Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 4: Processor Architecture April 10, 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.

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 th 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 the describe the processing of the `irmovq` on line 4 of the object code in Figure 4.17:

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

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

Solution:

Stage	Generic `irmovq V, rB`	Specific `irmovq $128,%rsp`

Fetch	icode:ifun $\leftarrow$ M_1[PC]
	`rA:rB` $\leftarrow$ M_1[PC + 1]
	`valC` $\leftarrow$ M_8[PC + 2]
	`valP` $\leftarrow$ PC+ 10
		M_1`[0x00a] = 3:0`
		`rA:rB` $\leftarrow$ M_1`[0x00b] = f:3`
		`valC` $\leftarrow$ M_8`[0x00c] = 0x15`
		`valP` $\leftarrow$ `0x00a` + 10 = `0x014`

Decode		
Execute	`valE` $\leftarrow$ 0 + `valC`	`valE` $\leftarrow$ 0 + `0x15`
Memory		
Write back	`R[rB]` $\leftarrow$ `valE`	`R[rB]` $\leftarrow$ `valE = 0x15`
PC update	`PC` $\leftarrow$ `valP`	`PC` $\leftarrow$ `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	icode:ifun $\leftarrow$ M_1[PC]	
	rA:rB $\leftarrow$ M_1[PC+1]	
	valP $\leftarrow$ PC + 2	
Decode	valA $\leftarrow$ R[%rsp]	
	valB $\leftarrow$ R[%rsp]	
Execute	valE $\leftarrow$ valB + 8	
Memory	valM $\leftarrow$ M_8[valA]	
Write back	R[%rsp] $\leftarrow$ valE	
	R[rA] $\leftarrow$ valM	
PC update	PC $\leftarrow$ valP	

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

Solution:

Stage	Generic `popq rA`	Specific `popq %rax`
Fetch	icode:ifun $\leftarrow$ M_1[PC]	icode:ifun $\leftarrow$ M_1[PC] = b:0
	rA:rB $\leftarrow$ M_1[PC+1]	rA:rB $\leftarrow$ M_1[PC+1] = 0:f
	valP $\leftarrow$ PC + 2	valP $\leftarrow$ PC + 2 = 0x02e
Decode	valA $\leftarrow$ R[%rsp]	valA $\leftarrow$ R[%rsp] = 128
	valB $\leftarrow$ R[%rsp]	valA $\leftarrow$ R[%rsp] = 128
Execute	valE $\leftarrow$ valB + 8	valE $\leftarrow$ valB + 8 = 136
Memory	valM $\leftarrow$ M_8[valA]	valM $\leftarrow$ M_8[valA] = 9
Write back	R[%rsp] $\leftarrow$ valE	R[%rsp] $\leftarrow$ valE = 136
	R[rA] $\leftarrow$ valM	R[rA] $\leftarrow$ valM = 9
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP = 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	`cmovXX rA, rB`
Fetch	icode:ifun $\leftarrow$ M_1[PC]
	rA:rB $\leftarrow$ M_1[PC+1]
	valP $\leftarrow$ PC + 2
Decode	valA $\leftarrow$ R[rA]
Execute	valE $\leftarrow$ 0 + valA
Memory	
Write back	R[rB] $\leftarrow$ valE
PC update	PC $\leftarrow$ 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	`cmovXX rA, rB`
Fetch	icode:ifun $\leftarrow$ M_1[PC]
	rA:rB $\leftarrow$ M_1[PC+1]
	valP $\leftarrow$ PC + 2
Decode	valA $\leftarrow$ R[rA]
	valB $\leftarrow$ R[rB]
Execute	Cnd $\leftarrow$ Cond(CC, ifun)
Memory	
Write back	R[rB] $\leftarrow$ Cnd ? valA: valB
PC update	PC $\leftarrow$ 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	icode:ifun $\leftarrow$ M_1[PC]	
	valC $\leftarrow$ M_8 [PC+1]	
	valP $\leftarrow$ PC + 9	
Decode	valB $\leftarrow$ R[%rsp]	
Execute	valE $\leftarrow$ valB + (−8)	
Memory	M_8[valE] $\leftarrow$ valP	
Write back	R[%rsp] $\leftarrow$ valE	
PC update	PC $\leftarrow$ 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	icode:ifun $\leftarrow$ M_1[PC]	icode:ifun $\leftarrow$ M_1[0x037] = 8:0
	valC $\leftarrow$ M_8 [PC+1]	valC $\leftarrow$ M_8 [0x038] = 0x0000000000000041
	valP $\leftarrow$ PC + 9	valP $\leftarrow$ 0x040
Decode	valB $\leftarrow$ R[%rsp]	valB $\leftarrow$ 128
Execute	valE $\leftarrow$ valB + (−8)	valE $\leftarrow$ 120
Memory	M_8[valE] $\leftarrow$ valP	M_8[120] $\leftarrow$ 0x040
Write back	R[%rsp] $\leftarrow$ valE	R[%rsp] $\leftarrow$ 120
PC update	PC $\leftarrow$ valC	PC $\leftarrow$ 0x0000000000000041

Exercise 4.19. Write HCL code for the signal `need_valC` in the SEQ implementation?

Solution: The `irmovq` instruction places the constant provided in the source into the destination. The `rmmovq` instruction requires specifying an offset to the destination memory address. The `mrmovq` instruction requires specifying an offset to the source memory address. The jump instructions, `jXX`, require specifying a label that corresponds to the address of instruction to execute after the jump. Finally, the `call` instruction requires specifying a label corresponding to the first instruction in the procedure being called.

```
bool need_valC =
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ IJXX, ICALL};
```

Exercise 4.20. The register signal `srcB` indicates which register should be read to generate the signal `valB`. The desired value is shown as the second step in the double stage in Figures 4.18 to 4.21. Write HCL code for `srcB`.

Solution: We need `srcB` in `rmmovq` because the second register gives the base address in a memory reference. For `mrmovq`, register `rB` is in the first position, where once again its address is used to compute the base memory address, as seen in Figure 4.19 on page 389. For `OPq`, `srcB` is the second integer operand.

For `pushq`, `srcB` gives the value of the stack pointer, which is use to compute the new stack pointer address `popq`, `valB` once again holds the stack pointer value, and same for `call` and `ret` (see Figure 4.20 and 4.21 in pages 391 and 393).

```
word srcB = [
    icode in { IRMMOVQ, IRMMOVQ, IOPQ } : rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET };
    1:  RNONE; # Don't need register
]
```

Exercise 4.21. Register ID `dstM` indicates the destination register for write port `M`, where `valM`, the value read from memory, is stored. This is shown in Figures 4.18 to 4.21 as the second step in the write-back stage. Write HCL code for `dstM`.

Solution: From the figures we see that the value read from memory is only ever stored into `rA`.

```
word destM = [
    icode in { IMRMOVQ, IPOPQ }: rA;
    1: RNONE;
];
```

Exercise 4.22. Only the `popq` instruction uses both register file write ports simultaneously. For the instruction `pop %rsp`, the same address will be used for both `E` and `M` write ports, but with different data. To handle this conflict, we must stablish a *priority* among the two write ports so that when both attempt tow rite the same register on the same cycle, only the write from the higher-priority port takes places. Which of the two ports should be given priority in order to implement the desired behavior, as determined in Practice Problem 4.8?

Solution: In Practice Problem 4.8 we determined that the resulting value is that which was read from memory, so port `M` should be given priority in order to be consistent with that.

Exercise 4.23. Based on the first operand of the first step of the execute stage in Figures 4.18 to 4.21, write an HCL description for the signal `aluB` in SEQ.

Solution: In `IOPQ`, the value `valB` is the second operand. It's also used in `mrmovq` and `rmmovq` to represent the base address of a memory reference, and in `pushq`, `popq`, `call`, and `ret` for the stack pointer value that we either add or decrement. In `rrmovq`, the first operand is `valA` and the second is `0`, the goal is to compute `valA` to store in the destination register. Therefore, `aluB` is 0 here. For `irrmovq`, `aluA` is `valC` while `aluB` is 0.

```
word aluB = [
    icode in { IRMMOVQ, IMRMOVQ, IOPQ, IPUSHQ, IPOPQ, ICALL, IRET } : valB;
    icode in { IRRMOVQ, IIRMOVQ} : 0;
    1 : IRNONE;
];
```

Exercise 4.24. The conditional move instructions, abbreviated `cmovXX`, have instruction code `IRRMOVQ`. As Figure 4.28 shows, we can implement these instructions by making use of the `Cnd` signal, generated in the execute stage. Modify the HCL code for `dstE` to implement these instructions.

Solution: For `cmovXX`, we only ever write to the destination register, `rB`, if `Cnd` is asserted.

```
word destE = [
    Cnd && icode in { IRRMOVQ } : rB;
    icode in { IRRMOVQ } : rB;
    icode in { IIRMOVQ, IOPQ } : rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE;
];
```

Exercise 4.25. Looking at memory operations for the different instructions shown in Figures 4.18 to 4.21, we can see that the data for memory writes are always either `valA` or `valP`. Write HCL code for the signal `mem_data` in `SEQ`.

Solution:

```
word mem_data = [
    icode in { IRMMOVQ, IPUSHQ} : valA;
    icode == ICALL : valP;
];
```

Exercise 4.26. We want to control the signal `mem_write` only for instructions that write data to memory. Write HCL code for the signal `mem_write` in SEQ.

Solution: The instructions that write data to memory are precisely those named in Exercise 4.25:

```
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
```

Exercise 4.27. Write HCL code for `Stat`, generating the four status `SAOK`, `SADR`, `SINS`, and `SHLT`.

Solution:

```
Stat = [
    imem_error || dmem_error: SADDR;
    !instr_valid : SINS;
    icode == IHALT : SHALT;
    1: SAOK;
];
```

Exercise 4.28. Suppose we analyze the combinational logic of Figure 4.32 and determine that it can be separated into a sequence of six blocks, named A to F, having delays of 80, 30, 60, 50, 70, and 10 ps, respectively, illustrated as follows:

We can create pipelined versions of this design by inserting pipeline registers between pairs of these blocks. Different combinations of pipeline depth (how many stages) and maximum throughput arise, depending on where we insert the pipeline registers. Assume that pipeline register has a delay of 20 ps.

(a) Inserting a single register gives a two-stage pipeline. Where should the register be inserted to maximize throughput? What would be the throughput and latency?

(b) Where should two registers be inserted to maximize throughput of a three-stage pipeline? What should be the throughput and latency?

(c) Where should three registers be inserted to maximize the throughput of a 4-stage pipeline? What would be the throughput and latency?

(d) What is the maximum number of stages that would yield a design with the maximum achievable throughput? Describe this design, its throughput, and its latency.

Solution:

(a) The sum of the delays of all the blocks is still 300 ps. If we place the register between C and D, the first stage will have a delay of 170 ps, while the second stage has a delay of 130 ps. Since the registers have a delay of 20 ps, this means can cycle the clocks every $170 + 20 = 190$ ps, yielding a throughput of around 5.26 GIPS. Since processing a full instruction requires 2 cycles, this means the latency is 2×190 ps $= 380$ ps.

(b) We can place one register between B and C, and another register between D and E. Then the first stage will take up 110 ps, the second stage will take up 110 ps, and the last stage will take 80 ps. Hence we can cycle the clocks every $110 + 20$ ps $= 130$ ps, making the throughput 7.69 GIPS. The latency however is now $3 \times 130 = 390$ ps.

(c) Ideally we will place one between A and B, another between C and D, and one more between D and E. The slowest stage would then be given by the sum of the delay of components B and C, for a total of 90 ps. With the added 20 ps register delay, the throughput becomes around 9.09 GIPS. The delay latency is $4 \times 9 = 360$ ps.

(d) Because the first component requires a delay of at least 80 ps and adding a register implies a delay of 20 ps, we know that we will be need to clock at least every 100 ps, for a maximum throughput of 10 GIPS. To attain it, we require a register between A and B, between B and C,between C and D, and between D and E, for a total of 5 stages. The latency is $5 \times 100 = 500$ ps.

Exercise 4.29. Suppose we could take the system of Figure 4.32 and divide it into an arbitrary number of pipeline stages k, each having a delay of $300/k$, and with each pipeline register having a delay of 20 ps.

(a) What would be the latency and the throughput of the system, as functions of k?

(b) What would be the ultimate limit on the throughput?

Solution:

(a) The throughput is given by $T(k)$, where

$$\begin{aligned} T(k) &= \frac{1}{\frac{300}{k} + 20} \times 10^{12} \\ &= \frac{1}{\frac{300+20k}{k}} \times 10^{12} \\ &= \frac{k}{300 + 20k} \times 10^{12} \end{aligned}$$

in units of instructions per second. For example, if $k = 1$, we get 3.125 GIPS. The latency is given by $L(k)$, where

$$\begin{aligned} L(k) &= k \cdot \left(\frac{300}{k} + 20 \right) \times 10^{-12} \\ &= (300 + 20k) \times 10^{-12} \end{aligned}$$

in units of seconds. For example, if $k = 1$, then the latency is 300 ps.

(b) We see that $\lim_{k \to \infty} T(k) = \frac{1}{20} \times 10^{12}$ instructions per second, which equates 50 GIPS.

Exercise 4.30. Write HCL code for the signal `f_stat`, providing the provisional status for the fetched instruction.

Solution: This is largely the same as Exercise 4.27, but instead of `icode`, we use `f_icode`, and we do not use the `dmem_error` signal because that is handled in the Memory stage.

```
word f_stat = [
    imem_error : SADR;
    !instr_valid : SINS;
    f_icode == SHLT : SHLT;
    1: SAOK;
];
```

Exercise 4.31. The block labeled "dstE" in the decode stage generates the register ID for the E port of the register file, based on fields from the fetched instruction in pipeline register D. The resulting signal is named `d_dstE` in the HCL description of PIPE. Write HCL code for this signal, based on the HCL description of the SEQ signal `dest_E`. (See the code for SEQ in Section 4.3.4). Do not concern yourself with the logic to implement condition moves yet.

Solution: In Section 4.3.4, the following HCL description was given for `dstE`

```
# HCL for dstE signal in SEQ implementation of Y86-64
# WARNING: Conditional move not implemented correctly here
word dstE = [
    icode in { IRRMOVQ } : rB;
    icode in { IIRMOVQ, IOPQ } : rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # Don't write any register
];
```

Figure 4.58 shows the PIPE decode and write-back stage logic, indicating that block `destE` receives inputs `D_icode` and `D_rB`. Its output is the same after using the new signal names:

```
word d_dstE = [
    D_icode in { IRRMOVQ } : D_rB;
    D_icode in { IIRMOVQ, IOPQ } : D_rB;
    D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE;
];
```

Exercise 4.32. Suppose the order of the third and fourth cases (the two forwarding sources from the memory stage) in the HCL code for `d_valA` were reversed. Describe the resulting behavior of the `rrmovq` instruction (line 5) for the following program:

```
irmovq $5, %rdx
irmovq $0x100 %rsp
rmmovq %rdx,0(%rsp)
popq   %rsp
rrmovq %rsp,%rax
```

Solution: The `rrmovq` on line 5 uses %rsp as a source register, but the the `popq` instruction on line 4 read and writes its value for memory, leading to a load/use data hazard as described on page 439. To handle this, the processor stalls the `rrmovq` instruction in line 5 in the decode stage, allowing the `pop %rsp` instruction to move on to the memory stage, at which point the value can be forwarded. However, the `popq %rsp` instruction produces signals `M_dstM` amd `M_dstE`, both of which are `R[%rsp]` (see Figure 4.20 on page 391),and hence they both match `d_srcA`, the source operand to `rrmovq` on line 5. Since the order of the third and fourth cases is reversed in the HCL Logic for `d_valA`, this causes the `M_dstE` branch to be chosen, causing `d_valA` to take on the value of `M_valE`, the value read, which is the new value of the stack pointer (namely, `0x108`), which is what will be written to `%rax`. However this is incorrect, because if the instructions were executed sequentially, then the instruction `pop %rsp` would have the value `5`, which is precisely `m_valM`, the value we would have chosen if we had not reversed the order.

Exercise 4.33. Suppose the order of the fifth and sixth cases (the two forwarding sources from the write-back stage) in the HCL code for `d_valA` were reversed. Write a Y86-64

program that would be executed incorrectly. Describe how the error would occur and its effect on the program behavior.

Solution: The following HCL code would execute incorrectly:

```
irmovq 5, %rdx
irmovq 0x100, %rsp
rmmovq %rdx, 0(%rsp)
pop %rsp
nop
nop
rrmovq %rsp, %rax
```

The code sets `R[%rdx]` to 5 and `R[%rsp]` to `0x100`. It then loads `5` at M_8`[%rsp]`. The `pushq` is meant to duplicate the value 5 at memory location `0x0F8`, the new memory location that `%rsp` holds. When `rrmovq` reaches the decode stage, it receives the forwarded values from the `popq` instruction that is in the Write-back stage. Here, `W_dstM` and `W_dstE` are both `%rsp`, and since the fifth and sixth cases are reversed, this means that the processor will take the action matching `W_dstE`, meaning that `d_valA` will be set to `W_valE`, which is the value `0x108`. However the intended behavior is to get the value 5.

Exercise 4.34. Write HCL code for the signal `d_valB`, giving thee value for source operand `valB` supplied to pipeline register E.

Solution: The cases are largely the same as for `d_valA`, but since `d_valB` does not have the dual usage of holding instruction addresses, it's slightly simpler. All forwarded values take precedence, giving priority to values written to memory, and to upcoming instructions at an earlier stage (indicating the last change).

```
word d_valB = [
    d_srcB == e_dstE : e_valE;
    d_srcB == M_dstM : m_valM;
    d_srcB == M_dstE : M_valE;
    d_srcB == W_dstM : W_valM;
    d_srcB == W_dstE : W_valE;
    1 : d_rvalB;
];
```

Exercise 4.35. Our second case in the HCL code for `d_valA` uses signal `e_dstE` to see whether to select the ALU output `e_valE` as the forwarding source. Suppose instead that we use signal `E_dstE`, the destination register ID in pipeline register E for this selection. Write a Y86-64 program that would give an incorrect result with this modified forwarding logic.

Solution: Based on Figure 4.60, the diagram for the PIPE execute logic shows that the control logic block for `dstE` that produces `e_dstE` accepts `e_Cnd` and `E_icode` as inputs. Therefore, the output `dstE` produced by `e_dstE` takes care of conditional move operations issued by `cmovXX` instructions, which does not get accounted for when forwarding `E_dstE`. Therefore, the following program would execute incorrectly:

```
irmovq $-1, %rsi
irmovq $50, %rdx
irmovq $-50, %rbx
addq %rdx, %rbx # Set CC
cmovne %rsi, %rbx
rrmovq %rbx, %rax
```

The addq will turn off ZF, causing e_Cnd to be on, because the addition yields 0. Because E_dstE does not account for this, the rrmovq instruction receives the value that %rbx would have had if the conditional move had succeeded – namely, −1, the value of %rsi. However, because the move failed, it should have used −50 from its register.

Exercise 4.36. In this stage, we can complete the computation of the status code Stat by detecting the case of an invalid address for the data memory. Write HCL code for the signal m_stat.

Solution: We can identify any existing errors due to an invalid memory address, and if none were detected, then forward the status we received.++

```
word m_stat = [
    dmem_error : SADR;
    1 : M_stat;
];
```

Exercise 4.37. Write a Y86-64 assembly-language program that causes combination A to arise and determines whether the control logic handles it correctly.

Solution: Combination A involves a jump which, if taken, would have led to a return statement being executed early.

```
irmovq $1, %rdx
irmovq $-1, %rax
add %rdx, %rax    # Set ZF to 1
jne .bad_input    # if != 0 goto .bad_input (not taken)
irmovq $32, %rax  # Should execute
ret
.bad_input:       # Should not execute, but if PIPE control logic is wrong, it
    will.
ret
```

Since the PIPE processor always takes jumps, the last ret instruction would cause the value 0 to be returned to the caller, but this instruction should be canceled because the branch is actually not taken and the irmovq should execute before returning. If the program behaves incorrectly, then %rax will have the value 0, when it should be 32.