

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 encoded 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 `0xffffffffffffffd`. We reverse it to `0xfdfdfdfdfdfdfdfdf` 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 fdfdfdfdfdfdfdfdf
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: 30f3fcffffffffffffffff40630008000000000000
- (b) 0x200: a06f800c020000000000000030f30a0000000000000
- (c) 0x300: 505407000000000000000010f0b01f
- (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 and an offset. The 6 means the source is `%rsi`, and the 3, means the destination is `%rbx`. The `0008000000000000` 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 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
```

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
<code>iaddq V, rB</code>	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, count-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 $\sim x + 1$ 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
    iaddq  $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 $\sim x + 1$ 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 x86-64 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

$$a \oplus b = (a \ \&\& \ !b) \ || \ (!a \ \&\& \ b)$$

`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.

Exercise 4.10. Suppose you want to implement a word-level equality circuit using the *EXCLUSIVE-OR* circuits from Problem 4.9 rather than bit-level equality circuits. Design such a circuit for a 64-bit *EXCLUSIVE-OR* circuits and two additional logic gates.

Solution: The circuit will have two 64-bit buses `A` and `B` as inputs. We use a total of 64 bit-level *EXCLUSIVE-OR* circuits operating in parallel. If *any* of them outputs 1, then the two inputs are distinct, and the overall output should be 0. If all of the bit-level XORs output 0, then the circuit should output 1. We therefore feed the outputs of of the bit-level XORs into a 64-bit OR gate, and pass its output through an inverter.

Exercise 4.11. The HCL code for computing the minimum of three words contains four comparison expressions of the form `X <= Y`. Rewrite the code to compute the same result, but using only three comparisons.

Solution: The given HCL was:

```
word Min3 = [
    A <= B && A <= C: A;
    B <= A && B <= C: B;
    1                : C;
];
```

If the first inequality fails, then `A` is not the minimum. Therefore either `B` or `C` is the minimum. This means it's enough to check that `B <= C`:

```
word Min3 = [
    A <= B && A <= C: A;
    B <= C           : B;
    1                : C;
];
```

Exercise 4.12. Write HCL code describing a circuit that for word inputs `A`, `B`, and `C` selects the *median* of the three values. That is, the output equals the word lying between the minimum and the maximum of the three inputs.

Solution:

```
word Median = [
    A <= B && C <= A || (A <= C && B <= A): A;
    A <= B &&
    1          : C;
];
```

Exercise 4.13. Fill in the right-hand column of the following table to describe the processing of the `irmovq` on line 4 of the object code in Figure 4.17:

Stage	Generic <code>irmovq V, rB</code>	Specific <code>irmovq \$128,%rsp</code>
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$	
Decode		
Execute	$\text{valE} \leftarrow 0 + \text{valC}$	
Memory		
Write back	$R[\text{rB}] \leftarrow \text{valE}$	
PC update	$\text{PC} \leftarrow \text{valP}$	

How does this instruction execution modify the registers and the PC?

Solution:

Stage	Generic <code>irmovq V, rB</code>	Specific <code>irmovq \$128,%rsp</code>
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$	$M_1[0x00a] = 3:0$ $\text{rA:rB} \leftarrow M_1[0x00b] = f:3$ $\text{valC} \leftarrow M_8[0x00c] = 0x15$ $\text{valP} \leftarrow 0x00a + 10 = 0x014$
Decode		
Execute	$\text{valE} \leftarrow 0 + \text{valC}$	$\text{valE} \leftarrow 0 + 0x15$
Memory		
Write back	$R[\text{rB}] \leftarrow \text{valE}$	$R[\text{rB}] \leftarrow \text{valE} = 0x15$
PC update	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP} = 0x014$

The stack pointer `%rsp` is updated with the value `0x15`, and the PC is updated with the value `0x014`.

Exercise 4.14. Fill in the right-hand column of the following table to describe the processing of the `popq` instruction on line 7 of the object in Figure 4.17.

Stage	Generic popq rA	Specific popq %rax
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC} + 2$	
Decode	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	
Write back	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$	
PC update	$\text{PC} \leftarrow \text{valP}$	

What effect does this instruction have on the registers and the PC?

Solution:

Stage	Generic popq rA	Specific popq %rax
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC} + 2$	$\text{icode:ifun} \leftarrow M_1[\text{PC}] = \text{b:0}$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1] = \text{0:f}$ $\text{valP} \leftarrow \text{PC} + 2 = \text{0x02e}$
Decode	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$	$\text{valA} \leftarrow R[\%rsp] = 128$ $\text{valB} \leftarrow R[\%rsp] = 128$
Execute	$\text{valE} \leftarrow \text{valB} + 8$	$\text{valE} \leftarrow \text{valB} + 8 = 136$
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	$\text{valM} \leftarrow M_8[\text{valA}] = 9$
Write back	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$	$R[\%rsp] \leftarrow \text{valE} = 136$ $R[\text{rA}] \leftarrow \text{valM} = 9$
PC update	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP} = \text{0x02e}$

Exercise 4.15. What would be the effect of the instruction `push %rsp` according to the steps listed in Figure 4.20? Does this conform to the desired behavior for Y86-64, as determined in Problem 4.7?

Solution: During the *Decode* step, `valA` and `valB` would both receive the value `R[%rsp]`, which is the address currently stored at the top of the stack. Since the *Memory* step occurs before the *Write back* step, we see that the old value in `%rsp` is written to the memory location stored at the new address and then the new address is stored at `%rsp`. This is consistent with Problem 4.7

Exercise 4.16. Assume the two register writes in the write-back stage for `popq` occur in the order listed in Figure 4.20. What would be the effect of executing `popq %rsp`? Does this conform to the desired behavior for Y86-64, as determined in Problem 4.8?

Solution: The *Decode* reads two copies of the address stored in `%rsp`, stored in `valA` and `valB`. In the *Execute* step, the newly computed stack pointer is computed as `valB + 8`, but the stack pointer is not updated yet. In the *Memory* step, the value at that address is

extracted into `valM`, but not assigned to anything yet. In the *Write back* step, the stack pointer `%rsp` is first updated to `valB + 8`, meaning that its address is being incremented by 8 bytes. Then, the value `valM` computed previously is stored at the stack pointer. The result is that the stack pointer address has been increased by 8, and the value at its new location is updated to `valM`, the value referred to by the old memory address of `%rsp`.

Exercise 4.17. We can see by the instruction encodings (Figure 4.2 and 4.3) that the `rmovq` instruction is the unconditional version of a more general class of instructions that include the conditional moves. Show how you would modify the steps for the `rrmovq` instruction below to also handle the six conditional move instructions. You may find it useful to see how the implementation of the `jXX` instructions (Figure 4.21) handles conditional behavior.

Stage	<code>cmovXX rA, rB</code>
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC} + 2$
Decode	$\text{valA} \leftarrow R[\text{rA}]$
Execute	$\text{valE} \leftarrow 0 + \text{valA}$
Memory	
Write back	$R[\text{rB}] \leftarrow \text{valE}$
PC update	$\text{PC} \leftarrow \text{valP}$

Solution: During the *Fetch* step, the `ifun` specifies the function that will be applied to the condition code to decide if the move should occur. Then, in the *Execute* step we compute `Cnd` as `Cond(CC,ifun)`, meaning that the condition codes will be used as an input to the decoded function, producing a signal `Cnd` that is 1 or 0. In the *Write back* step, this signal is used to assign either `valA` or `valB` to `rB`.

Stage	<code>cmovXX rA, rB</code>
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC} + 2$
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$
Execute	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$
Memory	
Write back	$R[\text{rB}] \leftarrow \text{Cnd} ? \text{valA} : \text{valB}$
PC update	$\text{PC} \leftarrow \text{valP}$

Exercise 4.18. Fill in the right-hand column of the following table to describe the processing of the `call` instruction on line 9 of the object code in Figure 4.17:

Stage	Generic call Dest	Specific call 0x041
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC} + 9$	
Decode	$\text{valB} \leftarrow R[\%rsp]$	
Execute	$\text{valE} \leftarrow \text{valB} + (-8)$	
Memory	$M_8[\text{valE}] \leftarrow \text{valP}$	
Write back	$R[\%rsp] \leftarrow \text{valE}$	
PC update	$\text{PC} \leftarrow \text{valC}$	

What effect would this instruction execution have on the registers, the PC, and the memory?

Solution:

Stage	Generic call Dest	Specific call 0x041
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC} + 9$	$\text{icode:ifun} \leftarrow M_1[0x037] = 8:0$ $\text{valC} \leftarrow M_8[0x038] = 0x0000000000000041$ $\text{valP} \leftarrow 0x040$
Decode	$\text{valB} \leftarrow R[\%rsp]$	$\text{valB} \leftarrow 128$
Execute	$\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valE} \leftarrow 120$
Memory	$M_8[\text{valE}] \leftarrow \text{valP}$	$M_8[120] \leftarrow 0x040$
Write back	$R[\%rsp] \leftarrow \text{valE}$	$R[\%rsp] \leftarrow 120$
PC update	$\text{PC} \leftarrow \text{valC}$	$\text{PC} \leftarrow 0x0000000000000041$