Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 4: Processor Architecture April 6, 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: 30f3fcffffffffffffff4063000800000000000`

(b) `0x200: a06f800c020000000000000030f30a00000000000000`

(c) `0x300: 505407000000000000010f0b01f`

(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 `00080000000000` reverses to `00000000008000`, 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 bytes0x0700000000000000, 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.

00040000000000000 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
```

Exercise 4.3. One common pattern in machine-level programs is to add a constant value to a register. With the Y86-64 presented thus far, this requires first using an irmovq to set a register to the constant, then an addq instruction to add this value to the destination register. Suppose we now want to add a new instruction iaddq with the following format:

Byte	0		1		2 3 4 5 6 7 8 9
iddq V, rB	C	0	F	rB	V

This instruction adds the constant value V to register rB.

Re-write the Y86-64 sum function of Figure 4.6 to make use of the iaddq instruction. In the original version, we dedicate registers %r8 and %r9 to hold constant values. Now, we can avoid using those registers altogether.

Solution:

```
    long sum(long *start, long count)
    start in %rdi, count in %rsi
sum:
    xorq   %rax,%rax  # sum = 0
```

```
    andq   %rsi,%rsi  # Set condition code
    jmp test
loop:
    mrmovq (%rdi),%r10 # Get *start
    addq   %r10,%rax  # Add to sum
    iaddq  $1,%rdi    # start++
    iaddq  $-1,%rsi   # count--, set condition code
test:
    jne    loop       # Stop when 0
    ret
```

Exercise 4.4. Write Y86-64 code to implement a recursive sum function `rsum`, based on the following code:

```
long rsum(long *start, long count)
{
    if (count <= 0)
        return 0;
    return *start + rsum(start+1, coount-1);
}
```

Use the same argument passing and register saving conventions as x86-64 code does. You might find it helpful to compile the C code on an x86-64 machine and then translate the instructions to Y86-64.

Solution: I compiled `rsum.c` with the flags: `gcc -S -O1 rsum.c`, and got the following x86-64 output:

```
    .file  "rsum.c"
    .text
    .globl rsum
    .type  rsum, @function
rsum:
.LFB0:
    .cfi_startproc
    endbr64
    movl   $0, %eax
    testq  %rsi, %rsi
    jle .L5
    pushq  %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq   %rdi, %rbx
    subq   $1, %rsi
    leaq   8(%rdi), %rdi
    call   rsum
    addq   (%rbx), %rax
```

```
    popq   %rbx
    .cfi_def_cfa_offset 8
    ret
.L5:
    .cfi_restore 3
    ret
    .cfi_endproc
.LFE0:
    .size  rsum, .-rsum
    .ident "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
    .section   .note.GNU-stack,"",@progbits
    .section   .note.gnu.property,"a"
    .align 8
    .long  1f - 0f
    .long  4f - 1f
    .long  5
0:
    .string "GNU"
1:
    .align 8
    .long  0xc0000002
    .long  3f - 2f
2:
    .long  0x3
3:
    .align 8
4:
```

I used it as the based for the following Y86-64 code, which I have annotated:

```
    long rsum(long *start, long count)
    start in %rdi, count in %rsi
rsum:
    irmovq $0,%rax    # Set return value to 0
    iaddq  $0,%rsi    # Set condition code
    jle    .L5:       # if <= 0 goto .L5
    pushq  %rbx       # Callee-saved register
    rrmovq %rdi,%rbx  # save start (memory address)
    iaddq  $1,%rsi    # count--
    iaddq  $8,%rdi    # start++ (add 8 bytes to pointer, pointing to next)
    call rsum         # rsum(start++, count--)
    mrmovq (%rbx),%rax # add *start to return value
    popq   %rbx       # restore register
    ret               # return
.L5:
    ret               # return
```

Exercise 4.5. Modify the Y86-64 code for the `sum` function (Figure 4.6) to implement a function `absSum` that computes the sum of absolute values of an array. Use a *conditional jump* instruction within your inner loop.

Solution: My implementation uses the fact that if `x` is negative, then `~x + 1` negates it, making it positive.

```
    long absSum(long *start, long count)
    start in %rdi, count in %rsi
absSum:
    irmovq $-1,%r11      # Constant -1
    xorq   %rax,%rax  # sum = 0
    andq   %rsi,%rsi  # Set condition code
    jmp test
loop:
    mrmovq (%rdi),%r10 # Get *start
    iadd0  $0,%r10    # Set condition code
    jge    .nonneg    # if >= goto .nonneg
    xorq   %r11,%r10  # -1 XOR *start
    iaddq  $1,%r10    # finish computing abs(*start)
.nonneg:
    addq   %r10,%rax  # Add to sum
    iaddq  $1,%rdi    # start++
    iaddq  $-1,%rsi   # count--, set condition code
test:
    jne    loop       # Stop when 0
    ret
```

Exercise 4.6. Modify the Y86-64 code for the `sum` function (Figure 4.6) to implement a function `absSum` that computes the sum of absolute values of an array. Use a *conditional move* instruction within your inner loop.

Solution: My implementation uses the fact that if `x` is negative, then `~x + 1` negates it, making it positive.

```
       long absSum(long *start, long count)
       start in %rdi, count in %rsi
absSum:
       irmovq $-1,%r11      # Constant -1
       xorq   %rax,%rax  # sum = 0
       andq   %rsi,%rsi  # Set condition code
       jmp test
loop:
       mrmovq (%rdi),%r10 # Get *start
       xorq   %r10,%r11  # -1 XOR *start
       iaddq  $1,%r11    # Finishes computing -*start
       cmovge %r11,%r10  # if %r11 is positive, then %r10 was negative
```

```
    addq   %r10,%rax  # Add to sum
    iaddq  $1,%rdi    # start++
    iaddq  $-1,%rsi   # count--, set condition code
    test:
    jne    loop       # Stop when 0
    ret
```

Exercise 4.7. Let us determine the behavior of the instruction `pushq %rsp` for an x86-64 processor. We could try reading the Intel documentation on this instruction, but a simpler approach is to conduct an experiment on an actual machine. The C compiler would not normally generate this instruction, so we must use hand-generated assembly code for this task. Here is a test function we have written (Web Aside ASM:EASM on page 178 describes how to write programs that combine C code with handwritten assembly code):

```
    .text
.global pushtest
pushtest:
    movq   %rsp,%rax  # Copy stack pointer
    pushq  %rsp       # Push stack pointer
    popq   %rdx       # Pop it back
    subq   %rdx,%rax  # Return 0 or 4
    ret
```

In our experiments, we find that `pushtest` always returns 0. What does this imply about the behavior of the instruction `pushq %rsp` under x86-64?

Solution: The first instruction stores the old value of the pointer in `%rax` Since the output is always 0, this means that `pushq %rsp` pushes the original value of `%rsp`.

Exercise 4.8. The following assembly-code function lets us determine the behavior of `popq %rsp` on x86-64:

```
    .text
.global poptest
poptest:
    movq   %rsp,%rdi  # Save stack pointer
    pushq  $0xabcd    # Push test value
    popq   %rsp       # Pop to stack pointer
    movq   %rsp,%rax  # Set popped value as returned value
    movq   %rdi,%rsp  # Restore stack pointer
    ret
```

We find this function always returns `0xabcd`. What does this imply about the behavior of `pop %rsp`? What other Y86-64 instruction would have the same behavior?

Solution: It implies that `pop %rsp` sets the stack pointer to the value read from memory. We could use `mrmovq` to read this value instead. This would give the correct return value;

however, the value we pushed would remain on the stack, and would therefore point to a lower address than the `popq` approach.

Exercise 4.9. Write an HCL expression for a signal `xor`, equal to the *exclusive-or* of inputs `a` and `b`. What is the relation between the signals `xor` and `eq` defined above?

Solution: If we let $\oplus$ be the XOR symbol, we can define it to mean

$$\texttt{a} \oplus \texttt{b} = (\texttt{a \&\& !b}) \texttt{ || } (\texttt{!a \&\& b})$$

`xor` and `eq` are negations of one another, because `xor` is 1 when `a` and `b` are distinct, and 0 otherwise. Meanwhile, `eq` is 1 when `a` and `b` are equal, and 0 otherwise.