Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 4: Processor Architecture April 5, 2024

Practice Problems

Exercise 4.1. Determine the byte encoding of the Y86-64 instruction sequence that follows. The line `.pos 0x100` indicates that the starting address of the object code should be `0x100`.

```
.pos 0x100 # Start code at address 0x100
    irmovq $15,%rbx
    rrmovq %rbx,%rcx
loop:
    rmmovq %rcx,-3(%rbx)
    addq   %rbx,%rcsx
    jmp    loop
```

Solution: The `irmovq` instruction has a code part of `0x3` and control part of `0x0`, which we put together as `0x30`. Its instruction format does not use a source register, and a `0x15` indicates; the destination register `%rbx` is encoded as `0x3`. Finally, \$15 is hexadecimal `0xf`, which after extending to 64 bits gives `0x000000000000000f`; we reserve the byte orders to get `0f00000000000000` for the encoding of the instruction. Altogether this takes up 10 bytes, so the next instruction has address `0x10A`, an offset of `.pos` by 10.

The `rrmovq` is encodded as `0x20`. Its source register is `%rbx` with encoding `0x3`, and its destination `%rcx` with encoding `0x1`. Overall, this instruction takes up `2 bytes`, so the next instruction has address `0x10C`.

The `loop` is a placeholder for an address, so it will be replaced by the address `0x1c`.

The `rmmovq` has encoding `0x40`. Its source register is `%rcx` with `0x1`, and the destination is a memory reference with base address given by register `%rbx` with encoding `0x3`, and offset `-3`, which is given by `0xfffffffffffffffd`. We reverse it to `0xfdffffffffffffff` for the instruction encoding. This instruction takes up 10 bytes, so the next one starts at address `0x116`.

The `addq` has encoding `0x60`. Its source and destination registers have encodings `0x3` and `0x1`, respectively. The whole instruction takes up 2 bytes, so the next starting address is `0x118`.

The `jmp` is encoded as `0x70`, and `loop` is replaced by the address `0x010c`, whose bytes we reverse to `0c01000000000000`. The entire translation is:

```
0x100: 30 f3 0f00000000000000
0x10A: 20 31
0x10C: 40 13 fdffffffffffffff
0x116: 60 31
0x118: 70 0c01000000000000
```

Exercise 4.2. For each byte sequence listed, determine the Y86-64 instruction sequence it encodes. IF there is some invalid byte in the sequence, show the instruction sequence up to that point, and indicate where the invalid byte occurs. For each sequence, we show the starting address then a colon, and then the byte sequence.

(a) `0x100:   30f3fcffffffffffffff40630008000000000000`

(b) `0x200:   a06f800c020000000000000030f30a00000000000000`

(c) `0x300:   5054070000000000000010f0b01f`

(d) `0x400:   611373000400000000000000`

(e) `0x500:   6362a0f0`

Solution:

(a) The initial `30` makes this a `irmovq` instruction. The following `f` represents that a source is not needed, and the `3` that the destination register is `%rbx`. The 8-byte value is `fcffffffffffffff`, which is `-4` in decimal. This instruction takes up 10 bytes.

The `40` that follows means we have a `rmmovq` instruction expecting two registers an an offset. The `6` means the source is `%rsi`, and the `3` , means the destination if `%rbx`. The `0008000000000000` reverses to `0000000000008000`, completing the instruction and taking up 9 bytes. The last two `00` indicate the `halt` instruction. Altogether, we have:

```
0x100: irmovq -4,%rbx
0x10C: rmmovq %rsi0x800(%rbx)
0x115: halt
```

(b) The `a0` makes this a `pushq` instruction, the following `6` is source register `%rsi`, and the following `f` that there is no destination register.

Starting at `80`, we have the start of the encoding for a `call` instruction. The destination is encoded by the 8 bytes `0c02000000000000`, which we reverse to `000000000000020c`, or simply `0x020c`.

The next `0x00` indicates a `halt` instruction.

The next `30` means we are parsing a `irmovq` command, with no source register instructed by the `f` and destination register `3` which is `%rbx`. The constant in reverse is given by `0x0a00000000000000`, so we reverse it to `000000000000000a`. The program is:

```
0x200: pushq %rsi
0x202: call 0x000000000000020c
0x20b: halt
0x20c: irmovq $10,%rbx
```

(c) The initial `50` makes this a `mrmovq` instruction. The following byte `54` means that the source and destination registers are `%rsp` and `%rbp`, respectively. The source is a memory reference with a an offset given by the following 8 bytes`0x0700000000000000`, which we reverse to `0x0000000000000007`, or decimal `7`.

The `10` means we have a `nop`. The `f0` is an invalid byte. `b01f` means `popq %rcx`. Altogether we have:

```
0x300: mrmovq 7(%rsp),%rbp
0x30a: nop
0x30b: # invalid
0x30c: popq %rcx
```

(d) The `61` makes this a `subq` instruction, with source register `%rcx` and destination register `%rbx`, as given by `0x13`. The instruction is `subq %rcx, %rbx`.

000400000000000000 The `73` that follows indicates a `je` instruction, followed by the reversed constant address `0004000000000000`, which we reverse to `0000000000000400`. The last `00` indicates a `halt`:

```
0x400: subq %rcx, %rbx
0x402: je 0x0000000000000400
0x40b: halt
```

(e) The `63` makes this a `xorq` instruction, with source register `%rsi` and destination register `%rdx`, as indicated by the `62`. The instruction is `xorq %rsi, %rdx`. We then have `a0` for a `pushq` instruction, but the `f` says that there is no source register, which is invalid:

```
0x500: xorq %rsi, %rdx
0x502: pushq f0 # invalid byte register f0
```