2.1 Introduction

This chapter discusses how to measure, report, and summarize performance and describes the major factors that determine the performance of a computer. A primary reason for examining performance is that hardware performance is often key to the effectiveness of an entire system of hardware and software.

Assessing the performance of such a system can be quite challenging. The scale and intricacy of modern software systems, together with the wide range of performance improvement techniques employed by hardware designers, have made performance assessment much more difficult. It is simply impossible to sit down with an instruction set manual and a significant software system and determine how fast the software will run on the machine. In fact, for different types of applications, different performance metrics may be appropriate and different aspects of a computer system may be the most significant in determining overall performance.

Of course, in trying to choose among different computers, performance is almost always an important attribute. Accurately measuring and comparing different machines is critical to purchasers, and therefore to designers. The people selling computers know this as well. Often, salespeople would like you to see their machine in the best possible light, whether or not this light accurately reflects the needs of the purchaser's application. In some cases, claims are made about computers that don't provide useful insight for any real applications. Hence, understanding how best to measure performance and the limitations of performance measurements is important in selecting a machine.

Our interest in performance, however, goes beyond issues of assessing performance only from the outside of a machine. To understand why a piece of software performs as it does, why one instruction set can be implemented to perform better than another, or how some hardware feature affects performance, we need to understand what determines the performance of a machine. For example, to improve the performance of a software system, we may need to understand what factors in the hardware contribute to the overall system performance and the relative importance of these factors. These factors may include how well the program uses the instructions of the machine, how well the underlying hardware implements the instructions, and how well the memory and I/O systems perform. Understanding how to determine the performance impact of these factors is crucial to understanding the motivation behind the design of particular aspects of the machine, as we will see in the chapters that follow.

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers x m.p.h.)
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

FIGURE 2.1 The capacity, range, and speed for a number of commercial airplanes. The last column shows the rate at which the airplane transports passengers, which is the capacity times the cruising speed (ignoring range and takeoff and landing times)

The rest of this section describes different ways in which performance can be determined. In section 2.2, we describe the metrics for measuring performance from the viewpoint of both a computer user and a designer. In section 2.3, we look at how these metrics are related and present the classical processor performance equation, which we will use throughout the text. Sections 2.4 and 2.5 describe how best to choose benchmarks to evaluate machines and how to accurately summarize the performance of a group of programs. Section 2.6 describes one set of commonly used CPU benchmarks and examines measurements for a variety of Intel processors using those benchmarks. Finally, in section 2.7, we'll examine some of the many pitfalls that have trapped designers and those who analyze and report performance.

Defining Performance

When we say one computer has better performance than another, what do we mean? Although this question might seem simple, an analogy with passenger airplanes shows how subtle the question of performance can be. Figure 2.1 shows some typical passenger airplanes, together with their cruising speed, range, and capacity. If we wanted to know which of the planes in this table had the best performance, we would first need to define performance. For example, considering different measures of performance, we see that the plane with the highest cruising speed is the Concorde, the plane with the longest range is the DC-8, and the plane with the largest capacity is the 747.

Let's suppose we define performance in terms of speed. This still leaves two possible definitions. You could define the fastest plane as the one with the highest cruising speed, taking a single passenger from one point to another in the least time. If you were interested in transporting 450 passengers from one point to another, however, the 747 would clearly be the fastest, as the last column of the figure shows. Similarly, we can define computer performance in several different ways.

If you were running a program on two different workstations, you'd say that the faster one is the workstation that gets the job done first. If you were

running a computer center that had two large timeshared computers running jobs submitted by many users, you'd say that the faster computer was the one that completed the most jobs during a day. As an individual computer user, you are interested in reducing *response time*—the time between the start and completion of a task—also referred to as *execution time*. Computer center managers are often interested in increasing *throughput*—the total amount of work done in a given time.

Throughput and Response Time

Example

To illustrate the application of new ideas, specific examples are used throughout this text. We highlight the example and then provide an answer. Try working out the answer yourself, or—if you feel unsure about the material—just follow along. The examples that appear are similar in type to the problems that you will have an opportunity to tackle in the exercises at the end of each chapter. Here's our first example:

Do the following changes to a computer system increase throughput, decrease response time, or both?

1. Replacing the processor in a computer with a faster version
2. Adding additional processors to a system that uses multiple processors for separate tasks—for example, handling an airline reservations system

Answer

Decreasing response time almost always improves throughput. Hence, in case 1, both response time and throughput are improved. In case 2, no one task gets work done faster, so only throughput increases. If, however, the demand for processing in the second case was almost as large as the throughput, the system might force requests to queue up. In this case, increasing the throughput could also improve response time, since it would reduce the waiting time in the queue. Thus, in many real computer systems, changing either execution time or throughput often affects the other.

In discussing the performance of machines, we will be primarily concerned with response time for the first few chapters. (In Chapter 8, on input/output systems, we will discuss throughput-related measures.) To maximize performance, we want to minimize response time or execution time for some task. Thus we can relate performance and execution time for a machine X: .

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two machines X and Y, if the performance of X is greater than the performance of Y, we have

$$\text{Performance}_X > \text{Performance}_Y$$

$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

In discussing a computer design, we often want to relate the performance of two different machines quantitatively. We will use the phrase "X is n times faster than Y" to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is n times faster than Y, then the execution time on Y is n times longer than it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Relative Performance

Example

If machine A runs a program in 10 seconds and machine B runs the same program in 15 seconds, how much faster is A than B?

Answer

We know that A is n times faster than B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = n$$

or

$$\frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times faster than B.

In the above example, we could also say that machine B is 1.5 times *slower than* machine A, since

$$\frac{\text{Performance}_A}{\text{Performance}_B} = 1.5$$

means that

$$\frac{\text{Performance}_A}{1.5} = \text{Performance}_B$$

For simplicity, we will normally use the terminology *faster than* when we try to compare machines quantitatively. Because performance and execution time are reciprocals, increasing performance requires decreasing execution time. To avoid the potential confusion between the terms *increasing* and *decreasing*, we usually say "improve performance" or "improve execution time" when we mean "increase performance" and "decrease execution time."

2.2 Measuring Performance

Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program *execution time* is measured in seconds per program. But time can be defined in different ways, depending on what we count. The most straightforward definition of time is called *wall-clock time*, *response time*, or *elapsed time*. These terms mean the total time to complete a task, including disk accesses, memory accesses, input/output (I/O) activities, operating system overhead—everything.

Computers are often timeshared, however, and a processor may work on several programs simultaneously. In such cases, the system may try to optimize throughput rather than attempt to minimize the elapsed time for one program. Hence, we often want to distinguish between the elapsed time and the time that the processor is working on our behalf. *CPU execution time* or simply *CPU time*, which recognizes this distinction, is the time the CPU spends computing for this task and does not include time spent waiting for I/O or running other programs. (Remember, though, that the response time experienced by the user will be the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called *user CPU time*, and the CPU time spent in the operating system performing tasks on behalf of the program, called *system CPU time*. Differentiating between system and user CPU time is difficult to do accurately because it is often hard to assign responsibility for operating system activities to one user program rather than another.

The breakdown of the elapsed time for a task is reflected in the Unix `time` command, which, for example, might return the following:

```
90.7u 12.9s 2:39 65%
```

User CPU time is 90.7 seconds, system CPU time is 12.9 seconds, elapsed time is 2 minutes and 39 seconds (159 seconds), and the percentage of elapsed time that is CPU time is

$$\frac{90.7 + 12.9}{159} = 0.65$$

or 65%. More than a third of the elapsed time in this example was spent waiting for I/O, running other programs, or both.

Sometimes we ignore system CPU time when examining CPU execution time because of the inaccuracy of operating systems' self-measurement and the inequity of including system CPU time when comparing performance between machines with different operating systems. On the other hand, system code on some machines is user code on others, and no program runs without some operating system running on the hardware, so a case can be made for using the sum of user CPU time and system CPU time as the measure of program execution time.

For consistency, we maintain a distinction between performance based on elapsed time and that based on CPU execution time. We will use the term *system performance* to refer to elapsed time on an unloaded system, and use *CPU performance* to refer to user CPU time. We will concentrate on CPU performance in this chapter, although our discussions of how to summarize performance can be applied to either elapsed time or to CPU time measurements.

Although as computer users we care about time, when we examine the details of a machine it's convenient to think about performance in other metrics. In particular, computer designers may want to think about a machine by using a measure that relates to how fast the hardware can perform basic functions. Almost all computers are constructed using a clock that runs at a constant rate and determines when events take place in the hardware. These discrete time intervals are called *clock cycles* (or ticks, clock ticks, clock periods, clocks, cycles). Designers refer to the length of a *clock period* both as the time for a complete *clock cycle* (e.g., 2 nanoseconds, or 2 ns) and as the *clock rate* (e.g., 500 megahertz, or 500 MHz), which is the inverse of the clock period. In the next section, we will formalize the relationship between the clock cycles of the hardware designer and the seconds of the computer user.

2.3 Relating the Metrics

Users and designers often examine performance using different metrics. If we could relate these different metrics, we could determine the effect of a design change on the performance as seen by the user. Since we are confining ourselves to CPU performance at this point, the bottom-line performance measure is CPU execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:

$$\text{CPU execution time for a program} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

This formula makes it clear that the hardware designer can improve performance by reducing either the length of the clock cycle or the number of clock cycles required for a program. As we will see in this chapter and later in Chapters 5, 6, and 7, the designer often faces a trade-off between the number of clock cycles needed for a program and the length of each cycle. Many techniques that decrease the number of clock cycles also increase the clock cycle time.

Improving Performance

Example Our favorite program runs in 10 seconds on computer A, which has a 400-MHz clock. We are trying to help a computer designer build a machine, B, that will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing machine B to require 1.2 times as many clock cycles as machine A for this program. What clock rate should we tell the designer to target?

Answer Let's first find the number of clock cycles required for the program on A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{400 \times 10^6 \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 400 \times 10^6 \frac{\text{cycles}}{\text{second}} = 4000 \times 10^6 \text{ cycles}$$

CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 4000 \times 10^6 \text{ cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 4000 \times 10^6 \text{ cycles}}{6 \text{ seconds}} = \frac{800 \times 10^6 \text{ cycles}}{\text{second}} = 800 \text{ MHz}$$

Machine B must therefore have twice the clock rate of A to run the program in 6 seconds.

Hardware Software Interface

Throughout this text, you will see sections called "Hardware Software Interface." These sections highlight major interactions between some aspect of the software (typically a program, a compiler, or an operating system) and some hardware aspect of a computer. In addition to highlighting such interactions, these sections are reminders that hardware and software design interact in many ways.

The equations in our previous examples do not include any reference to the number of instructions needed for the program. However, since the compiler clearly generated instructions to execute, and the machine had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program. One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \text{Average clock cycles per instruction}$$

The term *clock cycles per instruction*, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as CPI. Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program. CPI provides one way of comparing two different implementations of the same instruction set architecture, since the instruction count required for a program will, of course, be the same.

Using the Performance Equation

Example

Suppose we have two implementations of the same instruction set architecture. Machine A has a clock cycle time of 1 ns and a CPI of 2.0 for some program, and machine B has a clock cycle time of 2 ns and a CPI of 1.2 for the same program. Which machine is faster for this program, and by how much?

Answer

We know that each machine executes the same number of instructions for the program; let's call this number I. First, find the number of processor clock cycles for each machine:

$$\text{CPU clock cycles}_A = I \times 2.0$$

$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each machine:

$$\begin{aligned}\text{CPU time}_A &= \text{CPU clock cycles}_A \times \text{Clock cycle time}_A \\ &= I \times 2.0 \times 1 \text{ ns} = 2 \times I \text{ ns}\end{aligned}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 2 \text{ ns} = 2.4 \times I \text{ ns}$$

Clearly, machine A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{2.4 \times I \text{ ns}}{2 \times I \text{ ns}} = 1.2$$

We can conclude that machine A is 1.2 times faster than machine B for this program.

We can now write this basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

These formulas are particularly useful because they separate the three key factors that affect performance. We can use these formulas to compare two different implementations or to evaluate a design alternative if we know its impact on these three parameters.

The Big Picture

Figure 2.2 shows the basic measurements at different levels in the computer and what is being measured in each case. We can see how these factors are combined to yield execution time measured in seconds:

$$\text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Always bear in mind that the only complete and reliable measure of computer performance is time. For example, changing the instruction set to lower the instruction count may lead to an organization with a slower clock cycle time that offsets the improvement in instruction count. Similarly, because CPI depends on instruction mix, the code that executes the fewest number of instructions may not be the fastest.

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

FIGURE 2.2 The basic components of performance and how each is measured.

How can we determine the value of these factors in the performance equation? We can measure the CPU execution time by running the program, and the clock cycle time is usually published as part of the documentation for a machine. The instruction count and CPI can be more difficult to obtain. Of course, if we know the clock rate and CPU execution time, we need only one of the instruction count or the CPI to determine the other.

We can measure the instruction count by using software tools that profile the execution or by using a simulator of the architecture. Alternatively, we can use hardware counters, which have been included on some processors, to record a variety of measurements, including the number of instructions executed. Since the instruction count depends on the architecture, but not on the exact implementation, we can measure the instruction count without knowing all the details of the implementation. The CPI, however, depends on a wide variety of design details in the machine, including both the memory system and the processor structure (as we will see in Chapters 5, 6, and 7), as well as on the mix of instruction types executed in an application. Thus CPI varies by application, as well as among implementations with the same instruction set.

Designers often obtain CPI by a detailed simulation of an implementation or by combining hardware counters and simulation. Sometimes it is possible to compute the CPU clock cycles by looking at the different types of instructions and using their individual clock cycle counts. In such cases, the following formula is useful:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)$$

where C_i is the count of the number of instructions of class i executed, CPI_i is the average number of cycles per instruction for that instruction class, and n is the number of instruction classes. Remember that overall CPI for a program will depend on both the number of cycles for each instruction type and the frequency of each instruction type in the program execution.

Example

Comparing Code Segments

A compiler designer is trying to decide between two code sequences for a particular machine. The hardware designers have supplied the following facts:

Instruction class	CPI for this instruction class
A	1
B	2
C	3

For a particular high-level-language statement, the compiler writer is considering two code sequences that require the following instruction counts:

Code sequence	Instruction counts for instruction class		
	A	B	C
1	2	1	2
2	4	1	1

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

Answer

Sequence 1 executes 2 + 1 + 2 = 5 instructions. Sequence 2 executes 4 + 1 + 1 = 6 instructions. So sequence 1 executes fewer instructions.

We can use the equation for CPU clock cycles based on instruction count and CPI to find the total number of clock cycles for each sequence:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$

$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

So code sequence 2 is faster, even though it actually executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \frac{9}{6} = 1.5$$

The above example shows the danger of using only one factor (instruction count) to assess performance. When comparing two machines, you must look at all three components, which combine to form execution time. If some of the

factors are identical, like the clock rate in the above example, performance can be determined by comparing all the nonidentical factors. Since CPI varies by instruction mix, both instruction count and CPI must be compared, even if clock rates are identical. Exercises 2.18 through 2.24 explore this further by asking you to evaluate a series of machine and compiler enhancements that affect clock rate, CPI, and instruction count. In the next section, we'll examine a common performance measurement that does not incorporate all the terms and can thus be misleading.

2.4 Choosing Programs to Evaluate Performance

A computer user who runs the same programs day in and day out would be the perfect candidate to evaluate a new computer. The set of programs run would form a *workload*. To evaluate two computer systems, a user would simply compare the execution time of the workload on the two machines. Most users, however, are not in this situation. Instead, they must rely on other methods that measure the performance of a candidate machine, hoping that the methods will reflect how well the machine will perform with the user's workload. This alternative is usually followed by evaluating the machine using a set of *benchmarks*, which are programs specifically chosen to measure performance. The benchmarks form a workload that the user hopes will predict the performance of the actual workload.

Today, it is widely understood that the best type of programs to use for benchmarks are real applications. These may be applications that the user employs regularly or simply applications that are typical. For example, in an environment where the users are primarily engineers, you might use a set of benchmarks containing several typical engineering or scientific applications. If the user community were primarily software development engineers, the best benchmarks would probably include such applications as a compiler or document processing system. Using real applications as benchmarks makes it much more difficult to find trivial ways to speed up the execution of the benchmark. Furthermore, when techniques are found to improve performance, such techniques are much more likely to help other programs in addition to the benchmark.

The use of benchmarks whose performance depends on very small code segments encourages optimizations in either the architecture or compiler that target these segments. The compiler optimizations might recognize special code fragments and generate an instruction sequence that is particularly efficient for this code fragment. Likewise, a designer might try to make some sequence of instructions run especially fast because the sequence occurs in a benchmark. Recently, several companies have introduced compilers with

special-purpose optimizations targeted at specific benchmarks. Often these optimizations must be explicitly enabled with a specific compiler option, which would not be used when compiling other programs. Whether the compiler would produce good code, or even *correct* code, if a real application program used these switches, is unclear. Sometimes in the quest to produce highly optimized code for benchmarks, engineers introduce erroneous optimizations. For example, in late 1995, Intel published a new performance rating for the integer SPEC benchmarks (see sections 2.6 and 2.9 for a further discussion of SPEC) running on a Pentium processor and using an internal compiler, not used outside of Intel. Unfortunately, the code produced for one of the benchmarks was wrong, a fact that was discovered when a competitor read through the binary to understand how Intel had sped up one of the programs in the benchmark suite so dramatically. In January of 1996, Intel admitted the error and restated the performance.

Small programs or programs that spend almost all their execution time in a very small code fragment are especially vulnerable to such efforts. For example, the SPEC processor benchmark suite was chosen to use primarily real applications. Unfortunately, the first release of the SPEC suite in 1989 included a benchmark called matrix300, which consists solely of a series of matrix multiplications. In fact, 99% of the execution time is in a single line of this benchmark. The fact that so much time is spent in one line doing the same computation many times has led several companies to purchase or develop special compiler technology to improve the running time of this benchmark. Figure 2.3 shows the performance ratios (inverse to execution time) for one machine with two different compilers. The enhanced compiler has essentially no effect on the running time of 8 of the 10 benchmarks, but it improves performance on matrix300 by a factor of more than nine. On matrix300, the program runs 729.8 times faster using the enhanced compiler than the reference time obtained from a VAX-11/780—but the more typical performance of the machine is much slower. The other programs run from just over 30 times faster to just over 140 times faster. A user expecting a program to run 700 times faster than it does on a VAX-11/780 would likely be very disappointed! In the 1992 release of the SPEC benchmark suite, matrix300 was dropped.

So why doesn't everyone run real programs to measure performance? One reason is that small benchmarks are attractive when beginning a design, since they are small enough to compile and simulate easily, sometimes by hand. They are especially tempting when designers are working on a novel machine because compilers may not be available until much later in the design. Small benchmarks are also more easily standardized than large programs; hence numerous published performance results are available for small benchmarks.

Although the use of such small benchmarks early in the design process may be justified, there is no valid rationale for using them to evaluate working computer systems. In the past, it was hard to obtain large applications that could

4.1 Introduction

Computer words are composed of bits; thus words can be represented as binary numbers. Although the natural numbers 0, 1, 2, and so on can be represented either in decimal or binary form, what about the other numbers that commonly occur? For example:

- How are negative numbers represented?
- What is the largest number that can be represented in a computer word?
- What happens if an operation creates a number bigger than can be represented?
- What about fractions and real numbers?

We could also ask, What is the inside story about the infamous bug in the Pentium? And underlying all these questions is a mystery: How does hardware really add, subtract, multiply, or divide numbers?

The goal of this chapter is to unravel this mystery, including representation of numbers, arithmetic algorithms, hardware that follows these algorithms, and the implications of all this for instruction sets. These insights may even explain quirks that you have already encountered with computers. (If you are familiar with signed binary numbers, you may wish to skip the next section and go to section 4.3 on page 220.)

4.2 Signed and Unsigned Numbers

Numbers can be represented in any base; humans prefer base 10 and, as we examined in Chapter 3, base 2 is best for computers. Because we will frequently be dealing with both decimal and binary numbers, to avoid confusion we will subscript decimal numbers with *ten* and binary numbers with *two*.

In any number base, the value of *i*th digit d is

$$d \times \text{Base}^i$$

where i starts at 0 and increases from right to left. This leads to an obvious way to number the bits in the word: Simply use the power of the base for that bit. For example,

1011_{two}

represents

$$\begin{aligned}
&(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{ten} \\
&= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{ten} \\
&= 8 + 0 + 2 + 1_{ten} \\
&= 11_{ten}
\end{aligned}$$

Hence the bits are numbered 0, 1, 2, 3, . . . from *right to left* in a word. The drawing below shows the numbering of bits within a MIPS word and the placement of the number 1011_{two}:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

Since words are drawn vertically as well as horizontally, leftmost and rightmost may be unclear. Hence, the phrase *least significant bit* is used to refer to the rightmost bit (bit 0 above) and *most significant bit* to the leftmost bit (bit 31).

The MIPS word is 32 bits long, so we can represent 2^{32} different 32-bit patterns. It is natural to let these combinations represent the numbers from 0 to $2^{32} - 1$ ($4{,}294{,}967{,}295_{ten}$):

$$\begin{aligned}
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten} \\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten} \\
&\ldots \\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten} \\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten} \\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}
\end{aligned}$$

Hardware Software Interface

Base 2 is not natural to human beings; we have 10 fingers and so find base 10 natural. Why didn't computers use decimal? In fact, the first commercial computer *did* offer decimal arithmetic. The problem was that the computer still used on and off signals, so a decimal digit was simply represented by several binary digits. Decimal proved so inefficient that subsequent machines reverted to all binary, converting to base 10 only for the infrequent input/output events.

ASCII versus Binary Numbers

Example We could represent numbers as strings of ASCII digits instead of as two's complement integers (see Figure 3.15 on page 142). What is the expansion in storage if the number 1 billion is represented in ASCII versus a 32-bit integer?

Answer One billion is 1 000 000 000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be (10 × 8)/32 or 2.5. In addition to the expansion in storage, the hardware to add, subtract, multiply, and divide such numbers is also difficult. Such difficulties explain why computing professionals are raised to believe that binary is natural and that the occasional decimal machine is bizarre.

Keep in mind that the binary bit patterns above are simply *representatives* of numbers. Numbers really have an infinite number of digits, with almost all being 0 except for a few of the rightmost digits. We just don't normally show leading 0s.

As we shall see in sections 4.5 through 4.7, hardware can be designed to add, subtract, multiply, and divide these binary bit patterns. If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, *overflow* is said to have occurred. It's up to the operating system and program to determine what to do if overflow occurs.

Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative. The most obvious solution is to add a separate sign, which conveniently can be represented in a single bit; the name for this representation is *sign and magnitude*.

Alas, sign and magnitude representation has several shortcomings. First, it's not obvious where to put the sign bit. To the right? To the left? Early machines tried both. Second, adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be. Finally, a separate sign bit means that sign and magnitude has both a positive and negative zero, which can lead to problems for inattentive programmers. As a result of these shortcomings, sign and magnitude was soon abandoned.

In the search for a more attractive alternative, the question arose as to what would be the result for unsigned numbers if we tried to subtract a large number from a small one. The answer is that it would try to borrow from a string of leading 0s, so the result would have a string of leading 1s.

Given that there was no obvious better alternative, the final solution was to pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative. This convention for representing signed binary numbers is called *two's complement* representation:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
. . . . . .

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
. . . . . .

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

The positive half of the numbers, from 0 to $2{,}147{,}483{,}647_{ten}$ ($2^{31}-1$), use the same representation as before. The following bit pattern ($1000\ldots0000_{two}$) represents the most negative number $-2{,}147{,}483{,}648_{ten}$ (-2^{31}). It is followed by a declining set of negative numbers: $-2{,}147{,}483{,}647_{ten}$ ($1000\ldots0001_{two}$) down to -1_{ten} ($1111\ldots1111_{two}$).

Two's complement does have one negative number, $-2{,}147{,}483{,}648_{ten}$, that has no corresponding positive number. Such imbalance was a worry to the inattentive programmer, but sign and magnitude had problems for both the programmer *and* the hardware designer. Consequently, every computer today uses two's complement binary representations for signed numbers.

Two's complement representation has the advantage that all negative numbers have a 1 in the most significant bit. Consequently, hardware needs to test only this bit to see if a number is positive or negative (with 0 considered positive). This particular bit is often called the *sign bit*. By recognizing the role of the sign bit, we can represent positive and negative numbers in terms of the bit value times a power of 2 (here xi means the ith bit of x):

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

The sign bit is multiplied by -2^{31}, and the rest of the bits are then multiplied by positive versions of their respective base values.

Binary to Decimal Conversion

Example What is the decimal value of this 32-bit two's complement number?

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$

Answer Substituting the number's bit values into the formula above:

$$(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0})$$
$$= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0$$
$$= -2{,}147{,}483{,}648_{ten} + 2{,}147{,}483{,}644_{ten}$$
$$= -4_{ten}$$

We'll see a shortcut to simplify conversion soon.

Hardware Software Interface

Signed versus unsigned applies to loads as well as to arithmetic. The *function* of a signed load is to copy the sign repeatedly to fill the rest of the register—called *sign extension*—but its *purpose* is to place a correct representation of the number within that register. Unsigned loads simply fill with 0s to the left of the data, since the number represented by the bit pattern is unsigned.

When loading a 32-bit word into a 32-bit register, the point is moot; signed and unsigned loads are identical. MIPS does offer two flavors of byte loads: *load byte* (`lb`) treats the byte as a signed number and thus sign extends to fill the 24 leftmost bits of the register, while *load byte unsigned* (`lbu`) works with unsigned integers. Since programs almost always use bytes to represent characters rather than consider bytes as short signed integers, `lbu` is used practically exclusively for byte loads.

Just as an operation on unsigned numbers can overflow the capacity of hardware to represent the result, so can an operation on two's complement numbers. Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect): a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive.

Hardware Software Interface

Unlike the numbers discussed above, memory addresses naturally start at 0 and continue to the largest address. Put another way, negative addresses make no sense. Thus, programs want to deal sometimes with numbers that can be positive or negative and sometimes with numbers that can be only positive. Programming languages reflect this distinction. C, for example, names the former *integers* (declared as `int` in the program) and the latter *unsigned integers* (`unsigned int`).

Comparison instructions must deal with this dichotomy. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0.

MIPS offers two versions of the set on less than comparison to handle these alternatives. *Set on less than* (`slt`) and *set on less than immediate* (`slti`) work with signed integers. Unsigned integers are compared using *set on less than unsigned* (`sltu`) and *set on less than immediate unsigned* (`sltiu`).

Signed versus Unsigned Comparison

Example Suppose register `$s0` has the binary number

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

and that register `$s1` has the binary number

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$

What are the values of registers `$t0` and `$t1` after these two instructions?

```
slt     $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

Answer The value in register `$s0` represents –1 if it is an integer and $4{,}294{,}967{,}295_{ten}$ if it is an unsigned integer. The value in register `$s1` represents 1 in either case. Then register `$t0` has the value 1, since $-1_{ten} < 1_{ten}$, and register `$t1` has the value 0, since $4{,}294{,}967{,}295_{ten} > 1_{ten}$.

Before going on to addition and subtraction, let's examine a few useful shortcuts when working with two's complement numbers.

The first shortcut is a quick way to negate a two's complement binary number. Simply invert every 0 to 1 and every 1 to 0, then add one to the result. This shortcut is based on the observation that the sum of a number and its inverted representation must be $111\ldots111_{two}$, which represents –1. Since $x + \bar{x} \equiv -1$, therefore $x + \bar{x} + 1 = 0$ or $\bar{x} + 1 = -x$.

Negation Shortcut

Example Negate 2_{ten}, and then check the result by negating -2_{ten}.

Answer

$$2_{ten} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$$

Negating this number by inverting the bits and adding one,

$$\begin{array}{rl} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} \\ = & -2_{ten} \end{array}$$

Going the other direction,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

is first inverted and then incremented:

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} \\ = & 2_{ten} \end{array}$$

The second shortcut tells us how to convert a binary number represented in n bits to a number represented with more than n bits. For example, the immediate field in the load, store, branch, add, and set on less than instructions contains a two's complement 16-bit number, representing $-32{,}768_{ten}$ (-2^{15}) to $32{,}767_{ten}$(2^{15}–1). To add the immediate field to a 32-bit register, the machine must convert that 16-bit number to its 32-bit equivalent. The shortcut is to take the most significant bit from the smaller quantity—the sign bit—and replicate it to fill the new bits of the larger quantity. The old bits are simply copied into the right portion of the new word. This shortcut is commonly called *sign extension*.

Sign Extension Shortcut

Example Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 32-bit binary numbers.

Answer The 16-bit binary version of the number 2 is

$$0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

It is converted to a 32-bit number.by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word. The right half gets the old value:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

Let's negate the 16-bit version of 2 using the earlier shortcut. Thus,

$$0000\ 0000\ 0000\ 0010_{two}$$

becomes

$$\begin{array}{rl} & 1111\ 1111\ 1111\ 1101_{two} \\ + & \qquad\qquad\qquad 1_{two} \\ \hline = & 1111\ 1111\ 1111\ 1110_{two} \end{array}$$

Creating a 32-bit version of the negative number means copying the sign bit 16 times and placing it on the left:

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$$

This trick works because positive two's complement numbers really have an infinite number of 0s on the left and those that are negative two's complement numbers have an infinite number of 1s. The binary bit pattern representing a number hides leading bits to fit the width of the hardware; sign extension simply restores some of them.

A final shortcut, which we previewed in Chapter 3, is that we can save reading and writing long binary numbers by using a higher base than binary that converts easily into binary. Since almost all computer data sizes are multiples of 4, *hexadecimal* (base 16) numbers are popular. Since base 16 is a power of 2, we can trivially convert by replacing each group of four binary digits by a single hexadecimal digit, and vice versa. Figure 4.1 shows the hexadecimal Rosetta stone. We will use either the subscript *hex* or the C notation, which uses 0x*nnnn*, for hexadecimal numbers.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

FIGURE 4.1 The hexadecimal-binary conversion table. Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of four, go from right to left.

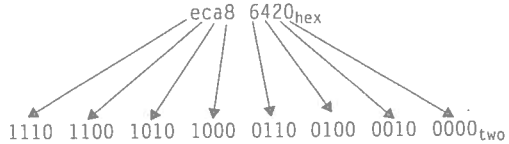
Binary-to-Hexadecimal Shortcut

Example Convert the following hexadecimal and binary numbers into the other base:

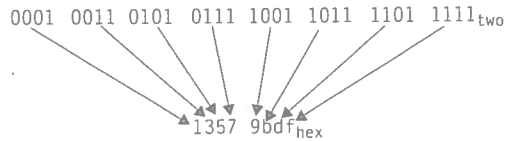
$$eca8\ 6420_{hex}$$

$$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_{two}$$

Answer Just a table lookup one way:



And then the other direction:



Summary

The main point of this section is that we need to represent both positive and negative integers within a computer word, and although there are pros and cons to any option, the overwhelming choice since 1965 has been two's complement. Figure 4.2 shows the additions to the MIPS assembly language revealed in this section. (The MIPS machine language is also illustrated on the back endpapers of this book.)

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $gp, $fp, $zero, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. Register `$at` is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three operands
	add immediate	`addi $s1,$s2,100`	$s1 = $s2 + 100	+ constant
Data transfer	load word	`lw $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1,100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte unsigned	`lbu $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1,100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1,100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	`slti $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; two's complement
	set less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; unsigned numbers
	set less than immediate unsigned	`sltiu $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; unsigned numbers
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

FIGURE 4.2 MIPS architecture revealed thus far. Color indicates portions from this section added to the MIPS architecture revealed in Chapter 3 (Figure 3.20 on page 155). MIPS machine language is listed in the back endpapers of this book.

Elaboration: Two's complement gets its name from the rule that the unsigned sum of an n-bit number and its negative is 2^n, hence the complement or negation of a two's complement number x is $2^n - x$.

A third alternative representation is called *one's complement*. The negative of a one's complement is found by inverting each bit, from 0 to 1 and from 1 to 0, which helps explain its name since the complement of x is $2^n - x - 1$. It was also an attempt

to be a better solution than sign and magnitude, and several scientific computers did use the notation. This representation is similar to two's complement except that it also has two 0s: $00\ldots00_{two}$ is positive 0 and $11\ldots11_{two}$ is negative 0. The most negative number $10\ldots000_{two}$ represents $-2{,}147{,}483{,}647_{ten}$, and so the positives and negatives are balanced. One's complement adders did need an extra step to subtract a number, and hence two's complement dominates today.

A final notation, which we will look at when we discuss floating point, is to represent the most negative value by $00\ldots000_{two}$ and the most positive value represented by $11\ldots11_{two}$, with 0 typically having the value $10\ldots00_{two}$. This is called a *biased* notation, for it biases the number such that the number plus the bias has a non-negative representation.

4.3 Addition and Subtraction

Subtraction: Addition's Tricky Pal

No. 10, Top Ten Courses for Athletes at a Football Factory,
David Letterman et al., *Book of Top Ten Lists*, 1990

Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: The appropriate operand is simply negated before being added.

Binary Addition and Subtraction

Example Let's try adding 6_{ten} to 7_{ten} in binary and then subtracting 6_{ten} from 7_{ten} in binary.

Answer

$$\begin{array}{rll} & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0111_{two} & = 7_{ten} \\ + & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0110_{two} & = 6_{ten} \\ \hline = & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,1101_{two} & = 13_{ten} \end{array}$$

The 4 bits to the right have all the action; Figure 4.3 shows the sums and carries. The carries are shown in parentheses, with the arrows showing how they are passed.

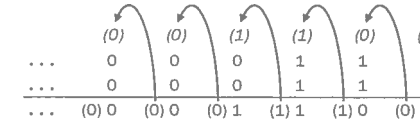


FIGURE 4.3 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$\begin{array}{rll} & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0111_{two} & = 7_{ten} \\ - & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0110_{two} & = 6_{ten} \\ \hline = & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{two} & = 1_{ten} \end{array}$$

or via addition using the two's complement representation of −6:

$$\begin{array}{rll} & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0111_{two} & = 7_{ten} \\ + & 1111\,1111\,1111\,1111\,1111\,1111\,1111\,1010_{two} & = -6_{ten} \\ \hline = & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{two} & = 1_{ten} \end{array}$$

We said earlier that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word. When can overflow occur in addition? When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, −10 + 4 = −6. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore no overflow can occur when adding positive and negative operands.

There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: When the signs of the operands are the *same*, overflow cannot occur. To see this, remember that $x - y = x + (-y)$ because we subtract by negating the second operand and then add. So, when we subtract operands of the same sign we end up by *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot occur in this case either.

Having examined when overflow cannot occur in addition and subtraction, we still haven't answered how to detect when it does occur. Overflow occurs when adding two positive numbers and the sum is negative, or vice versa. Clearly, adding or subtracting two 32-bit numbers can yield a result that needs

33 bits to be fully expressed. The lack of a 33rd bit means that when overflow occurs the sign bit is being set with the *value* of the result instead of the proper sign of the result. Since we need just one extra bit, only the sign bit can be wrong. This means a carry out occurred into the sign bit.

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. This means a borrow occurred from the sign bit. Figure 4.4 shows the combination of operations, operands, and results that indicate an overflow. (Exercise 4.42 gives a shortcut for detecting overflow more simply in hardware.)

We have just seen how to detect overflow for two's complement numbers in a machine. What about unsigned integers? Unsigned integers are commonly used for memory addresses where overflows are ignored.

The machine designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

- Add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow.
- Add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do *not* cause exceptions on overflow.

Because C ignores overflows, the MIPS C compilers will always generate the unsigned versions of the arithmetic instructions `addu`, `addiu`, and `subu` no matter what the type of the variables. The MIPS Fortran compilers, however, pick the appropriate arithmetic instructions, depending on the type of the operands.

Operation	Operand A	Operand B	Result indicating overflow
$A+B$	≥ 0	≥ 0	< 0
$A+B$	< 0	< 0	≥ 0
$A-B$	≥ 0	< 0	< 0
$A-B$	< 0	≥ 0	≥ 0

FIGURE 4.4 Overflow conditions for addition and subtraction.

Hardware Software Interface

The machine designer must decide how to handle arithmetic overflows. Although some languages like C leave the decision up to the machine designer, languages like Ada and Fortran require that the program be notified. The programmer or the programming environment must then decide what to do when overflow occurs.

MIPS detects overflow with an *exception*, also called an *interrupt* on many computers. An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed. (Section 5.6 covers exceptions in more detail; Chapters 7 and 8 describe other situations where exceptions and interrupts occur.)

MIPS includes a register called the *exception program counter* (*EPC*) to contain the address of the instruction that caused the exception. The instruction *move from system control* (`mfc0`) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the offending instruction via a jump register instruction.

Summary

The main point of this section is that, independent of the representation, the finite word size of computers means that arithmetic operations can create results that are too large to fit in this fixed word size. It's easy to detect overflow in unsigned numbers, although these are almost always ignored because programs don't want to detect overflow for address arithmetic, the most common use of natural numbers. Two's complement presents a greater challenge, yet some software systems require detection of overflow, so today all machines have a way to detect it. Figure 4.5 shows the additions to the MIPS architecture from this section.

Elaboration: MIPS can trap on overflow, but unlike many other machines there is no conditional branch to test overflow. A sequence of MIPS instructions can discover overflow. For signed addition, the sequence is the following (see the In More Depth section on page 329 for the definition of the `xor` and `nor` instructions):

```
addu $t0, $t1,    $t2          # $t0 = sum, but don't trap
xor  $t3, $t1,    $t2          # Check if signs differ
slt  $t3, $t3,    $zero        # $t3 = 1 if signs differ
bne  $t3, $zero,  No_overflow  # $t1, $t2 signs ≠, so no overflow
xor  $t3, $t0,    $t1          # signs =; sign of sum match too?
                               # $t3 negative if sum sign different
slt  $t3, $t3,    $zero        # $t3 = 1 if sum sign different
bne  $t3, $zero,  Overflow     # All three signs ≠; go to overflow
```

MIPS operands

Name	Example	Comments
32 registers	$s0–$s7, $t0–$t9, $gp, $fp, $zero, $sp, $ra, $at, .	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants. Hi and Lo contain the results of multiply and divide.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example		Meaning	Comments
Arithmetic	add	add	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0	$s1,$epc	$s1 = $epc	Used to copy Exception PC plus other special registers
	multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide		$ 2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned		$ 2,$ 3	Lo = $s2 / $s3, Hi = $s2 mod $ 3	Unsigned quotient and remainder
	move from Hi		$ 1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo
Logical	and	and	$s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; logical AND
	or	or	$s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; logical OR
	and immediate	andi	$s1,$s2,100	$s1 = $s2 & 100	Logical AND reg, constant
	or immediate	ori	$s1,$s2,100	$s1 = $s2 | 100	Logical OR reg, constant
	shift left logical	sll	$s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl	$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Data transfer	load word	lw	$s1,100($s2)	$s1 = Memory[$s2+100]	Word from memory to register
	store word	sw	$s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte unsigned	lbu	$s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb	$s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui	$s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq	$s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne	$s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	slti	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1=0	Compare < constant; two's complement
	set less than unsigned	sltu	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1=0	Compare less than; natural numbers
	set less than immediate unsigned	sltiu	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; natural numbers
Unconditional jump	jump	j	2500	go to 10000	Jump to target address
	jump register	jr	$ra	go to $ra	For switch, procedure return
	jump and link	jal	2500	$ra = PC + 4; go to 10000	For procedure call

FIGURE 4.43 MIPS architecture revealed thus far. Color indicates the portions revealed since Figure 4.7 on page 228. MIPS machine language is listed on the back endpapers of this book. *(page 274)*

4.8 Floating Point

Speed gets you nowhere if you're headed the wrong way.

American proverb

In addition to signed and unsigned integers, programming languages support numbers with fractions, which are called *reals* in mathematics. Here are some examples of reals:

$3.14159265\ldots_{ten}$ (π)

$2.71828\ldots_{ten}$ (e)

0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$ (seconds in a nanosecond)

$3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$ (seconds in a typical century)

Notice that in the last case, the number didn't represent a small fraction, but it was bigger than we could represent with a 32-bit signed integer. The alternative notation for the last two numbers is called *scientific notation*, which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a *normalized* number, which is the usual way to write it. For example, $1.0_{ten} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{ten} \times 10^{-8}$ and $10.0_{ten} \times 10^{-10}$ are not.

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{two} \times 2^{-1}$$

To keep a binary number in normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. Only a base of 2 fulfills our need. Since the base is not 10, we also need a new name for decimal point; *binary point* will do fine.

Computer arithmetic that supports such numbers is called *floating point* because it represents numbers in which the binary point is not fixed, as it is for integers. The programming language C uses the name *float* for such numbers. Just as in scientific notation, numbers are represented as a single nonzero digit to the left of the binary point. In binary, the form is

$$1.xxxxxxxxx_{two} \times 2^{yyyy}$$

(Although the computer represents the exponent in base 2 as well as the rest of the number, to simplify the notation we'll show the exponent in decimal.)

A standard scientific notation for reals in normalized form offers three advantages. It simplifies exchange of data that includes floating-point numbers; it simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form; and it increases the accuracy of the numbers that can be stored in a word, since the unnecessary leading 0s are replaced by real digits to the right of the binary point.

Floating-Point Representation

The designer of a floating-point representation must find a compromise between the size of the significand and the size of the exponent because a fixed word size means you must take a bit from one to add a bit to the other. This trade-off is between accuracy and range: Increasing the size of the significand enhances the accuracy of the significand, while increasing the size of the exponent increases the range of numbers that can be represented. As our design guideline from Chapter 3 reminds us, good design demands good compromises.

Floating-point numbers are usually a multiple of the size of a word. The representation of a MIPS floating-point number is shown below, where *s* is the sign of the floating-point number (1 meaning negative), *exponent* is the value of the 8-bit exponent field (including the sign of the exponent), and *significand* is the 23-bit number in the fraction. This representation is called *sign and magnitude*, since the sign has a separate bit from the rest of the number.

In general, floating-point numbers are of the form

$$(-1)^S \times F \times 2^E$$

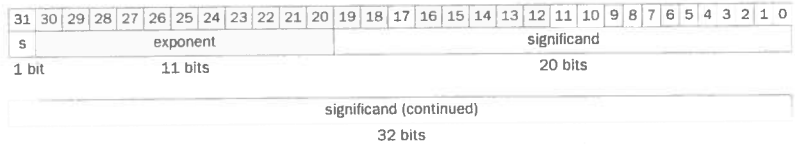
F involves the value in the significand field and E involves the value in the exponent field; the exact relationship to these fields will be spelled out soon.

These chosen sizes of exponent and significand give MIPS computer arithmetic an extraordinary range. Fractions as small as $2.0_{ten} \times 10^{-38}$ and numbers as large as $2.0_{ten} \times 10^{38}$ can be represented in a computer. Alas, extraordinary differs from infinite, so it is still possible for numbers to be too large. Thus, overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic. Notice that *overflow* here means that the exponent is too large to be represented in the exponent field.

Floating point offers a new kind of exceptional event as well. Just as programmers will want to know when they have calculated a number that is too large to be represented, they will want to know if the nonzero fraction they are calculating has become so small that it cannot be represented; either event could result in a program giving incorrect answers. This situation occurs when the negative exponent is too large to fit in the exponent field. To distinguish it from overflow, people call this event *underflow*.

One way to reduce chances of underflow or overflow is to use a notation that has a larger exponent. In C this is called *double*, and operations on doubles are called *double precision* floating-point arithmetic; *single precision* floating point is the name of the earlier format.

The representation of a double precision floating-point number takes two MIPS words, as shown below, where *s* is still the sign of the number, *exponent* is the value of the 11-bit exponent field, and *significand* is the 52-bit number in the fraction.



MIPS double precision allows numbers almost as small as $2.0_{ten} \times 10^{-308}$ and almost as large as $2.0_{ten} \times 10^{308}$. Although double precision does increase the exponent range, its primary advantage is its greater accuracy because of the large significand.

These formats go beyond MIPS. They are part of the *IEEE 754 floating-point standard*, found in virtually every computer invented since 1980. This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.

To pack even more bits into the significand, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit. Hence, the significand is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1+52). Since 0 has no leading 1, it is given the reserved exponent value 0 so that the hardware won't attach a leading 1 to it.

Thus $00\ldots00_{two}$ represents 0; the representation of the rest of the numbers uses the form from before with the hidden 1 added:

$$(-1)^S \times (1 + \text{Significand}) \times 2^E$$

where the bits of the significand represent the fraction between 0 and 1 and E specifies the value in the exponent field, to be given in detail shortly. If we

number the bits of the significand from *left to right* s1, s2, s3, . . . , then the value is

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots) \times 2^E$$

The designers of IEEE 754 also wanted a floating-point representation that could be easily processed by integer comparisons, especially for sorting. This desire is why the sign is in the most significant bit, allowing a test of less than, greater than, or equal to 0 to be performed quickly.

Placing the exponent before the significand also simplifies sorting of floating-point numbers using integer comparison instructions, since numbers with bigger exponents look larger than numbers with smaller exponents, as long as both exponents have the same sign. (It's a little more complicated than a simple integer sort, since this notation is essentially sign and magnitude rather than two's complement.)

Negative exponents pose a challenge to simplified sorting. If we use two's complement or any other notation in which negative exponents have a 1 in the most significant bit of the exponent field, a negative exponent will look like a big number. For example, $1.0_{two} \times 2^{-1}$ would be represented as

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

(Remember that the leading 1 is implicit in the significand.) The value $1.0_{two} \times 2^{+1}$ would look like the smaller binary number

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The desirable notation must therefore represent the most negative exponent as $00 \ldots 00_{two}$ and the most positive as $11 \ldots 11_{two}$. This convention is called *biased notation*, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.

IEEE 754 uses a bias of 127 for single precision, so –1 is represented by the bit pattern of the value $-1 + 127_{ten}$, or $126_{ten} = 0111\ 1110_{two}$, and +1 is represented by 1 + 127, or $128_{ten} = 1000\ 0000_{two}$. Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - \text{Bias})}$$

The exponent bias for double precision is 1023.

Thus IEEE 754 notation can be processed by integer compares to accelerate sorting of floating-point numbers. Let's show the representation.

Floating-Point Representation

Example Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

Answer The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2{}_{ten}$$

It is also represented by the binary fraction:

$$-11_{two}/2^2{}_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - 127)}$$

and so when we add the bias 127 to the exponent of $-1.1_{two} \times 2^{-1}$, the result is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126 - 127)}$$

The single precision binary representation of -0.75_{ten} is then

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(1022 - 1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

6.1 An Overview of Pipelining

Never waste time.

American proverb

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is key to making processors fast.

This section relies heavily on one analogy to give an overview of the pipelining terms and issues. If you are interested in just the big picture, you should concentrate on this section and then skip to section 6.8 to see how pipelining works and its implications on program performance. If you are interested in exploring the anatomy of a pipelined computer, you will find this section referred to repeatedly in sections 6.2 through 6.7.

Anyone who has done a lot of laundry has intuitively used pipelining. The *nonpipelined* approach to laundry would be:

1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

When your roommate is done, then start over with the next dirty load.

The *pipelined* approach takes much less time, as Figure 6.1 shows. As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load. When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and the next dirty load into the washer. Next you have your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer. At this point all steps—called *stages* in pipelining—are operating concurrently. As long as we have separate resources for each stage, we can pipeline the tasks.

The pipelining paradox is that the time from placing a single dirty sock in the washer until it is dried, folded, and put away is not shorter for pipelining; the reason pipelining is faster for many loads is that everything is working in parallel, so more loads are finished per hour. If all the stages take about the same amount of time and there is enough work to do, then the speedup due to pipelining is equal to the number of stages in the pipeline.

Pipelined laundry is potentially four times faster than nonpipelined: 20 loads would take about 5 times as long as 1 load, while 20 loads of sequential laundry takes 20 times as long as 1 load. It's only 2.3 times faster in Figure 6.1 because we only show 4 loads.

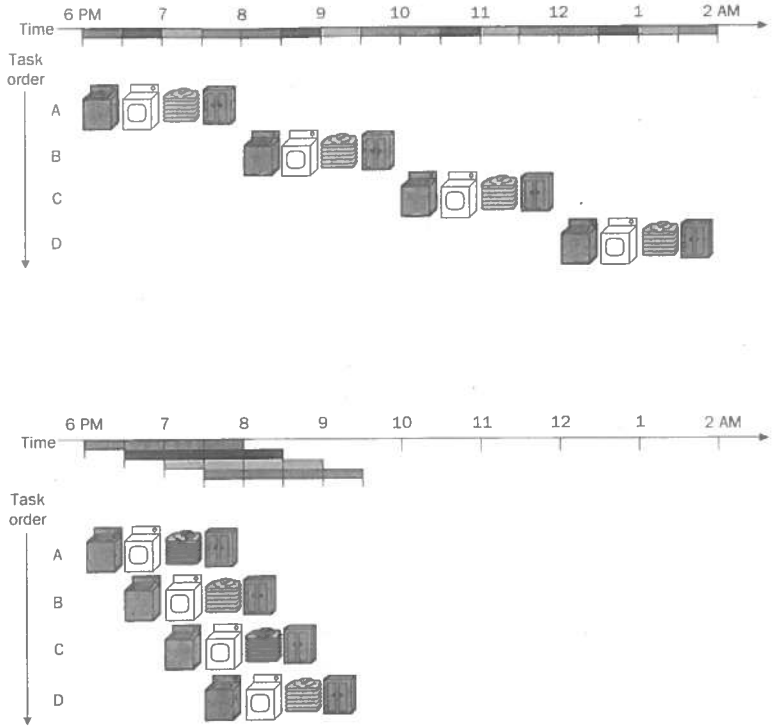


FIGURE 6.1 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for four loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional timeline, but we really have just one of each resource.

The same principles apply to processors where we pipeline instruction execution. MIPS instructions classically take five steps:

1. Fetch instruction from memory.
2. Read registers while decoding the instruction (the format of MIPS instructions allows reading and decoding to occur simultaneously).

3. Execute the operation or calculate an address.
4. Access an operand in data memory.
5. Write the result into a register.

Hence the MIPS pipeline we explore in this chapter has five stages. The following example shows that pipelining speeds up instruction execution just as it speeds up the laundry.

Single-Cycle versus Pipelined Performance

Example

To make this discussion concrete, let's create a pipeline. In this example, and in the rest of this chapter, we limit our attention to eight instructions: load word (`lw`), store word (`sw`), add (`add`), subtract (`sub`), and (`and`), or (`or`), set-less-than (`slt`), and branch-on-equal (`beq`).

Compare the average time between instructions of a single-cycle implementation, in which all instructions take 1 clock cycle, to a pipelined implementation. The operation times for the major functional units in this example are 2 ns for memory access, 2 ns for ALU operation, and 1 ns for register file read or write. (As we said in Chapter 5, in the single-cycle model every instruction takes exactly 1 clock cycle, so the clock cycle must be stretched to accommodate the slowest instruction.)

Answer

The time required for each of the eight instructions is shown in Figure 6.2. The single-cycle design must allow for the slowest instruction— in Figure 6.2 it is `lw`—so the time required for every instruction is 8 ns. Similarly to Figure 6.1, Figure 6.3 compares nonpipelined and pipelined execution of three load word instructions. Thus, the time between the first and fourth instructions in the nonpipelined design is 3×8 ns or 24 ns.

All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation. Just as the single-cycle design must take the worst-case clock cycle of 8 ns even though some instructions can be as fast as 5 ns, the pipelined execution clock cycle must have the worst-case clock cycle of 2 ns even though some stages take only 1 ns. Pipelining still offers a fourfold performance improvement: the time between the first and fourth instructions is 3×2 ns or 6 ns.

We can turn the pipelining speedup discussion above into a formula. If the stages are perfectly balanced, then the time between instructions on the pipelined machine—assuming ideal conditions—is equal to

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
Store word (`sw`)	2 ns	1 ns	2 ns	2 ns		7 ns
R-format (`add, sub, and, or, slt`)	2 ns	1 ns	2 ns		1 ns	6 ns
Branch (`beq`)	2 ns	1 ns	2 ns			5 ns

FIGURE 6.2 Total time for eight instructions calculated from the time for each component. This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

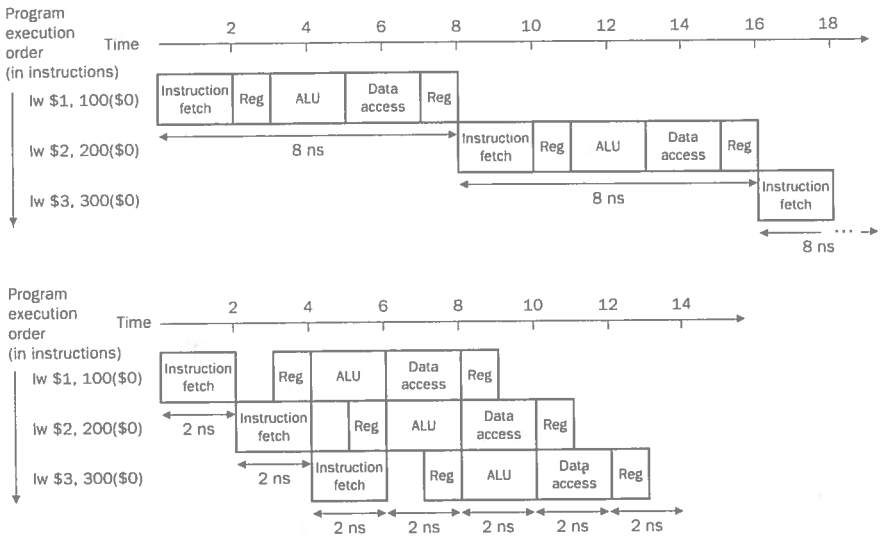


FIGURE 6.3 Single-cycle, nonpipelined execution in top vs. pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 6.2. In this case we see a fourfold speedup on average time between instructions, from 8 ns down to 2 ns. Compare this figure to Figure 6.1. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The computer pipeline stage times are limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

Under ideal conditions, the speedup from pipelining equals the number of pipe stages; a five-stage pipeline is five times faster.

The formula suggests that a five-stage pipeline should offer a fivefold improvement over the 8 ns nonpipelined time, or a 1.6-ns clock cycle. The example shows, however, that the stages may be imperfectly balanced. In addition, pipelining involves some overhead. Thus the time per instruction in the pipelined machine will exceed the minimum possible, and speedup will be less than the number of pipeline stages.

Moreover, even our claim of fourfold improvement for our example is not reflected in the total execution time for the three instructions: it's 14 ns versus 24 ns. To see why total execution time is less important, what would happen if we increased the number of instructions? We start by extending the previous figures to 1003 instructions. We would add 1000 instructions in the pipelined example; each instruction adds 2 ns to the total execution time. The total execution time would be 1000 × 2 ns + 14 ns, or 2,014 ns. In the nonpipelined example, we would add 1000 instructions, each taking 8 ns, so total execution time would be 1000 × 8 ns + 24 ns, or 8,024 ns. Under these ideal conditions, the ratio of total execution times for real programs on nonpipelined to pipelined machines is close to the ratio of times between instructions:

$$\frac{8{,}024 \text{ ns}}{2{,}014 \text{ ns}} = 3.98 \approx \frac{8 \text{ ns}}{2 \text{ ns}}$$

Pipelining improves performance by *increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction*, but instruction throughput is the important metric because real programs execute billions of instructions.

Designing Instruction Sets for Pipelining

Even with this simple explanation of pipelining, we can get insight into the design of the MIPS instruction set, which was designed for pipelined execution.

First, all MIPS instructions are the same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage. In an instruction set like the 80x86, where instructions vary from 1 byte to 17 bytes, pipelining is considerably more challenging.

Second, MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction. This symmetry means that the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched. If MIPS instruction formats were not symmetric, we would need to split stage 2, resulting in six pipeline stages. (We will shortly see the downside of longer pipelines.)

Third, memory operands only appear in loads or stores in MIPS. This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage. If we could operate on the operands in memory, as in the 80x86, stages 3 and 4 would expand to an address stage, memory stage, and then execute stage.

Fourth, operands must be aligned in memory (see the Hardware/Software Interface section on page 112 in Chapter 3). Hence we need not worry about a single data transfer instruction requiring two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage.

Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*. We explain the three types of hazards, using our analogy first, and then give the computer equivalent problem and solution.

Structural Hazards

The first hazard is called a *structural hazard*. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle. A structural hazard in the laundry room would occur if we used a washer-dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

As we said above, the MIPS instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline. Suppose, however, that we had a single memory instead of two memories. If the pipeline in Figure 6.3 had a fourth instruction, we would see that in the same clock cycle that the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipeline could have a structural hazard.

Control Hazards

The second hazard is called a *control hazard*, arising from the need to make a decision based on the results of one instruction while others are executing.

Suppose our laundry crew was given the happy task of cleaning the uniforms of a football team. Given how filthy the laundry is, we need to determine whether the detergent and water temperature setting we select is strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner. In our laundry pipeline, we have to wait until the second stage to examine the dry uniform to see if we need to change the washer setup or not. What to do?

Here are two solutions to control hazards in the laundry room and two computer equivalents.

> *Stall*: Just operate sequentially until the first batch is dry and then repeat until you have the right formula. This conservative option certainly works, but it is slow.

The equivalent decision task in a computer is the branch instruction. If the computer were to stall on a branch, then it would have to pause before continuing the pipeline. Let's assume that we put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage (see section 6.6 for details). Even with this extra hardware, the pipeline involving conditional branches would look like Figure 6.4. The `lw` instruction, executed if the branch fails, is stalled one extra 2-ns clock cycle before starting. This figure shows an important pipeline concept, officially called a *pipeline stall*, but often given the nickname *bubble*. We shall see stalls elsewhere in the pipeline.

Example

Stall on Branch Performance

Estimate the impact on the clock cycles per instruction (CPI) of stalling on branches. Assume all other instructions have a CPI of 1.

Answer

Figure 3.38 on page 189 in Chapter 3 shows conditional branches being 17% of the instructions executed for gcc. Since other instructions run have a CPI of 1 and branches took one extra clock cycle for the stall, then we would see a CPI of 1.17 and hence a slowdown of 1.17 versus the ideal case. (Since Figure 3.38 includes `slt` and `slti` as branch instructions, and they would not stall, this CPI result is appproximate.)

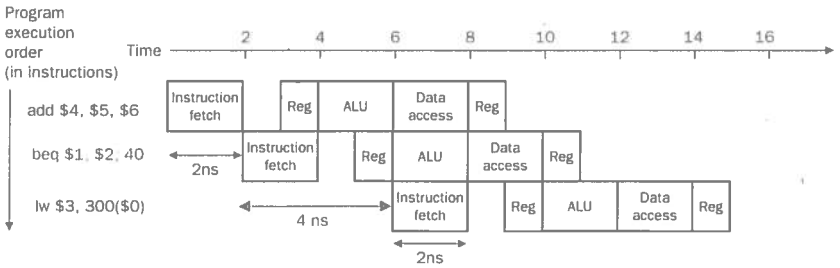


FIGURE 6.4 Pipeline showing stalling on every conditional branch as solution to control hazards. There is a one-stage pipeline stall, or bubble, after the branch.

If we cannot resolve the branch in the second stage, as is often the case for longer pipelines, then we'd see an even larger slowdown if we stall on branches. The cost of this option is too high for most computers to use and motivates a second solution to the control hazard:

> *Predict*: If you're pretty sure you have the right formula to wash uniforms, then just predict that it will work and wash the second load while waiting for the first load to dry. This option does not slow down the pipeline when you are correct. When you are wrong, however, you need to redo the load that was washed while guessing the decision.

Computers do indeed use prediction to handle branches. One simple approach is to always predict that branches will fail. When you're right, the pipeline proceeds at full speed. Only when branches succeed does the pipeline stall. Figure 6.5 shows such an example.

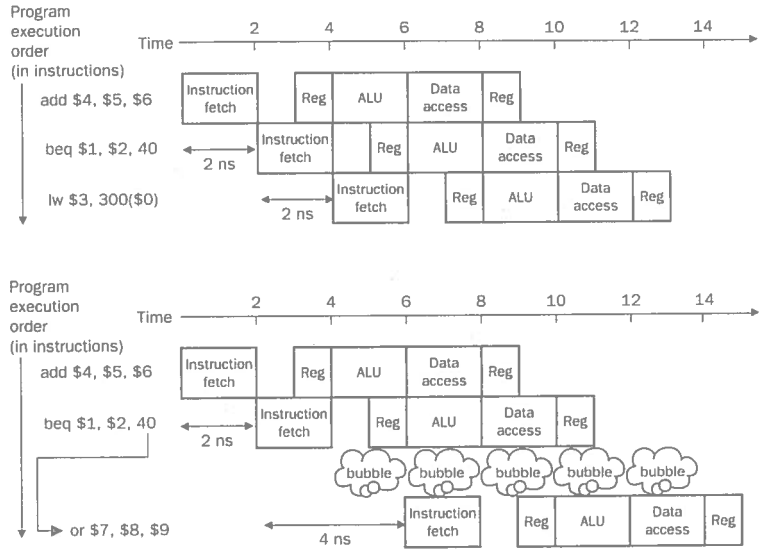


FIGURE 6.5 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows a taken branch.

A more sophisticated version of branch prediction would have some predicted as branching (*taken*) and some not branching (*untaken*). In our analogy, the dark or home uniforms might take one formula while the light or road uniforms might take another. As a computer example, at the bottom of loops are branches that jump back to the top of the loop. Since they are likely to be taken and they branch backwards, we could always predict taken for branches that jump to an earlier address.

Such rigid approaches to branch prediction rely on stereotypical behavior and don't account for the individuality of a specific branch instruction. *Dynamic* hardware predictors, in stark contrast, make their guesses depending on the behavior of each branch and may change predictions for a branch over the life of a program. Following our analogy, in dynamic prediction a person would look at how dirty the uniform was and guess at the formula, adjusting the next guess depending on the success of recent guesses. One popular approach to dynamic prediction in computers is keeping a history for each branch as taken or untaken, and then using the past to predict the future. Such hardware has about a 90% accuracy (see section 6.6). When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed branch have no effect and must restart the pipeline from the proper branch address.

As in the case of all other solutions to control hazards, longer pipelines exacerbate the problem, in this case by raising the cost of misprediction. Solutions to control hazards are described in more detail in section 6.6.

Elaboration: There is a third approach to the control hazard, called *delayed decision*. In our analogy, whenever you are going to make such a decision about laundry, just place a load of nonfootball clothes in the washer while waiting for football uniforms to dry. As long as you have enough dirty clothes that are not affected by the test, this solution works fine.

Called the *delayed branch* in computers, this is the solution actually used by the MIPS architecture. The delayed branch always executes the next sequential instruction, with the branch taking place *after* that one instruction delay. It is hidden from the MIPS assembly language programmer because the assembler can automatically arrange the instructions to get the branch behavior desired by the programmer. MIPS software will place an instruction immediately after the delayed branch instruction that is not affected by the branch, and a taken branch changes the address of the instruction that *follows* this safe instruction. In our example, the `add` instruction before the branch in Figure 6.4 does not affect the branch, so in Figure 6.6 we move it to the *delayed branch slot* following the branch.

Compilers typically fill about 50% of the branch delay slots with useful instructions. If the pipeline is longer than five stages, then we may get more branch delay slots, which are even harder to fill.

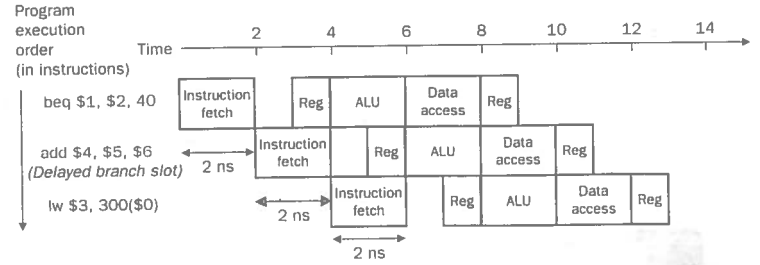


FIGURE 6.6 Pipeline delayed branch as solution to control hazard. The pipe bubble has been replaced by add.

Data Hazards

Returning to the laundry room, suppose that you are folding a load that is mostly socks. You realize that through bad luck the mate of every sock in this load is in another load that is still in the washer. You can't match the socks and put them away until that load is done. Hence you must stall the pipeline. This problem in computers is called a *data hazard*: an instruction depends on the results of a previous instruction still in the pipeline.

For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum (`$s0`):

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to add three bubbles to the pipeline.

Although we could try to rely on compilers to avoid such data hazards, we would fail. These dependencies happen just too often and the delay is just too long to expect the compiler to rescue us from this dilemma.

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Getting the missing item early from the internal resources is called *forwarding* or *bypassing*.

Forwarding with Two Instructions

Example For the two instructions above, show what pipeline stages would be connected by forwarding. Use the drawing in Figure 6.7 to represent the datapath during the five stages of the pipeline. Align a copy of the datapath for each instruction, similar to the laundry pipeline in Figure 6.1.

Answer Figure 6.8 shows the connection to forward the value in `$s0` after the execution stage of the `add` instruction as input to the execution stage of the `sub` instruction.

In this graphical representation of events, forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following, since that would mean going backwards in time.

Forwarding works very well, and is described in detail in section 6.4. It cannot prevent all pipeline stalls, however. For example, suppose the first instruction were a load of `$s0` instead of an add. As we can imagine from looking at Figure 6.8, the desired data would be available only *after* the fourth stage of the first instruction in the dependence, which is too late for the *input* of the third stage of sub. Hence, even with forwarding, we would have to stall one stage for a *load-use data hazard*, as Figure 6.9 shows. Section 6.5 shows how pipelining hardware handles hard cases like these.

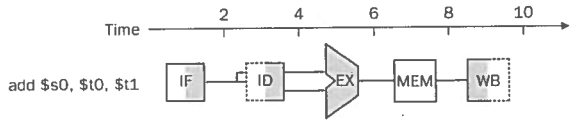


FIGURE 6.7 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 6.1 on page 437. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write back stage, with the drawing showing the register file being written. The shading indicates the element is used by the instruction. Hence MEM has a white background because `add` does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded in the fifth stage because the register file is written.

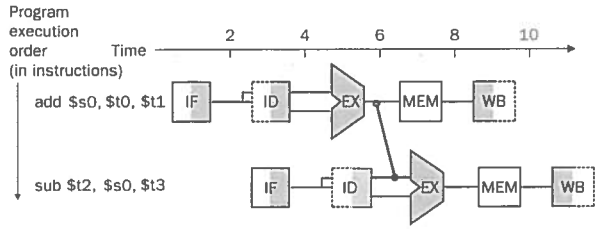


FIGURE 6.8 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `$s0` read in the second stage of `sub`.

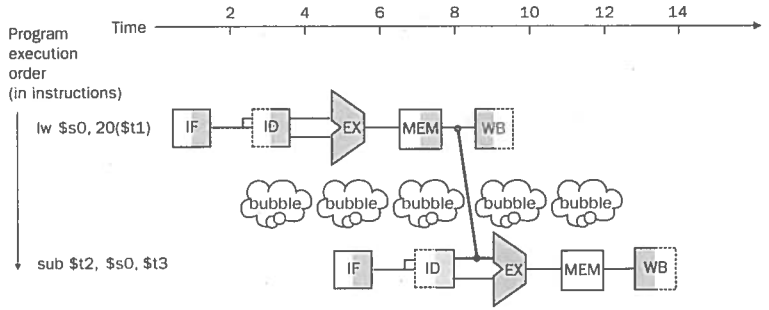


FIGURE 6.9 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backwards in time, which is impossible.

Reordering Code to Avoid Pipeline Stalls

Example Find the hazard in this code from the body of the `swap` procedure, from Figure 3.23 on page 164:

```
                          # reg $t1 has the address of v[k]
lw    $t0, 0($t1)         # reg $t0 (temp) = v[k]
lw    $t2, 4($t1)         # reg $t2 = v[k+1]
sw    $t2, 0($t1)         # v[k] = reg $t2
sw    $t0, 4($t1)         # v[k+1] = reg $t0 (temp)
```

Reorder the instructions to avoid pipeline stalls.

Answer The hazard occurs on register `$t2` between the second `lw` and the first `sw`. Swapping the two `sw` instructions removes this hazard:

```
                                # reg $t1 has the address of v[k]
lw      $t0, 0($t1)             # reg $t0 (temp) = v[k]
lw      $t2, 4($t1)             # reg $t2 = v[k+1]
sw      $t0, 4($t1)             # v[k+1] = reg $t0 (temp)
sw      $t2, 0($t1)             # v[k] = reg $t2
```

Note that we do not create a new hazard because there is still one instruction between the write of register `$t0` by the load and the read of register `$t0` in the store. Thus, on a machine with forwarding, the reordered sequence takes 4 clock cycles.

Hardware Software Interface

In an example of the trade-off between compiler and hardware complexity, the original MIPS processors avoided hardware to stall the pipeline by requiring software to follow a load with an instruction independent of that load. Such loads are called *delayed loads*.

Forwarding yields another insight into the MIPS architecture, in addition to the four mentioned on page 440. Each MIPS instruction writes a single result and does so at the end of its execution. Forwarding is harder if there are multiple results to forward per instruction or they need to write before the end of the instruction. For example, the PowerPC's load instructions may use update addressing (page 175 in Chapter 3), so the processor must be able to forward two results per load instruction.

Pipeline Overview Summary

Pipelining is a technique that exploits parallelism among the instructions in a sequential instruction stream. It has the substantial advantage that, unlike some speedup techniques (see Chapter 9), it is fundamentally invisible to the programmer.

In the next sections of this chapter, we cover the concept of pipelining using the MIPS instruction subset `lw`, `sw`, `add`, `sub`, `and`, `or`, `slt`, and `beq` (same as Chapter 5) and a simplified version of its pipeline. We then look at the problems that pipelining introduces and the performance attainable under typical situations.

The Big Picture

Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed. Pipelining does not reduce the time it takes to complete an individual instruction: the five-stage pipeline still takes 5 clock cycles for the instruction to complete. In the terms used in Chapter 2, page 56, pipelining improves instruction *throughput* rather than individual instruction *execution time*.

Instruction sets can either simplify or make life harder for pipeline designers, who must already cope with structural, control, and data hazards. Branch prediction, forwarding, and stalls help make a computer fast while still getting the right answers.

If you wish to take a more casual approach, we believe that after finishing this section, you have sufficient background to skip to sections 6.8 and 6.9 to familiarize yourself with advanced pipelining concepts, such as superscalar and dynamic pipelining, and to see how pipelining works in recent microprocessors.

Or if you are more dedicated, after finishing this section and Chapter 5, you are ready to understand the changes needed for pipelining in the datapath, explained in section 6.2, and the control lines, explained in section 6.3. You should be able to follow the datapath and control modifications for forwarding in section 6.4, and similar changes for stalls to resolve load-use hazards in section 6.5. You can then read section 6.6 to learn more details about solutions to branch hazards, and then see how exceptions are handled in section 6.7.

Elaboration: The name "forwarding" comes from the idea that the result is passed forward from an earlier instruction to a later instruction. "Bypassing" comes from passing the result by the register file to the desired unit.

6.2 A Pipelined Datapath

Figure 6.10 shows the single-cycle datapath from Chapter 5. The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus we must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

6.1 An Overview of Pipelining

Never waste time.

American proverb

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is key to making processors fast.

This section relies heavily on one analogy to give an overview of the pipelining terms and issues. If you are interested in just the big picture, you should concentrate on this section and then skip to section 6.8 to see how pipelining works and its implications on program performance. If you are interested in exploring the anatomy of a pipelined computer, you will find this section referred to repeatedly in sections 6.2 through 6.7.

Anyone who has done a lot of laundry has intuitively used pipelining. The *nonpipelined* approach to laundry would be:

1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

When your roommate is done, then start over with the next dirty load.

The *pipelined* approach takes much less time, as Figure 6.1 shows. As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load. When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and the next dirty load into the washer. Next you have your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer. At this point all steps—called *stages* in pipelining—are operating concurrently. As long as we have separate resources for each stage, we can pipeline the tasks.

The pipelining paradox is that the time from placing a single dirty sock in the washer until it is dried, folded, and put away is not shorter for pipelining; the reason pipelining is faster for many loads is that everything is working in parallel, so more loads are finished per hour. If all the stages take about the same amount of time and there is enough work to do, then the speedup due to pipelining is equal to the number of stages in the pipeline.

Pipelined laundry is potentially four times faster than nonpipelined: 20 loads would take about 5 times as long as 1 load, while 20 loads of sequential laundry takes 20 times as long as 1 load. It's only 2.3 times faster in Figure 6.1 because we only show 4 loads.

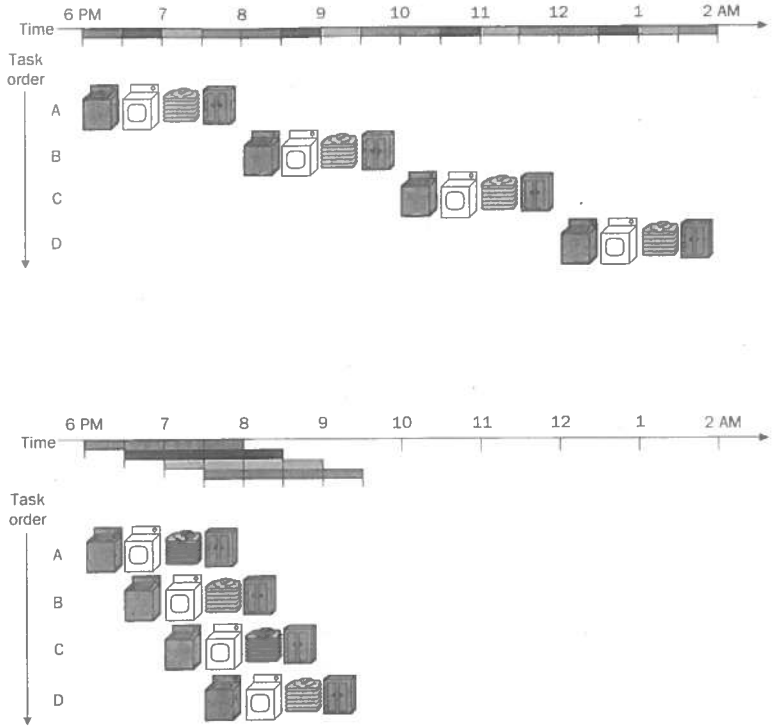


FIGURE 6.1 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for four loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional timeline, but we really have just one of each resource.

The same principles apply to processors where we pipeline instruction execution. MIPS instructions classically take five steps:

1. Fetch instruction from memory.
2. Read registers while decoding the instruction (the format of MIPS instructions allows reading and decoding to occur simultaneously).

3. Execute the operation or calculate an address.
4. Access an operand in data memory.
5. Write the result into a register.

Hence the MIPS pipeline we explore in this chapter has five stages. The following example shows that pipelining speeds up instruction execution just as it speeds up the laundry.

Single-Cycle versus Pipelined Performance

Example

To make this discussion concrete, let's create a pipeline. In this example, and in the rest of this chapter, we limit our attention to eight instructions: load word (`lw`), store word (`sw`), add (`add`), subtract (`sub`), and (`and`), or (`or`), set-less-than (`slt`), and branch-on-equal (`beq`).

Compare the average time between instructions of a single-cycle implementation, in which all instructions take 1 clock cycle, to a pipelined implementation. The operation times for the major functional units in this example are 2 ns for memory access, 2 ns for ALU operation, and 1 ns for register file read or write. (As we said in Chapter 5, in the single-cycle model every instruction takes exactly 1 clock cycle, so the clock cycle must be stretched to accommodate the slowest instruction.)

Answer

The time required for each of the eight instructions is shown in Figure 6.2. The single-cycle design must allow for the slowest instruction— in Figure 6.2 it is `lw`—so the time required for every instruction is 8 ns. Similarly to Figure 6.1, Figure 6.3 compares nonpipelined and pipelined execution of three load word instructions. Thus, the time between the first and fourth instructions in the nonpipelined design is 3×8 ns or 24 ns.

All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation. Just as the single-cycle design must take the worst-case clock cycle of 8 ns even though some instructions can be as fast as 5 ns, the pipelined execution clock cycle must have the worst-case clock cycle of 2 ns even though some stages take only 1 ns. Pipelining still offers a fourfold performance improvement: the time between the first and fourth instructions is 3×2 ns or 6 ns.

We can turn the pipelining speedup discussion above into a formula. If the stages are perfectly balanced, then the time between instructions on the pipelined machine—assuming ideal conditions—is equal to

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
Store word (`sw`)	2 ns	1 ns	2 ns	2 ns		7 ns
R-format (`add, sub, and, or, slt`)	2 ns	1 ns	2 ns		1 ns	6 ns
Branch (`beq`)	2 ns	1 ns	2 ns			5 ns

FIGURE 6.2 Total time for eight instructions calculated from the time for each component. This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

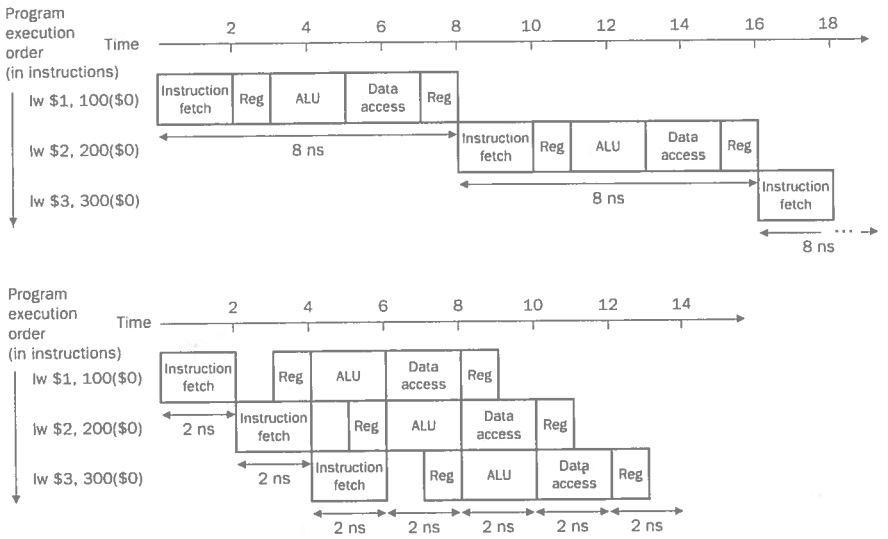


FIGURE 6.3 Single-cycle, nonpipelined execution in top vs. pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 6.2. In this case we see a fourfold speedup on average time between instructions, from 8 ns down to 2 ns. Compare this figure to Figure 6.1. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The computer pipeline stage times are limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

Under ideal conditions, the speedup from pipelining equals the number of pipe stages; a five-stage pipeline is five times faster.

The formula suggests that a five-stage pipeline should offer a fivefold improvement over the 8 ns nonpipelined time, or a 1.6-ns clock cycle. The example shows, however, that the stages may be imperfectly balanced. In addition, pipelining involves some overhead. Thus the time per instruction in the pipelined machine will exceed the minimum possible, and speedup will be less than the number of pipeline stages.

Moreover, even our claim of fourfold improvement for our example is not reflected in the total execution time for the three instructions: it's 14 ns versus 24 ns. To see why total execution time is less important, what would happen if we increased the number of instructions? We start by extending the previous figures to 1003 instructions. We would add 1000 instructions in the pipelined example; each instruction adds 2 ns to the total execution time. The total execution time would be 1000 × 2 ns + 14 ns, or 2,014 ns. In the nonpipelined example, we would add 1000 instructions, each taking 8 ns, so total execution time would be 1000 × 8 ns + 24 ns, or 8,024 ns. Under these ideal conditions, the ratio of total execution times for real programs on nonpipelined to pipelined machines is close to the ratio of times between instructions:

$$\frac{8{,}024 \text{ ns}}{2{,}014 \text{ ns}} = 3.98 \approx \frac{8 \text{ ns}}{2 \text{ ns}}$$

Pipelining improves performance by *increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction*, but instruction throughput is the important metric because real programs execute billions of instructions.

Designing Instruction Sets for Pipelining

Even with this simple explanation of pipelining, we can get insight into the design of the MIPS instruction set, which was designed for pipelined execution.

First, all MIPS instructions are the same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage. In an instruction set like the 80x86, where instructions vary from 1 byte to 17 bytes, pipelining is considerably more challenging.

Second, MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction. This symmetry means that the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched. If MIPS instruction formats were not symmetric, we would need to split stage 2, resulting in six pipeline stages. (We will shortly see the downside of longer pipelines.)

Third, memory operands only appear in loads or stores in MIPS. This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage. If we could operate on the operands in memory, as in the 80x86, stages 3 and 4 would expand to an address stage, memory stage, and then execute stage.

Fourth, operands must be aligned in memory (see the Hardware/Software Interface section on page 112 in Chapter 3). Hence we need not worry about a single data transfer instruction requiring two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage.

Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*. We explain the three types of hazards, using our analogy first, and then give the computer equivalent problem and solution.

Structural Hazards

The first hazard is called a *structural hazard*. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle. A structural hazard in the laundry room would occur if we used a washer-dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

As we said above, the MIPS instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline. Suppose, however, that we had a single memory instead of two memories. If the pipeline in Figure 6.3 had a fourth instruction, we would see that in the same clock cycle that the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipeline could have a structural hazard.

Control Hazards

The second hazard is called a *control hazard*, arising from the need to make a decision based on the results of one instruction while others are executing.

Suppose our laundry crew was given the happy task of cleaning the uniforms of a football team. Given how filthy the laundry is, we need to determine whether the detergent and water temperature setting we select is strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner. In our laundry pipeline, we have to wait until the second stage to examine the dry uniform to see if we need to change the washer setup or not. What to do?

Here are two solutions to control hazards in the laundry room and two computer equivalents.

> *Stall*: Just operate sequentially until the first batch is dry and then repeat until you have the right formula. This conservative option certainly works, but it is slow.

The equivalent decision task in a computer is the branch instruction. If the computer were to stall on a branch, then it would have to pause before continuing the pipeline. Let's assume that we put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage (see section 6.6 for details). Even with this extra hardware, the pipeline involving conditional branches would look like Figure 6.4. The `lw` instruction, executed if the branch fails, is stalled one extra 2-ns clock cycle before starting. This figure shows an important pipeline concept, officially called a *pipeline stall*, but often given the nickname *bubble*. We shall see stalls elsewhere in the pipeline.

Example

Stall on Branch Performance

Estimate the impact on the clock cycles per instruction (CPI) of stalling on branches. Assume all other instructions have a CPI of 1.

Answer

Figure 3.38 on page 189 in Chapter 3 shows conditional branches being 17% of the instructions executed for gcc. Since other instructions run have a CPI of 1 and branches took one extra clock cycle for the stall, then we would see a CPI of 1.17 and hence a slowdown of 1.17 versus the ideal case. (Since Figure 3.38 includes `slt` and `slti` as branch instructions, and they would not stall, this CPI result is appproximate.)

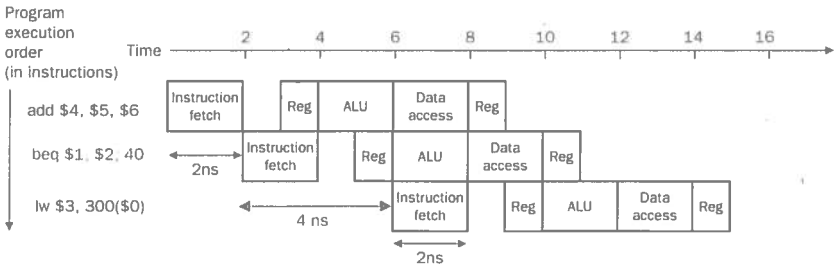


FIGURE 6.4 Pipeline showing stalling on every conditional branch as solution to control hazards. There is a one-stage pipeline stall, or bubble, after the branch.

If we cannot resolve the branch in the second stage, as is often the case for longer pipelines, then we'd see an even larger slowdown if we stall on branches. The cost of this option is too high for most computers to use and motivates a second solution to the control hazard:

> *Predict*: If you're pretty sure you have the right formula to wash uniforms, then just predict that it will work and wash the second load while waiting for the first load to dry. This option does not slow down the pipeline when you are correct. When you are wrong, however, you need to redo the load that was washed while guessing the decision.

Computers do indeed use prediction to handle branches. One simple approach is to always predict that branches will fail. When you're right, the pipeline proceeds at full speed. Only when branches succeed does the pipeline stall. Figure 6.5 shows such an example.

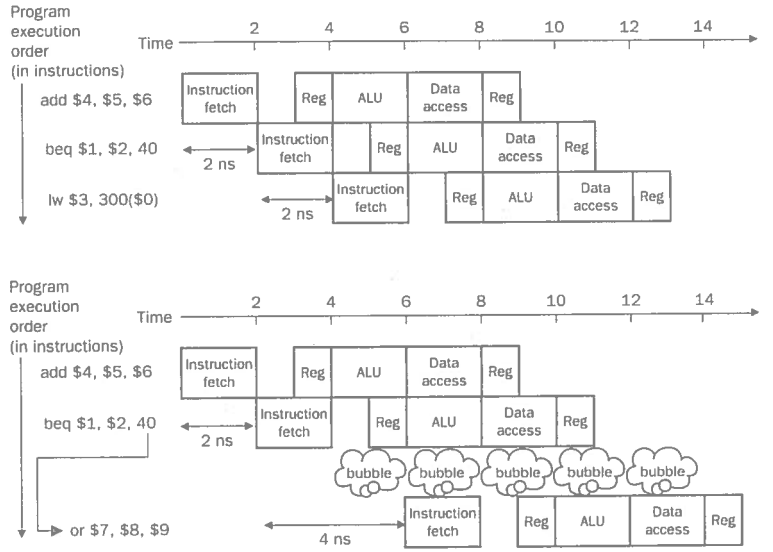


FIGURE 6.5 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows a taken branch.

A more sophisticated version of branch prediction would have some predicted as branching (*taken*) and some not branching (*untaken*). In our analogy, the dark or home uniforms might take one formula while the light or road uniforms might take another. As a computer example, at the bottom of loops are branches that jump back to the top of the loop. Since they are likely to be taken and they branch backwards, we could always predict taken for branches that jump to an earlier address.

Such rigid approaches to branch prediction rely on stereotypical behavior and don't account for the individuality of a specific branch instruction. *Dynamic* hardware predictors, in stark contrast, make their guesses depending on the behavior of each branch and may change predictions for a branch over the life of a program. Following our analogy, in dynamic prediction a person would look at how dirty the uniform was and guess at the formula, adjusting the next guess depending on the success of recent guesses. One popular approach to dynamic prediction in computers is keeping a history for each branch as taken or untaken, and then using the past to predict the future. Such hardware has about a 90% accuracy (see section 6.6). When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed branch have no effect and must restart the pipeline from the proper branch address.

As in the case of all other solutions to control hazards, longer pipelines exacerbate the problem, in this case by raising the cost of misprediction. Solutions to control hazards are described in more detail in section 6.6.

Elaboration: There is a third approach to the control hazard, called *delayed decision*. In our analogy, whenever you are going to make such a decision about laundry, just place a load of nonfootball clothes in the washer while waiting for football uniforms to dry. As long as you have enough dirty clothes that are not affected by the test, this solution works fine.

Called the *delayed branch* in computers, this is the solution actually used by the MIPS architecture. The delayed branch always executes the next sequential instruction, with the branch taking place *after* that one instruction delay. It is hidden from the MIPS assembly language programmer because the assembler can automatically arrange the instructions to get the branch behavior desired by the programmer. MIPS software will place an instruction immediately after the delayed branch instruction that is not affected by the branch, and a taken branch changes the address of the instruction that *follows* this safe instruction. In our example, the `add` instruction before the branch in Figure 6.4 does not affect the branch, so in Figure 6.6 we move it to the *delayed branch slot* following the branch.

Compilers typically fill about 50% of the branch delay slots with useful instructions. If the pipeline is longer than five stages, then we may get more branch delay slots, which are even harder to fill.

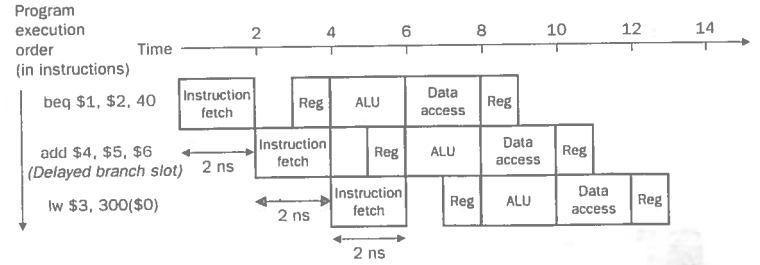


FIGURE 6.6 Pipeline delayed branch as solution to control hazard. The pipe bubble has been replaced by add.

Data Hazards

Returning to the laundry room, suppose that you are folding a load that is mostly socks. You realize that through bad luck the mate of every sock in this load is in another load that is still in the washer. You can't match the socks and put them away until that load is done. Hence you must stall the pipeline. This problem in computers is called a *data hazard*: an instruction depends on the results of a previous instruction still in the pipeline.

For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum (`$s0`):

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to add three bubbles to the pipeline.

Although we could try to rely on compilers to avoid such data hazards, we would fail. These dependencies happen just too often and the delay is just too long to expect the compiler to rescue us from this dilemma.

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Getting the missing item early from the internal resources is called *forwarding* or *bypassing*.

Forwarding with Two Instructions

Example For the two instructions above, show what pipeline stages would be connected by forwarding. Use the drawing in Figure 6.7 to represent the datapath during the five stages of the pipeline. Align a copy of the datapath for each instruction, similar to the laundry pipeline in Figure 6.1.

Answer Figure 6.8 shows the connection to forward the value in `$s0` after the execution stage of the `add` instruction as input to the execution stage of the `sub` instruction.

In this graphical representation of events, forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following, since that would mean going backwards in time.

Forwarding works very well, and is described in detail in section 6.4. It cannot prevent all pipeline stalls, however. For example, suppose the first instruction were a load of `$s0` instead of an add. As we can imagine from looking at Figure 6.8, the desired data would be available only *after* the fourth stage of the first instruction in the dependence, which is too late for the *input* of the third stage of sub. Hence, even with forwarding, we would have to stall one stage for a *load-use data hazard*, as Figure 6.9 shows. Section 6.5 shows how pipelining hardware handles hard cases like these.

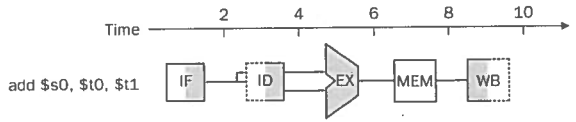


FIGURE 6.7 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 6.1 on page 437. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write back stage, with the drawing showing the register file being written. The shading indicates the element is used by the instruction. Hence MEM has a white background because `add` does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded in the fifth stage because the register file is written.

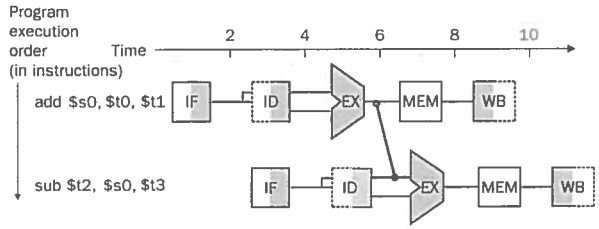


FIGURE 6.8 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `$s0` read in the second stage of `sub`.

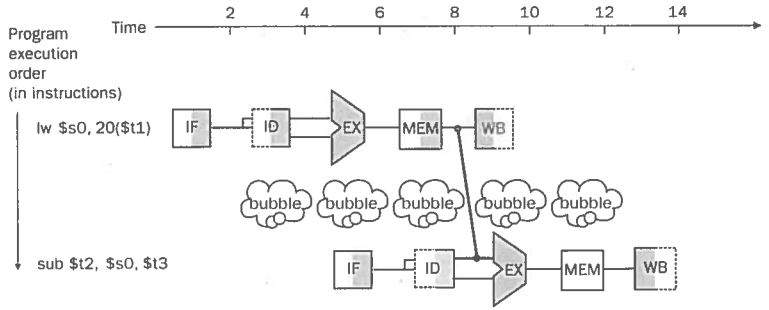


FIGURE 6.9 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backwards in time, which is impossible.

Reordering Code to Avoid Pipeline Stalls

Example Find the hazard in this code from the body of the `swap` procedure, from Figure 3.23 on page 164:

```
                          # reg $t1 has the address of v[k]
lw    $t0, 0($t1)         # reg $t0 (temp) = v[k]
lw    $t2, 4($t1)         # reg $t2 = v[k+1]
sw    $t2, 0($t1)         # v[k] = reg $t2
sw    $t0, 4($t1)         # v[k+1] = reg $t0 (temp)
```

Reorder the instructions to avoid pipeline stalls.

Answer The hazard occurs on register `$t2` between the second `lw` and the first `sw`. Swapping the two `sw` instructions removes this hazard:

```
                                # reg $t1 has the address of v[k]
lw      $t0, 0($t1)             # reg $t0 (temp) = v[k]
lw      $t2, 4($t1)             # reg $t2 = v[k+1]
sw      $t0, 4($t1)             # v[k+1] = reg $t0 (temp)
sw      $t2, 0($t1)             # v[k] = reg $t2
```

Note that we do not create a new hazard because there is still one instruction between the write of register `$t0` by the load and the read of register `$t0` in the store. Thus, on a machine with forwarding, the reordered sequence takes 4 clock cycles.

Hardware Software Interface

In an example of the trade-off between compiler and hardware complexity, the original MIPS processors avoided hardware to stall the pipeline by requiring software to follow a load with an instruction independent of that load. Such loads are called *delayed loads*.

Forwarding yields another insight into the MIPS architecture, in addition to the four mentioned on page 440. Each MIPS instruction writes a single result and does so at the end of its execution. Forwarding is harder if there are multiple results to forward per instruction or they need to write before the end of the instruction. For example, the PowerPC's load instructions may use update addressing (page 175 in Chapter 3), so the processor must be able to forward two results per load instruction.

Pipeline Overview Summary

Pipelining is a technique that exploits parallelism among the instructions in a sequential instruction stream. It has the substantial advantage that, unlike some speedup techniques (see Chapter 9), it is fundamentally invisible to the programmer.

In the next sections of this chapter, we cover the concept of pipelining using the MIPS instruction subset `lw`, `sw`, `add`, `sub`, `and`, `or`, `slt`, and `beq` (same as Chapter 5) and a simplified version of its pipeline. We then look at the problems that pipelining introduces and the performance attainable under typical situations.

The Big Picture

Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed. Pipelining does not reduce the time it takes to complete an individual instruction: the five-stage pipeline still takes 5 clock cycles for the instruction to complete. In the terms used in Chapter 2, page 56, pipelining improves instruction *throughput* rather than individual instruction *execution time*.

Instruction sets can either simplify or make life harder for pipeline designers, who must already cope with structural, control, and data hazards. Branch prediction, forwarding, and stalls help make a computer fast while still getting the right answers.

If you wish to take a more casual approach, we believe that after finishing this section, you have sufficient background to skip to sections 6.8 and 6.9 to familiarize yourself with advanced pipelining concepts, such as superscalar and dynamic pipelining, and to see how pipelining works in recent microprocessors.

Or if you are more dedicated, after finishing this section and Chapter 5, you are ready to understand the changes needed for pipelining in the datapath, explained in section 6.2, and the control lines, explained in section 6.3. You should be able to follow the datapath and control modifications for forwarding in section 6.4, and similar changes for stalls to resolve load-use hazards in section 6.5. You can then read section 6.6 to learn more details about solutions to branch hazards, and then see how exceptions are handled in section 6.7.

Elaboration: The name "forwarding" comes from the idea that the result is passed forward from an earlier instruction to a later instruction. "Bypassing" comes from passing the result by the register file to the desired unit.

6.2 A Pipelined Datapath

Figure 6.10 shows the single-cycle datapath from Chapter 5. The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus we must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

7.1 Introduction

From the earliest days of computing, programmers have wanted unlimited amounts of fast memory. The topics we will look at in this chapter all focus on aiding programmers by creating the illusion of unlimited fast memory. Before we look at how the illusion is actually created, let's consider a simple analogy that illustrates the key principles and mechanisms that we use.

Suppose you were a student writing a term paper on important historical developments in computer hardware. You are sitting at a desk in the engineering or math library with a collection of books that you have pulled from the shelves and are examining. You find that several of the important machines that you need to write about are described in the books you have, but there is nothing about the EDSAC. So, you go back to the shelves and look for an additional book. You find a book on early British computers that covers EDSAC. Once you have a good selection of books on the desk in front of you, there is a good probability that many of the topics you need can be found in them, and you may spend a great deal of time just using the books on the desk without going back to the shelves. Having several books on the desk in front of you saves time compared to having only one book there and constantly having to go back to the shelves to return it and take out another.

The same principle allows us to create the illusion of a large memory that we can access as fast as a very small memory. Just as you did not need to access all the books in the library at once with equal probability, a program does not access all of its code or data at once with equal probability. Otherwise, it would be impossible to make most memory accesses fast and still have large amounts of memory in machines, just as it would be impossible for you to fit all the library books on your desk and still have a chance of finding what you wanted quickly.

This *principle of locality* underlies both the way in which you did your work in the library and the way that programs operate. The principle of locality states that programs access a relatively small portion of their address space at any instant of time, just as you accessed a very small portion of the library's collection. There are two different types of locality:

- *Temporal locality* (locality in time): If an item is referenced, it will tend to be referenced again soon. If you recently brought a book to your desk to look at, you will probably need to look at it again soon.
- *Spatial locality* (locality in space): If an item is referenced, items whose addresses are close by will tend to be referenced soon. For example, when you brought out the book on early computers in England to find out about EDSAC, you also noticed that there was another book shelved next to it about early mechanical computers, so you also brought back that book and, later on, found something useful in that book. Books on the same topic are shelved together in the library to increase spatial locality. We'll see how spatial locality is used in memory hierarchies a little later in this chapter.

Just as accesses to books on the desk naturally exhibit locality, locality in programs arises from simple and natural program structures. For example, most programs contain loops, so instructions and data are likely to be accessed repeatedly, showing high amounts of temporal locality. Since instructions are normally accessed sequentially, programs show high spatial locality. Accesses to data also exhibit a natural spatial locality. For example, accesses to elements of an array or a record will naturally have high degrees of spatial locality.

We take advantage of the principle of locality by implementing the memory of a computer as a *memory hierarchy*. A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The fastest memories are more expensive per bit than the slower memories and thus are usually smaller.

Today, there are three primary technologies used in building memory hierarchies. Main memory is implemented from DRAM (dynamic random access memory), while levels closer to the CPU (caches) use SRAM (static random access memory). DRAM is less costly per bit than SRAM, although it is substantially slower. The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity for the same amount of silicon; the speed difference arises from several factors described in section B.5 of Appendix B. The final technology, used to implement the largest and slowest level in the hierarchy, is magnetic disk. The access time and price per bit vary widely among these technologies, as the table below shows, using typical values for 1997:

Memory technology	Typical access time	$ per MByte in 1997
SRAM	5–25 ns	$100–$250
DRAM	60–120 ns	$5–$10
Magnetic disk	10–20 million ns	$0.10–$0.20

Because of these differences in cost and access time, it is advantageous to build memory as a hierarchy of levels, with the faster memory close to the processor and the slower, less expensive memory below that, as shown in Figure 7.1. The goal is to present the user with as much memory as is available in the cheapest technology, while providing access at the speed offered by the fastest memory.

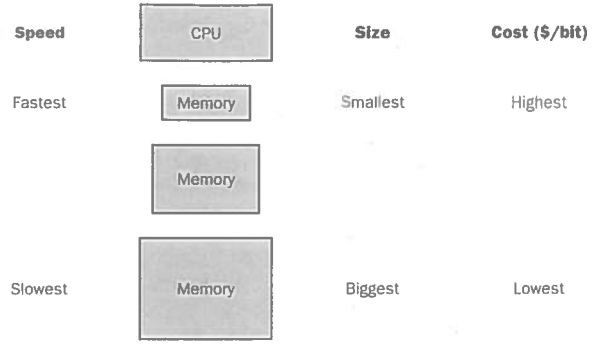


FIGURE 7.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory.

The memory system is organized as a hierarchy: a level closer to the processor is a subset of any level further away, and all the data is stored at the lowest level. By comparison, the books on your desk form a subset of the library you are working in, which is in turn a subset of all the libraries on campus. Furthermore, as we move away from the processor, the levels take progressively longer to access, just as we might encounter in a hierarchy of campus libraries.

A memory hierarchy can consist of multiple levels, but data is copied between only two adjacent levels at a time, so we can focus our attention on just two levels. The upper level—the one closer to the processor—is smaller and faster (since it uses more expensive technology) than the lower level. The minimum unit of information that can be either present or not present in the two-level hierarchy is called a *block*, as shown in Figure 7.2; in our library analogy, a block of information is one book.

If the data requested by the processor appears in some block in the upper level, this is called a *hit* (analogous to your finding the information in one of the books on your desk). If the data is not found in the upper level, the request is called a *miss*. The lower level in the hierarchy is then accessed to retrieve the block containing the requested data. (Continuing our analogy, you get up from your desk and go over to the shelves to look for the desired information.) The *hit rate*, or *hit ratio*, is the fraction of memory accesses found in the upper level; it is often used as a measure of the performance of the memory hierarchy. The *miss rate* (1 – hit rate) is the fraction of memory accesses not found in the upper level.

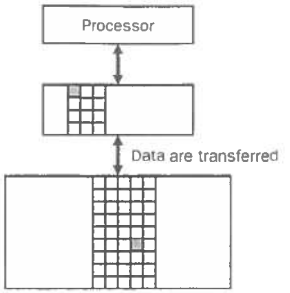


FIGURE 7.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a *block*. Usually we transfer an entire block when we copy something between levels.

Since performance is the major reason for having a memory hierarchy, the speed of hits and misses is important. *Hit time* is the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss (that is, the time needed to look through the books on the desk). The *miss penalty* is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor (or, the time to get another book from the shelves and place it on the desk). Because the upper level is smaller and built using faster memory parts, the hit time will be much smaller than the time to access the next level in the hierarchy, which is the major component of the miss penalty. (The time to examine the books on the desk is much smaller than the time to get up and go look for something in a book on the shelves.)

As we will see in this chapter, the concepts used to build memory systems affect many other aspects of a computer, including how the operating system manages memory and I/O, how compilers generate code, and even how applications use the machine. Of course, because all programs spend much of their time accessing memory, the memory system is necessarily a major factor in determining performance. The reliance on memory hierarchies to achieve performance has meant that programmers, who used to be able to think of memory as a flat, random access storage device, now need to understand how memory hierarchies work to get good performance. We show how important this understanding is with an example in the Fallacies and Pitfalls section.

Since memory systems are so critical to performance, computer designers have devoted a lot of attention to these systems and developed sophisticated mechanisms for improving the performance of the memory system. In this chapter we will see the major conceptual ideas, although many simplifications

and abstractions have been used to keep the material manageable in length and complexity. We could easily have written hundreds of pages on memory systems, as a number of recent doctoral theses have demonstrated.

The Big Picture

Programs exhibit both temporal locality, the tendency to reuse recently accessed data items, and spatial locality, the tendency to reference data items that are close to other recently accessed items. Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor. Memory hierarchies take advantage of spatial locality by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy.

A memory hierarchy uses smaller and faster memory technologies close to the processor, as shown in Figure 7.3. Thus accesses that hit in the highest level of the hierarchy can be processed quickly. Accesses that miss go to lower levels of the hierarchy, which are larger but slower. If the hit rate is high enough, the memory hierarchy has an effective access time close to that of the highest (and fastest) level and a size equal to that of the lowest (and largest) level.

In most systems, the memory is a true hierarchy, meaning that data cannot be present in level i unless it is present in level $i + 1$.

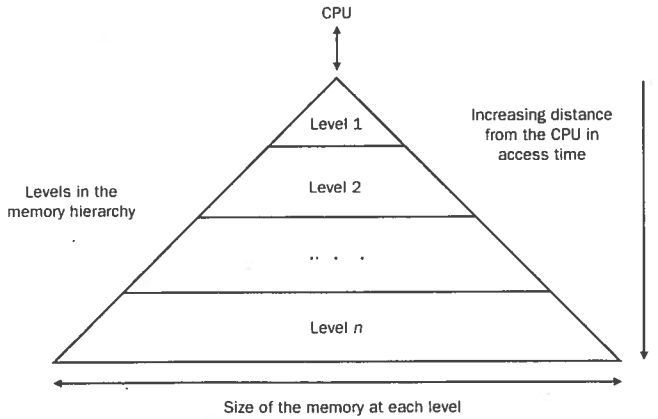


FIGURE 7.3 This diagram shows the structure of a memory hierarchy: as the distance from the CPU increases, so does the size. This structure with the appropriate operating mechanisms allows the CPU to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level n. Maintaining this illusion is the subject of this chapter.

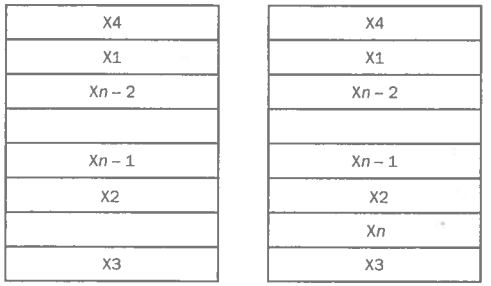
7.2 The Basics of Caches

Cache: a safe place for hiding or storing things.

Webster's New World Dictionary of the American Language, Third College Edition (1988)

In our library example, the desk acted as a cache—a safe place to store things (books) that we needed to examine. *Cache* was the name chosen to represent the level of the memory hierarchy between the CPU and main memory in the first commercial machine to have this extra level. Today, although this remains the dominant use of the word *cache*, the term is also used to refer to any storage managed to take advantage of locality of access. Caches first appeared in research machines in the early 1960s and in production machines later in that same decade; virtually every general-purpose machine built today, from the fastest to the slowest, includes a cache.

In this section, we begin by looking at a very simple cache in which the processor requests are each one word and the blocks also consist of a single word. Figure 7.4 shows such a simple cache, before and after requesting a data item that is not initially in the cache. Before the request, the cache contains a collection of recent references X1, X2, . . . , $Xn - 1$, and the processor requests a word Xn that is not in the cache. This request results in a miss, and the word Xn is brought from memory into cache.



a. Before the reference to Xn b. After the reference to Xn

FIGURE 7.4 The cache just before and just after a reference to a word Xn that is not initially in the cache. This reference causes a miss that forces the cache to fetch Xn from memory and insert it into the cache.

penalty of the primary cache, rather than directly affecting the primary cache hit time or the CPU cycle time.

The effect of these changes on the two caches can be seen by comparing each cache to the optimal design for a single level of cache. In comparison to a single-level cache, the primary cache of a multilevel cache is often smaller. Furthermore, the primary cache often uses a smaller block size, to go with the smaller cache size and reduced miss penalty. In comparison, the secondary cache will often be larger than in a single-level cache, since the access time of the secondary cache is less critical. With a larger total size, the secondary cache often will use a larger block size than appropriate with a single-level cache.

Elaboration: There are a number of complications that arise when multilevel caches are used. One of these is that there are now several different types of misses and corresponding miss rates. In the example above, we saw the primary cache miss rate and the *global miss rate*, that is, the fraction of references that missed in all levels. There is also a miss rate for the secondary cache that is given by the ratio of all misses in the secondary cache divided by the number of accesses. This miss rate is called the *local miss rate* of the secondary cache. Because the primary cache filters accesses, especially those with good spatial and temporal locality, the local miss rate of the secondary cache is much higher than the global miss rate. For the example above, we can compute the local miss rate of the secondary cache as: 2%/5% = 40%! Luckily, it is the combined miss rate that dictates how often we must access the main memory! Additional complications arise because the caches will likely have different block sizes to match the larger or smaller total size. Likewise, the associativity of the cache may change. On-chip primary caches are often built with associativity of two to four, while off-chip caches rarely have associativity of greater than two. These changes in block size and associativity introduce complications in the modeling of the caches, which typically means that both levels need to be simulated together to understand the behavior.

Summary

In this section, we focused on three topics: cache performance, using associativity to reduce miss rates, and the use of multilevel cache hierarchies to reduce miss penalties.

Since the total number of cycles spent on a program is the sum of the processor cycles and the memory-stall cycles, the memory system can have a significant effect on program execution time. In fact, as processors get faster (either by lowering CPI or by increasing the clock rate), the relative effect of the memory-stall cycles increases, making a good memory system critical to achieving high performance. The number of memory-stall cycles depends on both the miss rate and the miss penalty. The challenge, as we will see in section 7.5, is to reduce one of these factors without significantly affecting other critical factors in the memory hierarchy.

To reduce the miss rate, we examined the use of associative placement schemes. Such schemes can reduce the miss rate of a cache by allowing more flexible placement of blocks within the cache. Fully associative schemes allow blocks to be placed anywhere, but also require that every block in the cache be searched to satisfy a request. This search is usually implemented by having a comparator per cache block and searching the entries in parallel. The cost of the comparators makes large fully associative caches impractical. Set-associative caches are a practical alternative, since we need only search among the elements of a unique set that is chosen by indexing. Set-associative caches yield an improvement in hit rate but are slightly slower to access, because of the cost of the comparisons and the selection from among the elements of a set. Whether a direct-mapped cache or a set-associative cache yields better performance depends on both the technology and the details of the implementation.

Finally, we looked at multilevel caches as a technique to reduce the miss penalty by allowing a larger secondary cache to handle misses to the primary cache. Second-level caches have become commonplace as designers find that limited silicon and the goals of high clock rates prevent primary caches from becoming large. The secondary cache, which is often 10 or more times larger than the primary cache, catches many accesses that miss in the primary cache. In such cases, the miss penalty is that of the access time to the secondary cache (typically < 10 cycles) versus the access time to memory (typically > 40 cycles). As with associativity, the design trade-offs between size of the secondary cache and its access time depend on a number of aspects of the implementation.

7.4 Virtual Memory

. . . a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.

Kilburn et al., "One-level storage systems," 1962

In the previous section, we saw how caches served as a method for providing fast access to recently used portions of a program's code and data. Similarly, the main memory can act as a "cache" for the secondary storage, usually implemented with magnetic disks. This technique is called *virtual memory*. There are two major motivations for virtual memory: to allow efficient and safe sharing of memory among multiple programs and to remove the programming burdens of a small, limited amount of main memory.

Consider a collection of programs running at once on a machine. The total memory required by all the programs may be much larger than the amount of main memory available on the machine, but only a fraction of this memory is

actively being used at any point in time. Main memory need contain only the active portions of the many programs, just as a cache contains only the active portion of one program. This allows us to efficiently share the processor as well as the main memory. Of course, to allow multiple programs to share the same memory, we must be able to protect the programs from each other, ensuring that a program can only read and write the portions of main memory that have been assigned to it.

We cannot know which programs will share the memory with other programs when we compile them. In fact, the programs sharing the memory change dynamically while the programs are running. Because of this dynamic interaction, we would like to compile each program into its own *address space*, that is, a separate range of memory locations accessible only to this program. Virtual memory implements the translation of a program's address space to physical addresses. This translation process enforces protection of a program's address space from other programs.

A second motivation for virtual memory is to allow a single user program to exceed the size of primary memory. Formerly, if a program became too large for memory, it was up to the programmer to make it fit. Programmers divided programs into pieces and then identified the pieces that were mutually exclusive. These *overlays* were loaded or unloaded under user program control during execution, with the programmer ensuring that the program never tried to access an overlay that was not loaded and that the overlays loaded never exceeded the total size of the memory. Overlays were traditionally organized as modules, each containing both code and data. Calls between procedures in different modules would lead to overlaying of one module with another.

As you can well imagine, this responsibility was a substantial burden on programmers. Virtual memory, which was invented to relieve programmers of this difficulty, automatically manages the two levels of the memory hierarchy represented by main memory (sometimes called *physical memory* to distinguish it from virtual memory) and secondary storage.

Although the concepts at work in virtual memory and in caches are the same, their differing historical roots have led to the use of different terminology. A virtual memory block is called a *page*, and a virtual memory miss is called a *page fault*. With virtual memory, the CPU produces a *virtual address*, which is translated by a combination of hardware and software to a *physical address*, which in turn can be used to access main memory. Figure 7.20 shows the virtual addressed memory with pages mapped to main memory. This process is called *memory mapping* or *address translation*. Today, the two memory hierarchy levels controlled by virtual memory are DRAMs and magnetic disks (see Chapter 1, pages 19–20). If we return to our library analogy, we can think of a virtual address as the title of a book and a physical address as the location of that book in the library, such as might be given by the Library of Congress call number.

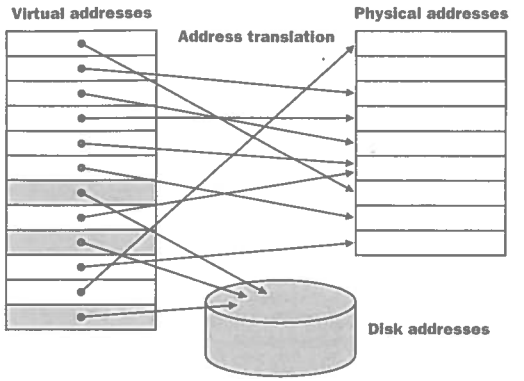


FIGURE 7.20 In virtual memory, blocks of memory (called pages) are mapped from one set of addresses (called virtual addresses) to another set (called physical addresses). The processor generates virtual addresses while the memory is accessed using physical addresses. Both the virtual memory and the physical memory are broken into pages, so that a virtual page is really mapped to a physical page. Of course, it is also possible for a virtual page to be absent from main memory and not be mapped to a physical address, residing instead on disk. Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code.

Virtual memory also simplifies loading the program for execution by providing *relocation*. Relocation maps the virtual addresses used by a program to different physical addresses before the addresses are used to access memory. This relocation allows us to load the program into any location in main memory. Furthermore, all virtual memory systems in use today relocate the program as a set of fixed-size blocks (pages), thereby eliminating the need to find a contiguous block of memory to allocate to a program; instead, the operating system need only find a sufficient number of pages in main memory. Formerly, relocation problems required special hardware and special support in the operating system; today, virtual memory also provides this function.

In virtual memory, the address is broken into a *virtual page number* and a *page offset*. Figure 7.21 shows the translation of the virtual page number to a *physical page number*. The physical page number constitutes the upper portion of the physical address, while the page offset, which is not changed, constitutes the lower portion. The number of bits in the page offset field determines the page size. The number of pages addressable with the virtual address need not match the number of pages addressable with the physical address. Having a larger number of virtual pages than physical pages is the basis for the illusion of an essentially unbounded amount of virtual memory.

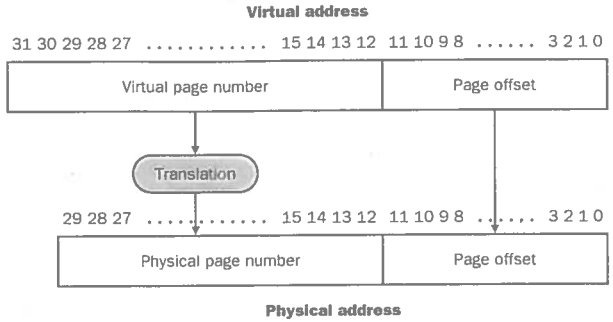


FIGURE 7.21 Mapping from a virtual to a physical address. The page size is 2^{12} = 4 KB. The number of physical pages allowed in memory is 2^{18}, since the physical page number has 18 bits in it. This means that main memory can have at most 1 GB, while the virtual address space is 4 GB.

Many design choices in virtual memory systems are motivated by the high cost of a miss, which in virtual memory is traditionally called a *page fault*. A page fault will take millions of cycles to process. (The table on page 541 shows the relative speeds of main memory and disk.) This enormous miss penalty, dominated by the time to get the first word for typical page sizes, leads to several key decisions in designing virtual memory systems:

- Pages should be large enough to amortize the high access time. Sizes from 4 KB to 16 KB are typical today, with new systems being developed to support 32-KB and 64-KB pages, and 4-KB pages are being phased out.
- Organizations that reduce the page fault rate are attractive. The primary technique used here is to allow fully associative placement of pages.
- Page faults can be handled in software because the overhead will be small compared to the access time to disk. Furthermore, software can afford to use clever algorithms for choosing how to place pages because even small reductions in the miss rate will pay for the cost of such algorithms.
- Using write-through to manage writes in virtual memory will not work, since writes take too long. Instead, virtual memory systems use write-back.

The next few sections address these factors in virtual memory design.

Elaboration: The discussion of virtual memory in this book focuses on paging, uses fixed-size blocks. There is also a variable-size block scheme called *segment*. In segmentation, an address consists of two parts: a segment number and a seg offset. The segment register is mapped to a physical address, and the offset is a to find the actual physical address. Because the segment can vary in size, a bo check is also needed to make sure that the offset is within the segment. The majo of segmentation is to support more powerful methods of protection and sharing address space. Most operating system textbooks contain extensive discussions o mentation compared to paging and of the use of segmentation to logically shar address space. The major disadvantage of segmentation is that it splits the ad space into logically separate pieces that must be manipulated as a tw address: the segment number and the offset. Paging, in contrast, makes the bou between page number and offset invisible to programmers and compilers.

Segments have also been used as a method to extend the address space wi changing the word size of the machine. Such attempts have been unsucce because of the awkwardness and performance penalties inherent in a two-part ad of which programmers and compilers must be aware.

Many architectures divide the address space into large fixed-size blocks that plify protection between the operating system and user programs and increase th ciency of implementing paging. Although these divisions are often called "segme this mechanism is much simpler than variable block size segmentation and is no ble to user programs; we discuss it in more detail shortly.

Placing a Page and Finding It Again

Because of the incredibly high penalty for a page fault, designers would to reduce the number of page faults by optimizing the page placement. I allow a virtual page to be mapped to any physical page, the operating sy can then choose to replace any page it wants when a page fault occurs example, the operating system can use a sophisticated algorithm and com data structures, which track page usage, to try to choose a page that wil be needed for a long time. The ability to use a clever and flexible replacer scheme is actually the primary motivation for using fully associative p ment of pages. Of course, fully associative placement also reduces the fault rate.

As we mentioned earlier, the difficulty in using fully associative placer is in locating an entry, since it can be anywhere in the upper level of the archy. A full search is impractical. In virtual memory, we locate pages by u a full table that indexes the memory; this structure is called a *page table*. A table, which resides in memory, is indexed with the page number from the tual address and contains the corresponding physical page number. Each gram has its own page table, which maps the virtual address space of program to main memory. In our library analogy, the page table corresp to a mapping between book titles and library locations. Just as the card cat may contain entries for books in another library on campus rather thar local branch library, we will see that the page table may contain entrie