.com/en-us/articles/introduction-to-x64-assembly