Aside Real-life memory design

The memory system in a full-scale microprocessor is far more complex than the simple one we assume in our design. It consists of several forms of hardware memories, including several random access memories, plus nonvolatile memory or magnetic disk, as well as a variety of hardware and software mechanisms for managing these devices. The design and characteristics of the memory system are described in Chapter 6.

Nonetheless, our simple memory design can be used for smaller systems, and it provides us with an abstraction of the interface between the processor and memory for more complex systems.

Our processor includes an additional read-only memory for reading instructions. In most actual systems, these memories are merged into a single memory with two ports: one for reading instructions, and the other for reading or writing data.

4.3 Sequential Y86-64 Implementations

Now we have the components required to implement a Y86-64 processor. As a first step, we describe a processor called SEQ (for "sequential" processor). On each clock cycle, SEQ performs all the steps required to process a complete instruction. This would require a very long cycle time, however, and so the clock rate would be unacceptably low. Our purpose in developing SEQ is to provide a first step toward our ultimate goal of implementing an efficient pipelined processor.

4.3.1 Organizing Processing into Stages

In general, processing an instruction involves a number of operations. We organize them in a particular sequence of stages, attempting to make all instructions follow a uniform sequence, even though the instructions differ greatly in their actions. The detailed processing at each step depends on the particular instruction being executed. Creating this framework will allow us to design a processor that makes best use of the hardware. The following is an informal description of the stages and the operations performed within them:

Fetch. The fetch stage reads the bytes of an instruction from memory, using the program counter (PC) as the memory address. From the instruction it extracts the two 4-bit portions of the instruction specifier byte, referred to as icode (the instruction code) and ifun (the instruction function). It possibly fetches a register specifier byte, giving one or both of the register operand specifiers rA and rB. It also possibly fetches an 8-byte constant word valC. It computes valP to be the address of the instruction following the current one in sequential order. That is, valP equals the value of the PC plus the length of the fetched instruction.

Decode. The decode stage reads up to two operands from the register file, giving values valA and/or valB. Typically, it reads the registers designated by instruction fields rA and rB, but for some instructions it reads register `%rsp`.

Execute. In the execute stage, the arithmetic/logic unit (ALU) either performs the operation specified by the instruction (according to the value of ifun), computes the effective address of a memory reference, or increments or decrements the stack pointer. We refer to the resulting value as valE. The condition codes are possibly set. For a conditional move instruction, the stage will evaluate the condition codes and move condition (given by ifun) and enable the updating of the destination register only if the condition holds. Similarly, for a jump instruction, it determines whether or not the branch should be taken.

Memory. The memory stage may write data to memory, or it may read data from memory. We refer to the value read as valM.

Write back. The write-back stage writes up to two results to the register file.

PC update. The PC is set to the address of the next instruction.

The processor loops indefinitely, performing these stages. In our simplified implementation, the processor will stop when any exception occurs—that is, when it executes a `halt` or invalid instruction, or it attempts to read or write an invalid address. In a more complete design, the processor would enter an exception-handling mode and begin executing special code determined by the type of exception.

As can be seen by the preceding description, there is a surprising amount of processing required to execute a single instruction. Not only must we perform the stated operation of the instruction, we must also compute addresses, update stack pointers, and determine the next instruction address. Fortunately, the overall flow can be similar for every instruction. Using a very simple and uniform structure is important when designing hardware, since we want to minimize the total amount of hardware and we must ultimately map it onto the two-dimensional surface of an integrated-circuit chip. One way to minimize the complexity is to have the different instructions share as much of the hardware as possible. For example, each of our processor designs contains a single arithmetic/logic unit that is used in different ways depending on the type of instruction being executed. The cost of duplicating blocks of logic in hardware is much higher than the cost of having multiple copies of code in software. It is also more difficult to deal with many special cases and idiosyncrasies in a hardware system than with software.

Our challenge is to arrange the computing required for each of the different instructions to fit within this general framework. We will use the code shown in Figure 4.17 to illustrate the processing of different Y86-64 instructions. Figures 4.18 through 4.21 contain tables describing how the different Y86-64 instructions proceed through the stages. It is worth the effort to study these tables carefully. They are in a form that enables a straightforward mapping into the hardware. Each line in these tables describes an assignment to some signal or stored state

```
1   0x000: 30f20900000000000000 |     irmovq $9,  %rdx
2   0x00a: 30f31500000000000000 |     irmovq $21, %rbx
3   0x014: 6123                 |     subq %rdx, %rbx          # subtract
4   0x016: 30f48000000000000000 |     irmovq $128,%rsp         # Problem 4.13
5   0x020: 4043640000000000000  |     rmmovq %rsp, 100(%rbx)   # store
6   0x02a: a02f                 |     pushq %rdx               # push
7   0x02c: b00f                 |     popq  %rax               # Problem 4.14
8   0x02e: 734000000000000000   |     je done                  # Not taken
9   0x037: 804100000000000000   |     call proc                # Problem 4.18
10  0x040:                      | done:
11  0x040: 00                   |     halt
12  0x041:                      | proc:
13  0x041: 90                   |     ret                      # Return
14                              |
```

Figure 4.17 Sample Y86-64 instruction sequence. We will trace the processing of these instructions through the different stages.

(indicated by the assignment operation '←'). These should be read as if they were evaluated in sequence from top to bottom. When we later map the computations to hardware, we will find that we do not need to perform these evaluations in strict sequential order.

Figure 4.18 shows the processing required for instruction types `OPq` (integer and logical operations), `rrmovq` (register-register move), and `irmovq` (immediate-register move). Let us first consider the integer operations. Examining Figure 4.2, we can see that we have carefully chosen an encoding of instructions so that the four integer operations (`addq`, `subq`, `andq`, and `xorq`) all have the same value of icode. We can handle them all by an identical sequence of steps, except that the ALU computation must be set according to the particular instruction operation, encoded in ifun.

The processing of an integer-operation instruction follows the general pattern listed above. In the fetch stage, we do not require a constant word, and so valP is computed as PC + 2. During the decode stage, we read both operands. These are supplied to the ALU in the execute stage, along with the function specifier ifun, so that valE becomes the instruction result. This computation is shown as the expression valB OP valA, where OP indicates the operation specified by ifun. Note the ordering of the two arguments—this order is consistent with the conventions of Y86-64 (and x86-64). For example, the instruction `subq %rax,%rdx` is supposed to compute the value R[%rdx] − R[%rax]. Nothing happens in the memory stage for these instructions, but valE is written to register rB in the write-back stage, and the PC is set to valP to complete the instruction execution.

Executing an `rrmovq` instruction proceeds much like an arithmetic operation. We do not need to fetch the second register operand, however. Instead, we set the second ALU input to zero and add this to the first, giving valE = valA, which is

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

Aside Tracing the execution of a `subq` instruction

As an example, let us follow the processing of the `subq` instruction on line 3 of the object code shown in Figure 4.17. We can see that the previous two instructions initialize registers `%rdx` and `%rbx` to 9 and 21, respectively. We can also see that the instruction is located at address `0x014` and consists of 2 bytes, having values `0x61` and `0x23`. The stages would proceed as shown in the following table, which lists the generic rule for processing an `OPq` instruction (Figure 4.18) on the left, and the computations for this specific instruction on the right.

Stage	`OPq` rA, rB	`subq %rdx, %rbx`
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1]	icode:ifun $\leftarrow$ M_1[0x014] = 6:1 rA:rB $\leftarrow$ M_1[0x015] = 2:3
	valP $\leftarrow$ PC + 2	valP $\leftarrow$ 0x014 + 2 = 0x016
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	valA $\leftarrow$ R[%rdx] = 9 valB $\leftarrow$ R[%rbx] = 21
Execute	valE $\leftarrow$ valB OP valA Set CC	valE $\leftarrow$ 21 − 9 = 12 ZF $\leftarrow$ 0, SF $\leftarrow$ 0, OF $\leftarrow$ 0
Memory		
Write back	R[rB] $\leftarrow$ valE	R[%rbx] $\leftarrow$ valE = 12
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP = 0x016

As this trace shows, we achieve the desired effect of setting register `%rbx` to 12, setting all three condition codes to zero, and incrementing the PC by 2.

Stage	Generic `irmovq` V, rB	Specific `irmovq $128, %rsp`
Memory		
Write back	R[rB] $\leftarrow$ valE	
PC update	PC $\leftarrow$ valP	

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

Figure 4.19 shows the processing required for the memory write and read instructions `rmmovq` and `mrmovq`. We see the same basic flow as before, but using the ALU to add valC to valB, giving the effective address (the sum of the displacement and the base register value) for the memory operation. In the memory stage, we either write the register value valA to memory or read valM from memory.

Stage	`rmmovq` rA, D(rB)	`mrmovq` D(rB), rA
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valC $\leftarrow$ M_8[PC + 2] valP $\leftarrow$ PC + 10	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]	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

Aside Tracing the execution of an `rmmovq` instruction

Let us trace the processing of the `rmmovq` instruction on line 5 of the object code shown in Figure 4.17. We can see that the previous instruction initialized register `%rsp` to 128, while `%rbx` still holds 12, as computed by the `subq` instruction (line 3). We can also see that the instruction is located at address `0x020` and consists of 10 bytes. The first 2 bytes have values `0x40` and `0x43`, while the final 8 bytes are a byte-reversed version of the number `0x0000000000000064` (decimal 100). The stages would proceed as follows:

Stage	Generic `rmmovq` rA, D(rB)	Specific `rmmovq %rsp, 100(%rbx)`
Fetch	icode:ifun ← M_1[PC]	icode:ifun ← M_1[0x020] = 4:0
	rA:rB ← M_1[PC + 1]	rA:rB ← M_1[0x021] = 4:3
	valC ← M_8[PC + 2]	valC ← M_8[0x022] = 100
	valP ← PC + 10	valP ← 0x020 + 10 = 0x02a
Decode	valA ← R[rA]	valA ← R[%rsp] = 128
	valB ← R[rB]	valB ← R[%rbx] = 12
Execute	valE ← valB + valC	valE ← 12 + 100 = 112
Memory	M_8[valE] ← valA	M_8[112] ← 128
Write back		
PC update	PC ← valP	PC ← 0x02a

As this trace shows, the instruction has the effect of writing 128 to memory address 112 and incrementing the PC by 10.

(and x86-64) convention that `popq` should first read memory and then increment the stack pointer.

Practice Problem 4.14 (solution page 522)

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

Stage	Generic `popq` rA	Specific `popq %rax`
Fetch	icode:ifun ← M_1[PC]	
	rA:rB ← M_1[PC + 1]	
	valP ← PC + 2	

Stage	`pushq` rA	`popq` rA
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1]	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1]
	valP $\leftarrow$ PC + 2	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?

Aside Tracing the execution of a `pushq` instruction

Let us trace the processing of the `pushq` instruction on line 6 of the object code shown in Figure 4.17. At this point, we have 9 in register `%rdx` and 128 in register `%rsp`. We can also see that the instruction is located at address `0x02a` and consists of 2 bytes having values `0xa0` and `0x2f`. The stages would proceed as follows:

Stage	Generic `pushq` rA	Specific `pushq %rdx`
Fetch	icode:ifun ← M_1[PC]	icode:ifun ← M_1[0x02a] = a:0
	rA:rB ← M_1[PC + 1]	rA:rB ← M_1[0x02b] = 2:f
	valP ← PC + 2	valP ← 0x02a + 2 = 0x02c
Decode	valA ← R[rA]	valA ← R[%rdx] = 9
	valB ← R[%rsp]	valB ← R[%rsp] = 128
Execute	valE ← valB + (−8)	valE ← 128 + (−8) = 120
Memory	M_8[valE] ← valA	M_8[120] ← 9
Write back	R[%rsp] ← valE	R[%rsp] ← 120
PC update	PC ← valP	PC ← 0x02c

As this trace shows, the instruction has the effect of setting `%rsp` to 120, writing 9 to address 120, and incrementing the PC by 2.

Practice Problem 4.16 (solution page 522)

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?

Figure 4.21 indicates the processing of our three control transfer instructions: the different jumps, `call`, and `ret`. We see that we can implement these instructions with the same overall flow as the preceding ones.

As with integer operations, we can process all of the jumps in a uniform manner, since they differ only when determining whether or not to take the branch. A jump instruction proceeds through fetch and decode much like the previous instructions, except that it does not require a register specifier byte. In the execute stage, we check the condition codes and the jump condition to determine whether or not to take the branch, yielding a 1-bit signal Cnd. During the PC update stage, we test this flag and set the PC to valC (the jump target) if the flag is 1 and to valP (the address of the following instruction) if the flag is 0. Our notation $x\ ?\ a : b$ is similar to the conditional expression in C—it yields a when x is 1 and b when x is 0.

Stage	`jXX` Dest	`call` Dest	`ret`
Fetch	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]
	valC ← M_8[PC + 1]	valC ← M_8[PC + 1]	
	valP ← PC + 9	valP ← PC + 9	valP ← PC + 1
Decode			valA ← R[%rsp]
		valB ← R[%rsp]	valB ← R[%rsp]
Execute		valE ← valB + (−8)	valE ← valB + 8
	Cnd ← Cond(CC, ifun)		
Memory		M_8[valE] ← valP	valM ← M_8[valA]
Write back		R[%rsp] ← valE	R[%rsp] ← valE
PC update	PC ← Cnd ? valC : valP	PC ← valC	PC ← 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 ← M_1[PC]
	rA:rB ← M_1[PC + 1]
	valP ← PC + 2
Decode	valA ← R[rA]
Execute	valE ← 0 + valA
Memory	
Write back	
	R[rB] ← valE
PC update	PC ← valP

Aside Tracing the execution of a `je` instruction

Let us trace the processing of the `je` instruction on line 8 of the object code shown in Figure 4.17. The condition codes were all set to zero by the `subq` instruction (line 3), and so the branch will not be taken. The instruction is located at address `0x02e` and consists of 9 bytes. The first has value `0x73`, while the remaining 8 bytes are a byte-reversed version of the number `0x0000000000000040`, the jump target. The stages would proceed as follows:

Stage	Generic `jXX` Dest	Specific `je 0x040`
Fetch	icode:ifun $\leftarrow$ M_1[PC]	icode:ifun $\leftarrow$ M_1[0x02e] = 7:3
	valC $\leftarrow$ M_8[PC + 1]	valC $\leftarrow$ M_8[0x02f] = 0x040
	valP $\leftarrow$ PC + 9	valP $\leftarrow$ 0x02e + 9 = 0x037
Decode		
Execute		
	Cnd $\leftarrow$ Cond(CC, ifun)	Cnd $\leftarrow$ Cond($\langle 0, 0, 0 \rangle$, 3) = 0
Memory		
Write back		
PC update	PC $\leftarrow$ Cnd ? valC : valP	PC $\leftarrow$ 0 ? 0x040 : 0x037 = 0x037

As this trace shows, the instruction has the effect of incrementing the PC by 9.

Instructions `call` and `ret` bear some similarity to instructions `pushq` and `popq`, except that we push and pop program counter values. With instruction `call`, we push valP, the address of the instruction that follows the `call` instruction. During the PC update stage, we set the PC to valC, the call destination. With instruction `ret`, we assign valM, the value popped from the stack, to the PC in the PC update stage.

Practice Problem 4.18 (solution page 523)

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	

Aside Tracing the execution of a `ret` instruction

Let us trace the processing of the `ret` instruction on line 13 of the object code shown in Figure 4.17. The instruction address is `0x041` and is encoded by a single byte `0x90`. The previous `call` instruction set `%rsp` to 120 and stored the return address `0x040` at memory address 120. The stages would proceed as follows:

Stage	Generic `ret`	Specific `ret`
Fetch	icode:ifun $\leftarrow$ M_1[PC]	icode:ifun $\leftarrow$ M_1[0x041] = 9:0
	valP $\leftarrow$ PC + 1	valP $\leftarrow$ 0x041 + 1 = 0x042
Decode	valA $\leftarrow$ R[%rsp]	valA $\leftarrow$ R[%rsp] = 120
	valB $\leftarrow$ R[%rsp]	valB $\leftarrow$ R[%rsp] = 120
Execute	valE $\leftarrow$ valB + 8	valE $\leftarrow$ 120 + 8 = 128
Memory	valM $\leftarrow$ M_8[valA]	valM $\leftarrow$ M_8[120] = 0x040
Write back	R[%rsp] $\leftarrow$ valE	R[%rsp] $\leftarrow$ 128
PC update	PC $\leftarrow$ valM	PC $\leftarrow$ 0x040

As this trace shows, the instruction has the effect of setting the PC to `0x040`, the address of the `halt` instruction. It also sets `%rsp` to 128.

Stage	Generic `call` Dest	Specific `call 0x041`
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?

We have created a uniform framework that handles all of the different types of Y86-64 instructions. Even though the instructions have widely varying behavior, we can organize the processing into six stages. Our task now is to create a hardware design that implements the stages and connects them together.

4.3.2 SEQ Hardware Structure

The computations required to implement all of the Y86-64 instructions can be organized as a series of six basic stages: fetch, decode, execute, memory, write back, and PC update. Figure 4.22 shows an abstract view of a hardware structure that can perform these computations. The program counter is stored in a register, shown in the lower left-hand corner (labeled "PC"). Information then flows along wires (shown grouped together as a heavy gray line), first upward and then around to the right. Processing is performed by *hardware units* associated with the different stages. The feedback paths coming back down on the right-hand side contain the updated values to write to the register file and the updated program counter. In SEQ, all of the processing by the hardware units occurs within a single clock cycle, as is discussed in Section 4.3.3. This diagram omits some small blocks of combinational logic as well as all of the control logic needed to operate the different hardware units and to route the appropriate values to the units. We will add this detail later. Our method of drawing processors with the flow going from bottom to top is unconventional. We will explain the reason for this convention when we start designing pipelined processors.

The hardware units are associated with the different processing stages:

Fetch. Using the program counter register as an address, the instruction memory reads the bytes of an instruction. The PC incrementer computes valP, the incremented program counter.

Decode. The register file has two read ports, A and B, via which register values valA and valB are read simultaneously.

Execute. The execute stage uses the arithmetic/logic (ALU) unit for different purposes according to the instruction type. For integer operations, it performs the specified operation. For other instructions, it serves as an adder to compute an incremented or decremented stack pointer, to compute an effective address, or simply to pass one of its inputs to its outputs by adding zero.

The condition code register (CC) holds the three condition code bits. New values for the condition codes are computed by the ALU. When executing a conditional move instruction, the decision as to whether or not to update the destination register is computed based on the condition codes and move condition. Similarly, when executing a jump instruction, the branch signal Cnd is computed based on the condition codes and the jump type.

Memory. The data memory reads or writes a word of memory when executing a memory instruction. The instruction and data memories access the same memory locations, but for different purposes.

Write back. The register file has two write ports. Port E is used to write values computed by the ALU, while port M is used to write values read from the data memory.

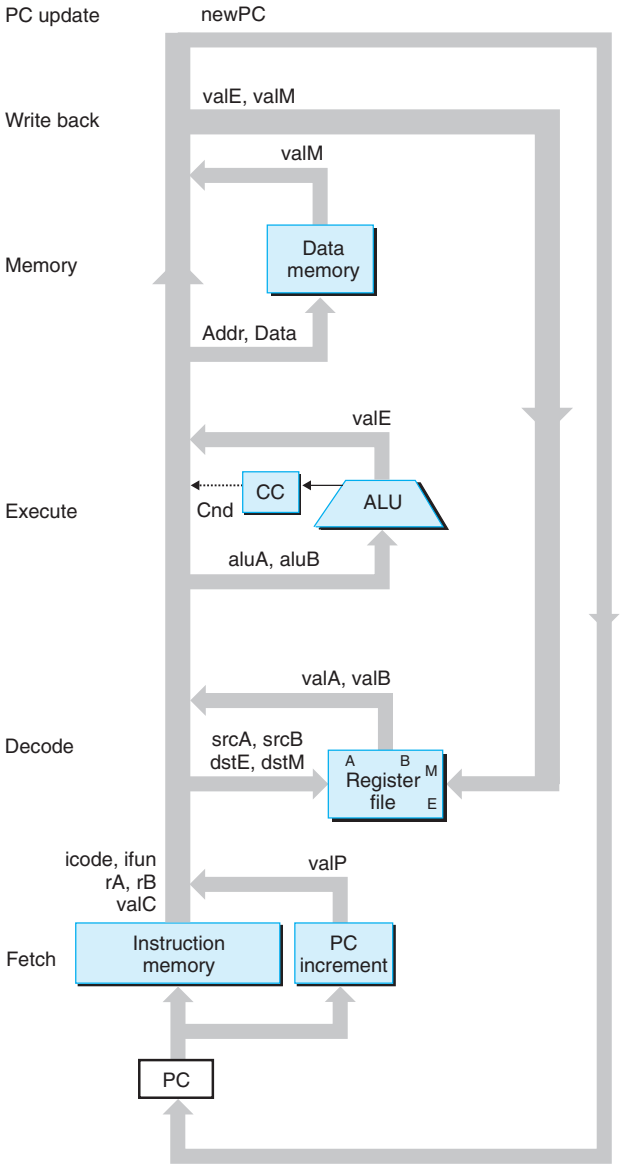

Figure 4.22 Abstract view of SEQ, a sequential implementation. The information processed during execution of an instruction follows a clockwise flow starting with an instruction fetch using the program counter (PC), shown in the lower left-hand corner of the figure.

PC update. The new value of the program counter is selected to be either valP, the address of the next instruction, valC, the destination address specified by a call or jump instruction, or valM, the return address read from memory.

Figure 4.23 gives a more detailed view of the hardware required to implement SEQ (although we will not see the complete details until we examine the individual stages). We see the same set of hardware units as earlier, but now the wires are shown explicitly. In this figure, as well as in our other hardware diagrams, we use the following drawing conventions:

- *Clocked registers are shown as white rectangles.* The program counter PC is the only clocked register in SEQ.
- *Hardware units are shown as light blue boxes.* These include the memories, the ALU, and so forth. We will use the same basic set of units for all of our processor implementations. We will treat these units as "black boxes" and not go into their detailed designs.
- *Control logic blocks are drawn as gray rounded rectangles.* These blocks serve to select from among a set of signal sources or to compute some Boolean function. We will examine these blocks in complete detail, including developing HCL descriptions.
- *Wire names are indicated in white circles.* These are simply labels on the wires, not any kind of hardware element.
- *Word-wide data connections are shown as medium lines.* Each of these lines actually represents a bundle of 64 wires, connected in parallel, for transferring a word from one part of the hardware to another.
- *Byte and narrower data connections are shown as thin lines.* Each of these lines actually represents a bundle of four or eight wires, depending on what type of values must be carried on the wires.
- *Single-bit connections are shown as dotted lines.* These represent control values passed between the units and blocks on the chip.

All of the computations we have shown in Figures 4.18 through 4.21 have the property that each line represents either the computation of a specific value, such as valP, or the activation of some hardware unit, such as the memory. These computations and actions are listed in the second column of Figure 4.24. In addition to the signals we have already described, this list includes four register ID signals: srcA, the source of valA; srcB, the source of valB; dstE, the register to which valE gets written; and dstM, the register to which valM gets written.

The two right-hand columns of this figure show the computations for the `OPq` and `mrmovq` instructions to illustrate the values being computed. To map the computations into hardware, we want to implement control logic that will transfer the data between the different hardware units and operate these units in such a way that the specified operations are performed for each of the different instruction types. That is the purpose of the control logic blocks, shown as gray rounded boxes

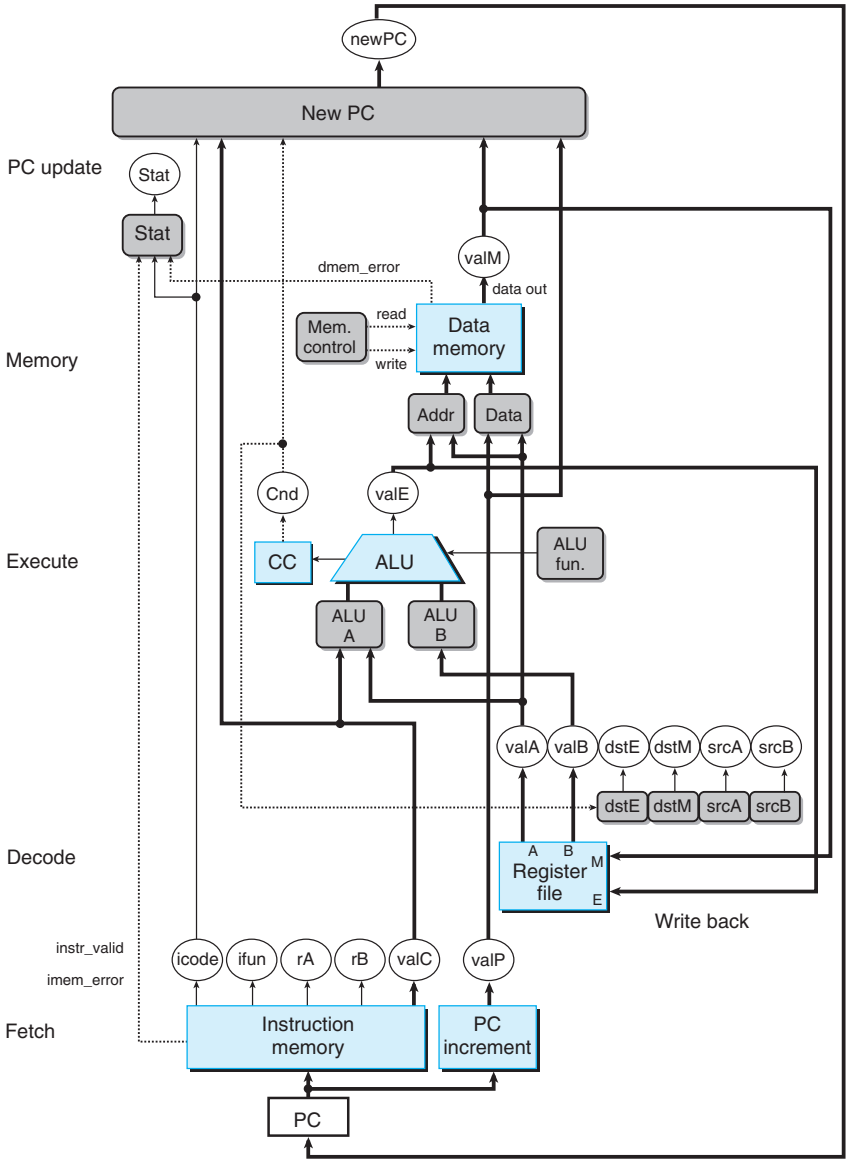

Figure 4.23 Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.

Stage	Computation	OPq rA, rB	mrmovq D(rB), rA
Fetch	icode, ifun	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]
	rA, rB	rA:rB ← M_1[PC + 1]	rA:rB ← M_1[PC + 1]
	valC		valC ← M_8[PC + 2]
	valP	valP ← PC + 2	valP ← PC + 10
Decode	valA, srcA	valA ← R[rA]	
	valB, srcB	valB ← R[rB]	valB ← R[rB]
Execute	valE	valE ← valB OP valA	valE ← valB + valC
	Cond. codes	Set CC	
Memory	Read/write		valM ← M_8[valE]
Write back	E port, dstE	R[rB] ← valE	
	M port, dstM		R[rA] ← valM
PC update	PC	PC ← valP	PC ← valP

Figure 4.24 Identifying the different computation steps in the sequential implementation. The second column identifies the value being computed or the operation being performed in the stages of SEQ. The computations for instructions OPq and mrmovq are shown as examples of the computations.

in Figure 4.23. Our task is to proceed through the individual stages and create detailed designs for these blocks.

4.3.3 SEQ Timing

In introducing the tables of Figures 4.18 through 4.21, we stated that they should be read as if they were written in a programming notation, with the assignments performed in sequence from top to bottom. On the other hand, the hardware structure of Figure 4.23 operates in a fundamentally different way, with a single clock transition triggering a flow through combinational logic to execute an entire instruction. Let us see how the hardware can implement the behavior listed in these tables.

Our implementation of SEQ consists of combinational logic and two forms of memory devices: clocked registers (the program counter and condition code register) and random access memories (the register file, the instruction memory, and the data memory). Combinational logic does not require any sequencing or control—values propagate through a network of logic gates whenever the inputs change. As we have described, we also assume that reading from a random access memory operates much like combinational logic, with the output word generated based on the address input. This is a reasonable assumption for smaller

memories (such as the register file), and we can mimic this effect for larger circuits using special clock circuits. Since our instruction memory is only used to read instructions, we can therefore treat this unit as if it were combinational logic.

We are left with just four hardware units that require an explicit control over their sequencing—the program counter, the condition code register, the data memory, and the register file. These are controlled via a single clock signal that triggers the loading of new values into the registers and the writing of values to the random access memories. The program counter is loaded with a new instruction address every clock cycle. The condition code register is loaded only when an integer operation instruction is executed. The data memory is written only when an `rmmovq`, `pushq`, or `call` instruction is executed. The two write ports of the register file allow two program registers to be updated on every cycle, but we can use the special register ID `0xF` as a port address to indicate that no write should be performed for this port.

This clocking of the registers and memories is all that is required to control the sequencing of activities in our processor. Our hardware achieves the same effect as would a sequential execution of the assignments shown in the tables of Figures 4.18 through 4.21, even though all of the state updates actually occur simultaneously and only as the clock rises to start the next cycle. This equivalence holds because of the nature of the Y86-64 instruction set, and because we have organized the computations in such a way that our design obeys the following principle:

PRINCIPLE: No reading back

The processor never needs to read back the state updated by an instruction in order to complete the processing of this instruction. ■

This principle is crucial to the success of our implementation. As an illustration, suppose we implemented the `pushq` instruction by first decrementing `%rsp` by 8 and then using the updated value of `%rsp` as the address of a write operation. This approach would violate the principle stated above. It would require reading the updated stack pointer from the register file in order to perform the memory operation. Instead, our implementation (Figure 4.20) generates the decremented value of the stack pointer as the signal valE and then uses this signal both as the data for the register write and the address for the memory write. As a result, it can perform the register and memory writes simultaneously as the clock rises to begin the next clock cycle.

As another illustration of this principle, we can see that some instructions (the integer operations) set the condition codes, and some instructions (the conditional move and jump instructions) read these condition codes, but no instruction must both set and then read the condition codes. Even though the condition codes are not set until the clock rises to begin the next clock cycle, they will be updated before any instruction attempts to read them.

Figure 4.25 shows how the SEQ hardware would process the instructions at lines 3 and 4 in the following code sequence, shown in assembly code with the instruction addresses listed on the left:

```
1      0x000:   irmovq $0x100,%rbx   # %rbx <-- 0x100
2      0x00a:   irmovq $0x200,%rdx   # %rdx <-- 0x200
3      0x014:   addq %rdx,%rbx       # %rbx <-- 0x300 CC <-- 000
4      0x016:   je dest              # Not taken
5      0x01f:   rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300
6      0x029: dest: halt
```

Each of the diagrams labeled 1 through 4 shows the four state elements plus the combinational logic and the connections among the state elements. We show the combinational logic as being wrapped around the condition code register, because some of the combinational logic (such as the ALU) generates the input to the condition code register, while other parts (such as the branch computation and the PC selection logic) have the condition code register as input. We show the register file and the data memory as having separate connections for reading and writing, since the read operations propagate through these units as if they were combinational logic, while the write operations are controlled by the clock.

The color coding in Figure 4.25 indicates how the circuit signals relate to the different instructions being executed. We assume the processing starts with the condition codes, listed in the order `ZF`, `SF`, and `OF`, set to `100`. At the beginning of clock cycle 3 (point 1), the state elements hold the state as updated by the second `irmovq` instruction (line 2 of the listing), shown in light gray. The combinational logic is shown in white, indicating that it has not yet had time to react to the changed state. The clock cycle begins with address `0x014` loaded into the program counter. This causes the `addq` instruction (line 3 of the listing), shown in blue, to be fetched and processed. Values flow through the combinational logic, including the reading of the random access memories. By the end of the cycle (point 2), the combinational logic has generated new values (`000`) for the condition codes, an update for program register `%rbx`, and a new value (`0x016`) for the program counter. At this point, the combinational logic has been updated according to the `addq` instruction (shown in blue), but the state still holds the values set by the second `irmovq` instruction (shown in light gray).

As the clock rises to begin cycle 4 (point 3), the updates to the program counter, the register file, and the condition code register occur, and so we show these in blue, but the combinational logic has not yet reacted to these changes, and so we show this in white. In this cycle, the `je` instruction (line 4 in the listing), shown in dark gray, is fetched and executed. Since condition code `ZF` is 0, the branch is not taken. By the end of the cycle (point 4), a new value of `0x01f` has been generated for the program counter. The combinational logic has been updated according to the `je` instruction (shown in dark gray), but the state still holds the values set by the `addq` instruction (shown in blue) until the next cycle begins.

As this example illustrates, the use of a clock to control the updating of the state elements, combined with the propagation of values through combinational logic, suffices to control the computations performed for each instruction in our implementation of SEQ. Every time the clock transitions from low to high, the processor begins executing a new instruction.

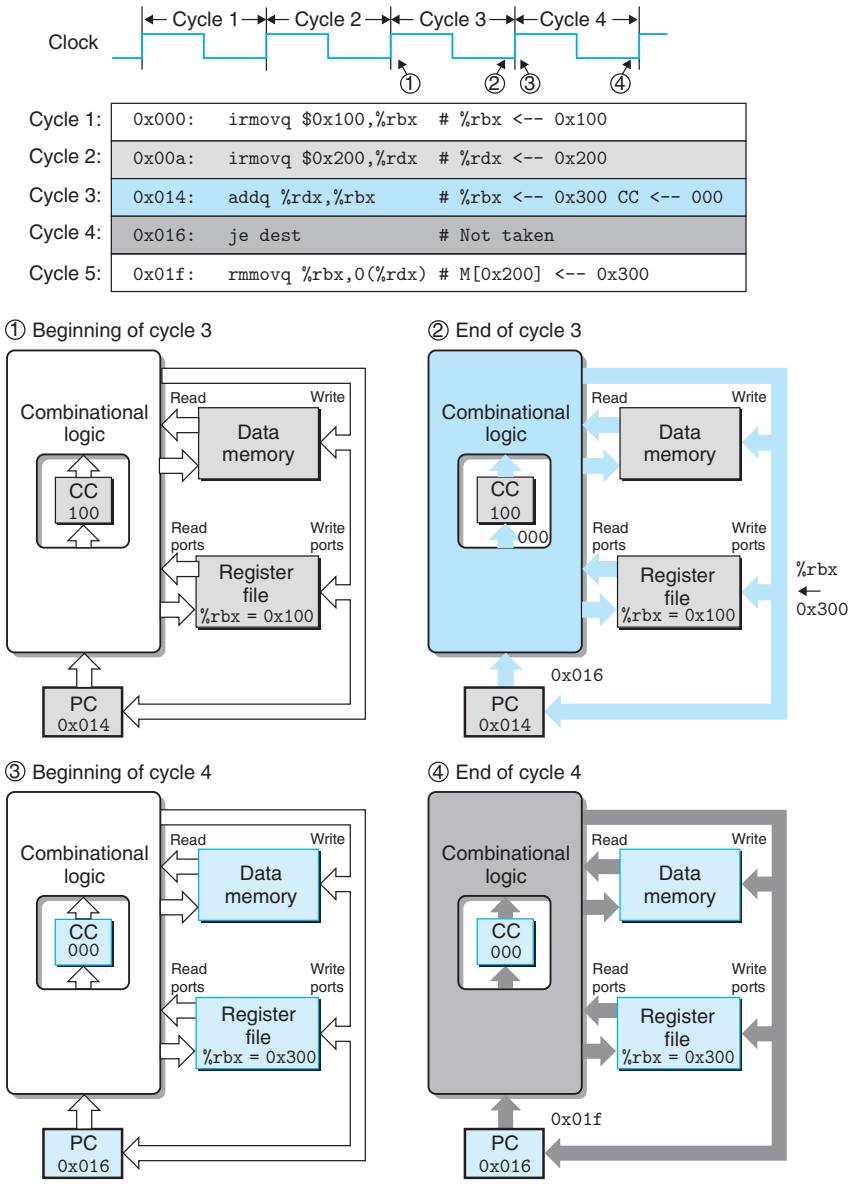

Figure 4.25 Tracing two cycles of execution by SEQ. Each cycle begins with the state elements (program counter, condition code register, register file, and data memory) set according to the previous instruction. Signals propagate through the combinational logic, creating new values for the state elements. These values are loaded into the state elements to start the next cycle.

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.

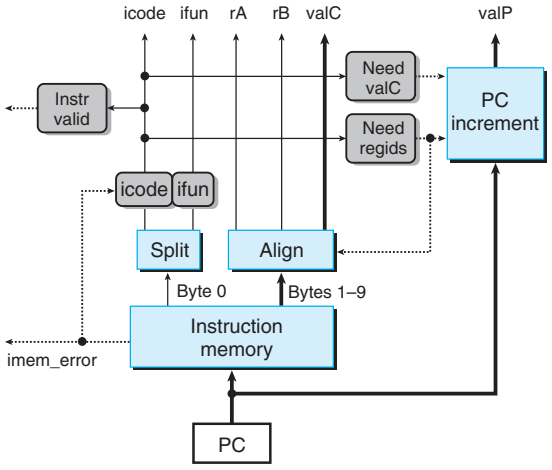

Figure 4.27
SEQ fetch stage. Six bytes are read from the instruction memory using the PC as the starting address. From these bytes, we generate the different instruction fields. The PC increment block computes signal valP.

referenced in the control logic are shown. The constants we use are documented in Figure 4.26. By convention, we use uppercase names for constant values.

In addition to the instructions shown in Figures 4.18 to 4.21, we include the processing for the `nop` and `halt` instructions. The `nop` instruction simply flows through stages without much processing, except to increment the PC by 1. The `halt` instruction causes the processor status to be set to `HLT`, causing it to halt operation.

Fetch Stage

As shown in Figure 4.27, the fetch stage includes the instruction memory hardware unit. This unit reads 10 bytes from memory at a time, using the PC as the address of the first byte (byte 0). This byte is interpreted as the instruction byte and is split (by the unit labeled "Split") into two 4-bit quantities. The control logic blocks labeled "icode" and "ifun" then compute the instruction and function codes as equaling either the values read from memory or, in the event that the instruction address is not valid (as indicated by the signal imem_error), the values corresponding to a `nop` instruction. Based on the value of icode, we can compute three 1-bit signals (shown as dashed lines):

instr_valid. Does this byte correspond to a legal Y86-64 instruction? This signal is used to detect an illegal instruction.

need_regids. Does this instruction include a register specifier byte?

need_valC. Does this instruction include a constant word?

The signals instr_valid and imem_error (generated when the instruction address is out of bounds) are used to generate the status code in the memory stage.

As an example, the HCL description for need_regids simply determines whether the value of icode is one of the instructions that has a register specifier byte:

```
bool need_regids =
        icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                   IIRMOVQ, IRMMOVQ, IMRMOVQ };
```

Practice Problem 4.19 (solution page 523)

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

As Figure 4.27 shows, the remaining 9 bytes read from the instruction memory encode some combination of the register specifier byte and the constant word. These bytes are processed by the hardware unit labeled "Align" into the register fields and the constant word. Byte 1 is split into register specifiers rA and rB when the computed signal need_regids is 1. If need_regids is 0, both register specifiers are set to `0xF` (`RNONE`), indicating there are no registers specified by this instruction. Recall also (Figure 4.2) that for any instruction having only one register operand, the other field of the register specifier byte will be `0xF` (`RNONE`). Thus, we can assume that the signals rA and rB either encode registers we want to access or indicate that register access is not required. The unit labeled "Align" also generates the constant word valC. This will either be bytes 1–8 or bytes 2–9, depending on the value of signal need_regids.

The PC incrementer hardware unit generates the signal valP, based on the current value of the PC, and the two signals need_regids and need_valC. For PC value p, need_regids value r, and need_valC value i, the incrementer generates the value $p + 1 + r + 8i$.

Decode and Write-Back Stages

Figure 4.28 provides a detailed view of logic that implements both the decode and write-back stages in SEQ. These two stages are combined because they both access the register file.

The register file has four ports. It supports up to two simultaneous reads (on ports A and B) and two simultaneous writes (on ports E and M). Each port has both an address connection and a data connection, where the address connection is a register ID, and the data connection is a set of 64 wires serving as either an output word (for a read port) or an input word (for a write port) of the register file. The two read ports have address inputs srcA and srcB, while the two write ports have address inputs dstE and dstM. The special identifier `0xF` (`RNONE`) on an address port indicates that no register should be accessed.

The four blocks at the bottom of Figure 4.28 generate the four different register IDs for the register file, based on the instruction code icode, the register specifiers rA and rB, and possibly the condition signal Cnd computed in the execute stage. Register ID srcA indicates which register should be read to generate valA.

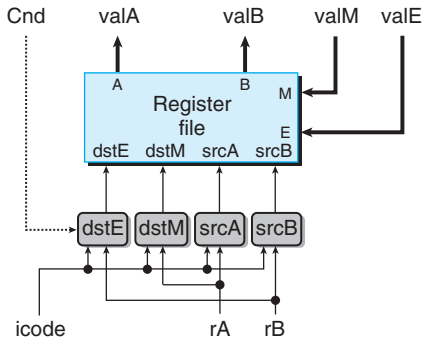

Figure 4.28
SEQ decode and write-back stage. The instruction fields are decoded to generate register identifiers for four addresses (two read and two write) used by the register file. The values read from the register file become the signals valA and valB. The two write-back values valE and valM serve as the data for the writes.

The desired value depends on the instruction type, as shown in the first row for the decode stage in Figures 4.18 to 4.21. Combining all of these entries into a single computation gives the following HCL description of srcA (recall that RESP is the register ID of %rsp):

```
word srcA = [
        icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
        icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];
```

Practice Problem 4.20 (solution page 524)

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 decode stage in Figures 4.18 to 4.21. Write HCL code for srcB.

Register ID dstE indicates the destination register for write port E, where the computed value valE is stored. This is shown in Figures 4.18 to 4.21 as the first step in the write-back stage. If we ignore for the moment the conditional move instructions, then we can combine the destination registers for all of the different instructions to give the following HCL description of dstE:

```
# 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
];
```

We will revisit this signal and how to implement conditional moves when we examine the execute stage.

Practice Problem 4.21 (solution page 524)

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.

Practice Problem 4.22 (solution page 524)

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

Execute Stage

The execute stage includes the arithmetic/logic unit (ALU). This unit performs the operation ADD, SUBTRACT, AND, or EXCLUSIVE-OR on inputs aluA and aluB based on the setting of the alufun signal. These data and control signals are generated by three control blocks, as diagrammed in Figure 4.29. The ALU output becomes the signal valE.

In Figures 4.18 to 4.21, the ALU computation for each instruction is shown as the first step in the execute stage. The operands are listed with aluB first, followed by aluA to make sure the `subq` instruction subtracts valA from valB. We can see that the value of aluA can be valA, valC, or either -8 or $+8$, depending on the instruction type. We can therefore express the behavior of the control block that generates aluA as follows:

```
word aluA = [
        icode in { IRRMOVQ, IOPQ } : valA;
        icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
```

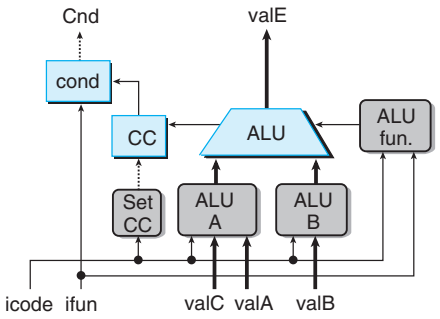

Figure 4.29
SEQ execute stage. The ALU either performs the operation for an integer operation instruction or acts as an adder. The condition code registers are set according to the ALU value. The condition code values are tested to determine whether a branch should be taken.

```
        icode in { ICALL, IPUSHQ } : -8;
        icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
];
```

Practice Problem 4.23 (solution page 524)

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.

Looking at the operations performed by the ALU in the execute stage, we can see that it is mostly used as an adder. For the OPq instructions, however, we want it to use the operation encoded in the ifun field of the instruction. We can therefore write the HCL description for the ALU control as follows:

```
word alufun = [
        icode == IOPQ : ifun;
        1 : ALUADD;
];
```

The execute stage also includes the condition code register. Our ALU generates the three signals on which the condition codes are based—zero, sign, and overflow—every time it operates. However, we only want to set the condition codes when an OPq instruction is executed. We therefore generate a signal set_cc that controls whether or not the condition code register should be updated:

```
bool set_cc = icode in { IOPQ };
```

The hardware unit labeled "cond" uses a combination of the condition codes and the function code to determine whether a conditional branch or data transfer should take place (Figure 4.3). It generates the Cnd signal used both for the setting of dstE with conditional moves and in the next PC logic for conditional branches. For other instructions, the Cnd signal may be set to either 1 or 0, depending on the instruction's function code and the setting of the condition codes, but it will be ignored by the control logic. We omit the detailed design of this unit.

Practice Problem 4.24 (solution page 524)

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.

Memory Stage

The memory stage has the task of either reading or writing program data. As shown in Figure 4.30, two control blocks generate the values for the memory

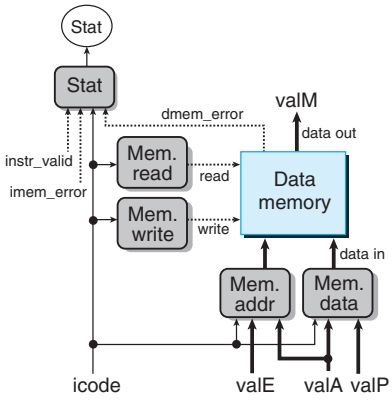

Figure 4.30
SEQ memory stage. The data memory can either write or read memory values. The value read from memory forms the signal valM.

address and the memory input data (for write operations). Two other blocks generate the control signals indicating whether to perform a read or a write operation. When a read operation is performed, the data memory generates the value valM.

The desired memory operation for each instruction type is shown in the memory stage of Figures 4.18 to 4.21. Observe that the address for memory reads and writes is always valE or valA. We can describe this block in HCL as follows:

```
word mem_addr = [
        icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
        icode in { IPOPQ, IRET } : valA;
        # Other instructions don't need address
];
```

Practice Problem 4.25 (solution page 524)

Looking at the 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.

We want to set the control signal mem_read only for instructions that read data from memory, as expressed by the following HCL code:

```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
```

Practice Problem 4.26 (solution page 525)

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

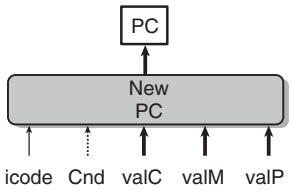

Figure 4.31
SEQ PC update stage. The next value of the PC is selected from among the signals valC, valM, and valP, depending on the instruction code and the branch flag.

A final function for the memory stage is to compute the status code Stat resulting from the instruction execution according to the values of icode, imem_error, and instr_valid generated in the fetch stage and the signal dmem_error generated by the data memory.

Practice Problem 4.27 **(solution page 525)**

Write HCL code for Stat, generating the four status codes SAOK, SADR, SINS, and SHLT (see Figure 4.26).

PC Update Stage

The final stage in SEQ generates the new value of the program counter (see Figure 4.31). As the final steps in Figures 4.18 to 4.21 show, the new PC will be valC, valM, or valP, depending on the instruction type and whether or not a branch should be taken. This selection can be described in HCL as follows:

```
word new_pc = [
        # Call.  Use instruction constant
        icode == ICALL : valC;
        # Taken branch.  Use instruction constant
        icode == IJXX && Cnd : valC;
        # Completion of RET instruction.  Use value from stack
        icode == IRET : valM;
        # Default: Use incremented PC
        1 : valP;
];
```

Surveying SEQ

We have now stepped through a complete design for a Y86-64 processor. We have seen that by organizing the steps required to execute each of the different instructions into a uniform flow, we can implement the entire processor with a small number of different hardware units and with a single clock to control the sequencing of computations. The control logic must then route the signals between these units and generate the proper control signals based on the instruction types and the branch conditions.

The only problem with SEQ is that it is too slow. The clock must run slowly enough so that signals can propagate through all of the stages within a single cycle. As an example, consider the processing of a `ret` instruction. Starting with an updated program counter at the beginning of the clock cycle, the instruction must be read from the instruction memory, the stack pointer must be read from the register file, the ALU must increment the stack pointer by 8, and the return address must be read from the memory in order to determine the next value for the program counter. All of these must be completed by the end of the clock cycle.

This style of implementation does not make very good use of our hardware units, since each unit is only active for a fraction of the total clock cycle. We will see that we can achieve much better performance by introducing pipelining.

4.4 General Principles of Pipelining

Before attempting to design a pipelined Y86-64 processor, let us consider some general properties and principles of pipelined systems. Such systems are familiar to anyone who has been through the serving line at a cafeteria or run a car through an automated car wash. In a pipelined system, the task to be performed is divided into a series of discrete stages. In a cafeteria, this involves supplying salad, a main dish, dessert, and beverage. In a car wash, this involves spraying water and soap, scrubbing, applying wax, and drying. Rather than having one customer run through the entire sequence from beginning to end before the next can begin, we allow multiple customers to proceed through the system at once. In a traditional cafeteria line, the customers maintain the same order in the pipeline and pass through all stages, even if they do not want some of the courses. In the case of the car wash, a new car is allowed to enter the spraying stage as the preceding car moves from the spraying stage to the scrubbing stage. In general, the cars must move through the system at the same rate to avoid having one car crash into the next.

A key feature of pipelining is that it increases the *throughput* of the system (i.e., the number of customers served per unit time), but it may also slightly increase the *latency* (i.e., the time required to service an individual customer). For example, a customer in a cafeteria who only wants a dessert could pass through a nonpipelined system very quickly, stopping only at the dessert stage. A customer in a pipelined system who attempts to go directly to the dessert stage risks incurring the wrath of other customers.

4.4.1 Computational Pipelines

Shifting our focus to computational pipelines, the "customers" are instructions and the stages perform some portion of the instruction execution. Figure 4.32(a) shows an example of a simple nonpipelined hardware system. It consists of some logic that performs a computation, followed by a register to hold the results of this computation. A clock signal controls the loading of the register at some regular time interval. An example of such a system is the decoder in a compact disk (CD) player. The incoming signals are the bits read from the surface of the CD, and