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

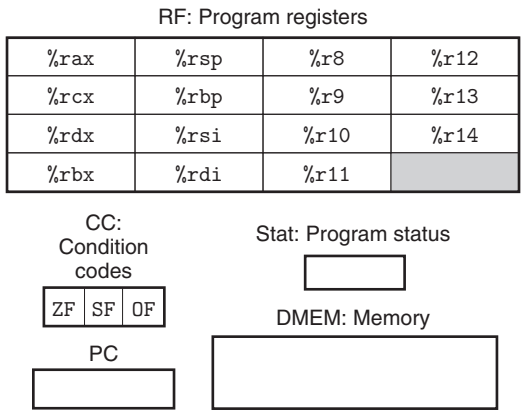


Figure 4.1
Y86-64 programmer-visible state. As with x86-64, programs for Y86-64 access and modify the program registers, the condition codes, the program counter (PC), and the memory. The status code indicates whether the program is running normally or some special event has occurred.

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).

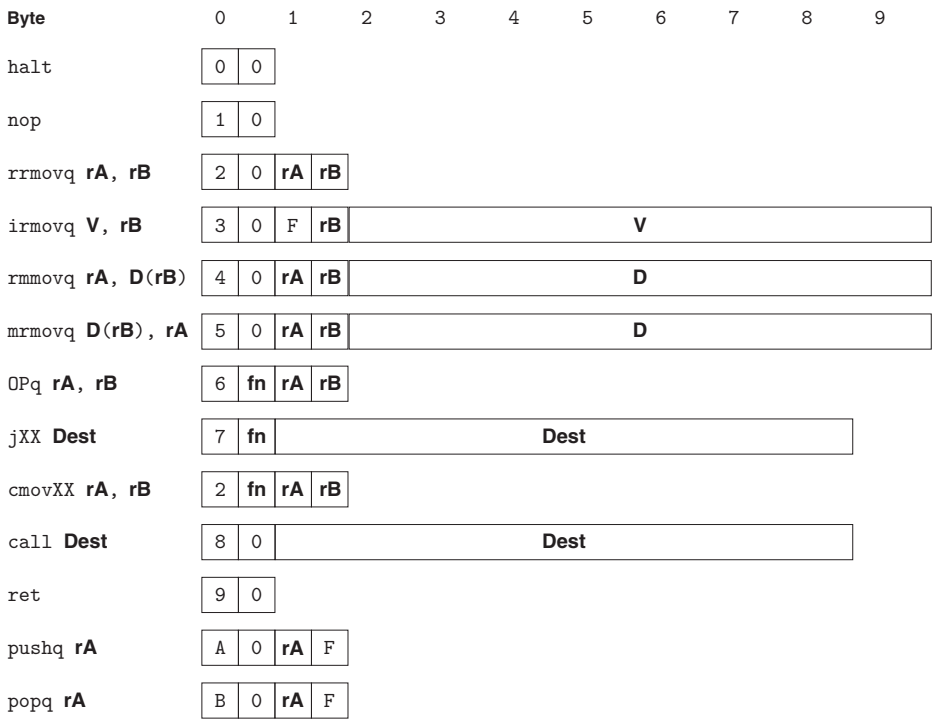


Figure 4.2 Y86-64 instruction set. Instruction encodings range between 1 and 10 bytes. An instruction consists of a 1-byte instruction specifier, possibly a 1-byte register specifier, and possibly an 8-byte constant word. Field `fn` specifies a particular integer operation (`OPq`), data movement condition (`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

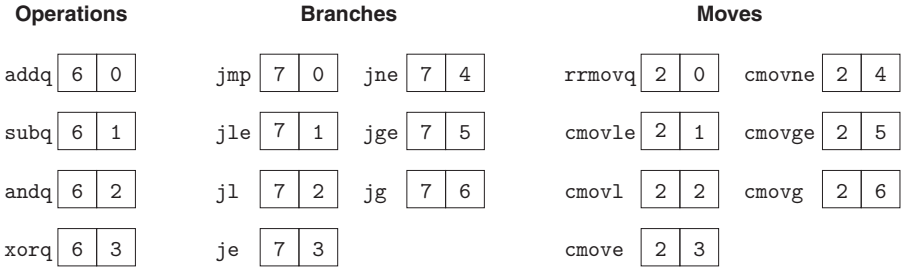


Figure 4.3 Function codes for Y86-64 instruction set. The code specifies a particular integer operation, branch condition, or data transfer condition. These instructions are shown as `OPq`, `jXX`, and `cmovXX` in Figure 4.2.

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: 30f3fcffffffffffffff4063000800000000000`

B. `0x200: a06f800c020000000000000030f30a00000000000000`

C. `0x300: 505407000000000000010f0b01f`

D. `0x400: 6113730004000000000000`

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:

```
1   long sum(long *start, long count)
2   {
3       long sum = 0;
4       while (count) {
5           sum += *start;
6           start++;
7           count--;
8       }
9       return sum;
10  }
```

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
1   sum:
2     movl    $0, %eax          sum = 0
3     jmp     .L2               Goto test
4   .L3:                      loop:
5     addq    (%rdi), %rax      Add *start to sum
6     addq    $8, %rdi          start++
7     subq    $1, %rsi          count--
8   .L2:                      test:
9     testq   %rsi, %rsi        Test sum
10    jne     .L3               If !=0, goto loop
11    rep; ret                  Return
```

Y86-64 code

```
    long sum(long *start, long count)
    start in %rdi, count in %rsi
1   sum:
2     irmovq $8,%r8           Constant 8
3     irmovq $1,%r9           Constant 1
4     xorq %rax,%rax          sum = 0
5     andq %rsi,%rsi          Set CC
6     jmp     test            Goto test
7   loop:
8     mrmovq (%rdi),%r10      Get *start
9     addq %r10,%rax          Add to sum
10    addq %r8,%rdi           start++
11    subq %r9,%rsi           count--.  Set CC
12  test:
13    jne    loop             Stop when 0
14    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

```
1   # Execution begins at address 0
2           .pos 0
3           irmovq stack, %rsp      # Set up stack pointer
4           call main               # Execute main program
5           halt                    # Terminate program
6
7   # Array of 4 elements
8           .align 8
9   array:
10          .quad 0x000d000d000d
11          .quad 0x00c000c000c0
12          .quad 0x0b000b000b00
13          .quad 0xa000a000a000
14
15  main:
16          irmovq array,%rdi
17          irmovq $4,%rsi
18          call sum                # sum(array, 4)
19          ret
20
21  # long sum(long *start, long count)
22  # start in %rdi, count in %rsi
23  sum:
24          irmovq $8,%r8           # Constant 8
25          irmovq $1,%r9           # Constant 1
26          xorq %rax,%rax          # sum = 0
27          andq %rsi,%rsi          # Set CC
28          jmp     test            # Goto test
29  loop:
30          mrmovq (%rdi),%r10      # Get *start
31          addq %r10,%rax          # Add to sum
32          addq %r8,%rdi           # start++
33          subq %r9,%rsi           # count--.  Set CC
34  test:
35          jne    loop             # Stop when 0
36          ret                     # Return
37
38  # Stack starts here and grows to lower addresses
39          .pos 0x200
40  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: 30f718000000000000 |   irmovq array,%rdi
0x042: 30f604000000000000 |   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: 30f808000000000000 |   irmovq $8,%r8         # Constant 8
0x060: 30f901000000000000 |   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: 50a700000000000000 |   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:

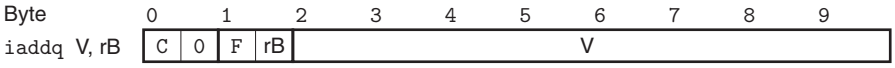
```
Stopped in 34 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000abcdabcdabcd
%rsp:   0x0000000000000000      0x0000000000000200
%rdi:   0x0000000000000000      0x0000000000000038
%r8:    0x0000000000000000      0x0000000000000008
%r9:  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

```
1      .text
2    .globl pushtest
3    pushtest:
4      movq    %rsp, %rax    Copy stack pointer
5      pushq   %rsp          Push stack pointer
6      popq    %rdx          Pop it back
7      subq    %rdx, %rax    Return 0 or 4
8      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:

```
1      .text
2    .globl poptest
3    poptest:
4      movq    %rsp, %rdi    Save stack pointer
5      pushq   $0xabcd       Push test value
6      popq    %rsp          Pop to stack pointer
7      movq    %rsp, %rax    Set popped value as return value
8      movq    %rdi, %rsp    Restore stack pointer
9      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.

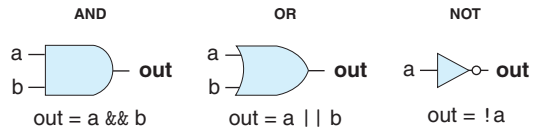


Figure 4.9
Logic gate types. Each gate generates output equal to some Boolean function of its inputs.

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.

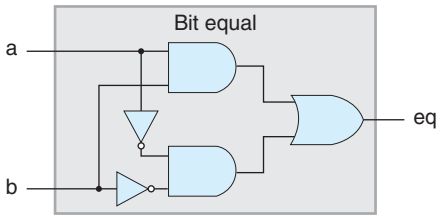


Figure 4.10
Combinational circuit to test for bit equality. The output will equal 1 when both inputs are 0 or both are 1.

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

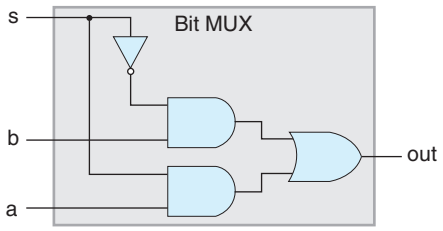


Figure 4.11
Single-bit multiplexor circuit. The output will equal input a if the control signal s is 1 and will equal input b when s is 0.

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.

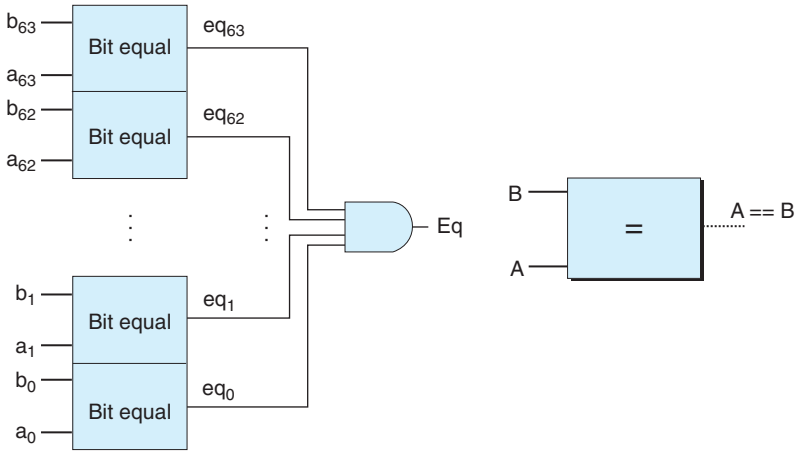


Figure 4.12 Word-level equality test circuit. The output will equal 1 when each bit from word A equals its counterpart from word B. Word-level equality is one of the operations in HCL.

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);
```

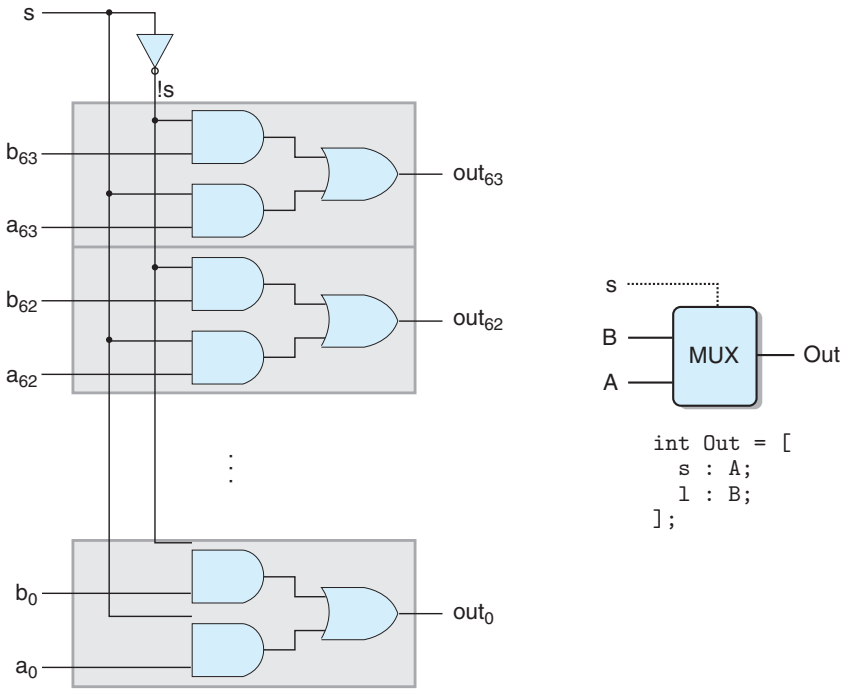
where arguments A and B are of type `int`. Note that we use the same syntax conventions as in C, where '=' denotes assignment and '==' denotes the equality operator.

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.



(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:

```
[
    select_1   :   expr_1;
    select_2   :   expr_2;
    .
    .
    .
    select_k   :   expr_k;
]
```

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
```

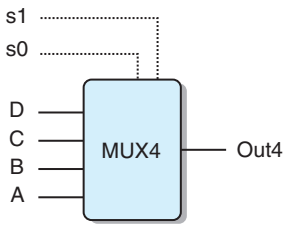
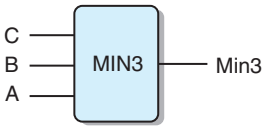


Figure 4.14
Four-way multiplexor. The different combinations of control signals s1 and s0 determine which data input is transmitted to the output.

```
        !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.

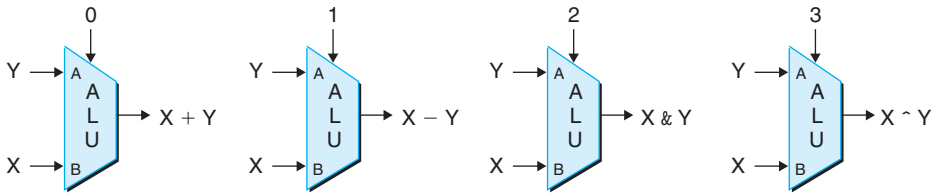


Figure 4.15 Arithmetic/logic unit (ALU). Depending on the setting of the function input, the circuit will perform one of four different arithmetic and logical operations.

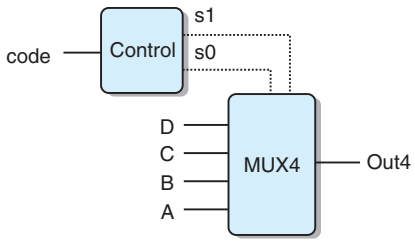
Practice Problem 4.12 (solution page 520)

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

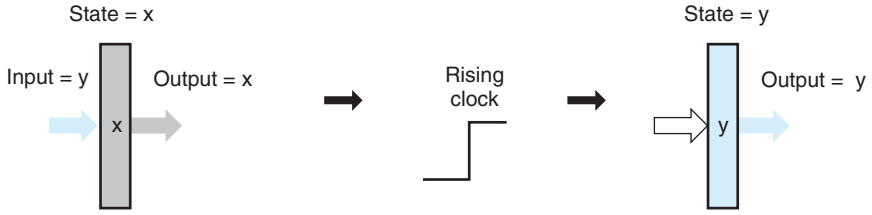
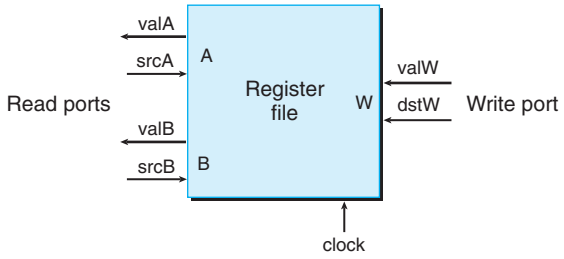


Figure 4.16 Register operation. The register outputs remain held at the current register state until the clock signal rises. When the clock rises, the values at the register inputs are captured to become the new register state.

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:



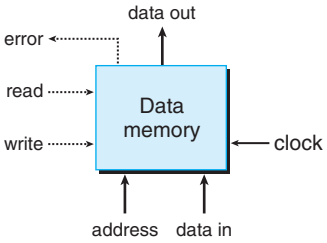
This register file has two *read ports*, named A and B, and one *write port*, named W. Such a *multiported* random access memory allows multiple read and write operations to take place simultaneously. In the register file diagrammed, the circuit can read the values of two program registers and update 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

```
1   0x000: 30f20900000000000000 |     irmovq $9,  %rdx
2   0x00a: 30f31500000000000000 |     irmovq $21, %rbx
3   0x014: 6123                 |     subq %rdx, %rbx          # subtract
4   0x016: 30f480000000000000000 |     irmovq $128,%rsp         # Problem 4.13
5   0x020: 4043640000000000000 |     rmmovq %rsp, 100(%rbx)   # store
6   0x02a: a02f                 |     pushq %rdx               # push
7   0x02c: b00f                 |     popq  %rax               # Problem 4.14
8   0x02e: 734000000000000000   |     je done                  # Not taken
9   0x037: 804100000000000000   |     call proc                # Problem 4.18
10  0x040:                      | done:
11  0x040: 00                   |     halt
12  0x041:                      | proc:
13  0x041: 90                   |     ret                      # Return
14                              |
```

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

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

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

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

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

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

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

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

Practice Problem 4.13 (solution page 521)

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

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

Aside Tracing the execution of a `subq` instruction

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

Stage	`OPq` rA, rB	`subq %rdx, %rbx`
Fetch	icode:ifun $\leftarrow$ M_1[PC]	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] rA:rB $\leftarrow$ M_1[PC + 1] valC $\leftarrow$ M_8[PC + 2] valP $\leftarrow$ PC + 10	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valC $\leftarrow$ M_8[PC + 2] valP $\leftarrow$ PC + 10
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	valB $\leftarrow$ R[rB]
Execute	valE $\leftarrow$ valB + valC	valE $\leftarrow$ valB + valC
Memory	M_8[valE] $\leftarrow$ valA	valM $\leftarrow$ M_8[valE]
Write back		R[rA] $\leftarrow$ valM
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP

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

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

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

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

Aside Tracing the execution of an `rmmovq` instruction

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

Stage	Generic `rmmovq rA, D(rB)`	Specific `rmmovq %rsp, 100(%rbx)`
Fetch	icode:ifun $\leftarrow$ M_1[PC]	icode:ifun $\leftarrow$ M_1[0x020] = 4:0
	rA:rB $\leftarrow$ M_1[PC + 1]	rA:rB $\leftarrow$ M_1[0x021] = 4:3
	valC $\leftarrow$ M_8[PC + 2]	valC $\leftarrow$ M_8[0x022] = 100
	valP $\leftarrow$ PC + 10	valP $\leftarrow$ 0x020 + 10 = 0x02a
Decode	valA $\leftarrow$ R[rA]	valA $\leftarrow$ R[%rsp] = 128
	valB $\leftarrow$ R[rB]	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]	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1]
	valP $\leftarrow$ PC + 2	valP $\leftarrow$ PC + 2
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[`%rsp`]	valA $\leftarrow$ R[`%rsp`] valB $\leftarrow$ R[`%rsp`]
Execute	valE $\leftarrow$ valB + (−8)	valE $\leftarrow$ valB + 8
Memory	M_8[valE] $\leftarrow$ valA	valM $\leftarrow$ M_8[valA]
Write back	R[`%rsp`] $\leftarrow$ valE	R[`%rsp`] $\leftarrow$ valE R[rA] $\leftarrow$ valM
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP

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

Stage	Generic `popq` rA	Specific `popq %rax`
Decode	valA $\leftarrow$ R[`%rsp`] valB $\leftarrow$ R[`%rsp`]	
Execute	valE $\leftarrow$ valB + 8	
Memory	valM $\leftarrow$ M_8[valA]	
Write back	R[`%rsp`] $\leftarrow$ valE R[rA] $\leftarrow$ valM	
PC update	PC $\leftarrow$ valP	

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

Practice Problem 4.15 (solution page 522)

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

Aside Tracing the execution of a `pushq` instruction

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

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

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

Practice Problem 4.16 (solution page 522)

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

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

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

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

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

Practice Problem 4.17 (solution page 522)

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

Stage	`cmovXX` rA, rB
Fetch	icode:ifun ← M_1[PC]
	rA:rB ← M_1[PC + 1]
	valP ← PC + 2
Decode	valA ← R[rA]
Execute	valE ← 0 + valA
Memory	
Write back	
	R[rB] ← valE
PC update	PC ← valP

Aside Tracing the execution of a `je` instruction

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

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

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

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

Practice Problem 4.18 **(solution page 523)**

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

Stage	Generic `call` Dest	Specific `call 0x041`
Fetch	icode:ifun $\leftarrow$ M_1[PC]	
	valC $\leftarrow$ M_8[PC + 1]	
	valP $\leftarrow$ PC + 9	