CHAPTER 4

Processor Architecture

Modern microprocessors are among the most complex systems ever created by humans. A single silicon chip, roughly the size of a fingernail, can contain several high-performance processors, large cache memories, and the logic required to interface them to external devices. In terms of performance, the processors implemented on a single chip today dwarf the room-size supercomputers that cost over $10 million just 20 years ago. Even the embedded processors found in everyday appliances such as cell phones, navigation systems, and programmable thermostats are far more powerful than the early developers of computers could ever have envisioned.

So far, we have only viewed computer systems down to the level of machine-language programs. We have seen that a processor must execute a sequence of instructions, where each instruction performs some primitive operation, such as adding two numbers. An instruction is encoded in binary form as a sequence of 1 or more bytes. The instructions supported by a particular processor and their byte-level encodings are known as its *instruction set architecture* (ISA). Different "families" of processors, such as Intel IA32 and x86-64, IBM/Freescale Power, and the ARM processor family, have different ISAs. A program compiled for one type of machine will not run on another. On the other hand, there are many different models of processors within a single family. Each manufacturer produces processors of ever-growing performance and complexity, but the different models remain compatible at the ISA level. Popular families, such as x86-64, have processors supplied by multiple manufacturers. Thus, the ISA provides a conceptual layer of abstraction between compiler writers, who need only know what instructions are permitted and how they are encoded, and processor designers, who must build machines that execute those instructions.

In this chapter, we take a brief look at the design of processor hardware. We study the way a hardware system can execute the instructions of a particular ISA. This view will give you a better understanding of how computers work and the technological challenges faced by computer manufacturers. One important concept is that the actual way a modern processor operates can be quite different from the model of computation implied by the ISA. The ISA model would seem to imply *sequential* instruction execution, where each instruction is fetched and executed to completion before the next one begins. By executing different parts of multiple instructions simultaneously, the processor can achieve higher performance than if it executed just one instruction at a time. Special mechanisms are used to make sure the processor computes the same results as it would with sequential execution. This idea of using clever tricks to improve performance while maintaining the functionality of a simpler and more abstract model is well known in computer science. Examples include the use of caching in Web browsers and information retrieval data structures such as balanced binary trees and hash tables.

Chances are you will never design your own processor. This is a task for experts working at fewer than 100 companies worldwide. Why, then, should you learn about processor design?

- *It is intellectually interesting and important.* There is an intrinsic value in learning how things work. It is especially interesting to learn the inner workings of

Aside The progress of computer technology

To get a sense of how much computer technology has improved over the past four decades, consider the following two processors.

The first Cray 1 supercomputer was delivered to Los Alamos National Laboratory in 1976. It was the fastest computer in the world, able to perform as many as 250 million arithmetic operations per second. It came with 8 megabytes of random access memory, the maximum configuration allowed by the hardware. The machine was also very large—it weighed 5,000 kg, consumed 115 kilowatts, and cost $9 million. In total, around 80 of them were manufactured.

The Apple ARM A7 microprocessor chip, introduced in 2013 to power the iPhone 5S, contains two CPUs, each of which can perform several billion arithmetic operations per second, and 1 gigabyte of random access memory. The entire phone weighs just 112 grams, consumes around 1 watt, and costs less than $800. Over 9 million units were sold in the first weekend of its introduction. In addition to being a powerful computer, it can be used to take pictures, to place phone calls, and to provide driving directions, features never considered for the Cray 1.

These two systems, spaced just 37 years apart, demonstrate the tremendous progress of semiconductor technology. Whereas the Cray 1's CPU was constructed using around 100,000 semiconductor chips, each containing less than 20 transistors, the Apple A7 has over 1 billion transistors on its single chip. The Cray 1's 8-megabyte memory required 8,192 chips, whereas the iPhone's gigabyte memory is contained in a single chip.

a system that is such a part of the daily lives of computer scientists and engineers and yet remains a mystery to many. Processor design embodies many of the principles of good engineering practice. It requires creating a simple and regular structure to perform a complex task.

- *Understanding how the processor works aids in understanding how the overall computer system works.* In Chapter 6, we will look at the memory system and the techniques used to create an image of a very large memory with a very fast access time. Seeing the processor side of the processor–memory interface will make this presentation more complete.
- *Although few people design processors, many design hardware systems that contain processors.* This has become commonplace as processors are embedded into real-world systems such as automobiles and appliances. Embedded-system designers must understand how processors work, because these systems are generally designed and programmed at a lower level of abstraction than is the case for desktop and server-based systems.
- *You just might work on a processor design.* Although the number of companies producing microprocessors is small, the design teams working on those processors are already large and growing. There can be over 1,000 people involved in the different aspects of a major processor design.

In this chapter, we start by defining a simple instruction set that we use as a running example for our processor implementations. We call this the "Y86-64" instruction set, because it was inspired by the x86-64 instruction set. Compared with x86-64, the Y86-64 instruction set has fewer data types, instructions, and addressing modes. It also has a simple byte-level encoding, making the machine code less compact than the comparable x86-64 code, but also much easier to design the CPU's decoding logic. Even though the Y86-64 instruction set is very simple, it is sufficiently complete to allow us to write programs manipulating integer data. Designing a processor to implement Y86-64 requires us to deal with many of the challenges faced by processor designers.

We then provide some background on digital hardware design. We describe the basic building blocks used in a processor and how they are connected together and operated. This presentation builds on our discussion of Boolean algebra and bit-level operations from Chapter 2. We also introduce a simple language, HCL (for "hardware control language"), to describe the control portions of hardware systems. We will later use this language to describe our processor designs. Even if you already have some background in logic design, read this section to understand our particular notation.

As a first step in designing a processor, we present a functionally correct, but somewhat impractical, Y86-64 processor based on *sequential* operation. This processor executes a complete Y86-64 instruction on every clock cycle. The clock must run slowly enough to allow an entire series of actions to complete within one cycle. Such a processor could be implemented, but its performance would be well below what could be achieved for this much hardware.

With the sequential design as a basis, we then apply a series of transformations to create a *pipelined* processor. This processor breaks the execution of each instruction into five steps, each of which is handled by a separate section or *stage* of the hardware. Instructions progress through the stages of the pipeline, with one instruction entering the pipeline on each clock cycle. As a result, the processor can be executing the different steps of up to five instructions simultaneously. Making this processor preserve the sequential behavior of the Y86-64 ISA requires handling a variety of *hazard* conditions, where the location or operands of one instruction depend on those of other instructions that are still in the pipeline.

We have devised a variety of tools for studying and experimenting with our processor designs. These include an assembler for Y86-64, a simulator for running Y86-64 programs on your machine, and simulators for two sequential and one pipelined processor design. The control logic for these designs is described by files in HCL notation. By editing these files and recompiling the simulator, you can alter and extend the simulator's behavior. A number of exercises are provided that involve implementing new instructions and modifying how the machine processes instructions. Testing code is provided to help you evaluate the correctness of your modifications. These exercises will greatly aid your understanding of the material and will give you an appreciation for the many different design alternatives faced by processor designers.

Web Aside ARCH:VLOG on page 503 presents a representation of our pipelined Y86-64 processor in the Verilog hardware description language. This involves creating modules for the basic hardware building blocks and for the overall processor structure. We automatically translate the HCL description of the control logic into Verilog. By first debugging the HCL description with our simulators, we eliminate many of the tricky bugs that would otherwise show up in the hardware design. Given a Verilog description, there are commercial and open-source tools to support simulation and *logic synthesis*, generating actual circuit designs for the microprocessors. So, although much of the effort we expend here is to create pictorial and textual descriptions of a system, much as one would when writing software, the fact that these designs can be automatically synthesized demonstrates that we are indeed creating a system that can be realized as hardware.

4.1 The Y86-64 Instruction Set Architecture

Defining an instruction set architecture, such as Y86-64, includes defining the different components of its state, the set of instructions and their encodings, a set of programming conventions, and the handling of exceptional events.

4.1.1 Programmer-Visible State

As Figure 4.1 illustrates, each instruction in a Y86-64 program can read and modify some part of the processor state. This is referred to as the *programmer-visible* state, where the “programmer” in this case is either someone writing programs in assembly code or a compiler generating machine-level code. We will see in our processor implementations that we do not need to represent and organize this state in exactly the manner implied by the ISA, as long as we can make sure that machine-level programs appear to have access to the programmer-visible state. The state for Y86-64 is similar to that for x86-64. There are 15 *program registers:* `%rax`, `%rcx`, `%rdx`, `%rbx`, `%rsp`, `%rbp`, `%rsi`, `%rdi`, and `%r8` through `%r14`. (We omit the x86-64 register `%r15` to simplify the instruction encoding.) Each of these stores a 64-bit word. Register `%rsp` is used as a stack pointer by the push, pop, call, and return instructions. Otherwise, the registers have no fixed meanings or values. There are three single-bit *condition codes*, `ZF`, `SF`, and `OF`, storing information

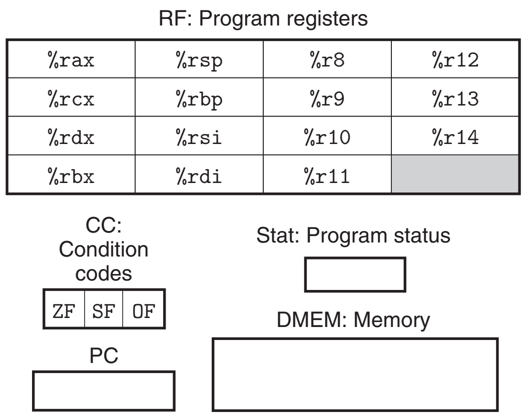


Figure 4.1
Y86-64 programmer-visible state. As with x86-64, programs for Y86-64 access and modify the program registers, the condition codes, the program counter (PC), and the memory. The status code indicates whether the program is running normally or some special event has occurred.

about the effect of the most recent arithmetic or logical instruction. The program counter (PC) holds the address of the instruction currently being executed.

The *memory* is conceptually a large array of bytes, holding both program and data. Y86-64 programs reference memory locations using *virtual addresses*. A combination of hardware and operating system software translates these into the actual, or *physical*, addresses indicating where the values are actually stored in memory. We will study virtual memory in more detail in Chapter 9. For now, we can think of the virtual memory system as providing Y86-64 programs with an image of a monolithic byte array.

A final part of the program state is a status code Stat, indicating the overall state of program execution. It will indicate either normal operation or that some sort of *exception* has occurred, such as when an instruction attempts to read from an invalid memory address. The possible status codes and the handling of exceptions is described in Section 4.1.4.

4.1.2 Y86-64 Instructions

Figure 4.2 gives a concise description of the individual instructions in the Y86-64 ISA. We use this instruction set as a target for our processor implementations. The set of Y86-64 instructions is largely a subset of the x86-64 instruction set. It includes only 8-byte integer operations, has fewer addressing modes, and includes a smaller set of operations. Since we only use 8-byte data, we can refer to these as "words" without any ambiguity. In this figure, we show the assembly-code representation of the instructions on the left and the byte encodings on the right. Figure 4.3 shows further details of some of the instructions. The assembly-code format is similar to the ATT format for x86-64.

Here are some details about the Y86-64 instructions.

- The x86-64 `movq` instruction is split into four different instructions: `irmovq`, `rrmovq`, `mrmovq`, and `rmmovq`, explicitly indicating the form of the source and destination. The source is either immediate (`i`), register (`r`), or memory (`m`). It is designated by the first character in the instruction name. The destination is either register (`r`) or memory (`m`). It is designated by the second character in the instruction name. Explicitly identifying the four types of data transfer will prove helpful when we decide how to implement them.

  The memory references for the two memory movement instructions have a simple base and displacement format. We do not support the second index register or any scaling of a register's value in the address computation.

  As with x86-64, we do not allow direct transfers from one memory location to another. In addition, we do not allow a transfer of immediate data to memory.
- There are four integer operation instructions, shown in Figure 4.2 as `OPq`. These are `addq`, `subq`, `andq`, and `xorq`. They operate only on register data, whereas x86-64 also allows operations on memory data. These instructions set the three condition codes `ZF`, `SF`, and `OF` (zero, sign, and overflow).

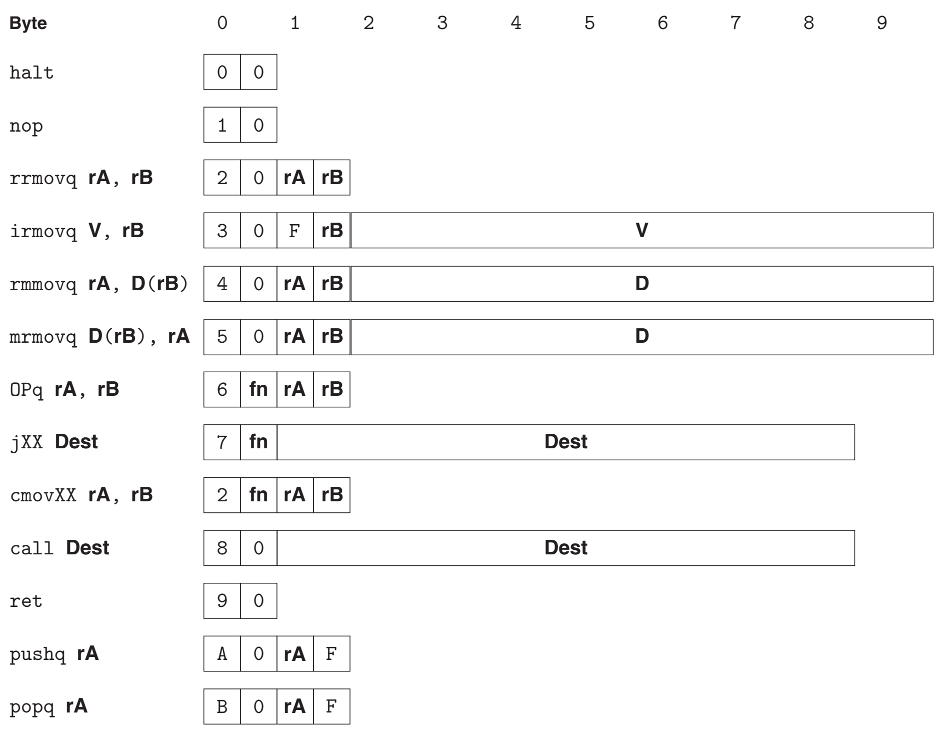


Figure 4.2 Y86-64 instruction set. Instruction encodings range between 1 and 10 bytes. An instruction consists of a 1-byte instruction specifier, possibly a 1-byte register specifier, and possibly an 8-byte constant word. Field `fn` specifies a particular integer operation (`OPq`), data movement condition (`cmovXX`), or branch condition (`jXX`). All numeric values are shown in hexadecimal.

- The seven jump instructions (shown in Figure 4.2 as `jXX`) are `jmp`, `jle`, `jl`, `je`, `jne`, `jge`, and `jg`. Branches are taken according to the type of branch and the settings of the condition codes. The branch conditions are the same as with x86-64 (Figure 3.15).
- There are six conditional move instructions (shown in Figure 4.2 as `cmovXX`): `cmovle`, `cmovl`, `cmove`, `cmovne`, `cmovge`, and `cmovg`. These have the same format as the register–register move instruction `rrmovq`, but the destination register is updated only if the condition codes satisfy the required constraints.
- The `call` instruction pushes the return address on the stack and jumps to the destination address. The `ret` instruction returns from such a call.
- The `pushq` and `popq` instructions implement push and pop, just as they do in x86-64.
- The `halt` instruction stops instruction execution. x86-64 has a comparable instruction, called `hlt`. x86-64 application programs are not permitted to use

this instruction, since it causes the entire system to suspend operation. For Y86-64, executing the `halt` instruction causes the processor to stop, with the status code set to `HLT`. (See Section 4.1.4.)

4.1.3 Instruction Encoding

Figure 4.2 also shows the byte-level encoding of the instructions. Each instruction requires between 1 and 10 bytes, depending on which fields are required. Every instruction has an initial byte identifying the instruction type. This byte is split into two 4-bit parts: the high-order, or *code*, part, and the low-order, or *function*, part. As can be seen in Figure 4.2, code values range from `0` to `0xB`. The function values are significant only for the cases where a group of related instructions share a common code. These are given in Figure 4.3, showing the specific encodings of the integer operation, branch, and conditional move instructions. Observe that `rrmovq` has the same instruction code as the conditional moves. It can be viewed as an "unconditional move" just as the `jmp` instruction is an unconditional jump, both having function code `0`.

As shown in Figure 4.4, each of the 15 program registers has an associated *register identifier* (ID) ranging from `0` to `0xE`. The numbering of registers in Y86-64 matches what is used in x86-64. The program registers are stored within the CPU in a *register file*, a small random access memory where the register IDs serve as addresses. ID value `0xF` is used in the instruction encodings and within our hardware designs when we need to indicate that no register should be accessed.

Some instructions are just 1 byte long, but those that require operands have longer encodings. First, there can be an additional *register specifier byte*, specifying either one or two registers. These register fields are called rA and rB in Figure 4.2. As the assembly-code versions of the instructions show, they can specify the registers used for data sources and destinations, as well as the base register used in an address computation, depending on the instruction type. Instructions that have no register operands, such as branches and `call`, do not have a register specifier byte. Those that require just one register operand (`irmovq`, `pushq`, and `popq`) have

Operations			Branches						Moves					
`addq`	6	0	`jmp`	7	0	`jne`	7	4	`rrmovq`	2	0	`cmovne`	2	4
`subq`	6	1	`jle`	7	1	`jge`	7	5	`cmovle`	2	1	`cmovge`	2	5
`andq`	6	2	`jl`	7	2	`jg`	7	6	`cmovl`	2	2	`cmovg`	2	6
`xorq`	6	3	`je`	7	3				`cmove`	2	3			

Figure 4.3 **Function codes for Y86-64 instruction set.** The code specifies a particular integer operation, branch condition, or data transfer condition. These instructions are shown as `OPq`, `jXX`, and `cmovXX` in Figure 4.2.

Number	Register name	Number	Register name
0	`%rax`	8	`%r8`
1	`%rcx`	9	`%r9`
2	`%rdx`	A	`%r10`
3	`%rbx`	B	`%r11`
4	`%rsp`	C	`%r12`
5	`%rbp`	D	`%r13`
6	`%rsi`	E	`%r14`
7	`%rdi`	F	No register

Figure 4.4 **Y86-64 program register identifiers.** Each of the 15 program registers has an associated identifier (ID) ranging from `0` to `0xE`. ID `0xF` in a register field of an instruction indicates the absence of a register operand.

the other register specifier set to value `0xF`. This convention will prove useful in our processor implementation.

Some instructions require an additional 8-byte *constant word*. This word can serve as the immediate data for `irmovq`, the displacement for `rmmovq` and `mrmovq` address specifiers, and the destination of branches and calls. Note that branch and call destinations are given as absolute addresses, rather than using the PC-relative addressing seen in x86-64. Processors use PC-relative addressing to give more compact encodings of branch instructions and to allow code to be shifted from one part of memory to another without the need to update all of the branch target addresses. Since we are more concerned with simplicity in our presentation, we use absolute addressing. As with x86-64, all integers have a little-endian encoding. When the instruction is written in disassembled form, these bytes appear in reverse order.

As an example, let us generate the byte encoding of the instruction `rmmovq %rsp,0x123456789abcd(%rdx)` in hexadecimal. From Figure 4.2, we can see that `rmmovq` has initial byte `40`. We can also see that source register `%rsp` should be encoded in the rA field, and base register `%rdx` should be encoded in the rB field. Using the register numbers in Figure 4.4, we get a register specifier byte of `42`. Finally, the displacement is encoded in the 8-byte constant word. We first pad `0x123456789abcd` with leading zeros to fill out 8 bytes, giving a byte sequence of `00 01 23 45 67 89 ab cd`. We write this in byte-reversed order as `cd ab 89 67 45 23 01 00`. Combining these, we get an instruction encoding of `4042cdab896745230100`.

One important property of any instruction set is that the byte encodings must have a unique interpretation. An arbitrary sequence of bytes either encodes a unique instruction sequence or is not a legal byte sequence. This property holds for Y86-64, because every instruction has a unique combination of code and function in its initial byte, and given this byte, we can determine the length and meaning of any additional bytes. This property ensures that a processor can execute an object-code program without any ambiguity about the meaning of the code. Even if the code is embedded within other bytes in the program, we can readily determine

Aside Comparing x86-64 to Y86-64 instruction encodings

Compared with the instruction encodings used in x86-64, the encoding of Y86-64 is much simpler but also less compact. The register fields occur only in fixed positions in all Y86-64 instructions, whereas they are packed into various positions in the different x86-64 instructions. An x86-64 instruction can encode constant values in 1, 2, 4, or 8 bytes, whereas Y86-64 always requires 8 bytes.

the instruction sequence as long as we start from the first byte in the sequence. On the other hand, if we do not know the starting position of a code sequence, we cannot reliably determine how to split the sequence into individual instructions. This causes problems for disassemblers and other tools that attempt to extract machine-level programs directly from object-code byte sequences.

Practice Problem 4.1 (solution page 516)

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,%rcx
    jmp loop
```

Practice Problem 4.2 (solution page 517)

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 value occurs. For each sequence, we show the starting address, then a colon, and then the byte sequence.

A. `0x100: 30f3fcffffffffffffff40630008000000000000`

B. `0x200: a06f800c020000000000000030f30a00000000000000`

C. `0x300: 5054070000000000000010f0b01f`

D. `0x400: 611373000400000000000000`

E. `0x500: 6362a0f0`

Aside RISC and CISC instruction sets

x86-64 is sometimes labeled as a "complex instruction set computer" (CISC—pronounced "sisk"), and is deemed to be the opposite of ISAs that are classified as "reduced instruction set computers" (RISC—pronounced "risk"). Historically, CISC machines came first, having evolved from the earliest computers. By the early 1980s, instruction sets for mainframe and minicomputers had grown quite large, as machine designers incorporated new instructions to support high-level tasks, such as manipulating circular buffers, performing decimal arithmetic, and evaluating polynomials. The first microprocessors appeared in the early 1970s and had limited instruction sets, because the integrated-circuit technology then posed severe constraints on what could be implemented on a single chip. Microprocessors evolved quickly and, by the early 1980s, were following the same path of increasing instruction set complexity that had been the case for mainframes and minicomputers. The x86 family took this path, evolving into IA32, and more recently into x86-64. The x86 line continues to evolve as new classes of instructions are added based on the needs of emerging applications.

The RISC design philosophy developed in the early 1980s as an alternative to these trends. A group of hardware and compiler experts at IBM, strongly influenced by the ideas of IBM researcher John Cocke, recognized that they could generate efficient code for a much simpler form of instruction set. In fact, many of the high-level instructions that were being added to instruction sets were very difficult to generate with a compiler and were seldom used. A simpler instruction set could be implemented with much less hardware and could be organized in an efficient pipeline structure, similar to those described later in this chapter. IBM did not commercialize this idea until many years later, when it developed the Power and PowerPC ISAs.

The RISC concept was further developed by Professors David Patterson, of the University of California at Berkeley, and John Hennessy, of Stanford University. Patterson gave the name RISC to this new class of machines, and CISC to the existing class, since there had previously been no need to have a special designation for a nearly universal form of instruction set.

When comparing CISC with the original RISC instruction sets, we find the following general characteristics:

CISC	Early RISC
A large number of instructions. The Intel document describing the complete set of instructions [51] is over 1,200 pages long.	Many fewer instructions—typically less than 100.
Some instructions with long execution times. These include instructions that copy an entire block from one part of memory to another and others that copy multiple registers to and from memory.	No instruction with a long execution time. Some early RISC machines did not even have an integer multiply instruction, requiring compilers to implement multiplication as a sequence of additions.
Variable-size encodings. x86-64 instructions can range from 1 to 15 bytes.	Fixed-length encodings. Typically all instructions are encoded as 4 bytes.

Aside RISC and CISC instruction sets *(continued)*

CISC	Early RISC
Multiple formats for specifying operands. In x86-64, a memory operand specifier can have many different combinations of displacement, base and index registers, and scale factors.	Simple addressing formats. Typically just base and displacement addressing.
Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logical operations only use register operands. Memory referencing is only allowed by *load* instructions, reading from memory into a register, and *store* instructions, writing from a register to memory. This convention is referred to as a *load/store architecture*.
Implementation artifacts hidden from machine-level programs. The ISA provides a clean abstraction between programs and how they get executed.	Implementation artifacts exposed to machine-level programs. Some RISC machines prohibit particular instruction sequences and have jumps that do not take effect until the following instruction is executed. The compiler is given the task of optimizing performance within these constraints.
Condition codes. Special flags are set as a side effect of instructions and then used for conditional branch testing.	No condition codes. Instead, explicit test instructions store the test results in normal registers for use in conditional evaluation.
Stack-intensive procedure linkage. The stack is used for procedure arguments and return addresses.	Register-intensive procedure linkage. Registers are used for procedure arguments and return addresses. Some procedures can thereby avoid any memory references. Typically, the processor has many more (up to 32) registers.

The Y86-64 instruction set includes attributes of both CISC and RISC instruction sets. On the CISC side, it has condition codes and variable-length instructions, and it uses the stack to store return addresses. On the RISC side, it uses a load/store architecture and a regular instruction encoding, and it passes procedure arguments through registers. It can be viewed as taking a CISC instruction set (x86) and simplifying it by applying some of the principles of RISC.

Aside The RISC versus CISC controversy

Through the 1980s, battles raged in the computer architecture community regarding the merits of RISC versus CISC instruction sets. Proponents of RISC claimed they could get more computing power for a given amount of hardware through a combination of streamlined instruction set design, advanced compiler technology, and pipelined processor implementation. CISC proponents countered that fewer CISC instructions were required to perform a given task, and so their machines could achieve higher overall performance.

Major companies introduced RISC processor lines, including Sun Microsystems (SPARC), IBM and Motorola (PowerPC), and Digital Equipment Corporation (Alpha). A British company, Acorn Computers Ltd., developed its own architecture, ARM (originally an acronym for "Acorn RISC machine"), which has become widely used in embedded applications, such as cell phones.

In the early 1990s, the debate diminished as it became clear that neither RISC nor CISC in their purest forms were better than designs that incorporated the best ideas of both. RISC machines evolved and introduced more instructions, many of which take multiple cycles to execute. RISC machines today have hundreds of instructions in their repertoire, hardly fitting the name "reduced instruction set machine." The idea of exposing implementation artifacts to machine-level programs proved to be shortsighted. As new processor models were developed using more advanced hardware structures, many of these artifacts became irrelevant, but they still remained part of the instruction set. Still, the core of RISC design is an instruction set that is well suited to execution on a pipelined machine.

More recent CISC machines also take advantage of high-performance pipeline structures. As we will discuss in Section 5.7, they fetch the CISC instructions and dynamically translate them into a sequence of simpler, RISC-like operations. For example, an instruction that adds a register to memory is translated into three operations: one to read the original memory value, one to perform the addition, and a third to write the sum to memory. Since the dynamic translation can generally be performed well in advance of the actual instruction execution, the processor can sustain a very high execution rate.

Marketing issues, apart from technological ones, have also played a major role in determining the success of different instruction sets. By maintaining compatibility with its existing processors, Intel with x86 made it easy to keep moving from one generation of processor to the next. As integrated-circuit technology improved, Intel and other x86 processor manufacturers could overcome the inefficiencies created by the original 8086 instruction set design, using RISC techniques to produce performance comparable to the best RISC machines. As we saw in Section 3.1, the evolution of IA32 into x86-64 provided an opportunity to incorporate several features of RISC into the x86 family. In the areas of desktop, laptop, and server-based computing, x86 has achieved near total domination.

RISC processors have done very well in the market for *embedded processors*, controlling such systems as cellular telephones, automobile brakes, and Internet appliances. In these applications, saving on cost and power is more important than maintaining backward compatibility. In terms of the number of processors sold, this is a very large and growing market.

4.1.4 Y86-64 Exceptions

The programmer-visible state for Y86-64 (Figure 4.1) includes a status code `Stat` describing the overall state of the executing program. The possible values for this code are shown in Figure 4.5. Code value 1, named `AOK`, indicates that the program

Value	Name	Meaning
1	`AOK`	Normal operation
2	`HLT`	`halt` instruction encountered
3	`ADR`	Invalid address encountered
4	`INS`	Invalid instruction encountered

Figure 4.5 **Y86-64 status codes.** In our design, the processor halts for any code other than `AOK`.

is executing normally, while the other codes indicate that some type of *exception* has occurred. Code 2, named `HLT`, indicates that the processor has executed a `halt` instruction. Code 3, named `ADR`, indicates that the processor attempted to read from or write to an invalid memory address, either while fetching an instruction or while reading or writing data. We limit the maximum address (the exact limit varies by implementation), and any access to an address beyond this limit will trigger an `ADR` exception. Code 4, named `INS`, indicates that an invalid instruction code has been encountered.

For Y86-64, we will simply have the processor stop executing instructions when it encounters any of the exceptions listed. In a more complete design, the processor would typically invoke an *exception handler*, a procedure designated to handle the specific type of exception encountered. As described in Chapter 8, exception handlers can be configured to have different effects, such as aborting the program or invoking a user-defined *signal handler*.

4.1.5 Y86-64 Programs

Figure 4.6 shows x86-64 and Y86-64 assembly code for the following C function:

```
long sum(long *start, long count)
{
    long sum = 0;
    while (count) {
        sum += *start;
        start++;
        count--;
    }
    return sum;
}
```

The x86-64 code was generated by the GCC compiler. The Y86-64 code is similar, but with the following differences:

- The Y86-64 code loads constants into registers (lines 2–3), since it cannot use immediate data in arithmetic instructions.

x86-64 code

```
    long sum(long *start, long count)
    start in %rdi, count in %rsi
sum:
  movl    $0, %eax            sum = 0
  jmp     .L2                 Goto test
.L3:                        loop:
  addq    (%rdi), %rax        Add *start to sum
  addq    $8, %rdi            start++
  subq    $1, %rsi            count--
.L2:                        test:
  testq   %rsi, %rsi          Test sum
  jne     .L3                 If !=0, goto loop
  rep; ret                    Return
```

Y86-64 code

```
    long sum(long *start, long count)
    start in %rdi, count in %rsi
sum:
  irmovq $8,%r8             Constant 8
  irmovq $1,%r9             Constant 1
  xorq %rax,%rax            sum = 0
  andq %rsi,%rsi            Set CC
  jmp    test               Goto test
loop:
  mrmovq (%rdi),%r10        Get *start
  addq %r10,%rax            Add to sum
  addq %r8,%rdi             start++
  subq %r9,%rsi             count--.  Set CC
test:
  jne    loop               Stop when 0
  ret                       Return
```

Figure 4.6 Comparison of Y86-64 and x86-64 assembly programs. The `sum` function computes the sum of an integer array. The Y86-64 code follows the same general pattern as the x86-64 code.

- The Y86-64 code requires two instructions (lines 8–9) to read a value from memory and add it to a register, whereas the x86-64 code can do this with a single `addq` instruction (line 5).
- Our hand-coded Y86-64 implementation takes advantage of the property that the `subq` instruction (line 11) also sets the condition codes, and so the `testq` instruction of the GCC-generated code (line 9) is not required. For this to work, though, the Y86-64 code must set the condition codes prior to entering the loop with an `andq` instruction (line 5).

Figure 4.7 shows an example of a complete program file written in Y86-64 assembly code. The program contains both data and instructions. Directives indicate where to place code or data and how to align it. The program specifies issues such as stack placement, data initialization, program initialization, and program termination.

In this program, words beginning with '`.`' are *assembler directives* telling the assembler to adjust the address at which it is generating code or to insert some words of data. The directive `.pos 0` (line 2) indicates that the assembler should begin generating code starting at address `0`. This is the starting address for all Y86-64 programs. The next instruction (line 3) initializes the stack pointer. We can see that the label `stack` is declared at the end of the program (line 40), to indicate address `0x200` using a `.pos` directive (line 39). Our stack will therefore start at this address and grow toward lower addresses. We must ensure that the stack does not grow so large that it overwrites the code or other program data.

Lines 8 to 13 of the program declare an array of four words, having the values

`0x000d000d000d000d`, `0x00c000c000c000c0`,
`0x0b000b000b000b00`, `0xa000a000a000a000`

The label `array` denotes the start of this array, and is aligned on an 8-byte boundary (using the `.align` directive). Lines 16 to 19 show a "main" procedure that calls the function `sum` on the four-word array and then halts.

As this example shows, since our only tool for creating Y86-64 code is an assembler, the programmer must perform tasks we ordinarily delegate to the compiler, linker, and run-time system. Fortunately, we only do this for small programs, for which simple mechanisms suffice.

Figure 4.8 shows the result of assembling the code shown in Figure 4.7 by an assembler we call YAS. The assembler output is in ASCII format to make it more readable. On lines of the assembly file that contain instructions or data, the object code contains an address, followed by the values of between 1 and 10 bytes.

We have implemented an *instruction set simulator* we call YIS, the purpose of which is to model the execution of a Y86-64 machine-code program without attempting to model the behavior of any specific processor implementation. This form of simulation is useful for debugging programs before actual hardware is available, and for checking the result of either simulating the hardware or running

```
# Execution begins at address 0
        .pos 0
        irmovq stack, %rsp      # Set up stack pointer
        call main               # Execute main program
        halt                    # Terminate program

# Array of 4 elements
        .align 8
array:
        .quad 0x000d000d000d
        .quad 0x00c000c000c0
        .quad 0x0b000b000b00
        .quad 0xa000a000a000

main:
        irmovq array,%rdi
        irmovq $4,%rsi
        call sum                # sum(array, 4)
        ret

# long sum(long *start, long count)
# start in %rdi, count in %rsi
sum:
        irmovq $8,%r8        # Constant 8
        irmovq $1,%r9        # Constant 1
        xorq %rax,%rax       # sum = 0
        andq %rsi,%rsi       # Set CC
        jmp     test         # Goto test
loop:
        mrmovq (%rdi),%r10   # Get *start
        addq %r10,%rax       # Add to sum
        addq %r8,%rdi        # start++
        subq %r9,%rsi        # count--.  Set CC
test:
        jne    loop          # Stop when 0
        ret                  # Return

# Stack starts here and grows to lower addresses
        .pos 0x200
stack:
```

Figure 4.7 Sample program written in Y86-64 assembly code. The `sum` function is called to compute the sum of a four-element array.

```
                             | # Execution begins at address 0
0x000:                       |   .pos 0
0x000: 30f40002000000000000 |   irmovq stack, %rsp      # Set up stack pointer
0x00a: 803800000000000000   |   call main               # Execute main program
0x013: 00                   |   halt                    # Terminate program
                             |
                             | # Array of 4 elements
0x018:                       |   .align 8
0x018:                       | array:
0x018: 0d000d000d000000     |   .quad 0x000d000d000d
0x020: c000c000c0000000     |   .quad 0x00c000c000c0
0x028: 000b000b000b0000     |   .quad 0x0b000b000b00
0x030: 00a000a000a00000     |   .quad 0xa000a000a000
                             |
0x038:                       | main:
0x038: 30f71800000000000000 |   irmovq array,%rdi
0x042: 30f60400000000000000 |   irmovq $4,%rsi
0x04c: 805600000000000000   |   call sum                # sum(array, 4)
0x055: 90                   |   ret
                             |
                             | # long sum(long *start, long count)
                             | # start in %rdi, count in %rsi
0x056:                       | sum:
0x056: 30f80800000000000000 |   irmovq $8,%r8        # Constant 8
0x060: 30f90100000000000000 |   irmovq $1,%r9        # Constant 1
0x06a: 6300                 |   xorq %rax,%rax       # sum = 0
0x06c: 6266                 |   andq %rsi,%rsi       # Set CC
0x06e: 708700000000000000   |   jmp     test         # Goto test
0x077:                       | loop:
0x077: 50a70000000000000000 |   mrmovq (%rdi),%r10   # Get *start
0x081: 60a0                 |   addq %r10,%rax       # Add to sum
0x083: 6087                 |   addq %r8,%rdi        # start++
0x085: 6196                 |   subq %r9,%rsi        # count--.  Set CC
0x087:                       | test:
0x087: 747700000000000000   |   jne    loop          # Stop when 0
0x090: 90                   |   ret                  # Return
                             |
                             | # Stack starts here and grows to lower addresses
0x200:                       |   .pos 0x200
0x200:                       | stack:
```

Figure 4.8 **Output of YAS assembler.** Each line includes a hexadecimal address and between 1 and 10 bytes of object code.

the program on the hardware itself. Running on our sample object code, YIS generates the following output:

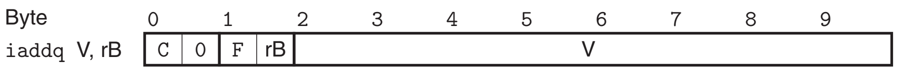
```
Stopped in 34 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000abcdabcdabcd
%rsp:   0x0000000000000000      0x0000000000000200
%rdi:   0x0000000000000000      0x0000000000000038
%r8:    0x0000000000000000      0x0000000000000008
%r9:    0x0000000000000000      0x0000000000000001
%r10:   0x0000000000000000      0x0000a000a000a000

Changes to memory:
0x01f0: 0x0000000000000000      0x0000000000000055
0x01f8: 0x0000000000000000      0x0000000000000013
```

The first line of the simulation output summarizes the execution and the resulting values of the PC and program status. In printing register and memory values, it only prints out words that change during simulation, either in registers or in memory. The original values (here they are all zero) are shown on the left, and the final values are shown on the right. We can see in this output that register `%rax` contains `0xabcdabcdabcdabcd`, the sum of the 4-element array passed to procedure `sum`. In addition, we can see that the stack, which starts at address `0x200` and grows toward lower addresses, has been used, causing changes to words of memory at addresses `0x1f0`–`0x1f8`. The maximum address for executable code is `0x090`, and so the pushing and popping of values on the stack did not corrupt the executable code.

Practice Problem 4.3 (solution page 518)

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



This instruction adds the constant value V to register rB.

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

Practice Problem 4.4 (solution page 518)

Write Y86-64 code to implement a recursive product function `rproduct`, based on the following C code:

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

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

Practice Problem 4.5 (solution page 519)

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.

Practice Problem 4.6 (solution page 519)

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.

4.1.6 Some Y86-64 Instruction Details

Most Y86-64 instructions transform the program state in a straightforward manner, and so defining the intended effect of each instruction is not difficult. Two unusual instruction combinations, however, require special attention.

The `pushq` instruction both decrements the stack pointer by 8 and writes a register value to memory. It is therefore not totally clear what the processor should do when executing the instruction `pushq %rsp`, since the register being pushed is being changed by the same instruction. Two different conventions are possible: (1) push the original value of `%rsp`, or (2) push the decremented value of `%rsp`.

For the Y86-64 processor, let us adopt the same convention as is used with x86-64, as determined in the following problem.

Practice Problem 4.7 (solution page 520)

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 214 describes how to write programs that combine C code with handwritten assembly code):

```
    .text
  .globl 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 function `pushtest` always returns 0. What does this imply about the behavior of the instruction `pushq %rsp` under x86-64?

A similar ambiguity occurs for the instruction `popq %rsp`. It could either set `%rsp` to the value read from memory or to the incremented stack pointer. As with Problem 4.7, let us run an experiment to determine how an x86-64 machine would handle this instruction, and then design our Y86-64 machine to follow the same convention.

Practice Problem 4.8 (solution page 520)

The following assembly-code function lets us determine the behavior of the instruction `popq %rsp` for x86-64:

```
    .text
  .globl 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 return value
    movq    %rdi, %rsp     Restore stack pointer
    ret
```

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

Aside Getting the details right: Inconsistencies across x86 models

Practice Problems 4.7 and 4.8 are designed to help us devise a consistent set of conventions for instructions that push or pop the stack pointer. There seems to be little reason why one would want to perform either of these operations, and so a natural question to ask is, "Why worry about such picky details?"

Several useful lessons can be learned about the importance of consistency from the following excerpt from the Intel documentation of the PUSH instruction [51]:

> For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true for Intel 64 architecture, real-address and virtual-8086 modes of IA-32 architecture.) For the Intel® 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2). (PUSH ESP instruction. Intel Corporation. 50.)

Although the exact details of this note may be difficult to follow, we can see that it states that, depending on what mode an x86 processor operates under, it will do different things when instructed to push the stack pointer register. Some modes push the original value, while others push the decremented value. (Interestingly, there is no corresponding ambiguity about popping to the stack pointer register.) There are two drawbacks to this inconsistency:

- It decreases code portability. Programs may have different behavior depending on the processor mode. Although the particular instruction is not at all common, even the potential for incompatibility can have serious consequences.
- It complicates the documentation. As we see here, a special note is required to try to clarify the differences. The documentation for x86 is already complex enough without special cases such as this one.

We conclude, therefore, that working out details in advance and striving for complete consistency can save a lot of trouble in the long run.

4.2 Logic Design and the Hardware Control Language HCL

In hardware design, electronic circuits are used to compute functions on bits and to store bits in different kinds of memory elements. Most contemporary circuit technology represents different bit values as high or low voltages on signal wires. In current technology, logic value 1 is represented by a high voltage of around 1.0 volt, while logic value 0 is represented by a low voltage of around 0.0 volts. Three major components are required to implement a digital system: *combinational logic* to compute functions on the bits, *memory elements* to store bits, and *clock signals* to regulate the updating of the memory elements.

In this section, we provide a brief description of these different components. We also introduce HCL (for "hardware control language"), the language that we use to describe the control logic of the different processor designs. We only describe HCL informally here. A complete reference for HCL can be found in Web Aside ARCH:HCL on page 508.

Aside Modern logic design

At one time, hardware designers created circuit designs by drawing schematic diagrams of logic circuits (first with paper and pencil, and later with computer graphics terminals). Nowadays, most designs are expressed in a *hardware description language* (HDL), a textual notation that looks similar to a programming language but that is used to describe hardware structures rather than program behaviors. The most commonly used languages are Verilog, having a syntax similar to C, and VHDL, having a syntax similar to the Ada programming language. These languages were originally designed for creating simulation models of digital circuits. In the mid-1980s, researchers developed *logic synthesis* programs that could generate efficient circuit designs from HDL descriptions. There are now a number of commercial synthesis programs, and this has become the dominant technique for generating digital circuits. This shift from hand-designed circuits to synthesized ones can be likened to the shift from writing programs in assembly code to writing them in a high-level language and having a compiler generate the machine code.

Our HCL language expresses only the control portions of a hardware design, with only a limited set of operations and with no modularity. As we will see, however, the control logic is the most difficult part of designing a microprocessor. We have developed tools that can directly translate HCL into Verilog, and by combining this code with Verilog code for the basic hardware units, we can generate HDL descriptions from which actual working microprocessors can be synthesized. By carefully separating out, designing, and testing the control logic, we can create a working microprocessor with reasonable effort. Web Aside ARCH:VLOG on page 503 describes how we can generate Verilog versions of a Y86-64 processor.

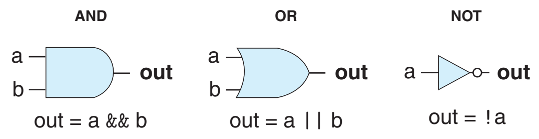


Figure 4.9
Logic gate types. Each gate generates output equal to some Boolean function of its inputs.

4.2.1 Logic Gates

Logic gates are the basic computing elements for digital circuits. They generate an output equal to some Boolean function of the bit values at their inputs. Figure 4.9 shows the standard symbols used for Boolean functions AND, OR, and NOT. HCL expressions are shown below the gates for the operators in C (Section 2.1.8): `&&` for AND, `||` for OR, and `!` for NOT. We use these instead of the bit-level C operators `&`, `|`, and `~`, because logic gates operate on single-bit quantities, not entire words. Although the figure illustrates only two-input versions of the AND and OR gates, it is common to see these being used as n-way operations for $n > 2$. We still write these in HCL using binary operators, though, so the operation of a three-input AND gate with inputs `a`, `b`, and `c` is described with the HCL expression `a && b && c`.

Logic gates are always active. If some input to a gate changes, then within some small amount of time, the output will change accordingly.

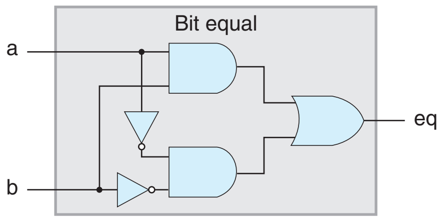


Figure 4.10
Combinational circuit to test for bit equality. The output will equal 1 when both inputs are 0 or both are 1.

4.2.2 Combinational Circuits and HCL Boolean Expressions

By assembling a number of logic gates into a network, we can construct computational blocks known as *combinational circuits*. Several restrictions are placed on how the networks are constructed:

- Every logic gate input must be connected to exactly one of the following: (1) one of the system inputs (known as a *primary input*), (2) the output connection of some memory element, or (3) the output of some logic gate.
- The outputs of two or more logic gates cannot be connected together. Otherwise, the two could try to drive the wire toward different voltages, possibly causing an invalid voltage or a circuit malfunction.
- The network must be *acyclic*. That is, there cannot be a path through a series of gates that forms a loop in the network. Such loops can cause ambiguity in the function computed by the network.

Figure 4.10 shows an example of a simple combinational circuit that we will find useful. It has two inputs, a and b. It generates a single output eq, such that the output will equal 1 if either a and b are both 1 (detected by the upper AND gate) or are both 0 (detected by the lower AND gate). We write the function of this network in HCL as

```
bool eq = (a && b) || (!a && !b);
```

This code simply defines the bit-level (denoted by data type `bool`) signal eq as a function of inputs a and b. As this example shows, HCL uses C-style syntax, with '=' associating a signal name with an expression. Unlike C, however, we do not view this as performing a computation and assigning the result to some memory location. Instead, it is simply a way to give a name to an expression.

Practice Problem 4.9 (solution page 520)

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?

Figure 4.11 shows another example of a simple but useful combinational circuit known as a *multiplexor* (commonly referred to as a "MUX"). A multiplexor

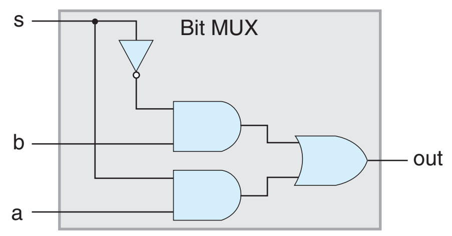


Figure 4.11
Single-bit multiplexor circuit. The output will equal input a if the control signal s is 1 and will equal input b when s is 0.

selects a value from among a set of different data signals, depending on the value of a control input signal. In this single-bit multiplexor, the two data signals are the input bits a and b, while the control signal is the input bit s. The output will equal a when s is 1, and it will equal b when s is 0. In this circuit, we can see that the two AND gates determine whether to pass their respective data inputs to the OR gate. The upper AND gate passes signal b when s is 0 (since the other input to the gate is `!s`), while the lower AND gate passes signal a when s is 1. Again, we can write an HCL expression for the output signal, using the same operations as are present in the combinational circuit:

```
bool out = (s && a) || (!s && b);
```

Our HCL expressions demonstrate a clear parallel between combinational logic circuits and logical expressions in C. They both use Boolean operations to compute functions over their inputs. Several differences between these two ways of expressing computation are worth noting:

- Since a combinational circuit consists of a series of logic gates, it has the property that the outputs continually respond to changes in the inputs. If some input to the circuit changes, then after some delay, the outputs will change accordingly. By contrast, a C expression is only evaluated when it is encountered during the execution of a program.
- Logical expressions in C allow arguments to be arbitrary integers, interpreting 0 as FALSE and anything else as TRUE. In contrast, our logic gates only operate over the bit values 0 and 1.
- Logical expressions in C have the property that they might only be partially evaluated. If the outcome of an AND or OR operation can be determined by just evaluating the first argument, then the second argument will not be evaluated. For example, with the C expression

  ```
  (a && !a) && func(b,c)
  ```

  the function `func` will not be called, because the expression `(a && !a)` evaluates to 0. In contrast, combinational logic does not have any partial evaluation rules. The gates simply respond to changing inputs.

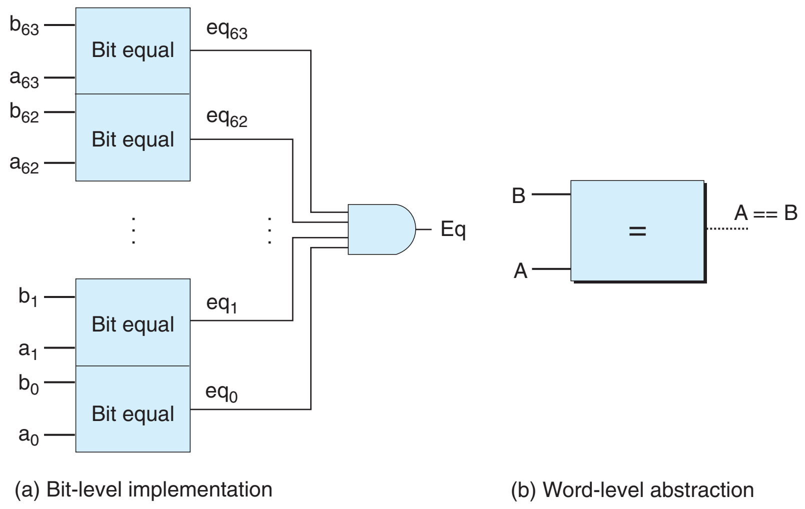


Figure 4.12 **Word-level equality test circuit.** The output will equal 1 when each bit from word A equals its counterpart from word B. Word-level equality is one of the operations in HCL.

4.2.3 Word-Level Combinational Circuits and HCL Integer Expressions

By assembling large networks of logic gates, we can construct combinational circuits that compute much more complex functions. Typically, we design circuits that operate on data *words*. These are groups of bit-level signals that represent an integer or some control pattern. For example, our processor designs will contain numerous words, with word sizes ranging between 4 and 64 bits, representing integers, addresses, instruction codes, and register identifiers.

Combinational circuits that perform word-level computations are constructed using logic gates to compute the individual bits of the output word, based on the individual bits of the input words. For example, Figure 4.12 shows a combinational circuit that tests whether two 64-bit words A and B are equal. That is, the output will equal 1 if and only if each bit of A equals the corresponding bit of B. This circuit is implemented using 64 of the single-bit equality circuits shown in Figure 4.10. The outputs of these single-bit circuits are combined with an AND gate to form the circuit output.

In HCL, we will declare any word-level signal as an `int`, without specifying the word size. This is done for simplicity. In a full-featured hardware description language, every word can be declared to have a specific number of bits. HCL allows words to be compared for equality, and so the functionality of the circuit shown in Figure 4.12 can be expressed at the word level as

```
bool Eq = (A == B);
```

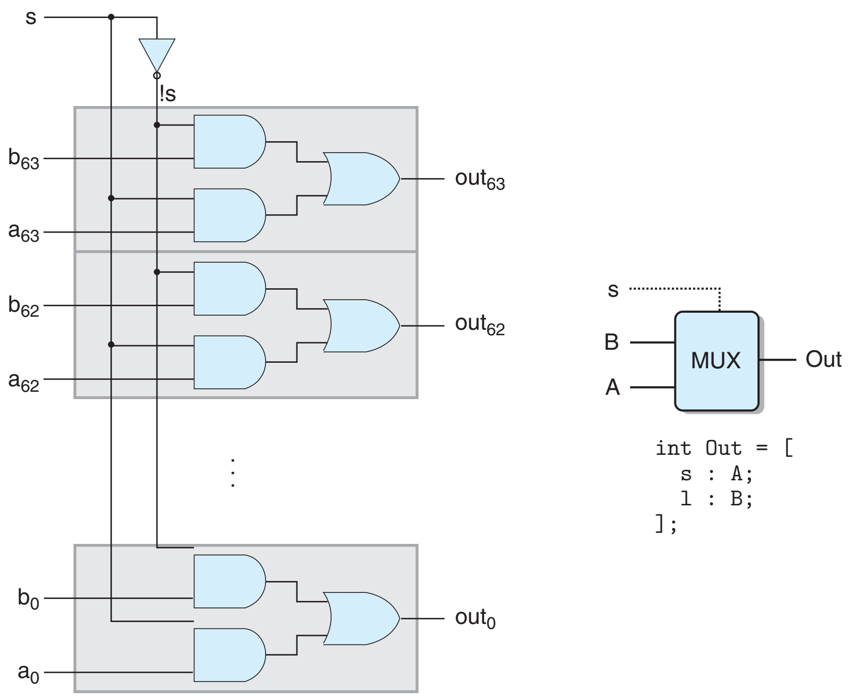
where arguments A and B are of type `int`. Note that we use the same syntax conventions as in C, where '=' denotes assignment and '==' denotes the equality operator.

As is shown on the right side of Figure 4.12, we will draw word-level circuits using medium-thickness lines to represent the set of wires carrying the individual bits of the word, and we will show a single-bit signal as a dashed line.

Practice Problem 4.10 (solution page 520)

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

Figure 4.13 shows the circuit for a word-level multiplexor. This circuit generates a 64-bit word Out equal to one of the two input words, A or B, depending on the control input bit s. The circuit consists of 64 identical subcircuits, each having a structure similar to the bit-level multiplexor from Figure 4.11. Rather than replicating the bit-level multiplexor 64 times, the word-level version reduces the number of inverters by generating !s once and reusing it at each bit position.



```
int Out = [
  s : A;
  1 : B;
];
```

(a) Bit-level implementation (b) Word-level abstraction

Figure 4.13 Word-level multiplexor circuit. The output will equal input word A when the control signal s is 1, and it will equal B otherwise. Multiplexors are described in HCL using case expressions.

We will use many forms of multiplexors in our processor designs. They allow us to select a word from a number of sources depending on some control condition. Multiplexing functions are described in HCL using *case expressions*. A case expression has the following general form:

$$
\begin{array}{l}
\texttt{[} \\
\quad select_1 \;:\; expr_1\texttt{;} \\
\quad select_2 \;:\; expr_2\texttt{;} \\
\quad \vdots \\
\quad select_k \;:\; expr_k\texttt{;} \\
\texttt{]}
\end{array}
$$

The expression contains a series of cases, where each case i consists of a Boolean expression $select_i$, indicating when this case should be selected, and an integer expression $expr_i$, indicating the resulting value.

Unlike the `switch` statement of C, we do not require the different selection expressions to be mutually exclusive. Logically, the selection expressions are evaluated in sequence, and the case for the first one yielding 1 is selected. For example, the word-level multiplexor of Figure 4.13 can be described in HCL as

```
word Out = [
        s: A;
        1: B;
];
```

In this code, the second selection expression is simply `1`, indicating that this case should be selected if no prior one has been. This is the way to specify a default case in HCL. Nearly all case expressions end in this manner.

Allowing nonexclusive selection expressions makes the HCL code more readable. An actual hardware multiplexor must have mutually exclusive signals controlling which input word should be passed to the output, such as the signals s and !s in Figure 4.13. To translate an HCL case expression into hardware, a logic synthesis program would need to analyze the set of selection expressions and resolve any possible conflicts by making sure that only the first matching case would be selected.

The selection expressions can be arbitrary Boolean expressions, and there can be an arbitrary number of cases. This allows case expressions to describe blocks where there are many choices of input signals with complex selection criteria. For example, consider the diagram of a 4-way multiplexor shown in Figure 4.14. This circuit selects from among the four input words A, B, C, and D based on the control signals s1 and s0, treating the controls as a 2-bit binary number. We can express this in HCL using Boolean expressions to describe the different combinations of control bit patterns:

```
word Out4 = [
        !s1 && !s0 : A; # 00
```

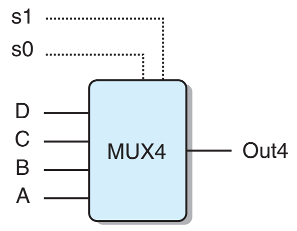
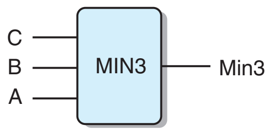


Figure 4.14
Four-way multiplexor. The different combinations of control signals s1 and s0 determine which data input is transmitted to the the output.

```
        !s1         : B; # 01
        !s0         : C; # 10
        1           : D; # 11
];
```

The comments on the right (any text starting with # and running for the rest of the line is a comment) show which combination of s1 and s0 will cause the case to be selected. Observe that the selection expressions can sometimes be simplified, since only the first matching case is selected. For example, the second expression can be written `!s1`, rather than the more complete `!s1&& s0`, since the only other possibility having s1 equal to 0 was given as the first selection expression. Similarly, the third expression can be written as `!s0`, while the fourth can simply be written as 1.

As a final example, suppose we want to design a logic circuit that finds the minimum value among a set of words A, B, and C, diagrammed as follows:



We can express this using an HCL case expression as

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

Practice Problem 4.11 (solution page 520)

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

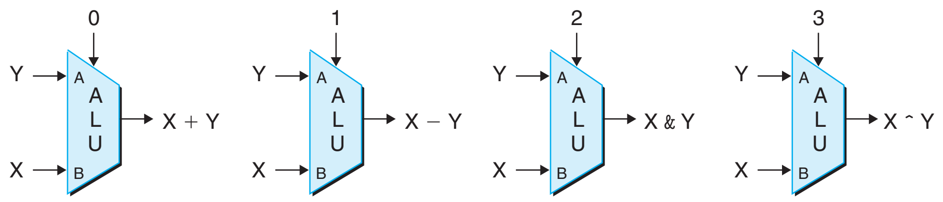


Figure 4.15 Arithmetic/logic unit (ALU). Depending on the setting of the function input, the circuit will perform one of four different arithmetic and logical operations.

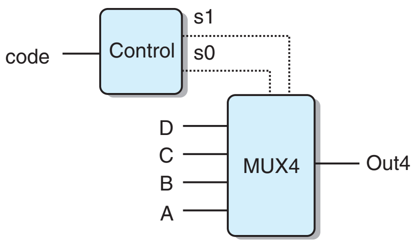
Practice Problem 4.12 (solution page 520)

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 maximum of the three inputs.

Combinational logic circuits can be designed to perform many different types of operations on word-level data. The detailed design of these is beyond the scope of our presentation. One important combinational circuit, known as an *arithmetic/logic unit* (ALU), is diagrammed at an abstract level in Figure 4.15. In our version, the circuit has three inputs: two data inputs labeled A and B and a control input. Depending on the setting of the control input, the circuit will perform different arithmetic or logical operations on the data inputs. Observe that the four operations diagrammed for this ALU correspond to the four different integer operations supported by the Y86-64 instruction set, and the control values match the function codes for these instructions (Figure 4.3). Note also the ordering of operands for subtraction, where the A input is subtracted from the B input. This ordering is chosen in anticipation of the ordering of arguments in the `subq` instruction.

4.2.4 Set Membership

In our processor designs, we will find many examples where we want to compare one signal against a number of possible matching signals, such as to test whether the code for some instruction being processed matches some category of instruction codes. As a simple example, suppose we want to generate the signals s1 and s0 for the 4-way multiplexor of Figure 4.14 by selecting the high- and low-order bits from a 2-bit signal code, as follows:



In this circuit, the 2-bit signal code would then control the selection among the four data words A, B, C, and D. We can express the generation of signals s1 and s0 using equality tests based on the possible values of code:

```
bool s1 = code == 2 || code == 3;
bool s0 = code == 1 || code == 3;
```

A more concise expression can be written that expresses the property that s1 is 1 when code is in the set {2, 3}, and s0 is 1 when code is in the set {1, 3}:

```
bool s1 = code in { 2, 3 };
bool s0 = code in { 1, 3 };
```

The general form of a set membership test is

iexpr `in` {$iexpr_1$, $iexpr_2$, . . . , $iexpr_k$}

where the value being tested (*iexpr*) and the candidate matches ($iexpr_1$ through $iexpr_k$) are all integer expressions.

4.2.5 Memory and Clocking

Combinational circuits, by their very nature, do not store any information. Instead, they simply react to the signals at their inputs, generating outputs equal to some function of the inputs. To create *sequential circuits*—that is, systems that have state and perform computations on that state—we must introduce devices that store information represented as bits. Our storage devices are all controlled by a single *clock*, a periodic signal that determines when new values are to be loaded into the devices. We consider two classes of memory devices:

Clocked registers (or simply *registers*) store individual bits or words. The clock signal controls the loading of the register with the value at its input.

Random access memories (or simply *memories*) store multiple words, using an address to select which word should be read or written. Examples of random access memories include (1) the virtual memory system of a processor, where a combination of hardware and operating system software make it appear to a processor that it can access any word within a large address space; and (2) the register file, where register identifiers serve as the addresses. In a Y86-64 processor, the register file holds the 15 program registers (`%rax` through `%r14`).

As we can see, the word “register” means two slightly different things when speaking of hardware versus machine-language programming. In hardware, a register is directly connected to the rest of the circuit by its input and output wires. In machine-level programming, the registers represent a small collection of addressable words in the CPU, where the addresses consist of register IDs. These words are generally stored in the register file, although we will see that the hardware can sometimes pass a word directly from one instruction to another to

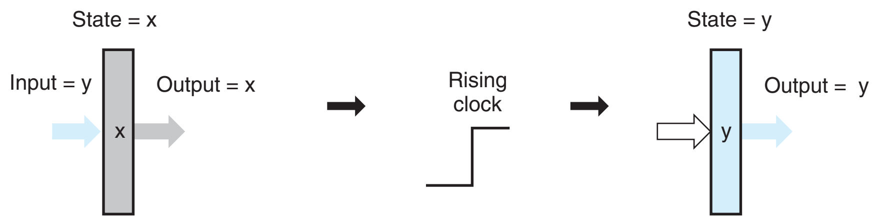
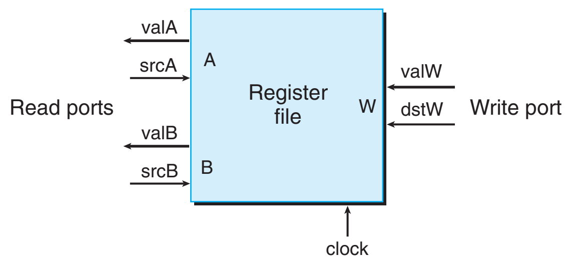


Figure 4.16 Register operation. The register outputs remain held at the current register state until the clock signal rises. When the clock rises, the values at the register inputs are captured to become the new register state.

avoid the delay of first writing and then reading the register file. When necessary to avoid ambiguity, we will call the two classes of registers "hardware registers" and "program registers," respectively.

Figure 4.16 gives a more detailed view of a hardware register and how it operates. For most of the time, the register remains in a fixed state (shown as x), generating an output equal to its current state. Signals propagate through the combinational logic preceding the register, creating a new value for the register input (shown as y), but the register output remains fixed as long as the clock is low. As the clock rises, the input signals are loaded into the register as its next state (y), and this becomes the new register output until the next rising clock edge. A key point is that the registers serve as barriers between the combinational logic in different parts of the circuit. Values only propagate from a register input to its output once every clock cycle at the rising clock edge. Our Y86-64 processors will use clocked registers to hold the program counter (PC), the condition codes (CC), and the program status (Stat).

The following diagram shows a typical register file:

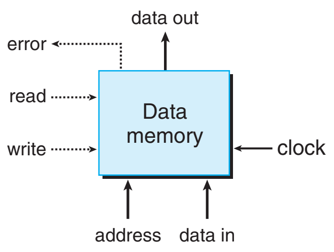


This register file has two *read ports*, named A and B, and one *write port*, named W. Such a *multiported* random access memory allows multiple read and write operations to take place simultaneously. In the register file diagrammed, the circuit can read the values of two program registers and update the state of a third. Each port has an address input, indicating which program register should be selected, and a data output or input giving a value for that program register. The addresses are register identifiers, using the encoding shown in Figure 4.4. The two read ports have address inputs srcA and srcB (short for "source A" and "source B") and data outputs valA and valB (short for “value A” and “value B”). The write port has address input dstW (short for “destination W”) and data input valW (short for “value W”).

The register file is not a combinational circuit, since it has internal storage. In our implementation, however, data can be read from the register file as if it were a block of combinational logic having addresses as inputs and the data as outputs. When either srcA or srcB is set to some register ID, then, after some delay, the value stored in the corresponding program register will appear on either valA or valB. For example, setting srcA to 3 will cause the value of program register `%rbx` to be read, and this value will appear on output valA.

The writing of words to the register file is controlled by the clock signal in a manner similar to the loading of values into a clocked register. Every time the clock rises, the value on input valW is written to the program register indicated by the register ID on input dstW. When dstW is set to the special ID value `0xF`, no program register is written. Since the register file can be both read and written, a natural question to ask is, “What happens if the circuit attempts to read and write the same register simultaneously?” The answer is straightforward: if the same register ID is used for both a read port and the write port, then, as the clock rises, there will be a transition on the read port’s data output from the old value to the new. When we incorporate the register file into our processor design, we will make sure that we take this property into consideration.

Our processor has a random access memory for storing program data, as illustrated below:



This memory has a single address input, a data input for writing, and a data output for reading. Like the register file, reading from our memory operates in a manner similar to combinational logic: If we provide an address on the address input and set the write control signal to 0, then after some delay, the value stored at that address will appear on data out. The error signal will be set to 1 if the address is out of range, and to 0 otherwise. Writing to the memory is controlled by the clock: We set address to the desired address, data in to the desired value, and write to 1. When we then operate the clock, the specified location in the memory will be updated, as long as the address is valid. As with the read operation, the error signal will be set to 1 if the address is invalid. This signal is generated by combinational logic, since the required bounds checking is purely a function of the address input and does not involve saving any state.

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

```
0x000: 30f20900000000000000 |     irmovq $9,  %rdx
0x00a: 30f31500000000000000 |     irmovq $21, %rbx
0x014: 6123                 |     subq %rdx, %rbx          # subtract
0x016: 30f48000000000000000 |     irmovq $128,%rsp         # Problem 4.13
0x020: 40436400000000000000 |     rmmovq %rsp, 100(%rbx)   # store
0x02a: a02f                 |     pushq %rdx               # push
0x02c: b00f                 |     popq  %rax               # Problem 4.14
0x02e: 734000000000000000   |     je done                  # Not taken
0x037: 804100000000000000   |     call proc                # Problem 4.18
0x040:                      | done:
0x040: 00                   |     halt
0x041:                      | proc:
0x041: 90                   |     ret                      # Return
                            |
```

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 $\leftarrow$ $M_1[PC]$ 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	

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]	icode : ifun $\leftarrow$ M_1[`0x014`] = `6:1`
	rA : rB $\leftarrow$ M_1[PC + 1]	rA : rB $\leftarrow$ M_1[`0x015`] = `2:3`
	valP $\leftarrow$ PC + 2	valP $\leftarrow$ `0x014` + 2 = `0x016`
Decode	valA $\leftarrow$ R[rA]	valA $\leftarrow$ R[`%rdx`] = 9
	valB $\leftarrow$ R[rB]	valB $\leftarrow$ R[`%rbx`] = 21
Execute	valE $\leftarrow$ valB OP valA	valE $\leftarrow$ 21 − 9 = 12
	Set CC	`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 $\leftarrow$ $M_1[PC]$	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

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 $\leftarrow$ 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[0x020] = 4:0 rA:rB $\leftarrow$ M_1[0x021] = 4:3 valC $\leftarrow$ M_8[0x022] = 100 valP $\leftarrow$ 0x020 + 10 = 0x02a
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	valA $\leftarrow$ R[%rsp] = 128 valB $\leftarrow$ R[%rbx] = 12
Execute	valE $\leftarrow$ valB + valC	valE $\leftarrow$ 12 + 100 = 112
Memory	M_8[valE] $\leftarrow$ valA	M_8[112] $\leftarrow$ 128
Write back		
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ 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 $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valP $\leftarrow$ PC + 2	

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?

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

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] valP $\leftarrow$ PC + 9	valC $\leftarrow$ M_8[`0x02f`] = `0x040` 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`] valB $\leftarrow$ R[`%rsp`]	valA $\leftarrow$ R[`%rsp`] = 120 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.

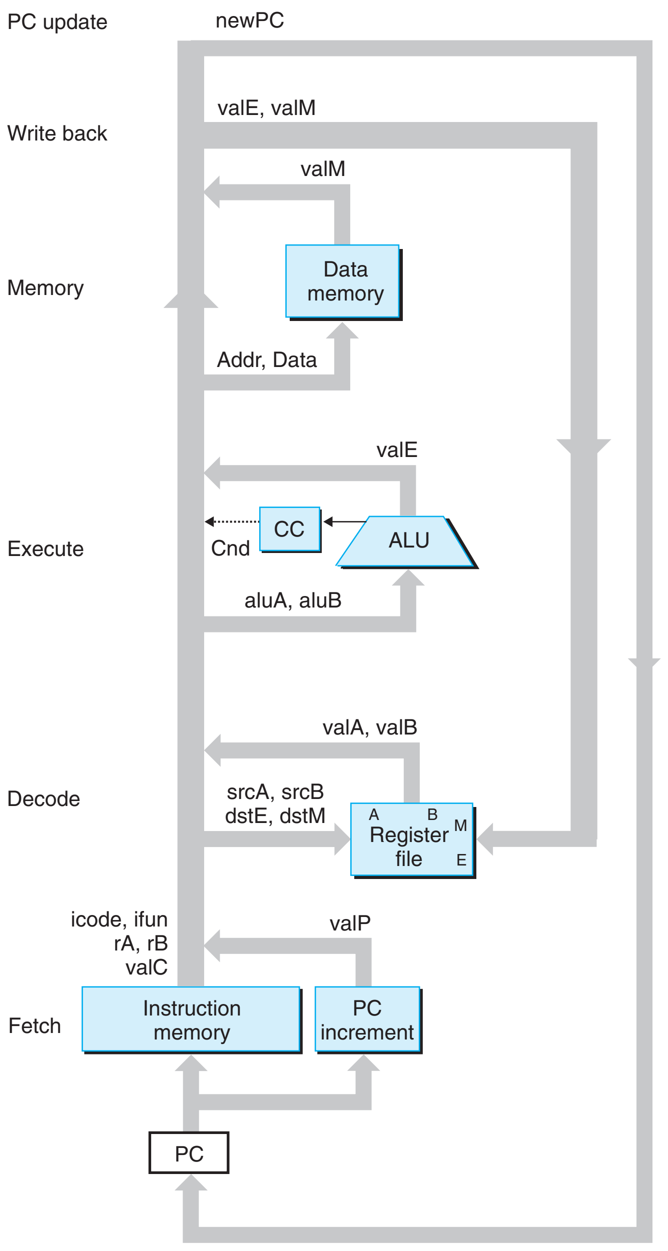


Figure 4.22 **Abstract view of SEQ, a sequential implementation.** The information processed during execution of an instruction follows a clockwise flow starting with an instruction fetch using the program counter (PC), shown in the lower left-hand corner of the figure.

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

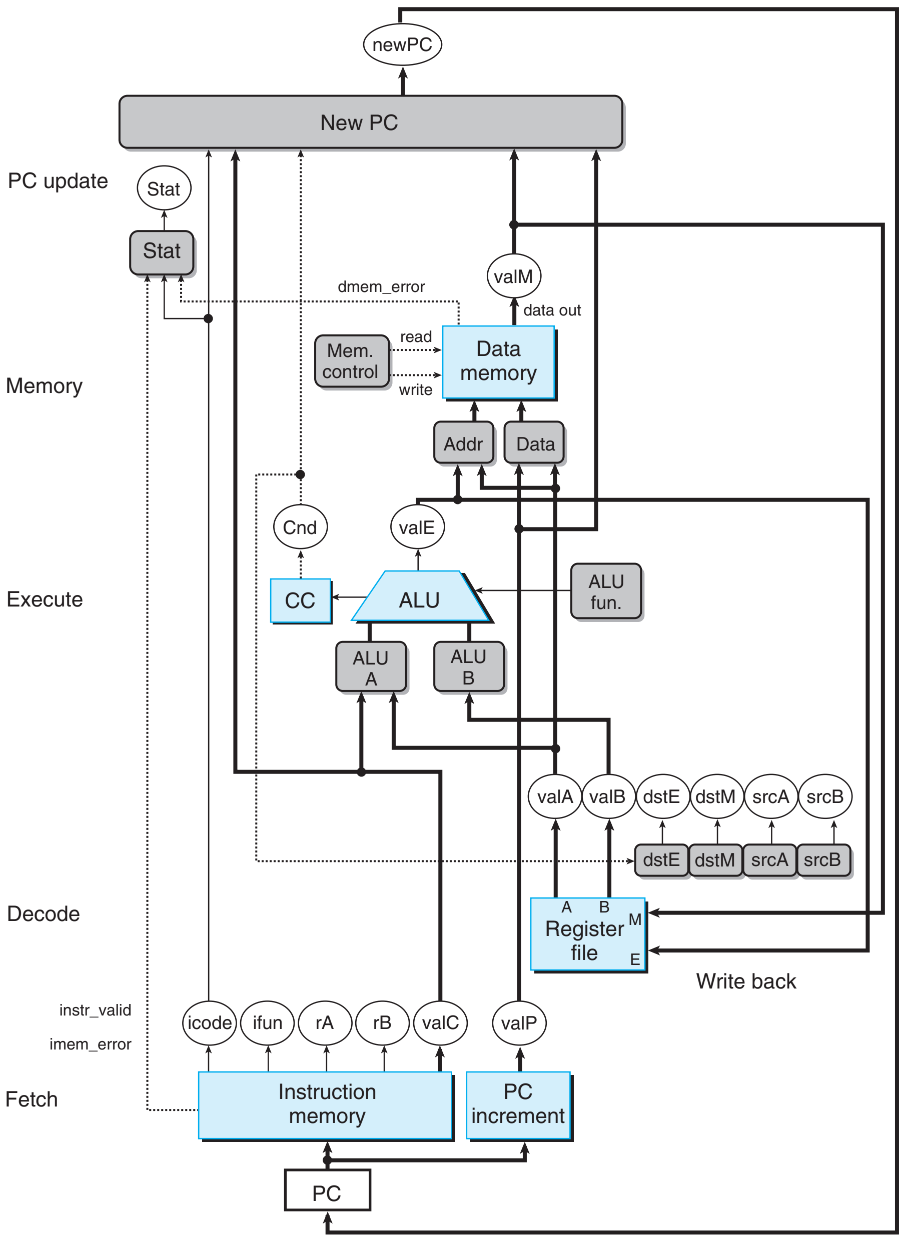


Figure 4.23 Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.

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

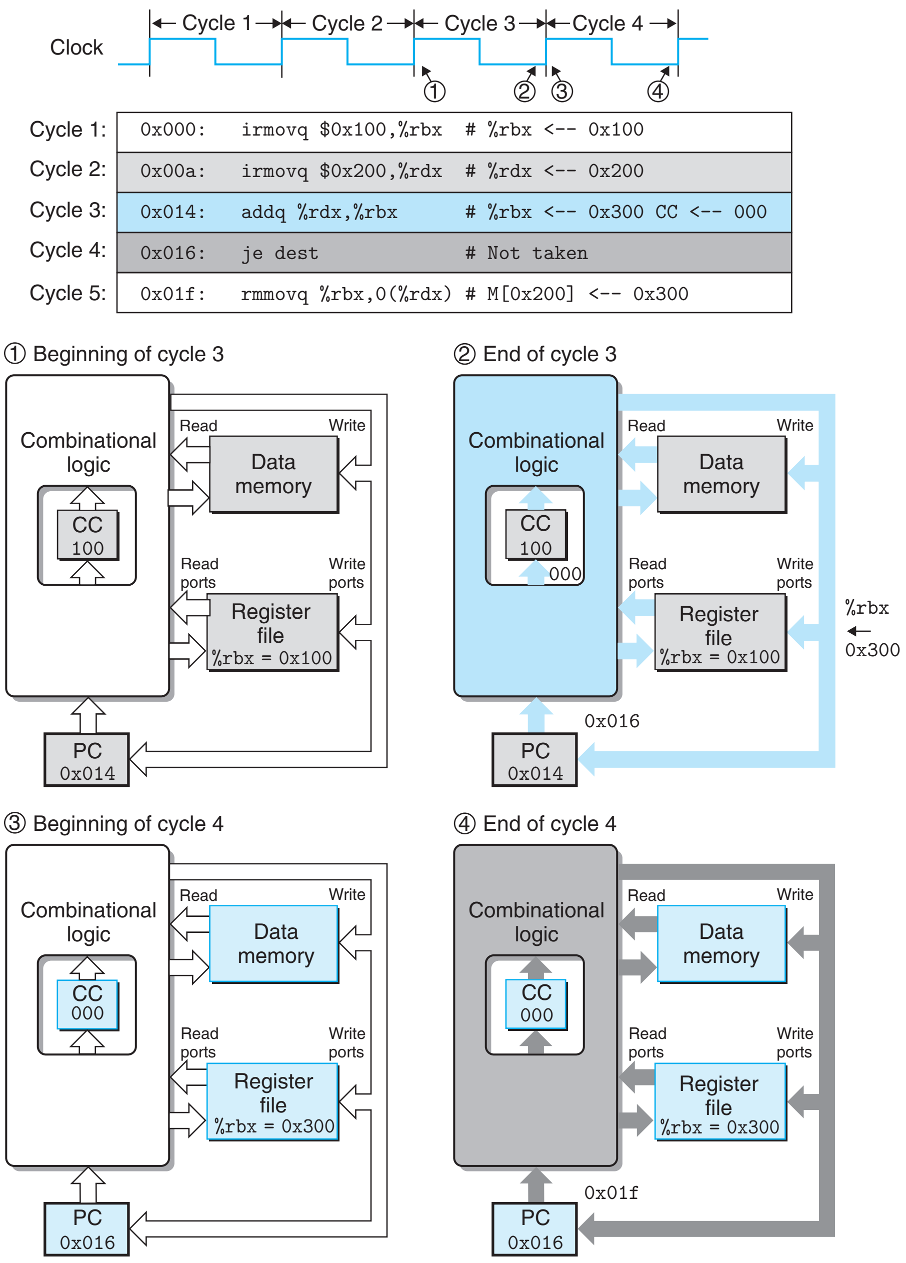
```
1    0x000:   irmovq $0x100,%rbx   # %rbx <-- 0x100
2    0x00a:   irmovq $0x200,%rdx   # %rdx <-- 0x200
3    0x014:   addq %rdx,%rbx       # %rbx <-- 0x300 CC <-- 000
4    0x016:   je dest              # Not taken
5    0x01f:   rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300
6    0x029: dest: halt
```

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.



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

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.

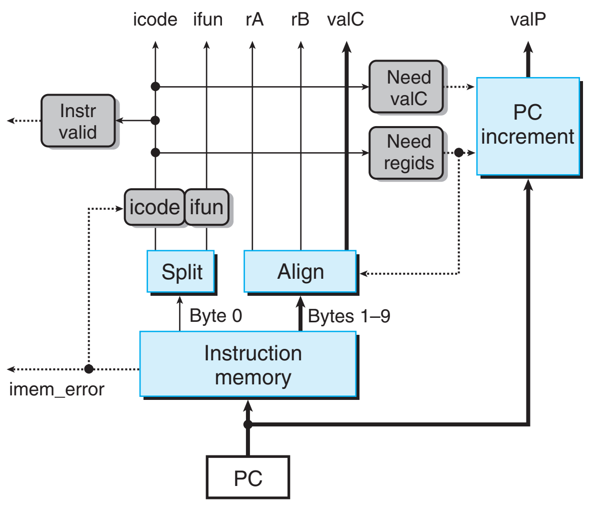


Figure 4.27
SEQ fetch stage. Six bytes are read from the instruction memory using the PC as the starting address. From these bytes, we generate the different instruction fields. The PC increment block computes signal valP.

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.

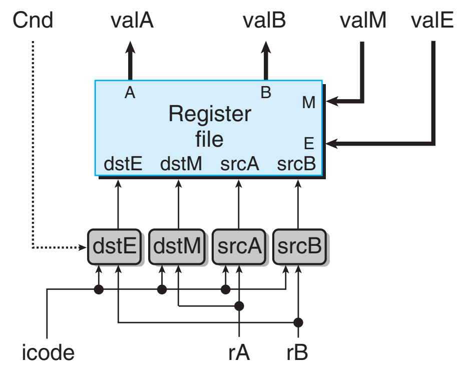


Figure 4.28
SEQ decode and write-back stage. The instruction fields are decoded to generate register identifiers for four addresses (two read and two write) used by the register file. The values read from the register file become the signals valA and valB. The two write-back values valE and valM serve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

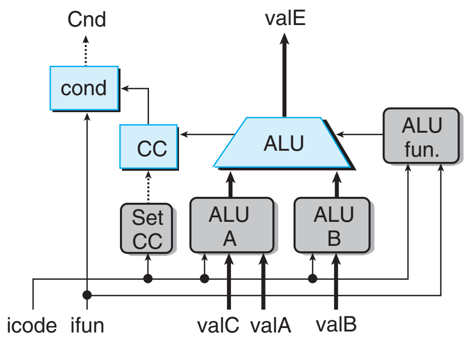


Figure 4.29
SEQ execute stage. The ALU either performs the operation for an integer operation instruction or acts as an adder. The condition code registers are set according to the ALU value. The condition code values are tested to determine whether a branch should be taken.

```
        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

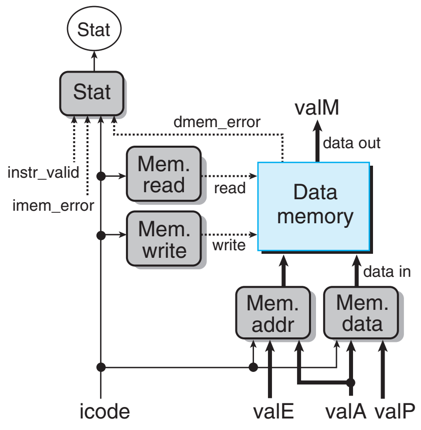


Figure 4.30
SEQ memory stage. The data memory can either write or read memory values. The value read from memory forms the signal valM.

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.

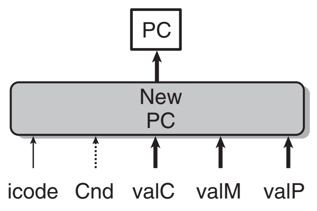


Figure 4.31
SEQ PC update stage. The next value of the PC is selected from among the signals valC, valM, and valP, depending on the instruction code and the branch flag.

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

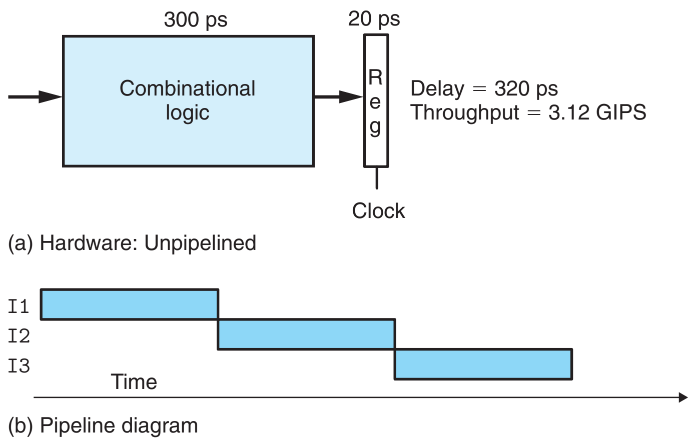


Figure 4.32
Unpipelined computation hardware. On each 320 ps cycle, the system spends 300 ps evaluating a combinational logic function and 20 ps storing the results in an output register.

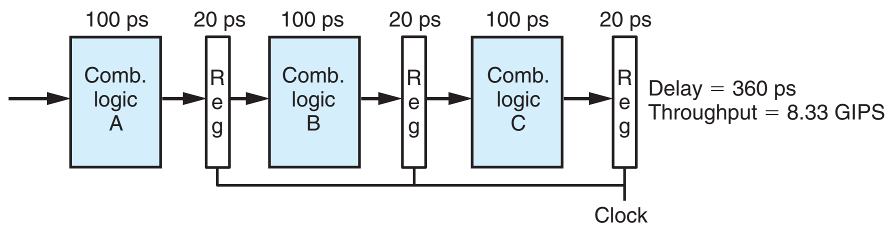
the logic decodes these to generate audio signals. The computational block in the figure is implemented as combinational logic, meaning that the signals will pass through a series of logic gates, with the outputs becoming some function of the inputs after some time delay.

In contemporary logic design, we measure circuit delays in units of *picoseconds* (abbreviated "ps"), or 10^{-12} seconds. In this example, we assume the combinational logic requires 300 ps, while the loading of the register requires 20 ps. Figure 4.32 shows a form of timing diagram known as a *pipeline diagram*. In this diagram, time flows from left to right. A series of instructions (here named I1, I2, and I3) are written from top to bottom. The solid rectangles indicate the times during which these instructions are executed. In this implementation, we must complete one instruction before beginning the next. Hence, the boxes do not overlap one another vertically. The following formula gives the maximum rate at which we could operate the system:

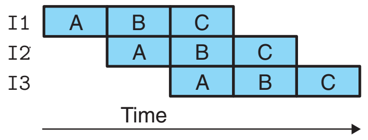
$$\mathit{Throughput} = \frac{1 \text{ instruction}}{(20+300) \text{ picoseconds}} \cdot \frac{1{,}000 \text{ picoseconds}}{1 \text{ nanosecond}} \approx 3.12 \text{ GIPS}$$

We express throughput in units of giga-instructions per second (abbreviated GIPS), or billions of instructions per second. The total time required to perform a single instruction from beginning to end is known as the *latency*. In this system, the latency is 320 ps, the reciprocal of the throughput.

Suppose we could divide the computation performed by our system into three stages, A, B, and C, where each requires 100 ps, as illustrated in Figure 4.33. Then we could put *pipeline registers* between the stages so that each instruction moves through the system in three steps, requiring three complete clock cycles from beginning to end. As the pipeline diagram in Figure 4.33 illustrates, we could allow I2 to enter stage A as soon as I1 moves from A to B, and so on. In steady state, all three stages would be active, with one instruction leaving and a new one entering the system every clock cycle. We can see this during the third clock cycle in the pipeline diagram where I1 is in stage C, I2 is in stage B, and I3 is in stage A. In



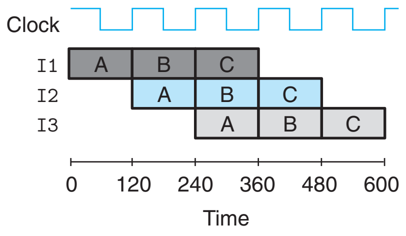
(a) Hardware: Three-stage pipeline



(b) Pipeline diagram

Figure 4.33 Three-stage pipelined computation hardware. The computation is split into stages A, B, and C. On each 120 ps cycle, each instruction progresses through one stage.

Figure 4.34 Three-stage pipeline timing. The rising edge of the clock signal controls the movement of instructions from one pipeline stage to the next.



this system, we could cycle the clocks every 100 + 20 = 120 picoseconds, giving a throughput of around 8.33 GIPS. Since processing a single instruction requires 3 clock cycles, the latency of this pipeline is $3 \times 120 = 360$ ps. We have increased the throughput of the system by a factor of $8.33/3.12 = 2.67$ at the expense of some added hardware and a slight increase in the latency ($360/320 = 1.12$). The increased latency is due to the time overhead of the added pipeline registers.

4.4.2 A Detailed Look at Pipeline Operation

To better understand how pipelining works, let us look in some detail at the timing and operation of pipeline computations. Figure 4.34 shows the pipeline diagram for the three-stage pipeline we have already looked at (Figure 4.33). The transfer of the instructions between pipeline stages is controlled by a clock signal, as shown above the pipeline diagram. Every 120 ps, this signal rises from 0 to 1, initiating the next set of pipeline stage evaluations.

Figure 4.35 traces the circuit activity between times 240 and 360, as instruction `I1` (shown in dark gray) propagates through stage C, `I2` (shown in blue)

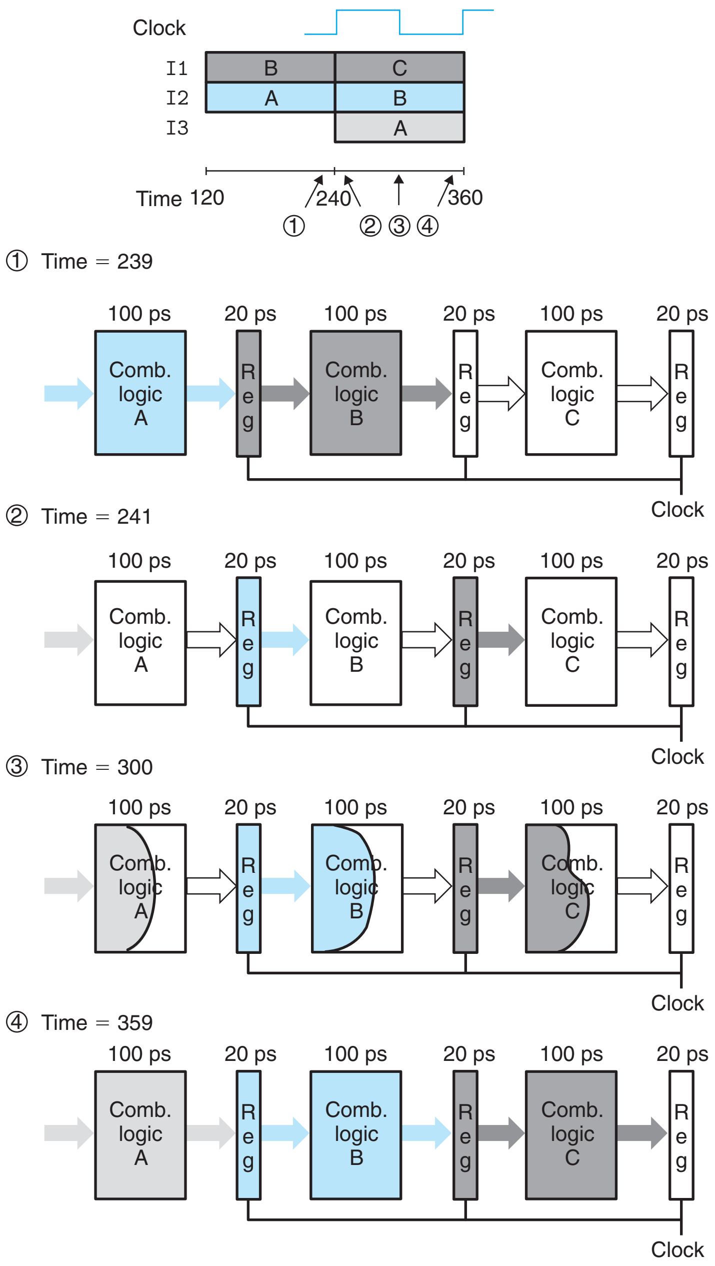


Figure 4.35 One clock cycle of pipeline operation. Just before the clock rises at time 240 (point 1), instructions I1 (shown in dark gray) and I2 (shown in blue) have completed stages B and A. After the clock rises, these instructions begin propagating through stages C and B, while instruction I3 (shown in light gray) begins propagating through stage A (points 2 and 3). Just before the clock rises again, the results for the instructions have propagated to the inputs of the pipeline registers (point 4).

propagates through stage B, and I3 (shown in light gray) propagates through stage A. Just before the rising clock at time 240 (point 1), the values computed in stage A for instruction I2 have reached the input of the first pipeline register, but its state and output remain set to those computed during stage A for instruction I1. The values computed in stage B for instruction I1 have reached the input of the second pipeline register. As the clock rises, these inputs are loaded into the pipeline registers, becoming the register outputs (point 2). In addition, the input to stage A is set to initiate the computation of instruction I3. The signals then propagate through the combinational logic for the different stages (point 3). As the curved wave fronts in the diagram at point 3 suggest, signals can propagate through different sections at different rates. Before time 360, the result values reach the inputs of the pipeline registers (point 4). When the clock rises at time 360, each of the instructions will have progressed through one pipeline stage.

We can see from this detailed view of pipeline operation that slowing down the clock would not change the pipeline behavior. The signals propagate to the pipeline register inputs, but no change in the register states will occur until the clock rises. On the other hand, we could have disastrous effects if the clock were run too fast. The values would not have time to propagate through the combinational logic, and so the register inputs would not yet be valid when the clock rises.

As with our discussion of the timing for the SEQ processor (Section 4.3.3), we see that the simple mechanism of having clocked registers between blocks of combinational logic suffices to control the flow of instructions in the pipeline. As the clock rises and falls repeatedly, the different instructions flow through the stages of the pipeline without interfering with one another.

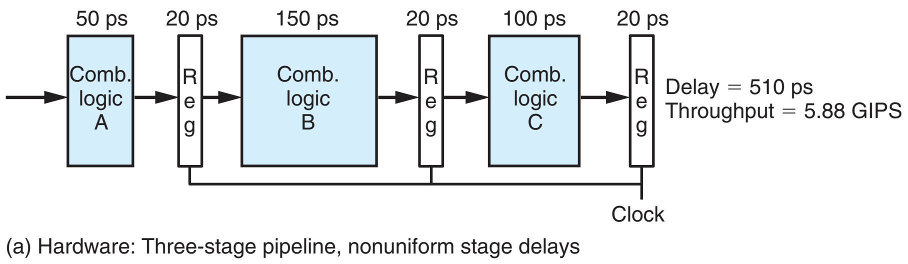
4.4.3 Limitations of Pipelining

The example of Figure 4.33 shows an ideal pipelined system in which we are able to divide the computation into three independent stages, each requiring one-third of the time required by the original logic. Unfortunately, other factors often arise that diminish the effectiveness of pipelining.

Nonuniform Partitioning

Figure 4.36 shows a system in which we divide the computation into three stages as before, but the delays through the stages range from 50 to 150 ps. The sum of the delays through all of the stages remains 300 ps. However, the rate at which we can operate the clock is limited by the delay of the slowest stage. As the pipeline diagram in this figure shows, stage A will be idle (shown as a white box) for 100 ps every clock cycle, while stage C will be idle for 50 ps every clock cycle. Only stage B will be continuously active. We must set the clock cycle to $150 + 20 = 170$ picoseconds, giving a throughput of 5.88 GIPS. In addition, the latency would increase to 510 ps due to the slower clock rate.

Devising a partitioning of the system computation into a series of stages having uniform delays can be a major challenge for hardware designers. Often,



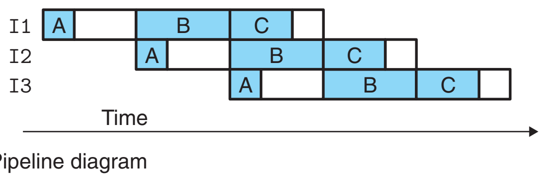
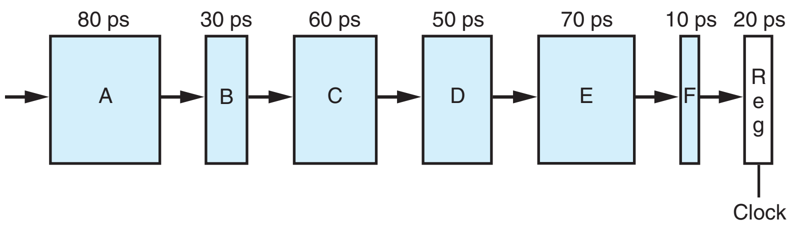


Figure 4.36 **Limitations of pipelining due to nonuniform stage delays.** The system throughput is limited by the speed of the slowest stage.

some of the hardware units in a processor, such as the ALU and the memories, cannot be subdivided into multiple units with shorter delay. This makes it difficult to create a set of balanced stages. We will not concern ourselves with this level of detail in designing our pipelined Y86-64 processor, but it is important to appreciate the importance of timing optimization in actual system design.

Practice Problem 4.28 (solution page 525)

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 a 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 the throughput of a three-stage pipeline? What would 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 minimum number of stages that would yield a design with the maximum achievable throughput? Describe this design, its throughput, and its latency.

Diminishing Returns of Deep Pipelining

Figure 4.37 illustrates another limitation of pipelining. In this example, we have divided the computation into six stages, each requiring 50 ps. Inserting a pipeline register between each pair of stages yields a six-stage pipeline. The minimum clock period for this system is $50 + 20 = 70$ picoseconds, giving a throughput of 14.29 GIPS. Thus, in doubling the number of pipeline stages, we improve the performance by a factor of $14.29/8.33 = 1.71$. Even though we have cut the time required for each computation block by a factor of 2, we do not get a doubling of the throughput, due to the delay through the pipeline registers. This delay becomes a limiting factor in the throughput of the pipeline. In our new design, this delay consumes 28.6% of the total clock period.

Modern processors employ very deep pipelines (15 or more stages) in an attempt to maximize the processor clock rate. The processor architects divide the instruction execution into a large number of very simple steps so that each stage can have a very small delay. The circuit designers carefully design the pipeline registers to minimize their delay. The chip designers must also carefully design the clock distribution network to ensure that the clock changes at the exact same time across the entire chip. All of these factors contribute to the challenge of designing high-speed microprocessors.

Practice Problem 4.29 (solution page 526)

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.

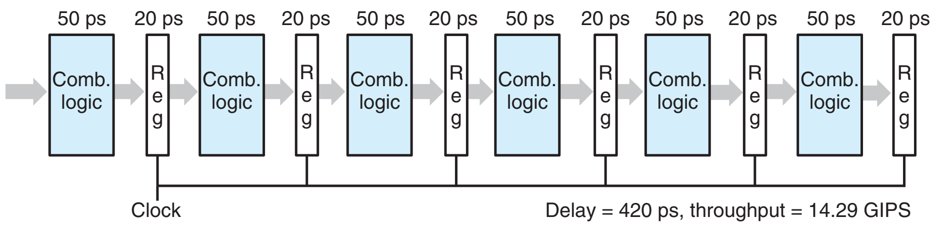


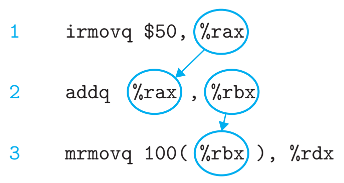
Figure 4.37 Limitations of pipelining due to overhead. As the combinational logic is split into shorter blocks, the delay due to register updating becomes a limiting factor.

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?

4.4.4 Pipelining a System with Feedback

Up to this point, we have considered only systems in which the objects passing through the pipeline—whether cars, people, or instructions—are completely independent of one another. For a system that executes machine programs such as x86-64 or Y86-64, however, there are potential dependencies between successive instructions. For example, consider the following Y86-64 instruction sequence:



```
1    irmovq $50, %rax
2    addq %rax , %rbx
3    mrmovq 100( %rbx ), %rdx
```

In this three-instruction sequence, there is a *data dependency* between each successive pair of instructions, as indicated by the circled register names and the arrows between them. The `irmovq` instruction (line 1) stores its result in `%rax`, which then must be read by the `addq` instruction (line 2); and this instruction stores its result in `%rbx`, which must then be read by the `mrmovq` instruction (line 3).

Another source of sequential dependencies occurs due to the instruction control flow. Consider the following Y86-64 instruction sequence:

```
1   loop:
2       subq %rdx,%rbx
3       jne targ
4       irmovq $10,%rdx
5       jmp loop
6   targ:
7       halt
```

The `jne` instruction (line 3) creates a *control dependency* since the outcome of the conditional test determines whether the next instruction to execute will be the `irmovq` instruction (line 4) or the `halt` instruction (line 7). In our design for SEQ, these dependencies were handled by the feedback paths shown on the right-hand side of Figure 4.22. This feedback brings the updated register values down to the register file and the new PC value down to the PC register.

Figure 4.38 illustrates the perils of introducing pipelining into a system containing feedback paths. In the original system (Figure 4.38(a)), the result of each

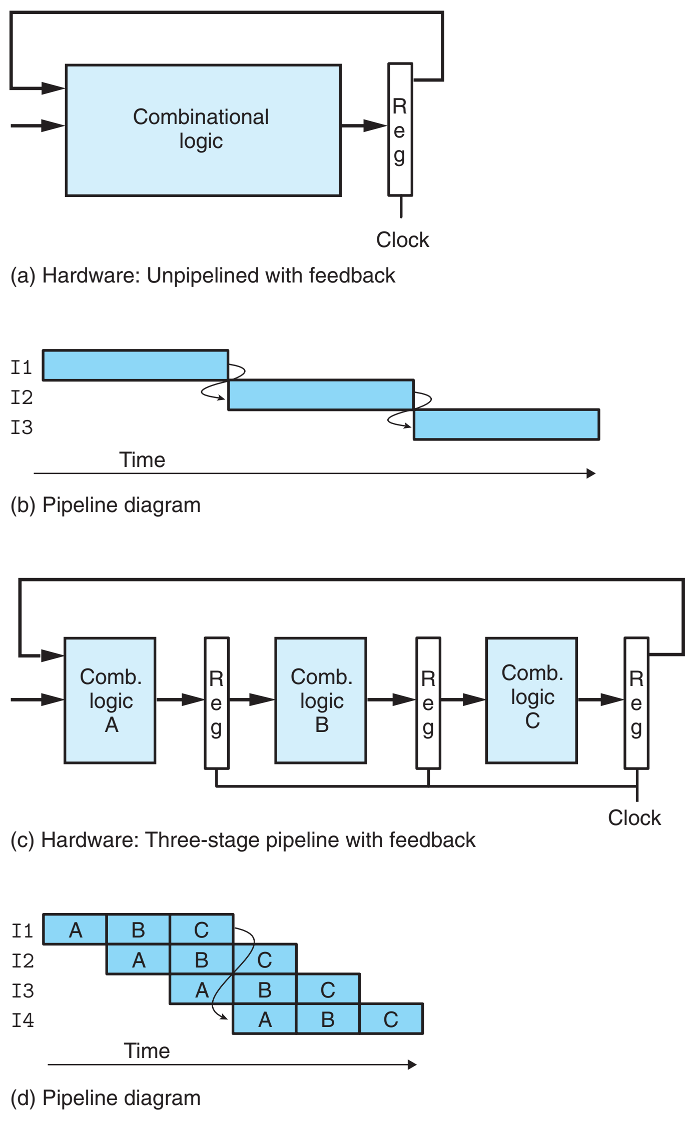


Figure 4.38 Limitations of pipelining due to logical dependencies. In going from an unpipelined system with feedback (a) to a pipelined one (c), we change its computational behavior, as can be seen by the two pipeline diagrams (b and d).

instruction is fed back around to the next instruction. This is illustrated by the pipeline diagram (Figure 4.38(b)), where the result of I1 becomes an input to I2, and so on. If we attempt to convert this to a three-stage pipeline in the most straightforward manner (Figure 4.38(c)), we change the behavior of the system. As Figure 4.38(c) shows, the result of I1 becomes an input to I4. In attempting to speed up the system via pipelining, we have changed the system behavior.

When we introduce pipelining into a Y86-64 processor, we must deal with feedback effects properly. Clearly, it would be unacceptable to alter the system behavior as occurred in the example of Figure 4.38. Somehow we must deal with the data and control dependencies between instructions so that the resulting behavior matches the model defined by the ISA.

4.5 Pipelined Y86-64 Implementations

We are finally ready for the major task of this chapter—designing a pipelined Y86-64 processor. We start by making a small adaptation of the sequential processor SEQ to shift the computation of the PC into the fetch stage. We then add pipeline registers between the stages. Our first attempt at this does not handle the different data and control dependencies properly. By making some modifications, however, we achieve our goal of an efficient pipelined processor that implements the Y86-64 ISA.

4.5.1 SEQ+: Rearranging the Computation Stages

As a transitional step toward a pipelined design, we must slightly rearrange the order of the five stages in SEQ so that the PC update stage comes at the beginning of the clock cycle, rather than at the end. This transformation requires only minimal change to the overall hardware structure, and it will work better with the sequencing of activities within the pipeline stages. We refer to this modified design as SEQ+.

We can move the PC update stage so that its logic is active at the beginning of the clock cycle by making it compute the PC value for the *current* instruction. Figure 4.39 shows how SEQ and SEQ+ differ in their PC computation. With SEQ (Figure 4.39(a)), the PC computation takes place at the end of the clock cycle, computing the new value for the PC register based on the values of signals computed during the current clock cycle. With SEQ+ (Figure 4.39(b)), we create state registers to hold the signals computed during an instruction. Then, as a new clock cycle begins, the values propagate through the exact same logic to compute the PC for the now-current instruction. We label the registers "pIcode,"

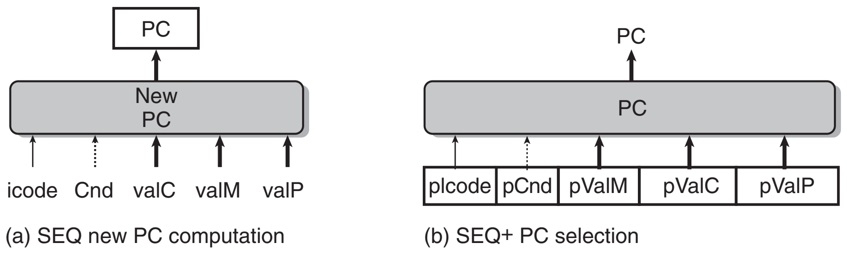


Figure 4.39 Shifting the timing of the PC computation. With SEQ+, we compute the value of the program counter for the current state as the first step in instruction execution.

Aside Where is the PC in SEQ+?

One curious feature of SEQ+ is that there is no hardware register storing the program counter. Instead, the PC is computed dynamically based on some state information stored from the previous instruction. This is a small illustration of the fact that we can implement a processor in a way that differs from the conceptual model implied by the ISA, as long as the processor correctly executes arbitrary machine-language programs. We need not encode the state in the form indicated by the programmer-visible state, as long as the processor can generate correct values for any part of the programmer-visible state (such as the program counter). We will exploit this principle even more in creating a pipelined design. Out-of-order processing techniques, as described in Section 5.7, take this idea to an extreme by executing instructions in a completely different order than they occur in the machine-level program.

"pCnd," and so on, to indicate that on any given cycle, they hold the control signals generated during the previous cycle.

Figure 4.40 shows a more detailed view of the SEQ+ hardware. We can see that it contains the exact same hardware units and control blocks that we had in SEQ (Figure 4.23), but with the PC logic shifted from the top, where it was active at the end of the clock cycle, to the bottom, where it is active at the beginning.

The shift of state elements from SEQ to SEQ+ is an example of a general transformation known as *circuit retiming* [68]. Retiming changes the state representation for a system without changing its logical behavior. It is often used to balance the delays between the different stages of a pipelined system.

4.5.2 Inserting Pipeline Registers

In our first attempt at creating a pipelined Y86-64 processor, we insert pipeline registers between the stages of SEQ+ and rearrange signals somewhat, yielding the PIPE− processor, where the "−" in the name signifies that this processor has somewhat less performance than our ultimate processor design. The structure of PIPE− is illustrated in Figure 4.41. The pipeline registers are shown in this figure as blue boxes, each containing different fields that are shown as white boxes. As indicated by the multiple fields, each pipeline register holds multiple bytes and words. Unlike the labels shown in rounded boxes in the hardware structure of the two sequential processors (Figures 4.23 and 4.40), these white boxes represent actual hardware components.

Observe that PIPE− uses nearly the same set of hardware units as our sequential design SEQ (Figure 4.40), but with the pipeline registers separating the stages. The differences between the signals in the two systems is discussed in Section 4.5.3.

The pipeline registers are labeled as follows:

F holds a *predicted* value of the program counter, as will be discussed shortly.

D sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.

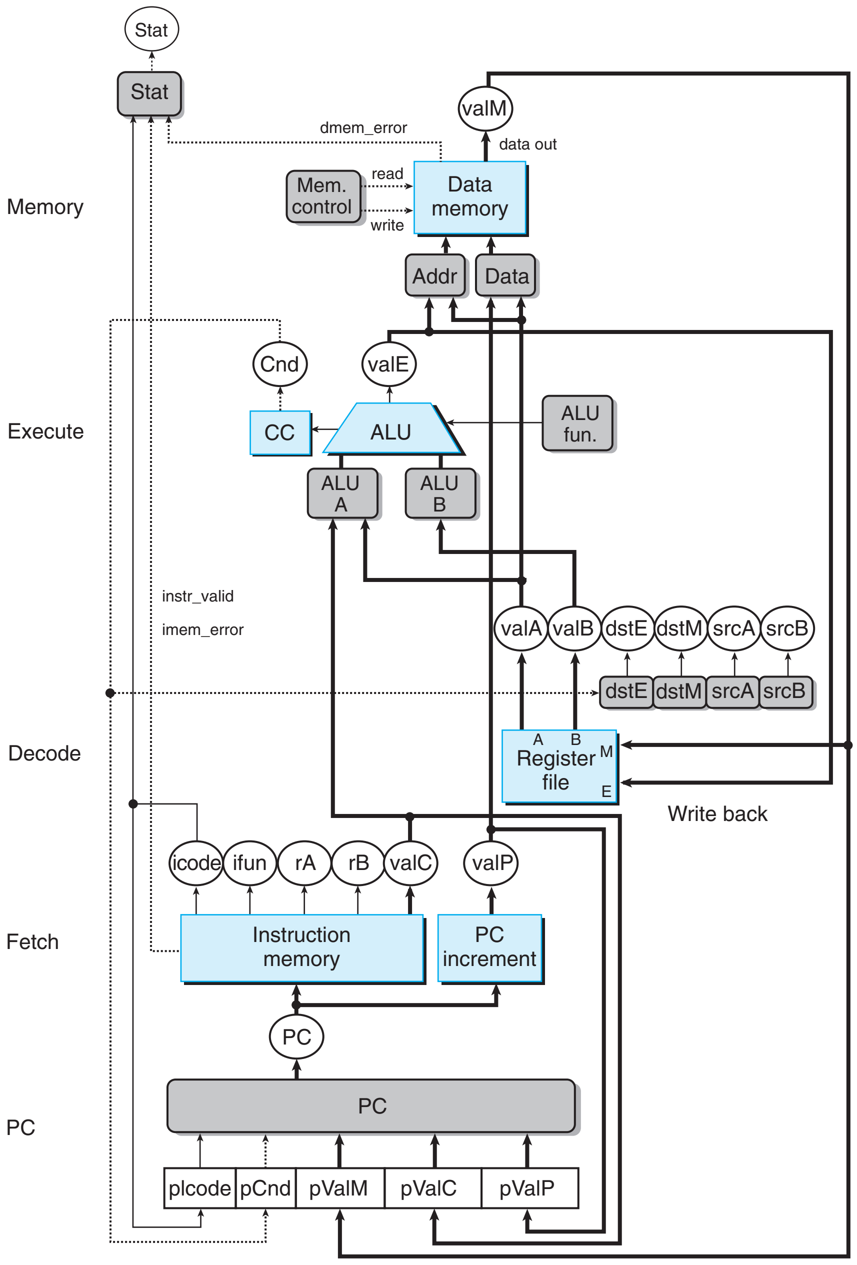


Figure 4.40 **SEQ+ hardware structure.** Shifting the PC computation from the end of the clock cycle to the beginning makes it more suitable for pipelining.

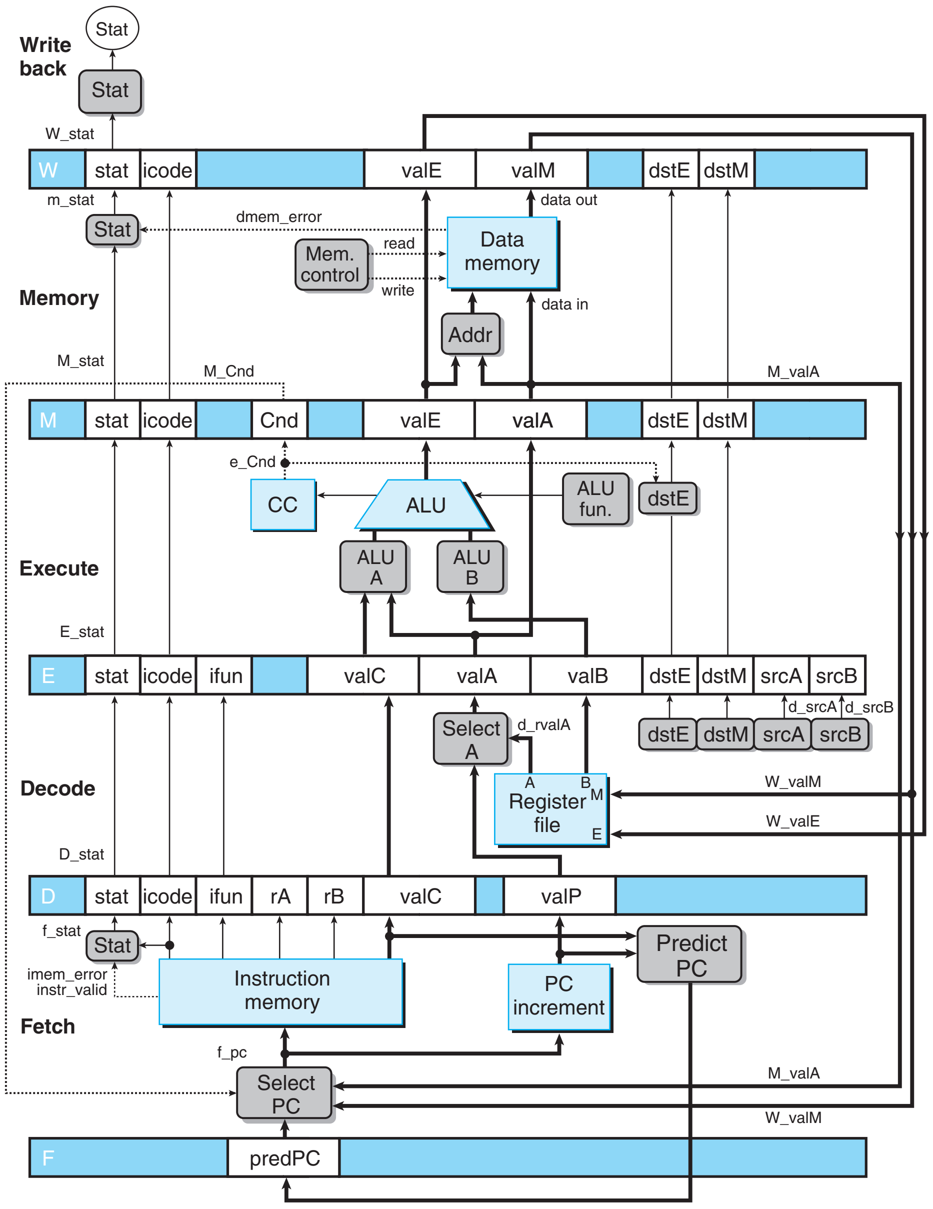


Figure 4.41 **Hardware structure of PIPE−, an initial pipelined implementation.** By inserting pipeline registers into SEQ+ (Figure 4.40), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.

E sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.

M sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.

W sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a `ret` instruction.

Figure 4.42 shows how the following code sequence would flow through our five-stage pipeline, where the comments identify the instructions as `I1` to `I5` for reference:

```
    irmovq  $1,%rax   # I1
    irmovq  $2,%rbx   # I2
    irmovq  $3,%rcx   # I3
    irmovq  $4,%rdx   # I4
    halt              # I5
```

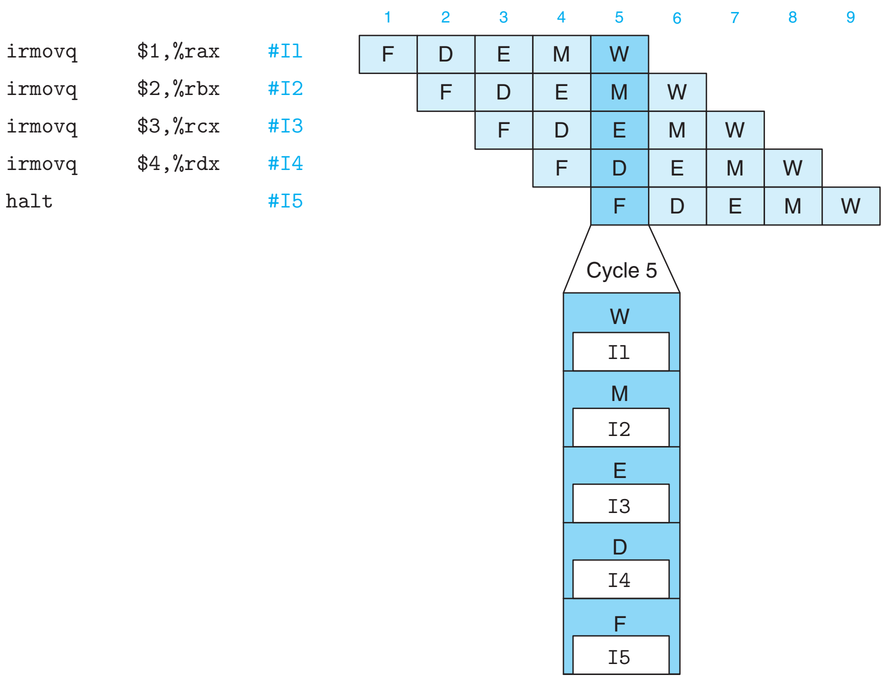


Figure 4.42 Example of instruction flow through pipeline.

The right side of the figure shows a pipeline diagram for this instruction sequence. As with the pipeline diagrams for the simple pipelined computation units of Section 4.4, this diagram shows the progression of each instruction through the pipeline stages, with time increasing from left to right. The numbers along the top identify the clock cycles at which the different stages occur. For example, in cycle 1, instruction `I1` is fetched, and it then proceeds through the pipeline stages, with its result being written to the register file after the end of cycle 5. Instruction `I2` is fetched in cycle 2, and its result is written back after the end of cycle 6, and so on. At the bottom, we show an expanded view of the pipeline for cycle 5. At this point, there is an instruction in each of the pipeline stages.

From Figure 4.42, we can also justify our convention of drawing processors so that the instructions flow from bottom to top. The expanded view for cycle 5 shows the pipeline stages with the fetch stage on the bottom and the write-back stage on the top, just as do our diagrams of the pipeline hardware (Figure 4.41). If we look at the ordering of instructions in the pipeline stages, we see that they appear in the same order as they do in the program listing. Since normal program flow goes from top to bottom of a listing, we preserve this ordering by having the pipeline flow go from bottom to top. This convention is particularly useful when working with the simulators that accompany this text.

4.5.3 Rearranging and Relabeling Signals

Our sequential implementations SEQ and SEQ+ only process one instruction at a time, and so there are unique values for signals such as valC, srcA, and valE. In our pipelined design, there will be multiple versions of these values associated with the different instructions flowing through the system. For example, in the detailed structure of PIPE−, there are four white boxes labeled “Stat” that hold the status codes for four different instructions (see Figure 4.41). We need to take great care to make sure we use the proper version of a signal, or else we could have serious errors, such as storing the result computed for one instruction at the destination register specified by another instruction. We adopt a naming scheme where a signal stored in a pipeline register can be uniquely identified by prefixing its name with that of the pipe register written in uppercase. For example, the four status codes are named D_stat, E_stat, M_stat, and W_stat. We also need to refer to some signals that have just been computed within a stage. These are labeled by prefixing the signal name with the first character of the stage name, written in lowercase. Using the status codes as examples, we can see control logic blocks labeled “Stat” in the fetch and memory stages. The outputs of these blocks are therefore named f_stat and m_stat. We can also see that the actual status of the overall processor Stat is computed by a block in the write-back stage, based on the status value in pipeline register W.

The decode stages of SEQ+ and PIPE− both generate signals dstE and dstM indicating the destination register for values valE and valM. In SEQ+, we could connect these signals directly to the address inputs of the register file write ports. With PIPE−, these signals are carried along in the pipeline through the execute and memory stages and are directed to the register file only once they reach

Aside What is the difference between signals M_stat and m_stat?

With our naming system, the uppercase prefixes 'D', 'E', 'M', and 'W' refer to pipeline *registers*, and so M_stat refers to the status code field of pipeline register M. The lowercase prefixes 'f', 'd', 'e', 'm', and 'w' refer to the pipeline *stages*, and so m_stat refers to the status signal generated in the memory stage by a control logic block.

Understanding this naming convention is critical to understanding the operation of our pipelined processors.

the write-back stage (shown in the more detailed views of the stages). We do this to make sure the write port address and data inputs hold values from the same instruction. Otherwise, the write back would be writing the values for the instruction in the write-back stage, but with register IDs from the instruction in the decode stage. As a general principle, we want to keep all of the information about a particular instruction contained within a single pipeline stage.

One block of PIPE− that is not present in SEQ+ in the exact same form is the block labeled "Select A" in the decode stage. We can see that this block generates the value valA for the pipeline register E by choosing either valP from pipeline register D or the value read from the A port of the register file. This block is included to reduce the amount of state that must be carried forward to pipeline registers E and M. Of all the different instructions, only the `call` requires valP in the memory stage. Only the jump instructions require the value of valP in the execute stage (in the event the jump is not taken). None of these instructions requires a value read from the register file. Therefore, we can reduce the amount of pipeline register state by merging these two signals and carrying them through the pipeline as a single signal valA. This eliminates the need for the block labeled "Data" in SEQ (Figure 4.23) and SEQ+ (Figure 4.40), which served a similar purpose. In hardware design, it is common to carefully identify how signals get used and then reduce the amount of register state and wiring by merging signals such as these.

As shown in Figure 4.41, our pipeline registers include a field for the status code stat, initially computed during the fetch stage and possibly modified during the memory stage. We will discuss how to implement the processing of exceptional events in Section 4.5.6, after we have covered the implementation of normal instruction execution. Suffice it to say at this point that the most systematic approach is to associate a status code with each instruction as it passes through the pipeline, as we have indicated in the figure.

4.5.4 Next PC Prediction

We have taken some measures in the design of PIPE− to properly handle control dependencies. Our goal in the pipelined design is to *issue* a new instruction on every clock cycle, meaning that on each clock cycle, a new instruction proceeds into the execute stage and will ultimately be completed. Achieving this goal would

Aside Other branch prediction strategies

Our design uses an *always taken* branch prediction strategy. Studies show this strategy has around a 60% success rate [44, 122]. Conversely, a *never taken* (NT) strategy has around a 40% success rate. A slightly more sophisticated strategy, known as *backward taken, forward not taken* (BTFNT), predicts that branches to lower addresses than the next instruction will be taken, while those to higher addresses will not be taken. This strategy has a success rate of around 65%. This improvement stems from the fact that loops are closed by backward branches and loops are generally executed multiple times. Forward branches are used for conditional operations, and these are less likely to be taken. In Problems 4.55 and 4.56, you can modify the Y86-64 pipeline processor to implement the NT and BTFNT branch prediction strategies.

As we saw in Section 3.6.6, mispredicted branches can degrade the performance of a program considerably, thus motivating the use of conditional data transfer rather than conditional control transfer when possible.

yield a throughput of one instruction per cycle. To do this, we must determine the location of the next instruction right after fetching the current instruction. Unfortunately, if the fetched instruction is a conditional branch, we will not know whether or not the branch should be taken until several cycles later, after the instruction has passed through the execute stage. Similarly, if the fetched instruction is a `ret`, we cannot determine the return location until the instruction has passed through the memory stage.

With the exception of conditional jump instructions and `ret`, we can determine the address of the next instruction based on information computed during the fetch stage. For `call` and `jmp` (unconditional jump), it will be valC, the constant word in the instruction, while for all others it will be valP, the address of the next instruction. We can therefore achieve our goal of issuing a new instruction every clock cycle in most cases by *predicting* the next value of the PC. For most instruction types, our prediction will be completely reliable. For conditional jumps, we can predict either that a jump will be taken, so that the new PC value would be valC, or that it will not be taken, so that the new PC value would be valP. In either case, we must somehow deal with the case where our prediction was incorrect and therefore we have fetched and partially executed the wrong instructions. We will return to this matter in Section 4.5.8.

This technique of guessing the branch direction and then initiating the fetching of instructions according to our guess is known as *branch prediction*. It is used in some form by virtually all processors. Extensive experiments have been conducted on effective strategies for predicting whether or not branches will be taken [46, Section 2.3]. Some systems devote large amounts of hardware to this task. In our design, we will use the simple strategy of predicting that conditional branches are always taken, and so we predict the new value of the PC to be valC.

We are still left with predicting the new PC value resulting from a `ret` instruction. Unlike conditional jumps, we have a nearly unbounded set of possible

Aside Return address prediction with a stack

With most programs, it is very easy to predict return addresses, since procedure calls and returns occur in matched pairs. Most of the time that a procedure is called, it returns to the instruction following the call. This property is exploited in high-performance processors by including a hardware stack within the instruction fetch unit that holds the return address generated by procedure call instructions. Every time a procedure call instruction is executed, its return address is pushed onto the stack. When a return instruction is fetched, the top value is popped from this stack and used as the predicted return address. Like branch prediction, a mechanism must be provided to recover when the prediction was incorrect, since there are times when calls and returns do not match. In general, the prediction is highly reliable. This hardware stack is not part of the programmer-visible state.

results, since the return address will be whatever word is on the top of the stack. In our design, we will not attempt to predict any value for the return address. Instead, we will simply hold off processing any more instructions until the `ret` instruction passes through the write-back stage. We will return to this part of the implementation in Section 4.5.8.

The PIPE− fetch stage, diagrammed at the bottom of Figure 4.41, is responsible for both predicting the next value of the PC and selecting the actual PC for the instruction fetch. We can see the block labeled “Predict PC” can choose either valP (as computed by the PC incrementer) or valC (from the fetched instruction). This value is stored in pipeline register F as the *predicted* value of the program counter. The block labeled “Select PC” is similar to the block labeled “PC” in the SEQ+ PC selection stage (Figure 4.40). It chooses one of three values to serve as the address for the instruction memory: the predicted PC, the value of valP for a not-taken branch instruction that reaches pipeline register M (stored in register M_valA), or the value of the return address when a `ret` instruction reaches pipeline register W (stored in W_valM).

4.5.5 Pipeline Hazards

Our structure PIPE− is a good start at creating a pipelined Y86-64 processor. Recall from our discussion in Section 4.4.4, however, that introducing pipelining into a system with feedback can lead to problems when there are dependencies between successive instructions. We must resolve this issue before we can complete our design. These dependencies can take two forms: (1) *data* dependencies, where the results computed by one instruction are used as the data for a following instruction, and (2) *control* dependencies, where one instruction determines the location of the following instruction, such as when executing a jump, call, or return. When such dependencies have the potential to cause an erroneous computation by the pipeline, they are called *hazards*. Like dependencies, hazards can be classified as either *data hazards* or *control hazards*. We first concern ourselves with data hazards and then consider control hazards.

```
# prog1
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
```

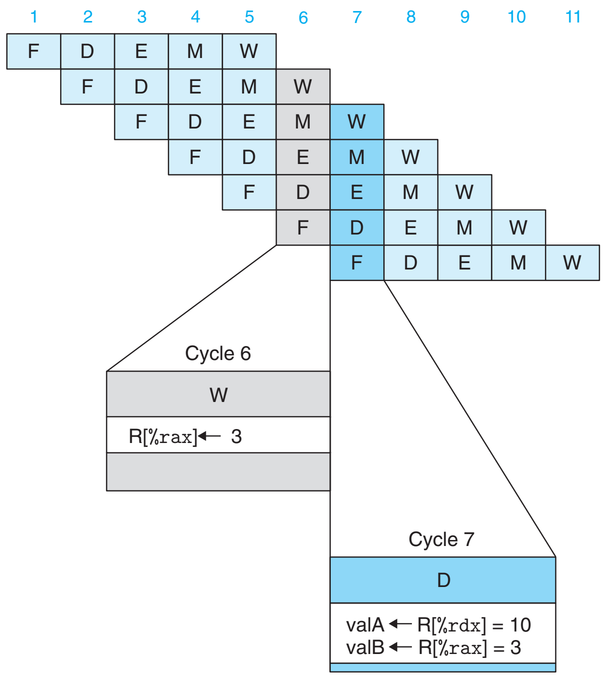


Figure 4.43 Pipelined execution of `prog1` without special pipeline control. In cycle 6, the second `irmovq` writes its result to program register `%rax`. The `addq` instruction reads its source operands in cycle 7, so it gets correct values for both `%rdx` and `%rax`.

Figure 4.43 illustrates the processing of a sequence of instructions we refer to as `prog1` by the PIPE− processor. Let us assume in this example and successive ones that the program registers initially all have value 0. The code loads values 10 and 3 into program registers `%rdx` and `%rax`, executes three `nop` instructions, and then adds register `%rdx` to `%rax`. We focus our attention on the potential data hazards resulting from the data dependencies between the two `irmovq` instructions and the `addq` instruction. On the right-hand side of the figure, we show a pipeline diagram for the instruction sequence. The pipeline stages for cycles 6 and 7 are shown highlighted in the pipeline diagram. Below this, we show an expanded view of the write-back activity in cycle 6 and the decode activity during cycle 7. After the start of cycle 7, both of the `irmovq` instructions have passed through the write-back stage, and so the register file holds the updated values of `%rdx` and `%rax`. As the `addq` instruction passes through the decode stage during cycle 7, it will therefore read the correct values for its source operands. The data dependencies between the two `irmovq` instructions and the `addq` instruction have not created data hazards in this example.

We saw that `prog1` will flow through our pipeline and get the correct results, because the three `nop` instructions create a delay between instructions with data

```
# prog2
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

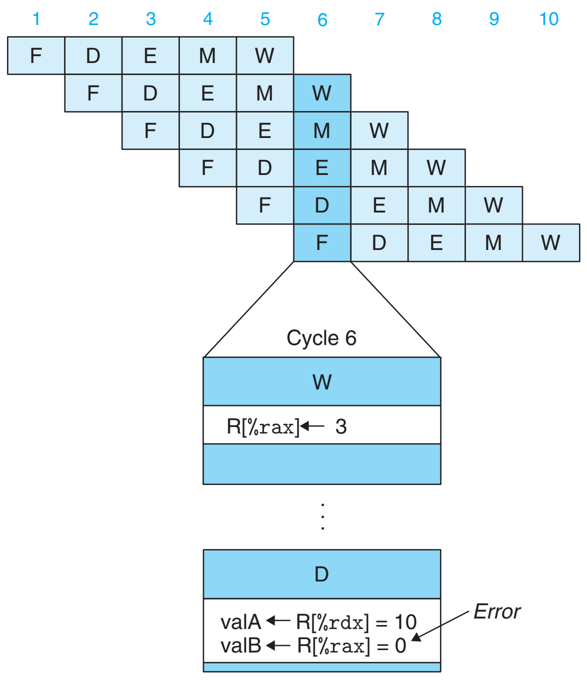
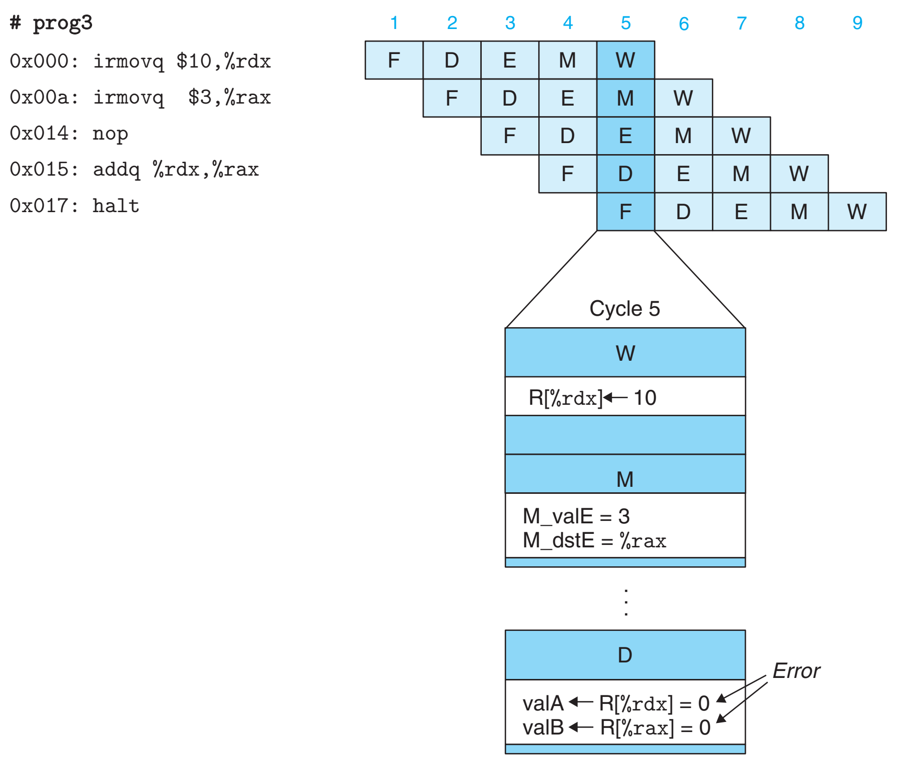


Figure 4.44 Pipelined execution of `prog2` without special pipeline control. The write to program register `%rax` does not occur until the start of cycle 7, and so the `addq` instruction gets the incorrect value for this register in the decode stage.

dependencies. Let us see what happens as these `nop` instructions are removed. Figure 4.44 illustrates the pipeline flow of a program, named `prog2`, containing two `nop` instructions between the two `irmovq` instructions generating values for registers `%rdx` and `%rax` and the `addq` instruction having these two registers as operands. In this case, the crucial step occurs in cycle 6, when the `addq` instruction reads its operands from the register file. An expanded view of the pipeline activities during this cycle is shown at the bottom of the figure. The first `irmovq` instruction has passed through the write-back stage, and so program register `%rdx` has been updated in the register file. The second `irmovq` instruction is in the write-back stage during this cycle, and so the write to program register `%rax` only occurs at the start of cycle 7 as the clock rises. As a result, the incorrect value zero would be read for register `%rax` (recall that we assume all registers are initially zero), since the pending write for this register has not yet occurred. Clearly, we will have to adapt our pipeline to handle this hazard properly.

Figure 4.45 shows what happens when we have only one `nop` instruction between the `irmovq` instructions and the `addq` instruction, yielding a program `prog3`. Now we must examine the behavior of the pipeline during cycle 5 as the `addq` instruction passes through the decode stage. Unfortunately, the pending



```
# prog3
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```

Figure 4.45 Pipelined execution of prog3 without special pipeline control. In cycle 5, the addq instruction reads its source operands from the register file. The pending write to register %rdx is still in the write-back stage, and the pending write to register %rax is still in the memory stage. Both operands valA and valB get incorrect values.

write to register %rdx is still in the write-back stage, and the pending write to %rax is still in the memory stage. Therefore, the addq instruction would get the incorrect values for both operands.

Figure 4.46 shows what happens when we remove all of the nop instructions between the irmovq instructions and the addq instruction, yielding a program prog4. Now we must examine the behavior of the pipeline during cycle 4 as the addq instruction passes through the decode stage. Unfortunately, the pending write to register %rdx is still in the memory stage, and the new value for %rax is just being computed in the execute stage. Therefore, the addq instruction would get the incorrect values for both operands.

These examples illustrate that a data hazard can arise for an instruction when one of its operands is updated by any of the three preceding instructions. These hazards occur because our pipelined processor reads the operands for an instruction from the register file in the decode stage but does not write the results for the instruction to the register file until three cycles later, after the instruction passes through the write-back stage.

```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

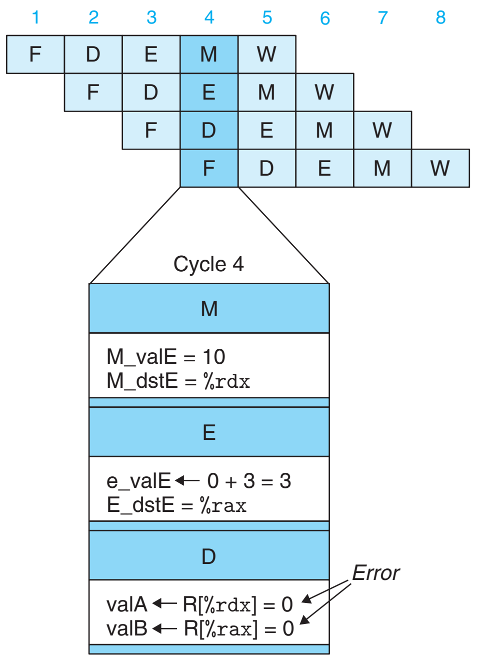


Figure 4.46 Pipelined execution of `prog4` without special pipeline control. In cycle 4, the `addq` instruction reads its source operands from the register file. The pending write to register `%rdx` is still in the memory stage, and the new value for register `%rax` is just being computed in the execute stage. Both operands valA and valB get incorrect values.

Avoiding Data Hazards by Stalling

One very general technique for avoiding hazards involves *stalling*, where the processor holds back one or more instructions in the pipeline until the hazard condition no longer holds. Our processor can avoid data hazards by holding back an instruction in the decode stage until the instructions generating its source operands have passed through the write-back stage. The details of this mechanism will be discussed in Section 4.5.8. It involves simple enhancements to the pipeline control logic. The effect of stalling is diagrammed in Figure 4.47 (`prog2`) and Figure 4.48 (`prog4`). (We omit `prog3` from this discussion, since it operates similarly to the other two examples.) When the `addq` instruction is in the decode stage, the pipeline control logic detects that at least one of the instructions in the execute, memory, or write-back stage will update either register `%rdx` or register `%rax`. Rather than letting the `addq` instruction pass through the stage with the incorrect results, it stalls the instruction, holding it back in the decode stage for either one (for `prog2`) or three (for `prog4`) extra cycles. For all three programs, the `addq` instruction finally gets correct values for its two source operands in cycle 7 and then proceeds down the pipeline.

```
# prog2
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addlq %rdx,%rax
0x018: halt
```

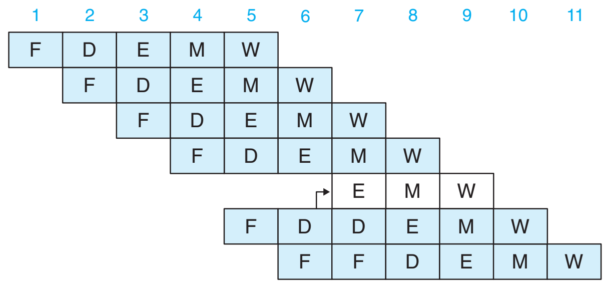


Figure 4.47 **Pipelined execution of** `prog2` **using stalls.** After decoding the `addq` instruction in cycle 6, the stall control logic detects a data hazard due to the pending write to register `%rax` in the write-back stage. It injects a bubble into the execute stage and repeats the decoding of the `addq` instruction in cycle 7. In effect, the machine has dynamically inserted a `nop` instruction, giving a flow similar to that shown for `prog1` (Figure 4.43).

```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
       bubble
       bubble
       bubble
0x014: addq %rdx,%rax
0x016: halt
```

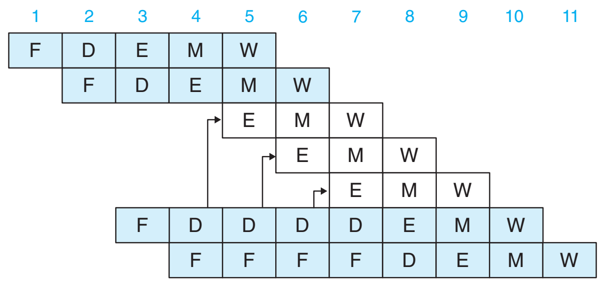


Figure 4.48 **Pipelined execution of** `prog4` **using stalls.** After decoding the `addq` instruction in cycle 4, the stall control logic detects data hazards for both source registers. It injects a bubble into the execute stage and repeats the decoding of the `addq` instruction on cycle 5. It again detects hazards for both source registers, injects a bubble into the execute stage, and repeats the decoding of the `addq` instruction on cycle 6. Still, it detects a hazard for source register `%rax`, injects a bubble into the execute stage, and repeats the decoding of the `addq` instruction on cycle 7. In effect, the machine has dynamically inserted three `nop` instructions, giving a flow similar to that shown for `prog1` (Figure 4.43).

In holding back the `addq` instruction in the decode stage, we must also hold back the `halt` instruction following it in the fetch stage. We can do this by keeping the program counter at a fixed value, so that the `halt` instruction will be fetched repeatedly until the stall has completed.

Stalling involves holding back one group of instructions in their stages while allowing other instructions to continue flowing through the pipeline. What then should we do in the stages that would normally be processing the `addq` instruction? We handle these by injecting a *bubble* into the execute stage each time we hold an instruction back in the decode stage. A bubble is like a dynamically generated `nop` instruction—it does not cause any changes to the registers, the memory, the

Aside Enumerating classes of data hazards

Hazards can potentially occur when one instruction updates part of the program state that will be read by a later instruction. For Y86-64, the program state includes the program registers, the program counter, the memory, the condition code register, and the status register. Let us look at the hazard possibilities in our proposed design for each of these forms of state.

Program registers. These are the hazards we have already identified. They arise because the register file is read in one stage and written in another, leading to possible unintended interactions between different instructions.

Program counter. Conflicts between updating and reading the program counter give rise to control hazards. No hazard arises when our fetch-stage logic correctly predicts the new value of the program counter before fetching the next instruction. Mispredicted branches and `ret` instructions require special handling, as will be discussed in Section 4.5.5.

Memory. Writes and reads of the data memory both occur in the memory stage. By the time an instruction reading memory reaches this stage, any preceding instructions writing memory will have already done so. On the other hand, there can be interference between instructions writing data in the memory stage and the reading of instructions in the fetch stage, since the instruction and data memories reference a single address space. This can only happen with programs containing *self-modifying code*, where instructions write to a portion of memory from which instructions are later fetched. Some systems have complex mechanisms to detect and avoid such hazards, while others simply mandate that programs should not use self-modifying code. We will assume for simplicity that programs do not modify themselves, and therefore we do not need to take special measures to update the instruction memory based on updates to the data memory during program execution.

Condition code register. These are written by integer operations in the execute stage. They are read by conditional moves in the execute stage and by conditional jumps in the memory stage. By the time a conditional move or jump reaches the execute stage, any preceding integer operation will have already completed this stage. No hazards can arise.

Status register. The program status can be affected by instructions as they flow through the pipeline. Our mechanism of associating a status code with each instruction in the pipeline enables the processor to come to an orderly halt when an exception occurs, as will be discussed in Section 4.5.6.

This analysis shows that we only need to deal with register data hazards, control hazards, and making sure exceptions are handled properly. A systematic analysis of this form is important when designing a complex system. It can identify the potential difficulties in implementing the system, and it can guide the generation of test programs to be used in checking the correctness of the system.

condition codes, or the program status. These are shown as white boxes in the pipeline diagrams of Figures 4.47 and 4.48. In these figures the arrow between the box labeled "D" for the `addq` instruction and the box labeled "E" for one of the pipeline bubbles indicates that a bubble was injected into the execute stage in place of the `addq` instruction that would normally have passed from the decode to the execute stage. We will look at the detailed mechanisms for making the pipeline stall and for injecting bubbles in Section 4.5.8.

In using stalling to handle data hazards, we effectively execute programs `prog2` and `prog4` by dynamically generating the pipeline flow seen for `prog1` (Figure 4.43). Injecting one bubble for `prog2` and three for `prog4` has the same effect as having three `nop` instructions between the second `irmovq` instruction and the `addq` instruction. This mechanism can be implemented fairly easily (see Problem 4.53), but the resulting performance is not very good. There are numerous cases in which one instruction updates a register and a closely following instruction uses the same register. This will cause the pipeline to stall for up to three cycles, reducing the overall throughput significantly.

Avoiding Data Hazards by Forwarding

Our design for PIPE− reads source operands from the register file in the decode stage, but there can also be a pending write to one of these source registers in the write-back stage. Rather than stalling until the write has completed, it can simply pass the value that is about to be written to pipeline register E as the source operand. Figure 4.49 shows this strategy with an expanded view of the pipeline diagram for cycle 6 of `prog2`. The decode-stage logic detects that register

```
# prog2
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

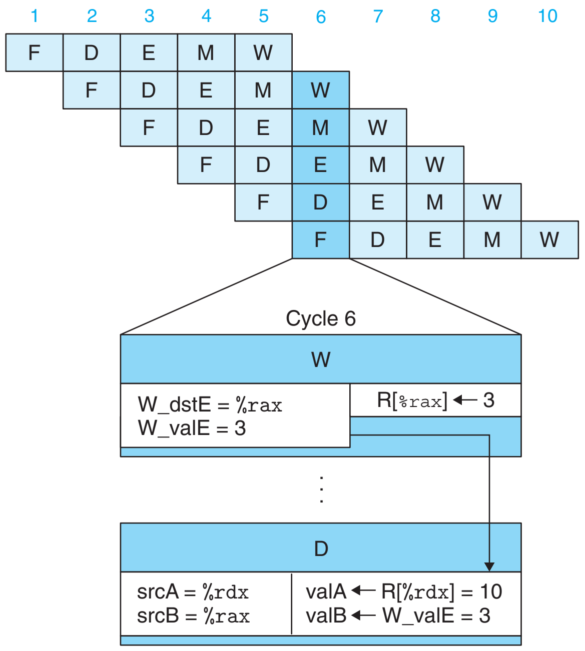


Figure 4.49 Pipelined execution of `prog2` using forwarding. In cycle 6, the decode-stage logic detects the presence of a pending write to register `%rax` in the write-back stage. It uses this value for source operand valB rather than the value read from the register file.

```
# prog3
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x005: addq %rdx,%rax
0x017: halt
```

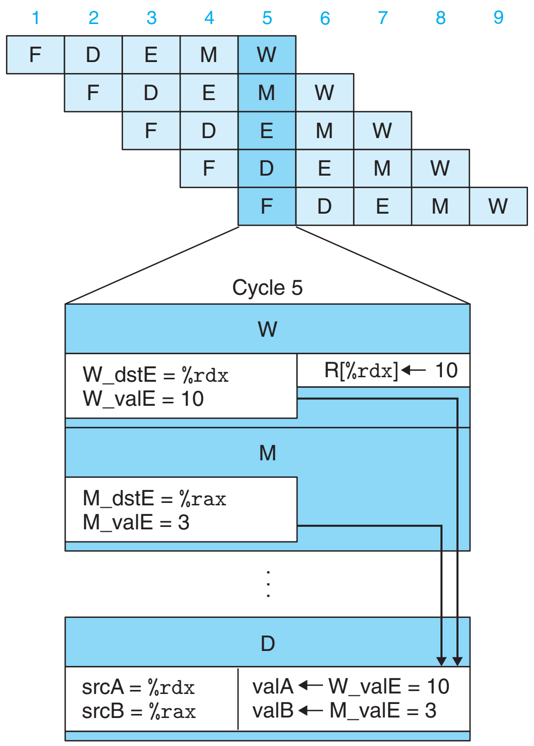


Figure 4.50 Pipelined execution of prog3 using forwarding. In cycle 5, the decode-stage logic detects a pending write to register %rdx in the write-back stage and to register %rax in the memory stage. It uses these as the values for valA and valB rather than the values read from the register file.

%rax is the source register for operand valB, and that there is also a pending write to %rax on write port E. It can therefore avoid stalling by simply using the data word supplied to port E (signal W_valE) as the value for operand valB. This technique of passing a result value directly from one pipeline stage to an earlier one is commonly known as *data forwarding* (or simply *forwarding*, and sometimes *bypassing*). It allows the instructions of prog2 to proceed through the pipeline without any stalling. Data forwarding requires adding additional data connections and control logic to the basic hardware structure.

As Figure 4.50 illustrates, data forwarding can also be used when there is a pending write to a register in the memory stage, avoiding the need to stall for program prog3. In cycle 5, the decode-stage logic detects a pending write to register %rdx on port E in the write-back stage, as well as a pending write to register %rax that is on its way to port E but is still in the memory stage. Rather than stalling until the writes have occurred, it can use the value in the write-back stage (signal W_valE) for operand valA and the value in the memory stage (signal M_valE) for operand valB.

```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

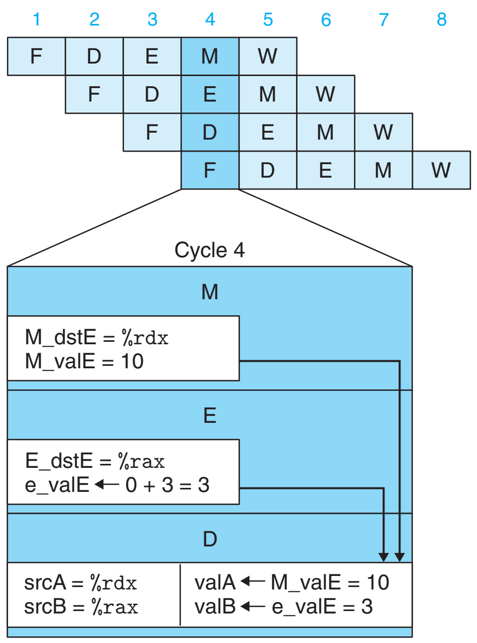


Figure 4.51 Pipelined execution of `prog4` using forwarding. In cycle 4, the decode-stage logic detects a pending write to register `%rdx` in the memory stage. It also detects that a new value is being computed for register `%rax` in the execute stage. It uses these as the values for valA and valB rather than the values read from the register file.

To exploit data forwarding to its full extent, we can also pass newly computed values from the execute stage to the decode stage, avoiding the need to stall for program `prog4`, as illustrated in Figure 4.51. In cycle 4, the decode-stage logic detects a pending write to register `%rdx` in the memory stage, and also that the value being computed by the ALU in the execute stage will later be written to register `%rax`. It can use the value in the memory stage (signal M_valE) for operand valA. It can also use the ALU output (signal e_valE) for operand valB. Note that using the ALU output does not introduce any timing problems. The decode stage only needs to generate signals valA and valB by the end of the clock cycle so that pipeline register E can be loaded with the results from the decode stage as the clock rises to start the next cycle. The ALU output will be valid before this point.

The uses of forwarding illustrated in programs `prog2` to `prog4` all involve the forwarding of values generated by the ALU and destined for write port E. Forwarding can also be used with values read from the memory and destined for write port M. From the memory stage, we can forward the value that has just been read from the data memory (signal m_valM). From the write-back stage, we can forward the pending write to port M (signal W_valM). This gives a total of five different forwarding sources (e_valE, m_valM, M_valE, W_valM, and W_valE) and two different forwarding destinations (valA and valB).

The expanded diagrams of Figures 4.49 to 4.51 also show how the decode-stage logic can determine whether to use a value from the register file or to use a forwarded value. Associated with every value that will be written back to the register file is the destination register ID. The logic can compare these IDs with the source register IDs srcA and srcB to detect a case for forwarding. It is possible to have multiple destination register IDs match one of the source IDs. We must establish a priority among the different forwarding sources to handle such cases. This will be discussed when we look at the detailed design of the forwarding logic.

Figure 4.52 shows the structure of PIPE, an extension of PIPE− that can handle data hazards by forwarding. Comparing this to the structure of PIPE− (Figure 4.41), we can see that the values from the five forwarding sources are fed back to the two blocks labeled “Sel+Fwd A” and “Fwd B” in the decode stage. The block labeled “Sel+Fwd A” combines the role of the block labeled “Select A” in PIPE− with the forwarding logic. It allows valA for pipeline register E to be either the incremented program counter valP, the value read from the A port of the register file, or one of the forwarded values. The block labeled “Fwd B” implements the forwarding logic for source operand valB.

Load/Use Data Hazards

One class of data hazards cannot be handled purely by forwarding, because memory reads occur late in the pipeline. Figure 4.53 illustrates an example of a *load/use hazard*, where one instruction (the `mrmovq` at address `0x028`) reads a value from memory for register `%rax` while the next instruction (the `addq` at address `0x032`) needs this value as a source operand. Expanded views of cycles 7 and 8 are shown in the lower part of the figure, where we assume all program registers initially have value 0. The `addq` instruction requires the value of the register in cycle 7, but it is not generated by the `mrmovq` instruction until cycle 8. In order to “forward” from the `mrmovq` to the `addq`, the forwarding logic would have to make the value go backward in time! Since this is clearly impossible, we must find some other mechanism for handling this form of data hazard. (The data hazard for register `%rbx`, with the value being generated by the `irmovq` instruction at address `0x01e` and used by the `addq` instruction at address `0x032`, can be handled by forwarding.)

As Figure 4.54 demonstrates, we can avoid a load/use data hazard with a combination of stalling and forwarding. This requires modifications of the control logic, but it can use existing bypass paths. As the `mrmovq` instruction passes through the execute stage, the pipeline control logic detects that the instruction in the decode stage (the `addq`) requires the result read from memory. It stalls the instruction in the decode stage for one cycle, causing a bubble to be injected into the execute stage. As the expanded view of cycle 8 shows, the value read from memory can then be forwarded from the memory stage to the `addq` instruction in the decode stage. The value for register `%rbx` is also forwarded from the write-back to the memory stage. As indicated in the pipeline diagram by the arrow from the box labeled “D” in cycle 7 to the box labeled “E” in cycle 8, the injected bubble replaces the `addq` instruction that would normally continue flowing through the pipeline.

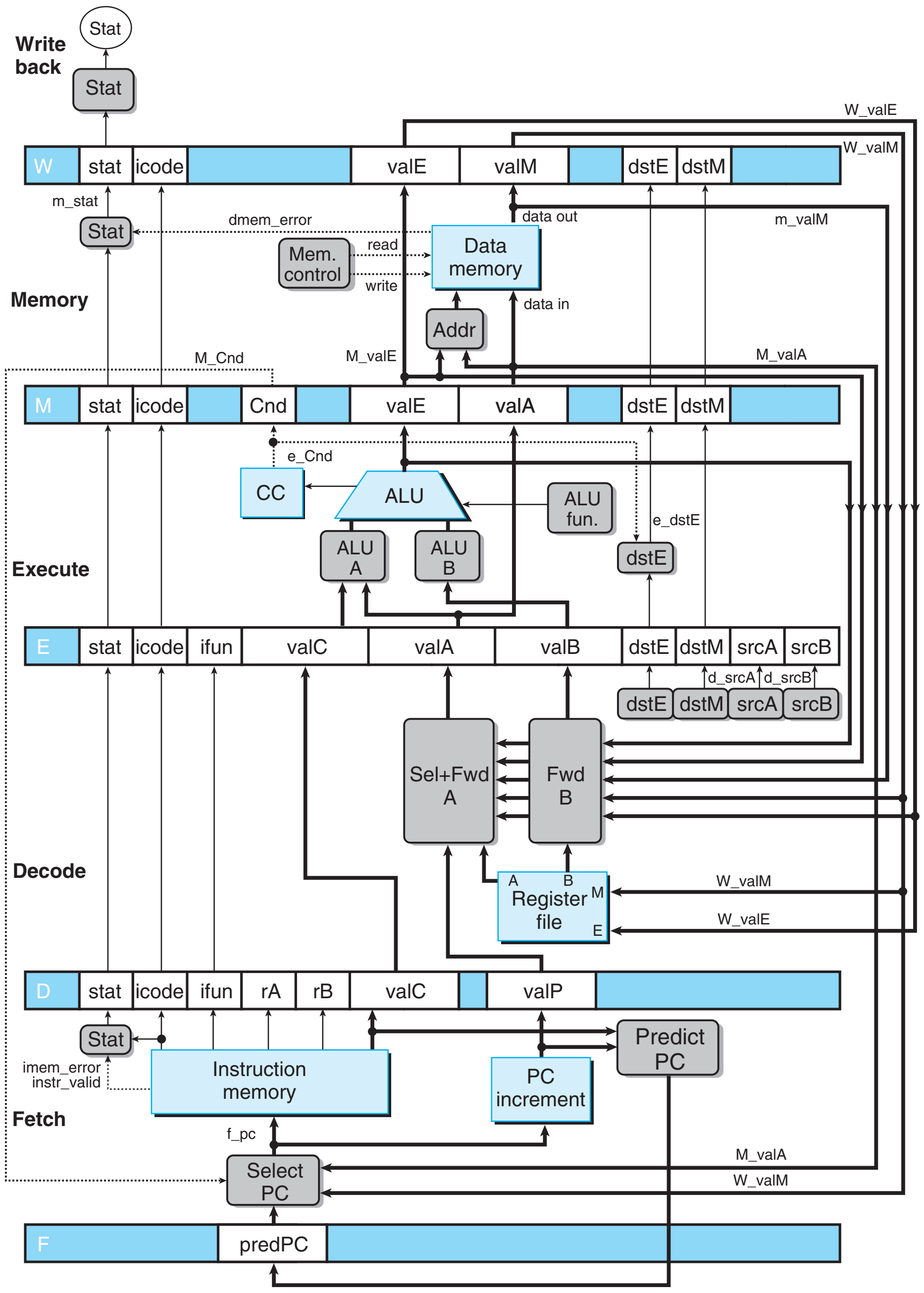
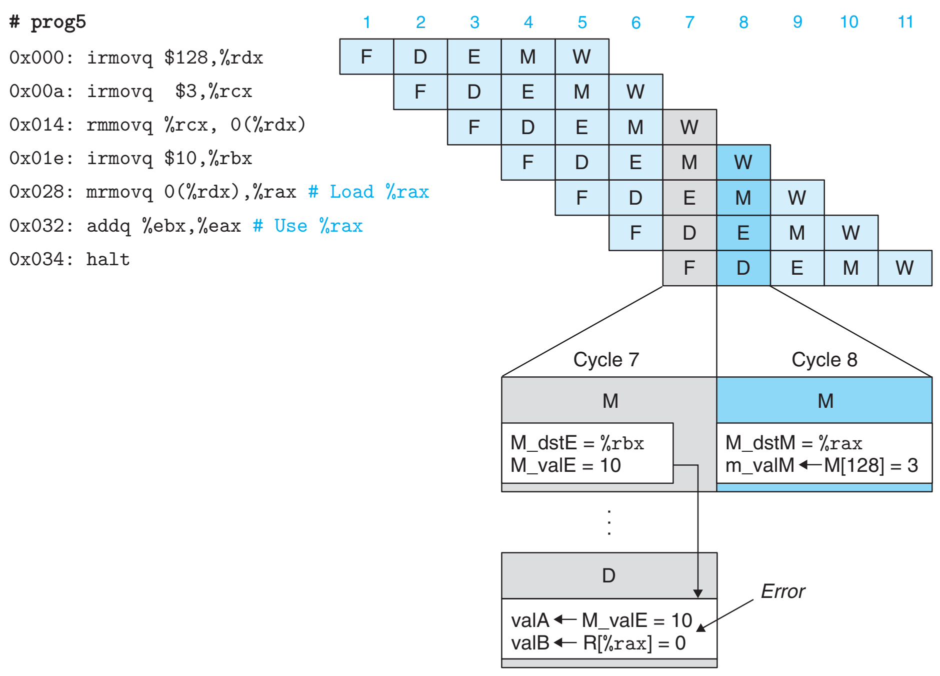


Figure 4.52 **Hardware structure of PIPE, our final pipelined implementation.** The additional bypassing paths enable forwarding the results from the three preceding instructions. This allows us to handle most forms of data hazards without stalling the pipeline.



```
# prog5
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %ebx,%eax # Use %rax
0x034: halt
```

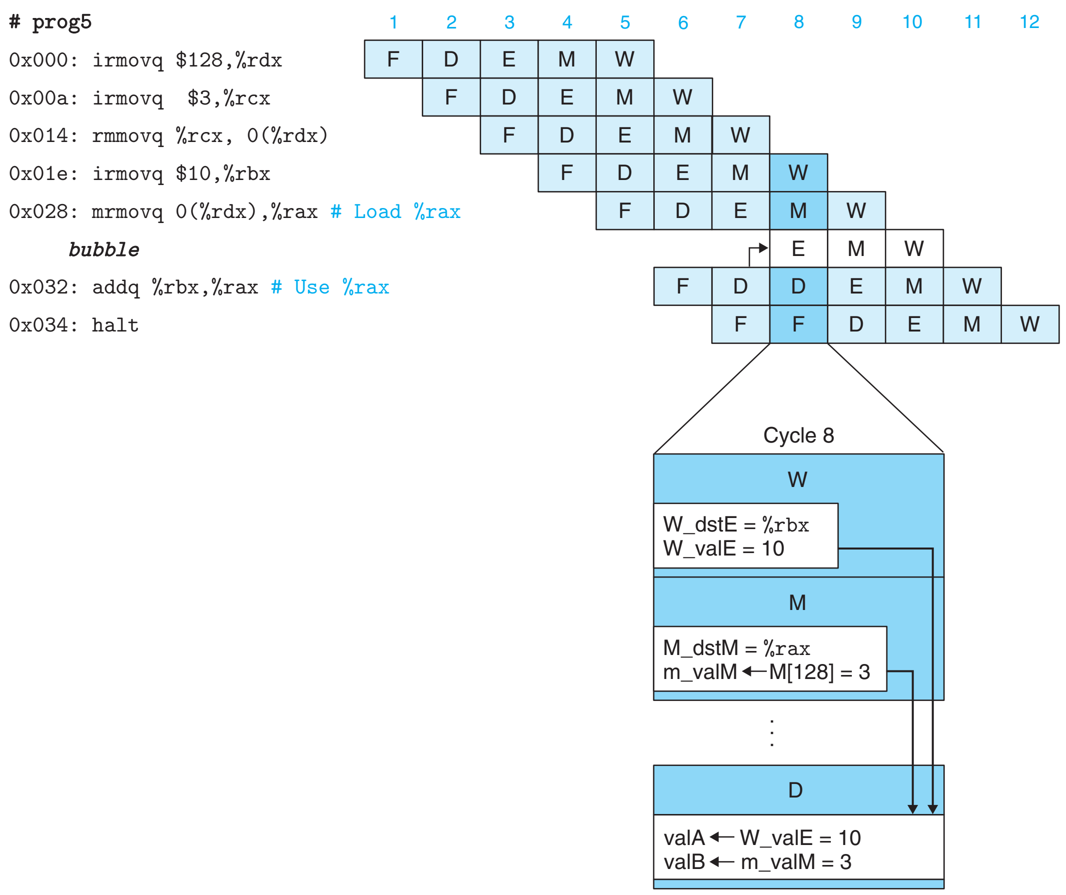
Figure 4.53 **Example of load/use data hazard.** The addq instruction requires the value of register %rax during the decode stage in cycle 7. The preceding mrmovq reads a new value for this register during the memory stage in cycle 8, which is too late for the addq instruction.

This use of a stall to handle a load/use hazard is called a *load interlock*. Load interlocks combined with forwarding suffice to handle all possible forms of data hazards. Since only load interlocks reduce the pipeline throughput, we can nearly achieve our throughput goal of issuing one new instruction on every clock cycle.

Avoiding Control Hazards

Control hazards arise when the processor cannot reliably determine the address of the next instruction based on the current instruction in the fetch stage. As was discussed in Section 4.5.4, control hazards can only occur in our pipelined processor for `ret` and jump instructions. Moreover, the latter case only causes difficulties when the direction of a conditional jump is mispredicted. In this section, we provide a high-level view of how these hazards can be handled. The detailed implementation will be presented in Section 4.5.8 as part of a more general discussion of the pipeline control.

For the `ret` instruction, consider the following example program. This program is shown in assembly code, but with the addresses of the different instructions on the left for reference:



```
# prog5
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
       bubble
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```

Figure 4.54 **Handling a load/use hazard by stalling.** By stalling the addq instruction for one cycle in the decode stage, the value for valB can be forwarded from the mrmovq instruction in the memory stage to the addq instruction in the decode stage.

```
0x000:    irmovq stack,%rsp  #   Initialize stack pointer
0x00a:    call proc          #   Procedure call
0x013:    irmovq $10,%rdx    #   Return point
0x01d:    halt
0x020: .pos 0x20
0x020: proc:                 # proc:
0x020:    ret                #   Return immediately
0x021:    rrmovq %rdx,%rbx   #   Not executed
0x030: .pos 0x30
0x030: stack:                # stack: Stack pointer
```

Figure 4.55 shows how we want the pipeline to process the ret instruction. As with our earlier pipeline diagrams, this figure shows the pipeline activity with

```
# prog7
0x000: irmovq Stack,%edx
0x00a: call proc
0x020: ret
       bubble
       bubble
       bubble
0x013: irmovq $10,%edx # Return point
```

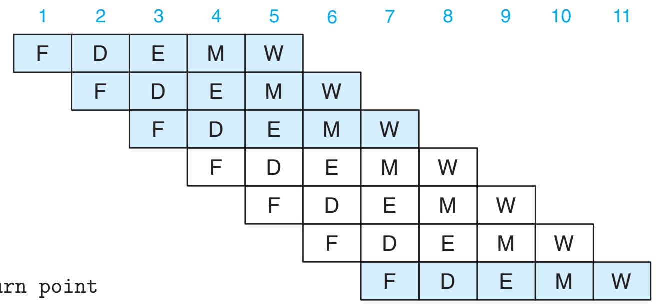


Figure 4.55 Simplified view of `ret` instruction processing. The pipeline should stall while the `ret` passes through the decode, execute, and memory stages, injecting three bubbles in the process. The PC selection logic will choose the return address as the instruction fetch address once the `ret` reaches the write-back stage (cycle 7).

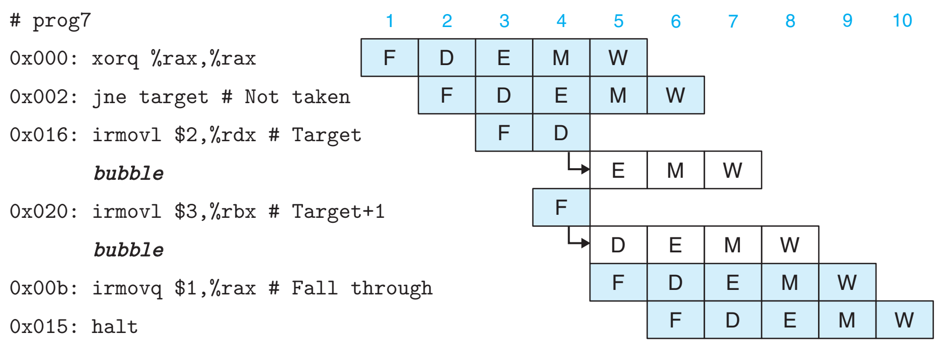
time growing to the right. Unlike before, the instructions are not listed in the same order they occur in the program, since this program involves a control flow where instructions are not executed in a linear sequence. It is useful to look at the instruction addresses to identify the different instructions in the program.

As this diagram shows, the `ret` instruction is fetched during cycle 3 and proceeds down the pipeline, reaching the write-back stage in cycle 7. While it passes through the decode, execute, and memory stages, the pipeline cannot do any useful activity. Instead, we want to inject three bubbles into the pipeline. Once the `ret` instruction reaches the write-back stage, the PC selection logic will set the program counter to the return address, and therefore the fetch stage will fetch the `irmovq` instruction at the return point (address `0x013`).

To handle a mispredicted branch, consider the following program, shown in assembly code but with the instruction addresses shown on the left for reference:

```
0x000:    xorq %rax,%rax
0x002:    jne  target        # Not taken
0x00b:    irmovq $1, %rax    # Fall through
0x015:    halt
0x016: target:
0x016:    irmovq $2, %rdx    # Target
0x020:    irmovq $3, %rbx    # Target+1
0x02a:    halt
```

Figure 4.56 shows how these instructions are processed. As before, the instructions are listed in the order they enter the pipeline, rather than the order they occur in the program. Since the jump instruction is predicted as being taken, the instruction at the jump target will be fetched in cycle 3, and the instruction following this one will be fetched in cycle 4. By the time the branch logic detects that the jump should not be taken during cycle 4, two instructions have been fetched that should not continue being executed. Fortunately, neither of these instructions has caused a change in the programmer-visible state. That can only occur when an instruction



```
# prog7
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovl $2,%rdx # Target
       bubble
0x020: irmovl $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```

Figure 4.56 Processing mispredicted branch instructions. The pipeline predicts branches will be taken and so starts fetching instructions at the jump target. Two instructions are fetched before the misprediction is detected in cycle 4 when the jump instruction flows through the execute stage. In cycle 5, the pipeline *cancels* the two target instructions by injecting bubbles into the decode and execute stages, and it also fetches the instruction following the jump.

reaches the execute stage, where it can cause the condition codes to change. At this point, the pipeline can simply *cancel* (sometimes called *instruction squashing*) the two misfetched instructions by injecting bubbles into the decode and execute stages on the following cycle while also fetching the instruction following the jump instruction. The two misfetched instructions will then simply disappear from the pipeline and therefore not have any effect on the programmer-visible state. The only drawback is that two clock cycles' worth of instruction processing capability have been wasted.

This discussion of control hazards indicates that they can be handled by careful consideration of the pipeline control logic. Techniques such as stalling and injecting bubbles into the pipeline dynamically adjust the pipeline flow when special conditions arise. As we will discuss in Section 4.5.8, a simple extension to the basic clocked register design will enable us to stall stages and to inject bubbles into pipeline registers as part of the pipeline control logic.

4.5.6 Exception Handling

As we will discuss in Chapter 8, a variety of activities in a processor can lead to *exceptional control flow*, where the normal chain of program execution gets broken. Exceptions can be generated either *internally*, by the executing program, or *externally*, by some outside signal. Our instruction set architecture includes three different internally generated exceptions, caused by (1) a `halt` instruction, (2) an instruction with an invalid combination of instruction and function code, and (3) an attempt to access an invalid address, either for instruction fetch or data read or write. A more complete processor design would also handle external exceptions, such as when the processor receives a signal that the network interface has received a new packet or the user has clicked a mouse button. Handling exceptions correctly is a challenging aspect of any microprocessor design. They can occur at unpredictable times, and they require creating a clean break in the flow of instructions through the processor pipeline. Our handling of the three internal exceptions gives just a glimpse of the true complexity of correctly detecting and handling exceptions.

Let us refer to the instruction causing the exception as the *excepting instruction*. In the case of an invalid instruction address, there is no actual excepting instruction, but it is useful to think of there being a sort of "virtual instruction" at the invalid address. In our simplified ISA model, we want the processor to halt when it reaches an exception and to set the appropriate status code, as listed in Figure 4.5. It should appear that all instructions up to the excepting instruction have completed, but none of the following instructions should have any effect on the programmer-visible state. In a more complete design, the processor would continue by invoking an exception handler, a procedure that is part of the operating system, but implementing this part of exception handling is beyond the scope of our presentation.

In a pipelined system, exception handling involves several subtleties. First, it is possible to have exceptions triggered by multiple instructions simultaneously. For example, during one cycle of pipeline operation, we could have a `halt` instruction in the fetch stage, and the data memory could report an out-of-bounds data address for the instruction in the memory stage. We must determine which of these exceptions the processor should report to the operating system. The basic rule is to put priority on the exception triggered by the instruction that is furthest along the pipeline. In the example above, this would be the out-of-bounds address attempted by the instruction in the memory stage. In terms of the machine-language program, the instruction in the memory stage should appear to execute before one in the fetch stage, and therefore only this exception should be reported to the operating system.

A second subtlety occurs when an instruction is first fetched and begins execution, causes an exception, and later is canceled due to a mispredicted branch. The following is an example of such a program in its object-code form:

```
0x000: 6300                 |    xorq %rax,%rax
0x002: 741600000000000000   |    jne  target      # Not taken
0x00b: 30f00100000000000000 |    irmovq $1, %rax  # Fall through
0x015: 00                   |    halt
0x016:                      | target:
0x016: ff                   |    .byte 0xFF       # Invalid instruction code
```

In this program, the pipeline will predict that the branch should be taken, and so it will fetch and attempt to use a byte with value `0xFF` as an instruction (generated in the assembly code using the `.byte` directive). The decode stage will therefore detect an invalid instruction exception. Later, the pipeline will discover that the branch should not be taken, and so the instruction at address `0x016` should never even have been fetched. The pipeline control logic will cancel this instruction, but we want to avoid raising an exception.

A third subtlety arises because a pipelined processor updates different parts of the system state in different stages. It is possible for an instruction following one causing an exception to alter some part of the state before the excepting instruction completes. For example, consider the following code sequence, in which we assume that user programs are not allowed to access addresses at the upper end of the 64-bit range:

```
    irmovq $1,%rax
    xorq %rsp,%rsp   # Set stack pointer to 0 and CC to 100
    pushq %rax       # Attempt to write to 0xfffffffffffffff8
    addq  %rax,%rax  # (Should not be executed) Would set CC to 000
```

The pushq instruction causes an address exception, because decrementing the stack pointer causes it to wrap around to 0xfffffffffffffff8. This exception is detected in the memory stage. On the same cycle, the addq instruction is in the execute stage, and it will cause the condition codes to be set to new values. This would violate our requirement that none of the instructions following the excepting instruction should have had any effect on the system state.

In general, we can both correctly choose among the different exceptions and avoid raising exceptions for instructions that are fetched due to mispredicted branches by merging the exception-handling logic into the pipeline structure. That is the motivation for us to include a status code stat in each of our pipeline registers (Figures 4.41 and 4.52). If an instruction generates an exception at some stage in its processing, the status field is set to indicate the nature of the exception. The exception status propagates through the pipeline with the rest of the information for that instruction, until it reaches the write-back stage. At this point, the pipeline control logic detects the occurrence of the exception and stops execution.

To avoid having any updating of the programmer-visible state by instructions beyond the excepting instruction, the pipeline control logic must disable any updating of the condition code register or the data memory when an instruction in the memory or write-back stages has caused an exception. In the example program above, the control logic will detect that the pushq in the memory stage has caused an exception, and therefore the updating of the condition code register by the addq instruction in the execute stage will be disabled.

Let us consider how this method of handling exceptions deals with the subtleties we have mentioned. When an exception occurs in one or more stages of a pipeline, the information is simply stored in the status fields of the pipeline registers. The event has no effect on the flow of instructions in the pipeline until an excepting instruction reaches the final pipeline stage, except to disable any updating of the programmer-visible state (the condition code register and the memory) by later instructions in the pipeline. Since instructions reach the write-back stage in the same order as they would be executed in a nonpipelined processor, we are guaranteed that the first instruction encountering an exception will arrive first in the write-back stage, at which point program execution can stop and the status code in pipeline register W can be recorded as the program status. If some instruction is fetched but later canceled, any exception status information about the instruction gets canceled as well. No instruction following one that causes an exception can alter the programmer-visible state. The simple rule of carrying the exception status together with all other information about an instruction through the pipeline provides a simple and reliable mechanism for handling exceptions.

4.5.7 PIPE Stage Implementations

We have now created an overall structure for PIPE, our pipelined Y86-64 processor with forwarding. It uses the same set of hardware units as the earlier sequential designs, with the addition of pipeline registers, some reconfigured logic blocks, and additional pipeline control logic. In this section, we go through the design of the different logic blocks, deferring the design of the pipeline control logic to the next section. Many of the logic blocks are identical to their counterparts in SEQ and SEQ+, except that we must choose proper versions of the different signals from the pipeline registers (written with the pipeline register name, written in uppercase, as a prefix) or from the stage computations (written with the first character of the stage name, written in lowercase, as a prefix).

As an example, compare the HCL code for the logic that generates the srcA signal in SEQ to the corresponding code in PIPE:

```
    # Code from SEQ

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

    # Code from PIPE

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

They differ only in the prefixes added to the PIPE signals: D_ for the source values, to indicate that the signals come from pipeline register D, and d_ for the result value, to indicate that it is generated in the decode stage. To avoid repetition, we will not show the HCL code here for blocks that only differ from those in SEQ because of the prefixes on names. As a reference, the complete HCL code for PIPE is given in Web Aside ARCH:HCL on page 508.

PC Selection and Fetch Stage

Figure 4.57 provides a detailed view of the PIPE fetch stage logic. As discussed earlier, this stage must also select a current value for the program counter and predict the next PC value. The hardware units for reading the instruction from

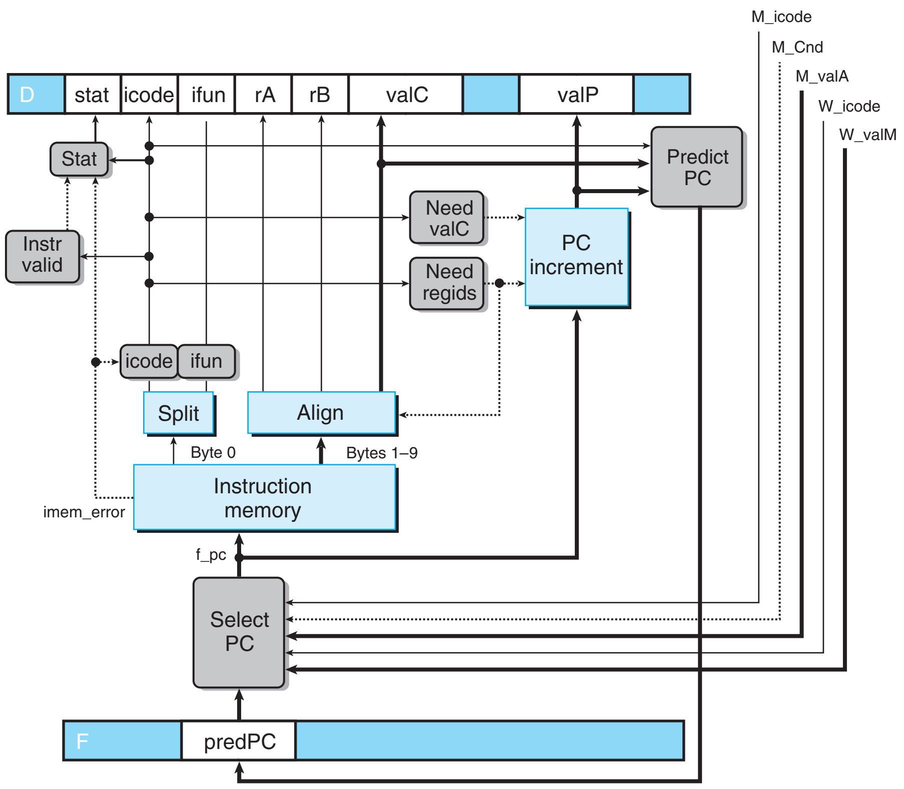


Figure 4.57 **PIPE PC selection and fetch logic.** Within the one cycle time limit, the processor can only predict the address of the next instruction.

memory and for extracting the different instruction fields are the same as those we considered for SEQ (see the fetch stage in Section 4.3.4).

The PC selection logic chooses between three program counter sources. As a mispredicted branch enters the memory stage, the value of valP for this instruction (indicating the address of the following instruction) is read from pipeline register M (signal M_valA). When a `ret` instruction enters the write-back stage, the return address is read from pipeline register W (signal W_valM). All other cases use the predicted value of the PC, stored in pipeline register F (signal F_predPC):

```
word f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        M_icode == IJXX && !M_Cnd : M_valA;
        # Completion of RET instruction
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
];
```

The PC prediction logic chooses valC for the fetched instruction when it is either a call or a jump, and valP otherwise:

```
word f_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        1 : f_valP;
];
```

The logic blocks labeled "Instr valid," "Need regids," and "Need valC" are the same as for SEQ, with appropriately named source signals.

Unlike in SEQ, we must split the computation of the instruction status into two parts. In the fetch stage, we can test for a memory error due to an out-of-range instruction address, and we can detect an illegal instruction or a `halt` instruction. Detecting an invalid data address must be deferred to the memory stage.

Practice Problem 4.30 **(solution page 526)**

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

Decode and Write-Back Stages

Figure 4.58 gives a detailed view of the decode and write-back logic for PIPE. The blocks labeled dstE, dstM, srcA, and srcB are very similar to their counterparts in the implementation of SEQ. Observe that the register IDs supplied to the write ports come from the write-back stage (signals W_dstE and W_dstM), rather than from the decode stage. This is because we want the writes to occur to the destination registers specified by the instruction in the write-back stage.

Practice Problem 4.31 **(solution page 526)**

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 dstE. (See the decode stage for SEQ in Section 4.3.4.) Do not concern yourself with the logic to implement conditional moves yet.

Most of the complexity of this stage is associated with the forwarding logic. As mentioned earlier, the block labeled "Sel+Fwd A" serves two roles. It merges the valP signal into the valA signal for later stages in order to reduce the amount of state in the pipeline register. It also implements the forwarding logic for source operand valA.

The merging of signals valA and valP exploits the fact that only the `call` and jump instructions need the value of valP in later stages, and these instructions

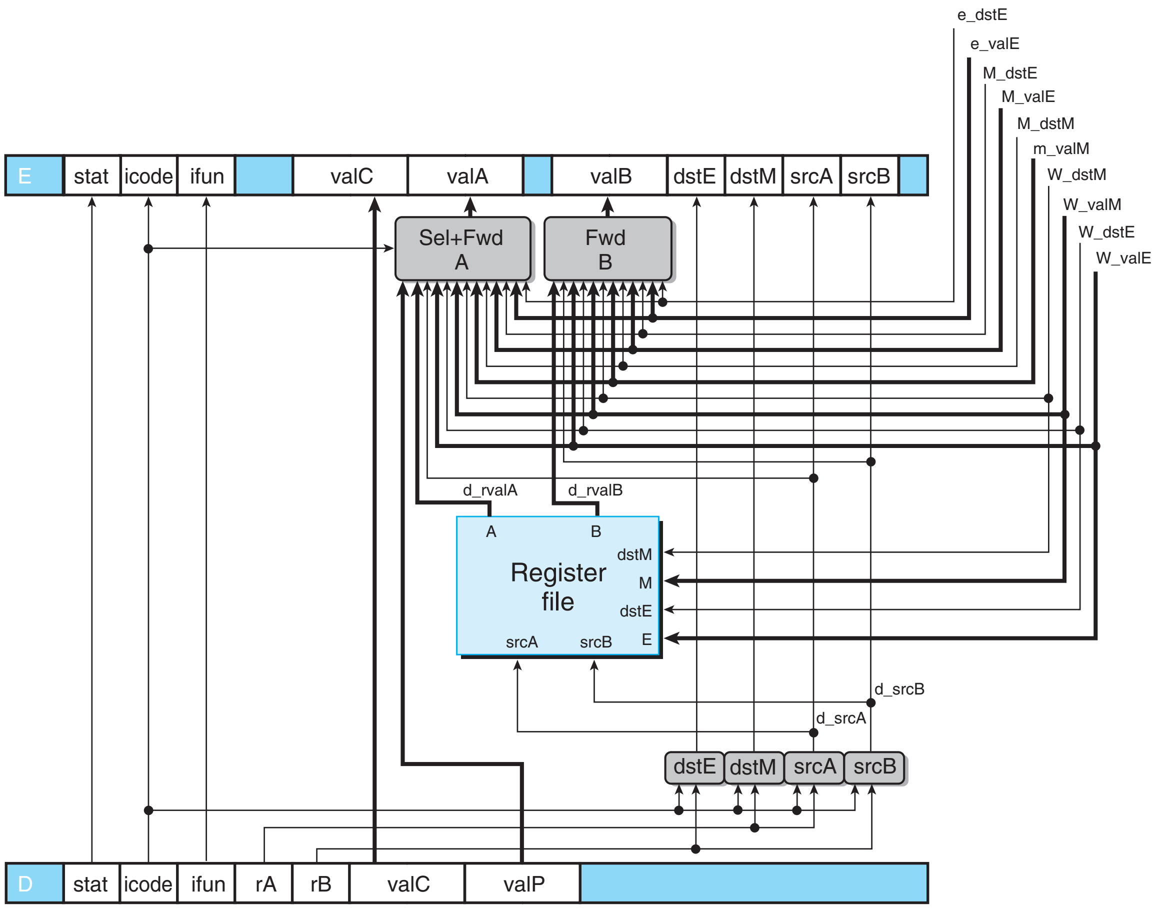


Figure 4.58 **PIPE decode and write-back stage logic.** No instruction requires both valP and the value read from register port A, and so these two can be merged to form the signal valA for later stages. The block labeled "Sel+Fwd A" performs this task and also implements the forwarding logic for source operand valA. The block labeled "Fwd B" implements the forwarding logic for source operand valB. The register write locations are specified by the dstE and dstM signals from the write-back stage rather than from the decode stage, since it is writing the results of the instruction currently in the write-back stage.

do not need the value read from the A port of the register file. This selection is controlled by the icode signal for this stage. When signal D_icode matches the instruction code for either `call` or `jXX`, this block should select D_valP as its output.

As mentioned in Section 4.5.5, there are five different forwarding sources, each with a data word and a destination register ID:

Data word	Register ID	Source description
e_valE	e_dstE	ALU output
m_valM	M_dstM	Memory output
M_valE	M_dstE	Pending write to port E in memory stage
W_valM	W_dstM	Pending write to port M in write-back stage
W_valE	W_dstE	Pending write to port E in write-back stage

If none of the forwarding conditions hold, the block should select d_rvalA, the value read from register port A, as its output.

Putting all of this together, we get the following HCL description for the new value of valA for pipeline register E:

```
word d_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == e_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : W_valM;    # Forward valM from write back
        d_srcA == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];
```

The priority given to the five forwarding sources in the above HCL code is very important. This priority is determined in the HCL code by the order in which the five destination register IDs are tested. If any order other than the one shown were chosen, the pipeline would behave incorrectly for some programs. Figure 4.59 shows an example of a program that requires a correct setting of priority among the forwarding sources in the execute and memory stages. In this program, the first two instructions write to register `%rdx`, while the third uses this register as its source operand. When the `rrmovq` instruction reaches the decode stage in cycle 4, the forwarding logic must choose between two values destined for its source register. Which one should it choose? To set the priority, we must consider the behavior of the machine-language program when it is executed one instruction at a time. The first `irmovq` instruction would set register `%rdx` to 10, the second would set the register to 3, and then the `rrmovq` instruction would read 3 from `%rdx`. To imitate this behavior, our pipelined implementation should always give priority to the forwarding source in the earliest pipeline stage, since it holds the latest instruction in the program sequence setting the register. Thus, the logic in the HCL code above first tests the forwarding source in the execute stage, then those in the memory stage, and finally the sources in the write-back stage. The forwarding priority between the two sources in either the memory or the write-back stages is only a concern for the instruction `popq %rsp`, since only this instruction can attempt two simultaneous writes to the same register.

```
# prog8
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rdx
0x014: rrmovq %rdx,%rax
0x016: halt
```

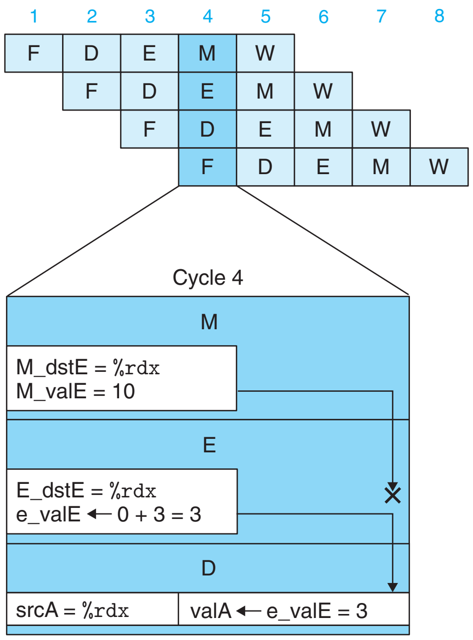


Figure 4.59 Demonstration of forwarding priority. In cycle 4, values for `%rdx` are available from both the execute and memory stages. The forwarding logic should choose the one in the execute stage, since it represents the most recently generated value for this register.

Practice Problem 4.32 (solution page 526)

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:

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

Practice Problem 4.33 (solution page 527)

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.

Practice Problem 4.34 (solution page 527)

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

One small part of the write-back stage remains. As shown in Figure 4.52, the overall processor status Stat is computed by a block based on the status value in pipeline register W. Recall from Section 4.1.1 that the code should indicate either normal operation (AOK) or one of the three exception conditions. Since pipeline register W holds the state of the most recently completed instruction, it is natural to use this value as an indication of the overall processor status. The only special case to consider is when there is a bubble in the write-back stage. This is part of normal operation, and so we want the status code to be AOK for this case as well:

```
word Stat = [
        W_stat == SBUB : SAOK;
        1 : W_stat;
];
```

Execute Stage

Figure 4.60 shows the execute stage logic for PIPE. The hardware units and the logic blocks are identical to those in SEQ, with an appropriate renaming of signals. We can see the signals e_valE and e_dstE directed toward the decode stage as one of the forwarding sources. One difference is that the logic labeled "Set CC," which determines whether or not to update the condition codes, has signals m_stat and W_stat as inputs. These signals are used to detect cases where an instruction

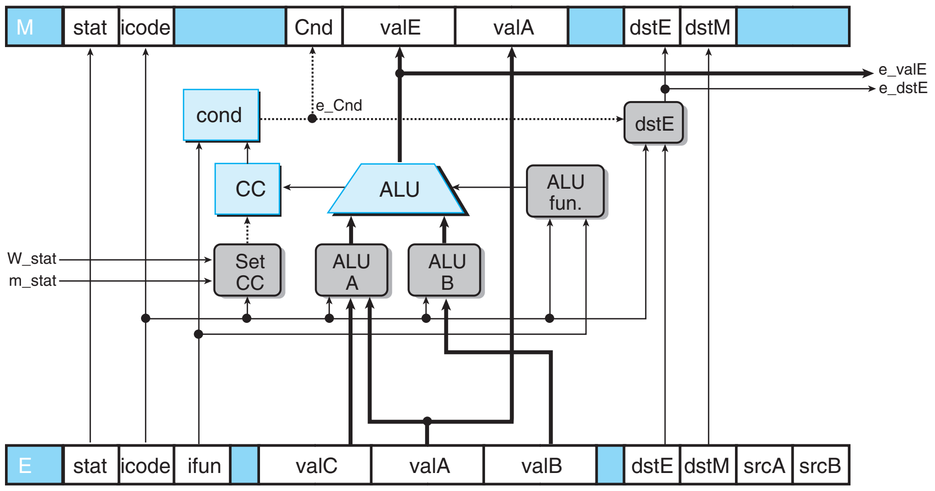


Figure 4.60 **PIPE execute stage logic.** This part of the design is very similar to the logic in the SEQ implementation.

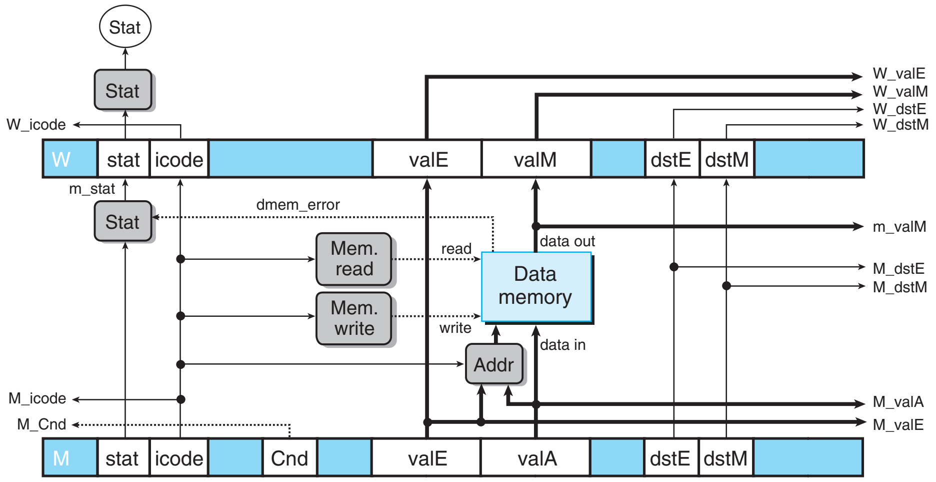


Figure 4.61 **PIPE memory stage logic.** Many of the signals from pipeline registers M and W are passed down to earlier stages to provide write-back results, instruction addresses, and forwarded results.

causing an exception is passing through later pipeline stages, and therefore any updating of the condition codes should be suppressed. This aspect of the design is discussed in Section 4.5.8.

Practice Problem 4.35 (solution page 527)

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.

Memory Stage

Figure 4.61 shows the memory stage logic for PIPE. Comparing this to the memory stage for SEQ (Figure 4.30), we see that, as noted before, the block labeled "Mem. data" in SEQ is not present in PIPE. This block served to select between data sources valP (for `call` instructions) and valA, but this selection is now performed by the block labeled "Sel+Fwd A" in the decode stage. Most other blocks in this stage are identical to their counterparts in SEQ, with an appropriate renaming of the signals. In this figure, you can also see that many of the values in pipeline registers and M and W are supplied to other parts of the circuit as part of the forwarding and pipeline control logic.

Practice Problem 4.36 **(solution page 528)**

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.

4.5.8 Pipeline Control Logic

We are now ready to complete our design for PIPE by creating the pipeline control logic. This logic must handle the following four control cases for which other mechanisms, such as data forwarding and branch prediction, do not suffice:

Load/use hazards. The pipeline must stall for one cycle between an instruction that reads a value from memory and an instruction that uses this value.

Processing `ret`. The pipeline must stall until the `ret` instruction reaches the write-back stage.

Mispredicted branches. By the time the branch logic detects that a jump should not have been taken, several instructions at the branch target will have started down the pipeline. These instructions must be canceled, and fetching should begin at the instruction following the jump instruction.

Exceptions. When an instruction causes an exception, we want to disable the updating of the programmer-visible state by later instructions and halt execution once the excepting instruction reaches the write-back stage.

We will go through the desired actions for each of these cases and then develop control logic to handle all of them.

Desired Handling of Special Control Cases

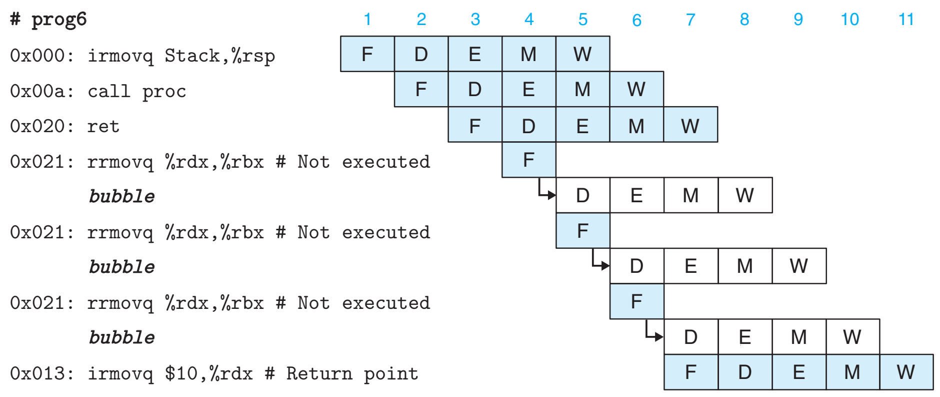
For a load/use hazard, we have described the desired pipeline operation in Section 4.5.5, as illustrated by the example of Figure 4.54. Only the `mrmovq` and `popq` instructions read data from memory. When (1) either of these is in the execute stage and (2) an instruction requiring the destination register is in the decode stage, we want to hold back the second instruction in the decode stage and inject a bubble into the execute stage on the next cycle. After this, the forwarding logic will resolve the data hazard. The pipeline can hold back an instruction in the decode stage by keeping pipeline register D in a fixed state. In doing so, it should also keep pipeline register F in a fixed state, so that the next instruction will be fetched a second time. In summary, implementing this pipeline flow requires detecting the hazard condition, keeping pipeline registers F and D fixed, and injecting a bubble into the execute stage.

For the processing of a `ret` instruction, we have described the desired pipeline operation in Section 4.5.5. The pipeline should stall for three cycles until the return address is read as the `ret` instruction passes through the memory stage.

This was illustrated by a simplified pipeline diagram in Figure 4.55 for processing the following program:

```
0x000:    irmovq stack,%rsp  #   Initialize stack pointer
0x00a:    call proc          #   Procedure call
0x013:    irmovq $10,%rdx    #   Return point
0x01d:    halt
0x020: .pos 0x20
0x020: proc:                 # proc:
0x020:    ret                #   Return immediately
0x021:    rrmovq %rdx,%rbx   #   Not executed
0x030: .pos 0x30
0x030: stack:                # stack: Stack pointer
```

Figure 4.62 provides a detailed view of the processing of the `ret` instruction for the example program. The key observation here is that there is no way to inject a bubble into the fetch stage of our pipeline. On every cycle, the fetch stage reads *some* instruction from the instruction memory. Looking at the HCL code for implementing the PC prediction logic in Section 4.5.7, we can see that for the `ret` instruction, the new value of the PC is predicted to be valP, the address of the following instruction. In our example program, this would be `0x021`, the address of the `rrmovq` instruction following the `ret`. This prediction is not correct for this example, nor would it be for most cases, but we are not attempting to predict return addresses correctly in our design. For three clock cycles, the fetch stage stalls, causing the `rrmovq` instruction to be fetched but then replaced by a bubble in the decode stage. This process is illustrated in Figure 4.62 by the three fetches, with an arrow leading down to the bubbles passing through the remaining pipeline stages. Finally, the `irmovq` instruction is fetched on cycle 7. Comparing Figure 4.62 with



```
# prog6
0x000: irmovq Stack,%rsp
0x00a: call proc
0x020: ret
0x021: rrmovq %rdx,%rbx # Not executed
       bubble
0x021: rrmovq %rdx,%rbx # Not executed
       bubble
0x021: rrmovq %rdx,%rbx # Not executed
       bubble
0x013: irmovq $10,%rdx # Return point
```

Figure 4.62 **Detailed processing of the `ret` instruction.** The fetch stage repeatedly fetches the `rrmovq` instruction following the `ret` instruction, but then the pipeline control logic injects a bubble into the decode stage rather than allowing the `rrmovq` instruction to proceed. The resulting behavior is equivalent to that shown in Figure 4.55.

Figure 4.55, we see that our implementation achieves the desired effect, but with a slightly peculiar fetching of an incorrect instruction for three consecutive cycles.

When a mispredicted branch occurs, we have described the desired pipeline operation in Section 4.5.5 and illustrated it in Figure 4.56. The misprediction will be detected as the jump instruction reaches the execute stage. The control logic then injects bubbles into the decode and execute stages on the next cycle, causing the two incorrectly fetched instructions to be canceled. On the same cycle, the pipeline reads the correct instruction into the fetch stage.

For an instruction that causes an exception, we must make the pipelined implementation match the desired ISA behavior, with all prior instructions completing and with none of the following instructions having any effect on the program state. Achieving these effects is complicated by the facts that (1) exceptions are detected during two different stages (fetch and memory) of program execution, and (2) the program state is updated in three different stages (execute, memory, and write-back).

Our stage designs include a status code stat in each pipeline register to track the status of each instruction as it passes through the pipeline stages. When an exception occurs, we record that information as part of the instruction's status and continue fetching, decoding, and executing instructions as if nothing were amiss. As the excepting instruction reaches the memory stage, we take steps to prevent later instructions from modifying the programmer-visible state by (1) disabling the setting of condition codes by instructions in the execute stage, (2) injecting bubbles into the memory stage to disable any writing to the data memory, and (3) stalling the write-back stage when it has an excepting instruction, thus bringing the pipeline to a halt.

The pipeline diagram in Figure 4.63 illustrates how our pipeline control handles the situation where an instruction causing an exception is followed by one that would change the condition codes. On cycle 6, the `pushq` instruction reaches the memory stage and generates a memory error. On the same cycle, the `addq` instruction in the execute stage generates new values for the condition codes. We disable the setting of condition codes when an excepting instruction is in the memory or write-back stage (by examining the signals m_stat and W_stat and then setting the signal set_cc to zero). We can also see the combination of injecting bubbles into the memory stage and stalling the excepting instruction in the write-back stage in the example of Figure 4.63—the `pushq` instruction remains stalled in the write-back stage, and none of the subsequent instructions get past the execute stage.

By this combination of pipelining the status signals, controlling the setting of condition codes, and controlling the pipeline stages, we achieve the desired behavior for exceptions: all instructions prior to the excepting instruction are completed, while none of the following instructions has any effect on the programmer-visible state.

Detecting Special Control Conditions

Figure 4.64 summarizes the conditions requiring special pipeline control. It gives expressions describing the conditions under which the three special cases arise.

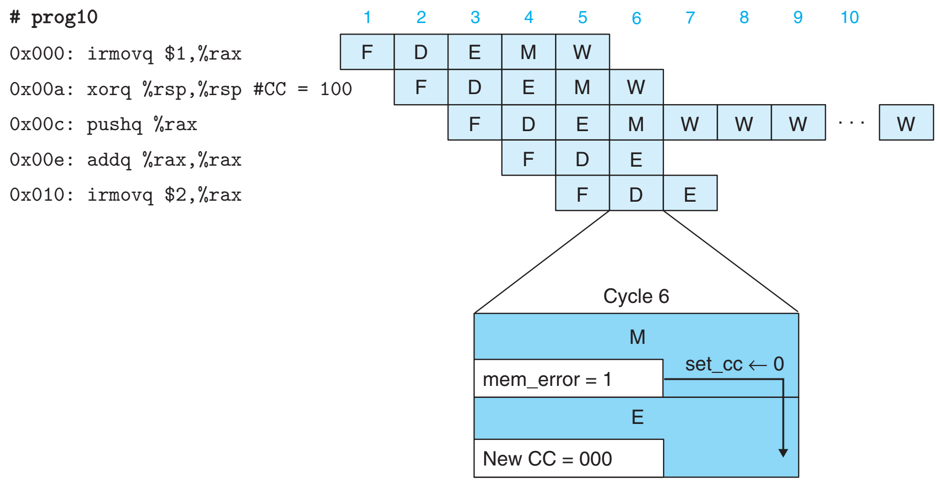


Figure 4.63 Processing invalid memory reference exception. On cycle 6, the invalid memory reference by the pushq instruction causes the updating of the condition codes to be disabled. The pipeline starts injecting bubbles into the memory stage and stalling the excepting instruction in the write-back stage.

Condition	Trigger
Processing `ret`	IRET ∈ {D_icode, E_icode, M_icode}
Load/use hazard	E_icode ∈ {IMRMOVQ, IPOPQ} && E_dstM ∈ {d_srcA, d_srcB}
Mispredicted branch	E_icode = IJXX && !e_Cnd
Exception	m_stat ∈ {SADR, SINS, SHLT} || W_stat ∈ {SADR, SINS, SHLT}

Figure 4.64 Detection conditions for pipeline control logic. Four different conditions require altering the pipeline flow by either stalling the pipeline or canceling partially executed instructions.

These expressions are implemented by simple blocks of combinational logic that must generate their results before the end of the clock cycle in order to control the action of the pipeline registers as the clock rises to start the next cycle. During a clock cycle, pipeline registers D, E, and M hold the states of the instructions that are in the decode, execute, and memory pipeline stages, respectively. As we approach the end of the clock cycle, signals d_srcA and d_srcB will be set to the register IDs of the source operands for the instruction in the decode stage. Detecting a `ret` instruction as it passes through the pipeline simply involves checking the instruction codes of the instructions in the decode, execute, and memory stages. Detecting a load/use hazard involves checking the instruction type (`mrmovq` or `popq`) of the instruction in the execute stage and comparing its destination register with the source registers of the instruction in the decode stage. The pipeline control logic should detect a mispredicted branch while the jump instruction is in the execute stage, so that it can set up the conditions required to recover from the misprediction as the instruction enters the memory stage. When a jump instruction is in the execute stage, the signal e_Cnd indicates whether or not the jump should be taken. We detect an excepting instruction by examining the instruction status values in the memory and write-back stages. For the memory stage, we use the signal m_stat, computed within the stage, rather than M_stat from the pipeline register. This internal signal incorporates the possibility of a data memory address error.

Pipeline Control Mechanisms

Figure 4.65 shows low-level mechanisms that allow the pipeline control logic to hold back an instruction in a pipeline register or to inject a bubble into the pipeline. These mechanisms involve small extensions to the basic clocked register described

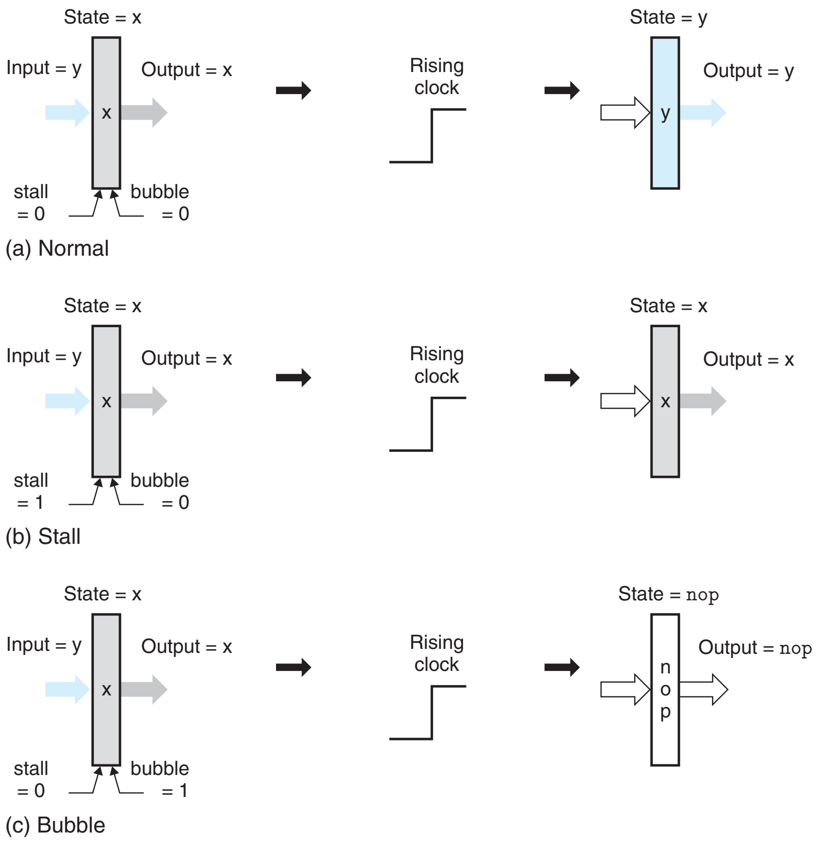


Figure 4.65 **Additional pipeline register operations.** (a) Under normal conditions, the state and output of the register are set to the value at the input when the clock rises. (b) When operated in *stall* mode, the state is held fixed at its previous value. (c) When operated in *bubble* mode, the state is overwritten with that of a nop operation.

	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

Figure 4.66 Actions for pipeline control logic. The different conditions require altering the pipeline flow by either stalling the pipeline or canceling partially executed instructions.

in Section 4.2.5. Suppose that each pipeline register has two control inputs stall and bubble. The settings of these signals determine how the pipeline register is updated as the clock rises. Under normal operation (Figure 4.65(a)), both of these inputs are set to 0, causing the register to load its input as its new state. When the stall signal is set to 1 (Figure 4.65(b)), the updating of the state is disabled. Instead, the register will remain in its previous state. This makes it possible to hold back an instruction in some pipeline stage. When the bubble signal is set to 1 (Figure 4.65(c)), the state of the register will be set to some fixed *reset configuration*, giving a state equivalent to that of a `nop` instruction. The particular pattern of ones and zeros for a pipeline register's reset configuration depends on the set of fields in the pipeline register. For example, to inject a bubble into pipeline register D, we want the icode field to be set to the constant value `INOP` (Figure 4.26). To inject a bubble into pipeline register E, we want the icode field to be set to `INOP` and the dstE, dstM, srcA, and srcB fields to be set to the constant `RNONE`. Determining the reset configuration is one of the tasks for the hardware designer in designing a pipeline register. We will not concern ourselves with the details here. We will consider it an error to set both the bubble and the stall signals to 1.

The table in Figure 4.66 shows the actions the different pipeline stages should take for each of the three special conditions. Each involves some combination of normal, stall, and bubble operations for the pipeline registers. In terms of timing, the stall and bubble control signals for the pipeline registers are generated by blocks of combinational logic. These values must be valid as the clock rises, causing each of the pipeline registers to either load, stall, or bubble as the next clock cycle begins. With this small extension to the pipeline register designs, we can implement a complete pipeline, including all of its control, using the basic building blocks of combinational logic, clocked registers, and random access memories.

Combinations of Control Conditions

In our discussion of the special pipeline control conditions so far, we assumed that at most one special case could arise during any single clock cycle. A common bug in designing a system is to fail to handle instances where multiple special conditions arise simultaneously. Let us analyze such possibilities. We need not worry about combinations involving program exceptions, since we have carefully designed our exception-handling mechanism to consider other instructions in the pipeline. Figure 4.67 diagrams the pipeline states that cause the other three special control

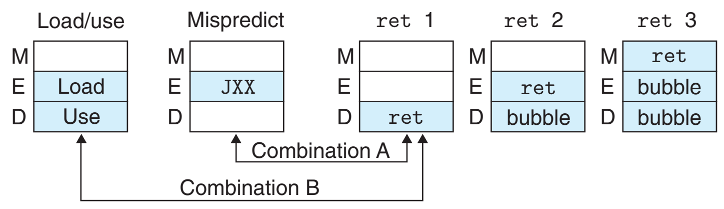


Figure 4.67 **Pipeline states for special control conditions.** The two pairs indicated can arise simultaneously.

conditions. These diagrams show blocks for the decode, execute, and memory stages. The shaded boxes represent particular constraints that must be satisfied for the condition to arise. A load/use hazard requires that the instruction in the execute stage reads a value from memory into a register, and that the instruction in the decode stage has this register as a source operand. A mispredicted branch requires the instruction in the execute stage to have a jump instruction. There are three possible cases for `ret`—the instruction can be in either the decode, execute, or memory stage. As the `ret` instruction moves through the pipeline, the earlier pipeline stages will have bubbles.

We can see by these diagrams that most of the control conditions are mutually exclusive. For example, it is not possible to have a load/use hazard and a mispredicted branch simultaneously, since one requires a load instruction (`mrmovq` or `popq`) in the execute stage, while the other requires a jump. Similarly, the second and third `ret` combinations cannot occur at the same time as a load/use hazard or a mispredicted branch. Only the two combinations indicated by arrows can arise simultaneously.

Combination A involves a not-taken jump instruction in the execute stage and a `ret` instruction in the decode stage. Setting up this combination requires the `ret` to be at the target of a not-taken branch. The pipeline control logic should detect that the branch was mispredicted and therefore cancel the `ret` instruction.

Practice Problem 4.37 (solution page 528)

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

Combining the control actions for the combination A conditions (Figure 4.66), we get the following pipeline control actions (assuming that either a bubble or a stall overrides the normal case):

	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

That is, it would be handled like a mispredicted branch, but with a stall in the fetch stage. Fortunately, on the next cycle, the PC selection logic will choose the address of the instruction following the jump, rather than the predicted program counter, and so it does not matter what happens with the pipeline register F. We conclude that the pipeline will correctly handle this combination.

Combination B involves a load/use hazard, where the loading instruction sets register `%rsp` and the `ret` instruction then uses this register as a source operand, since it must pop the return address from the stack. The pipeline control logic should hold back the `ret` instruction in the decode stage.

Practice Problem 4.38 (solution page 528)

Write a Y86-64 assembly-language program that causes combination B to arise and completes with a `halt` instruction if the pipeline operates correctly.

Combining the control actions for the combination B conditions (Figure 4.66), we get the following pipeline control actions:

	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Combination	stall	bubble+stall	bubble	normal	normal
Desired	stall	stall	bubble	normal	normal

If both sets of actions were triggered, the control logic would try to stall the `ret` instruction to avoid the load/use hazard but also inject a bubble into the decode stage due to the `ret` instruction. Clearly, we do not want the pipeline to perform both sets of actions. Instead, we want it to just take the actions for the load/use hazard. The actions for processing the `ret` instruction should be delayed for one cycle.

This analysis shows that combination B requires special handling. In fact, our original implementation of the PIPE control logic did not handle this combination correctly. Even though the design had passed many simulation tests, it had a subtle bug that was uncovered only by the analysis we have just shown. When a program having combination B was executed, the control logic would set both the bubble and the stall signals for pipeline register D to 1. This example shows the importance of systematic analysis. It would be unlikely to uncover this bug by just running normal programs. If left undetected, the pipeline would not faithfully implement the ISA behavior.

Control Logic Implementation

Figure 4.68 shows the overall structure of the pipeline control logic. Based on signals from the pipeline registers and pipeline stages, the control logic generates

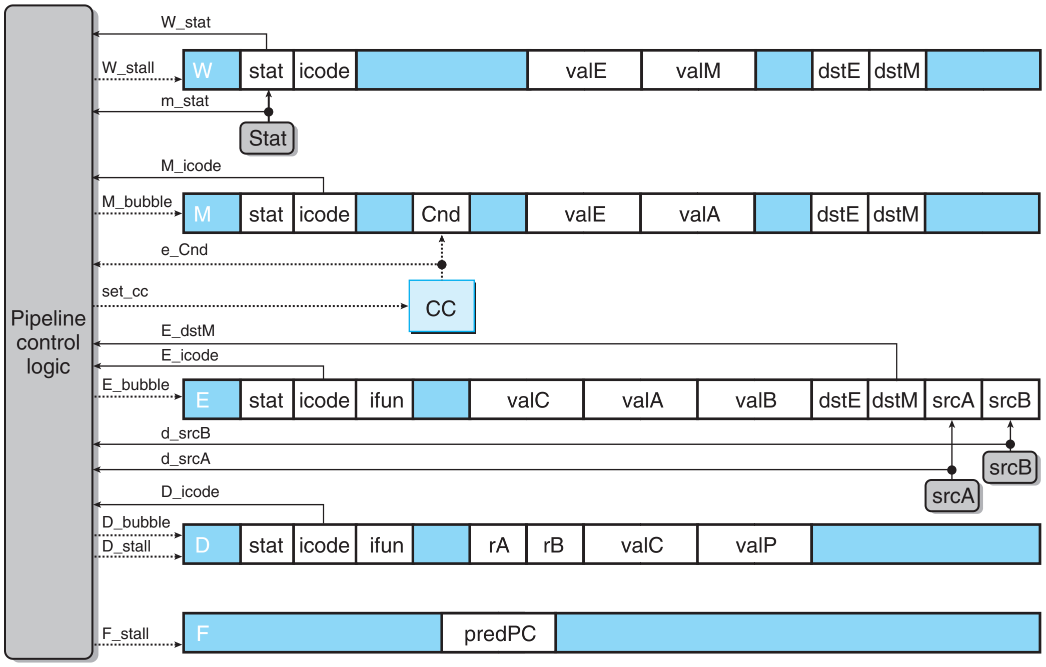


Figure 4.68 PIPE pipeline control logic. This logic overrides the normal flow of instructions through the pipeline to handle special conditions such as procedure returns, mispredicted branches, load/use hazards, and program exceptions.

stall and bubble control signals for the pipeline registers and also determines whether the condition code registers should be updated. We can combine the detection conditions of Figure 4.64 with the actions of Figure 4.66 to create HCL descriptions for the different pipeline control signals.

Pipeline register F must be stalled for either a load/use hazard or a `ret` instruction:

```
bool F_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB } ||
        # Stalling at fetch while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };
```

Practice Problem 4.39 (solution page 529)

Write HCL code for the signal D_stall in the PIPE implementation.

Pipeline register D must be set to bubble for a mispredicted branch or a `ret` instruction. As the analysis in the preceding section shows, however, it should not inject a bubble when there is a load/use hazard in combination with a `ret` instruction:

```
bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) &&
          IRET in { D_icode, E_icode, M_icode };
```

Practice Problem 4.40 (solution page 529)

Write HCL code for the signal E_bubble in the PIPE implementation.

Practice Problem 4.41 (solution page 529)

Write HCL code for the signal set_cc in the PIPE implementation. This should only occur for `OPq` instructions, and should consider the effects of program exceptions.

Practice Problem 4.42 (solution page 529)

Write HCL code for the signals M_bubble and W_stall in the PIPE implementation. The latter signal requires modifying the exception condition listed in Figure 4.64.

This covers all of the special pipeline control signal values. In the complete HCL code for PIPE, all other pipeline control signals are set to zero.

4.5.9 Performance Analysis

We can see that the conditions requiring special action by the pipeline control logic all cause our pipeline to fall short of the goal of issuing a new instruction on every clock cycle. We can measure this inefficiency by determining how often a bubble gets injected into the pipeline, since these cause unused pipeline cycles. A return instruction generates three bubbles, a load/use hazard generates one, and a mispredicted branch generates two. We can quantify the effect these penalties have on the overall performance by computing an estimate of the average number of clock cycles PIPE would require per instruction it executes, a measure known as the CPI (for "cycles per instruction"). This measure is the reciprocal of the average throughput of the pipeline, but with time measured in clock cycles rather than picoseconds. It is a useful measure of the architectural efficiency of a design.

If we ignore the performance implications of exceptions (which, by definition, will only occur rarely), another way to think about CPI is to imagine we run the

Aside Testing the design

As we have seen, there are many ways to introduce bugs into a design, even for a simple microprocessor. With pipelining, there are many subtle interactions between the instructions at different pipeline stages. We have seen that many of the design challenges involve unusual instructions (such as popping to the stack pointer) or unusual instruction combinations (such as a not-taken jump followed by a `ret`). We also see that exception handling adds an entirely new dimension to the possible pipeline behaviors. How, then, can we be sure that our design is correct? For hardware manufacturers, this is a dominant concern, since they cannot simply report an error and have users download code patches over the Internet. Even a simple logic design error can have serious consequences, especially as microprocessors are increasingly used to operate systems that are critical to our lives and health, such as automotive antilock braking systems, heart pacemakers, and aircraft control systems.

Simply simulating a design while running a number of "typical" programs is not a sufficient means of testing a system. Instead, thorough testing requires devising ways of systematically generating many tests that will exercise as many different instructions and instruction combinations as possible. In creating our Y86-64 processor designs, we also devised a number of testing scripts, each of which generates many different tests, runs simulations of the processor, and compares the resulting register and memory values to those produced by our YIS instruction set simulator. Here is a brief description of the scripts:

optest. Runs 49 tests of different Y86-64 instructions with different source and destination registers

jtest. Runs 64 tests of the different jump and call instructions, with different combinations of whether or not the branches are taken

cmtest. Runs 28 tests of the different conditional move instructions, with different control combinations

htest. Runs 600 tests of different data hazard possibilities, with different combinations of source and destination instructions, and with different numbers of `nop` instructions between the instruction pairs

ctest. Tests 22 different control combinations, based on an analysis similar to what we did in Section 4.5.8

etest. Tests 12 different combinations where an instruction causes an exception and the instructions following it could alter the programmer-visible state

The key idea of this testing method is that we want to be as systematic as possible, generating tests that create the different conditions that are likely to cause pipeline errors.

processor on some benchmark program and observe the operation of the execute stage. On each cycle, the execute stage either (1) processes an instruction and this instruction continues through the remaining stages to completion, or (2) processes a bubble injected due to one of the three special cases. If the stage processes a total of C_i instructions and C_b bubbles, then the processor has required around $C_i + C_b$ total clock cycles to execute C_i instructions. We say "around" because we ignore

Aside Formally verifying our design

Even when a design passes an extensive set of tests, we cannot be certain that it will operate correctly for all possible programs. The number of possible programs we could test is unimaginably large, even if we only consider tests consisting of short code segments. Newer methods of *formal verification*, however, hold the promise that we can have tools that rigorously consider all possible behaviors of a system and determine whether or not there are any design errors.

We were able to apply formal verification to an earlier version of our Y86-64 processors [13]. We set up a framework to compare the behavior of the pipelined design PIPE to the unpipelined version SEQ. That is, it was able to prove that for an arbitrary machine-language program, the two processors would have identical effects on the programmer-visible state. Of course, our verifier cannot actually run all possible programs, since there are an infinite number of them. Instead, it uses a form of proof by induction, showing a consistency between the two processors on a cycle-by-cycle basis. Carrying out this analysis requires reasoning about the hardware using *symbolic methods* in which we consider all program values to be arbitrary integers, and we abstract the ALU as a sort of "black box," computing some unspecified function over its arguments. We assume only that the ALUs for SEQ and PIPE compute identical functions.

We used the HCL descriptions of the control logic to generate the control logic for our symbolic processor models, and so we could catch any bugs in the HCL code. Being able to show that SEQ and PIPE are identical does not guarantee that either of them faithfully implements the instruction set architecture. However, it would uncover any bug due to an incorrect pipeline design, and this is the major source of design errors.

In our experiments, we verified not only a version of PIPE similar to the one we have presented in this chapter but also several variants that we give as homework problems, in which we add more instructions, modify the hardware capabilities, or use different branch prediction strategies. Interestingly, we found only one bug in all of our designs, involving control combination B (described in Section 4.5.8) for our solution to the variant described in Problem 4.58. This exposed a weakness in our testing regime that caused us to add additional cases to the ctest testing script.

Formal verification is still in an early stage of development. The tools are often difficult to use, and they do not have the capacity to verify large-scale designs. We were able to verify our processors in part because of their relative simplicity. Even then, it required several weeks of effort and multiple runs of the tools, each requiring up to 8 hours of computer time. This is an active area of research, with some tools becoming commercially available and some in use at companies such as Intel, AMD, and IBM.

the cycles required to start the instructions flowing through the pipeline. We can then compute the CPI for this benchmark as follows:

$$\text{CPI} = \frac{C_i + C_b}{C_i} = 1.0 + \frac{C_b}{C_i}$$

That is, the CPI equals 1.0 plus a penalty term C_b/C_i indicating the average number of bubbles injected per instruction executed. Since only three different instruction types can cause a bubble to be injected, we can break this penalty term into three components:

Web Aside ARCH:VLOG Verilog implementation of a pipelined Y86-64 processor

As we have mentioned, modern logic design involves writing textual representations of hardware designs in a *hardware description language*. The design can then be tested by both simulation and a variety of formal verification tools. Once we have confidence in the design, we can use *logic synthesis* tools to translate the design into actual logic circuits.

We have developed models of our Y86-64 processor designs in the Verilog hardware description language. These designs combine modules implementing the basic building blocks of the processor, along with control logic generated directly from the HCL descriptions. We have been able to synthesize some of these designs, download the logic circuit descriptions onto field-programmable gate array (FPGA) hardware, and run the processors on actual programs.

$$\text{CPI} = 1.0 + lp + mp + rp$$

where *lp* (for "load penalty") is the average frequency with which bubbles are injected while stalling for load/use hazards, *mp* (for "mispredicted branch penalty") is the average frequency with which bubbles are injected when canceling instructions due to mispredicted branches, and *rp* (for "return penalty") is the average frequency with which bubbles are injected while stalling for `ret` instructions. Each of these penalties indicates the total number of bubbles injected for the stated reason (some portion of C_b) divided by the total number of instructions that were executed (C_i.)

To estimate each of these penalties, we need to know how frequently the relevant instructions (load, conditional branch, and return) occur, and for each of these how frequently the particular condition arises. Let us pick the following set of frequencies for our CPI computation (these are comparable to measurements reported in [44] and [46]):

- Load instructions (`mrmovq` and `popq`) account for 25% of all instructions executed. Of these, 20% cause load/use hazards.
- Conditional branches account for 20% of all instructions executed. Of these, 60% are taken and 40% are not taken.
- Return instructions account for 2% of all instructions executed.

We can therefore estimate each of our penalties as the product of the frequency of the instruction type, the frequency the condition arises, and the number of bubbles that get injected when the condition occurs:

Cause	Name	Instruction frequency	Condition frequency	Bubbles	Product
Load/use	*lp*	0.25	0.20	1	0.05
Mispredict	*mp*	0.20	0.40	2	0.16
Return	*rp*	0.02	1.00	3	0.06
Total penalty					0.27

The sum of the three penalties is 0.27, giving a CPI of 1.27.

Our goal was to design a pipeline that can issue one instruction per cycle, giving a CPI of 1.0. We did not quite meet this goal, but the overall performance is still quite good. We can also see that any effort to reduce the CPI further should focus on mispredicted branches. They account for 0.16 of our total penalty of 0.27, because conditional branches are common, our prediction strategy often fails, and we cancel two instructions for every misprediction.

Practice Problem 4.43 (solution page 530)

Suppose we use a branch prediction strategy that achieves a success rate of 65%, such as backward taken, forward not taken (BTFNT), as described in Section 4.5.4. What would be the impact on CPI, assuming all of the other frequencies are not affected?

Practice Problem 4.44 (solution page 530)

Let us analyze the relative performance of using conditional data transfers versus conditional control transfers for the programs you wrote for Problems 4.5 and 4.6. Assume that we are using these programs to compute the sum of the absolute values of a very long array, and so the overall performance is determined largely by the number of cycles required by the inner loop. Assume that our jump instructions are predicted as being taken, and that around 50% of the array values are positive.

A. On average, how many instructions are executed in the inner loops of the two programs?

B. On average, how many bubbles would be injected into the inner loops of the two programs?

C. What is the average number of clock cycles required per array element for the two programs?

4.5.10 Unfinished Business

We have created a structure for the PIPE pipelined microprocessor, designed the control logic blocks, and implemented pipeline control logic to handle special cases where normal pipeline flow does not suffice. Still, PIPE lacks several key features that would be required in an actual microprocessor design. We highlight a few of these and discuss what would be required to add them.

Multicycle Instructions

All of the instructions in the Y86-64 instruction set involve simple operations such as adding numbers. These can be processed in a single clock cycle within the execute stage. In a more complete instruction set, we would also need to implement instructions requiring more complex operations such as integer multiplication and division and floating-point operations. In a medium-performance processor such as PIPE, typical execution times for these operations range from 3 or 4 cycles for floating-point addition up to 64 cycles for integer division. To implement these instructions, we require both additional hardware to perform the computations and a mechanism to coordinate the processing of these instructions with the rest of the pipeline.

One simple approach to implementing multicycle instructions is to simply expand the capabilities of the execute stage logic with integer and floating-point arithmetic units. An instruction remains in the execute stage for as many clock cycles as it requires, causing the fetch and decode stages to stall. This approach is simple to implement, but the resulting performance is not very good.

Better performance can be achieved by handling the more complex operations with special hardware functional units that operate independently of the main pipeline. Typically, there is one functional unit for performing integer multiplication and division, and another for performing floating-point operations. As an instruction enters the decode stage, it can be *issued* to the special unit. While the unit performs the operation, the pipeline continues processing other instructions. Typically, the floating-point unit is itself pipelined, and thus multiple operations can execute concurrently in the main pipeline and in the different units.

The operations of the different units must be synchronized to avoid incorrect behavior. For example, if there are data dependencies between the different operations being handled by different units, the control logic may need to stall one part of the system until the results from an operation handled by some other part of the system have been completed. Often, different forms of forwarding are used to convey results from one part of the system to other parts, just as we saw between the different stages of PIPE. The overall design becomes more complex than we have seen with PIPE, but the same techniques of stalling, forwarding, and pipeline control can be used to make the overall behavior match the sequential ISA model.

Interfacing with the Memory System

In our presentation of PIPE, we assumed that both the instruction fetch unit and the data memory could read or write any memory location in one clock cycle. We also ignored the possible hazards caused by self-modifying code where one instruction writes to the region of memory from which later instructions are fetched. Furthermore, we reference memory locations according to their virtual addresses, and these require a translation into physical addresses before the actual read or write operation can be performed. Clearly, it is unrealistic to do all of this processing in a single clock cycle. Even worse, the memory values being accessed may reside on disk, requiring millions of clock cycles to read into the processor memory.

As will be discussed in Chapters 6 and 9, the memory system of a processor uses a combination of multiple hardware memories and operating system software to manage the virtual memory system. The memory system is organized as a hierarchy, with faster but smaller memories holding a subset of the memory being backed up by slower and larger memories. At the level closest to the processor, the *cache* memories provide fast access to the most heavily referenced memory locations. A typical processor has two first-level caches—one for reading instructions and one for reading and writing data. Another type of cache memory, known as a *translation look-aside buffer*, or TLB, provides a fast translation from virtual to physical addresses. Using a combination of TLBs and caches, it is indeed possible to read instructions and read or write data in a single clock cycle most of the time. Thus, our simplified view of memory referencing by our processors is actually quite reasonable.

Although the caches hold the most heavily referenced memory locations, there will be times when a cache *miss* occurs, where some reference is made to a location that is not held in the cache. In the best case, the missing data can be retrieved from a higher-level cache or from the main memory of the processor, requiring 3 to 20 clock cycles. Meanwhile, the pipeline simply stalls, holding the instruction in the fetch or memory stage until the cache can perform the read or write operation. In terms of our pipeline design, this can be implemented by adding more stall conditions to the pipeline control logic. A cache miss and the consequent synchronization with the pipeline is handled completely by hardware, keeping the time required down to a small number of clock cycles.

In some cases, the memory location being referenced is actually stored in the disk or nonvolatile memory. When this occurs, the hardware signals a *page fault* exception. Like other exceptions, this will cause the processor to invoke the operating system's exception handler code. This code will then set up a transfer from the disk to the main memory. Once this completes, the operating system will return to the original program, where the instruction causing the page fault will be re-executed. This time, the memory reference will succeed, although it might cause a cache miss. Having the hardware invoke an operating system routine, which then returns control back to the hardware, allows the hardware and system software to cooperate in the handling of page faults. Since accessing a disk can require millions of clock cycles, the several thousand cycles of processing performed by the OS page fault handler has little impact on performance.

From the perspective of the processor, the combination of stalling to handle short-duration cache misses and exception handling to handle long-duration page faults takes care of any unpredictability in memory access times due to the structure of the memory hierarchy.

4.6 Summary

We have seen that the instruction set architecture, or ISA, provides a layer of abstraction between the behavior of a processor—in terms of the set of instructions and their encodings—and how the processor is implemented. The ISA provides a very sequential view of program execution, with one instruction executed to completion before the next one begins.

Aside State-of-the-art microprocessor design

A five-stage pipeline, such as we have shown with the PIPE processor, represented the state of the art in processor design in the mid-1980s. The prototype RISC processor developed by Patterson's research group at Berkeley formed the basis for the first SPARC processor, developed by Sun Microsystems in 1987. The processor developed by Hennessy's research group at Stanford was commercialized by MIPS Technologies (a company founded by Hennessy) in 1986. Both of these used five-stage pipelines. The Intel i486 processor also uses a five-stage pipeline, although with a different partitioning of responsibilities among the stages, with two decode stages and a combined execute/memory stage [27].

These pipelined designs are limited to a throughput of at most one instruction per clock cycle. The CPI (for "cycles per instruction") measure described in Section 4.5.9 can never be less than 1.0. The different stages can only process one instruction at a time. More recent processors support *superscalar* operation, meaning that they can achieve a CPI less than 1.0 by fetching, decoding, and executing multiple instructions in parallel. As superscalar processors have become widespread, the accepted performance measure has shifted from CPI to its reciprocal—the average number of instructions executed per cycle, or IPC. It can exceed 1.0 for superscalar processors. The most advanced designs use a technique known as *out-of-order* execution to execute multiple instructions in parallel, possibly in a totally different order than they occur in the program, while preserving the overall behavior implied by the sequential ISA model. This form of execution is described in Chapter 5 as part of our discussion of program optimization.

Pipelined processors are not just historical artifacts, however. The majority of processors sold are used in embedded systems, controlling automotive functions, consumer products, and other devices where the processor is not directly visible to the system user. In these applications, the simplicity of a pipelined processor, such as the one we have explored in this chapter, reduces its cost and power requirements compared to higher-performance models.

More recently, as multicore processors have gained a following, some have argued that we could get more overall computing power by integrating many simple processors on a single chip rather than a smaller number of more complex ones. This strategy is sometimes referred to as "many-core" processors [10].

We defined the Y86-64 instruction set by starting with the x86-64 instructions and simplifying the data types, address modes, and instruction encoding considerably. The resulting ISA has attributes of both RISC and CISC instruction sets. We then organized the processing required for the different instructions into a series of five stages, where the operations at each stage vary according to the instruction being executed. From this, we constructed the SEQ processor, in which an entire instruction is executed every clock cycle by having it flow through all five stages.

Pipelining improves the throughput performance of a system by letting the different stages operate concurrently. At any given time, multiple operations are being processed by the different stages. In introducing this concurrency, we must be careful to provide the same program-level behavior as would a sequential execution of the program. We introduced pipelining by reordering parts of SEQ to get SEQ+ and then adding pipeline registers to create the PIPE− pipeline.

Web Aside ARCH:HCL HCL descriptions of Y86-64 processors

In this chapter, we have looked at portions of the HCL code for several simple logic designs and for the control logic for Y86-64 processors SEQ and PIPE. For reference, we provide documentation of the HCL language and complete HCL descriptions for the control logic of the two processors. Each of these descriptions requires only five to seven pages of HCL code, and it is worthwhile to study them in their entirety.

We enhanced the pipeline performance by adding forwarding logic to speed the sending of a result from one instruction to another. Several special cases require additional pipeline control logic to stall or cancel some of the pipeline stages.

Our design included rudimentary mechanisms to handle exceptions, where we make sure that only instructions up to the excepting instruction affect the programmer-visible state. Implementing a complete handling of exceptions would be significantly more challenging. Properly handling exceptions gets even more complex in systems that employ greater degrees of pipelining and parallelism.

In this chapter, we have learned several important lessons about processor design:

- *Managing complexity is a top priority*. We want to make optimum use of the hardware resources to get maximum performance at minimum cost. We did this by creating a very simple and uniform framework for processing all of the different instruction types. With this framework, we could share the hardware units among the logic for processing the different instruction types.
- *We do not need to implement the ISA directly*. A direct implementation of the ISA would imply a very sequential design. To achieve higher performance, we want to exploit the ability in hardware to perform many operations simultaneously. This led to the use of a pipelined design. By careful design and analysis, we can handle the various pipeline hazards, so that the overall effect of running a program exactly matches what would be obtained with the ISA model.
- *Hardware designers must be meticulous.* Once a chip has been fabricated, it is nearly impossible to correct any errors. It is very important to get the design right on the first try. This means carefully analyzing different instruction types and combinations, even ones that do not seem to make sense, such as popping to the stack pointer. Designs must be thoroughly tested with systematic simulation test programs. In developing the control logic for PIPE, our design had a subtle bug that was uncovered only after a careful and systematic analysis of control combinations.

4.6.1 Y86-64 Simulators

The lab materials for this chapter include simulators for the SEQ and PIPE processors. Each simulator has two versions:

- The GUI (graphic user interface) version displays the memory, program code, and processor state in graphic windows. This provides a way to readily see how the instructions flow through the processors. The control panel also allows you to reset, single-step, or run the simulator interactively.
- The text version runs the same simulator, but it only displays information by printing to the terminal. This version is not as useful for debugging, but it allows automated testing of the processor.

The control logic for the simulators is generated by translating the HCL declarations of the logic blocks into C code. This code is then compiled and linked with the rest of the simulation code. This combination makes it possible for you to test out variants of the original designs using the simulators. Testing scripts are also available that thoroughly exercise the different instructions and the different hazard possibilities.

Bibliographic Notes

For those interested in learning more about logic design, the Katz and Borriello logic design textbook [58] is a standard introductory text, emphasizing the use of hardware description languages. Hennessy and Patterson's computer architecture textbook [46] provides extensive coverage of processor design, including both simple pipelines, such as the one we have presented here, and advanced processors that execute more instructions in parallel. Shriver and Smith [101] give a very thorough presentation of an Intel-compatible x86-64 processor manufactured by AMD.

Homework Problems

4.45 ◆

In Section 3.4.2, the x86-64 `pushq` instruction was described as decrementing the stack pointer and then storing the register at the stack pointer location. So, if we had an instruction of the form `pushq` *REG*, for some register *REG*, it would be equivalent to the code sequence

```
subq $8,%rsp          Decrement stack pointer
movq REG, (%rsp)      Store REG on stack
```

A. In light of analysis done in Practice Problem 4.7, does this code sequence correctly describe the behavior of the instruction `pushq %rsp`? Explain.

B. How could you rewrite the code sequence so that it correctly describes both the cases where *REG* is `%rsp` as well as any other register?

4.46 ◆

In Section 3.4.2, the x86-64 `popq` instruction was described as copying the result from the top of the stack to the destination register and then incrementing the stack pointer. So, if we had an instruction of the form `popq` *REG*, it would be equivalent to the code sequence

```
movq (%rsp), REG        Read REG from stack
addq $8,%rsp            Increment stack pointer
```

A. In light of analysis done in Practice Problem 4.8, does this code sequence correctly describe the behavior of the instruction `popq %rsp`? Explain.

B. How could you rewrite the code sequence so that it correctly describes both the cases where *REG* is `%rsp` as well as any other register?

4.47 ◆◆◆

Your assignment will be to write a Y86-64 program to perform bubblesort. For reference, the following C function implements bubblesort using array referencing:

```
1   /* Bubble sort: Array version */
2   void bubble_a(long *data, long count) {
3       long i, last;
4       for (last = count-1; last > 0; last--) {
5           for (i = 0; i < last; i++)
6               if (data[i+1] < data[i]) {
7                   /* Swap adjacent elements */
8                   long t = data[i+1];
9                   data[i+1] = data[i];
10                  data[i] = t;
11              }
12      }
13  }
```

A. Write and test a C version that references the array elements with pointers, rather than using array indexing.

B. Write and test a Y86-64 program consisting of the function and test code. You may find it useful to pattern your implementation after x86-64 code generated by compiling your C code. Although pointer comparisons are normally done using unsigned arithmetic, you can use signed arithmetic for this exercise.

4.48 ◆◆

Modify the code you wrote for Problem 4.47 to implement the test and swap in the bubblesort function (lines 6–11) using no jumps and at most three conditional moves.

4.49 ◆◆◆

Modify the code you wrote for Problem 4.47 to implement the test and swap in the bubblesort function (lines 6–11) using no jumps and just one conditional move.

4.50 ◆◆◆

In Section 3.6.8, we saw that a common way to implement `switch` statements is to create a set of code blocks and then index those blocks using a jump table. Consider

```
#include <stdio.h>
/* Example use of switch statement */

long switchv(long idx) {
    long result = 0;
    switch(idx) {
    case 0:
        result = 0xaaa;
        break;
    case 2:
    case 5:
        result = 0xbbb;
        break;
    case 3:
        result = 0xccc;
        break;
    default:
        result = 0xddd;

    }
    return result;
 }

/* Testing Code */
#define CNT 8
#define MINVAL -1

int main() {
    long vals[CNT];
    long i;
    for (i = 0; i < CNT; i++) {
        vals[i] = switchv(i + MINVAL);
        printf("idx = %ld, val = 0x%lx\n", i + MINVAL, vals[i]);
    }
    return 0;
}
```

Figure 4.69 **Switch statements can be translated into Y86-64 code.** This requires implementation of a jump table.

the C code shown in Figure 4.69 for a function `switchv`, along with associated test code.

Implement `switchv` in Y86-64 using a jump table. Although the Y86-64 instruction set does not include an indirect jump instruction, you can get the same effect by pushing a computed address onto the stack and then executing the `ret` instruction. Implement test code similar to what is shown in C to demonstrate that your implementation of `switchv` will handle both the cases handled explicitly as well as those that trigger the `default` case.

4.51 ◆

Practice Problem 4.3 introduced the `iaddq` instruction to add immediate data to a register. Describe the computations performed to implement this instruction. Use the computations for `irmovq` and `OPq` (Figure 4.18) as a guide.

4.52 ◆◆

The file `seq-full.hcl` contains the HCL description for SEQ, along with the declaration of a constant `IIADDQ` having hexadecimal value `C`, the instruction code for `iaddq`. Modify the HCL descriptions of the control logic blocks to implement the `iaddq` instruction, as described in Practice Problem 4.3 and Problem 4.51. See the lab material for directions on how to generate a simulator for your solution and how to test it.

4.53 ◆◆◆

Suppose we wanted to create a lower-cost pipelined processor based on the structure we devised for PIPE− (Figure 4.41), without any bypassing. This design would handle all data dependencies by stalling until the instruction generating a needed value has passed through the write-back stage.

The file `pipe-stall.hcl` contains a modified version of the HCL code for PIPE in which the bypassing logic has been disabled. That is, the signals e_valA and e_valB are simply declared as follows:

```
##  DO NOT MODIFY THE FOLLOWING CODE.
## No forwarding.  valA is either valP or value from register file
word d_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        1 : d_rvalA;  # Use value read from register file
];

## No forwarding.  valB is value from register file
word d_valB = d_rvalB;
```

Modify the pipeline control logic at the end of this file so that it correctly handles all possible control and data hazards. As part of your design effort, you should analyze the different combinations of control cases, as we did in the design of the pipeline control logic for PIPE. You will find that many different combinations can occur, since many more conditions require the pipeline to stall. Make sure your control logic handles each combination correctly. See the lab material for directions on how to generate a simulator for your solution and how to test it.

4.54 ◆◆

The file `pipe-full.hcl` contains a copy of the PIPE HCL description, along with a declaration of the constant value `IIADDQ`. Modify this file to implement the `iaddq` instruction, as described in Practice Problem 4.3 and Problem 4.51. See the lab material for directions on how to generate a simulator for your solution and how to test it.

4.55 ◆◆◆

The file `pipe-nt.hcl` contains a copy of the HCL code for PIPE, plus a declaration of the constant `J_YES` with value `0`, the function code for an unconditional jump instruction. Modify the branch prediction logic so that it predicts conditional jumps as being not taken while continuing to predict unconditional jumps and `call` as being taken. You will need to devise a way to get valC, the jump target address, to pipeline register M to recover from mispredicted branches. See the lab material for directions on how to generate a simulator for your solution and how to test it.

4.56 ◆◆◆

The file `pipe-btfnt.hcl` contains a copy of the HCL code for PIPE, plus a declaration of the constant `J_YES` with value `0`, the function code for an unconditional jump instruction. Modify the branch prediction logic so that it predicts conditional jumps as being taken when valC $<$ valP (backward branch) and as being not taken when valC $\geq$ valP (forward branch). (Since Y86-64 does not support unsigned arithmetic, you should implement this test using a signed comparison.) Continue to predict unconditional jumps and `call` as being taken. You will need to devise a way to get both valC and valP to pipeline register M to recover from mispredicted branches. See the lab material for directions on how to generate a simulator for your solution and how to test it.

4.57 ◆◆◆

In our design of PIPE, we generate a stall whenever one instruction performs a *load*, reading a value from memory into a register, and the next instruction has this register as a source operand. When the source gets used in the execute stage, this stalling is the only way to avoid a hazard. For cases where the second instruction stores the source operand to memory, such as with an `rmmovq` or `pushq` instruction, this stalling is not necessary. Consider the following code examples:

```
1         mrmovq 0(%rcx),%rdx    # Load  1
2         pushq  %rdx            # Store 1
3         nop
4         popq %rdx              # Load  2
5         rmmovq %rax,0(%rdx)    # Store 2
```

In lines 1 and 2, the `mrmovq` instruction reads a value from memory into `%rdx`, and the `pushq` instruction then pushes this value onto the stack. Our design for PIPE would stall the `pushq` instruction to avoid a load/use hazard. Observe, however, that the value of `%rdx` is not required by the `pushq` instruction until it reaches the memory stage. We can add an additional bypass path, as diagrammed in Figure 4.70, to forward the memory output (signal m_valM) to the valA field in pipeline register M. On the next clock cycle, this forwarded value can then be written to memory. This technique is known as *load forwarding*.

Note that the second example (lines 4 and 5) in the code sequence above cannot make use of load forwarding. The value loaded by the `popq` instruction is

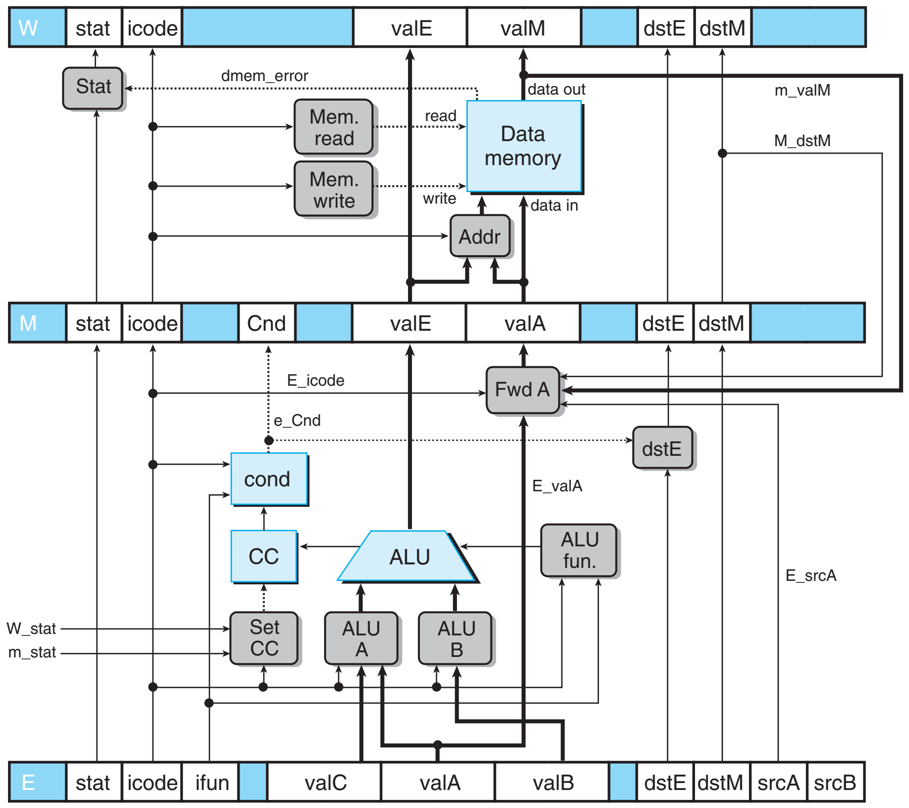


Figure 4.70 Execute and memory stages capable of load forwarding. By adding a bypass path from the memory output to the source of valA in pipeline register M, we can use forwarding rather than stalling for one form of load/use hazard. This is the subject of Problem 4.57.

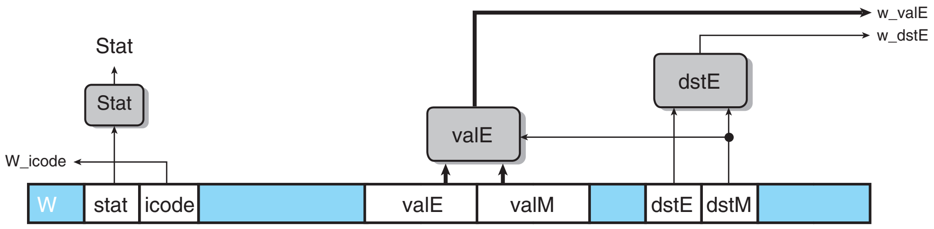
used as part of the address computation by the next instruction, and this value is required in the execute stage rather than the memory stage.

A. Write a logic formula describing the detection condition for a load/use hazard, similar to the one given in Figure 4.64, except that it will not cause a stall in cases where load forwarding can be used.

B. The file `pipe-lf.hcl` contains a modified version of the control logic for PIPE. It contains the definition of a signal e_valA to implement the block labeled “Fwd A” in Figure 4.70. It also has the conditions for a load/use hazard in the pipeline control logic set to zero, and so the pipeline control logic will not detect any forms of load/use hazards. Modify this HCL description to implement load forwarding. See the lab material for directions on how to generate a simulator for your solution and how to test it.

4.58 ◆◆◆

Our pipelined design is a bit unrealistic in that we have two write ports for the register file, but only the popq instruction requires two simultaneous writes to the register file. The other instructions could therefore use a single write port, sharing this for writing valE and valM. The following figure shows a modified version of the write-back logic, in which we merge the write-back register IDs (W_dstE and W_dstM) into a single signal w_dstE and the write-back values (W_valE and W_valM) into a single signal w_valE:



The logic for performing the merges is written in HCL as follows:

```
## Set E port register ID
word w_dstE = [
        ## writing from valM
        W_dstM != RNONE : W_dstM;
        1: W_dstE;
];

## Set E port value
word w_valE = [
        W_dstM != RNONE : W_valM;
        1: W_valE;
];
```

The control for these multiplexors is determined by dstE—when it indicates there is some register, then it selects the value for port E, and otherwise it selects the value for port M.

In the simulation model, we can then disable register port M, as shown by the following HCL code:

```
## Disable register port M
## Set M port register ID
word w_dstM = RNONE;

## Set M port value
word w_valM = 0;
```

The challenge then becomes to devise a way to handle popq. One method is to use the control logic to dynamically process the instruction popq rA so that it has the same effect as the two-instruction sequence

```
    iaddq  $8, %rsp
    mrmovq -8(%rsp), rA
```

(See Practice Problem 4.3 for a description of the `iaddq` instruction.) Note the ordering of the two instructions to make sure `popq %rsp` works properly. You can do this by having the logic in the decode stage treat `popq` the same as it would the `iaddq` listed above, except that it predicts the next PC to be equal to the current PC. On the next cycle, the `popq` instruction is refetched, but the instruction code is converted to a special value `IPOP2`. This is treated as a special instruction that has the same behavior as the `mrmovq` instruction listed above.

The file `pipe-1w.hcl` contains the modified write port logic described above. It contains a declaration of the constant `IPOP2` having hexadecimal value `E`. It also contains the definition of a signal f_icode that generates the icode field for pipeline register D. This definition can be modified to insert the instruction code `IPOP2` the second time the `popq` instruction is fetched. The HCL file also contains a declaration of the signal f_pc, the value of the program counter generated in the fetch stage by the block labeled "Select PC" (Figure 4.57).

Modify the control logic in this file to process `popq` instructions in the manner we have described. See the lab material for directions on how to generate a simulator for your solution and how to test it.

4.59 ◆◆

Compare the performance of the three versions of bubblesort (Problems 4.47, 4.48, and 4.49). Explain why one version performs better than the other.

Solutions to Practice Problems

Solution to Problem 4.1 (page 396)

Encoding instructions by hand is rather tedious, but it will solidify your understanding of the idea that assembly code gets turned into byte sequences by the assembler. In the following output from our Y86-64 assembler, each line shows an address and a byte sequence that starts at that address:

```
1   0x100:                      | .pos 0x100  # Start code at address
0x100
2   0x100: 30f30f00000000000000 |     irmovq $15,%rbx
3   0x10a: 2031                 |     rrmovq %rbx,%rcx
4   0x10c:                      | loop:
5   0x10c: 4013fdffffffffffffff |     rmmovq %rcx,-3(%rbx)
6   0x116: 6031                 |     addq   %rbx,%rcx
7   0x118: 700c01000000000000   |     jmp loop
```

Several features of this encoding are worth noting:

- Decimal 15 (line 2) has hex representation `0x000000000000000f`. Writing the bytes in reverse order gives `0f 00 00 00 00 00 00 00`.

- Decimal −3 (line 5) has hex representation `0xfffffffffffffffd`. Writing the bytes in reverse order gives `fd ff ff ff ff ff ff ff`.
- The code starts at address `0x100`. The first instruction requires 10 bytes, while the second requires 2. Thus, the loop target will be `0x0000010c`. Writing these bytes in reverse order gives `0c 01 00 00 00 00 00 00`.

Solution to Problem 4.2 (page 396)

Decoding a byte sequence by hand helps you understand the task faced by a processor. It must read byte sequences and determine what instructions are to be executed. In the following, we show the assembly code used to generate each of the byte sequences. To the left of the assembly code, you can see the address and byte sequence for each instruction.

A. Some operations with immediate data and address displacements:

```
0x100: 30f3fcffffffffffffff |     irmovq $-4,%rbx
0x10a: 40630008000000000000 |     rmmovq %rsi,0x800(%rbx)
0x114: 00                   |     halt
```

B. Code including a function call:

```
0x200: a06f                 |     pushq %rsi
0x202: 800c02000000000000   |     call proc
0x20b: 00                   |     halt
0x20c:                      | proc:
0x20c: 30f30a00000000000000 |     irmovq $10,%rbx
0x216: 90                   |     ret
```

C. Code containing illegal instruction specifier byte `0xf0`:

```
0x300: 50540700000000000000 |     mrmovq 7(%rsp),%rbp
0x30a: 10                   |     nop
0x30b: f0                   | .byte 0xf0  # Invalid instruction code
0x30c: b01f                 |     popq %rcx
```

D. Code containing a jump operation:

```
0x400:                      | loop:
0x400: 6113                 |     subq %rcx, %rbx
0x402: 730004000000000000   |     je loop
0x40b: 00                   |     halt
```

E. Code containing an invalid second byte in a `pushq` instruction:

```
0x500: 6362                 |     xorq %rsi,%rdx
0x502: a0                   |     .byte 0xa0  # pushq instruction
code
0x503: f0                   |     .byte 0xf0  # Invalid register
specifier byte
```

Solution to Problem 4.3 (page 405)

Using the iaddq instruction, we can rewrite the sum function as

```
# long sum(long *start, long count)
# start in %rdi, count in %rsi
sum:
        xorq %rax,%rax          # sum = 0
        andq %rsi,%rsi          # Set condition codes
        jmp    test
loop:
        mrmovq (%rdi),%r10      # Get *start
        addq %r10,%rax          # Add to sum
        iaddq $8,%rdi           # start++
        iaddq $-1,%rsi          # count--
test:
        jne    loop             # Stop when 0
        ret
```

Solution to Problem 4.4 (page 406)

Gcc, running on an x86-64 machine, produces the following code for rproduct:

```
  long rproduct(long *start, long count)
  start in %rdi, count in %rsi
rproduct:
  movl    $1, %eax
  testq   %rsi, %rsi
  jle     .L9
  pushq   %rbx
  movq    (%rdi), %rbx
  subq    $1, %rsi
  addq    $8, %rdi
  call    rproduct
  imulq   %rbx, %rax
  popq    %rbx
.L9:
  rep; ret
```

This can easily be adapted to produce Y86-64 code:

```
# long rproduct(long *start, long count)
# start in %rdi, count in %rsi
rproduct:
        xorq   %rax,%rax        # Set return value to 1
        andq   %rsi,%rsi        # Set condition codes
        je     return           # If count <= 0, return 1
        pushq  %rbx             # Save callee-saved register
        mrmovq (%rdi),%rbx      # Get *start
        irmovq $-1,%r10
        addq   %r10,%rsi        # count--
        irmovq $8,%r10
        addq   %r10,%rdi        # start++
        call   rproduct
        imulq  %rbx,%rax        # Multiply *start to product
        popq   %rbx             # Restore callee-saved register
return:
        ret
```

Solution to Problem 4.5 (page 406)

This problem gives you a chance to try your hand at writing assembly code.

```
# long absSum(long *start, long count)
# start in %rdi, count in %rsi
absSum:
        irmovq $8,%r8           # Constant 8
        irmovq $1,%r9           # Constant 1
        xorq %rax,%rax          # sum = 0
        andq %rsi,%rsi          # Set condition codes
        jmp  test
loop:
        mrmovq (%rdi),%r10      # x = *start
        xorq %r11,%r11          # Constant 0
        subq %r10,%r11          # -x
        jle pos                 # Skip if -x <= 0
        rrmovq %r11,%r10        # x = -x
pos:
        addq %r10,%rax          # Add to sum
        addq %r8,%rdi           # start++
        subq %r9,%rsi           # count--
test:
        jne    loop             # Stop when 0
        ret
```

Solution to Problem 4.6 (page 406)

This problem gives you a chance to try your hand at writing assembly code with conditional moves. We show only the code for the loop. The rest is the same as for Problem 4.5:

```
9   loop:
10          mrmovq (%rdi),%r10      # x = *start
11          xorq %r11,%r11          # Constant 0
12          subq %r10,%r11          # -x
13          cmovg %r11,%r10         # If -x > 0 then x = -x
```

```
            addq %r10,%rax          # Add to sum
            addq %r8,%rdi           # start++
            subq %r9,%rsi           # count--
    test:
            jne    loop             # Stop when 0
```

Solution to Problem 4.7 (page 406)

Although it is hard to imagine any practical use for this particular instruction, it is important when designing a system to avoid any ambiguities in the specification. We want to determine a reasonable convention for the instruction's behavior and to make sure each of our implementations adheres to this convention.

The `subq` instruction in this test compares the starting value of `%rsp` to the value pushed onto the stack. The fact that the result of this subtraction is zero implies that the old value of `%rsp` gets pushed.

Solution to Problem 4.8 (page 407)

It is even more difficult to imagine why anyone would want to pop to the stack pointer. Still, we should decide on a convention and stick with it. This code sequence pushes `0xabcd` onto the stack, pops to `%rsp`, and returns the popped value. Since the result equals `0xabcd`, we can deduce that `popq %rsp` sets the stack pointer to the value read from memory. It is therefore equivalent to the instruction `mrmovq (%rsp),%rsp`.

Solution to Problem 4.9 (page 410)

The EXCLUSIVE-OR function requires that the 2 bits have opposite values:

```
bool xor = (!a && b) || (a && !b);
```

In general, the signals `eq` and `xor` will be complements of each other. That is, one will equal 1 whenever the other is 0.

Solution to Problem 4.10 (page 413)

The outputs of the EXCLUSIVE-OR circuits will be the complements of the bit equality values. Using DeMorgan's laws (Web Aside DATA:BOOL on page 88), we can implement AND using OR and NOT, yielding the circuit shown in Figure 4.71.

Solution to Problem 4.11 (page 415)

We can see that the second part of the case expression can be written as

```
        B <= C           : B;
```

Since the first line will detect the case where `A` is the minimum element, the second line need only determine whether `B` or `C` is minimum.

Solution to Problem 4.12 (page 416)

This design is a variant of the one to find the minimum of the three inputs:

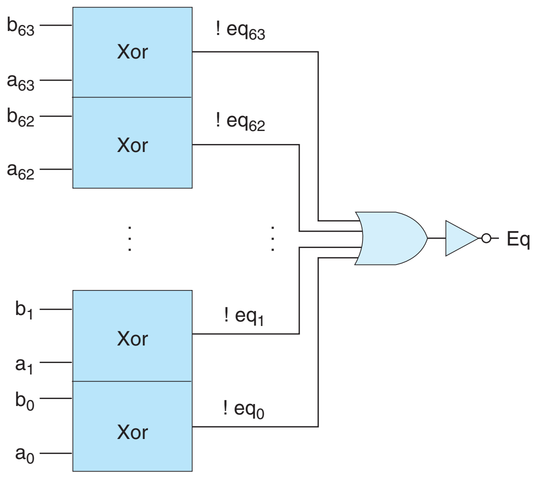


Figure 4.71
Solution for Problem 4.10.

```
word Med3 = [
        A <= B && B <= C : B;
        C <= B && B <= A : B;
        B <= A && A <= C : A;
        C <= A && A <= B : A;
        1                : C;
];
```

Solution to Problem 4.13 (page 423)

These exercises help make the stage computations more concrete. We can see from the object code that this instruction is located at address `0x016`. It consists of 10 bytes, with the first two being `0x30` and `0xf4`. The last 8 bytes are a byte-reversed version of `0x0000000000000080` (decimal 128).

Stage	Generic `irmovq V, rB`	Specific `irmovq $128, %rsp`
Fetch	icode:ifun ← M_1[PC]	icode:ifun ← M_1[`0x016`] = `3:0`
	rA:rB ← M_1[PC + 1]	rA:rB ← M_1[`0x017`] = `f:4`
	valC ← M_8[PC + 2]	valC ← M_8[`0x018`] = 128
	valP ← PC + 10	valP ← `0x016` + 10 = `0x020`
Decode		
Execute	valE ← 0 + valC	valE ← 0 + 128 = 128
Memory		
Write back	R[rB] ← valE	R[`%rsp`] ← valE = 128
PC update	PC ← valP	PC ← valP = `0x020`

This instruction sets register `%rsp` to 128 and increments the PC by 10.

Solution to Problem 4.14 (page 426)

We can see that the instruction is located at address `0x02c` and consists of 2 bytes with values `0xb0` and `0x00f`. Register `%rsp` was set to 120 by the `pushq` instruction (line 6), which also stored 9 at this memory location.

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	icode : ifun $\leftarrow$ M_1[`0x02c`] = `b:0` rA : rB $\leftarrow$ M_1[`0x02d`] = `0:f` valP $\leftarrow$ `0x02c` + 2 = `0x02e`
Decode	valA $\leftarrow$ R[`%rsp`] valB $\leftarrow$ R[`%rsp`]	valA $\leftarrow$ R[`%rsp`] = 120 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] = 9
Write back	R[`%rsp`] $\leftarrow$ valE R[rA] $\leftarrow$ valM	R[`%rsp`] $\leftarrow$ 128 R[`%rax`] $\leftarrow$ 9
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ `0x02e`

The instruction sets `%rax` to 9, sets `%rsp` to 128, and increments the PC by 2.

Solution to Problem 4.15 (page 427)

Tracing the steps listed in Figure 4.20 with rA equal to `%rsp`, we can see that in the memory stage the instruction will store valA, the original value of the stack pointer, to memory, just as we found for x86-64.

Solution to Problem 4.16 (page 428)

Tracing the steps listed in Figure 4.20 with rA equal to `%rsp`, we can see that both of the write-back operations will update `%rsp`. Since the one writing valM would occur last, the net effect of the instruction will be to write the value read from memory to `%rsp`, just as we saw for x86-64.

Solution to Problem 4.17 (page 429)

Implementing conditional moves requires only minor changes from register-to-register moves. We simply condition the write-back step on the outcome of the conditional test:

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]

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

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`] = `0x041`
	valP $\leftarrow$ PC + 9	valP $\leftarrow$ `0x037` + 9 = `0x040`
Decode		
	valB $\leftarrow$ R[`%rsp`]	valB $\leftarrow$ R[`%rsp`] = 128
Execute	valE $\leftarrow$ valB + −8	valE $\leftarrow$ 128 + −8 = 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 $\leftarrow$ `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 };
```

Solution to Problem 4.20 (page 443)

This code is similar to the code for srcA.

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

Solution to Problem 4.21 (page 444)

This code is similar to the code for dstE.

```
word dstM = [
        icode in { IMRMOVQ, IPOPQ } : rA;
        1 : RNONE;  # Don't write any register
];
```

Solution to Problem 4.22 (page 444)

As we found in Practice Problem 4.16, we want the write via the M port to take priority over the write via the E port in order to store the value read from memory into `%rsp`.

Solution to Problem 4.23 (page 445)

This code is similar to the code for aluA.

```
word aluB = [
        icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                      IPUSHQ, IRET, IPOPQ } : valB;
        icode in { IRRMOVQ, IIRMOVQ } : 0;
        # Other instructions don't need ALU
];
```

Solution to Problem 4.24 (page 445)

Implementing conditional moves is surprisingly simple: we disable writing to the register file by setting the destination register to `RNONE` when the condition does not hold.

```
word dstE = [
        icode in { IRRMOVQ } && Cnd : rB;
        icode in { IIRMOVQ, IOPQ} : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];
```

Solution to Problem 4.25 (page 446)

This code is similar to the code for mem_addr.

```
word mem_data = [
        # Value from register
        icode in { IRMMOVQ, IPUSHQ } : valA;
        # Return PC
        icode == ICALL : valP;
        # Default: Don't write anything
];
```

Solution to Problem 4.26 (page 446)

This code is similar to the code for mem_read.

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

Solution to Problem 4.27 (page 447)

Computing the Stat field requires collecting status information from several stages:

```
## Determine instruction status
word Stat = [
        imem_error || dmem_error : SADR;
        !instr_valid: SINS;
        icode == IHALT : SHLT;
        1 : SAOK;
];
```

Solution to Problem 4.28 (page 453)

This problem is an interesting exercise in trying to find the optimal balance among a set of partitions. It provides a number of opportunities to compute throughputs and latencies in pipelines.

A. For a two-stage pipeline, the best partition would be to have blocks A, B, and C in the first stage and D, E, and F in the second. The first stage has a delay of 170 ps, giving a total cycle time of $170 + 20 = 190$ ps. We therefore have a throughput of 5.26 GIPS and a latency of 380 ps.

B. For a three-stage pipeline, we should have blocks A and B in the first stage, blocks C and D in the second, and blocks E and F in the third. The first two stages have a delay of 110 ps, giving a total cycle time of 130 ps and a throughput of 7.69 GIPS. The latency is 390 ps.

C. For a four-stage pipeline, we should have block A in the first stage, blocks B and C in the second, block D in the third, and blocks E and F in the fourth. The second stage requires 90 ps, giving a total cycle time of 110 ps and a throughput of 9.09 GIPS. The latency is 440 ps.

D. The optimal design would be a five-stage pipeline, with each block in its own stage, except that the fifth stage has blocks E and F. The cycle time is $80 + 20 = 100$ ps, for a throughput of around 10.00 GIPS and a latency of

500 ps. Adding more stages would not help, since we cannot run the pipeline any faster than one cycle every 100 ps.

Solution to Problem 4.29 (page 454)

Each stage would have combinational logic requiring $300/k$ ps and a pipeline register requiring 20 ps.

A. The total latency would be $300 + 20k$ ps, while the throughput (in GIPS) would be

$$\frac{1{,}000}{\frac{300}{k} + 20} = \frac{1{,}000k}{300 + 20k}$$

B. As we let k go to infinity, the throughput becomes $1{,}000/20 = 50$ GIPS. Of course, the latency would approach infinity as well.

This exercise quantifies the diminishing returns of deep pipelining. As we try to subdivide the logic into many stages, the latency of the pipeline registers becomes a limiting factor.

Solution to Problem 4.30 (page 485)

This code is very similar to the corresponding code for SEQ, except that we cannot yet determine whether the data memory will generate an error signal for this instruction.

```
# Determine status code for fetched instruction
word f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];
```

Solution to Problem 4.31 (page 485)

This code simply involves prefixing the signal names in the code for SEQ with `d_` and `D_`.

```
word d_dstE = [
        D_icode in { IRRMOVQ, IIRMOVQ, IOPQ} : D_rB;
        D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];
```

Solution to Problem 4.32 (page 488)

The `rrmovq` instruction (line 5) would stall for one cycle due to a load/use hazard caused by the `popq` instruction (line 4). As it enters the decode stage, the `popq` instruction would be in the memory stage, giving both M_dstE and M_dstM equal to `%rsp`. If the two cases were reversed, then the write back from M_valE would take priority, causing the incremented stack pointer to be passed as the argument to the rrmovq instruction. This would not be consistent with the convention for handling popq %rsp determined in Practice Problem 4.8.

Solution to Problem 4.33 (page 488)

This problem lets you experience one of the important tasks in processor design—devising test programs for a new processor. In general, we should have test programs that will exercise all of the different hazard possibilities and will generate incorrect results if some dependency is not handled properly.

For this example, we can use a slightly modified version of the program shown in Practice Problem 4.32:

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

The two nop instructions will cause the popq instruction to be in the write-back stage when the rrmovq instruction is in the decode stage. If the two forwarding sources in the write-back stage are given the wrong priority, then register %rax will be set to the incremented program counter rather than the value read from memory.

Solution to Problem 4.34 (page 489)

This logic only needs to check the five forwarding sources:

```
word d_valB = [
        d_srcB == e_dstE : e_valE;    # Forward valE from execute
        d_srcB == M_dstM : m_valM;    # Forward valM from memory
        d_srcB == M_dstE : M_valE;    # Forward valE from memory
        d_srcB == W_dstM : W_valM;    # Forward valM from write back
        d_srcB == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalB;  # Use value read from register file
];
```

Solution to Problem 4.35 (page 490)

This change would not handle the case where a conditional move fails to satisfy the condition, and therefore sets the dstE value to RNONE. The resulting value could get forwarded to the next instruction, even though the conditional transfer does not occur.

```
            irmovq $0x123,%rax
            irmovq $0x321,%rdx
            xorq %rcx,%rcx          # CC = 100
            cmovne  %rax,%rdx       # Not transferred
            addq %rdx,%rdx          # Should be 0x642
            halt
```

This code initializes register %rdx to 0x321. The conditional data transfer does not take place, and so the final addq instruction should double the value in %rdx to 0x642. With the altered design, however, the conditional move source value 0x321 gets forwarded into ALU input valA, while input valB correctly gets operand value 0x123. These inputs get added to produce result 0x444.

Solution to Problem 4.36 (page 491)

This code completes the computation of the status code for this instruction.

```
## Update the status
word m_stat = [
        dmem_error : SADR;
        1 : M_stat;
];
```

Solution to Problem 4.37 (page 497)

The following test program is designed to set up control combination A (Figure 4.67) and detect whether something goes wrong:

```
# Code to generate a combination of not-taken branch and ret
        irmovq Stack, %rsp
        irmovq rtnp,%rax
        pushq %rax       # Set up return pointer
        xorq %rax,%rax   # Set Z condition code
        jne target       # Not taken (First part of combination)
        irmovq $1,%rax   # Should execute this
        halt
target: ret              # Second part of combination
        irmovq $2,%rbx   # Should not execute this
        halt
rtnp:   irmovq $3,%rdx   # Should not execute this
        halt
.pos 0x40
Stack:
```

This program is designed so that if something goes wrong (for example, if the ret instruction is actually executed), then the program will execute one of the extra irmovq instructions and then halt. Thus, an error in the pipeline would cause some register to be updated incorrectly. This code illustrates the care required to implement a test program. It must set up a potential error condition and then detect whether or not an error occurs.

Solution to Problem 4.38 (page 498)

The following test program is designed to set up control combination B (Figure 4.67). The simulator will detect a case where the bubble and stall control signals for a pipeline register are both set to zero, and so our test program need only set up the combination for it to be detected. The biggest challenge is to make the program do something sensible when handled correctly.

```
# Test instruction that modifies %esp followed by ret
        irmovq mem,%rbx
        mrmovq  0(%rbx),%rsp # Sets %rsp to point to return point
        ret                  # Returns to return point
        halt                 #
rtnpt:  irmovq $5,%rsi       # Return point
        halt
.pos 0x40
mem:    .quad stack          # Holds desired stack pointer
.pos 0x50
stack:  .quad rtnpt          # Top of stack: Holds return point
```

This program uses two initialized words in memory. The first word (`mem`) holds the address of the second (`stack`—the desired stack pointer). The second word holds the address of the desired return point for the `ret` instruction. The program loads the stack pointer into `%rsp` and executes the `ret` instruction.

Solution to Problem 4.39 (page 499)

From Figure 4.66, we can see that pipeline register D must be stalled for a load/use hazard:

```
bool D_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB };
```

Solution to Problem 4.40 (page 500)

From Figure 4.66, we can see that pipeline register E must be set to bubble for a load/use hazard or for a mispredicted branch:

```
bool E_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB};
```

Solution to Problem 4.41 (page 500)

This control requires examining the code of the executing instruction and checking for exceptions further down the pipeline.

```
## Should the condition codes be updated?
bool set_cc = E_icode == IOPQ &&
       # State changes only during normal operation
       !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };
```

Solution to Problem 4.42 (page 500)

Injecting a bubble into the memory stage on the next cycle involves checking for an exception in either the memory or the write-back stage during the current cycle.

```
# Start injecting bubbles as soon as exception passes through memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };
```

For stalling the write-back stage, we check only the status of the instruction in this stage. If we also stalled when an excepting instruction was in the memory stage, then this instruction would not be able to enter the write-back stage.

```
bool W_stall = W_stat in { SADR, SINS, SHLT };
```

Solution to Problem 4.43 (page 504)

We would then have a misprediction frequency of 0.35, giving $mp = 0.20 \times 0.35 \times 2 = 0.14$, giving an overall CPI of 1.25. This seems like a fairly marginal gain, but it would be worthwhile if the cost of implementing the new branch prediction strategy were not too high.

Solution to Problem 4.44 (page 504)

This simplified analysis, where we focus on the inner loop, is a useful way to estimate program performance. As long as the array is sufficiently large, the time spent in other parts of the code will be negligible.

A. The inner loop of the code using the conditional jump has 11 instructions, all of which are executed when the array element is zero or negative, and 10 of which are executed when the array element is positive. The average is 10.5. The inner loop of the code using the conditional move has 10 instructions, all of which are executed every time.

B. The loop-closing jump will be predicted correctly, except when the loop terminates. For a very long array, this one misprediction will have a negligible effect on the performance. The only other source of bubbles for the jump-based code is the conditional jump, depending on whether or not the array element is positive. This will cause two bubbles, but it only occurs 50% of the time, so the average is 1.0. There are no bubbles in the conditional move code.

C. Our conditional jump code requires an average of $10.5 + 1.0 = 11.5$ cycles per array element (11 cycles in the best case and 12 cycles in the worst), while our conditional move code requires 10.0 cycles in all cases.

Our pipeline has a branch misprediction penalty of only two cycles—far better than those for the deep pipelines of higher-performance processors. As a result, using conditional moves does not affect program performance very much.