Stage	`OPq` rA, rB	`rrmovq` rA, rB	`irmovq` V, rB
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valP $\leftarrow$ PC + 2	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valP $\leftarrow$ PC + 2	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valC $\leftarrow$ M_8[PC + 2] valP $\leftarrow$ PC + 10
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	valA $\leftarrow$ R[rA]	
Execute	valE $\leftarrow$ valB OP valA Set CC	valE $\leftarrow$ 0 + valA	valE $\leftarrow$ 0 + valC
Memory			
Write back	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Figure 4.18 Computations in sequential implementation of Y86-64 instructions `OPq`, `rrmovq`, and `irmovq`. These instructions compute a value and store the result in a register. The notation icode : ifun indicates the two components of the instruction byte, while rA : rB indicates the two components of the register specifier byte. The notation $M_1[x]$ indicates accessing (either reading or writing) 1 byte at memory location x, while $M_8[x]$ indicates accessing 8 bytes.

then written to the register file. Similar processing occurs for `irmovq`, except that we use constant value valC for the first ALU input. In addition, we must increment the program counter by 10 for `irmovq` due to the long instruction format. Neither of these instructions changes the condition codes.

Practice Problem 4.13 (solution page 521)

Fill in the right-hand column of the following table to describe the processing of the `irmovq` instruction 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	

Stage	`rmmovq` rA, D(rB)	`mrmovq` D(rB), rA
Fetch	icode:ifun $\leftarrow$ M_1[PC]	icode:ifun $\leftarrow$ M_1[PC]
	rA:rB $\leftarrow$ M_1[PC + 1]	rA:rB $\leftarrow$ M_1[PC + 1]
	valC $\leftarrow$ M_8[PC + 2]	valC $\leftarrow$ M_8[PC + 2]
	valP $\leftarrow$ PC + 10	valP $\leftarrow$ PC + 10
Decode	valA $\leftarrow$ R[rA]	
	valB $\leftarrow$ R[rB]	valB $\leftarrow$ R[rB]
Execute	valE $\leftarrow$ valB + valC	valE $\leftarrow$ valB + valC
Memory	M_8[valE] $\leftarrow$ valA	valM $\leftarrow$ M_8[valE]
Write back		
		R[rA] $\leftarrow$ valM
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Figure 4.19 Computations in sequential implementation of Y86-64 instructions `rmmovq` and `mrmovq`. These instructions read or write memory.

Figure 4.20 shows the steps required to process `pushq` and `popq` instructions. These are among the most difficult Y86-64 instructions to implement, because they involve both accessing memory and incrementing or decrementing the stack pointer. Although the two instructions have similar flows, they have important differences.

The `pushq` instruction starts much like our previous instructions, but in the decode stage we use `%rsp` as the identifier for the second register operand, giving the stack pointer as value valB. In the execute stage, we use the ALU to decrement the stack pointer by 8. This decremented value is used for the memory write address and is also stored back to `%rsp` in the write-back stage. By using valE as the address for the write operation, we adhere to the Y86-64 (and x86-64) convention that `pushq` should decrement the stack pointer before writing, even though the actual updating of the stack pointer does not occur until after the memory operation has completed.

The `popq` instruction proceeds much like `pushq`, except that we read two copies of the stack pointer in the decode stage. This is clearly redundant, but we will see that having the stack pointer as both valA and valB makes the subsequent flow more similar to that of other instructions, enhancing the overall uniformity of the design. We use the ALU to increment the stack pointer by 8 in the execute stage, but use the unincremented value as the address for the memory operation. In the write-back stage, we update both the stack pointer register with the incremented stack pointer and register rA with the value read from memory. Using the unincremented stack pointer as the memory read address preserves the Y86-64

Stage	pushq rA	popq rA
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valP $\leftarrow$ PC + 2	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[%rsp]	valA $\leftarrow$ R[%rsp] valB $\leftarrow$ R[%rsp]
Execute	valE $\leftarrow$ valB + (−8)	valE $\leftarrow$ valB + 8
Memory	M_8[valE] $\leftarrow$ valA	valM $\leftarrow$ M_8[valA]
Write back	R[%rsp] $\leftarrow$ valE	R[%rsp] $\leftarrow$ valE R[rA] $\leftarrow$ valM
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Figure 4.20 Computations in sequential implementation of Y86-64 instructions `pushq` **and** `popq`. These instructions push and pop the stack.

Stage	Generic popq rA	Specific popq %rax
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 execution have on the registers and the PC?

Practice Problem 4.15 (solution page 522)

What would be the effect of the instruction `pushq %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?

Stage	`jXX Dest`	`call Dest`	`ret`
Fetch	icode:ifun $\leftarrow$ M_1[PC]	icode:ifun $\leftarrow$ M_1[PC]	icode:ifun $\leftarrow$ M_1[PC]
	valC $\leftarrow$ M_8[PC + 1]	valC $\leftarrow$ M_8[PC + 1]	
	valP $\leftarrow$ PC + 9	valP $\leftarrow$ PC + 9	valP $\leftarrow$ PC + 1
Decode			valA $\leftarrow$ R[`%rsp`]
		valB $\leftarrow$ R[`%rsp`]	valB $\leftarrow$ R[`%rsp`]
Execute		valE $\leftarrow$ valB + (−8)	valE $\leftarrow$ valB + 8
	Cnd $\leftarrow$ Cond(CC, ifun)		
Memory		M_8[valE] $\leftarrow$ valP	valM $\leftarrow$ M_8[valA]
Write back		R[`%rsp`] $\leftarrow$ valE	R[`%rsp`] $\leftarrow$ valE
PC update	PC $\leftarrow$ Cnd ? valC : valP	PC $\leftarrow$ valC	PC $\leftarrow$ valM

Figure 4.21 Computations in sequential implementation of Y86-64 instructions `jXX`, `call`, and `ret`. These instructions cause control transfers.

Practice Problem 4.17 (solution page 522)

We can see by the instruction encodings (Figures 4.2 and 4.3) that the `rrmovq` 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

Stage	`cmovXX` rA, rB
Execute	valE ← $0 + $ valA
	Cnd ← Cond(CC, ifun)
Memory	
Write back	`if` (Cnd) R[rB] ← valE
PC update	PC ← valP

Solution to Problem 4.18 (page 430)

We can see that this instruction is located at address `0x037` and is 9 bytes long. The first byte has value `0x80`, while the last 8 bytes are a byte-reversed version of `0x0000000000000041`, the call target. The stack pointer was set to 128 by the `popq` instruction (line 7).

	Generic	Specific
Stage	`call` Dest	`call 0x041`
Fetch	icode:ifun ← M_1[PC]	icode:ifun ← M_1[`0x037`] = `8:0`
	valC ← M_8[PC + 1]	valC ← M_8[`0x038`] = `0x041`
	valP ← PC + 9	valP ← `0x037` + 9 = `0x040`
Decode		
	valB ← R[`%rsp`]	valB ← R[`%rsp`] = `128`
Execute	valE ← valB + −8	valE ← `128` + −8 = `120`
Memory	M_8[valE] ← valP	M_8[`120`] ← `0x040`
Write back	R[`%rsp`] ← valE	R[`%rsp`] ← `120`
PC update	PC ← valC	PC ← `0x041`

The effect of this instruction is to set `%rsp` to 120, to store `0x040` (the return address) at this memory address, and to set the PC to `0x041` (the call target).

Solution to Problem 4.19 (page 442)

All of the HCL code in this and other practice problems is straightforward, but trying to generate it yourself will help you think about the different instructions and how they are processed. For this problem, we can simply look at the set of Y86-64 instructions (Figure 4.2) and determine which have a constant field.

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

4.3.4 SEQ Stage Implementations

In this section, we devise HCL descriptions for the control logic blocks required to implement SEQ. A complete HCL description for SEQ is given in Web Aside ARCH:HCL on page 508. We show some example blocks here, and others are given as practice problems. We recommend that you work these problems as a way to check your understanding of how the blocks relate to the computational requirements of the different instructions.

Part of the HCL description of SEQ that we do not include here is a definition of the different integer and Boolean signals that can be used as arguments to the HCL operations. These include the names of the different hardware signals, as well as constant values for the different instruction codes, function codes, register names, ALU operations, and status codes. Only those that must be explicitly

Name	Value (hex)	Meaning
`IHALT`	0	Code for `halt` instruction
`INOP`	1	Code for `nop` instruction
`IRRMOVQ`	2	Code for `rrmovq` instruction
`IIRMOVQ`	3	Code for `irmovq` instruction
`IRMMOVQ`	4	Code for `rmmovq` instruction
`IMRMOVQ`	5	Code for `mrmovq` instruction
`IOPL`	6	Code for integer operation instructions
`IJXX`	7	Code for jump instructions
`ICALL`	8	Code for `call` instruction
`IRET`	9	Code for `ret` instruction
`IPUSHQ`	A	Code for `pushq` instruction
`IPOPQ`	B	Code for `popq` instruction
`FNONE`	0	Default function code
`RESP`	4	Register ID for `%rsp`
`RNONE`	F	Indicates no register file access
`ALUADD`	0	Function for addition operation
`SAOK`	1	Status code for normal operation
`SADR`	2	Status code for address exception
`SINS`	3	Status code for illegal instruction exception
`SHLT`	4	Status code for `halt`

Figure 4.26 Constant values used in HCL descriptions. These values represent the encodings of the instructions, function codes, register IDs, ALU operations, and status codes.