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 and because of the functionality differences among operating systems.

user CPU time The CPU time spent in a program itself.

system CPU time The CPU time spent in the operating system performing tasks on behalf of the program.

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 focus 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 computer it's convenient to think about performance in other metrics. In particular, computer designers may want to think about a computer 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., 0.25 nanoseconds, 0.25 ns, 250 picoseconds, or 250 ps) and as the *clock rate* (e.g., 4 gigahertz, or 4 GHz), 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.

clock cycle Also called tick, clock tick, clock period, clock, cycle. The time for one clock period, usually of the processor clock, which runs at a constant rate.

clock period The length of each clock cycle.

Understanding Program Performance

Different applications are sensitive to different aspects of the performance of a computer system. Many applications, especially those running on servers, depend as much on I/O performance, which, in turn, relies on both hardware and software, and total elapsed time measured by a wall clock is the measurement of interest. In some application environments, the user may care about throughput, response time, or a complex combination of the two (e.g., maximum throughput with a worst-case response time). To improve the performance of a program, one must have a clear definition of what performance metric matters and then proceed to look for the performance bottlenecks by measuring program execution and looking for the likely bottlenecks. In the following chapters, we will describe how to search for bottlenecks and improve performance in various parts of the system.

Check Yourself

1. Suppose we know that an application that uses both a desktop client and a remote server is limited by network performance. For the following changes state whether only the throughput improves, both response time and throughput improve, or neither improves.
   a. An extra network channel is added between the client and the server, increasing the total network throughput and reducing the delay to obtain network access (since there are now two channels).
   b. The networking software is improved, thereby reducing the network communication delay, but not increasing throughput.
   c. More memory is added to the computer.
2. Computer C's performance is 4 times better than the performance of computer B, which runs a given application in 28 seconds. How long will computer C take to run that application?

4.2 CPU Performance and Its Factors

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 4 GHz clock. We are trying to help a computer designer build a computer, 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 computer B to require 1.2 times as many clock cycles as computer 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:

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

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{4 \times 10^9 \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 4 \times 10^9 \frac{\text{cycles}}{\text{second}} = 40 \times 10^9 \text{ cycles}$$

CPU time for B can be found using this equation:

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

$$6 \text{ seconds} = \frac{1.2 \times 40 \times 10^9 \text{ cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 40 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = \frac{8 \times 10^9 \text{ cycles}}{\text{second}} = 8 \text{ GHz}$$

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

Hardware Software Interface

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 computer 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

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

clock cycles per instruction (CPI) Average number of clock cycles per instruction for a program or program fragment.

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.

EXAMPLE

Using the Performance Equation

Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program, and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program, and by how much?

ANSWER

We know that each computer 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 computer:

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

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

Now we can compute the CPU time for each computer:

$$\begin{aligned}\text{CPU time}_A &= \text{CPU clock cycles}_A \times \text{Clock cycle time}_A \\ &= I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}\end{aligned}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

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

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer 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

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 computer. 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 are included on many processors, to record a variety of measurements, including the number of instructions executed, the average CPI, and often, the sources of performance loss. Since the instruction count depends

The BIG Picture

Figure 4.2 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 per program:

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

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 type of instructions executed, 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 4.2 The basic components of performance and how each is measured.

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 computer, 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 using hardware counters, when a CPU is operational. 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.

Understanding Program Performance

As we described in Chapter 1, the performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware. The following table summarizes how these components affect the factors in the CPU performance equation.

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more floating-point operations, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher-CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

EXAMPLE

Comparing Code Segments

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

	CPI for this instruction class		
	A	B	C
CPI	1	2	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 computers, 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. Several of the exercises ask you to evaluate a series of computer 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.

instruction mix A measure of the dynamic frequency of instructions across one or many programs.

Two of the major factors that affect CPI are the performance of the pipeline, which is the technique used by all modern processors to execute instructions, and the performance of the memory system. In Chapter 6, we will see how pipeline performance adds to the CPI through stalls, and in Chapter 7 we will see how the performance of the caches can increase the CPI due to stalls in the memory system.

Check Yourself

A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately it increases the CPI by 1.1. How fast can we expect the application to run using this new compiler?

a. $\frac{15 \times 0.6}{1.1} = 8.2\,\text{sec}$

b. $15 \times 0.6 \times 1.1 = 9.9\,\text{sec}$

c. $\frac{15 \times 1.1}{0.6} = 27.5\,\text{sec}$

4.3 Evaluating 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 computers. Most users, however, are not in this situation. Instead, they must rely on other methods that measure the performance of a candidate computer, hoping that the methods will reflect how well the computer will perform with the user's workload. This alternative is usually followed by evaluating the computer using a set of *benchmarks*—programs specifically chosen to measure performance. The benchmarks form a workload that the user hopes will predict the performance of the actual workload.

workload A set of programs run on a computer that is either the actual collection of applications run by a user or is constructed from real programs to approximate such a mix. A typical workload specifies both the programs as well as the relative frequencies.

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. In fact, 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 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 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.

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 computer because compilers may not be available until much later in the design. 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.

As mentioned earlier, different classes and applications of computers will require different types of benchmarks. For desktop computers, the most common benchmarks are either measures of CPU performance or benchmarks focusing on a specific task, such as DVD playback or graphics performance for games. In Section 4.4, we will examine the SPEC CPU benchmarks, which focus on CPU performance and measure response time to complete a benchmark. For servers, the decision of which benchmark to use depends heavily on the nature of the intended application. For scientific servers, CPU-oriented benchmarks with scientific applications are typically used, and response time to complete a benchmark is the metric. For other server environments, benchmarks of Web serving, file serving, and databases are commonly used. These server benchmarks usually emphasize throughput, albeit with possible requirements on response time to individual events, such as a database query or Web page request. Section 4.4 examines the SPECweb99 benchmark designed to test Web server performance. In embedded computing, good benchmarks are much more rare. Often customers use their specific embedded application or a segment of it for benchmarking purposes. The one major benchmark suite developed for embedded computers is EEMBC, and we discuss those benchmarks in the In More Depth section on the CD.

Once we have selected a set of suitable benchmarks and obtained performance measurements, we can write a performance report. The guiding principle in reporting performance measurements should be *reproducibility*—we should list everything another experimenter would need to duplicate the results. This list must include the version of the operating system, compilers, and the input, as well as the computer configuration. As an example, the system description section of a SPEC CPU2000 benchmark report is in Figure 4.3.

One important element of reproducibility is the choice of input. Different inputs can generate quite different behavior. For example, an input can trigger certain execution paths that may be typical, or it may exercise rarely used, and

Hardware	
Hardware vendor	Dell
Model number	Precision WorkStation 360 (3.2 GHz Pentium 4 Extreme Edition)
CPU	Intel Pentium 4 (800 MHz system bus)
CPU MHz	3200
FPU	Integrated
CPU(s) enabled	1
CPU(s) orderable	1
Parallel	No
Primary cache	12K(I) micro-ops + 8KB(D) on chip
Secondary cache	512KB(I+D) on chip
L3 cache	2048KB(I+D) on chip
Other cache	N/A
Memory	4 x 512MB ECC DDR400 SDRAM CL3
Disk subsystem	1x 80GB ATA/100 7200 RPM
Other hardware	

Software	
Operating system	Windows XP Professional SP1
Compiler	Intel C++ Compiler 7.1 (20030402Z) Microsoft Visual Studio.NET (7.0.9466) MicroQuill SmartHeap Library 6.01
File system type	NTFS
System state	Default

FIGURE 4.3 System description of a desktop system using the fastest Pentium 4 available in 2003. In addition to this formatted mandatory description, there are 23 lines of notes describing special flag settings used for portability (4), optimization (2), tuning (12), base timing (2), a special library (2), and BIOS setting (1).

hence less important, parts of an application. Some of the most important effects from the input set are in the memory system. Larger input sets tend to stress the memory system to a greater extent, and the use of realistically sized workloads in servers both for commercial and scientific applications is critical if a benchmark is intended to predict what real applications may see.

Comparing and Summarizing Performance

Once we have selected programs to use as benchmarks and agreed on whether we are measuring response time or throughput, you might think that performance comparison would be straightforward. However, we must still decide how to summarize the performance of a group of benchmarks. Although summarizing a set of

	Computer A	Computer B
Program 1 (seconds)	1	10
Program 2 (seconds)	1000	100
Total time (seconds)	1001	110

FIGURE 4.4 Execution times of two programs on two different computers. Taken from Figure 1 of Smith [1988].

measurements results in less information, marketers and even users often prefer to have a single number to compare performance. The key question is, How should a summary be computed? Figure 4.4, which is abstracted from an article about summarizing performance, illustrates some of the difficulties facing such efforts.

Using our definition of *faster*, the following statements hold for the program measurements in Figure 4.4:

- A is 10 times faster than B for program 1.
- B is 10 times faster than A for program 2.

Taken individually, each of these statements is true. Collectively, however, they present a confusing picture—the relative performance of computers A and B is unclear.

Total Execution Time: A Consistent Summary Measure

The simplest approach to summarizing relative performance is to use total execution time of the two programs. Thus

$$\frac{\text{Performance}_B}{\text{Performance}_A} = \frac{\text{Execution time}_A}{\text{Execution time}_B} = \frac{1001}{110} = 9.1$$

That is, B is 9.1 times as fast as A for programs 1 and 2 together.

This summary is directly proportional to execution time, our final measure of performance. If the workload consists of running programs 1 and 2 an equal number of times, this statement would predict the relative execution times for the workload on each computer.

The average of the execution times that is directly proportional to total execution time is the **arithmetic mean** (AM):

arithmetic mean The average of the execution times that is directly proportional to total execution time.

$$\text{AM} = \frac{1}{n}\sum_{i=1}^{n} \text{Time}_i$$

where Time_i is the execution time for the ith program of a total of n in the workload. Since it is the mean of execution times, a smaller mean indicates a smaller average execution time and thus improved performance.

The arithmetic mean is proportional to execution time, assuming that the programs in the workload are each run an equal number of times. Is that the right workload? If not, we can assign a weighting factor w_i to each program to indicate the frequency of the program in that workload. If, for example, 20% of the tasks in the workload were program 1 and 80% of the tasks in the workload were program 2, then the weighting factors would be 0.2 and 0.8. By summing the products of weighting factors and execution times, we can obtain a clear picture of the performance of the workload. This sum is called the **weighted arithmetic mean**. One method of weighting programs is to choose weights so that the execution time of each benchmark is equal on the computer used as the base. The standard arithmetic mean is a special case of the weighted arithmetic mean where all weights are equal. We explore the weighted mean in more detail in Exercises 4.15 and 4.16.

weighted arithmetic mean An average of the execution time of a workload with weighting factors designed to reflect the presence of the programs in a workload; computed as the sum of the products of weighting factors and execution times.

Check Yourself

1. Suppose you are choosing between four different desktop computers: one is an Apple MacIntosh and the other three are PC-compatible computers that use a Pentium 4, an AMD processor (using the same compiler as the Pentium 4), and a Pentium 5 (which does not yet exist in 2004 but has the same architecture as the Pentium 4 and uses the same compiler). Which of the following statements are true?
   a. The fastest computer will be the one with the highest clock rate.
   b. Since all PCs use the same Intel-compatible instruction set and execute the same number of instructions for a program, the fastest PC will be the one with the highest clock rate.
   c. Since AMD uses different techniques than Intel to execute instructions, they may have different CPIs. But, you can still tell which of the two Pentium-based PCs is fastest by looking at the clock rate.
   d. Only by looking at the results of benchmarks for tasks similar to your workload can you get an accurate picture of likely performance.
2. Assume the following measurements of execution time were taken:

Program	Computer A	Computer B
1	2 sec	4 sec
2	5 sec	2 sec

Which of the following statements is true?

a. A is faster than B for program 1.
b. A is faster than B for program 2.
c. A is faster than B for a workload with equal numbers of executions of program 1 and 2.
d. A is faster than B for a workload with twice as many executions of program 1 as of program 2.

4.4 Real Stuff: Two SPEC Benchmarks and the Performance of Recent Intel Processors

SPEC (System Performance Evaluation Corporation) is an effort funded and supported by a number of computer vendors to create standard sets of benchmarks for modern computer systems. It began in 1989 focusing on benchmarking workstations and servers using CPU-intensive benchmarks. (A more detailed history is contained in Section 4.7.) Today, SPEC offers a dozen different benchmark sets designed to test a wide variety of computing environments using real applications and strictly specified execution rules and reporting requirements. The SPEC benchmark sets include benchmarks for CPU performance, graphics, high-performance computing, object-oriented computing, Java applications, client-server models, mail systems, file systems, and Web servers. In this section, we examine the performance of a variety of Dell computer systems that use Pentium III and Pentium 4 processors using a CPU performance benchmark and a Web-oriented system benchmark.

Performance with SPEC CPU Benchmarks

The latest release of the **SPEC CPU benchmarks** is the SPEC CPU2000 suite, which consists of 12 integer and 14 floating-point programs, as shown in Figure 4.5. The SPEC CPU benchmarks are intended to measure CPU performance, although wall clock time is the reported measurement. Separate summaries are reported for the integer and floating-point benchmark sets. The execution time measurements are first normalized by dividing the execution time on a Sun Ultra 5_10 with a 300 MHz processor by the execution time on the measured computer; this normalization yields a measure, called the *SPEC ratio*, which has the advantage that bigger numeric results indicate faster performance (i.e., SPEC ratio is the inverse of execution time). A CINT2000 or CFP2000 summary measurement is obtained by taking the geometric mean of the SPEC ratios. (See the In More Depth section on the CD for a discussion of trade-offs in using geometric mean.)

system performance evaluation cooperative (SPEC) benchmark A set of standard CPU-intensive, integer and floating point benchmarks based on real programs.

Integer benchmarks		FP benchmarks	
Name	Description	Name	Type
gzip	Compression	wupwise	Quantum chromodynamics
vpr	FPGA circuit placement and routing	swim	Shallow water model
gcc	The Gnu C compiler	mgrid	Multigrid solver in 3-D potential field
mcf	Combinatorial optimization	applu	Parabolic/elliptic partial differential equation
crafty	Chess program	mesa	Three-dimensional graphics library
parser	Word processing program	galgel	Computational fluid dynamics
eon	Computer visualization	art	Image recognition using neural networks
perlbmk	perl application	equake	Seismic wave propagation simulation
gap	Group theory, interpreter	facerec	Image recognition of faces
vortex	Object-oriented database	ammp	Computational chemistry
bzip2	Compression	lucas	Primality testing
twolf	Place and rote simulator	fma3d	Crash simulation using finite-element method
		sixtrack	High-energy nuclear physics accelerator design
		apsi	Meteorology: pollutant distribution

FIGURE 4.5 The SPEC CPU2000 benchmarks. The 12 integer benchmarks in the left half of the table are written in C and C++, while the floating-point benchmarks in the right half are written in Fortran (77 or 90) and C. For more information on SPEC and on the SPEC benchmarks, see www.spec.org. The SPEC CPU benchmarks use wall clock time as the metric, but because there is little I/O, they measure CPU performance.

For a given instruction set architecture, increases in CPU performance can come from three sources:

1. Increases in clock rate
2. Improvements in processor organization that lower the CPI
3. Compiler enhancements that lower the instruction count or generate instructions with a lower average CPI (e.g., by using simpler instructions)

To illustrate such performance improvements, Figure 4.6 shows the SPEC CINT2000 and CFP2000 measurements for a series of Intel Pentium III and Pentium 4 processors measured using the Dell Precision desktop computers. Since SPEC requires that the benchmarks be run on real hardware and the memory system has a significant effect on performance, other systems with these processors may produce different performance levels. In addition to the clock rate differences, the Pentium III and Pentium 4 use different pipeline structures, which we describe in more detail in Chapter 6.

There are several important observations from these two performance graphs. First, observe that the performance of each of these processors scales almost linearly with clock rate increases. Often this is not the case, since losses in the mem-

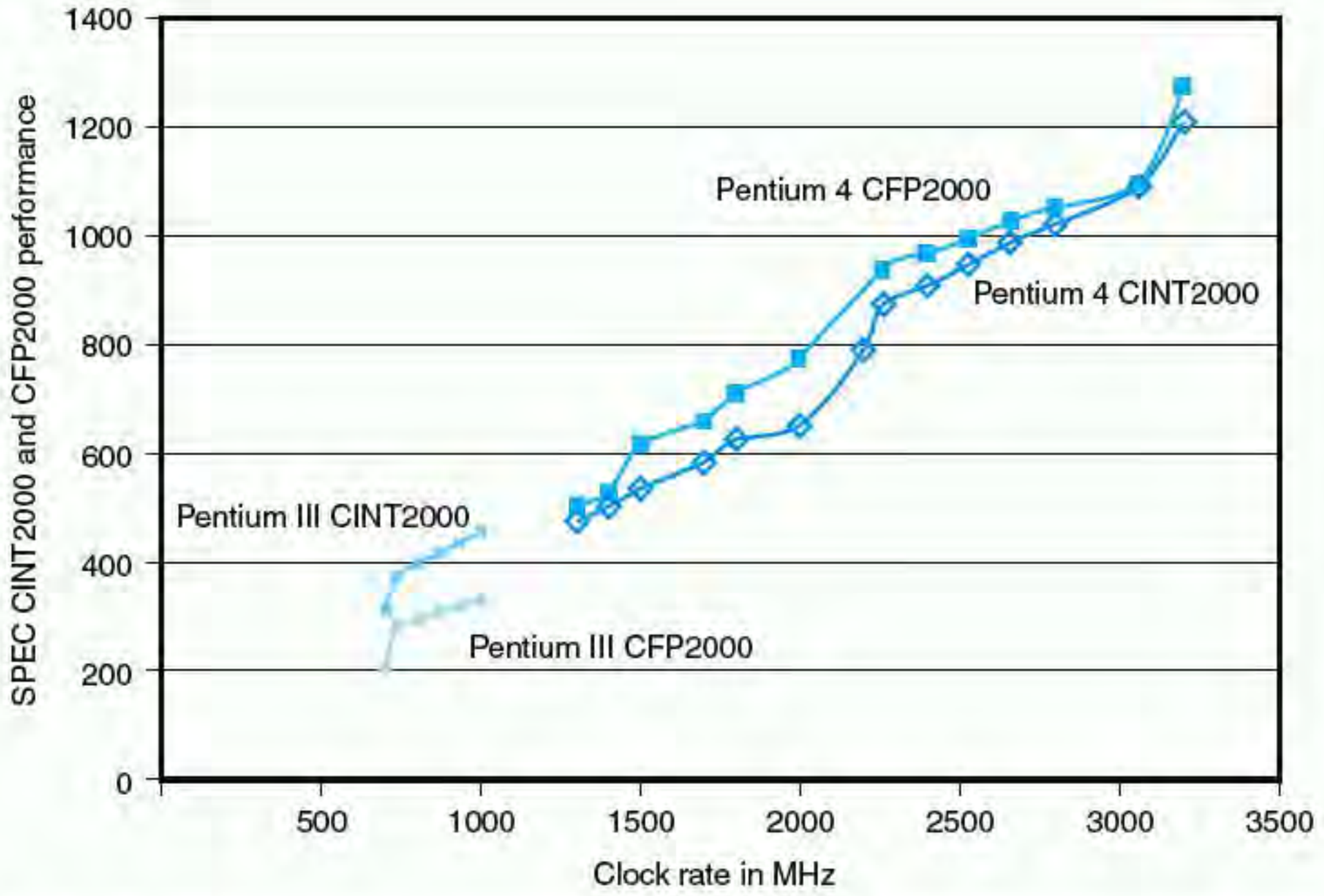


FIGURE 4.6 The SPEC CINT2000 and CFP2000 ratings for the Intel Pentium III and Pentium 4 processors at different clock speeds. SPEC requires two sets of measurements: one that allows aggressive optimization with benchmark-specific switches, and one that allows only the standard optimization switches (called the "base" measurements). This chart contains base measurements; for the integer benchmarks the differences are small. For more details on SPEC, see www.spec.org. The Pentium 4 Extreme (a version of the Pentium 4 introduced in late 2003) is not included in these results, since it uses a different cache architecture than the rest of the Pentium 4 processors.

ory system, which we discuss in Chapter 7, often worsen with higher clock rates. The strong performance of these processors is due both to the aggressive cache systems used in these processors, and the inability of many of the SPEC benchmarks to stress such a memory system.

Comparing the Pentium III and Pentium 4 performances yields even more interesting insights. In particular, note the relative positions of the CINT2000 and CFP2000 curves for the Pentium III versus the Pentium 4. One can quickly draw the inference that the Pentium 4 is either relatively better than the Pentium III on floating-point benchmarks or relatively worse on integer benchmarks. But, which is the case?

The Pentium 4 uses a more advanced integrated circuit technology as well as a more aggressive pipeline structure, both of which allow for a significant clock rate

increase. One comparison that is particularly interesting is the SPEC CINT2000 and CFP2000 measurement divided by the clock rate in MHz. The following tables summarizes the average value of this ratio across different clock rates for each processor:

Ratio	Pentium III	Pentium 4
CINT2000 / Clock rate in MHz	0.47	0.36
CFP2000 / Clock rate in MHz	0.34	0.39

Metrics such as benchmark performance divided by clock rate are sometimes thought of as measurements of implementation efficiency, although, as we have seen, one cannot separate the interaction of clock rate and other improvements.

These measurements are particularly interesting because of the differences between the integer and floating-point benchmarks. The CINT2000 performance ratios are typical: when a faster version of a processor is introduced it may sacrifice one aspect of a design (such as CPI) to enhance another (such as clock rate). Assuming one compiler for both processors, and hence identical code, the CINT2000 ratios tell us that the CPI of the Pentium 4 is 1.3 (0.47/0.36) times that of the Pentium 3.

How then can these numbers be reversed for the floating-point benchmarks? The answer is that the Pentium 4 provides a set of new instructions (called the Streaming SIMD Extensions 2; see Chapter 3) that provide a significant boost for floating point. Thus, both the instruction count and the CPI for the Pentium 4 will differ from that of the Pentium 3, producing improved performance.

SPECweb99: A Throughput Benchmark for Web Servers

In 1996, SPEC introduced its first benchmark designed to measure Web server performance; the benchmark was superseded by a new version in 1999. SPECweb99 differs from SPEC CPU in a number of ways:

- SPECweb99 focuses on throughput, measuring the maximum number of connections that a system running as a Web server can support. The system must provide response to a client request within a bounded time and with a bounded number of errors.
- Because SPECweb99 measures throughput, multiprocessors (systems with more than one CPU) are often used in benchmarks.
- SPECweb99 provides only a program to generate Web server requests; the Web server software becomes part of the system being measured.
- SPECweb99 performance depends on a wide measure of system characteristics, including the disk system and the network.

System	Processor	Number of disk drives	Number of CPUs	Number of networks	Clock rate (GHz)	Result
1550/1000	Pentium III	2	2	2	1	2765
1650	Pentium III	3	2	1	1.4	1810
2500	Pentium III	8	2	4	1.13	3435
2550	Pentium III	1	2	1	1.26	1454
2650	Pentium 4 Xeon	5	2	4	3.06	5698
4600	Pentium 4 Xeon	10	2	4	2.2	4615
6400/700	Pentium III Xeon	5	4	4	0.7	4200
6600	Pentium 4 Xeon MP	8	4	8	2	6700
8450/700	Pentium III Xeon	7	8	8	0.7	8001

FIGURE 4.7 SPECweb9999 performance for a variety of Dell PowerEdge systems using the Xeon versions of the Pentium III and Pentium 4 microprocessors.

To show how these characteristics produce a remarkably varied picture of Web server performance, we selected the SPECweb99 results for a series of Dell PowerEdge servers that use the Xeon versions of the Pentium III and Pentium 4 processors. The Xeon processors are built using the basic structure of the Pentium III or Pentium 4, but support multiprocessing. In addition the Xeon MP supports a third level of off-chip cache and can support more than two processors. The results for a variety of these Dell systems are shown in Figure 4.7.

Looking at the data in Figure 4.7, we can see that clock rate of the processors is clearly not the most important factor in determining Web server performance. In fact, the 8400 has twice as many slow processors as the 6600 and yet offers better performance. We expect these systems to be configured to achieve the best performance. That is, for a given set of CPUs, disks and additional networks are added until the processor becomes the bottleneck.

Performance, Power, and Energy Efficiency

As mentioned in Chapter 1, power is increasingly becoming the key limitation in processor performance. In the embedded market, where many processors go into environments that rely solely on passive cooling or on battery power, power consumption is often a constraint that is as important as performance and cost.

No doubt, many readers will have encountered power limitations when using their laptops. Indeed, between the challenges of removing excess heat and the limitations of battery life, power consumption has become a critical factor in the

design of processors for laptops. Battery capacity has improved only slightly over time, with the major improvements coming from new materials. Hence, the ability of the processor to operate efficiently and conserve power is crucial. To save power, techniques ranging from putting parts of the computer to sleep, to reducing clock rate and voltage, have all been used. In fact, power consumption is so important that Intel has designed a line of processors, the Pentium M series, specifically for mobile, battery-powered applications.

As we discussed in Chapter 1, for CMOS technology, we can reduce power by reducing frequency. Hence, recent processors intended for laptop use all have the ability to adapt frequency to reduce power consumption, simultaneously, of course, reducing performance. Thus, adequately evaluating the energy efficiency of a processor requires examining its performance at maximum power, at an intermediate level that conserves battery life, and at a level that maximizes battery life. In the Intel Mobile Pentium and Pentium M lines, there are two available clock rates: maximum and a reduced clock rate. The best performance is obtained by running at maximum speed, the best battery life by running always at the reduced rate, and the intermediate, performance-power optimized level by switching dynamically between these two clock rates.

Figure 4.8 shows the performance of three Intel Pentium processors designed for use in mobile applications running SPEC CINT2000 and SPEC CFP2000 as benchmarks. As we can see, the newest processor, the Pentium M, has the best performance when run a full clock speed, as well as with the adaptive clock rate mode. The Pentium M's 600 MHz clock makes it slower when run in minimum power mode than the Pentium 4-M, but still faster than the older Pentium III-M design.

For power-limited applications, the most important metric is probably energy efficiency, which is computed by taking performance and dividing by average power consumption when running the benchmark. Figure 4.9 shows the relative energy efficiency for these processors running the SPEC2000 benchmarks.This data clearly shows the energy efficiency advantage of the newer Pentium M design. In all three modes, it has a significant advantage in energy efficiency over the mobile versions of the Pentium III and Pentium 4. Notice that the Pentium 4-M has only a slight efficiency advantage over the Pentium III-M. This data clearly shows the advantage of a processor like the Pentium M, which is designed for reduced power usage from the start, as opposed to a design like the Pentium III-M or Pentium 4-M, which are modified versions of the standard processors. Of course, adequately measuring energy efficiency also requires the use of additional benchmarks designed to reflect how users employ battery-powered computers. Both PC review magazines and Intel's technical journal regularly undertake such studies.

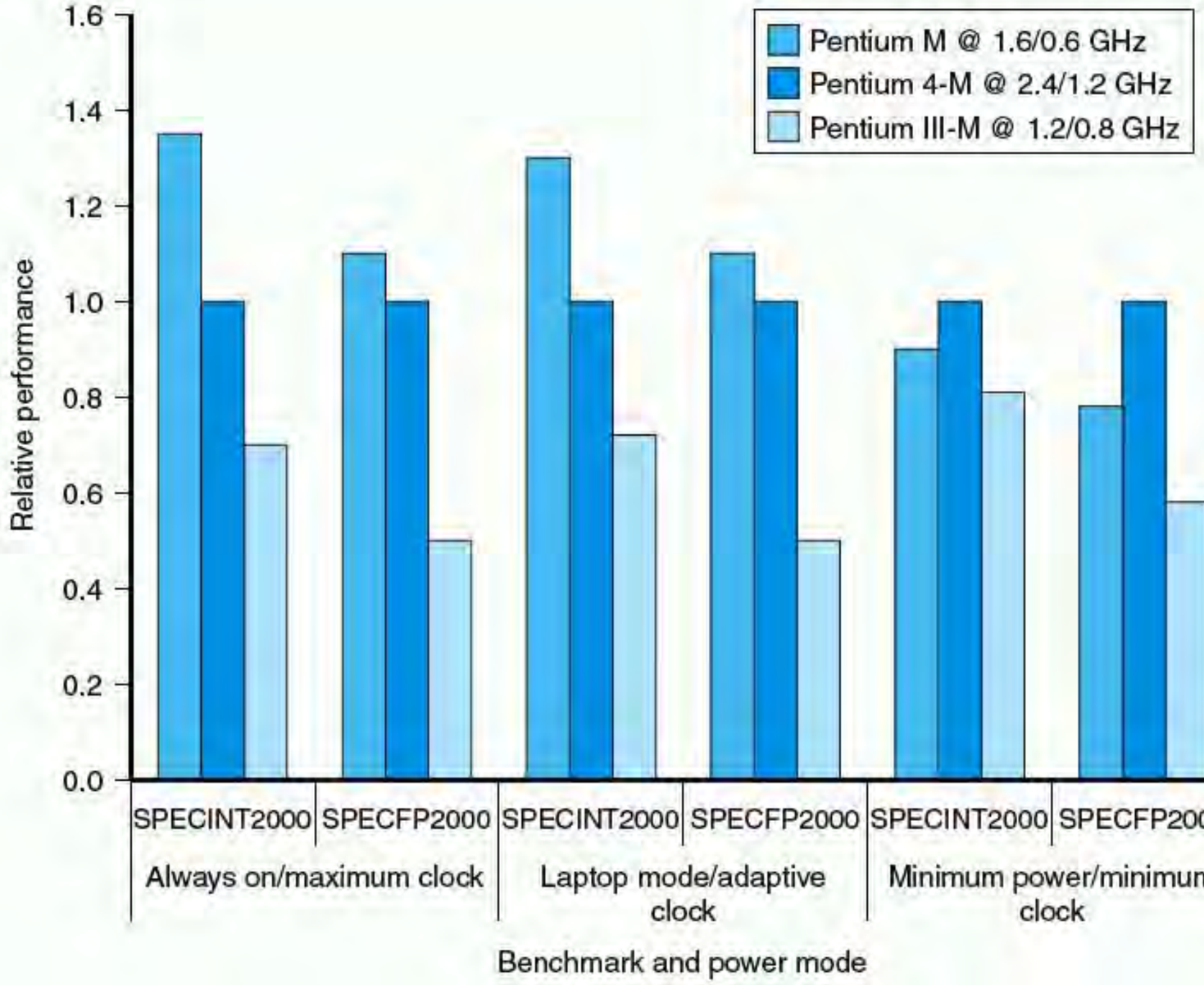


FIGURE 4.8 Relative performance of three Intel processors on SPECINT2000 and SPECFP2000 in three different modes. Each processor operates at two different clock rates, listed in the legend.

Check Yourself

Which of the following one-processor Pentium III configurations is likely to produce the best performance on SPECweb99 based on the data in Figure 4.7?

a. 1.26 GHz processor, 1 disk, 1 network connections
b. 1 GHz processor, 6 disks, 3 network connections
c. 1.1 GHz processor, 2 disks, 2 network connections

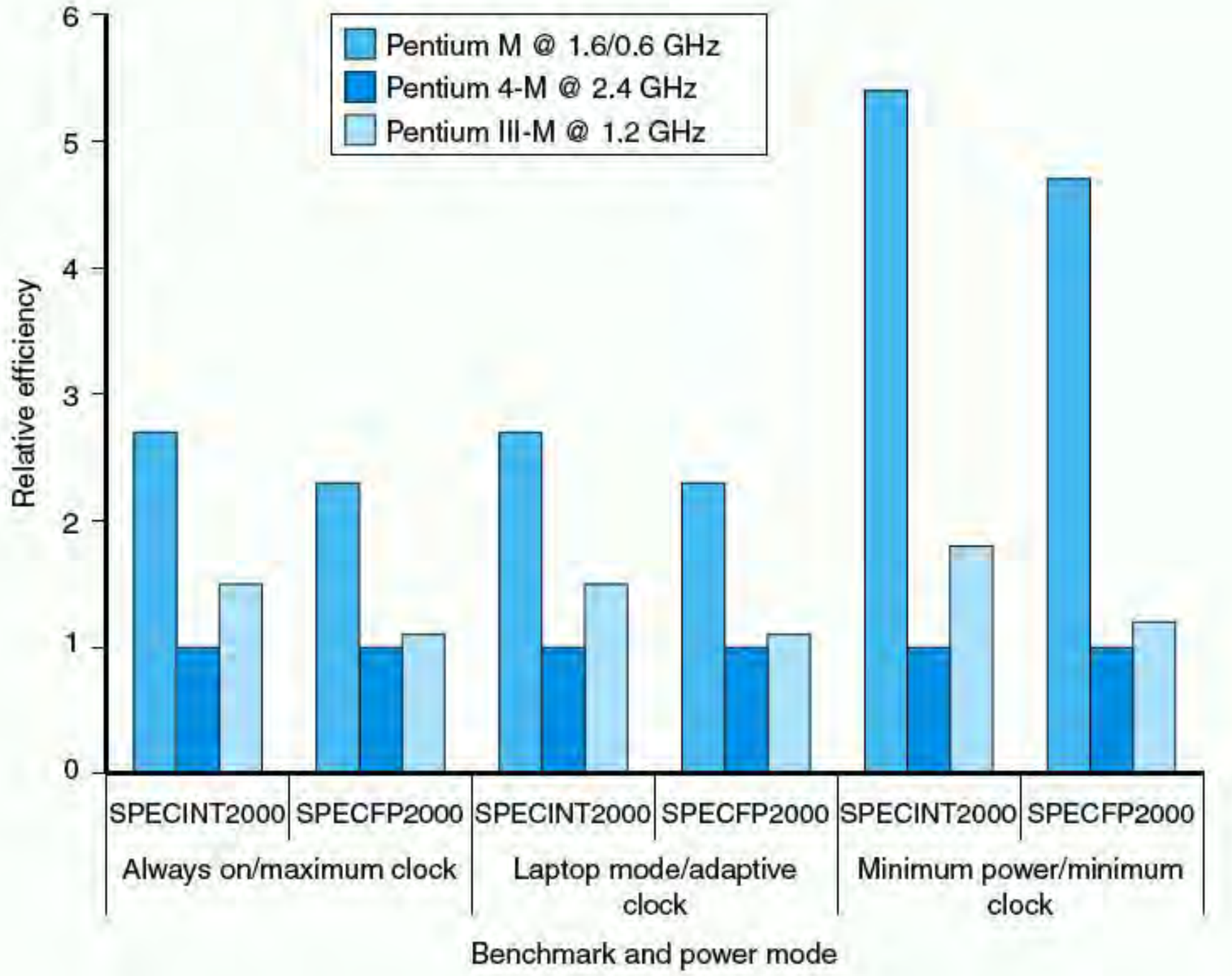


FIGURE 4.9 The relative energy efficiency of three mobile Pentium processors running SPEC2000 in three modes. Energy efficiency is measured as the inverse of joules consumed per benchmark, which is computed by dividing the inverse of the execution time for a benchmark by the watts dissipated.

4.5 Fallacies and Pitfalls

Cost/performance fallacies and pitfalls have ensnared many a computer architect, including us. Accordingly, this section suffers no shortage of relevant examples. We start with a pitfall that has trapped many designers and reveals an important relationship in computer design.

Pitfall: Expecting the improvement of one aspect of a computer to increase performance by an amount proportional to the size of the improvement.

This pitfall has visited designers of both hardware and software. A simple design problem illustrates it well. Suppose a program runs in 100 seconds on a computer, with multiply operations responsible for 80 seconds of this time. How much do I have to improve the speed of multiplication if I want my program to run five times faster?

The execution time of the program after I make the improvement is given by the following simple equation known as **Amdahl's law**:

$$\text{Execution time after improvement} = \left(\frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}\right)$$

Amdahl's law A rule stating that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.

For this problem:

$$\text{Execution time after improvement} = \frac{80 \text{ seconds}}{n} + (100 - 80 \text{ seconds})$$

Since we want the performance to be five times faster, the new execution time should be 20 seconds, giving

$$20 \text{ seconds} = \frac{80 \text{ seconds}}{n} + 20 \text{ seconds}$$

$$0 = \frac{80 \text{ seconds}}{n}$$

That is, there is *no amount* by which we can enhance multiply to achieve a fivefold increase in performance, if multiply accounts for only 80% of the workload. The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.

This concept also yields what we call the law of diminishing returns in everyday life. We can use Amdahl's law to estimate performance improvements when we know the time consumed for some function and its potential speed up. Amdahl's law, together with the CPU performance equation, are handy tools for evaluating potential enhancements. Amdahl's law is explored in more detail in the exercises and in the **In More Depth**: Amdal's Law on the CD.

A common theme in hardware design is a corollary of Amdahl's law: *Make the common case fast.* This simple guideline reminds us that in many cases the frequency with which one event occurs may be much higher than another. Amdahl's law reminds us that the opportunity for improvement is affected by how much time the event consumes. Thus, making the common case fast will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is often easier to enhance.

Pitfall: Using a subset of the performance equation as a performance metric.

We have already shown the fallacy of predicting performance based on simply one of clock rate, instruction count, or CPI. Another common mistake is to use two of the three factors to compare performance. Although using two of the three factors may be valid in a limited context, it is also easily misused. Indeed, nearly all proposed alternatives to the use of time as the performance metric have led eventually to misleading claims, distorted results, or incorrect interpretations.

million instructions per second (MIPS) A measurement of program execution speed based on the number of millions of instructions. MIPS is computed as the instruction count divided by the product of the execution time and 10^6.

One alternative to time as the metric is **MIPS (million instructions per second)**. For a given program, MIPS is simply

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

This MIPS measurement is also called *native MIPS* to distinguish it from some alternative definitions of MIPS that we discuss in in Section 4.8 on the CD.

Since MIPS is an instruction execution rate, MIPS specifies performance inversely to execution time; faster computers have a higher MIPS rating. The good news about MIPS is that it is easy to understand, and faster computers mean bigger MIPS, which matches intuition.

There are three problems with using MIPS as a measure for comparing computers. First, MIPS specifies the instruction execution rate but does not take into account the capabilities of the instructions. We cannot compare computers with different instruction sets using MIPS, since the instruction counts will certainly differ. In our earlier example examining the SPEC CFP2000 performance, using MIPS to compare the performance of the Pentium III and Pentium 4 would yield misleading results. Second, MIPS varies between programs on the same computer; thus a computer cannot have a single MIPS rating for all programs. Finally and most importantly, MIPS can vary inversely with performance! There are many examples of this anomalous behavior; one is given below.

EXAMPLE

MIPS as a Performance Measure

Consider the computer with three instruction classes and CPI measurements from the last example on page 252. Now suppose we measure the code for the same program from two different compilers and obtain the following data:

Code from	Instruction counts (in billions) for each instruction class		
	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

Assume that the computer's clock rate is 4 GHz. Which code sequence will execute faster according to MIPS? According to execution time?

ANSWER

First we find the execution time for the two different compilers using the following equation:

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

We can use an earlier formula for CPU clock cycles:

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

$$\text{CPU clock cycles}_1 = (5 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 10 \times 10^9$$

$$\text{CPU clock cycles}_2 = (10 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 15 \times 10^9$$

Now, we find the execution time for the two compilers:

$$\text{Execution time}_1 = \frac{10 \times 10^9}{4 \times 10^9} = 2.5 \text{ seconds}$$

$$\text{Execution time}_2 = \frac{15 \times 10^9}{4 \times 10^9} = 3.75 \text{ seconds}$$

So, we conclude that compiler 1 generates the faster program, according to execution time. Now, let's compute the MIPS rate for each version of the program, using

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

$$\text{MIPS}_1 = \frac{(5 + 1 + 1) \times 10^9}{2.5 \times 10^6} = 2800$$

$$\text{MIPS}_2 = \frac{(10 + 1 + 1) \times 10^9}{3.75(30) \times 10^6} = 3200$$

So, the code from compiler 2 has a higher MIPS rating, but the code from compiler 1 runs faster!

As examples such as this show, MIPS can fail to give a true picture of performance—even when comparing two versions of the same program on the same computer. In Section 2.7, we discuss other uses of the term *MIPS*, and how such usages can also be misleading.

Check Yourself

Consider the following performance measurements for a program:

Measurement	Computer A	Computer B
Instruction count	10 billion	8 billion
Clock rate	4 GHz	4 GHz
CPI	1.0	1.1

a. Which computer has the higher MIPS rating?
b. Which computer is faster?

4.6 Concluding Remarks

Although we have focused on performance and how to evaluate it in this chapter, designing only for performance without considering cost, functionality, and other requirements is unrealistic. All computer designers must balance performance and cost. Of course, there exists a domain of *high-performance design*, in which performance is the primary goal and cost is secondary. Much of the supercomputer and high-end server industry designs in this fashion. At the other extreme is *low-cost design*, typified by the embedded market, where cost and power take precedence over performance. In the middle are most desktop designs, as well as low-end servers; these computers require *cost/performance design*, in which the designer balances cost against performance. Examples from the desktop computer industry typify the kinds of trade-offs that designers in this region must live with.

We have seen in this chapter that there is a reliable method of determining and reporting performance, using the execution time of real programs as the metric. This execution time is related to other important measurements we can make by the following equation:

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

We will use this equation and its constituent factors many times. Remember, though, that individually the factors do not determine performance: Only the product, which equals execution time, is a reliable measure of performance.

The BIG Picture

Execution time is the only valid and unimpeachable measure of performance. Many other metrics have been proposed and found wanting. Sometimes these metrics are flawed from the start by not reflecting execution time; other times a metric valid in a limited context is extended and used beyond that context or without the additional clarification needed to make it valid.

Similarly, any measure that summarizes performance should reflect execution time. Weighted arithmetic means summarize performance while tracking execution time. Through the use of weights, a weighted arithmetic mean can adjust for different running times, balancing the contribution of each benchmark to the summary.

Of course, simply knowing this equation is not enough to guide the design or evaluation of a computer. We must understand how the different aspects of a design affect each of these key parameters. This insight involves a wide variety of issues, from the effectiveness of the compiler, to the effects of instruction set design on instruction count, to the impact of pipelining and memory systems on CPI, to the interaction between the technology and organization that determine the clock rate. The art of computer design and evaluation lies not in plugging numbers into a performance equation, but in accurately determining how alternatives will affect performance and cost.

Most computer users care about both cost and performance. While understanding the relationship among aspects of a design and its performance is challenging, determining the cost of various design features is often a more difficult problem. The cost of a computer is affected not only by the cost of the components, but by the costs of labor to assemble the computer, of research and development overhead, of sales and marketing, and of the profit margin. Finally, because of the rapid change in implementation technologies, the most cost-effective choice today is often suboptimal in six months or a year.

Computer designs will always be measured by cost and performance, as well as other important factors such as power, reliability, cost of ownership, and scalability. Although this chapter has focused on performance, the best designs will strike the appropriate balance for a given market among all these factors.

4.7 Historical Perspective and Further Reading

This section, which reviews the history of performance measurement and benchmarking, appears as Section 4.7on the CD.

4.8 Exercises

4.1 [5] <§4.1> We wish to compare the performance of two different computers: M1 and M2. The following measurements have been made on these computers:

Program	Time on M1	Time on M2
1	2.0 seconds	1.5 seconds
2	5.0 seconds	10.0 seconds

Which computer is faster for each program, and how many times as fast is it?

4.2 [5] <§4.1> Consider the two computers and programs in Exercise 4.1. The following additional measurements were made:

Program	Instructions executed on M1	Instructions executed on M2
1	5×10^9	6×10^9

Find the instruction execution rate (instructions per second) for each computer when running program 1.

4.3 [5] <§4.1> Suppose that M1 in Exercise 4.1 costs $500 and M2 costs $800. If you needed to run program 1 a large number of times, which computer would you buy in large quantities? Why?

4.4 [10] <§4.1> **For More Practice:** Cost-Effective Computing

4.5 [5] <§4.1> **For More Practice:** Cost-Effective Computing

4.6 [5] <§4.1> Another user has the following requirements for the computers discussed in Exercise 4.1: P1 must be executed 1600 times each hour. Any remaining time is used to run P2. If the computer has enough performance to execute program 1 the required number of times per hour, then performance is measured

by the throughput for program 2. Which computer is faster for this workload? Which computer is more cost-effective?

4.7 [10] <§4.2> Suppose you wish to run a program P with 7.5×10^9 instructions on a 5 GHz machine with a CPI of 0.8.

a. What is the expected CPU time?

b. When you run P, it takes 3 seconds of wall clock time to complete. What is the percentage of the CPU time P received?

4.8 [10] <§4.2> Consider two different implementations, P1 and P2, of the same instruction set. There are five classes of instructions (A, B, C, D, and E) in the instruction set.

P1 has a clock rate of 4 GHz. P2 has a clock rate of 6 GHz. The average number of cycles for each instruction class for P1 and P2 is as follows:

Class	CPI on P1	CPI on P2
A	1	2
B	2	2
C	3	2
D	4	4
E	3	4

Assume that peak performance is defined as the fastest rate that a computer can execute any instruction sequence. What are the peak performances of P1 and P2 expressed in instructions per second?

4.9 [5] <§§4.1–4.2> If the number of instructions executed in a certain program is divided equally among the classes of instructions in Exercise 4.8 except for class A, which occurs twice as often as each of the others, how much faster is P2 than P1?

4.10 [12] <§4.2> Consider two different implementations, I1 and I2, of the same instruction set. There are three classes of instructions (A, B, and C) in the instruction set. I1 has a clock rate of 6 GHz, and I2 has a clock rate of 3 GHz. The average number of cycles for each instruction class on I1 and I2 is given in the following table:

Class	CPI on M1	CPI on M2	C1 Usage	C2 Usage	C3 Usage
A	2	1	40%	40%	50%
B	3	2	40%	20%	25%
C	5	2	20%	40%	25%

The table also contains a summary of average proportion of instruction classes generated by three different compilers. C1 is a compiler produced by the makers of I1,

C2 is produced by the makers of I2, and the other compiler is a third-party product. Assume that each compiler uses the same number of instructions for a given program but that the instruction mix is as described in the table. Using C1 on both I1 and I2, how much faster can the makers of I1 claim I1 is compared to I2? Using C2, how much faster can the makers of I2 claim that I2 is compared to I1? If you purchase I1, which compiler would you use? If you purchased I2, which compiler would you use? Which computer and compiler would you purchase if all other criteria are identical, including cost?

4.11 [5] <§4.2> Consider program P, which runs on a 1 GHz machine M in 10 seconds. An optimization is made to P, replacing all instances of multiplying a value by 4 (mult X, X,4) with two instructions that set x to $x + x$ twice (add X,X; add X,X). Call this new optimized program P′. The CPI of a multiply instruction is 4, and the CPI of an add is 1. After recompiling, the program now runs in 9 seconds on machine M. How many multiplies were replaced by the new compiler?

4.12 [5] <§4.2> Your company could speed up a Java program on their new computer by adding hardware support for garbage collection. Garbage collection currently comprises 20% of the cycles of the program. You have two possible changes to the machine. The first one would be to automatically handle garbage collection in hardware. This causes an increase in cycle time by a factor of 1.2. The second would be to provide for new hardware instructions to be added to the ISA that could be used during garbage collection. This would halve the number of instruction needed for garbage collections but increase the cycle time by 1.1. Which of these two options, if either, should you choose?

4.13 [5] <§4.2> For the following set of variables, identify all of the subsets that can be used to calculate execution time. Each subset should be minimal; that is, it should not contain any variable that is not needed.

{CPI, clock rate, cycle time, MIPS, number of instructions in program, number of cycles in program}

4.14 [5] <§4.2> The table below shows the number of floating-point operations executed in three different programs and the runtime for those programs on three different computers:

Program	Floating-point operations	Execution time in seconds		
		Computer A	Computer B	Computer C
Program 1	5×10^9	2	5	10
Program 2	20×10^9	20	20	20
Program 3	40×10^9	200	50	15

Which computer is fastest according to total execution time? How many times as fast is it compared to the other two computers?

4.15 [15] <§§4.2, 4.3> One user has told you that the three programs in Exercise 4.14 constitute the bulk of his workload, but he does not run them equally. The user wants to determine how the three computers compare when the workload consists of different mixes of these three programs. (You know you can use the arithmetic mean to find the relative performance.)

Suppose the total number of floating-point operations (FLOPs) executed in the workload is equally divided among the three programs. That is, program 1 runs 8 times for every time program 3 runs, and program 2 runs twice for every time program 3 runs. Find which computer is fastest for this workload and by what factor. How does this compare with the total execution time with equal numbers of program executions?

4.16 [15] <§§4.2, 4.3> An alternative weighting to that of Exercise 4.15 is to assume that equal amounts of time will be spent running each program on one of the computers. Which computer is fastest using the data given for Exercise 4.14 and assuming a weighting that generates equal execution time for each of the benchmark programs on computer A? Which computer is fastest if we assume a weighting that generates equal execution time on computer B? Computer C? Explain the results.

4.17 [5] <§§4.2–4.3> If the clock rates of computers M1 and M2 in Exercise 4.1 are 4 GHz and 6 GHz, respectively, find the clock cycles per instruction (CPI) for program 1 on both computers using the data in Exercises 4.1 and 4.2.

4.18 [5] <§§4.2–4.3> Assuming the CPI for program 2 on each computer in Exercise 4.1 is the same as the CPI for program 1 found in Exercise 4.17, find the instruction count for program 2 running on each computer using the execution times from Exercise 4.1.

4.19 [5] <§4.3> **In More Depth:** Amdahl's Law

4.20 [10] <§4.3> **In More Depth:** Amdahl's Law

4.21 [10] <§4.3> **In More Depth:** Amdahl's Law

4.22 [5] <§4.3> **In More Depth:** Amdahl's Law

4.23 [5] <§4.3> **In More Depth:** Amdahl's Law

4.24 [20] <§4.3> **In More Depth:** Amdahl's Law

4.25 [5] <§4.3> **In More Depth:** Choosing the Right Mean

4.26 [15] <§4.3> **In More Depth:** Choosing the Right Mean

4.27 [3 hours] <§4.3> **In More Depth:** Synthetic Benchmarks: Attempts to Replicate "Typical" Behavior

4.28 [3 hours] <§4.3> **In More Depth:** Synthetic Benchmarks: Attempts to Replicate "Typical" Behavior

4.29 [4 hours] <§4.3> In More Depth: MIPS, MOPS, and Other FLOPS

4.30 [5] <§4.3> In More Depth: MFLOPS as a Performance Metric

4.31 [15] <§4.3> In More Depth: MFLOPS as a Performance metric

4.32 [4 hours] <§4.3> In More Depth: MFLOPS as a Performance Metric

4.33 [5] <§4.3> In More Depth: Embedded Benchmarks

4.34 [20] <§§4.2, 4.3> In More Depth: Using Hardware-Independent Metrics to Try to Predict Performance

4.35 [10] <§§4.2, 4.3> For More Practice: Analyzing a Processor with Floating-Point Implemented in Hardware or Software

4.36 [10] <§§4.2, 4.3> For More Practice: Analyzing a Processor with Floating point implemented in Hardware or Software

4.37 [5] <§§4.2, 4.3> For More Practice: Analyzing a Processor with Floating Point Implemented in Hardware or Software

4.38 [10] <§§4.2, 4.3> For More Practice: Analyzing Enhancements to a Processor

4.39 [5] <§§4.2, 4.3> For More Practice: Analyzing Enhancements to a Processor

4.40 [10] <§§4.2, 4.3> For More Practice: Analyzing Enhancements to a Processor

4.41 [5] <§§4.2, 4.3> For More Practice: Analyzing Enhancements to a Processor

4.42 [5] <§§4.2, 4.3> For More Practice: Analyzing Enhancements to a Processor

4.43 [10] <§§4.2, 4.3> For More Practice: Analyzing Enhancements to a Processor

4.44 [10] <§§4.2, 4.3> For More Practice: Analyzing Enhancements to a Processor

4.45 [5] <§4.3> Assume that multiply instructions take 12 cycles and account for 15% of the instructions in a typical program, and the other 85% of the instructions require an average of 4 cycles for each instruction. What percentage of time does the CPU spend doing multiplication?

4.46 [5] <§4.3> Your hardware engineering team has indicated that it would be possible to reduce the number of cycles required for multiplication to 8 in Exercise

4.45, but this will require a 20% increase in the cycle time. Nothing else will be affected by the change. Should they proceed with the modification?

4.47 [10] <§4.4> Look at the current list of SPEC programs in Figure 4.5 on page 260. Does it include applications that match the ways you typically use your computer? What classes of programs are irrelevant or missing? Why do you think they were or were not included in SPEC? What would have to be done to include/exclude such programs in the next SPEC release?

4.48 [5] <§4.4> If benchmark suites are designed to provide a real-world metric for a specific computing task, explain why benchmarks suites need to be updated.

4.49 [5] <§§4.2, 4.3, 4.4> **In More Depth:** The Difficulty with Kernel Benchmarks

4.50 [15] <§§4.2, 4.3, 4.4> **In More Depth:** The Difficulty with Kernel Benchmarks

4.51 [10] <§§4.1–4.5> Consider the following hypothetical news release:

> *"The company will unveil the industry's first 5 GHz version of the chip, which offers a 25% performance boost over the company's former speed champ, which runs at 4 GHz. The new chip can be plugged into system boards for the older original chip (which ran at 1 GHz) to provide a 70% performance boost."*

Comment on the definition (or definitions) of performance that you believe the company used. Do you think the news release is misleading?

4.52 [indefinite] <§§4.1–4.5> Collect a set of articles that you believe contain incorrect analyses of performance or use misleading performance metrics to try to persuade readers. For example, an article in the *New York Times* (April 20, 1994, p. D1) described a video game player "that will surpass the computing power of even the most powerful personal computers" and presented the following chart to support the argument that "video game computers may be the supercomputers of tomorrow":

Computer	Approximate number of instructions per second	Price
1975 IBM mainframe	10,000,000	$10,000,000
1976 Cray-1	160,000,000	$20,000,000
1979 Digital VAX	1,000,000	$200,000
1981 IBM PC	250,000	$3,000
1984 Sun 2	1,000,000	$10,000
1994 Pentium-chip PC	66,000,000	$3,000
1995 Sony PCX video game	500,000,000	$500
1995 Microunity set-top	1,000,000,000	$500

The article never discussed how the nature of the instructions should impact the definition of "powerful." For each article you collect, describe why you think it is misleading or incorrect. Good places to look for material include the business or technology sections of newspapers, magazines (both articles and ads), and the Internet (newsgroups and the Web).

Answers to Check Yourself

§4.1, page 246: 1. a: both, b: latency, c: neither. 2. 7 sec.
§4.2, page 253: 6.
§4.3, page 258: 1. F, F, F, T. 2. T, F, F, T
§4.4, page 265: b.
§4.5, page 270: a. Computer A. b. Computer B.

Computers in the Real World

Moving People Faster and More Safely

Problem: Find ways to help transport people quickly while maintaining safety, comfort, and efficiency.

Solution: For more than 20 years, computers have played an increasingly important role in the control of transportation systems including planes, trains, automobiles, and even ships. Essentially all modern transportation systems rely on computers to enhance their safety, comfort, and efficiency. Computers also play a critical role in improving fuel consumption and reducing air pollution. In these two pages, we examine some of the uses of computers in trains and automobiles. In designing pipelines we must prevent hazards, at all costs, and we try to avoid pipeline delays. Computer-controlled trains try to do the same: hazards are life-threatening and can never be permitted, and delays are to be avoided!

The French TGV (*Train à Grande Vitesse*) is one of the fastest train systems in the world, with a typical top speed of 300 km/hour. Traditionally, trains were controlled by an engineer, using a set of track-side lights and signals that tell the engineer to proceed, slow down, or stop. At 300 km/hour, it is difficult to read such signals, and easy to miss one completely, leading to a possible disaster. In addition, some tracks, such as those containing tight curves, can be unsafe at the higher speeds; since the TGV was designed to run on existing track, finding a fail-safe method to communicate track conditions was critical.

The TGV designers solved this problem by a clever signaling system, called TVM (*Transmission Voie-Machine*), that runs through the rails and is picked up by antennas in the locomotive. The track is divided into blocks, which are typically about 1.5 km in length. Shorter blocks are used when track conditions change quickly or where a higher degree of safety is critical, such as in the Chunnel, where the block length is about one-tenth as long. Transmitters at the beginning of each block are used to communicate instructions to the cab, where they are carried out by an engineer; a computer also watches the communications, and implements the commands if the engineer fails to.



The Eurostar TGV train in Nice, France.

One challenge is that the stopping distance for the TGV is nominally four blocks (somewhat shorter in an absolute emergency). The time to travel four blocks is 1.2 minutes, and the newest signaling system trains are run at a 3 minute headway, even in fog! So the system must monitor the presence of all trains and ensure the most important property: only one train ever occupies a block of track at the same time! The system constantly communicates the maximum safe speed for the current block, improving performance and safety.

The TVM system has been built with extensive attention to safety, which means heavy use of redundancy to ensure the operation of the system in the face of failure of a component. The failure rate of TVM has been estimated at less than 1 failure in a million years. This attention to safety has paid off: in over two decades of service, there have been no fatal accidents caused by a TVM failure

The images on the previous page and below show the Eurostar TGV and the cab and operator's seat.

Computers have also played a key role in making cars safer, more efficient, and less polluting. The modern automobile has dozens of microprocessors controlling everything from braking, to ignition, to air bag deployment.

In the area of safety, airbags and antilock brakes have been two of the most important innovations since the seat belt. Antilock brakes preserve the ability to steer during the severe braking that might occur in emergency conditions. By detecting wheel lockup and alternately applying and releasing the brakes under computer control, an antilock brake system can avoid wheel lockup.

Airbags use a force sensor to detect severe deacceleration, which occurs during a collision. The airbags are deployed by a computer that reads the sensor. The new generation of airbags uses a two-stage deployment: when the deacceleration indicates a collision of moderate severity, the airbag inflates more slowly, reducing the possibility of passenger injury from rapid deployment in a moderate collision. Reliability of these safety systems has been enhanced by a computer-controlled test that is run every time the vehicle is started.

Modern ignition systems in automobiles aim to enhance mileage while reducing pollution. Happily these two goals are doubly congruent: enhancing mileage reduces pollution through the use of less fuel, and mileage is enhanced by more effective combustion, which also reduces the emission of partially combusted fuel. Computers control the injection of fuel, the amount of air injected, and the spark timing, which must change as the engine runs faster. Careful control of these elements over the full operating range from 1000 to 6000 RPM and different temperature conditions has helped improve mileage and reduce pollution.



The interior of a Eurostar TGV cab.

To learn more see these references on the library

"An investigation of the Therac-25 accidents," Nancy G. Leveson and Clark S. Turner. *IEEE Computer,* 26(7): 18–41, July 1993.

5

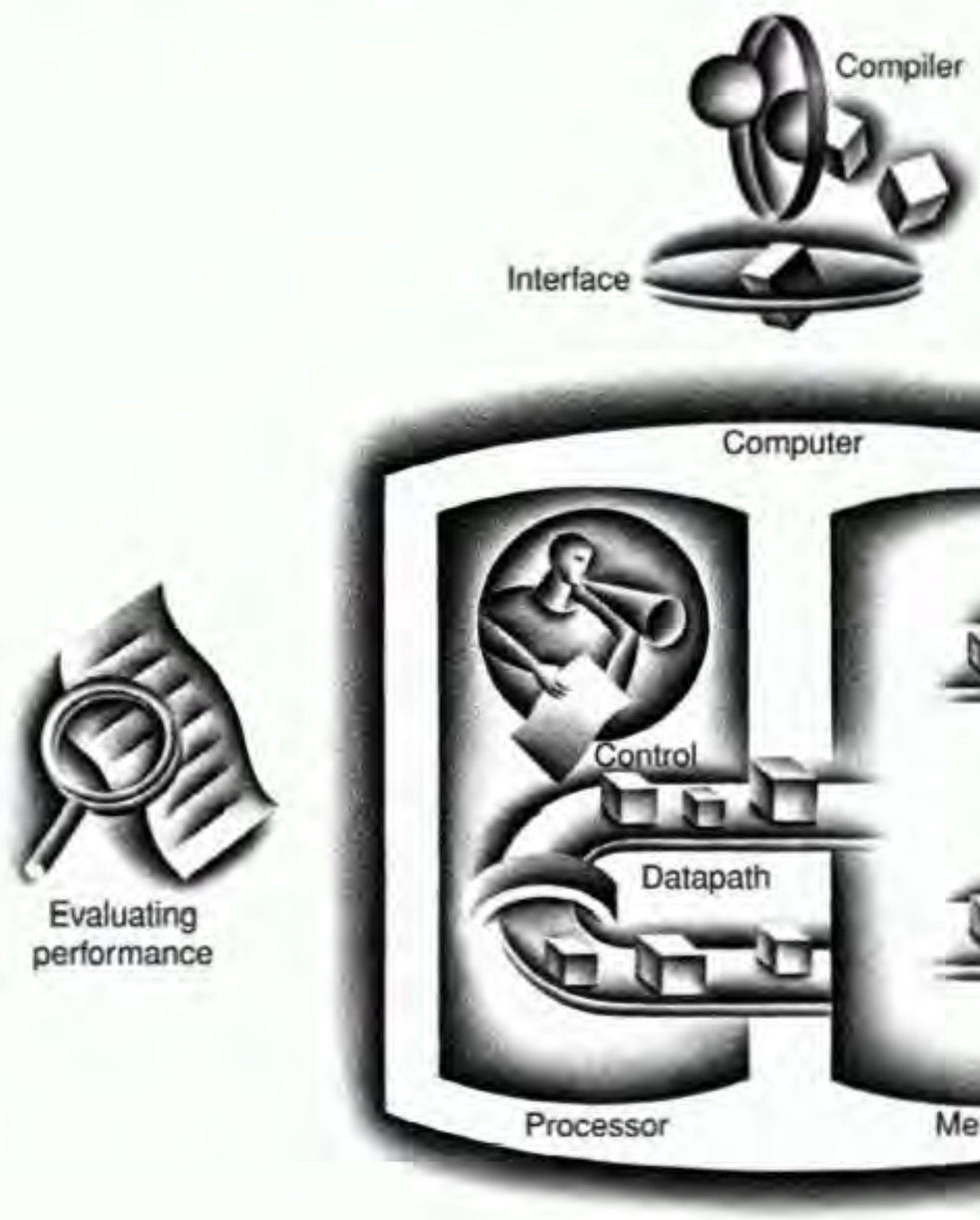
The Processor: Datapath and Control

In a major matter,
no details are small.

French Proverb

5.1 **Introduction** 284
5.2 **Logic Design Conventions** 289
5.3 **Building a Datapath** 292
5.4 **A Simple Implementation Scheme** 300
5.5 **A Multicycle Implementation** 318
5.6 **Exceptions** 340
5.7 **Microprogramming: Simplifying Control Design** 346
5.8 **An Introduction to Digital Design Using a Hardware Design Language** 346
5.9 **Real Stuff: The Organization of Recent Pentium Implementations** 347
5.10 **Fallacies and Pitfalls** 350
5.11 **Concluding Remarks** 352
5.12 **Historical Perspective and Further Reading** 353
5.13 **Exercises** 354

The Five Classic Components of a Computer



5.1 Introduction

In Chapter 4, we saw that the performance of a machine was determined by three key factors: instruction count, clock cycle time, and clock cycles per instruction (CPI). The compiler and the instruction set architecture, which we examined in Chapters 2 and 3, determine the instruction count required for a given program. However, both the clock cycle time and the number of clock cycles per instruction are determined by the implementation of the processor. In this chapter, we construct the datapath and control unit for two different implementations of the MIPS instruction set.

This chapter contains an explanation of the principles and techniques used in implementing a processor, starting with a highly abstract and simplified overview in this section, followed by sections that build up a datapath and construct a simple version of a processor sufficient to implement instructions sets like MIPS, and finally, developing the concepts necessary to implement more complex instructions sets, like the IA-32.

For the reader interested in understanding the high-level interpretation of instructions and its impact on program performance, this initial section provides enough background to understand these concepts as well as the basic concepts of pipelining, which are explained in Section 6.1 of the next chapter.

For those readers desiring an understanding of how hardware implements instructions, Sections 5.3 and 5.4 are all the additional material that is needed. Furthermore, these two sections are sufficient to understand all the material in Chapter 6 on pipelining. Only those readers with an interest in hardware design should go further.

The remaining sections of this chapter cover how modern hardware—including more complex processors such as the Intel Pentium series—is usually implemented. The basic principles of finite state control are explained, and different methods of implementation, including microprogramming, are examined. For the reader interested in understanding the processor and its performance in more depth, Sections 5.4 and 5.5 will be useful. For readers with an interest in modern hardware design, Section 5.7 covers microprogramming, a technique used to implement more complex control such as that present in IA-32 processors, and Section 5.8 describes how hardware design languages and CAD tools are used to implement hardware.

A Basic MIPS Implementation

We will be examining an implementation that includes a subset of the core MIPS instruction set:

- The memory-reference instructions load word (`lw`) and store word (`sw`)
- The arithmetic-logical instructions `add`, `sub`, `and`, `or`, and `slt`
- The instructions branch equal (`beq`) and jump (`j`), which we add last

This subset does not include all the integer instructions (for example, shift, multiply, and divide are missing), nor does it include any floating-point instructions. However, the key principles used in creating a datapath and designing the control will be illustrated. The implementation of the remaining instructions is similar.

In examining the implementation, we will have the opportunity to see how the instruction set architecture determines many aspects of the implementation, and how the choice of various implementation strategies affects the clock rate and CPI for the machine. Many of the key design principles introduced in Chapter 4 can be illustrated by looking at the implementation, such as the guidelines *Make the common case fast* and *Simplicity favors regularity.* In addition, most concepts used to implement the MIPS subset in this chapter and the next are the same basic ideas that are used to construct a broad spectrum of computers, from high-performance servers to general-purpose microprocessors to embedded processors, which are used increasingly in products ranging from VCRs to automobiles.

An Overview of the Implementation

In Chapters 2 and 3, we looked at the core MIPS instructions, including the integer arithmetic-logical instructions, the memory-reference instructions, and the branch instructions. Much of what needs to be done to implement these instructions is the same, independent of the exact class of instruction. For every instruction, the first two steps are identical:

1. Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require that we read two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact opcode.

Even across different instruction classes there are some similarities. For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and branches for comparison. As we can see, the simplicity and regularity of the instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

After using the ALU, the actions required to complete various instruction classes differ. A memory-reference instruction will need to access the memory either to write data for a store or read data for a load. An arithmetic-logical instruction must write the data from the ALU back into a register. Lastly, for a branch instruction, we may need to change the next instruction address based on the comparison; otherwise the PC should be incremented by 4 to get the address of the next instruction.

Figure 5.1 shows the high-level view of a MIPS implementation, focusing on the various functional units and their interconnection. Although this figure shows most of the flow of data through the processor, it omits two important aspects of instruction execution.

First, in several places, Figure 5.1 shows data going to a particular unit as coming from two different sources. For example, the value written into the PC can come from one of two adders, and the data written into the register file can come from either the ALU or the data memory. In practice, these data lines cannot simply be wired together; we must add an element that chooses from among the multiple sources and steers one of those sources to its destination. This selection is commonly done with a device called a *multiplexor*, although this device might better be called a *data selector*. The multiplexor, which is described in detail in Appendix B, selects from among several inputs based on the setting of its control lines. The control lines are set based primarily on information taken from the instruction being executed.

Second, several of the units must be controlled depending on the type of insrtruction. For example, the data memory must read on a load and write on a store. The register file must be written on a load and an arithmetic-logical instruction. And, of course, the ALU must perform one of several operations, as we saw in Chapter 3. (Appendix B describes the detailed logic design of the ALU.) Like the muxes, these operations are directed by control lines that are set on the basis of various fields in the instruction.

Figure 5.2 shows the datapath of Figure 5.1 with the three required multiplexors added, as well as control lines for the major functional units. A control unit that has the instruction as an input is used to determine how to set the control lines for the functional units and two of the multiplexors. The third multiplexor, which determines whether PC + 4 or the branch destination address is written

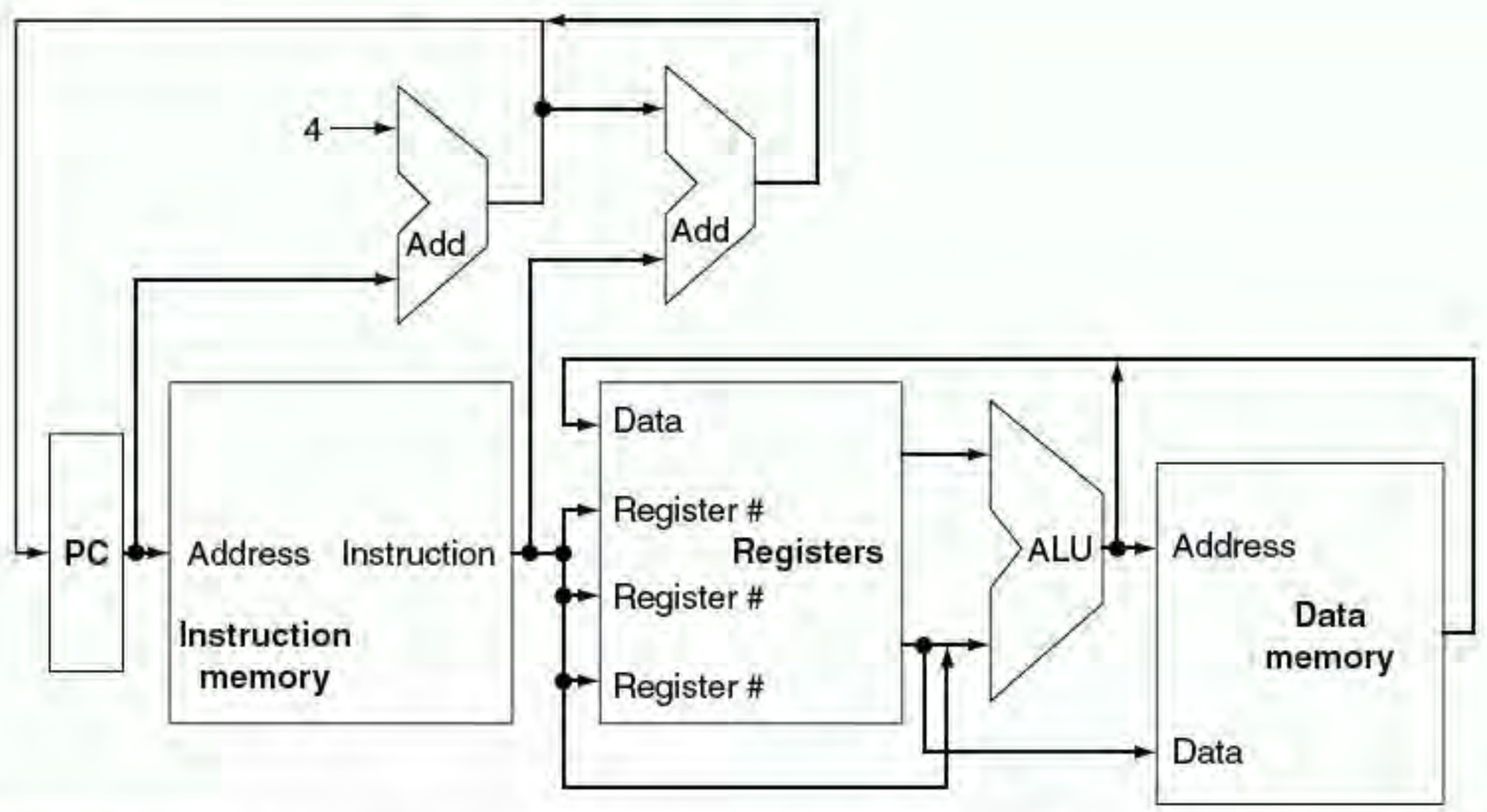


FIGURE 5.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes from either the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

into the PC, is set based on the zero output of the ALU, which is used to perform the comparison of a `beq` instruction. The regularity and simplicity of the MIPS instruction set means that a simple decoding process can be used to determine how to set the control lines.

In the remainder of the chapter, we refine this view to fill in the details, which requires that we add further functional units, increase the number of connections between units, and, of course, add a control unit to control what actions are taken for different instruction classes. Sections 5.3 and 5.4 describe a simple implementation that uses a single long clock cycle for every instruction and follows the general form of Figures 5.1 and 5.2. In this first design, every instruction begins execution on one clock edge and completes execution on the next clock edge.

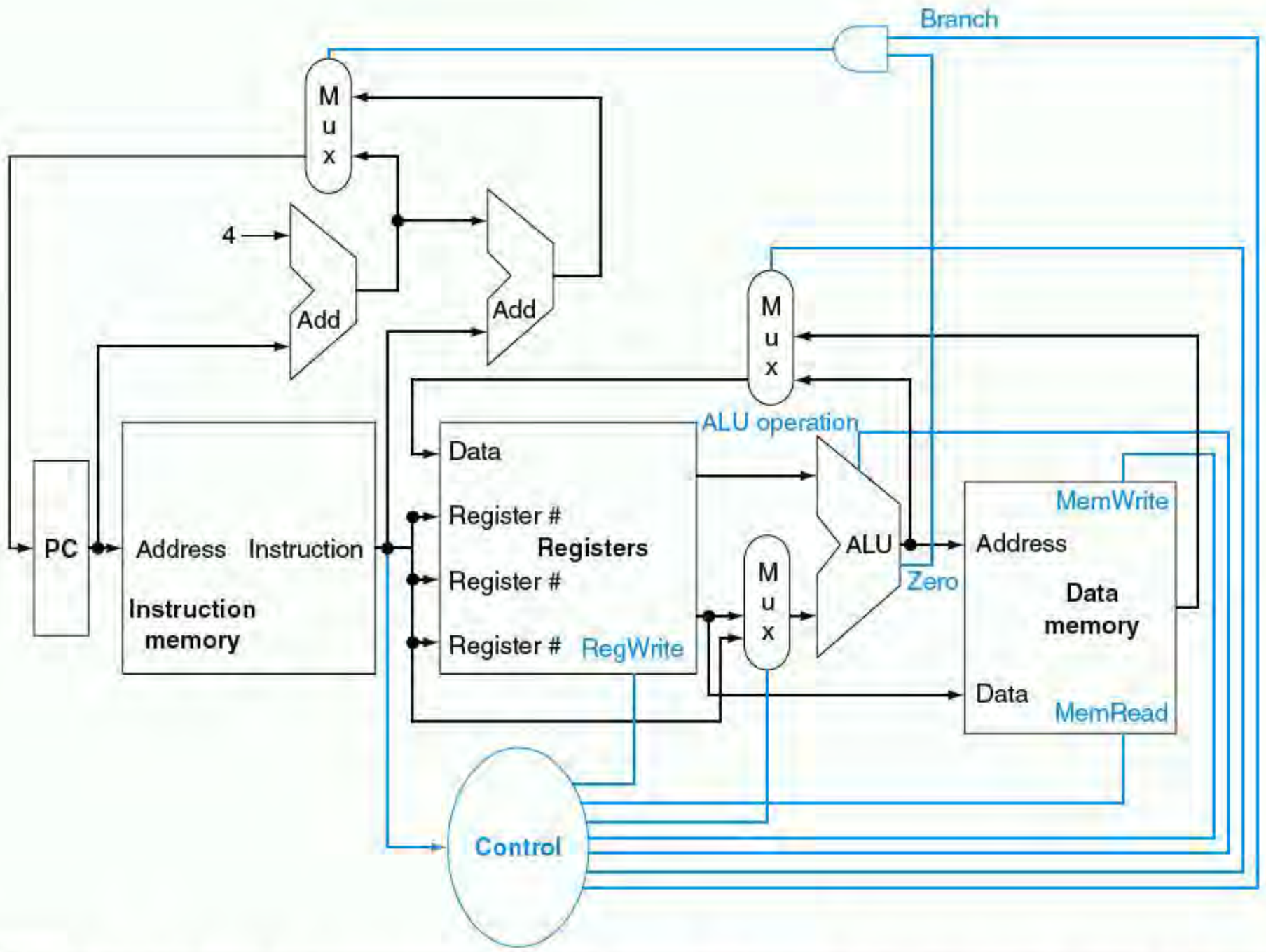


FIGURE 5.2 The basic implementation of the MIPS subset including the necessary multiplexors and control lines. The top multiplexor controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that "ands" together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The multiplexor whose output returns to the register file is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for a nonimmediate arithmetic-logical instruction) or from the offset field of the instruction (for an immediate operation, a load or store, or a branch). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.

While easier to understand, this approach is not practical, since it would be slower than an implementation that allows different instruction classes to take different numbers of clock cycles, each of which could be much shorter. After designing the control for this simple machine, we will look at an implementation that uses multiple clock cycles for each instruction. This multicycle design is used

when we discuss more advanced control concepts, handling exceptions, and the use of hardware design languages in Sections 5.5 through 5.8.

Check Yourself

The single-cycle datapath conceptually described in this section *must* have separate instruction and data memories because

1. the format of data and instructions is different in MIPS and hence different memories are needed
2. having separate memories is less expensive
3. the processor operates in one cycle and cannot use a single-ported memory for two different accesses within that cycle

5.2 Logic Design Conventions

To discuss the design of a machine, we must decide how the logic implementing the machine will operate and how the machine is clocked. This section reviews a few key ideas in digital logic that we will use extensively in this chapter. If you have little or no background in digital logic, you will find it helpful to read through Appendix B before continuing.

The functional units in the MIPS implementation consist of two different types of logic elements: elements that operate on data values and elements that contain state. The elements that operate on data values are all *combinational*, which means that their outputs depend only on the current inputs. Given the same input, a combinational element always produces the same output. The ALU shown in Figure 5.1 and discussed in Chapter 3 and Appendix B is a combinational element. Given a set of inputs, it always produces the same output because it has no internal storage.

Other elements in the design are not combinational, but instead contain *state*. An element contains state if it has some internal storage. We call these elements **state elements** because, if we pulled the plug on the machine, we could restart it by loading the state elements with the values they contained before we pulled the plug. Furthermore, if we saved and restored the state elements, it would be as if the machine had never lost power. Thus, these state elements completely characterize the machine. In Figure 5.1, the instruction and data memories as well as the registers are all examples of state elements.

state element A memory element.

A state element has at least two inputs and one output. The required inputs are the data value to be written into the element and the clock, which determines

when the data value is written. The output from a state element provides the value that was written in an earlier clock cycle. For example, one of the logically simplest state elements is a D-type flip-flop (see Appendix B), which has exactly these two inputs (a value and a clock) and one output. In addition to flip-flops, our MIPS implementation also uses two other types of state elements: memories and registers, both of which appear in Figure 5.1. The clock is used to determine when the state element should be written; a state element can be read at any time.

Logic components that contain state are also called *sequential* because their outputs depend on both their inputs and the contents of the internal state. For example, the output from the functional unit representing the registers depends both on the register numbers supplied and on what was written into the registers previously. The operation of both the combinational and sequential elements and their construction are discussed in more detail in Appendix B.

We will use the word *asserted* to indicate a signal that is logically high and *assert* to specify that a signal should be driven logically high, and *deassert* or *deasserted* to represent logical low.

Clocking Methodology

clocking methodology The approach used to determine when data is valid and stable relative to the clock.

A **clocking methodology** defines when signals can be read and when they can be written. It is important to specify the timing of reads and writes because, if a signal is written at the same time it is read, the value of the read could correspond to the old value, the newly written value, or even some mix of the two! Needless to say, computer designs cannot tolerate such unpredictability. A clocking methodology is designed to prevent this circumstance.

edge-triggered clocking A clocking scheme in which all state changes occur on a clock edge.

For simplicity, we will assume an **edge-triggered** clocking methodology. An edge-triggered clocking methodology means that any values stored in a sequential logic element are updated only on a clock edge. Because only state elements can store a data value, any collection of combinational logic must have its inputs coming from a set of state elements and its outputs written into a set of state elements. The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle.

Figure 5.3 shows the two state elements surrounding a block of combinational logic, which operates in a single clock cycle: All signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle. The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

control signal A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a data signal, which contains information that is operated on by a functional unit.

For simplicity, we do not show a write **control signal** when a state element is written on every active clock edge. In contrast, if a state element is not updated on every clock, then an explicit write control signal is required. Both the clock signal and the write control signal are inputs, and the state element is changed only when the write control signal is asserted and a clock edge occurs.

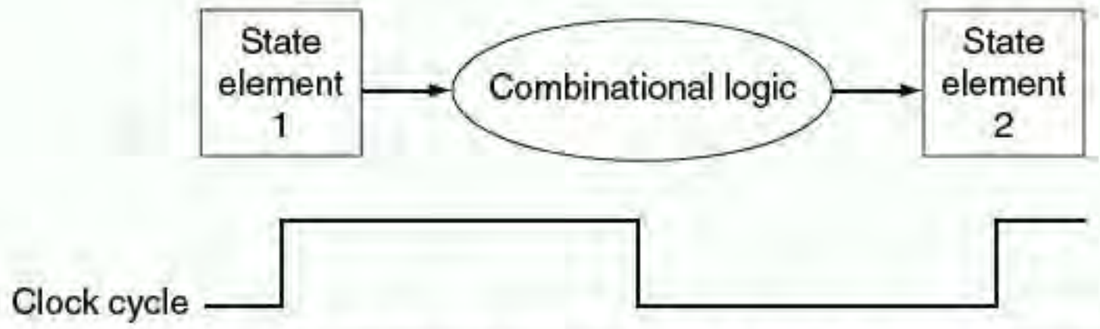


FIGURE 5.3 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements, including memory, are assumed to be edge-triggered.

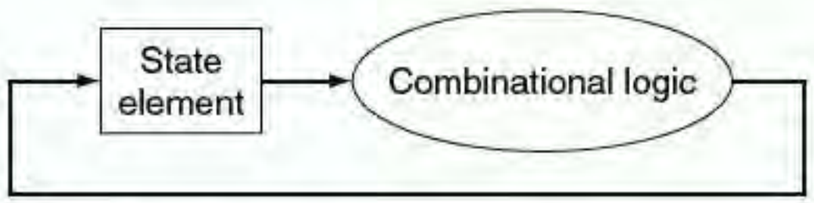


FIGURE 5.4 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values. Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. Feedback cannot occur within 1 clock cycle because of the edge-triggered update of the state element. If feedback were possible, this design could not work properly. Our designs in this chapter and the next rely on the edge-triggered timing methodology and structures like the one shown in this figure.

An edge-triggered methodology allows us to read the contents of a register, send the value through some combinational logic, and write that register in the same clock cycle, as shown in Figure 5.4. It doesn't matter whether we assume that all writes take place on the rising clock edge or on the falling clock edge, since the inputs to the combinational logic block cannot change except on the chosen clock edge. With an edge-triggered timing methodology, there is *no* feedback within a single clock cycle, and the logic in Figure 5.4 works correctly. In Appendix B we briefly discuss additional timing constraints (such as setup and hold times) as well as other timing methodologies.

Nearly all of these state and logic elements will have inputs and outputs that are 32 bits wide, since that is the width of most of the data handled by the processor. We will make it clear whenever a unit has an input or output that is other than 32 bits in width. The figures will indicate *buses*, which are signals wider than 1 bit, with thicker lines. At times we will want to combine several buses to form a wider bus; for example, we may want to obtain a 32-bit bus by combining two 16-bit

buses. In such cases, labels on the bus lines will make it clear that we are concatenating buses to form a wider bus. Arrows are also added to help clarify the direction of the flow of data between elements. Finally, color indicates a control signal as opposed to a signal that carries data; this distinction will become clearer as we proceed through this chapter.

Check Yourself True or false: Because the register file is both read and written on the same clock cycle, any MIPS datapath using edge-triggered writes must have more than one copy of the register file.

5.3 Building a Datapath

A reasonable way to start a datapath design is to examine the major components required to execute each class of MIPS instruction. Let's start by looking at which **datapath elements** each instruction needs. When we show the datapath elements, we will also show their control signals.

datapath element A functional unit used to operate on or hold data within a processor. In the MIPS implementation the datapath elements include the instruction and data memories, the register file, the arithmetic logic unit (ALU), and adders.

program counter (PC) The register containing the address of the instruction in the program being executed

Figure 5.5 shows the first element we need: a memory unit to store the instructions of a program and supply instructions given an address. Figure 5.5 also shows a register, which we call the **program counter (PC)**, that is used to hold the address of the current instruction. Lastly, we will need an adder to increment the PC to the address of the next instruction. This adder, which is combinational, can be built from the ALU we described in Chapter 3 and designed in detail in Appendix B, simply by wiring the control lines so that the control always specifies an add operation. We will draw such an ALU with the label *Add*, as in Figure 5.5, to indicate that it has been permanently made an adder and cannot perform the other ALU functions.

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later. Figure 5.6 shows how the three elements from Figure 5.5 are combined to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

Now let's consider the R-format instructions (see Figure 2.7 on page 67). They all read two registers, perform an ALU operation on the contents of the registers, and write the result. We call these instructions either *R-type instructions* or *arithmetic-logical instructions* (since they perform arithmetic or logical operations). This instruction class includes `add`, `sub`, `and`, `or`, and `slt`, which were intro-

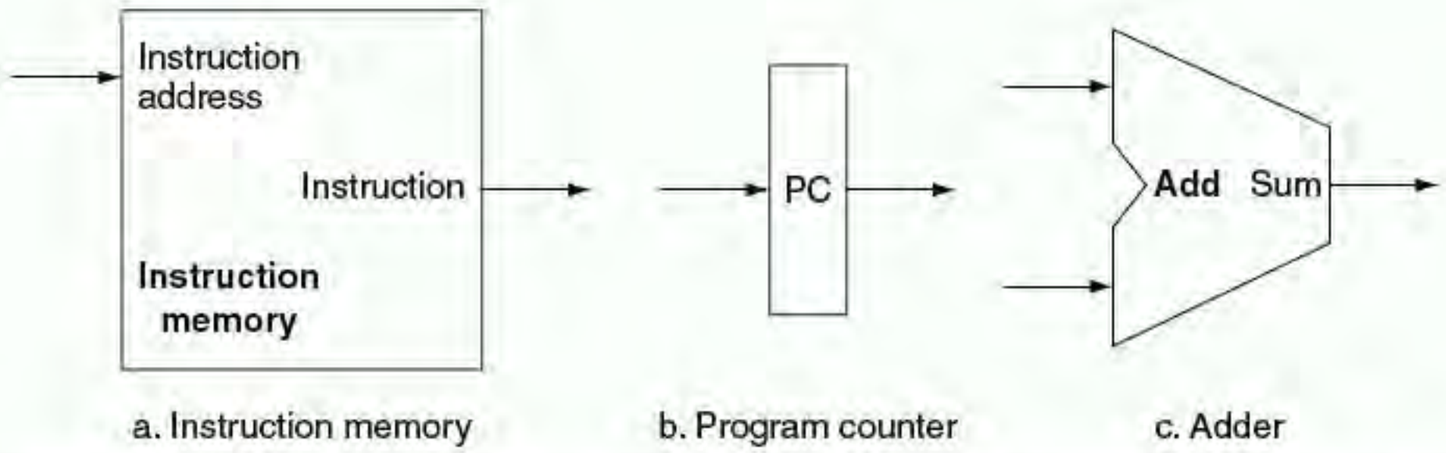


FIGURE 5.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that will be written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always perform an add of its two 32-bit inputs and place the result on its output.

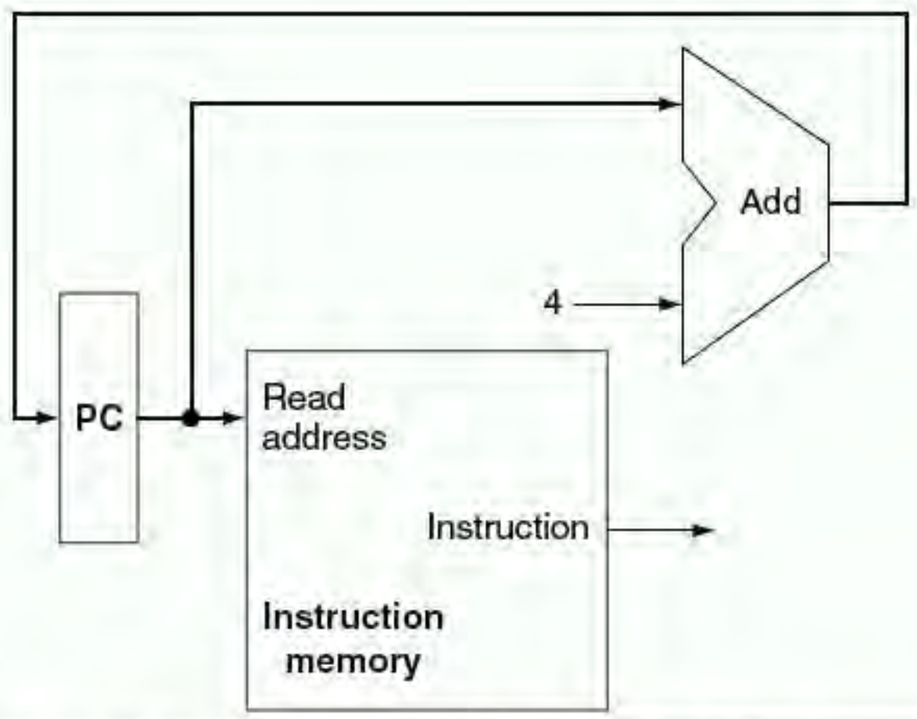


FIGURE 5.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

duced in Chapter 2. Recall that a typical instance of such an instruction is `add $t1,$t2,$t3`, which reads `$t2` and `$t3` and writes `$t1`.

The processor's 32 general-purpose registers are stored in a structure called a **register file**. A register file is a collection of registers in which any register can be

register file A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.

read or written by specifying the number of the register in the file. The register file contains the register state of the machine. In addition, we will need an ALU to operate on the values read from the registers.

Because the R-format instructions have three register operands, we will need to read two data words from the register file and write one data word into the register file for each instruction. For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers. To write a data word, we will need two inputs: one to specify the *register number* to be written and one to supply the *data* to be written into the register. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge. Thus, we need a total of four inputs (three for register numbers and one for data) and two outputs (both for data), as shown in Figure 5.7. The register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$), whereas the data input and two data output buses are each 32 bits wide.

Figure 5.7 shows the ALU, which takes two 32-bit inputs and produces a 32-bit result, as well as a 1-bit signal if the result is 0. The four-bit control signal of the ALU is described in detail in Appendix B; we will review the ALU control shortly when we need to know how to set it.

Next, consider the MIPS load word and store word instructions, which have the general form `lw $t1,offset_value($t2)` or `sw $t1,offset_value ($t2)`. These instructions compute a memory address by adding the base register, which is `$t2`, to the 16-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in `$t1`. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is `$t1`. Thus, we will need both the register file and the ALU from Figure 5.7.

sign-extend To increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item.

In addition, we will need a unit to **sign-extend** the 16-bit offset field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to. The data memory must be written on store instructions; hence, it has both read and write control signals, an address input, as well as an input for the data to be written into memory. Figure 5.8 shows these two elements.

branch target address The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branch.

The `beq` instruction has three operands, two registers that are compared for equality, and a 16-bit offset used to compute the **branch target address** relative to the branch instruction address. Its form is `beq $t1,$t2,offset`. To implement this instruction, we must compute the branch target address by adding the sign-extended offset field of the instruction to the PC. There are two details in the definition of branch instructions (see Chapter 2) to which we must pay attention:

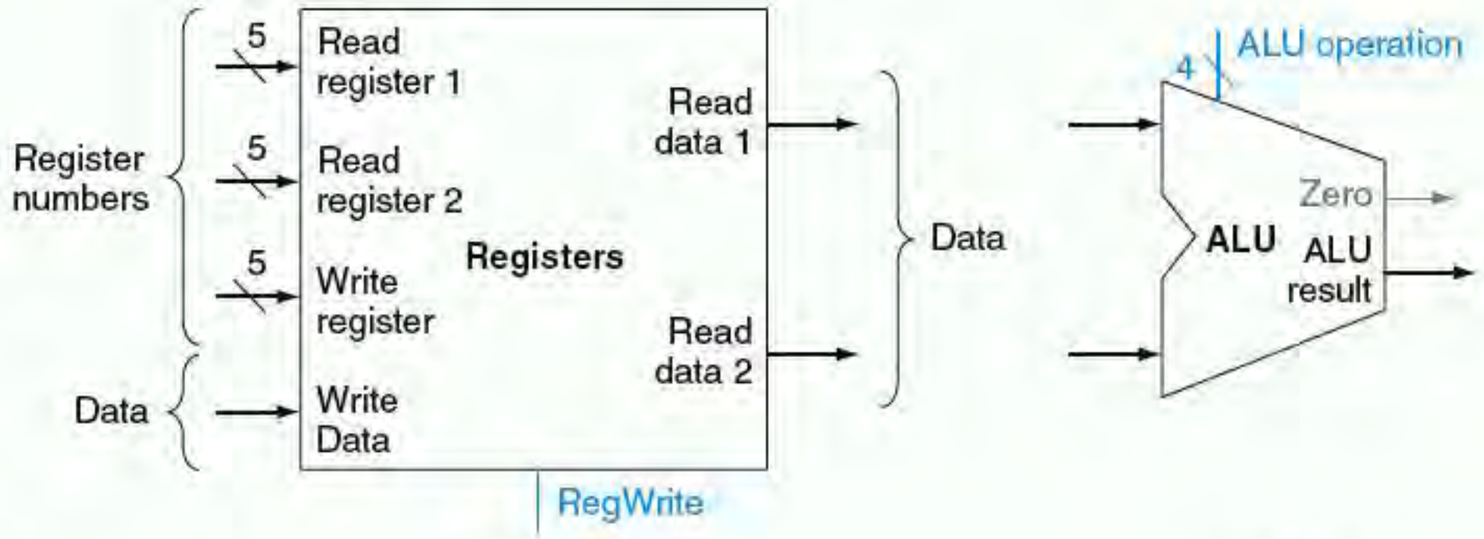


FIGURE 5.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section B.8 of Appendix B. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in Appendix B. We will use the Zero detection output of the ALU shortly to implement branches. The overflow output will not be needed until Section 5.6, when we discuss exceptions; we omit it until then.

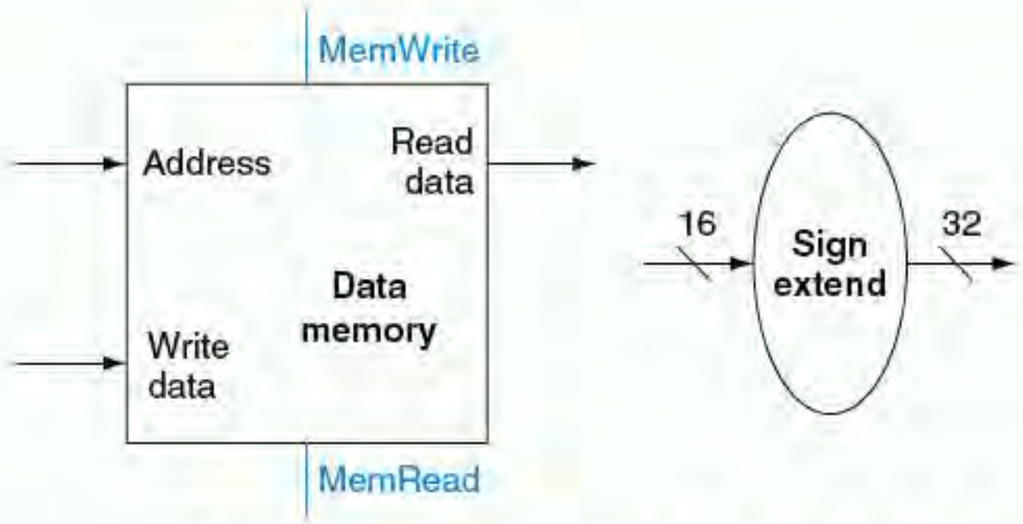
- The instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch. Since we compute PC + 4 (the address of the next instruction) in the instruction fetch datapath, it is easy to use this value as the base for computing the branch target address.
- The architecture also states that the offset field is shifted left 2 bits so that it is a word offset; this shift increases the effective range of the offset field by a factor of four.

To deal with the latter complication, we will need to shift the offset field by two.

In addition to computing the branch target address, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address. When the condition is true (i.e., the operands are equal), the branch target address becomes the new PC, and we say that the **branch** is **taken**. If the operands are not equal, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the **branch** is **not taken**.

branch taken A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional branches are taken branches.

branch not taken A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.



a. Data memory unit b. Sign-extension unit

FIGURE 5.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 5.7, are the data memory unit and the sign extension unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in Chapter 7. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output (see Chapter 3). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See Section B.8 of Appendix B for a further discussion of how real memory chips work.

Thus, the branch datapath must do two operations: compute the branch target address and compare the register contents. (Branches also affect the instruction fetch portion of the datapath, as we will deal with shortly.) Because of the complexity of handling branches, we show the structure of the datapath segment that handles branches in Figure 5.9. To compute the branch target address, the branch datapath includes a sign extension unit, just like that in Figure 5.8, and an adder. To perform the compare, we need to use the register file shown in Figure 5.7 to supply the two register operands (although we will not need to write into the register file). In addition, the comparison can be done using the ALU we designed in Appendix B. Since that ALU provides an output signal that indicates whether the result was 0, we can send the two register operands to the ALU with the control set to do a subtract. If the Zero signal out of the ALU unit is asserted, we know that the two values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equal test of branches. Later, we will show exactly how to connect the control signals of the ALU for use in the datapath.

The jump instruction operates by replacing the lower 28 bits of the PC with the lower 26 bits of the instruction shifted left by 2 bits. This shift is accomplished simply by concatenating 00 to the jump offset, as described in Chapter 2.

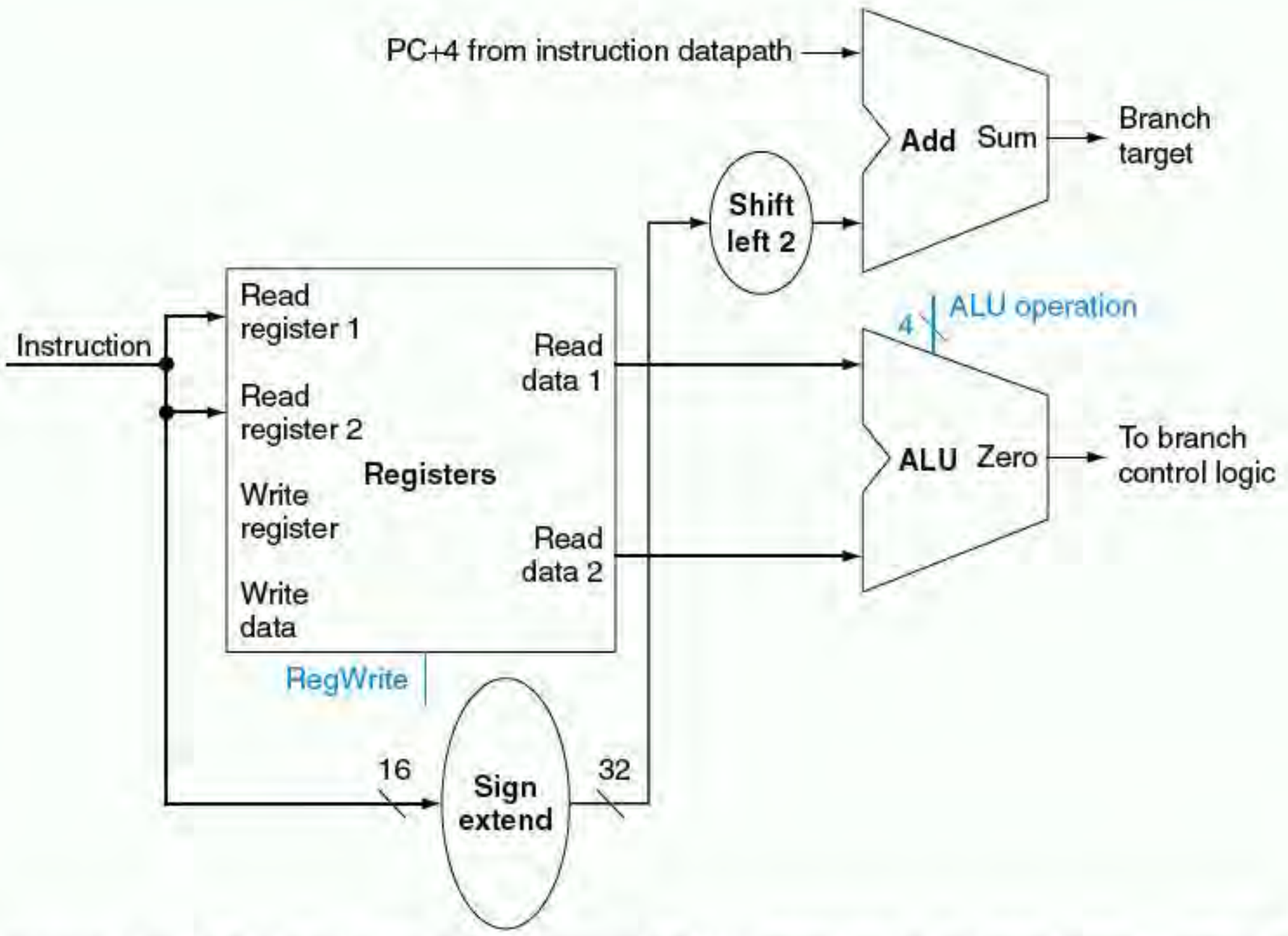


FIGURE 5.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. The unit labeled *Shift left 2* is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended offset field; no actual shift hardware is needed, since the amount of the "shift" is constant. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only "sign bits." Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

Elaboration: In the MIPS instruction set, branches are **delayed**, meaning that the instruction immediately following the branch is always executed, *independent* of whether the branch condition is true or false. When the condition is false, the execution looks like a normal branch. When the condition is true, a delayed branch first executes the instruction immediately following the branch in sequential instruction order before jumping to the specified branch target address. The motivation for delayed branches arises from how pipelining affects branches (see Section 6.6). For simplicity, we ignore delayed branches in this chapter and implement a nondelayed `beq` instruction.

delayed branch A type of branch where the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

Creating a Single Datapath

Now that we have examined the datapath components needed for the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation. The simplest datapath might attempt to execute all instructions in one clock cycle. This means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated. We therefore need a memory for instructions separate from one for data. Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows.

To share a datapath element between two different instruction classes, we may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.

EXAMPLE

Building a Datapath

The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The key differences are the following:

- The arithmetic-logical instructions use the ALU with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the sign-extended 16-bit offset field from the instruction.
- The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

Show how to build a datapath for the operational portion of the memory reference and arithmetic-logical instructions that uses a single register file and a single ALU to handle both types of instructions, adding any necessary multiplexors.

ANSWER

To create a datapath with only a single register file and a single ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus, one multiplexor is placed at the ALU input and another at the data input to the register file. Figure 5.10 shows the operational portion of the combined datapath.

Now we can combine all the pieces to make a simple datapath for the MIPS architecture by adding the datapath for instruction fetch (Figure 5.6 on page 293),

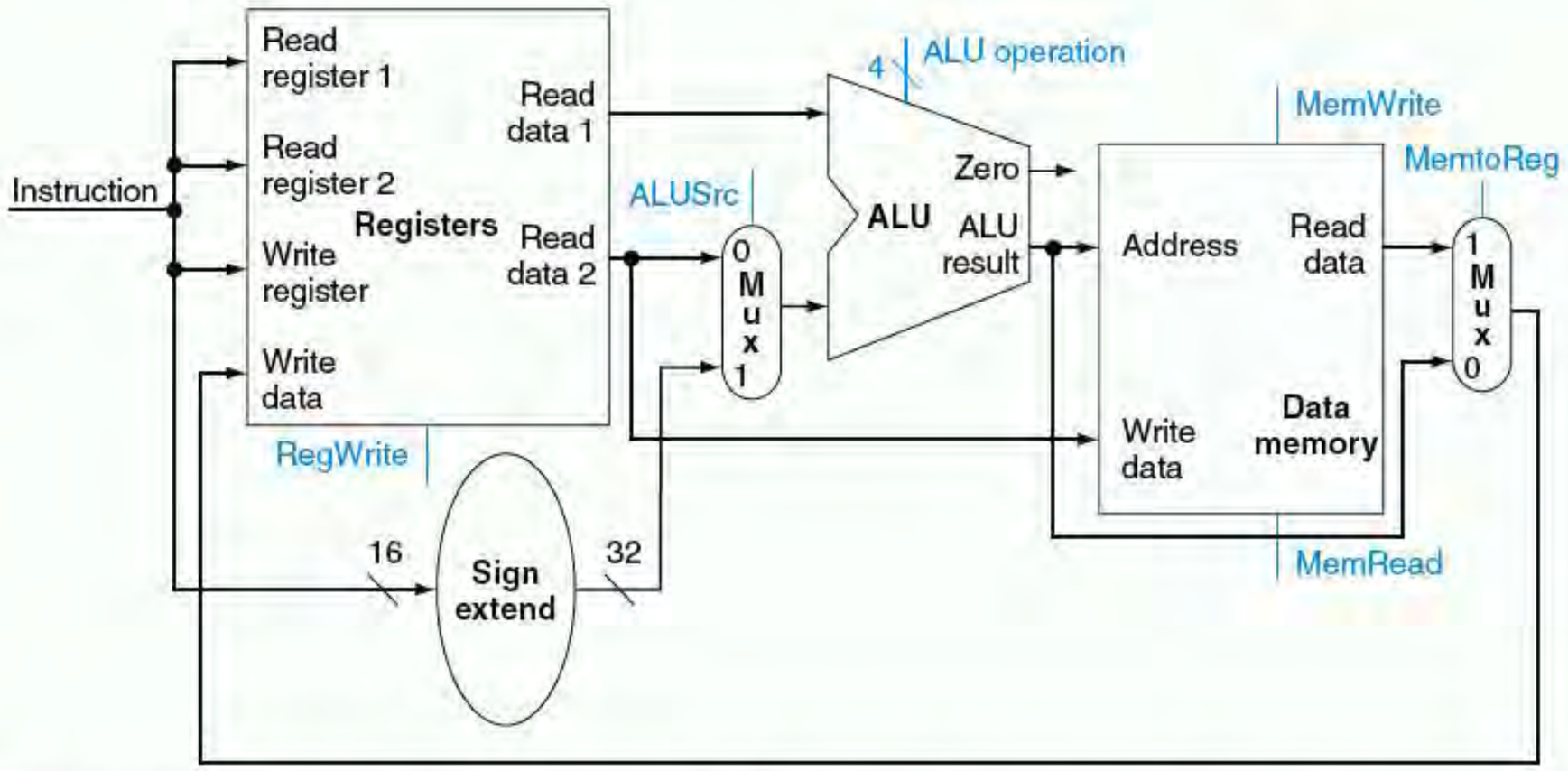


FIGURE 5.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 5.7 and 5.8 by adding multiplexors. Two multiplexors are needed, as described as in the example.

the datapath from R-type and memory instructions (Figure 5.10 on page 299), and the datapath for branches (Figure 5.9 on page 297). Figure 5.11 shows the datapath we obtain by composing the separate pieces. The branch instruction uses the main ALU for comparison of the register operands, so we must keep the adder in Figure 5.9 for computing the branch target address. An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC.

Now that we have completed this simple datapath, we can add the control unit. The control unit must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control. The ALU control is different in a number of ways, and it will be useful to design it first before we design the rest of the control unit.

Check Yourself

Which of the following is correct for a load instruction?

a. MemtoReg should be set to cause the data from memory to be sent to the register file.

b. MemtoReg should be set to cause the correct register destination to be sent to the register file.

c. We do not care about the setting of MemtoReg.

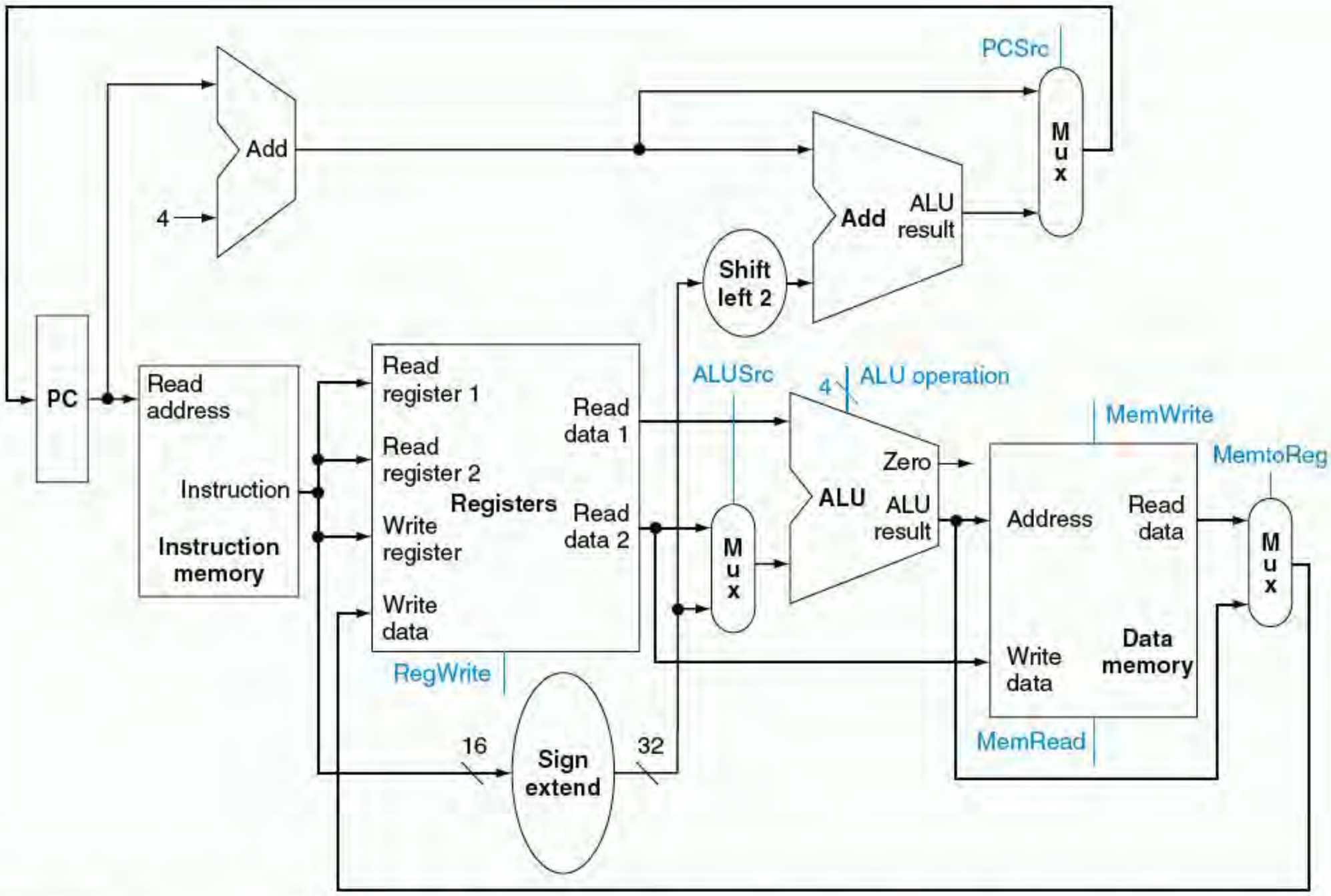


FIGURE 5.11 The simple datapath for the MIPS architecture combines the elements required by different instruction classes. This datapath can execute the basic instructions (load/store word, ALU operations, and branches) in a single clock cycle. An additional multiplexor is needed to integrate branches. The support for jumps will be added later.

5.4 A Simple Implementation Scheme

In this section, we look at what might be thought of as the simplest possible implementation of our MIPS subset. We build this simple implementation using the datapath of the last section and adding a simple control function. This simple implementation covers load word (`lw`), store word (`sw`), branch equal (`beq`), and the arithmetic-logical instructions `add`, `sub`, `and`, `or`, and `set on less than`. We will later enhance the design to include a jump instruction (`j`).

The ALU Control

As can be seen in Appendix B, the ALU has four control inputs. These bits were not encoded; hence, only 6 of the possible 16 possible input combinations are used in this subset. The MIPS ALU in Appendix B shows the 6 following combinations:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Depending on the instruction class, the ALU will need to perform one of these first five functions. (NOR is needed for other parts of the MIPS instruction set.) For load word and store word instructions, we use the ALU to compute the memory address by addition. For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field in the low-order bits of the instruction (see Chapter 2). For branch equal, the ALU must perform a subtraction.

We can generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract (01) for `beq`, or determined by the operation encoded in the funct field (10). The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations shown previously.

In Figure 5.12, we show how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code. For completeness, the relationship between the ALUOp bits and the instruction opcode is also shown. Later in this chapter we will see how the ALUOp bits are generated from the main control unit.

This style of using multiple levels of decoding—that is, the main control unit generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit—is a common implementation technique. Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially increase the speed of the control unit. Such optimizations are important, since the control unit is often performance-critical.

There are several different ways to implement the mapping from the 2-bit ALUOp field and the 6-bit funct field to the three ALU operation control bits.

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 5.12 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we "don't care" about the value of the function code, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input.

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 5.13 The truth table for the three ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

Because only a small number of the 64 possible values of the function field are of interest and the function field is used only when the ALUOp bits equal 10, we can use a small piece of logic that recognizes the subset of possible values and causes the correct setting of the ALU control bits.

As a step in designing this logic, it is useful to create a truth table for the interesting combinations of the function code field and the ALUOp bits, as we've done in Figure 5.13; this truth table shows how the 3-bit ALU control is set depending

on these two input fields. Since the full truth table is very large ($2^8 = 256$ entries) and we don't care about the value of the ALU control for many of these input combinations, we show only the truth table entries for which the ALU control must have a specific value. Throughout this chapter, we will use this practice of showing only the truth table entries that must be asserted and not showing those that are all zero or don't care. (This practice has a disadvantage, which we discuss in Section C.2 of Appendix C.)

Because in many instances we do not care about the values of some of the inputs and to keep the tables compact, we also include **don't-care terms**. A don't-care term in this truth table (represented by an X in an input column) indicates that the output does not depend on the value of the input corresponding to that column. For example, when the ALUOp bits are 00, as in the first line of the table in Figure 5.13, we always set the ALU control to 010, independent of the function code. In this case, then, the function code inputs will be don't cares in this line of the truth table. Later, we will see examples of another type of don't-care term. If you are unfamiliar with the concept of don't-care terms, see Appendix B for more information.

don't-care term An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways.

Once the truth table has been constructed, it can be optimized and then turned into gates. This process is completely mechanical. Thus, rather than show the final steps here, we describe the process and the result in Section C.2 of Appendix C.

Designing the Main Control Unit

Now that we have described how to design an ALU that uses the function code and a 2-bit signal as its control inputs, we can return to looking at the rest of the control. To start this process, let's identify the fields of an instruction and the control lines that are needed for the datapath we constructed in Figure 5.11 on page 300. To understand how to connect the fields of an instruction to the datapath, it is useful to review the formats of the three instruction classes: the R-type, branch, and load/store instructions. Figure 5.14 shows these formats.

There are several major observations about this instruction format that we will rely on:

- The op field, also called the **opcode**, is always contained in bits 31:26. We will refer to this field as Op[5:0].
- The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and for store.
- The base register for load and store instructions is always in bit positions 25:21 (rs).

opcode The field that denotes the operation and format of an instruction.

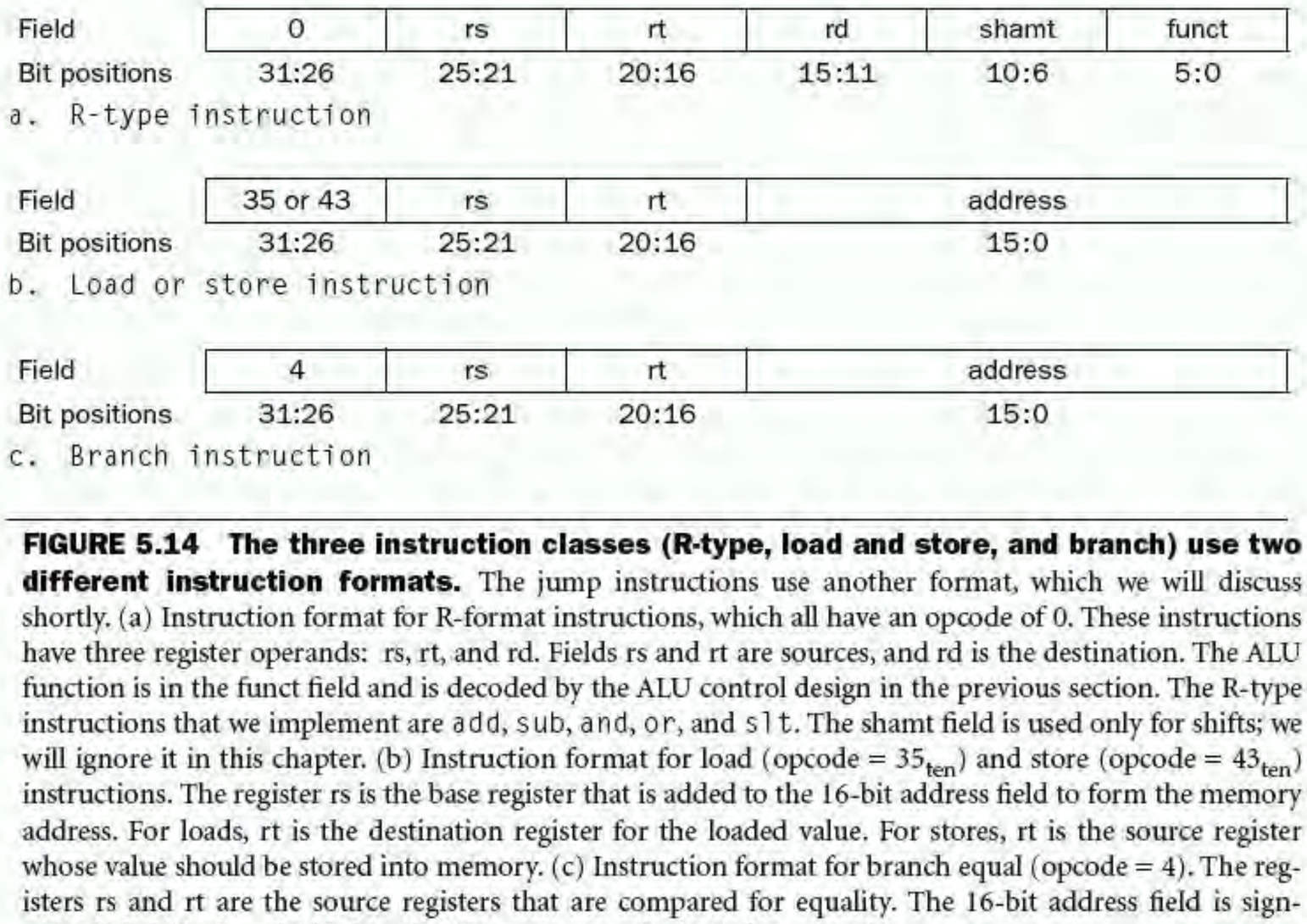


FIGURE 5.14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats. The jump instructions use another format, which we will discuss shortly. (a) Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. The ALU function is in the funct field and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are `add`, `sub`, `and`, `or`, and `slt`. The shamt field is used only for shifts; we will ignore it in this chapter. (b) Instruction format for load (opcode = 35_{ten}) and store (opcode = 43_{ten}) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory. (c) Instruction format for branch equal (opcode = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC to compute the branch target address.

- The 16-bit offset for branch equal, load, and store is always in positions 15:0.
- The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd). Thus we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

Using this information, we can add the instruction labels and extra multiplexor (for the Write register number input of the register file) to the simple datapath. Figure 5.15 shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexors. Since all the multiplexors have two inputs, they each require a single control line.

Figure 5.15 shows seven single-bit control lines plus the 2-bit ALUOp control signal. We have already defined how the ALUOp control signal works, and it is useful to define what the seven other control signals do informally before we determine how to set these control signals during instruction execution. Figure 5.16 describes the function of these seven control lines.

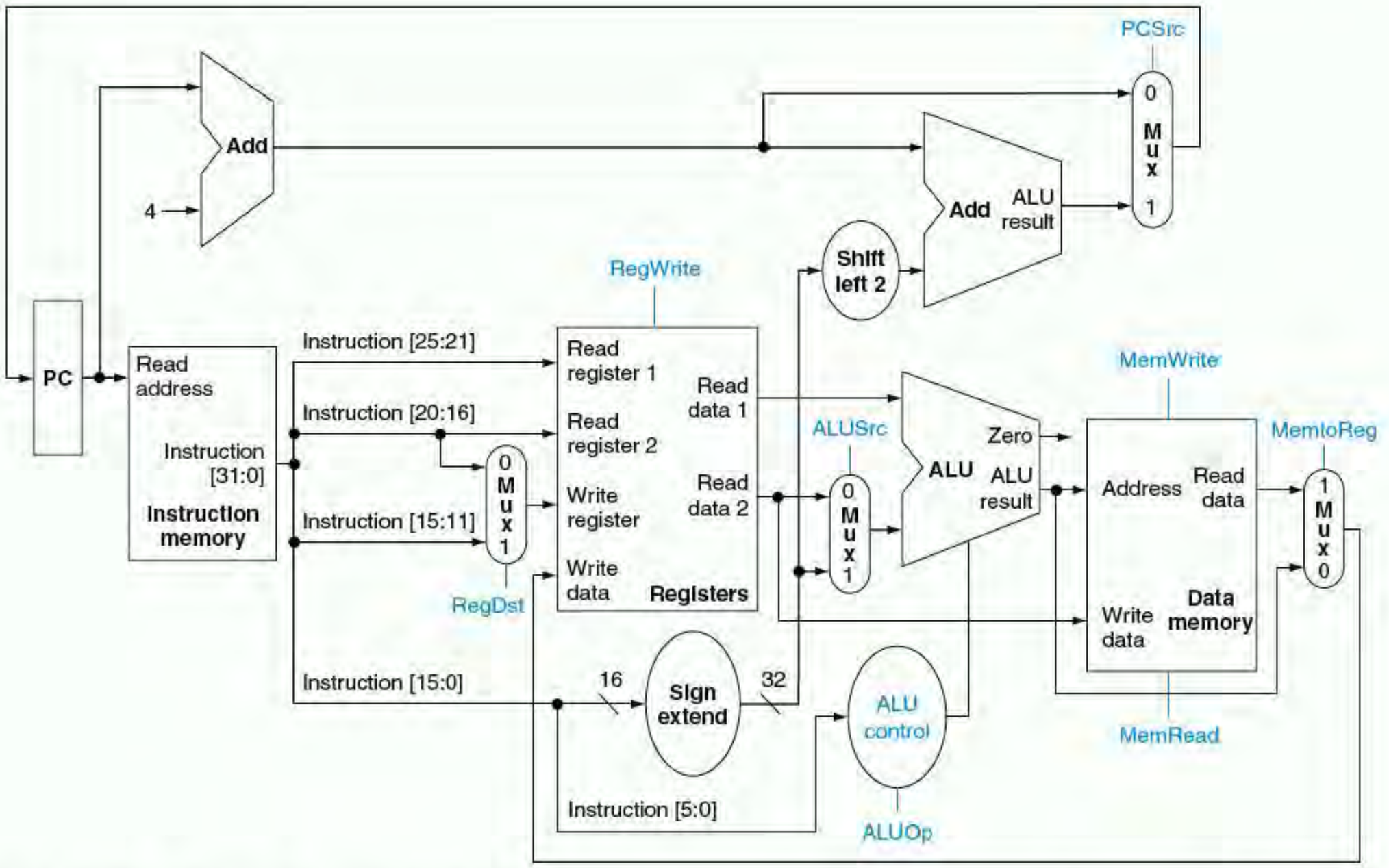


FIGURE 5.15 The datapath of Figure 5.12 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

Now that we have looked at the function of each of the control signals, we can look at how to set them. The control unit can set all but one of the control signals based solely on the opcode field of the instruction. The PCSrc control line is the exception. That control line should be set if the instruction is branch on equal (a decision that the control unit can make) *and* the Zero output of the ALU, which is used for equality comparison, is true. To generate the PCSrc signal, we will need to AND together a signal from the control unit, which we call *Branch*, with the Zero signal out of the ALU.

These nine control signals (seven from Figure 5.16 and two for ALUOp) can now be set on the basis of six input signals to the control unit, which are the opcode bits. Figure 5.17 shows the datapath with the control unit and the control signals.

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 5.16 The effect of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. The clock is never gated externally to a state element, since this can create timing problems. (See Appendix B for further discussion of this problem.)

Before we try to write a set of equations or a truth table for the control unit, it will be useful to try to define the control function informally. Because the setting of the control lines depends only on the opcode, we define whether each control signal should be 0, 1, or don't care (X), for each of the opcode values. Figure 5.18 defines how the control signals should be set for each opcode; this information follows directly from Figures 5.12, 5.16, and 5.17.

Operation of the Datapath

With the information contained in Figures 5.16 and 5.18, we can design the control unit logic, but before we do that, let's look at how each instruction uses the datapath. In the next few figures, we show the flow of three different instruction classes through the datapath. The asserted control signals and active datapath elements are highlighted in each of these. Note that a multiplexor whose control is 0 has a definite action, even if its control line is not highlighted. Multiple-bit control signals are highlighted if any constituent signal is asserted.

Figure 5.19 shows the operation of the datapath for an R-type instruction, such as `add $t1,$t2,$t3`. Although everything occurs in 1 clock cycle, we can think of four steps to execute the instruction; these steps are ordered by the flow of information:

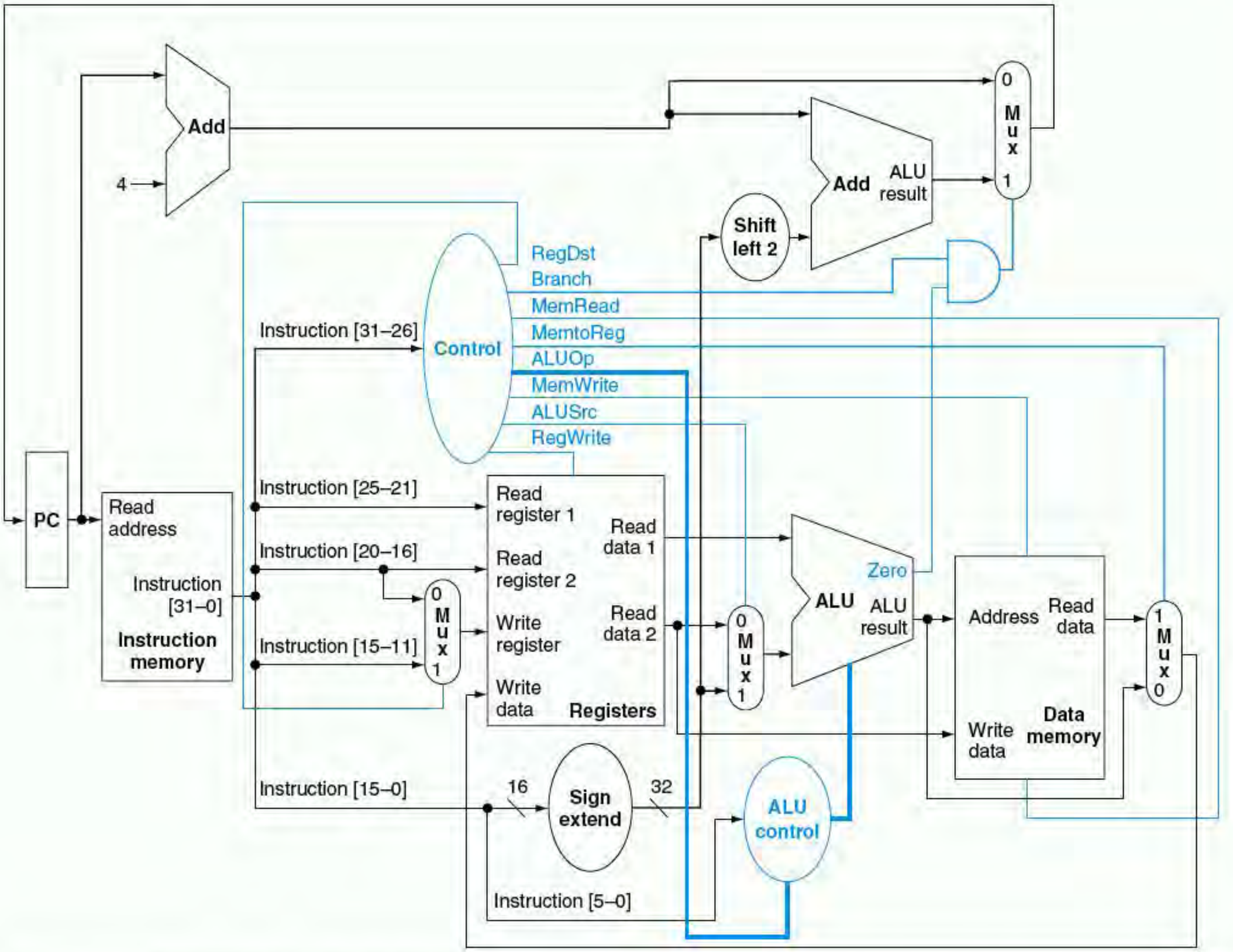


FIGURE 5.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus we drop the signal name in subsequent figures.

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

FIGURE 5.18 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of the table corresponds to the R-format instructions (`add`, `sub`, `and`, `or`, and `slt`). For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set. Furthermore, an R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field. The second and third rows of this table give the control signal settings for `lw` and `sw`. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register. The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

1. The instruction is fetched, and the PC is incremented.
2. Two registers, `$t2` and `$t3`, are read from the register file, and the main control unit computes the setting of the control lines during this step also.
3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.
4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register (`$t1`).

Similarly, we can illustrate the execution of a load word, such as

```
lw $t1, offset($t2)
```

in a style similar to Figure 5.19. Figure 5.20 on page 310 shows the active functional units and asserted control lines for a load. We can think of a load instruction as operating in five steps (similar to the R-type executed in four):

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register (`$t2`) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (`offset`).
4. The sum from the ALU is used as the address for the data memory.

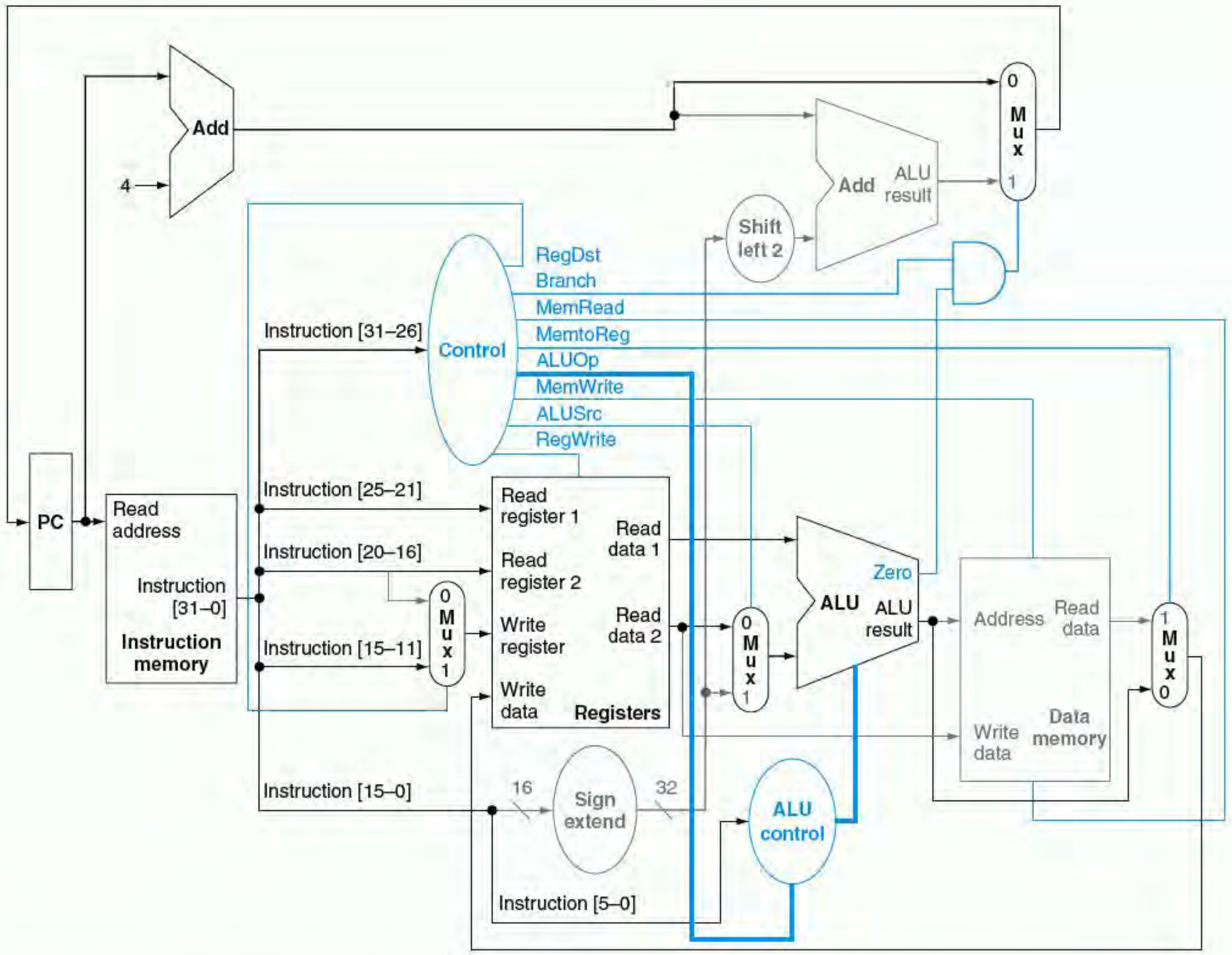


FIGURE 5.19 The datapath in operation for an R-type instruction such as add `$t1,$t2,$t3`. The control lines, datapath units, and connections that are active are highlighted.

5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction (`$t1`).

Finally, we can show the operation of the branch-on-equal instruction, such as `beq $t1,$t2,offset`, in the same fashion. It operates much like an R-format

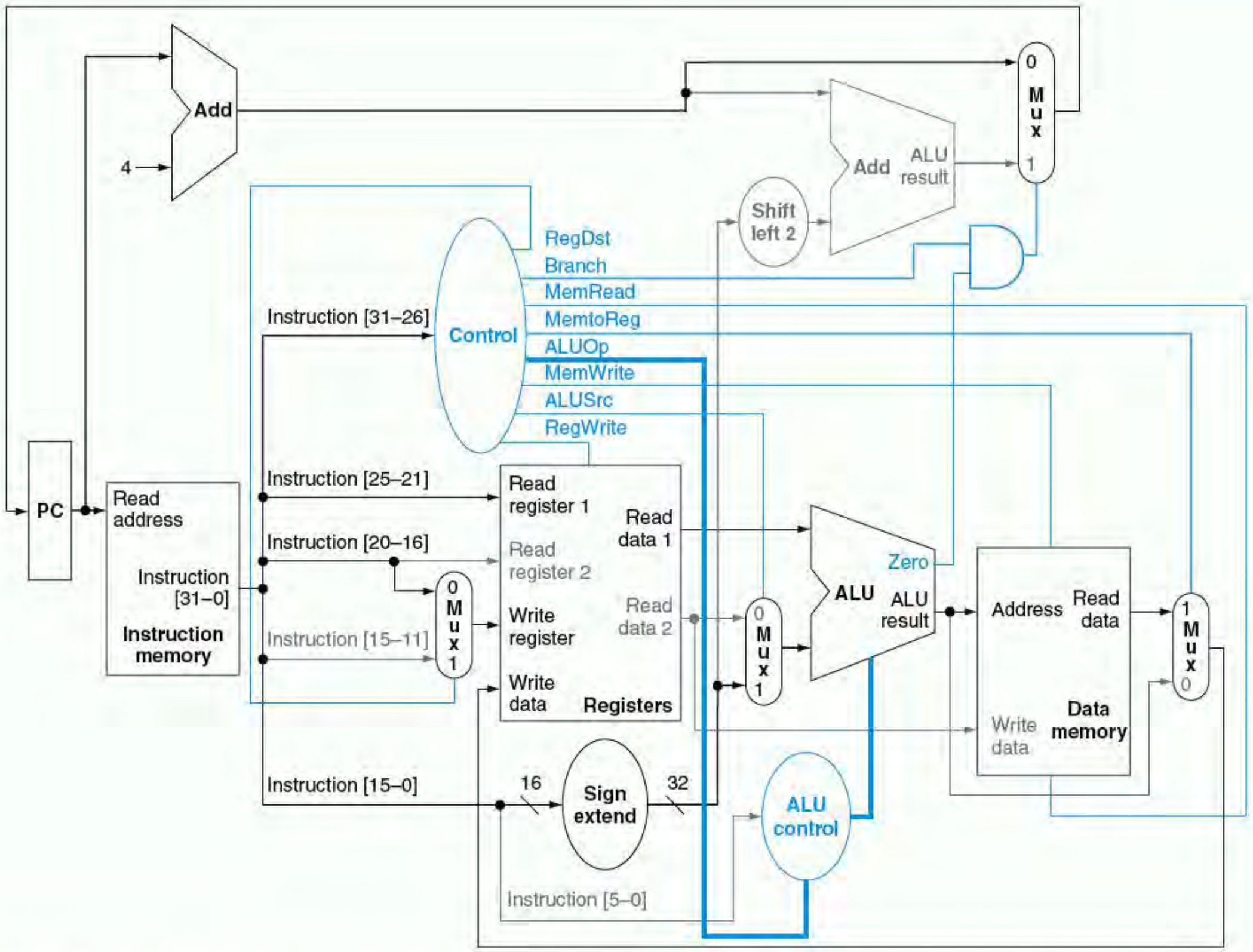


FIGURE 5.20 The datapath in operation for a load instruction. The control lines, datapath units, and connections that are active are highlighted. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occur.

instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address. Figure 5.21 shows the four steps in execution:

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers, `$t1` and `$t2`, are read from the register file.

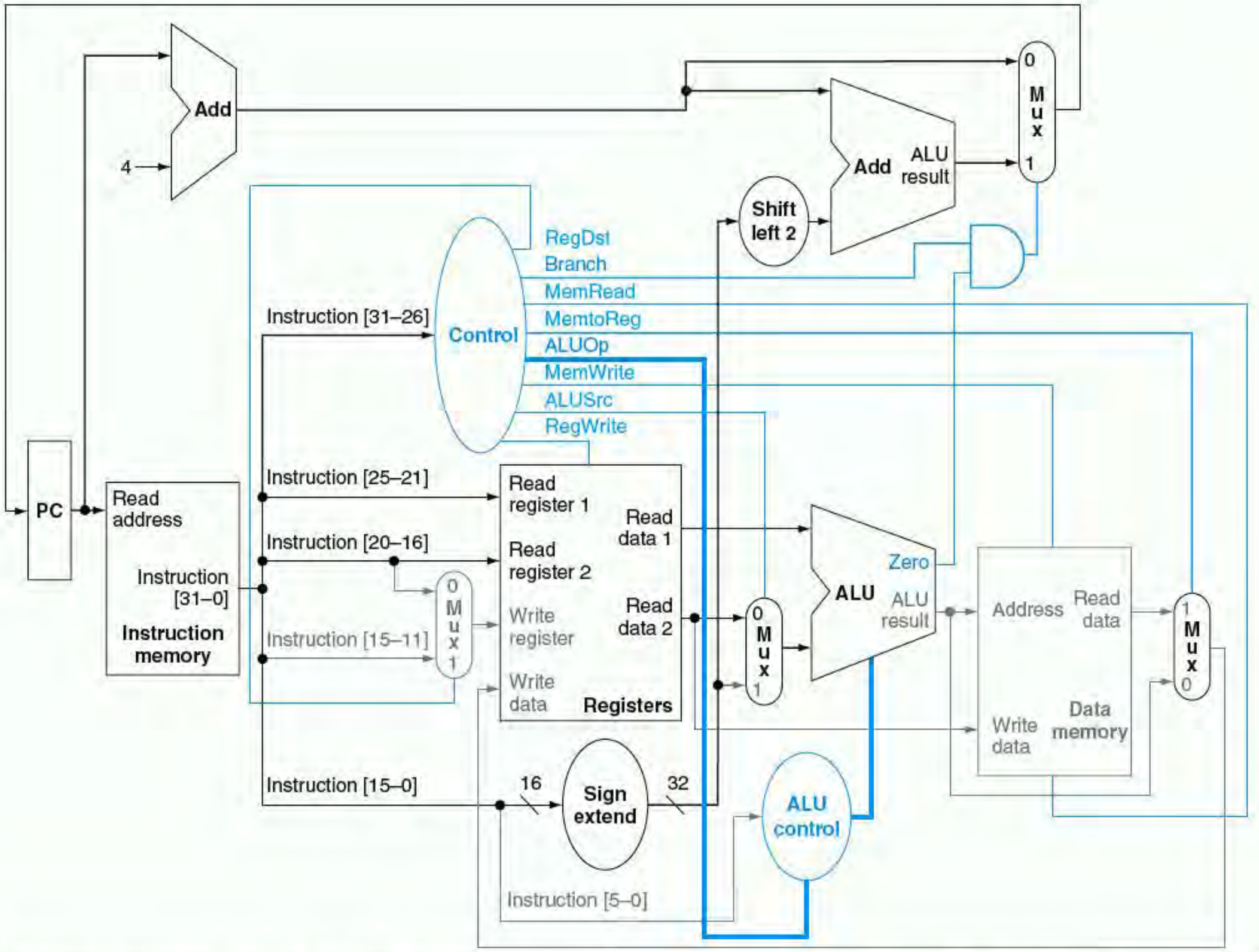


FIGURE 5.21 The datapath in operation for a branch equal instruction. The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.

3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (`offset`) shifted left by two; the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

In the next section, we will examine machines that are truly sequential, namely, those in which each of these steps is a distinct clock cycle.

Finalizing the Control

Now that we have seen how the instructions operate in steps, let's continue with the control implementation. The control function can be precisely defined using the contents of Figure 5.18 on page 308. The outputs are the control lines, and the input is the 6-bit opcode field, Op [5:0]. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.

Figure 5.22 shows the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and we can implement it directly in gates in an automated fashion. We show this final step in Section C.2 in Appendix C.

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

single-cycle implementation Also called single clock cycle implementation. An implementation in which an instruction is executed in one clock cycle.

FIGURE 5.22 The control function for the simple single-cycle implementation is completely specified by this truth table. The top half of the table gives the combinations of input signals that correspond to the four opcodes that determine the control output settings. (Remember that Op [5:0] corresponds to bits 31:26 of the instruction, which is the op field.) The bottom portion of the table gives the outputs. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression $\overline{Op5} \cdot \overline{Op2}$, since this is sufficient to distinguish the R-format instructions from `lw`, `sw`, and `beq`. We do not take advantage of this simplification, since the rest of the MIPS opcodes are used in a full implementation.

Now, let's add the jump instruction to show how the basic datapath and control can be extended to handle other instructions in the instruction set.

Implementing Jumps

EXAMPLE

Figure 5.17 on page 307 shows the implementation of many of the instructions we looked at in Chapter 2. One class of instructions missing is that of the jump instruction. Extend the datapath and control of Figure 5.17 to include the jump instruction. Describe how to set any new control lines.

ANSWER

The jump instruction looks somewhat like a branch instruction but computes the target PC differently and is not conditional. Like a branch, the low-order 2 bits of a jump address are always 00_{two}. The next lower 26 bits of this 32-bit address come from the 26-bit immediate field in the instruction, as shown in Figure 5.23. The upper 4 bits of the address that should replace the PC come from the PC of the jump instruction plus 4. Thus, we can implement a jump by storing into the PC the concatenation of

- the upper 4 bits of the current PC + 4 (these are bits 31:28 of the sequentially following instruction address)
- the 26-bit immediate field of the jump instruction
- the bits 00_{two}

Figure 5.24 shows the addition of the control for jump added to Figure 5.17. An additional multiplexor is used to select the source for the new PC value, which is either the incremented PC (PC + 4), the branch target PC, or the jump target PC. One additional control signal is needed for the additional multiplexor. This control signal, called *Jump*, is asserted only when the instruction is a jump—that is, when the opcode is 2.

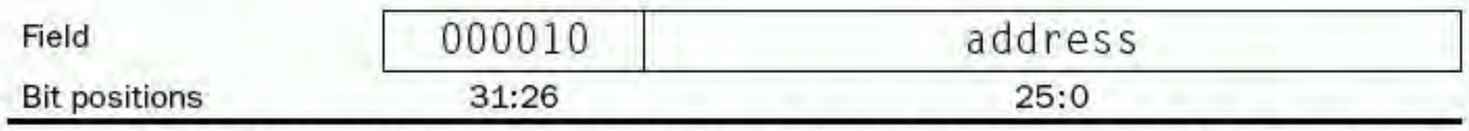


FIGURE 5.23 Instruction format for the jump instruction (opcode = 2). The destination address for a jump instruction is formed by concatenating the upper 4 bits of the current PC + 4 to the 26-bit address field in the jump instruction and adding 00 as the 2 low-order bits.

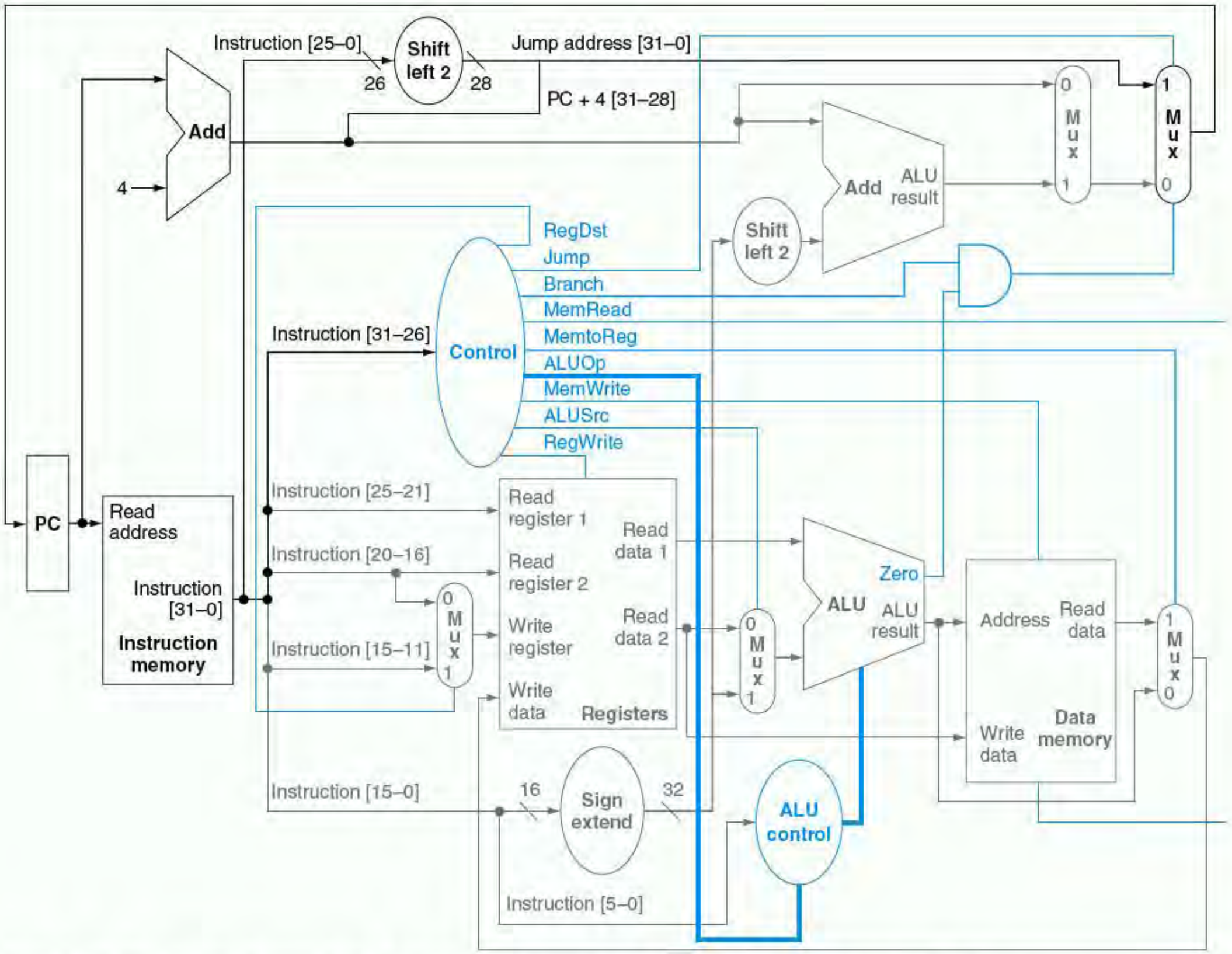


FIGURE 5.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

Why a Single-Cycle Implementation Is Not Used Today

Although the single-cycle design will work correctly, it would not be used in modern designs because it is inefficient. To see why this is so, notice that the clock cycle must have the same length for every instruction in this single-cycle design, and the CPI

(see Chapter 4) will therefore be 1. Of course, the clock cycle is determined by the longest possible path in the machine. This path is almost certainly a load instruction, which uses five functional units in series: the instruction memory, the register file, the ALU, the data memory, and the register file. Although the CPI is 1, the overall performance of a single-cycle implementation is not likely to be very good, since several of the instruction classes could fit in a shorter clock cycle.

Performance of Single-Cycle Machines

EXAMPLE

Assume that the operation times for the major functional units in this implementation are the following:

- Memory units: 200 picoseconds (ps)
- ALU and adders: 100 ps
- Register file (read or write): 50 ps

Assuming that the multiplexors, control unit, PC accesses, sign extension unit, and wires have no delay, which of the following implementations would be faster and by how much?

1. An implementation in which every instruction operates in 1 clock cycle of a fixed length.
2. An implementation where every instruction executes in 1 clock cycle using a variable-length clock, which for each instruction is only as long as it needs to be. (Such an approach is not terribly practical, but it will allow us to see what is being sacrificed when all the instructions must execute in a single clock of the same length.)

To compare the performance, assume the following instruction mix: 25% loads, 10% stores, 45% ALU instructions, 15% branches, and 5% jumps.

ANSWER

Let's start by comparing the CPU execution times. Recall from Chapter 4 that

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

Since CPI must be 1, we can simplify this to

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

We need only find the clock cycle time for the two implementations, since the instruction count and CPI are the same for both implementations. The critical path for the different instruction classes is as follows:

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

Using these critical paths, we can compute the required length for each instruction class:

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

The clock cycle for a machine with a single clock for all instructions will be determined by the longest instruction, which is 600 ps. (This timing is approximate, since our timing model is quite simplistic. In reality, the timing of modern digital systems is complex.)

A machine with a variable clock will have a clock cycle that varies between 200 ps and 600 ps. We can find the average clock cycle length for a machine with a variable-length clock using the information above and the instruction frequency distribution.

Thus, the average time per instruction with a variable clock is

$$\text{CPU clock cycle} = 600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\%$$

$$= 447.5 \text{ ps}$$

Since the variable clock implementation has a shorter average clock cycle, it is clearly faster. Let's find the performance ratio:

$$\frac{\text{CPU performance}_{\text{variable clock}}}{\text{CPU performance}_{\text{single clock}}} = \frac{\text{CPU execution time}_{\text{single clock}}}{\text{CPU execution time}_{\text{variable clock}}}$$

$$= \left(\frac{\text{IC} \times \text{CPU clock cycle}_{\text{single clock}}}{\text{IC} \times \text{CPU clock cycle}_{\text{variable clock}}} = \frac{\text{CPU clock cycle}_{\text{single clock}}}{\text{CPU clock cycle}_{\text{variable clock}}}\right)$$

$$= \frac{600}{447.5} = 1.34$$

The variable clock implementation would be 1.34 times faster. Unfortunately, implementing a variable-speed clock for each instruction class is extremely difficult, and the overhead for such an approach could be larger than any advantage gained. As we will see in the next section, an alternative is to use a shorter clock cycle that does less work and then vary the number of clock cycles for the different instruction classes.

The penalty for using the single-cycle design with a fixed clock cycle is significant, but might be considered acceptable for this small instruction set. Historically, early machines with very simple instruction sets did use this implementation technique. However, if we tried to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design wouldn't work well at all. An example of this is shown in the For More Practice Exercise 5.4.

Because we must assume that the clock cycle is equal to the worst-case delay for all instructions, we can't use implementation techniques that reduce the delay of the common case but do not improve the worst-case cycle time. A single-cycle implementation thus violates our key design principle of making the common case fast. In addition, in this single-cycle implementation, each functional unit can be used only once per clock; therefore, some functional units must be duplicated, raising the cost of the implementation. A single-cycle design is inefficient both in its performance and in its hardware cost!

We can avoid these difficulties by using implementation techniques that have a shorter clock cycle—derived from the basic functional unit delays—and that require multiple clock cycles for each instruction. The next section explores this alternative implementation scheme. In Chapter 6, we'll look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath, but is much more efficient. Pipelining gains efficiency by overlapping the execution of multiple instructions, increasing hardware utilization and improving performance. For those readers interested primarily in the high-level concepts used in processors, the material of this section is sufficient to read the introductory sections of Chapter 6 and understand the basic functional-

ity of a pipelined processor. For those, who want to understand how the hardware really implements the control, forge ahead!

Check Yourself Look at the control signal in Figure 5.22 on page 312. Can any control signal in the figure be replaced by the inverse of another? (Hint: Take into account the don't cares.) If so, can you use one signal for the other without adding an inverter?

5.5 A Multicycle Implementation

multicycle implementation Also called multiple clock cycle implementation. An implementation in which an instruction is executed in multiple clock cycles.

In an earlier example, we broke each instruction into a series of steps corresponding to the functional unit operations that were needed. We can use these steps to create a **multicycle implementation**. In a multicycle implementation, each *step* in the execution will take 1 clock cycle. The multicycle implementation allows a functional unit to be used more than once per instruction, as long as it is used on different clock cycles. This sharing can help reduce the amount of hardware required. The ability to allow instructions to take different numbers of clock cycles and the ability to share functional units within the execution of a single instruction are the major advantages of a multicycle design. Figure 5.25 shows the abstract version of the mul-

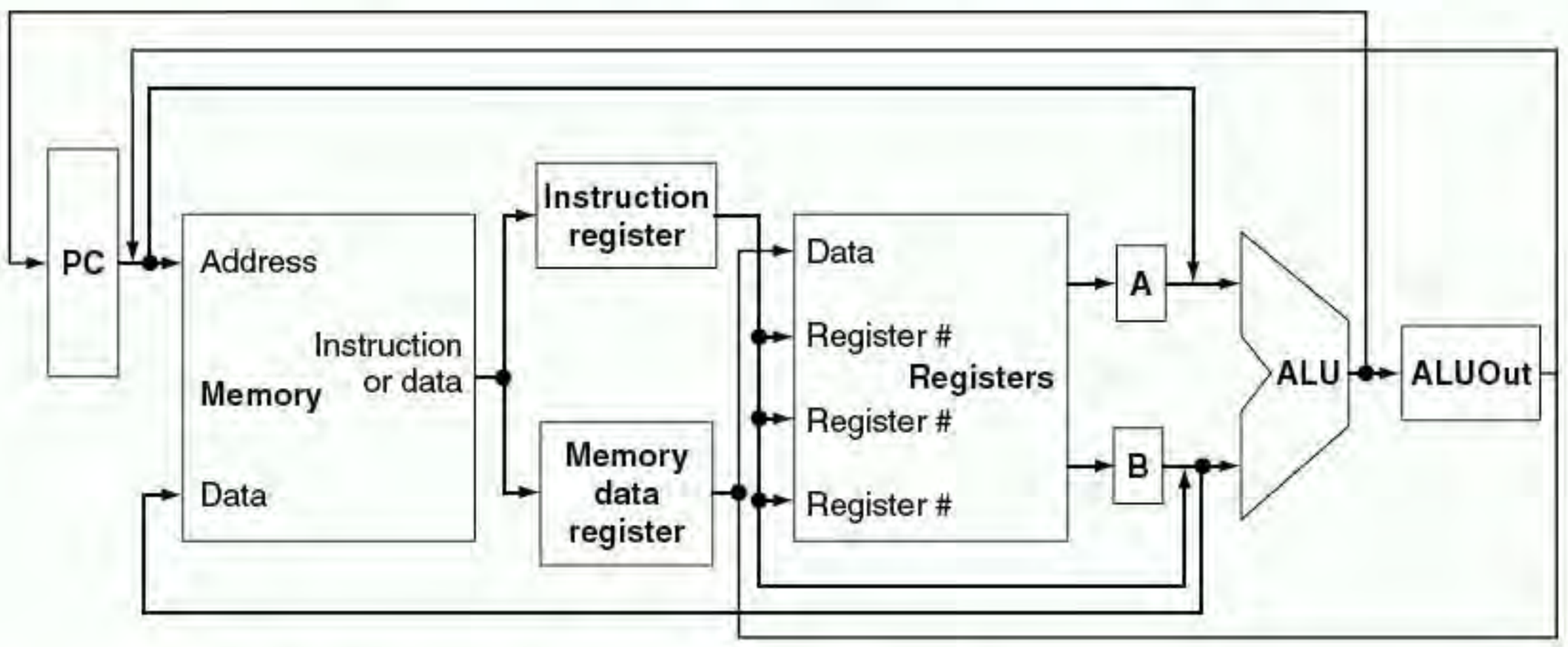


FIGURE 5.25 The high-level view of the multicycle datapath. This picture shows the key elements of the datapath: a shared memory unit, a single ALU shared among instructions, and the connections among these shared units. The use of shared functional units requires the addition or widening of multiplexors as well as new temporary registers that hold data between clock cycles of the same instruction. The additional registers are the Instruction register (IR), the Memory data register (MDR), A, B, and ALUOut.

ticycle datapath. If we compare Figure 5.25 to the datapath for the single-cycle version in Figure 5.11 on page 300, we can see the following differences:

- A single memory unit is used for both instructions and data.
- There is a single ALU, rather than an ALU and two adders.
- One or more registers are added after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle.

At the end of a clock cycle, all data that is used in subsequent clock cycles must be stored in a state element. Data used by *subsequent instructions* in a later clock cycle is stored into one of the programmer-visible state elements: the register file, the PC, or the memory. In contrast, data used by the *same instruction* in a later cycle must be stored into one of these additional registers.

Thus, the position of the additional registers is determined by the two factors: what combinational units will fit in one clock cycle and what data are needed in later cycles implementing the instruction. In this multicycle design, we assume that the clock cycle can accommodate at most one of the following operations: a memory access, a register file access (two reads or one write), or an ALU operation. Hence, any data produced by one of these three functional units (the memory, the register file, or the ALU) must be saved, into a temporary register for use on a later cycle. If it were not saved then the possibility of a timing race could occur, leading to the use of an incorrect value.

The following temporary registers are added to meet these requirements:

- The Instruction register (IR) and the Memory data register (MDR) are added to save the output of the memory for an instruction read and a data read, respectively. Two separate registers are used, since, as will be clear shortly, both values are needed during the same clock cycle.
- The A and B registers are used to hold the register operand values read from the register file.
- The ALUOut register holds the output of the ALU.

All the registers except the IR hold data only between a pair of adjacent clock cycles and will thus not need a write control signal. The IR needs to hold the instruction until the end of execution of that instruction, and thus will require a write control signal. This distinction will become more clear when we show the individual clock cycles for each instruction.

Because several functional units are shared for different purposes, we need both to add multiplexors and to expand existing multiplexors. For example, since one memory is used for both instructions and data, we need a multiplexor to select between the two sources for a memory address, namely, the PC (for instruction access) and ALUOut (for data access).

Replacing the three ALUs of the single-cycle datapath by a single ALU means that the single ALU must accommodate all the inputs that used to go to the three different ALUs. Handling the additional inputs requires two changes to the datapath:

1. An additional multiplexor is added for the first ALU input. The multiplexor chooses between the A register and the PC.
2. The multiplexor on the second ALU input is changed from a two-way to a four-way multiplexor. The two additional inputs to the multiplexor are the constant 4 (used to increment the PC) and the sign-extended and shifted offset field (used in the branch address computation).

Figure 5.26 shows the details of the datapath with these additional multiplexors. By introducing a few registers and multiplexors, we are able to reduce the number of memory units from two to one and eliminate two adders. Since registers and multiplexors are fairly small compared to a memory unit or ALU, this could yield a substantial reduction in the hardware cost.

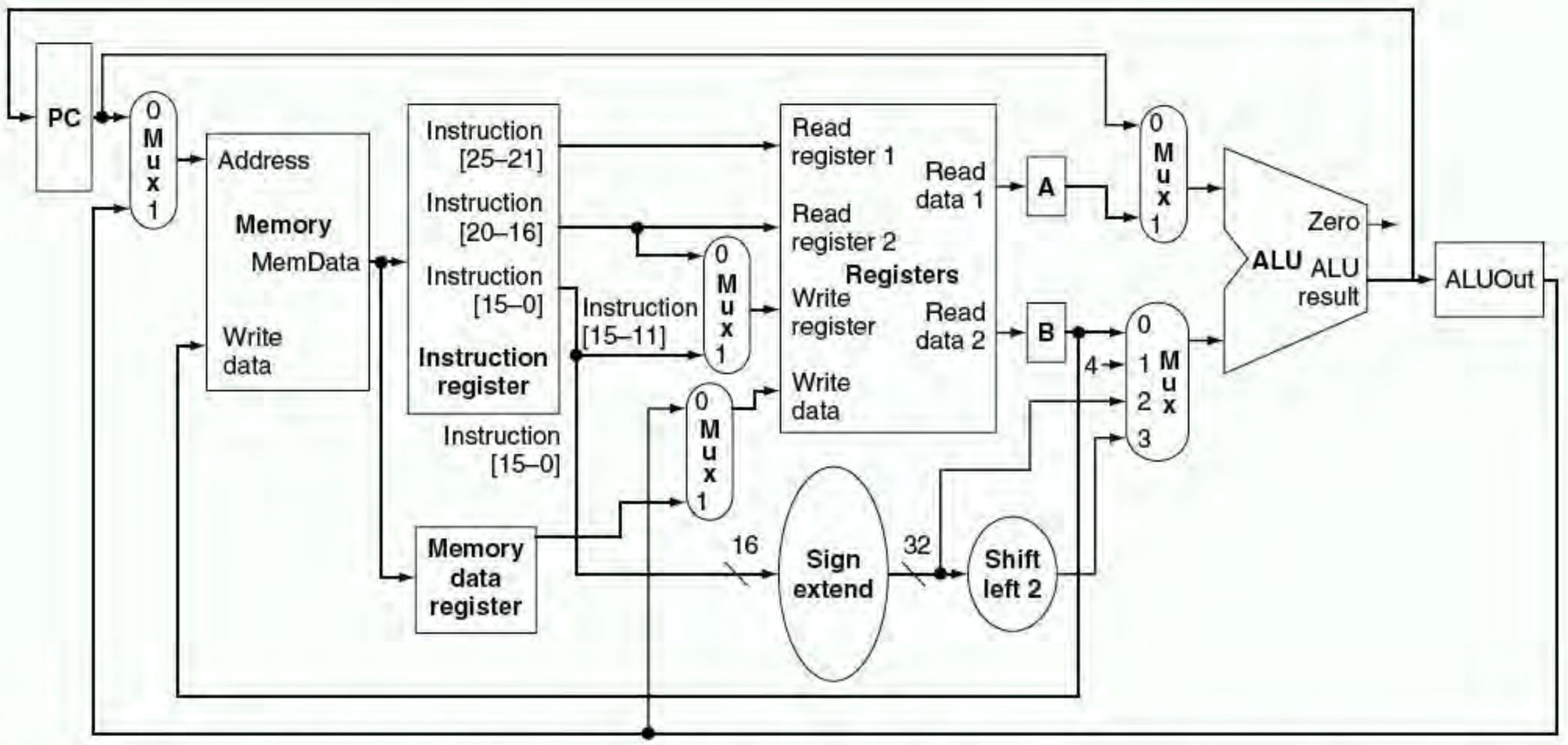


FIGURE 5.26 Multicycle datapath for MIPS handles the basic instructions. Although this datapath supports normal incrementing of the PC, a few more connections and a multiplexor will be needed for branches and jumps; we will add these shortly. The additions versus the single-clock datapath include several registers (IR, MDR, A, B, ALUOut), a multiplexor for the memory address, a multiplexor for the top ALU input, and expanding the multiplexor on the bottom ALU input into a four-way selector. These small additions allow us to remove two adders and a memory unit.

Because the datapath shown in Figure 5.26 takes multiple clock cycles per instruction, it will require a different set of control signals. The programmer-visible state units (the PC, the memory, and the registers) as well as the IR will need write control signals. The memory will also need a read signal. We can use the ALU control unit from the single-cycle datapath (see Figure 5.13 and Appendix C) to control the ALU here as well. Finally, each of the two-input multiplexors requires a single control line, while the four-input multiplexor requires two control lines. Figure 5.27 shows the datapath of Figure 5.26 with these control lines added.

The multicycle datapath still requires additions to support branches and jumps; after these additions, we will see how the instructions are sequenced and then generate the datapath control.

With the jump instruction and branch instruction, there are three possible sources for the value to be written into the PC:

1. The output of the ALU, which is the value PC + 4 during instruction fetch. This value should be stored directly into the PC.
2. The register ALUOut, which is where we will store the address of the branch target after it is computed.
3. The lower 26 bits of the Instruction register (IR) shifted left by two and concatenated with the upper 4 bits of the incremented PC, which is the source when the instruction is a jump.

As we observed when we implemented the single-cycle control, the PC is written both unconditionally and conditionally. During a normal increment and for jumps, the PC is written unconditionally. If the instruction is a conditional branch, the incremented PC is replaced with the value in ALUOut only if the two designated registers are equal. Hence, our implementation uses two separate control signals: PCWrite, which causes an unconditional write of the PC, and PCWriteCond, which causes a write of the PC if the branch condition is also true.

We need to connect these two control signals to the PC write control. Just as we did in the single-cycle datapath, we will use a few gates to derive the PC write control signal from PCWrite, PCWriteCond, and the Zero signal of the ALU, which is used to detect if the two register operands of a `beq` are equal. To determine whether the PC should be written during a conditional branch, we AND together the Zero signal of the ALU with the PCWriteCond. The output of this AND gate is then ORed with PCWrite, which is the unconditional PC write signal. The output of this OR gate is connected to the write control signal for the PC.

Figure 5.28 shows the complete multicycle datapath and control unit, including the additional control signals and multiplexor for implementing the PC updating.

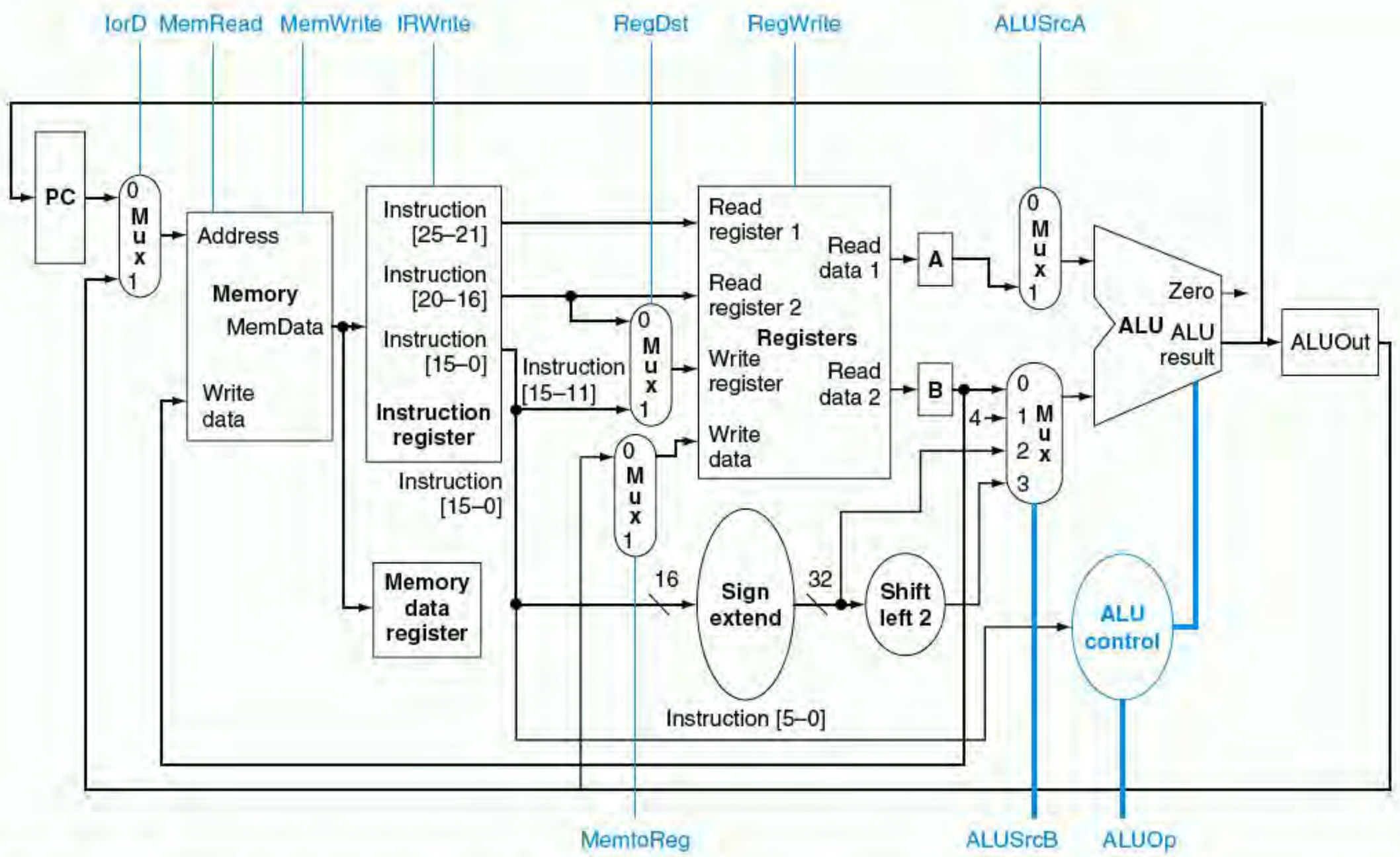


FIGURE 5.27 The multicycle datapath from Figure 5.26 with the control lines shown. The signals ALUOp and ALUSrcB are 2-bit control signals, while all the other control lines are 1-bit signals. Neither register A nor B requires a write signal, since their contents are only read on the cycle immediately after it is written. The memory data register has been added to hold the data from a load when the data returns from memory. Data from a load returning from memory cannot be written directly into the register file since the clock cycle cannot accommodate the time required for both the memory access and the register file write. The MemRead signal has been moved to the top of the memory unit to simplify the figures. The full set of datapaths and control lines for branches will be added shortly.

Before examining the steps to execute each instruction, let us informally examine the effect of all the control signals (just as we did for the single-cycle design in Figure 5.16 on page 306). Figure 5.29 shows what each control signal does when asserted and deasserted.

Elaboration: To reduce the number of signal lines interconnecting the functional units, designers can use *shared buses*. A shared bus is a set of lines that connect multiple units; in most cases, they include multiple sources that can place data on the bus

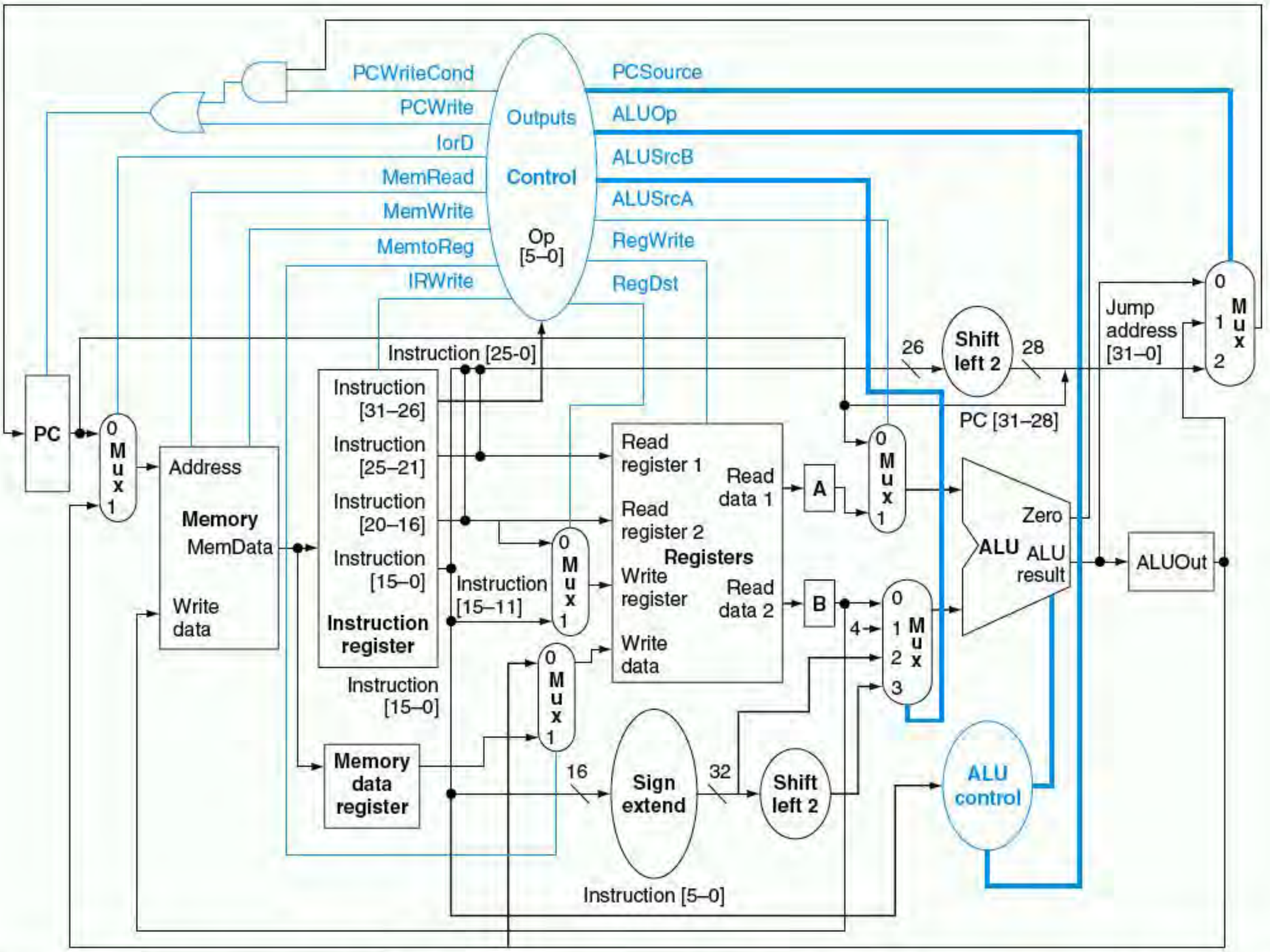


FIGURE 5.28 The complete datapath for the multicycle implementation together with the necessary control lines. The control lines of Figure 5.27 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure 5.27 include the multiplexor used to select the source of a new PC value; gates used to combine the PC write signals; and the control signals PCSource, PCWrite, and PCWriteCond. The PCWriteCond signal is used to decide whether a conditional branch should be taken. Support for jumps is included.

and multiple readers of the value. Just as we reduced the number of functional units for the datapath, we can reduce the number of buses interconnecting these units by sharing the buses. For example, there are six sources coming to the ALU; however, only two of them are needed at any one time. Thus, a pair of buses can be used to hold values that are being sent to the ALU. Rather than placing a large multiplexor in front of the

Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSource.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC + 4[31:28]) is sent to the PC for writing.

FIGURE 5.29 The action caused by the setting of each control signal in Figure 5.28 on page 323. The top table describes the 1-bit control signals, while the bottom table describes the 2-bit signals. Only those control lines that affect multiplexors have an action when they are deasserted. This information is similar to that in Figure 5.16 on page 306 for the single-cycle datapath, but adds several new control lines (IRWrite, PCWrite, PCWriteCond, ALUSrcB, and PCSource) and removes control lines that are no longer used or have been replaced (PCSrc, Branch, and Jump).

ALU, a designer can use a shared bus and then ensure that only one of the sources is driving the bus at any point. Although this saves signal lines, the same number of control lines will be needed to control what goes on the bus. The major drawback to using such bus structures is a potential performance penalty, since a bus is unlikely to be as fast as a point-to-point connection.

Breaking the Instruction Execution into Clock Cycles

Given the datapath in Figure 5.28, we now need to look at what should happen in each clock cycle of the multicycle execution, since this will determine what additional control signals may be needed, as well as the setting of the control signals. Our goal in breaking the execution into clock cycles should be to maximize performance. We can begin by breaking the execution of any instruction into a series of steps, each taking one clock cycle, attempting to keep the amount of work per cycle roughly equal. For example, we will restrict each step to contain at most one ALU operation, or one register file access, or one memory access. With this restriction, the clock cycle could be as short as the longest of these operations.

Recall that at the end of every clock cycle any data values that will be needed on a subsequent cycle must be stored into a register, which can be either one of the major state elements (e.g., the PC, the register file, or the memory), a temporary register written on every clock cycle (e.g., A, B, MDR, or ALUOut), or a temporary register with write control (e.g., IR). Also remember that because our design is edge-triggered, we can continue to read the current value of a register; the new value does not appear until the next clock cycle.

In the single-cycle datapath, each instruction uses a set of datapath elements to carry out its execution. Many of the datapath elements operate in series, using the output of another element as an input. Some datapath elements operate in parallel; for example, the PC is incremented and the instruction is read at the same time. A similar situation exists in the multicycle datapath. All the operations listed in one step occur in parallel within 1 clock cycle, while successive steps operate in series in different clock cycles. The limitation of one ALU operation, one memory access, and one register file access determines what can fit in one step.

Notice that we distinguish between reading from or writing into the PC or one of the stand-alone registers and reading from or writing into the register file. In the former case, the read or write is part of a clock cycle, while reading or writing a result into the register file takes an additional clock cycle. The reason for this distinction is that the register file has additional control and access overhead compared to the single stand-alone registers. Thus, keeping the clock cycle short motivates dedicating separate clock cycles for register file accesses.

The potential execution steps and their actions are given below. Each MIPS instruction needs from three to five of these steps:

1. Instruction fetch step

Fetch the instruction from memory and compute the address of the next sequential instruction:

```
IR <= Memory[PC];
PC <= PC + 4;
```

Operation: Send the PC to the memory as the address, perform a read, and write the instruction into the Instruction register (IR), where it will be stored. Also, increment the PC by 4. We use the symbol "<=" from Verilog; it indicates that all right-hand sides are evaluated and then all assignments are made, which is effectively how the hardware executes during the clock cycle.

To implement this step, we will need to assert the control signals MemRead and IRWrite, and set IorD to 0 to select the PC as the source of the address. We also increment the PC by 4, which requires setting the ALUSrcA signal to 0 (sending the PC to the ALU), the ALUSrcB signal to 01 (sending 4 to the ALU), and ALUOp to 00 (to make the ALU add). Finally, we will also want to store the incremented instruction address back into the PC, which requires setting PC source to 00 and setting PCWrite. The increment of the PC and the instruction memory access can occur in parallel. The new value of the PC is not visible until the next clock cycle. (The incremented PC will also be stored into ALUOut, but this action is benign.)

2. Instruction decode and register fetch step

In the previous step and in this one, we do not yet know what the instruction is, so we can perform only actions that are either applicable to all instructions (such as fetching the instruction in step 1) or are not harmful, in case the instruction isn't what we think it might be. Thus, in this step we can read the two registers indicated by the rs and rt instruction fields, since it isn't harmful to read them even if it isn't necessary. The values read from the register file may be needed in later stages, so we read them from the register file and store the values into the temporary registers A and B.

We will also compute the branch target address with the ALU, which also is not harmful because we can ignore the value if the instruction turns out not to be a branch. The potential branch target is saved in ALUOut.

Performing these "optimistic" actions early has the benefit of decreasing the number of clock cycles needed to execute an instruction. We can do these optimistic actions early because of the regularity of the instruction formats. For instance, if the instruction has two register inputs, they are always in the rs and rt fields, and if the instruction is a branch, the offset is always the low-order 16 bits:

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend (IR[15-0]) << 2);
```

Operation: Access the register file to read registers rs and rt and store the results into the registers A and B. Since A and B are overwritten on every cycle, the register file can be read on every cycle with the values stored into A and B. This step also computes the branch target address and stores the address in ALUOut, where

it will be used on the next clock cycle if the instruction is a branch. This requires setting ALUSrcA to 0 (so that the PC is sent to the ALU), ALUSrcB to the value 11 (so that the sign-extended and shifted offset field is sent to the ALU), and ALUOp to 00 (so the ALU adds). The register file accesses and computation of branch target occur in parallel.

After this clock cycle, determining the action to take can depend on the instruction contents.

3. Execution, memory address computation, or branch completion

This is the first cycle during which the datapath operation is determined by the instruction class. In all cases, the ALU is operating on the operands prepared in the previous step, performing one of four functions, depending on the instruction class. We specify the action to be taken depending on the instruction class:

Memory reference:

```
ALUOut <= A + sign-extend (IR[15:0]);
```

Operation: The ALU is adding the operands to form the memory address. This requires setting ALUSrcA to 1 (so that the first ALU input is register A) and setting ALUSrcB to 10 (so that the output of the sign extension unit is used for the second ALU input). The ALUOp signals will need to be set to 00 (causing the ALU to add).

Arithmetic-logical instruction (R-type):

```
ALUOut <= A op B;
```

Operation: The ALU is performing the operation specified by the function code on the two values read from the register file in the previous cycle. This requires setting ALUSrcA = 1 and setting ALUSrcB = 00, which together cause the registers A and B to be used as the ALU inputs. The ALUOp signals will need to be set to 10 (so that the funct field is used to determine the ALU control signal settings).

Branch:

```
if (A == B) PC <= ALUOut;
```

Operation: The ALU is used to do the equal comparison between the two registers read in the previous step. The Zero signal out of the ALU is used to determine whether or not to branch. This requires setting ALUSrcA = 1 and setting ALUSrcB = 00 (so that the register file outputs are the ALU inputs). The ALUOp signals will need to be set to 01 (causing the ALU to subtract) for equality testing. The PCWriteCond signal will need to be asserted to update the PC if the Zero output of the ALU is asserted. By set-

ting PCSource to 01, the value written into the PC will come from ALUOut, which holds the branch target address computed in the previous cycle. For conditional branches that are taken, we actually write the PC twice: once from the output of the ALU (during the Instruction decode/register fetch) and once from ALUOut (during the Branch completion step). The value written into the PC last is the one used for the next instruction fetch.

Jump:

```
# {x, y} is the Verilog notation for concatenation of
bit fields x and y
PC <= {PC [31:28], (IR[25:0]],2'b00)};
```

Operation: The PC is replaced by the jump address. PCSource is set to direct the jump address to the PC, and PCWrite is asserted to write the jump address into the PC.

4. Memory access or R-type instruction completion step

During this step, a load or store instruction accesses memory and an arithmetic-logical instruction writes its result. When a value is retrieved from memory, it is stored into the memory data register (MDR), where it must be used on the next clock cycle.

Memory reference:

```
MDR <= Memory [ALUOut];
```

or

```
Memory [ALUOut] <= B;
```

Operation: If the instruction is a load, a data word is retrieved from memory and is written into the MDR. If the instruction is a store, then the data is written into memory. In either case, the address used is the one computed during the previous step and stored in ALUOut. For a store, the source operand is saved in B. (B is actually read twice, once in step 2 and once in step 3. Luckily, the same value is read both times, since the register number—which is stored in IR and used to read from the register file—does not change.) The signal MemRead (for a load) or MemWrite (for store) will need to be asserted. In addition, for loads and stores, the signal IorD is set to 1 to force the memory address to come from the ALU, rather than the PC. Since MDR is written on every clock cycle, no explicit control signal need be asserted.

Arithmetic-logical instruction (R-type):

```
Reg[IR[15:11]] <= ALUOut;
```

Operation: Place the contents of ALUOut, which corresponds to the output of the ALU operation in the previous cycle, into the Result register. The signal RegDst must be set to 1 to force the rd field (bits 15:11) to be used to select the register file entry to write. RegWrite must be asserted, and MemtoReg must be set to 0 so that the output of the ALU is written, as opposed to the memory data output.

5. Memory read completion step

During this step, loads complete by writing back the value from memory.

Load:

```
Reg[IR[20:16]] <= MDR;
```

Operation: Write the load data, which was stored into MDR in the previous cycle, into the register file. To do this, we set MemtoReg = 1 (to write the result from memory), assert RegWrite (to cause a write), and we make RegDst = 0 to choose the rt (bits 20:16) field as the register number.

This five-step sequence is summarized in Figure 5.30. From this sequence we can determine what the control must do on each clock cycle.

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

Defining the Control

Now that we have determined what the control signals are and when they must be asserted, we can implement the control unit. To design the control unit for the single-cycle datapath, we used a set of truth tables that specified the setting of the control signals based on the instruction class. For the multicycle datapath, the control is more complex because the instruction is executed in a series of steps. The control for the multicycle datapath must specify both the signals to be set in any step and the next step in the sequence.

In this subsection and in Section 5.7, we will look at two different techniques to specify the control. The first technique is based on finite state machines that are usually represented graphically. The second technique, called **microprogramming**, uses a programming representation for control. Both of these techniques represent the control in a form that allows the detailed implementation—using gates, ROMs, or PLAs—to be synthesized by a CAD system. In this chapter, we will focus on the design of the control and its representation in these two forms.

microprogram A symbolic representation of control in the form of instructions, called microinstructions, that are executed on a simple micromachine.

Section 5.8 shows how hardware design languages are used to design modern processors with examples of both the multicycle datapath and the finite state control. In modern digital systems design, the final step of taking a hardware description to actual gates is handled by logic and datapath synthesis tools. Appendix C shows how this process operates by translating the multicycle control unit to a detailed hardware implementation. The key ideas of control can be grasped from this chapter without examining the material in either Section 5.8 or Appendix C. However, if you want to actually do some hardware design, Section 5.9 is useful, and Appendix C can show you what the implementations are likely to look like at the gate level.

Given this implementation, and the knowledge that each state requires 1 clock cycle, we can find the CPI for a typical instruction mix.

EXAMPLE

CPI in a Multicycle CPU

Using the SPECINT2000 instruction mix shown in Figure 3.26, what is the CPI, assuming that each state in the multicycle CPU requires 1 clock cycle?

ANSWER

The mix is 25% loads (1% load byte + 24% load word), 10% stores (1% store byte + 9% store word), 11% branches (6% `beq`, 5% `bne`), 2% jumps (1% `jal` + 1% `jr`), and 52% ALU (all the rest of the mix, which we assume to be ALU instructions). From Figure 5.30 on page 329, the number of clock cycles for each instruction class is the following:

- Loads: 5
- Stores: 4
- ALU instructions: 4
- Branches: 3
- Jumps: 3

The CPI is given by the following:

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

$$= \sum \frac{\text{Instruction count}_i}{\text{Instruction count}} \times \text{CPI}_i$$

The ratio

$$\frac{\text{Instruction count}_i}{\text{Instruction count}}$$

is simply the instruction frequency for the instruction class *i*. We can therefore substitute to obtain

$$\text{CPI} = 0.25 \times 5 + 0.10 \times 4 + 0.52 \times 4 + 0.11 \times 3 + 0.02 \times 3 = 4.12$$

This CPI is better than the worst-case CPI of 5.0 when all the instructions take the same number of clock cycles. Of course, overheads in both designs may reduce or increase this difference. The multicycle design is probably also more cost-effective, since it uses fewer separate components in the datapath.

The first method we use to specify the multicycle control is a **finite state machine**. A finite state machine consists of a set of states and directions on how to change states. The directions are defined by a **next-state function**, which maps the current state and the inputs to a new state. When we use a finite state machine for control, each state also specifies a set of outputs that are asserted when the machine is in that state. The implementation of a finite state machine usually assumes that all outputs that are not explicitly asserted are deasserted. Similarly, the correct operation of the datapath depends on the fact that a signal that is not explicitly asserted is deasserted, rather than acting as a don't care. For example, the RegWrite signal should be asserted only when a register file entry is to be written; when it is not explicitly asserted, it must be deasserted.

finite state machine A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

next-state function A combinational function that, given the inputs and the current state, determines the next state of a finite state machine.

Multiplexor controls are slightly different, since they select one of the inputs whether they are 0 or 1. Thus, in the finite state machine, we always specify the setting of all the multiplexor controls that we care about. When we implement the finite state machine with logic, setting a control to 0 may be the default and thus may not require any gates. A simple example of a finite state machine appears in Appendix B, and if you are unfamiliar with the concept of a finite state machine, you may want to examine Appendix B before proceeding.

The finite state control essentially corresponds to the five steps of execution shown on pages 325 through 329; each state in the finite state machine will take 1 clock cycle. The finite state machine will consist of several parts. Since the first two steps of execution are identical for every instruction, the initial two states of the finite state machine will be common for all instructions. Steps 3 through 5 differ, depending on the opcode. After the execution of the last step for a particular instruction class, the finite state machine will return to the initial state to begin fetching the next instruction.

Figure 5.31 shows this abstracted representation of the finite state machine. To fill in the details of the finite state machine, we will first expand the instruction fetch and decode portion, and then we will show the states (and actions) for the different instruction classes.

We show the first two states of the finite state machine in Figure 5.32 using a traditional graphic representation. We number the states to simplify the explanation, though the numbers are arbitrary. State 0, corresponding to step 1, is the starting state of the machine.

The signals that are asserted in each state are shown within the circle representing the state. The arcs between states define the next state and are labeled with

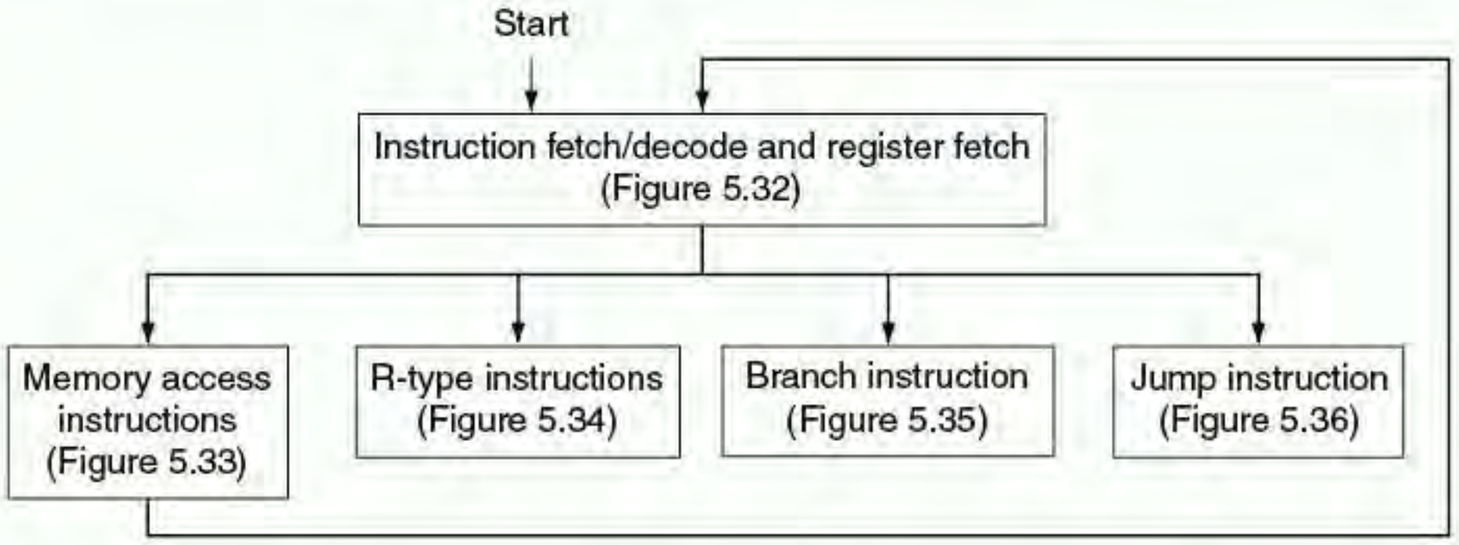


FIGURE 5.31 The high-level view of the finite state machine control. The first steps are independent of the instruction class; then a series of sequences that depend on the instruction opcode are used to complete each instruction class. After completing the actions needed for that instruction class, the control returns to fetch a new instruction. Each box in this figure may represent one to several states. The arc labeled *Start* marks the state in which to begin when the first instruction is to be fetched.

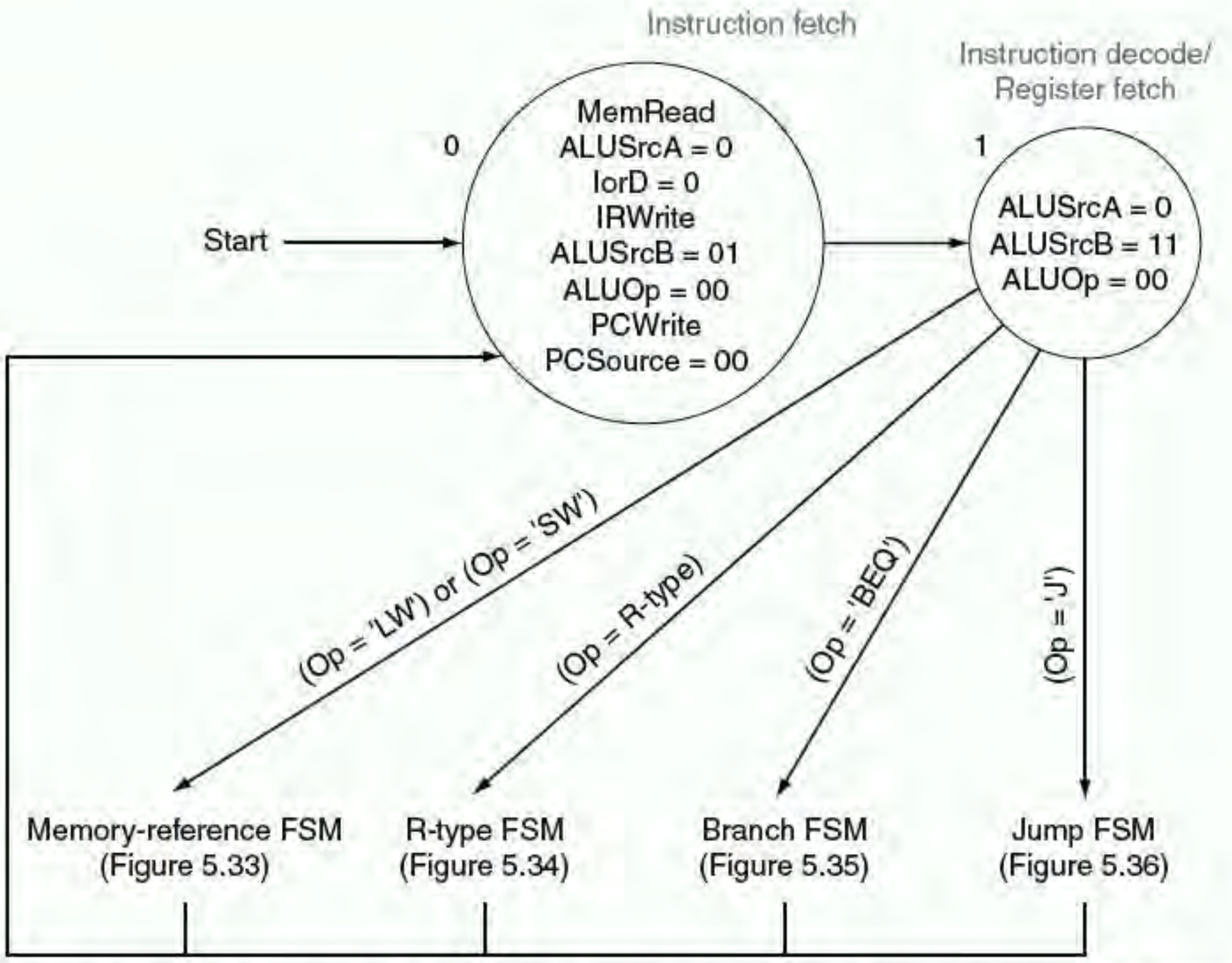


FIGURE 5.32 The instruction fetch and decode portion of every instruction is identical. These states correspond to the top box in the abstract finite state machine in Figure 5.31. In the first state we assert two signals to cause the memory to read an instruction and write it into the Instruction register (MemRead and IRWrite), and we set IorD to 0 to choose the PC as the address source. The signals ALUSrcA, ALUSrcB, ALUOp, PCWrite, and PCSource are set to compute PC + 4 and store it into the PC. (It will also be stored into ALUOut, but never used from there.) In the next state, we compute the branch target address by setting ALUSrcB to 11 (causing the shifted and sign-extended lower 16 bits of the IR to be sent to the ALU), setting ALUSrcA to 0 and ALUOp to 00; we store the result in the ALUOut register, which is written on every cycle. There are four next states that depend on the class of the instruction, which is known during this state. The control unit input, called Op, is used to determine which of these arcs to follow. Remember that all signals not explicitly asserted are deasserted; this is particularly important for signals that control writes. For multiplexors controls, lack of a specific setting indicates that we do not care about the setting of the multiplexor.

conditions that select a specific next state when multiple next states are possible. After state 1, the signals asserted depend on the class of instruction. Thus, the finite state machine has four arcs exiting state 1, corresponding to the four instruction classes: memory reference, R-type, branch on equal, and jump. This process of branching to different states depending on the instruction is called *decoding*, since the choice of the next state, and hence the actions that follow, depend on the instruction class.

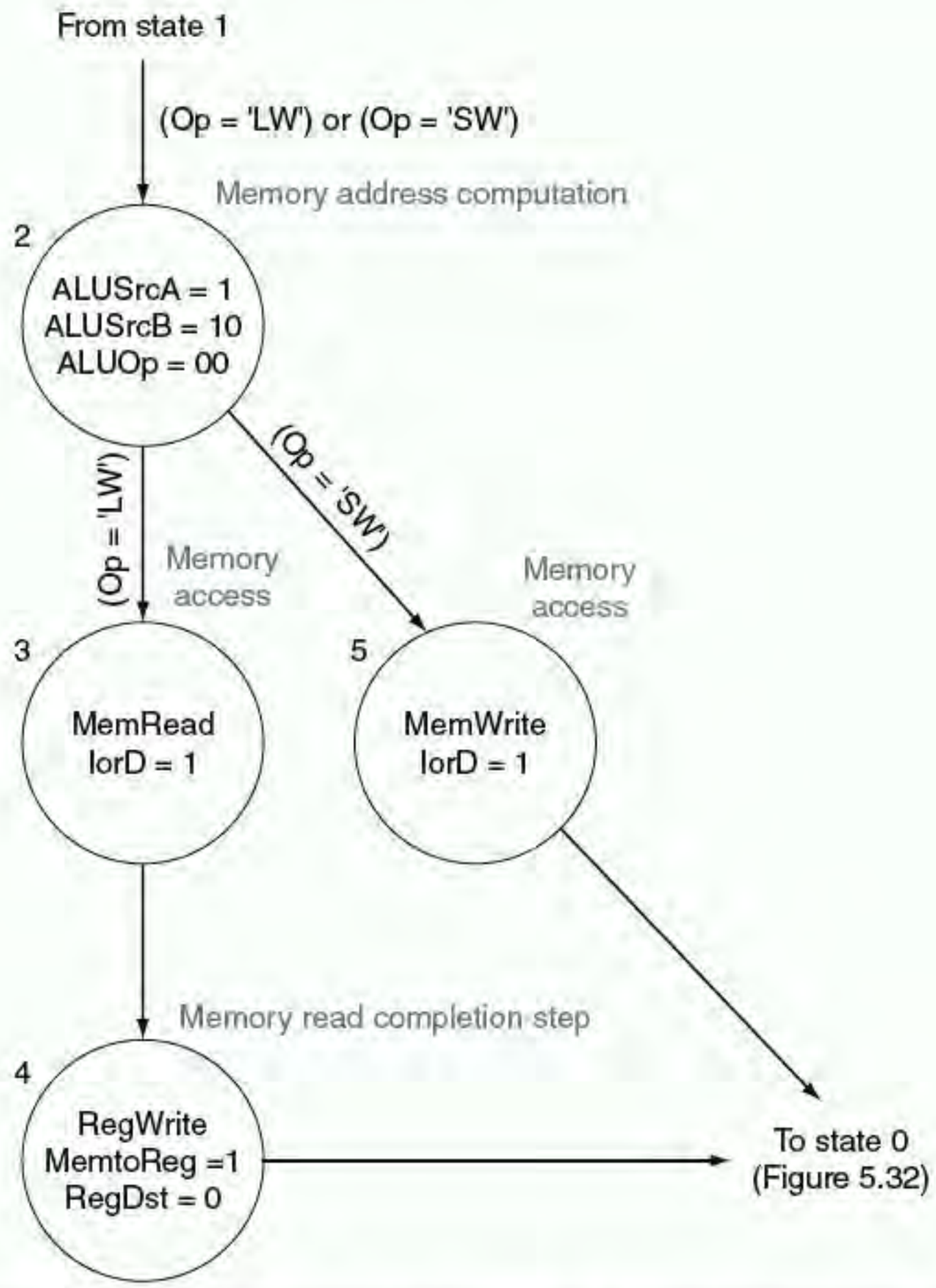


FIGURE 5.33 The finite state machine for controlling memory-reference instructions has four states. These states correspond to the box labeled "Memory access instructions" in Figure 5.31. After performing a memory address calculation, a separate sequence is needed for load and for store. The setting of the control signals ALUSrcA, ALUSrcB, and ALUOp is used to cause the memory address computation in state 2. Loads require an extra state to write the result from the MDR (where the result is written in state 3) into the register file.

Figure 5.33 shows the portion of the finite state machine needed to implement the memory-reference instructions. For the memory-reference instructions, the first state after fetching the instruction and registers computes the memory address (state 2). To compute the memory address, the ALU input multiplexors must be set so that the first input is the A register, while the second input is the sign-extended displacement field; the result is written into the ALUOut register. After the memory address calculation, the memory should be read or written; this requires two different states. If the instruction opcode is `lw`, then state 3 (corre-

sponding to the step Memory access) does the memory read (MemRead is asserted). The output of the memory is always written into MDR. If it is `sw`, state 5 does a memory write (MemWrite is asserted). In states 3 and 5, the signal IorD is set to 1 to force the memory address to come from the ALU. After performing a write, the instruction `sw` has completed execution, and the next state is state 0. If the instruction is a load, however, another state (state 4) is needed to write the result from the memory into the register file. Setting the multiplexor controls MemtoReg = 1 and RegDst = 0 will send the loaded value in the MDR to be written into the register file, using rt as the register number. After this state, corresponding to the Memory read completion step, the next state is state 0.

To implement the R-type instructions requires two states corresponding to steps 3 (Execute) and 4 (R-type completion). Figure 5.34 shows this two-state portion of the finite state machine. State 6 asserts ALUSrcA and sets the ALUSrcB

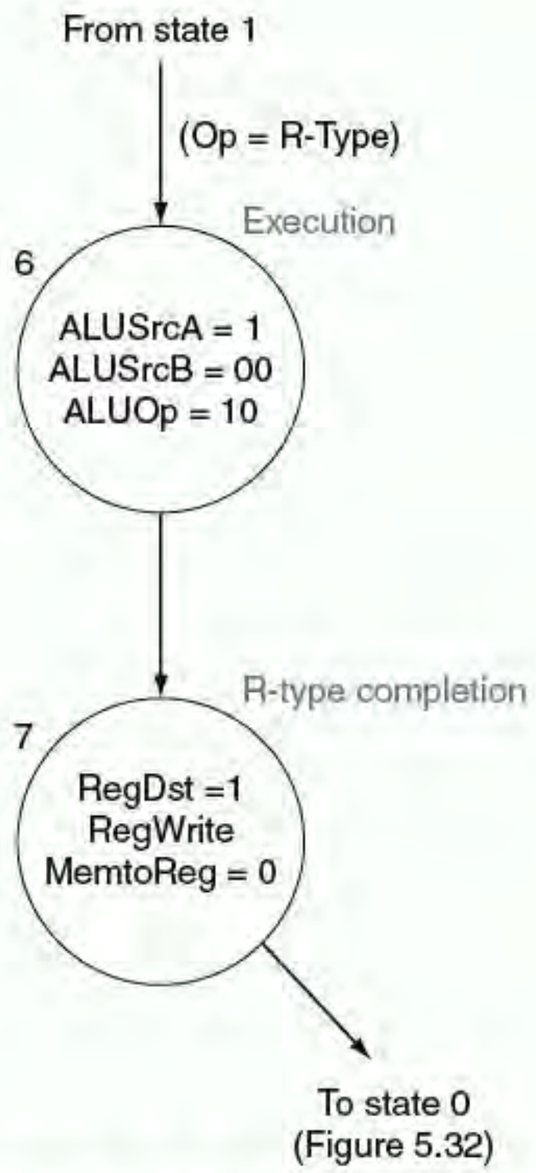


FIGURE 5.34 R-type instructions can be implemented with a simple two-state finite state machine. These states correspond to the box labeled "R-type instructions" in Figure 5.31. The first state causes the ALU operation to occur, while the second state causes the ALU result (which is in ALUOut) to be written in the register file. The three signals asserted during state 7 cause the contents of ALUOut to be written into the register file in the entry specified by the rd field of the Instruction register.

signals to 00; this forces the two registers that were read from the register file to be used as inputs to the ALU. Setting ALUOp to 10 causes the ALU control unit to use the function field to set the ALU control signals. In state 7, RegWrite is asserted to cause the register file to write, RegDst is asserted to cause the rd field to be used as the register number of the destination, and MemtoReg is deasserted to select ALUOut as the source of the value to write into the register file.

For branches, only a single additional state is necessary because they complete execution during the third step of instruction execution. During this state, the control signals that cause the ALU to compare the contents of registers A and B must be set, and the signals that cause the PC to be written conditionally with the address in the ALUOut register are also set. To perform the comparison requires that we assert ALUSrcA and set ALUSrcB to 00, and set the ALUOp value to 01 (forcing a subtract). (We use only the Zero output of the ALU, not the result of the subtraction.) To control the writing of the PC, we assert PCWriteCond and set PCSource = 01, which will cause the value in the ALUOut register (containing the branch address calculated in state 1, Figure 5.32 on page 333) to be written into the PC if the Zero bit out of the ALU is asserted. Figure 5.35 shows this single state.

The last instruction class is jump; like branch, it requires only a single state (shown in Figure 5.36) to complete its execution. In this state, the signal PCWrite is asserted to cause the PC to be written. By setting PCSource to 10, the value supplied for writing will be the lower 26 bits of the Instruction register with 00_{two} added as the low-order bits concatenated with the upper 4 bits of the PC.

We can now put these pieces of the finite state machine together to form a specification for the control unit, as shown in Figure 5.38. In each state, the signals that are asserted are shown. The next state depends on the opcode bits of the instruction, so we label the arcs with a comparison for the corresponding instruction opcodes.

A finite state machine can be implemented with a temporary register that holds the current state and a block of combinational logic that determines both the datapath signals to be asserted as well as the next state. Figure 5.37 shows how such an implementation might look. Appendix C describes in detail how the finite state machine is implemented using this structure. In Section C.3, the combinational control logic for the finite state machine of Figure 5.38 is implemented both with a ROM (read-only memory) and a PLA (programmable logic array). (Also see Appendix B for a description of these logic elements.) In the next section of this chapter, we consider another way to represent control. Both of these techniques are simply different representations of the same control information.

Pipelining, which is the subject of Chapter 6, is almost always used to accelerate the execution of instructions. For simple instructions, pipelining is capable of achieving the higher clock rate of a multicycle design and a single-cycle CPI of a single-clock design. In most pipelined processors, however, some instructions take longer than a single cycle and require multicycle control. Floating point-

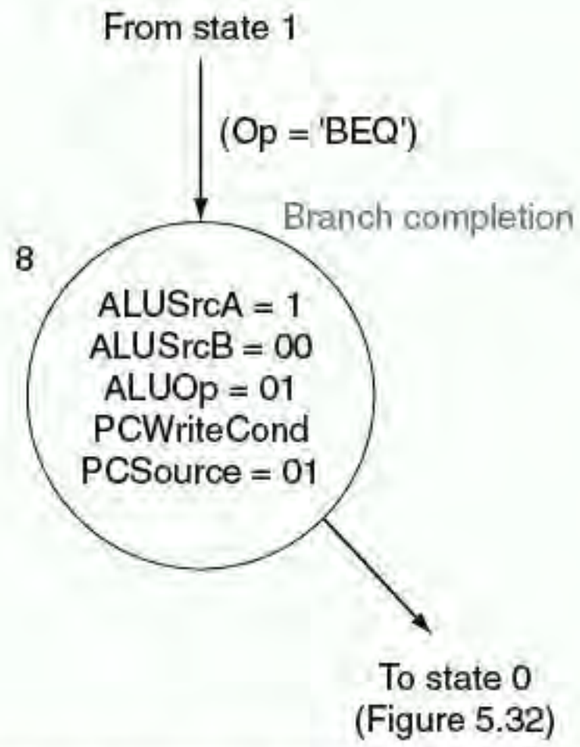


FIGURE 5.35 The branch instruction requires a single state. The first three outputs that are asserted cause the ALU to compare the registers (ALUSrcA, ALUSrcB, and ALUOp), while the signals PCSource and PCWriteCond perform the conditional write if the branch condition is true. Notice that we do not use the value written into ALUOut; instead, we use only the Zero output of the ALU. The branch target address is read from ALUOut, where it was saved at the end of state 1.

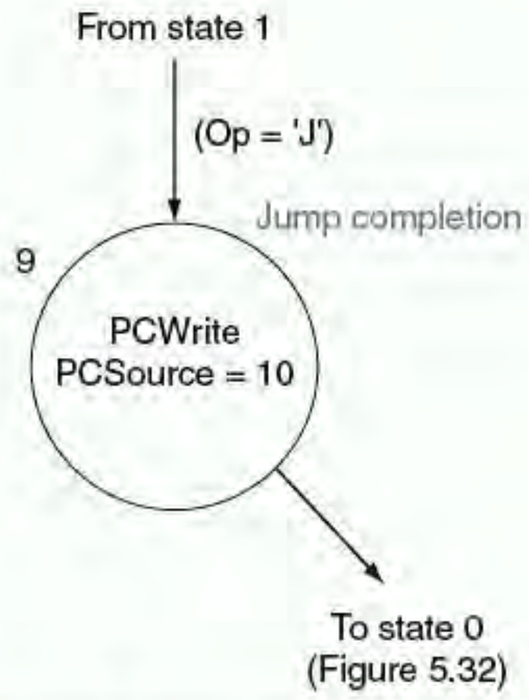


FIGURE 5.36 The jump instruction requires a single state that asserts two control signals to write the PC with the lower 26 bits of the Instruction register shifted left 2 bits and concatenated to the upper 4 bits of the PC of this instruction.

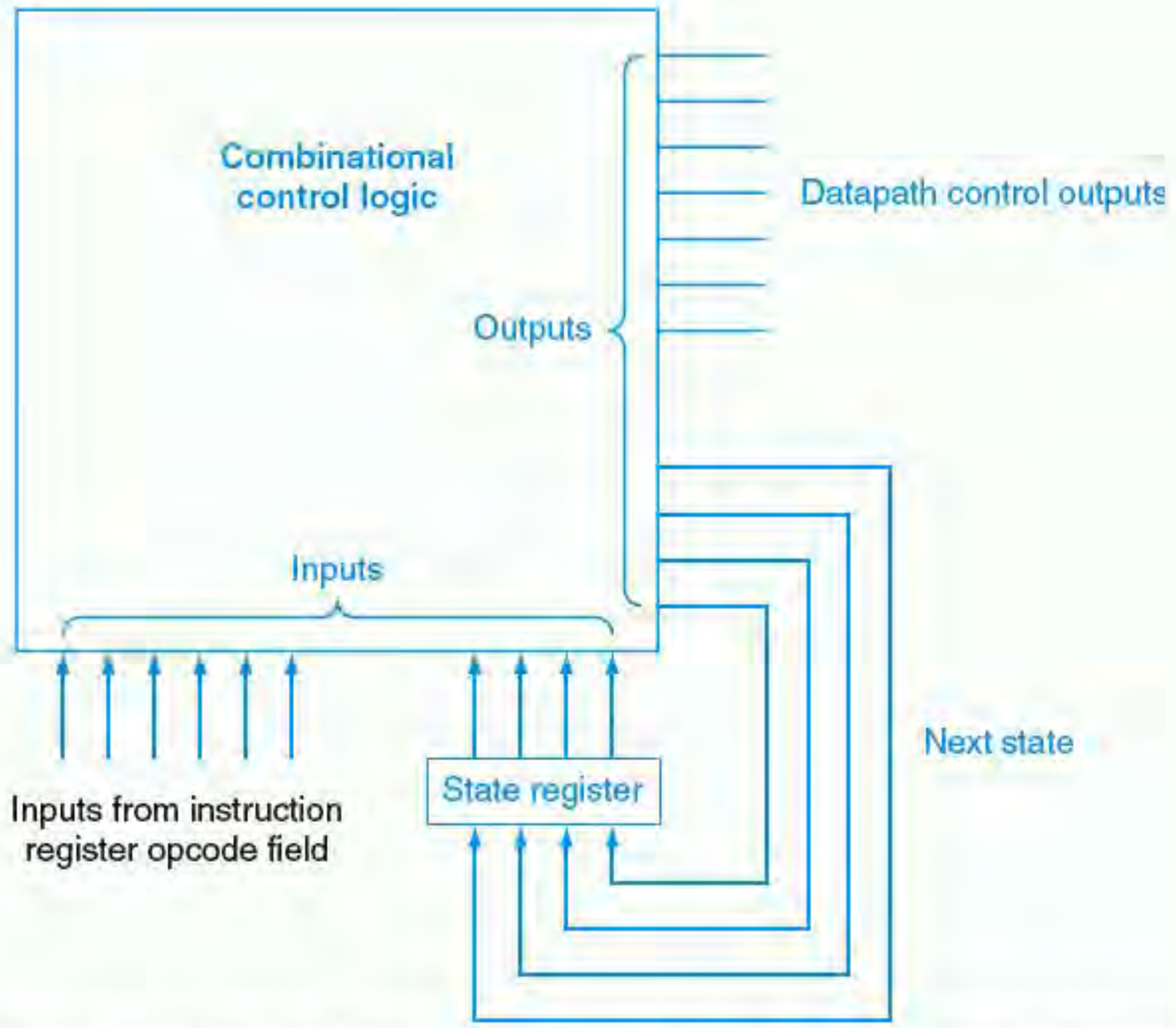


FIGURE 5.37 Finite state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state. The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational logic are the current state and any inputs used to determine the next state. In this case, the inputs are the instruction register opcode bits. Notice that in the finite state machine used in this chapter, the outputs depend only on the current state, not on the inputs. The Elaboration above explains this in more detail.

instructions are one universal example. There are many examples in the IA-32 architecture that require the use of multicycle control.

Elaboration: The style of finite state machine in Figure 5.37 is called a Moore machine, after Edward Moore. Its identifying characteristic is that the output depends only on the current state. For a Moore machine, the box labeled combinational control logic can be split into two pieces. One piece has the control output and only the state input, while the other has only the next-state output.

An alternative style of machine is a Mealy machine, named after George Mealy. The Mealy machine allows both the input and the current state to be used to determine the output. Moore machines have potential implementation advantages in speed and size of the control unit. The speed advantages arise because the control outputs, which are needed early in the clock cycle, do not depend on the inputs, but only on the current state. In Appendix C, when the implementation of this finite state machine is taken down to logic gates, the size advantage can be clearly seen.The potential disadvantage of a Moore machine is that it may require additional states. For example, in situations

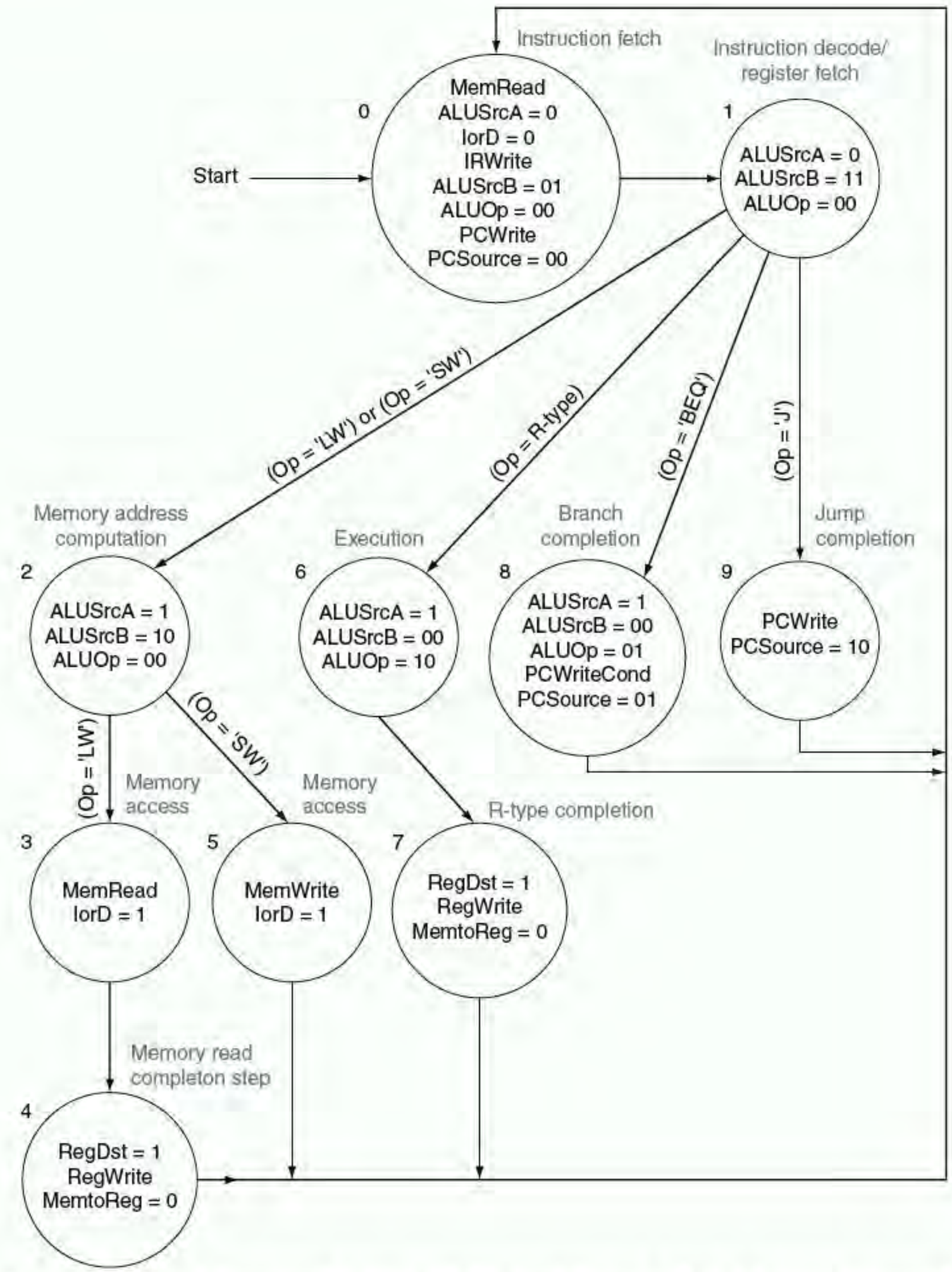


FIGURE 5.38 The complete finite state machine control for the datapath shown in Figure 5.28. The labels on the arcs are conditions that are tested to determine which state is the next state; when the next state is unconditional, no label is given. The labels inside the nodes indicate the output signals asserted during that state; we always specify the setting of a multiplexor control signal if the correct operation requires it. Hence, in some states a multiplexor control will be set to 0.

where there is a one-state difference between two sequences of states, the Mealy machine may unify the states by making the outputs depend on the inputs.

Understanding Program Performance

For a processor with a given clock rate, the relative performance between two code segments will be determined by the product of the CPI and the instruction count to execute each segment. As we have seen here, instructions can vary in their CPI, even for a simple processor. In the next two chapters, we will see that the introduction of pipelining and the use of caches create even larger opportunities for variation in the CPI. Although many factors that affect the CPI are controlled by the hardware designer, the programmer, the compiler, and software system dictate what instructions are executed, and it is this process that determines what the effective CPI for the program will be. Programmers seeking to improve performance must understand the role of CPI and the factors that affect it.

Check Yourself

1. True or false: Since the jump instruction does not depend on the register values or on computing the branch target address, it can be completed during the second state, rather than waiting until the third.
2. True, false, or maybe: The control signal PCWriteCond can be replaced by PCSource[0].

5.6 Exceptions

exception Also called interrupt. An unscheduled event that disrupts program execution; used to detect overflow.

interrupt An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

Control is the most challenging aspect of processor design: it is both the hardest part to get right and the hardest part to make fast. One of the hardest parts of control is implementing **exceptions** and **interrupts**—events other than branches or jumps that change the normal flow of instruction execution. An exception is an unexpected event from within the processor; arithmetic overflow is an example of an exception. An interrupt is an event that also causes an unexpected change in control flow but comes from outside of the processor. Interrupts are used by I/O devices to communicate with the processor, as we will see in Chapter 8.

Many architectures and authors do not distinguish between interrupts and exceptions, often using the older name *interrupt* to refer to both types of events. We follow the MIPS convention, using the term *exception* to refer to *any* unexpected change in control flow without distinguishing whether the cause is internal

or external; we use the term *interrupt* only when the event is externally caused. The Intel IA-32 architecture uses the word *interrupt* for all these events.

Interrupts were initially created to handle unexpected events like arithmetic overflow and to signal requests for service from I/O devices. The same basic mechanism was extended to handle internally generated exceptions as well. Here are some examples showing whether the situation is generated internally by the processor or externally generated:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Many of the requirements to support exceptions come from the specific situation that causes an exception to occur. Accordingly, we will return to this topic in Chapter 7, when we discuss memory hierarchies, and in Chapter 8, when we discuss I/O, and we better understand the motivation for additional capabilities in the exception mechanism. In this section, we deal with the control implementation for detecting two types of exceptions that arise from the portions of the instruction set and implementation that we have already discussed.

Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a machine, which determines the clock cycle time and thus performance. Without proper attention to exceptions during design of the control unit, attempts to add exceptions to a complicated implementation can significantly reduce performance, as well as complicate the task of getting the design correct.

How Exceptions Are Handled

The two types of exceptions that our current implementation can generate are execution of an undefined instruction and an arithmetic overflow. The basic action that the machine must perform when an exception occurs is to save the address of the offending instruction in the exception program counter (EPC) and then transfer control to the operating system at some specified address.

The operating system can then take the appropriate action, which may involve providing some service to the user program, taking some predefined action in response to an overflow, or stopping the execution of the program and reporting an error. After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution

of the program. In Chapter 7, we will look more closely at the issue of restarting the execution.

For the operating system to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it. There are two main methods used to communicate the reason for an exception. The method used in the MIPS architecture is to include a status register (called the *Cause register*), which holds a field that indicates the reason for the exception.

vectored interrupt An interrupt for which the address to which control is transferred is determined by the cause of the exception.

A second method is to use **vectored interrupts**. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception. For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:

Exception type	Exception vector address (in hex)
Undefined instruction	$C000\ 0000_{hex}$
Arithmetic overflow	$C000\ 0020_{hex}$

The operating system knows the reason for the exception by the address at which it is initiated. The addresses are separated by 32 bytes or 8 instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence. When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause.

We can perform the processing required for exceptions by adding a few extra registers and control signals to our basic implementation and by slightly extending the finite state machine. Let's assume that we are implementing the exception system used in the MIPS architecture. (Implementing vectored exceptions is no more difficult.) We will need to add two additional registers to the datapath:

- *EPC:* A 32-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)
- *Cause:* A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused. Assume that the low-order bit of this register encodes the two possible exception sources mentioned above: undefined instruction = 0 and arithmetic overflow = 1.

We will need to add two control signals to cause the EPC and Cause registers to be written; call these *EPCWrite* and *CauseWrite*. In addition, we will need a 1-bit control signal to set the low-order bit of the Cause register appropriately; call this signal *IntCause*. Finally, we will need to be able to write the *exception address*, which is the operating system entry point for exception handling, into the PC; in

the MIPS architecture, this address is $8000\ 0180_{hex}$. (The SPIM simulator for MIPS uses $8000\ 0080_{hex}$.) Currently, the PC is fed from the output of a three-way multiplexor, which is controlled by the signal PCSource (see Figure 5.28 on page 323). We can change this to a four-way multiplexor, with additional input wired to the constant value $8000\ 0180_{hex}$. Then PCSource can be set to 11_{two} to select this value to be written into the PC.

Because the PC is incremented during the first cycle of every instruction, we cannot just write the value of the PC into the EPC, since the value in the PC will be the instruction address plus 4. However, we can use the ALU to subtract 4 from the PC and write the output into the EPC. This requires no additional control signals or paths, since we can use the ALU to subtract, and the constant 4 is already a selectable ALU input. The data write port of the EPC, therefore, is connected to the ALU output. Figure 5.39 shows the multicycle datapath with these additions needed for implementing exceptions.

Using the datapath of Figure 5.39, the action to be taken for each different type of exception can be handled in one state apiece. In each case, the state sets the Cause register, computes and saves the original PC into the EPC, and writes the exception address into the PC. Thus, to handle the two exception types we are considering, we will need to add only the two states, but before we add them we must determine how to check for exceptions, since these checks will control the arcs to the new states.

How Control Checks for Exceptions

Now we have to design a method to detect these exceptions and to transfer control to the appropriate state in the exception states. Figure 5.40 shows the two new states (10 and 11) as well as their connection to the rest of the finite state control. Each of the two possible exceptions is detected differently:

- *Undefined instruction:* This is detected when no next state is defined from state 1 for the op value. We handle this exception by defining the next-state value for all op values other than `lw`, `sw`, 0 (R-type), `j`, and `beq` as state 10. We show this by symbolically using *other* to indicate that the op field does not match any of the opcodes that label arcs out of state 1 to the new state 10, which is used for this exception.
- *Arithmetic overflow:* The ALU, designed in Appendix B, included logic to detect overflow, and a signal called *Overflow* is provided as an output from the ALU. This signal is used in the modified finite state machine to specify an additional possible next state (state 11) for state 7, as shown in Figure 5.40.

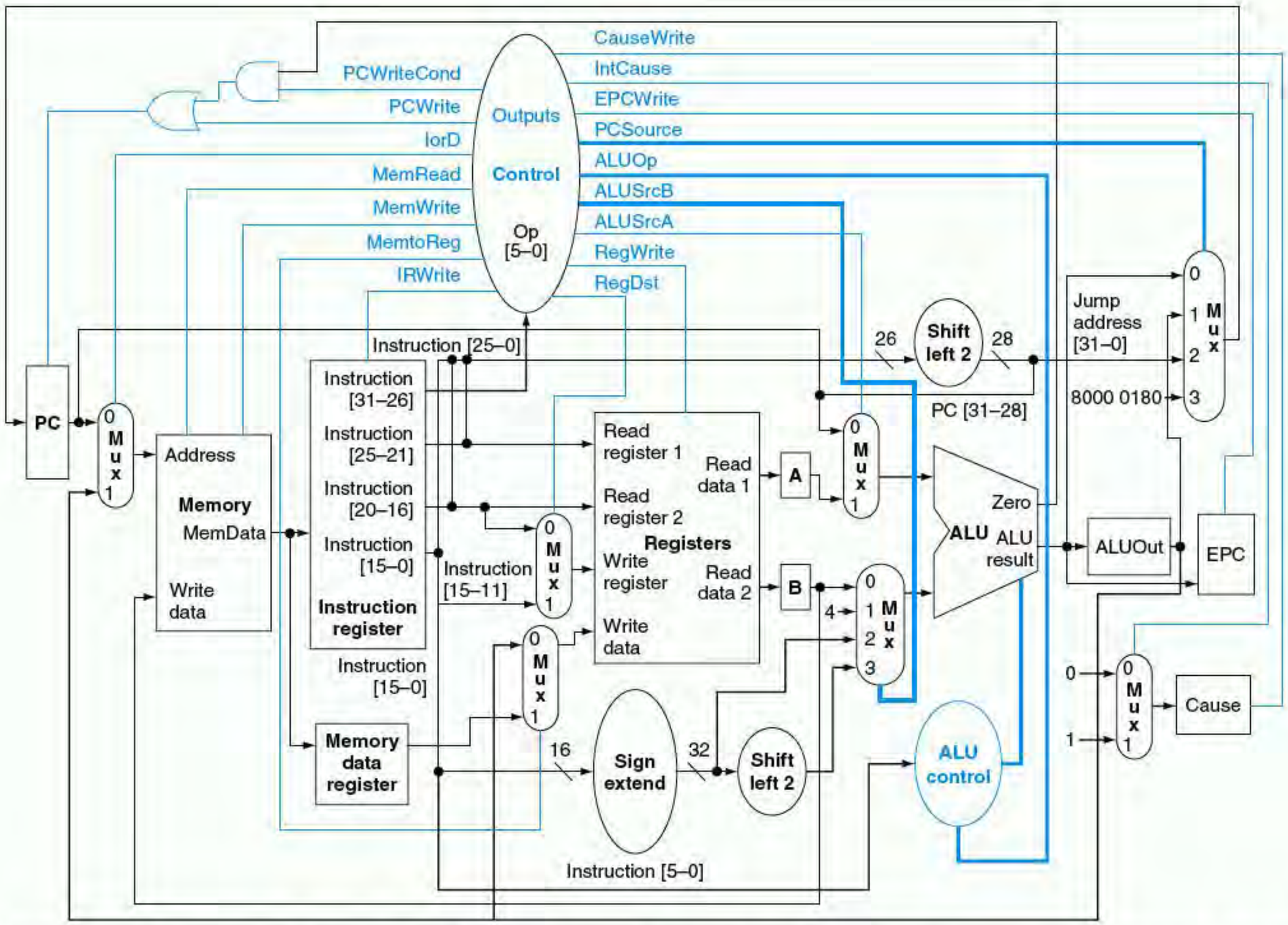


FIGURE 5.39 The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause and EPC registers, a multiplexor to control the value sent to the Cause register, an expansion of the multiplexor controlling the value written into the PC, and control lines for the added multiplexor and registers. For simplicity, this figure does not show the ALU overflow signal, which would need to be stored in a one-bit register and delivered as an additional input to the control unit (see Figure 5.40 to see how it is used).

Figure 5.40 represents a complete specification of the control for this MIPS subset with two types of exceptions. Remember that the challenge in designing the control of a real machine is to handle the variety of different interactions between instructions and other exception-causing events in such a way that the control logic remains both small and fast. The complex interactions that are possible are what make the control unit the most challenging aspect of hardware design.

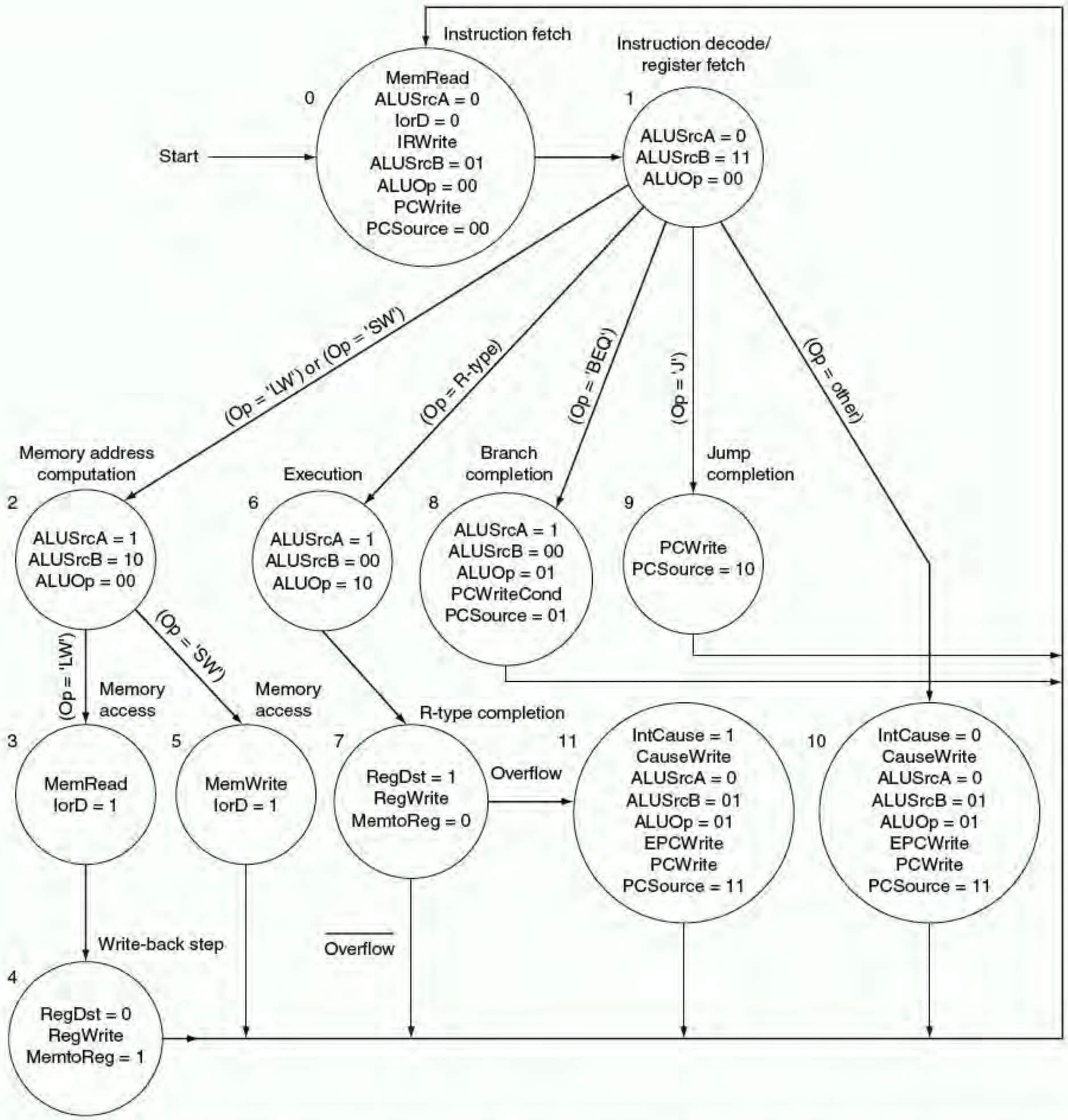


FIGURE 5.40 This shows the finite state machine with the additions to handle exception detection. States 10 and 11 are the new states that generate the appropriate control for exceptions. The branch out of state 1 labeled (*Op = other*) indicates the next state when the input does not match the opcode of any of `lw`, `sw`, 0 (R-type), `j`, or `beq`. The branch out of state 7 labeled *Overflow* indicates the action to be taken when the ALU signals an overflow.

Elaboration: If you examine the finite state machine in Figure 5.40 closely, you can see that some problems could occur in the way the exceptions are handled. For example, in the case of arithmetic overflow, the instruction causing the overflow completes writing its result because the overflow branch is in the state when the write completes. However, it's possible that the architecture defines the instruction as having no effect if the instruction causes an exception; this is what the MIPS instruction set architecture specifies. In Chapter 7, we will see that certain classes of exceptions require us to prevent the instruction from changing the machine state, and that this aspect of handling exceptions becomes complex and potentially limits performance.

Check Yourself Is this optimization proposed in the Check Yourself on page 340 concerning PCSource still valid in the extended control for exceptions shown in Figure 5.40 on page 345? Why or why not?



5.7 Microprogramming: Simplifying Control Design

Microprogramming is a technique for designing complex control units. It uses a very simple hardware engine that can then be programmed to implement a more complex instruction set. Microprogramming is used today to implement some parts of a complex instruction set, such as a Pentium, as well as in special-purpose processors. This section, which appears on the CD, explains the basic concepts and shows how they can be used to implement the MIPS multicycle control.



5.8 An Introduction to Digital Design Using a Hardware Design Language

Modern digital design is done using hardware description languages and modern computer-aided synthesis tools that can create detailed hardware designs from the descriptions using both libraries and logic synthesis. Entire books are written on such languages and their use in digital design. This section, which appears on the CD, gives a brief introduction and shows how a hardware design language, Verilog in this case, can be used to describe the MIPS multicycle control both behaviorally and in a form suitable for hardware synthesis.

5.9 Real Stuff: The Organization of Recent Pentium Implementations

The techniques described in this chapter for building datapaths and control units are at the heart of every computer. All recent computers, however, go beyond the techniques of this chapter and use pipelining. *Pipelining*, which is the subject of the next chapter, improves performance by overlapping the execution of multiple instructions, achieving throughput close to one instruction per clock cycle (like our single-cycle implementation) with a clock cycle time determined by the delay of individual functional units rather than the entire execution path of an instruction (like our multicycle design). The last Intel IA-32 processor without pipelining was the 80386, introduced in 1985; the very first MIPS processor, the R2000, also introduced in 1985, was pipelined.

Recent Intel IA-32 processors (the Pentium II, III, and 4) employ sophisticated pipelining approaches. These processors, however, are still faced with the challenge of implementing control for the complex IA-32 instruction set, described in Chapter 2. The basic functional units and datapaths in use in modern processors, while significantly more complex than those described in this chapter, have the same basic functionality and similar types of control signals. Thus the task of designing a control unit builds on the same principles used in this chapter.

Challenges Implementing More Complex Architectures

Unlike the MIPS architecture, the IA-32 architecture contains instructions that are very complex and can take tens, if not hundreds, of cycles to execute. For example, the string move instruction (`MOVS`) requires calculating and updating two different memory addresses as well as loading and storing a byte of the string. The larger number and greater complexity of addressing modes in the IA-32 architecture complicates implementation of even simple instructions similar to those on MIPS. Fortunately, a multicycle datapath is well structured to adapt to variations in the amount of work required per instruction that are inherent in IA-32 instructions. This adaptability comes from two capabilities:

1. A multicycle datapath allows instructions to take varying numbers of clock cycles. Simple IA-32 instructions that are similar to those in the MIPS architecture can execute in 3 or 4 clock cycles, while more complex instructions can take tens of cycles.
2. A multicycle datapath can use the datapath components more than once per instruction. This is critical to handling more complex addressing modes, as well as implementing more complex operations, both of which are present in the IA-32 architecture. Without this capability, the datapath

would need to be extended to handle the demands of the more complex instructions without reusing components, which would be completely impractical. For example, a single-cycle datapath, which doesn't reuse components, for the IA-32 would require several data memories and a very large number of ALUs.

microprogrammed control A method of specifying control that uses microcode rather than a finite state representation.

hardwired control An implementation of finite state machine control typically using programmable logic arrays (PLAs) or collections of PLAs and random logic.

microcode The set of microinstructions that control a processor.

Using the multicycle datapath and a **microprogrammed controller** provides a framework for implementing the IA-32 instruction set. The challenging task, however, is creating a high-performance implementation, which requires dealing with the diversity of the requirements arising from different instructions. Simply put, a high-performance implementation needs to ensure that the simple instructions execute quickly, and that the burden of the complexities of the instruction set penalize primarily the complex, less frequently used, instructions.

To accomplish this goal, every Intel implementation of the IA-32 architecture since the 486 has used a combination of **hardwired control** to handle simple instructions, and **microcoded** control to handle the more complex instructions. For those instructions that can be executed in a single pass through the datapath—those with complexity similar to a MIPS instruction—the hardwired control generates the control information and executes the instruction in one pass through the datapath that takes a small number of clock cycles. Those instructions that require multiple datapath passes and complex sequencing are handled by the microcoded controller that takes a larger number of cycles and multiple passes through the datapath to complete the execution of the instruction. The benefit of this approach is that it enables the designer to achieve low cycle counts for the simple instructions without having to build the enormously complex datapath that would be required to handle the full generality of the most complex instructions.

The Structure of the Pentium 4 Implementation

superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle.

microinstruction A representation of control using low-level instructions, each of which asserts a set of control signals that are active on a given clock cycle as well as specifies what microinstruction to execute next.

micro-operations The RISC-like instructions directly executed by the hardware in recent Pentium implementations.

Recent Pentium processors are capable of executing more than one instruction per clock, using an advanced pipelining technique, called **superscalar**. We describe how a superscalar processor works in the next chapter. The important thing to understand here is that executing more than one instruction per clock requires duplicating the datapath resources. The simplest way to think about this is that the processor has multiple datapaths, although these are tailored to handle one class of instructions: say, loads and stores, ALU operations, or branches. In this way, the processor is able to execute a load or store in the same clock cycle that it is also executing a branch and an ALU operation. The Pentium III and 4 allow up to three IA-32 instructions to execute in a clock cycle.

The Pentium III and Pentium 4 execute simple **microinstructions** similar to MIPS instructions, called **micro-operations** in Intel terminology. These microinstructions are fully self-contained operations that are initially about 70 bits wide. The control of datapath to implement these microinstructions is completely hard-

wired. This last level of control expands up to three microinstructions into about 120 control lines for the integer datapaths and 275 to over 400 control lines for the floating-point datapath—the latter number for the new SSE2 instructions included in the Pentium 4. This last step of expanding the microinstructions into control lines is very similar to the control generation for the single-cycle datapath or for the ALU control.

How is the translation between IA-32 instructions and microinstructions performed? In earlier Pentium implementations (i.e., the Pentium Pro, Pentium II, and Pentium III), the instruction decode unit would look at up to three IA-32 instructions at a time and use a set of PLAs to generate up to six microinstructions per cycle. With the significantly higher clock rate introduced in the Pentium 4, this solution was no longer adequate and an entirely new method of generating microinstructions was needed.

The solution adopted in the Pentium 4 is to include a **trace cache** of microinstructions, which is accessed by the IA-32 program counter. A trace cache is a sophisticated form of instruction cache, which we explain in detail in Chapter 7. For now, think of it as a buffer that holds the microinstructions that implement a given IA-32 instruction. When the trace cache is accessed with the address of the next IA-32 instruction to be executed, one of several events occurs:

trace cache An instruction cache that holds a sequence of instructions with a given starting address; in recent Pentium implementations the trace cache holds microoperations rather than IA-32 instructions.

- The translation of the IA-32 instruction is in the trace cache. In this case, up to three microinstructions are produced from the trace cache. These three microinstructions represent from one to three IA-32 instructions. The IA-32 PC is advanced one to three instructions depending on how many fit in the three microinstruction sequence.
- The translation of the IA-32 instruction is in the trace cache, but it requires more than four microinstructions to implement. For such complex IA-32 instructions, there is a microcode ROM; the control unit transfers control to the microprogram residing in the ROM. Microinstructions are produced from the microprogram until the more complex IA-32 instruction has been completed. The microcode ROM provides a total of more than 8000 microinstructions, with a number of sequences being shared among IA-32 instructions. Control then transfers back to fetching instructions from the trace cache.
- The translation of the designated IA-32 instruction is not in the trace cache. In this case, an IA-32 instruction decoder is used to decode the IA-32 instruction. If the number of microinstructions is four or less, the decoded microinstructions are placed in the trace cache, where they may be found on the next execution of this instruction. Otherwise, the microcode ROM is used to complete the sequence.

From one to three microinstructions are sent from the trace cache to the Pentium 4 microinstruction pipeline, which we describe in detail at the end of Chapter 6. The use of simple low-level hardwired control and simple datapaths for

handling the microinstructions together with the trace cache of decoded instructions allows the Pentium 4 to achieve impressive clock rates, similar to those for microprocessors implementing simpler instruction set architectures. Furthermore, the translation process, which combines direct hardwired control for simple instructions with microcoded control for complex instructions, allows the Pentium 4 to execute the simple, high-frequency instructions in the IA-32 instruction set at a high rate, yielding a low, and very competitive, CPI.

Understanding Program Performance

Although most of the Pentium 4 performance, ignoring the memory system, depends on the efficiency of the pipelined microoperations, the effectiveness of the front end in decoding IA-32 instructions can have a significant effect on performance. In particular, because of the structure of the decoder, using simpler IA-32 instructions that require four or fewer microoperations, and hence, avoiding a microcode **dispatch**, is likely to lead to better performance. Because of this implementation strategy (and a similar one on the Pentium III), compiler writers and assembly language programmers should try to make use of sequences of simple IA-32 instructions rather than more complex alternatives.

dispatch An operation in a microprogrammed control unit in which the next microinstruction is selected on the basis of one or more fields of a macroinstruction, usually by creating a table containing the addresses of the target microinstructions and indexing the table using a field of the macroinstruction. The dispatch tables are typically implemented in ROM or programmable logic array (PLA). The term *dispatch* is also used in dynamically scheduled processors to refer to the process of sending an instruction to a queue.

5.10 Fallacies and Pitfalls

Pitfall: Adding a complex instruction implemented with microprogramming may not be faster than a sequence using simpler instructions.

Most machines with a large and complex instruction set are implemented, at least in part, using microcode stored in ROM. Surprisingly, on such machines, sequences of individual simpler instructions are sometimes as fast as or even faster than the custom microcode sequence for a particular instruction.

How can this possibly be true? At one time, microcode had the advantage of being fetched from a much faster memory than instructions in the program. Since caches came into use in 1968, microcode no longer has such a consistent edge in fetch time. Microcode does, however, still have the advantage of using internal temporary registers in the computation, which can be helpful on machines with few general-purpose registers. The disadvantage of microcode is that the algorithms must be selected before the machine is announced and can't be

changed until the next model of the architecture. The instructions in a program, on the other hand, can utilize improvements in its algorithms at any time during the life of the machine. Along the same lines, the microcode sequence is probably not optimal for all possible combinations of operands.

One example of such an instruction in the IA-32 implementations is the move string instruction (`MOVS`) used with a repeat prefix that we discussed in Chapter 2. This instruction is often slower than a loop that moves words at a time, as we saw earlier in the Fallacies and Pitfalls (see page 350).

Another example involves the `LOOP` instruction, which decrements a register and branches to the specified label if the decremented register is not equal to zero. This instruction is designed to be used as the branch at the bottom of loops that have a fixed number of iterations (e.g., many *for* loops). Such an instruction, in addition to packing in some extra work, has benefits in minimizing the potential losses from the branch in pipelined machines (as we will see when we discuss branches in the next chapter).

Unfortunately, on all recent Intel IA-32 implementations, the `LOOP` instruction is always slower than the macrocode sequence consisting of simpler individual instructions (assuming that the small code size difference is not a factor). Thus, optimizing compilers focusing on speed never generate the `LOOP` instruction. This, in turn, makes it hard to motivate making `LOOP` fast in future implementations, since it is so rarely used!

Fallacy: If there is space in the control store, new instructions are free of cost.

One of the benefits of a microprogrammed approach is that control store implemented in ROM is not very expensive, and as transistor budgets grew, extra ROM was practically free. The analogy here is that of building a house and discovering, near completion, that you have enough land and materials left to add a room. This room wouldn't be free, however, since there would be the costs of labor and maintenance for the life of the home. The temptation to add "free" instructions can occur only when the instruction set is not fixed, as is likely to be the case in the first model of a computer. Because upward compatibility of binary programs is a highly desirable feature, all future models of this machine will be forced to include these so-called free instructions, even if space is later at a premium.

During the design of the 80286, many instructions were added to the instruction set. The availability of more silicon resource and the use of microprogrammed implementation made such additions seem painless. Possibly the largest addition was a sophisticated protection mechanism, which is largely unused today, but still must be implemented in newer implementations. This addition was motivated by a perceived need for such a mechanism and the desire to enhance microprocessor architectures to provide functionality equal to that of

larger computers. Likewise, a number of decimal instructions were added to provide decimal arithmetic on bytes. Such instructions are rarely used today because using binary arithmetic on 32 bits and converting back and forth to decimal representation is considerably faster. Like the protection mechanisms, the decimal instructions must be implemented in newer processors even if only rarely used.

5.11 Concluding Remarks

As we have seen in this chapter, both the datapath and control for a processor can be designed starting with the instruction set architecture and an understanding of the basic characteristics of the technology. In Section 5.3, we saw how the datapath for a MIPS processor could be constructed based on the architecture and the decision to build a single-cycle implementation. Of course, the underlying technology also affects many design decisions by dictating what components can be used in the datapath, as well as whether a single-cycle implementation even makes sense. Along the same lines, in the first portion of Section 5.5, we saw how the decision to break the clock cycle into a series of steps led to the revised multicycle datapath. In both cases, the top-level organization—a single-cycle or multicycle machine—together with the instruction set, prescribed many characteristics of the datapath design.

The BIG Picture

Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique. Figure 5.41 shows the variety of methods for specifying the control and moving from the specification to an implementation using some form of structured logic.

Similarly, the control is largely defined by the instruction set architecture, the organization, and the datapath design. In the single-cycle organization, these three aspects essentially define how the control signals must be set. In the multicycle design, the exact decomposition of the instruction execution into cycles, which is based on the instruction set architecture, together with the datapath, defines the requirements on the control.

Control is one of the most challenging aspects of computer design. A major reason is that designing the control requires an understanding of how all the com-

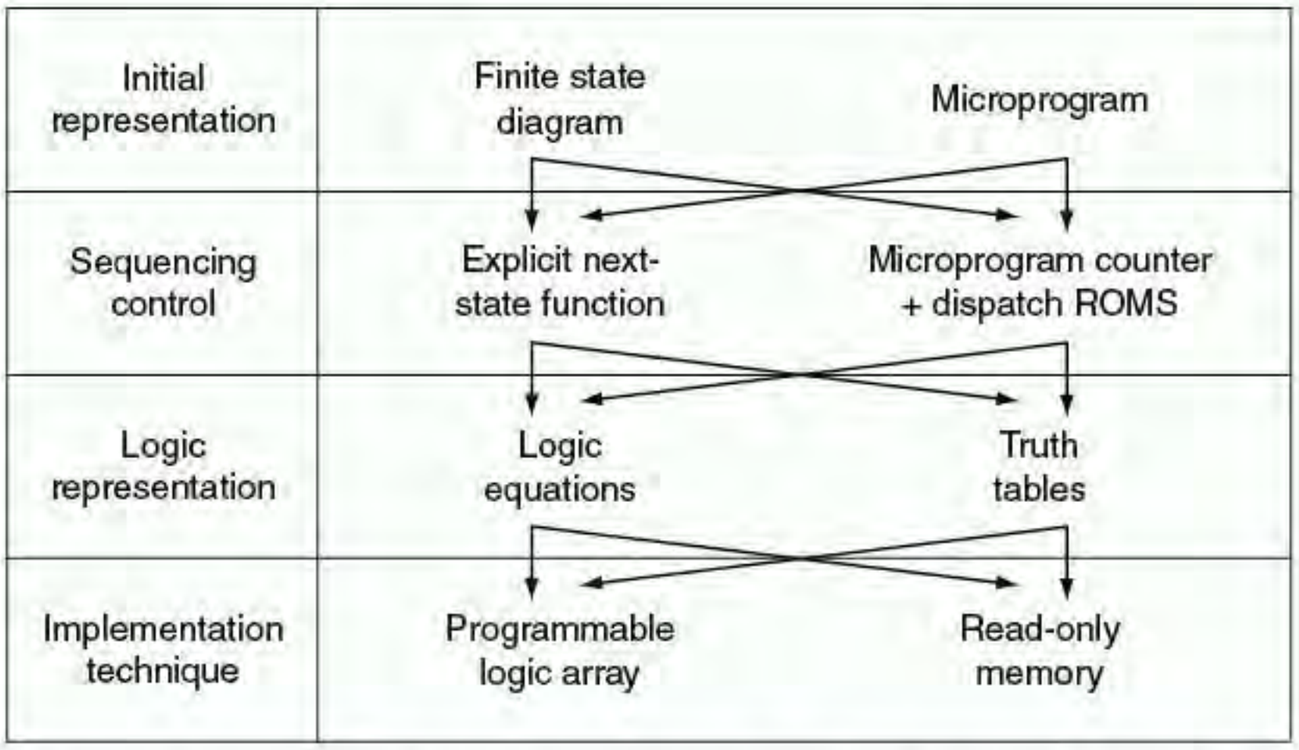


FIGURE 5.41 Alternative methods for specifying and implementing control. The arrows indicate possible design paths: any path from the initial representation to the final implementation technology is viable. Traditionally, "hardwired control" means that the techniques on the left-hand side are used, and "microprogrammed control" means that the techniques on the right-hand side are used.

ponents in the processor operate. To help meet this challenge, we examined two techniques for specifying control: finite state diagrams and microprogramming. These control representations allow us to abstract the specification of the control from the details of how to implement it. Using abstraction in this fashion is the major method we have to cope with the complexity of computer designs.

Once the control has been specified, we can map it to detailed hardware. The exact details of the control implementation will depend on both the structure of the control and on the underlying technology used to implement it. Abstracting the specification of control is also valuable because the decisions of how to implement the control are technology dependent and likely to change over time.



Historical Perspective and Further Reading

The rise of microprogramming and its effect on instruction set design and computer development is one of the more interesting interactions in the first few decades of the electronic computer. This story is the focus of the historical perspectives section on the CD.

5.13 Exercises

5.1 [6] <§5.2> Do we need combinational logic, sequential logic, or a combination of the two to implement each of the following:

a. multiplexor
b. comparator
c. incrementer/decrementer
d. barrel shifter
e. multiplier with shifters and adders
f. register
g. memory
h. ALU (the ones in single-cycle and multiple-cycle datapaths)
i. carry look-ahead adder
j. latch
k. general finite state machine (FSM)

5.2 [10] <§5.4> Describe the effect that a single stuck-at-0 fault (i.e., regardless of what it should be, the signal is always 0) would have for the signals shown below, in the single-cycle datapath in Figure 5.17 on page 307. Which instructions, if any, will not work correctly? Explain why.

Consider each of the following faults separately:

a. RegWrite = 0
b. ALUop0 = 0
c. ALUop1 = 0
d. Branch = 0
e. MemRead = 0
f. MemWrite = 0

5.3 [5] <§5.4> This exercise is similar to Exercise 5.2, but this time consider stuck-at-1 faults (the signal is always 1).

5.4 [5] <§5.4> For More Practice: Single Cycle Datapaths with Floating Point.

5.5 [5] <§5.4> For More Practice: Single Cycle Datapaths with Floating Point.

5.6 [10] <§5.4> For More Practice: Single Cycle Datapaths with Floating Point.

5.7 [2–3 months] <§§5.1–5.4> Using standard parts, build a machine that implements the single-cycle machine in this chapter.

5.8 [15] <§5.4> We wish to add the instruction `jr` (jump register) to the single-cycle datapath described in this chapter. Add any necessary datapaths and control signals to the single-cycle datapath of Figure 5.17 on page 307 and show the necessary additions to Figure 5.18 on page 308. You can photocopy these figures to make it faster to show the additions.

5.9 [10] <§5.4> This question is similar to Exercise 5.8 except that we wish to add the instruction `sll` (shift left logical), which is described in Section 2.5.

5.10 [15] <§5.4> This question is similar to Exercise 5.8 except that we wish to add the instruction `lui` (load upper immediate), which is described in Section 2.9.

5.11 [20] <§5.4> This question is similar to Exercise 5.8 except that we wish to add a variant of the `lw` (load word) instruction, which increments the index register after loading word from memory. This instruction (`l_inc`) corresponds to the following two instructions:

```
lw   $rs,L($rt)
addi $rt,$rt,1
```

5.12 [5] <§5.4> Explain why it is not possible to modify the single-cycle implementation to implement the load with increment instruction described in Exercise 5.12 without modifying the register file.

5.13 [7] <§5.4> Consider the single-cycle datapath in Figure 5.17. A friend is proposing to modify this single-cycle datapath by eliminating the control signal MemtoReg. The multiplexor that has MemtoReg as an input will instead use either the ALUSrc or the MemRead control signal. Will your friend's modification work? Can one of the two signals (MemRead and ALUSrc) substitute for the other? Explain.

5.14 [10] <§5.4> MIPS chooses to simplify the structure of its instructions. The way we implement complex instructions through the use of MIPS instructions is to decompose such complex instructions into multiple simpler MIPS ones. Show how MIPS can implement the instruction `swap $rs, $rt`, which swaps the contents of registers `$rs` and `$rt`. Consider the case in which there is an available register that may be destroyed as well as the care in which no such register exists.

If the implementation of this instruction in hardware will increase the clock period of a single-instruction implementation by 10%, what percentage of swap operations in the instruction mix would recommend implementing it in hardware?

5.15 [5] <§5.4> For More Practice: Effects of Faults in Control Multiplexors

5.16 [5] <§5.4> For More Practice: Effects of Faults in Control Multiplexors

5.17 [5] <§5.5> For More Practice: Effects of Faults in Control Multiplexors

5.18 [5] <§5.5> For More Practice: Effects of Faults in Control Multiplexors

5.19 [15] <§5.4> For More Practice: Adding Instructions to the Datapath

5.20 [15] <§5.4> For More Practice: Adding Instructions to the Datapath

5.21 [8] <§5.4> For More Practice: Adding Instructions to the Datapath

5.22 [8] <§5.4> For More Practice: Adding Instructions to the Datapath

5.23 [5] <§5.4> For More Practice: Adding Instructions to the Datapath

5.24 [10] <§5.4> For More Practice: Datapath Control Signals

5.25 [10] <§5.4> For More Practice: Datapath Control Signals

5.26 [15] <§5.4> For More Practice: Modifying the Datapath and Control

5.27 [8] <§5.4> Repeat Exercise 5.14, but apply your solution to the instruction load with increment: `l_incr $rt,Address($rs)`.

5.28 [5] <§5.4> The concept of the "critical path," the longest possible path in the machine, was introduced in 5.4 on page 315. Based on your understanding of the single-cycle implementation, show which units can tolerate more delays (i.e., are not on the critical path), and which units can benefit from hardware optimization. Quantify your answers taking the same numbers presented on page 315 (Section 5.4, "Example: Performance of Single-Cycle Machines").

5.29 [5] <§5.5> This exercise is similar to Exercise 5.2, but this time consider the effect that the stuck-at-0 faults would have on the *multiple-cycle* datapath in Figure 5.27. Consider each of the following faults:

a. RegWrite = 0
b. MemRead = 0
c. MemWrite = 0
d. IRWrite = 0
e. PCWrite = 0
f. PCWriteCond = 0.

5.30 [5] <§5.5> This exercise is similar to Exercise 5.29, but this time consider stuck-at-1 faults (the signal is always 1).

5.31 [[15] <§§5.4, 5.5> This exercise is similar to Exercise 5.13 but more general. Determine whether any of the control signals in the single-cycle implementation

can be eliminated and replaced by another existing control signal, or its inverse. Note that such redundancy is there because we have a very small set of instructions at this point, and it will disappear (or be harder to find) when we implement a larger number of instructions.

5.32 15] <§5.5> We wish to add the instruction `lui` (load upper immediate) described in Chapter 3 to the multicycle datapath described in this chapter. Use the same structure of the multicycle datapath of Figure 5.28 on page 323 and show the necessary modifications to the finite state machine of Figure 5.38 on page 339. You may find it helpful to examine the execution steps shown on pages 325 through 329 and consider the steps that will need to be performed to execute the new instruction. How many cycles are required to implement this instruction?

5.33 [15] <§5.5> You are asked to modify the implementation of lui in Exercise 5.32 in order to cut the execution time by 1 cycle. Add any necessary datapaths and control signals to the multicycle datapath of Figure 5.28 on page 323. You can photocopy existing figures to make it easier to show your modifications. You have to maintain the assumption that you don't know what the instruction is before the end of state 1 (end of second cycle). Please explicitly state how many cycles it takes to execute the new instruction on your modified datapath and finite state machine.

5.34 [20] <§5.5> This question is similar to Exercise 5.32 except that we wish to implement a new instruction `ldi` (load immediate) that loads a 32-bit immediate value from the memory location following the instruction address.

5.35 [15] <§5.5> Consider a change to the multiple-cycle implementation that alters the register file so that it has only one read port. Describe (via a diagram) any additional changes that will need to be made to the datapath in order to support this modification. Modify the finite state machine to indicate how the instructions will work, given your new datapath.

5.36 [15] <§5.5> Two important parameters control the performance of a processor: cycle time and cycles per instruction. There is an enduring trade-off between these two parameters in the design process of microprocessors. While some designers prefer to increase the processor frequency at the expense of large CPI, other designers follow a different school of thought in which reducing the CPI comes at the expense of lower processor frequency.

Consider the following machines, and compare their performance using the SPEC CPUint 2000 data from Figure 3.26 on page 228.

M1: The multicycle datapath of Chapter 5 with a 1 GHz clock.

M2: A machine like the multicycle datapath of Chapter 5, except that register updates are done in the same clock cycle as a memory read or ALU operation.

Thus in Figure 5.38 on page 339, states 6 and 7 and states 3 and 4 are combined. This machine has an 3.2 GHz clock, since the register update increases the length of the critical path.

M3: A machine like M2 except that effective address calculations are done in the same clock cycle as a memory access. Thus states 2, 3, and 4 can be combined, as can 2 and 5, as well as 6 and 7. This machine has a 2.8 GHz clock because of the long cycle created by combining address calculation and memory access.

Find out which of the machines is fastest. Are there instruction mixes that would make another machine faster, and if so, what are they?

5.37 [20] <§5.5> Your friends at C^3 (Creative Computer Corporation) have determined that the critical path that sets the clock cycle length of the multicycle datapath is memory access for loads and stores (not for fetching instructions). This has caused their newest implementation of the MIPS 30000 to run at a clock rate of 4.8 GHz rather than the target clock rate of 5.6 GHz. However, Clara at C^3 has a solution. If all the cycles that access memory are broken into two clock cycles, then the machine can run at its target clock rate.

Using the SPEC CPUint 2000 mixes shown in Chapter 3 (Figure 3.26 on page 228), determine how much faster the machine with the two-cycle memory accesses is compared with the 4.8 GHz machine with single-cycle memory access. Assume that all jumps and branches take the same number of cycles and that the set instructions and arithmetic immediate instructions are implemented as R-type instructions. Would you consider the further step of splitting instruction fetch into two cycles if it would raise the clock rate up to 6.4 GHz? Why?

5.38 [20] <§5.5> Suppose there were a MIPS instruction, called `bcmp`, that compares two blocks of words in two memory addresses. Assume that this instruction requires that the starting address of the first block is in register `$t1` and the starting address of the second block is in `$t2`, and that the number of words to compare is in `$t3` (which is `$t3≥0`). Assume the instruction can leave the result (the address of the first mismatch or zero if a complete match) in `$t1` and/or `$t2`. Furthermore, assume that the values of these registers as well as registers `$t4` and `t5` can be destroyed in executing this instruction (so that the registers can be used as temporaries to execute the instruction).

Write the MIPS assembly language program to implement (emulate the behavior of) block compare. How many instructions will be executed to compare two 100-word blocks? Using the CPI of the instructions in the multicycle implementation, how many cycles are needed for the 100-word block compare?

5.39 [2–3 months] <§§5.1–5.5> Using standard parts, build a machine that implements the multicycle machine in this chapter.

5.40 [15] <§5.5> For More Practice: Adding Instructions to the Datapath

5.41 [15] <§5.5> For More Practice: Adding Instructions to the Datapath

5.42 [15] <§5.5> For More Practice: Adding Instructions to the Datapath

5.43 [15] <§5.5> For More Practice: Adding Instructions to the Datapath

5.44 [15] <§5.5> For More Practice: Adding Instructions to the Datapath

5.45 [20] <§5.5> For More Practice: Adding Instructions to the Datapath

5.46 [10] <§5.5> For More Practice: Adding Instructions to the Datapath

5.47 [15] <§§5.1–5.5> For More Practice: Comparing Processor Performance

5.48 [20] <§5.5> For More Practice: Implementing Instructions in MIPS

5.49 [30] <§5.6> We wish to add the instruction `eret` (exception return) to the multicycle datapath described in this chapter. A primary task of the `eret` instruction is to reload the PC with the return address at which an exception, or error trap occurred. Suppose that if the processor is serving an error trap, then the PC has to be loaded from a register ErrorPC. Otherwise the processor is serving an exception) the PC has to be loaded from EPC. Suppose that there is a bit in the cause register called trap to encode an error trap when it occurs and to save the PC in the ErrorPC register. Add any necessary datapaths and control signals to the multicycle datapath of Figure 5.39 on page 344 to accommodate the trap/exception call and return, and show the necessary modifications to the finite state machine of Figure 5.40 on page 345 to implement the `eret` instruction. You can photocopy the figures to make it easier to show your modifications.

5.50 [6] <§5.6> Exceptions occur when a control flow change is required to handle an unexpected event in the processor. How can the cause and the instruction that caused the exception, be represented by the hardware in a MIPS machine? Give two examples for conditions that a processor can handle by restarting execution of instructions after handling the exception, and two others for exceptions that lead to program termination.

5.51 [6] <§5.6> Exception detection is an important aspect of exception handling. Try to identify the cycle in which the following exceptions can be detected for the multicycle datapath in Figure 5.28 on page 323.

Consider the following exceptions:

a. Divide by zero exception (suppose we use the same ALU for division in one cycle, and that it is recognized by the rest of the control)

b. Overflow exception

c. Invalid instruction

d. External interrupt

e. Invalid instruction memory address

f. Invalid data memory address

5.52 [15] <§5.6> For More Practice: Adding Instructions to the Datapath

5.53 [30] <§5.7> Microcode has been used to add more powerful instructions to an instruction set; let's explore the potential benefits of this approach. Devise a strategy for implementing the `bcmp` instruction described in Exercise 5.38 using the multicycle datapath and microcode. You will probably need to make some changes to the datapath in order to efficiently implement the `bcmp` instruction. Provide a description of your proposed changes and describe how the `bcmp` instruction will work. Are there any advantages that can be obtained by adding internal registers to the datapath to help support the `bcmp` instruction? Estimate the improvement in performance that you can achieve by implementing the instruction in hardware (as opposed to the software solution you obtained in Exercise 5.38) and explain where the performance increase comes from.

5.54 [30] <§5.7> For More Practice: Microcode

5.55 [30] <§5.7> For More Practice: Microcode

5.56 [5] <§5.7> For More Practice: Microcode

5.57 [30] <§5.8> Using the strategy you developed in Exercise 5.53, modify the MIPS microinstruction format described in Figure 5.7.1 and provide the complete microprogram for the `bcmp` instruction. Describe in detail how you extended the microcode so as to support the creation of more complex control structures (such as a loop) within the microcode. Has support for the `bcmp` instruction changed the size of the microcode? Will other instructions besides `bcmp` be affected by the change in the microinstruction format?

5.58 [5] <§5.8> A and B are registers defined through the following Verilog initialization code:

```
 reg A,B
initial begin
    A = 1;
    B = 2;
end
```

Analyze the following two segments of Verilog description lines, and compare the results of variables A and B, and the operation done in each example.

```
a)      always @(negedge clock) begin
            A = B;
            B = A;
        end
b)      always @(negedge clock) begin
            A <= B;
            B <= A;
        end
```

5.59 [15] <§§5.4, 5.8> Write the ALUControl module in combinational Verilog using the following form as the basis:

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
                    input ALUOp[1:0], FuncCode[5:0];
                    output ALUCtl[3:0];
                    ....
                    endmodule
```

5.60 [1 week] <§§5.3, 5.4, 5.8> Using a hardware simulation language such as Verilog, implement a functional simulator for the single-cycle version. Build your simulator using an existing library of parts, if such a library is available. If the parts contain timing information, determine what the cycle time of your implementation will be.

5.61 [2–4 hours] <§§4.7, 5.5, 5.8, 5.8> Extend the multicycle Verilog description in 5.8 by adding an implementation of the unsigned MIPS multiply instruction; assume it is implemented using the MIPS ALU and a shift and add operation.

5.62 [2-4 hours] <§§4.7, 5.5, 5.8, 5.9> Extend the multicycle Verilog description in 5.8 by adding an implementation of the unsigned MIPS divide instruction; assume it is implemented using the MIPS ALU with a one-bit-at-a-time algorithm.

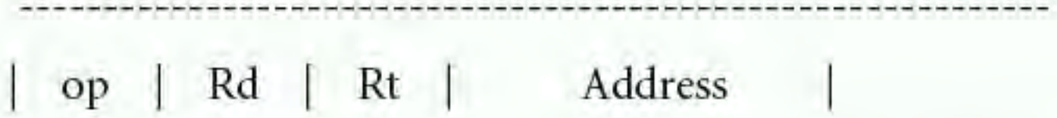
5.63 [1 week] <§§5.5, 5.8> Using a hardware simulation language such as Verilog, implement a functional simulator for a multicycle implementation of the design of a PowerPC processor. Build your simulator using an existing library of parts, if such a library is available. If the parts contain timing information, determine what the cycle time of your implementation will be.

Like MIPS, the PowerPC instructions are 32 bits each. Assume that your instruction set supports the following instruction formats:

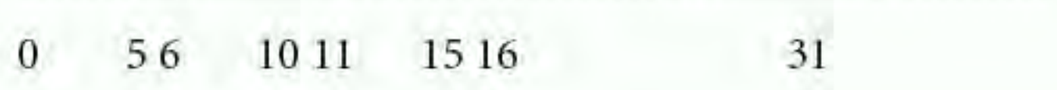
R-type



Load/store & immediate



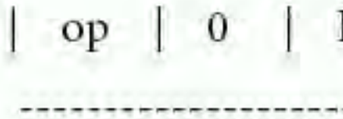
Branch conditional



Jump



RC-reg



---

0 1 2 3

Func field (22:30): Similar to MIPS, identifies function code.

RC bit(31): IF set (1), update the RC-reg control bits to reflect the results of the instruction (all R-type).

AA(30): 1 indicates that the given address is an absolute address;

0 indicates relative address.

LK: IF 1, updates LNKR (the link register), which can be later used for subroutine return implementation.

BI: Encodes the branch condition (e.g., beg -> BI = 2, blt -> BI = 0, etc.)

BD: Branch relative destination.

Your simplified PowerPC implementation should be able to implement the following instructions:

```
add:          add $Rd, $Rt, $Rs      ($Rd <- $Rt + $Rs)
              addi $Rd, $Rt, #n      ($Rd <- $Rt + #n)
subtract:     sub $Rd, $Rt, $Rs    ($Rd <- $Rt - $Rs)
              subi $Rd, $Rt, #n    ($Rd <- $Rt - #n)
load:       lw $Rd, Addr($Rt)    ($Rd <- Memory[$Rt + Addr])
store:     sw $Rd, Addr($Rt)    (Memory[$Rt + Addr] <- $Rd)
AND, OR:      and/or $Rd, $Rt, $Rs      ($Rd <- $Rt AND/OR $Rs)
              andi/ori $Rd, $Rt, #n      ($Rd <- $Rt AND/OR #n)
Jump:         jmp Addr  (PC <- Addr)
Branch conditional:  Beq Addr   (CR[2]==1? PC<- PC+BD : PC <-
PC+4)
subroutine call:  jal Addr (LNKR <- PC+4; PC<- Addr)
subroutine restore: Ret     (PC <- LNKR)
```

5.64 [Discussion] <§§5.7, 5.10, 5.11> Hypothesis: If the first implementation of an architecture uses microprogramming, it affects the instruction set architecture.

implementation do you think the architect had in mind when designing the instruction set architecture?

5.65 [Discussion] <§§5.7, 5.12> Wilkes invented microprogramming in large part to simplify construction of control. Since 1980, there has been an explosion of computer-aided design software whose goal is also to simplify construction of control. This has made control design much easier. Can you find evidence, based either on the tools or on real designs, that supports or refutes this hypothesis?

5.66 [Discussion] <§5.12> The MIPS instructions and the MIPS microinstructions have many similarities. What would make it difficult for a compiler to produce MIPS microcode rather than macrocode? What changes to the microarchitecture would make the microcode more useful for this application.

Answers to Check Yourself

§5.1, page 289: 3.
§5.2, page 292: false.
§5.3, page 299: A.
§5.4, page 318: Yes, MemtoReg and RegDst are inverses of one another. Yes, simply use the other signal and flip the order of the inputs to the multiplexor!
§5.5, page 340: 1. False. 2. Maybe: If the signal PCSource[0] is always set to zero when it is a don't care (which is most states), then it is identical to PCWriteCond.
§5.6, page 346: No, since the value of 11, which was formerly unused, is now used!
§5.7, page 5.7-13: 4 tables with 55 entries (don't forget the primary dispatch!)
§5.8, page 5.8-7: 1. 0, 1, 1, X, 0. 2. No, since state is not assigned on every path.

Computers in the Real World

Empowering the Disabled

Problem to solve: To overcome the obstacles faced by disabled people.

Solution: Use robotics, sensors, and computer control to replace or supplement damaged limbs and organs.

The picture on the right shows a system developed for a firefighter who was injured while fighting a fire. Sensors in latex fingers instantly register hot and cold, and an electronic interface in his artificial limb stimulates the nerve endings in his upper arm, which then pass the information to his brain. The $3000 system allows his hand to feel pressure and weight, so for the first time since losing his arms in a 1986 accident, he can pick up a can of soda without crushing it or having it slip through his fingers. The main enabling device is an electronic interface that can transmit signals to nerve endings in Whitten's upper arm, which then pass this information to his brain.



Firefighter Ken Whitten proudly displays his new bionic arm.

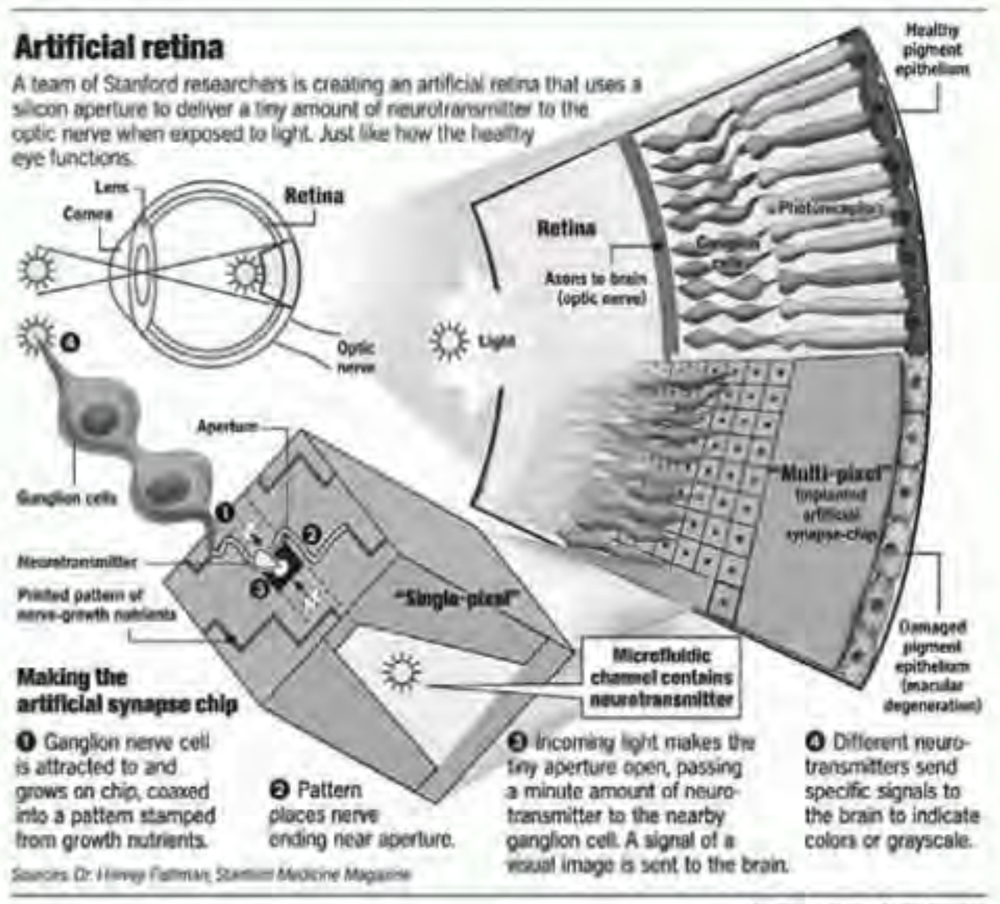
Harvey Fishman and Mark Peterman of Stanford have taken steps towards information technology that might someday treat age-related blindness. Their approach is to bypass the photoreceptors of the eye with a signal from a digital camera that connects directly to the visual system. They are developing a neural interface to the visual system called the artificial synapse chip. The challenge is to turn electrical signals into the chemicals that cells use to communicate. This chip is attached to cells, and from the cell's perspective the artificial synapse is simply a hole in the silicon. This hole is connected to a reservoir of neurotransmitter. When an electric field is applied to the chip, the neurotransmitter is pumped through the hole, stimulating nearby cells. In 2003 they have

created four artificial synapses on a chip one centimeter on a side.

Although this work is in its early stages, the potential is not limited to eye problems. According to Fishman, "Anywhere there's a severing of a nerve connection, there's a potential for us to reconnect it."

To learn more see these references on the ◎ library

- Rick Smolan and Jennifer Erwitt, *One Digital Day: How the Microchip Is Changing Our World*, Times Publishing, 1998
- Peterman et al, "The artificial synapse chip: A flexible retinal interface based on directed retinal cell growth and neurotransmitter stimulation," *Artificial Organs*: 27(11), November 18, 2003



Artificial retina using artificial synapse chips. From *The San Francisco Chronicle*, January 5, 2004.

6

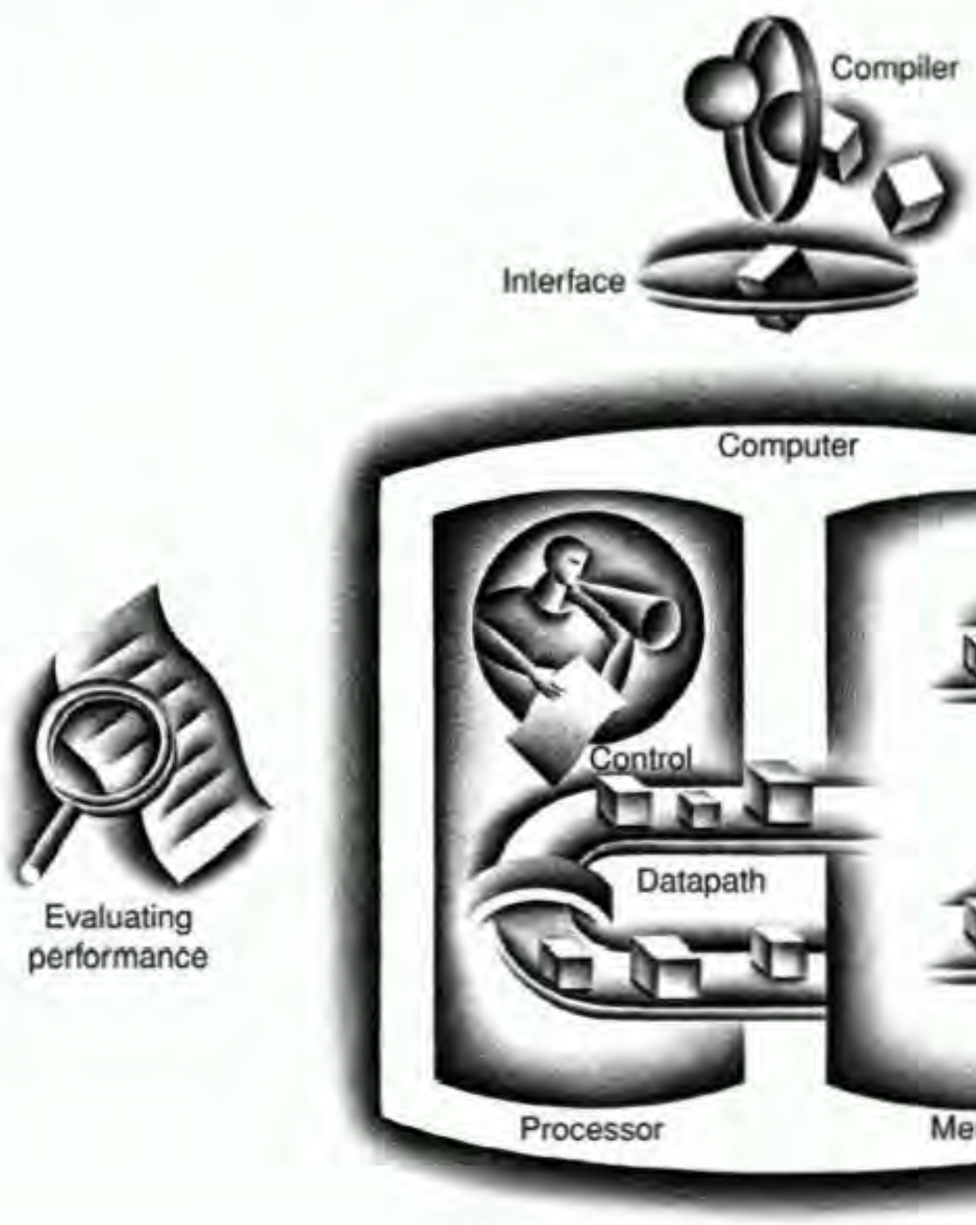
Enhancing Performance with Pipelining

Thus times do shift,
each thing his turn does hold;
New things succeed,
as former things grow old.

Robert Herrick
Hesperides: Ceremonies for Christmas Eve, 1648

6.1 **An Overview of Pipelining** 370
6.2 **A Pipelined Datapath** 384
6.3 **Pipelined Control** 399
6.4 **Data Hazards and Forwarding** 402
6.5 **Data Hazards and Stalls** 413
6.6 **Branch Hazards** 416
6.7 **Using a Hardware Description Language to Describe and Model a Pipeline** 426
6.8 **Exceptions** 427
6.9 **Advanced Pipelining: Extracting More Performance** 432
6.10 **Real Stuff: The Pentium 4 Pipeline** 448
6.11 **Fallacies and Pitfalls** 451
6.12 **Concluding Remarks** 452
6.13 **Historical Perspective and Further Reading** 454
6.14 **Exercises** 454

The Five Classic Components of a Computer



Never waste time.

American proverb

6.1 An Overview of Pipelining

pipelining An implementation technique in which multiple instructions are overlapped in execution, much like to an assembly line.

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 Sections 6.9 and 6.10 to see an introduction to the advanced pipelining techniques used in recent processors such as the Pentium III and 4. If you are interested in exploring the anatomy of a pipelined computer, this section is a good introduction to Sections 6.2 through 6.8.

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

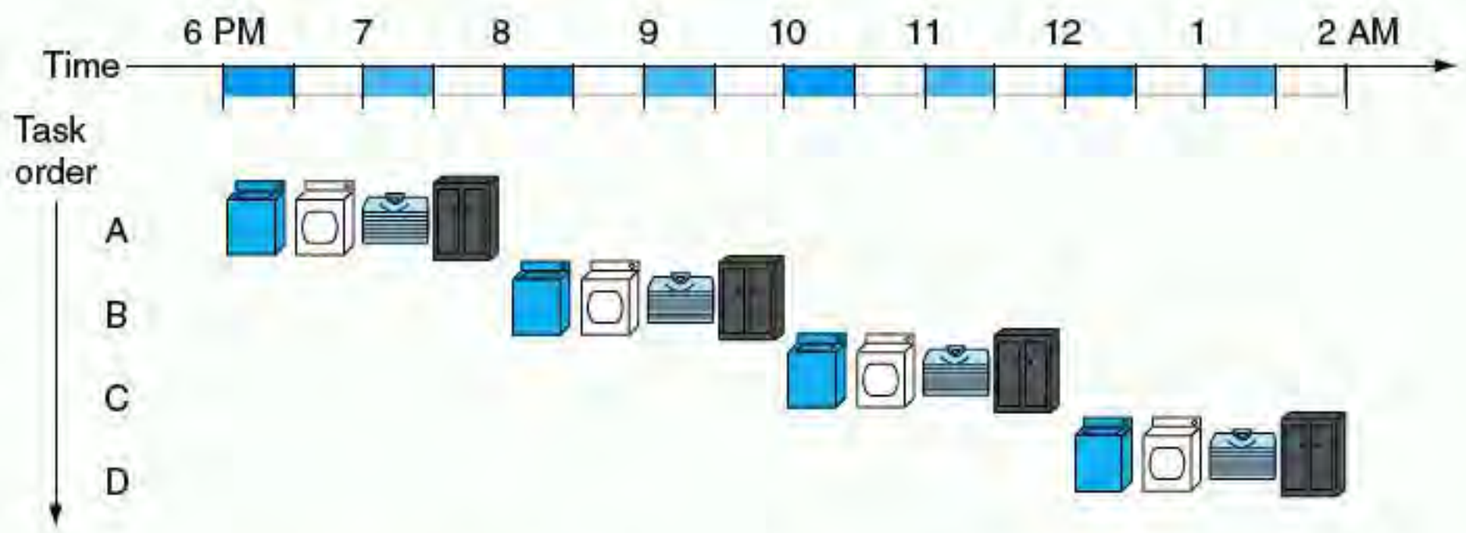
1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

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. Pipelining improves throughput of our laundry system without improving the time to complete a single load. Hence, pipelining would not decrease the time to complete one load of laundry, but when we have many loads of laundry to do, the improvement in throughput decreases the total time to complete the work.

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, in this case four: washing, drying, folding, and putting away. So, pipelined laundry is potentially four times faster than nonpipelined: 20 loads would



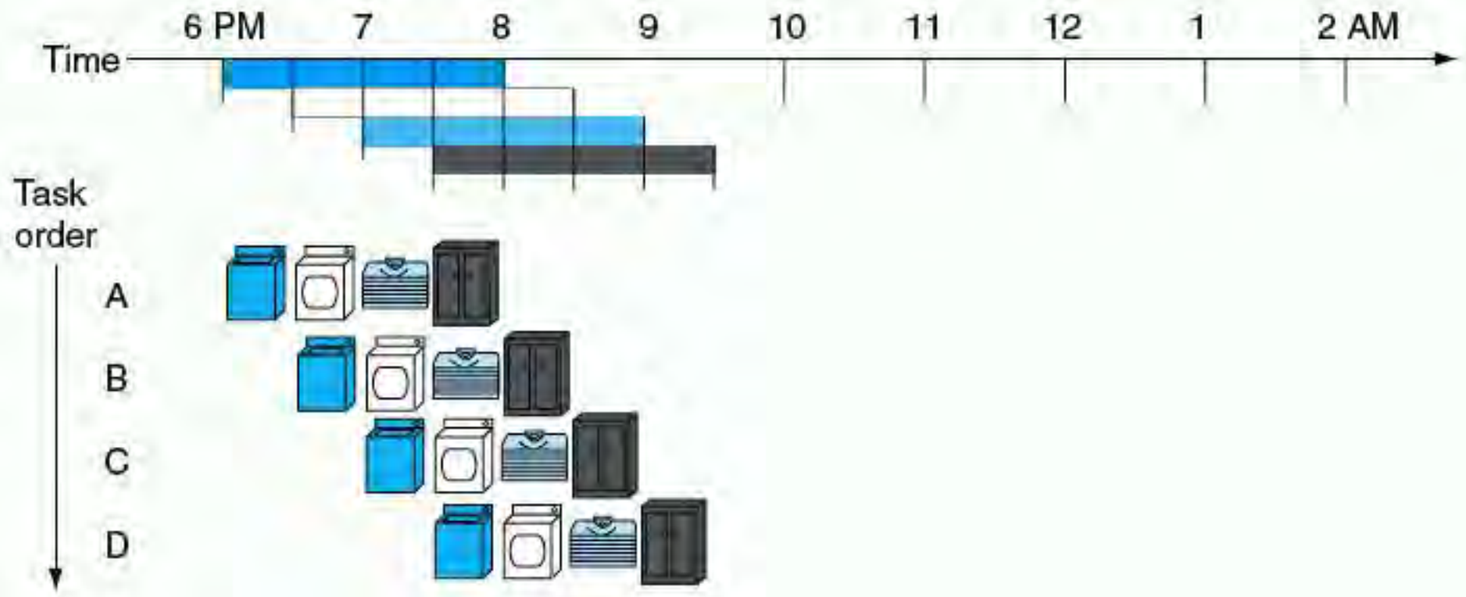


FIGURE 6.1 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for four loads of wash, while pipelined laundry 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 time line, but we really have just one of each resource.

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. Notice that at the beginning and end of the workload in the pipelined version in Figure 6.1, the pipeline is not completely full, this start-up and wind-down affects performance when the number of tasks is not large compared to the number of stages in the pipeline. If the number of loads is much larger than 4, then the stages will be full most of the time and the increase in throughput will be very close to 4.

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.

EXAMPLE

Single-Cycle versus Pipelined Performance

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 200 ps for memory access, 200 ps for ALU operation, and 100 ps 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

Figure 6.2 shows the time required for each of the eight instructions. 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 800 ps. 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 3×800 ns or 2400 ps.

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 800 ps even though some instructions can be as fast as 500 ps, the pipelined execution clock cycle must have the worst-case clock cycle of 200 ps even though some stages take only 100 ps. Pipelining still offers a fourfold performance improvement: the time between the first and fourth instructions is 3×200 ps or 600 ps.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `and`, `or`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

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

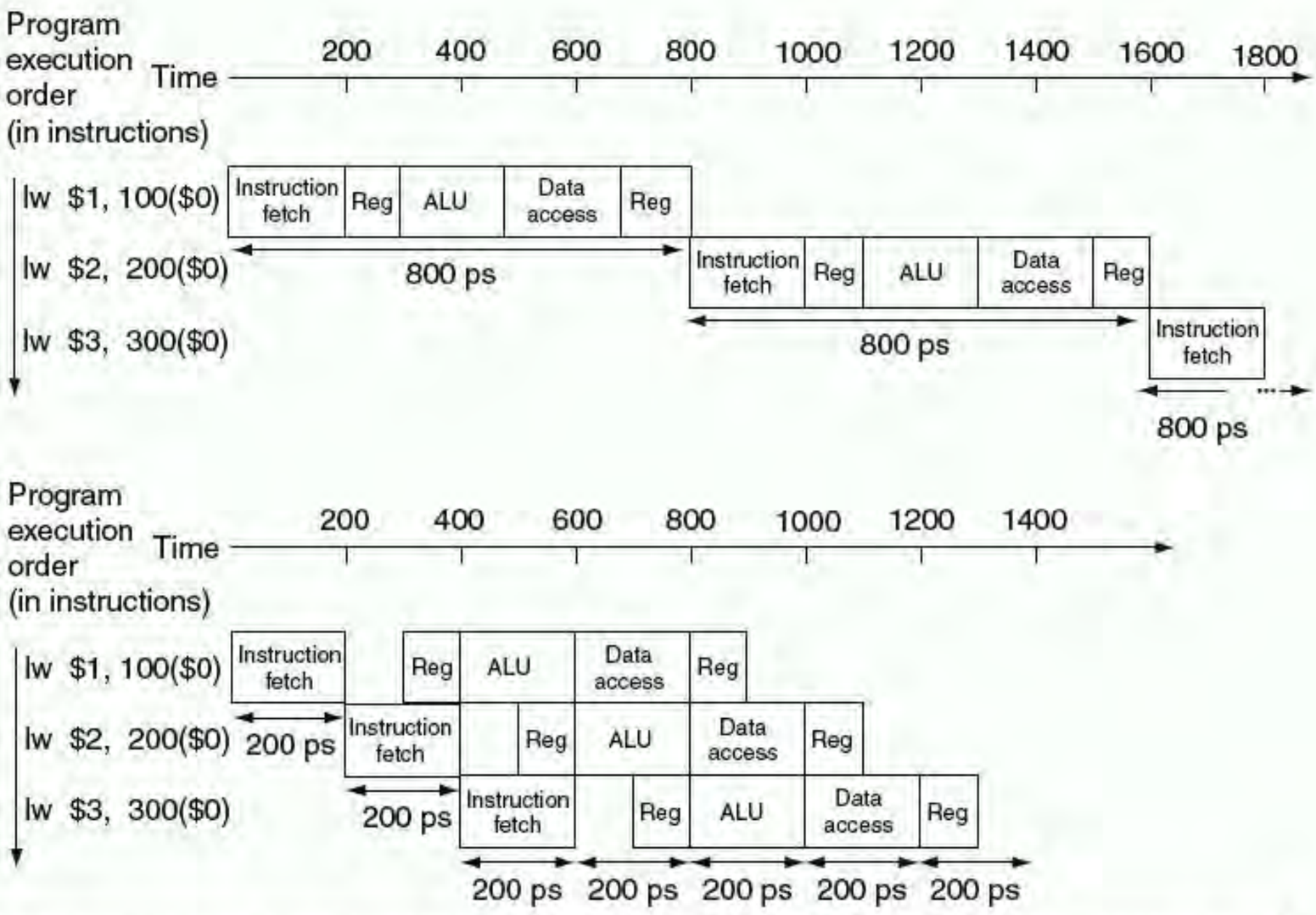


FIGURE 6.3 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 6.2. In this case we see a fourfold speedup on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 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.

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 processor—assuming ideal conditions—is equal to

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

Under ideal conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages; a five-stage pipeline is nearly five times faster.

The formula suggests that a five-stage pipeline should offer nearly a fivefold improvement over the 800 ps nonpipelined time, or a 160 ps clock cycle. The example shows, however, that the stages may be imperfectly balanced. In addition, pipelining involves some overhead, the source of which will be more clear shortly. Thus, the time per instruction in the pipelined processor 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 1400 ps versus 2400 ps. Of course, this is because the number of instructions is not large. What would happen if we increased the number of instructions? We could extend the previous figures to 1,000,003 instructions. We would add 1,000,000 instructions in the pipelined example; each instruction adds 200 ps to the total execution time. The total execution time would be 1,000,000 × 200 ps + 1400 ps, or 200,001,400 ps. In the nonpipelined example, we would add 1,0000,000 instructions, each taking 800 ps, so total execution time would be 1,000,000 × 800 ps + 2400 ps, or 800,002,400 ps. Under these ideal conditions, the ratio of total execution times for real programs on nonpipelined to pipelined processors is close to the ratio of times between instructions:

$$\frac{800{,}002{,}400 \text{ ps}}{200{,}001{,}400 \text{ ps}} \approx 4.00 \approx \frac{800 \text{ ps}}{200 \text{ ps}}$$

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 IA-32, where instructions vary from 1 byte to 17 bytes, pipelining is considerably more challenging. As we saw in Chapter 5, all recent implementations of the IA-32 architecture actually translate IA-32 instructions into simple microoperations that look like MIPS instructions. As we will see in Section 6.10, the Pentium 4 actually pipelines the microoperations rather than the native IA-32 instructions!

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 IA-32, stages 3 and 4 would expand to an address stage, memory stage, and then execute stage.

Fourth, as discussed in Chapter 2, operands must be aligned in memory. 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*, and there are three different types.

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.

structural hazard An occurrence in which a planned instruction cannot execute in the proper clock cycle because the hardware cannot support the combination of instructions that are set to execute in the given clock cycle.

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.

Data Hazards

data hazard Also called pipeline data hazard. An occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Suppose you found a sock at the folding station for which no match existed. One possible strategy is to run down to your room and search through your clothes bureau to see if you can find the match. Obviously, while you are doing the search, loads that have completed drying and are ready to fold and those that have finished washing and are ready to dry must wait.

In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry). 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 remove all such hazards, the results would not be satisfactory. These dependences happen just too often and the delay is just too long to expect the compiler to rescue us from this dilemma.

forwarding Also called **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

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. Adding extra hardware to retrieve 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 6.4 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 6.5 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

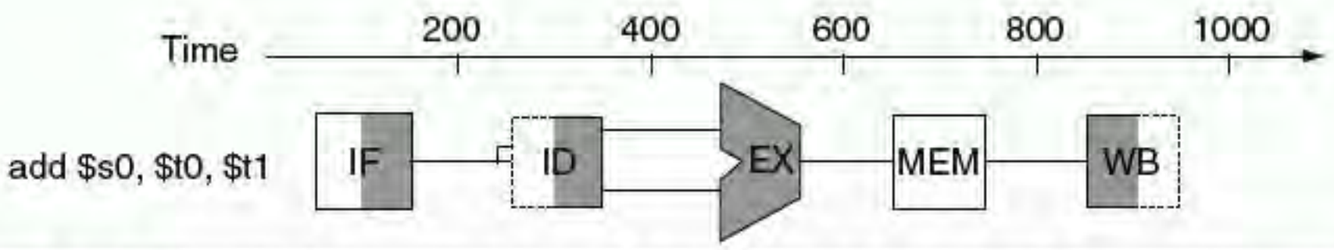


FIGURE 6.4 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 6.1 on page 371. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for 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.

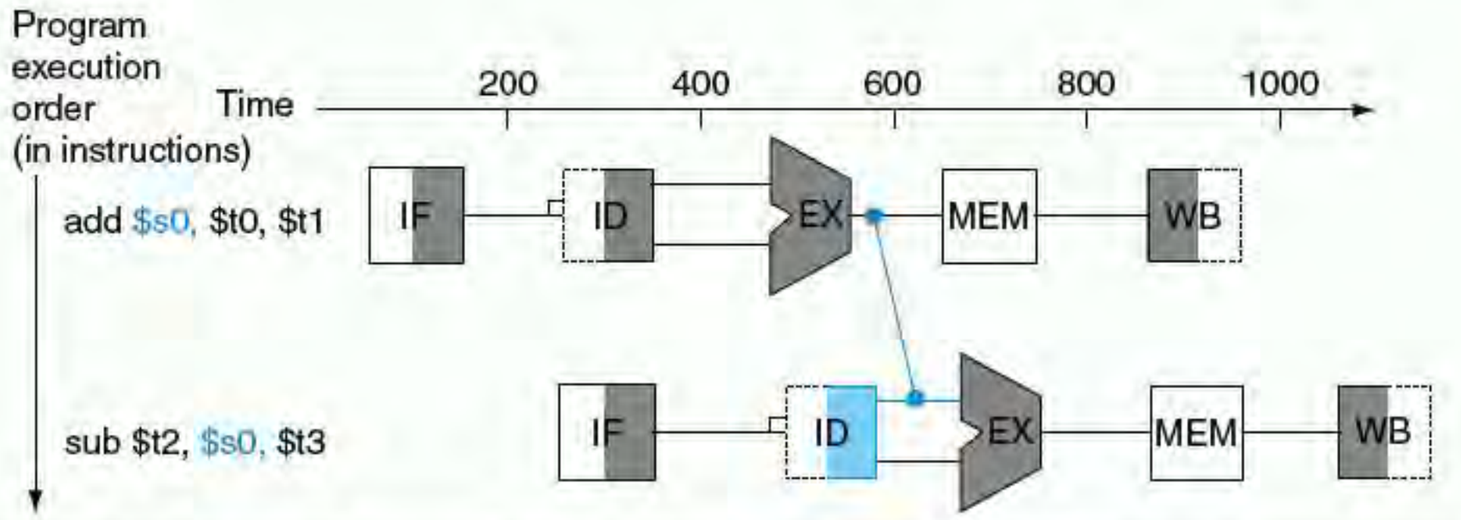


FIGURE 6.5 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `$s0` read in the second stage of `sub`.

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 6.5, 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 6.6 shows. 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. Section 6.5 shows how we can handle hard cases like these, using either hardware detection and stalls or software that treats the load delay like a branch delay.

load-use data hazard A specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested.

pipeline stall Also called bubble. A stall initiated in order to resolve a hazard.

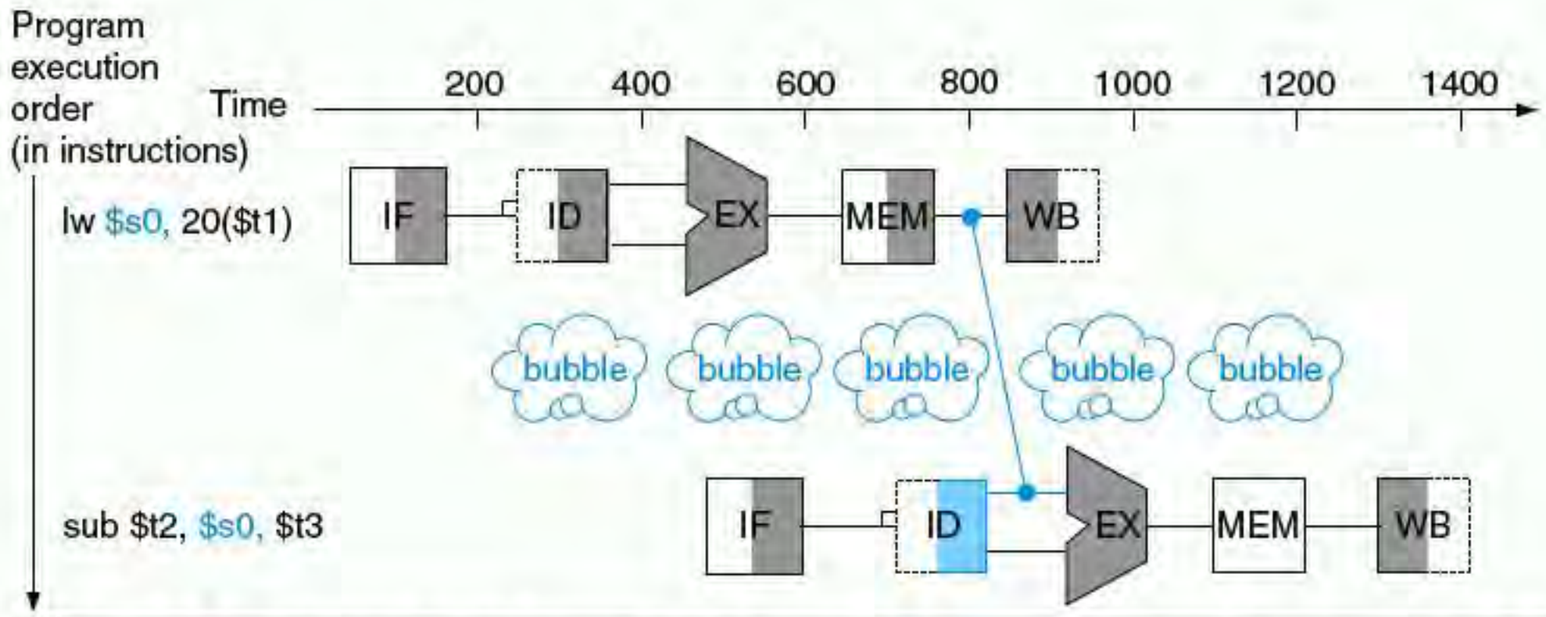


FIGURE 6.6 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backwards in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 6.5 shows the details of what really happens in the case of a hazard.

EXAMPLE

Reordering Code to Avoid Pipeline Stalls

Consider the following code segment in C:

```
A = B + E;
C = B + F;
```

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from `$t0`:

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
lw      $t4, 8($01)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

Find the hazards in the following code segment and reorder the instructions to avoid any pipeline stalls.

ANSWER

Both `add` instructions have a hazard because of their respective dependence on the immediately preceding `lw` instruction. Notice that bypassing eliminates several other potential hazards including the dependence of the first `add` on the first `lw` and any hazards for store instructions. Moving up the third `lw` instruction eliminates both hazards:

```
lw      $t1, 0($t0)
lw      $t2, 4($t1)
lw      $t4, 8($01)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

Forwarding yields another insight into the MIPS architecture, in addition to the four mentioned on page 374–375. Each MIPS instruction writes at most one result and does so near the end of the pipeline. Forwarding is harder if there are multiple results to forward per instruction or they need to write a result early on in instruction execution.

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.

Control Hazards

The third type of 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.

control hazard Also called **branch hazard**. An occurrence in which the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

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 is the first of two solutions to control hazards in the laundry room and its computer equivalent.

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. Notice that we must begin fetching the instruction following the branch on the very next clock cycle. But, the pipeline cannot possibly know what the next instruction should be, since it *only just received* the branch instruction from memory! Just as with laundry, one possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.

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 of the pipeline (see Section 6.6 for details). Even with this extra hardware, the pipeline involving conditional branches would look like Figure 6.7. The `lw` instruction, executed if the branch fails, is stalled one extra 200-ps clock cycle before starting.

EXAMPLE

Performance of "Stall on Branch"

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 3.26 on page 228 in Chapter 3 shows that branches are 13% of the instructions executed in SPECint2000. 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 1.13 and hence a slowdown of 1.13 versus the ideal case. Notice that this includes only branches and that jumps might also incur a stall.

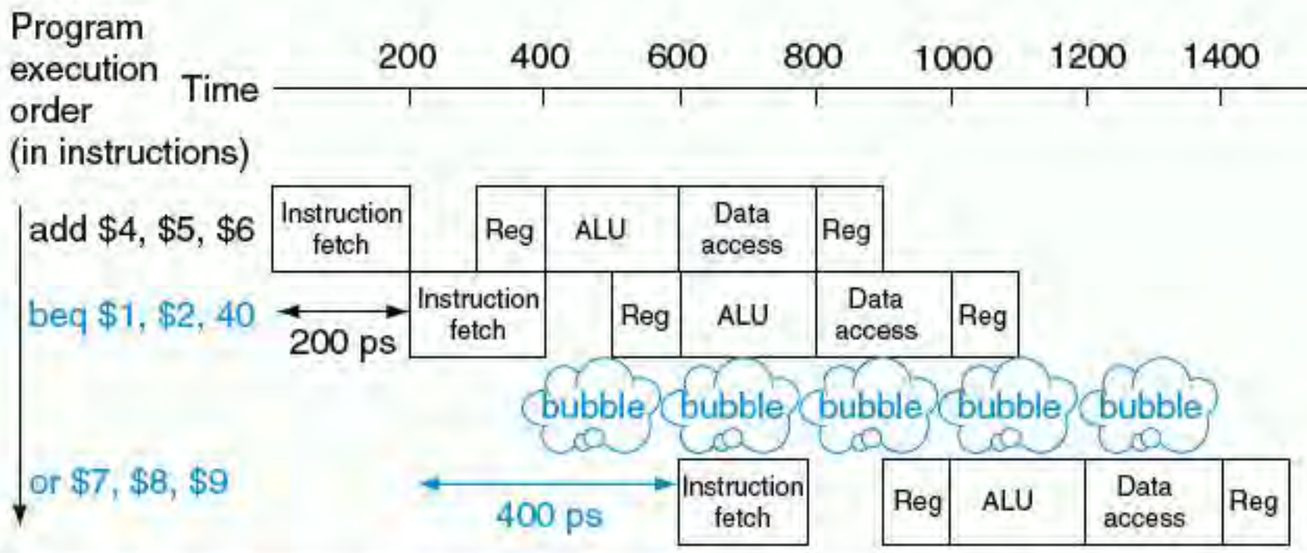


FIGURE 6.7 Pipeline showing stalling on every conditional branch as solution to control hazards. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 6.6. The effect on performance, however, is the same as would occur if a bubble were inserted.

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 be **untaken**. When you're right, the pipeline proceeds at full speed. Only when branches are taken does the pipeline stall. Figure 6.8 shows such an example.

untaken branch One that falls through to the successive instruction. A taken branch is one that causes transfer to the branch target.

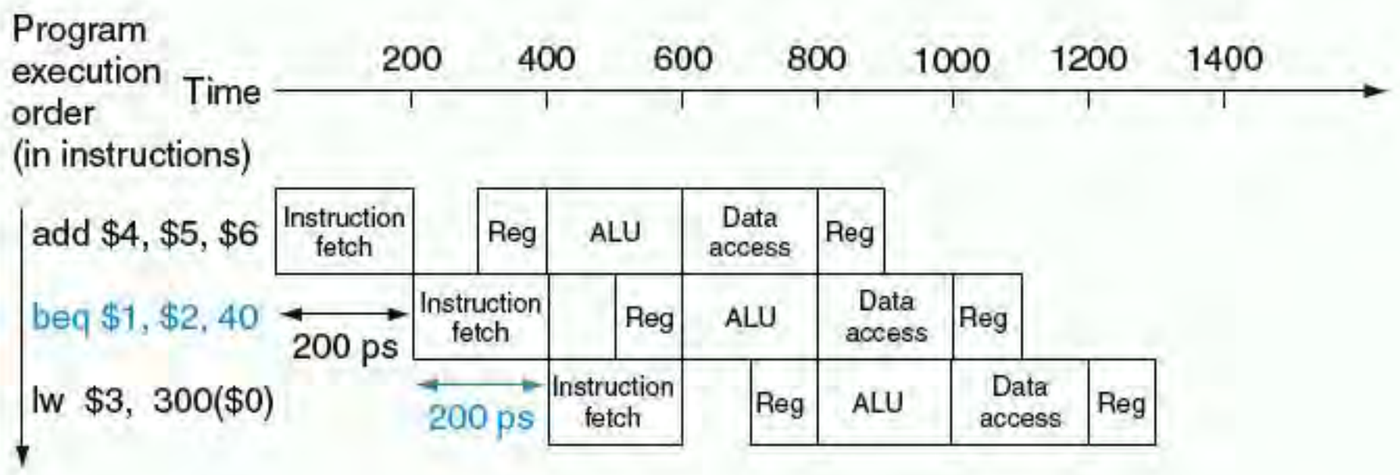


FIGURE 6.8 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 6.7, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. Section 6.6 will reveal the details.

branch prediction A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

A more sophisticated version of **branch prediction** would have some branches predicted as taken and some as 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 of branches is keeping a history for each branch as taken or untaken, and then using the recent past behavior to predict the future. As we will see later, the amount and type of history kept have become extensive with the result being that dynamic branch predictors can correctly predict branches with over 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. In our laundry analogy, we must stop taking new loads so that we can restart the load that we incorrectly predicted.

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 6.7 does not affect the branch and can be moved after the branch to fully hide the branch delay. Since delayed branches are useful when the branches are short, no processor uses a delayed branch of more than 1 cycle. For longer branch delays, hardware-based branch prediction is usually used.

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.

If you wish to focus more on the software and the performance implications of pipelining, you now have sufficient background to skip to Section 6.9. Section 6.9 introduces advanced pipelining concepts, such as superscalar and dynamic scheduling, and Section 6.10 examines the pipeline of the Pentium 4 microprocessor.

Alternatively, if you are interested in understanding how pipelining is implemented and the challenges of dealing with hazards, you can proceed to examine the design of a pipelined datapath, explained in Section 6.2, and the basic control, explained in Section 6.3. You can then use this understanding to explore the implementation of forwarding in Section 6.4, and the implementation of stalls in Section 6.5. You can then read Section 6.6 to learn more about solutions to branch hazards, and then see how exceptions are handled in Section 6.8.

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, also called the **latency**. For example, the five-stage pipeline still takes 5 clock cycles for the instruction to complete. In the terms used in Chapter 4, pipelining improves instruction *throughput* rather than individual instruction *execution time* or *latency*.

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.

The BIG Picture

latency (pipeline) The number of stages in a pipeline or the number of stages between two instructions during execution.

Understanding Program Performance

Outside of the memory system, the effective operation of the pipeline is usually the most important factor in determining the CPI of the processor and hence its performance. As we will see in Section 6.9, understanding the performance of a modern multiple-issue pipelined processor is complex and requires understanding more than just the issues that arise in a simple pipelined processor. Nonetheless, structural, data, and control hazards remain important in both simple pipelines and in more sophisticated ones.

For modern pipelines, structural hazards usually revolve around the floating-point unit, which may not be fully pipelined, while control hazards are usually more of a problem in integer programs, which tend to have higher branch frequencies as well as less predictable branches. Data hazards can be performance bottlenecks in both integer and floating-point programs. Often it is easier to deal with data hazards in floating-point programs because the lower branch frequency and more regular access patterns allow the compiler to try to schedule instructions to avoid hazards. It is more difficult to perform such optimizations in integer programs that have less regular access involving more use of pointers. As we will see in Section 6.9, there are more ambitious compiler and hardware techniques for reducing data dependences through scheduling.

Check Yourself

For each code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding:

Sequence 1	Sequence 2	Sequence 3
`lw  $t0,0($t0)` `add  $t1,$t0,$t0`	`add  $t1,$t0,$t0` `addi  $t2,$t0,#5` `addi  $t4,$t1,#5`	`addi  $t1,$t0,#1` `addi  $t2,$t0,#2` `addi  $t3,$t0,#2` `addi  $t3,$t0,#4` `addi  $t5,$t0,#5`

There is less in this than meets the eye.

Tallulah Bankhead, remark to Alexander Wollcott, 1922

6.2 A Pipelined Datapath

Figure 6.9 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

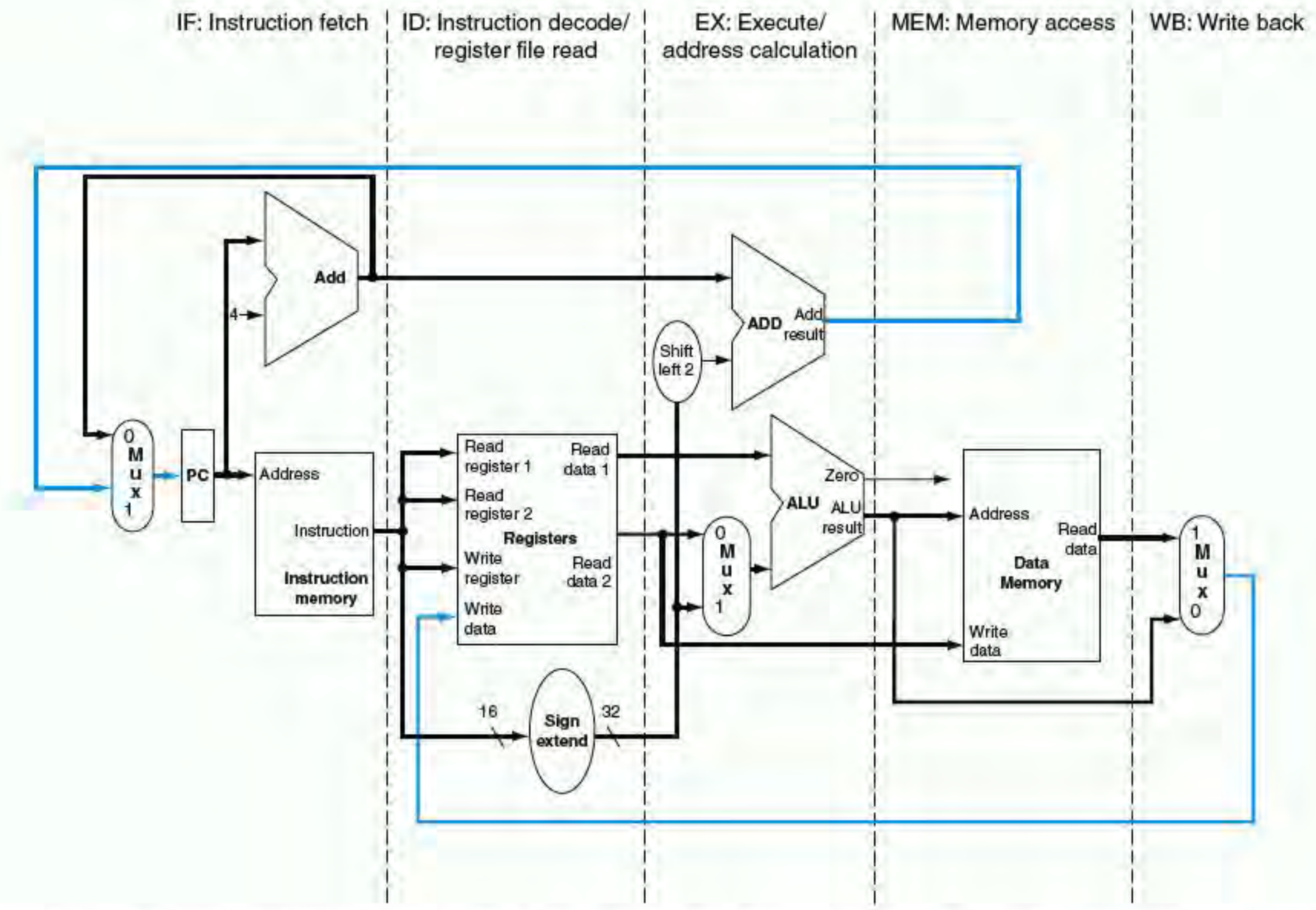


FIGURE 6.9 The single-cycle datapath from Chapter 5 (similar to Figure 5.17 on page 307). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

In Figure 6.9, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the five stages as they complete execution. Going back to our laundry analogy, clothes get cleaner, drier, and more organized as they move through the line, and they never move backwards.

There are, however, two exceptions to this left-to-right flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the datapath
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage

Data flowing from right to left does not affect the current instruction; only later instructions in the pipeline are influenced by these reverse data movements. Note that the first right-to-left arrow can lead to data hazards and the second leads to control hazards.

One way to show what happens in pipelined execution is to pretend that each instruction has its own datapath, and then to place these datapaths on a time line to show their relationship. Figure 6.10 shows the execution of the instructions in Figure 6.3 by displaying their private datapaths on a common time line. We use a stylized version of the datapath in Figure 6.9 to show the relationships in Figure 6.10.

Figure 6.10 seems to suggest that three instructions need three datapaths. In Chapter 5, we added registers to hold data so that portions of the datapath could be shared during instruction execution; we use the same technique here to share the multiple datapaths. For example, as Figure 6.10 shows, the instruction memory is used during only one of the five stages of an instruction, allowing it to be shared by other instructions during the other four stages.

To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register. Similar arguments apply to every pipeline stage, so we must place registers wherever there are dividing lines between stages in Figure 6.9. This change is similar to the registers added in Chapter 5 when we went from a single-cycle to a multicycle datapath. Returning to our laundry analogy, we might have a basket between each pair of stages to hold the clothes for the next step.

Figure 6.11 shows the pipelined datapath with the pipeline registers highlighted. All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID.

Notice that there is no pipeline register at the end of the write-back stage. All instructions must update some state in the processor—the register file, memory, or the PC—so a separate pipeline register is redundant to the state that is updated. For example, a load instruction will place its result in 1 of the 32 registers, and any later instruction that needs that data will simply read the appropriate register.

Of course, every instruction updates the PC, whether by incrementing it or by setting it to a branch destination address. The PC can be thought of as a pipeline register: one that feeds the IF stage of the pipeline. Unlike the shaded pipeline reg-

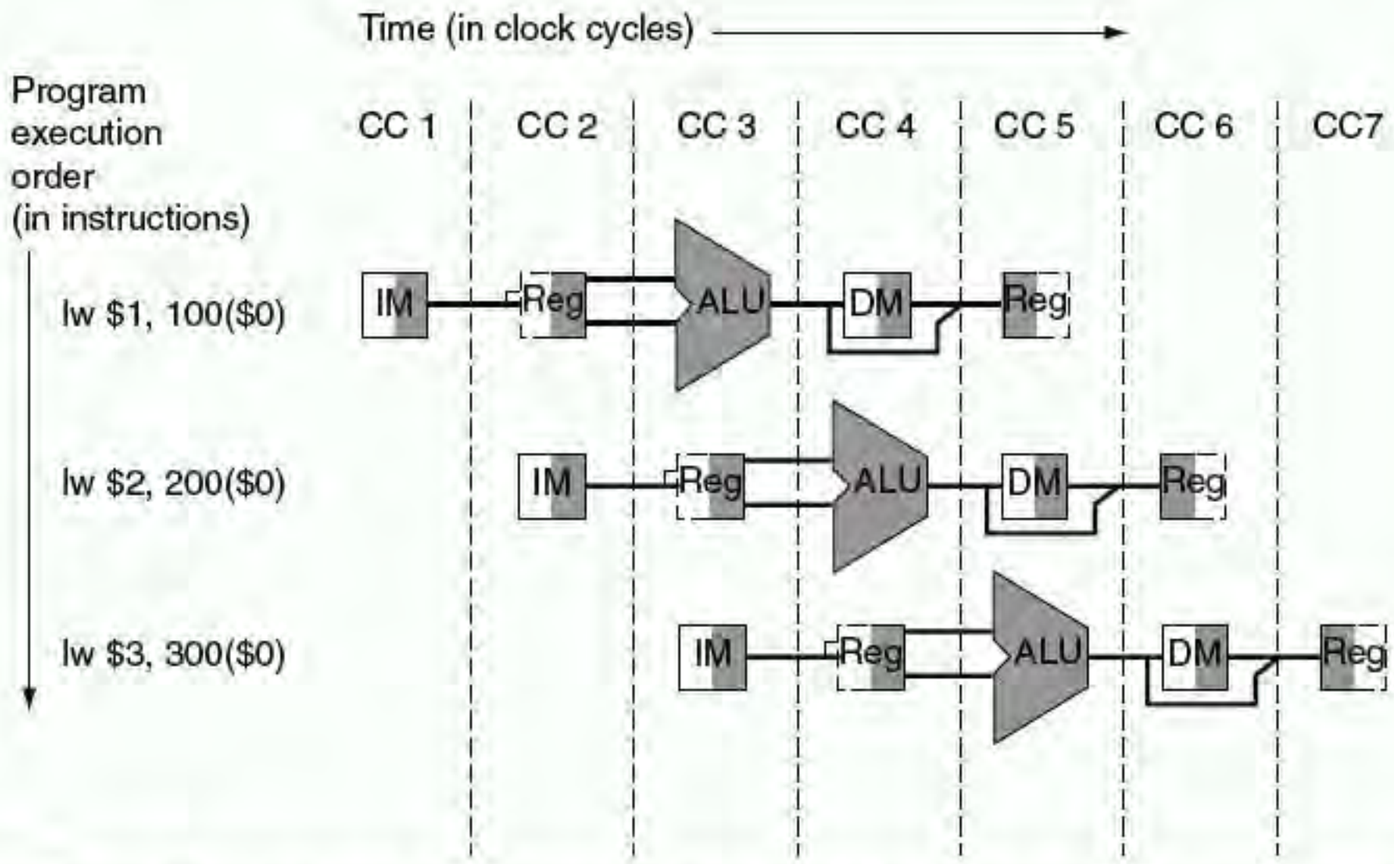


FIGURE 6.10 Instructions being executed using the single-cycle datapath in Figure 6.9, assuming pipelined execution. Similar to Figures 6.4 through 6.6, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 6.9. *IM* represents the instruction memory and the PC in the instruction fetch stage, *Reg* stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on. To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB). This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

isters in Figure 6.11, however, the PC is part of the visible architectural state; its contents must be saved when an exception occurs, while the contents of the pipeline registers can be discarded. In the laundry analogy, you could think of the PC as corresponding to the basket that holds the load of dirty clothes before the wash step!

To show how the pipelining works, throughout this chapter we show sequences of figures to demonstrate operation over time. These extra pages would seem to require much more time for you to understand. Fear not; the sequences take much less time than it might appear because you can compare them to see what changes occur in each clock cycle. Sections 6.4 and 6.5 describe what happens when there are data hazards between pipelined instructions; ignore them for now.

Figures 6.12 through 6.14, our first sequence, show the active portions of the datapath highlighted as a load instruction goes through the five stages of pipelined

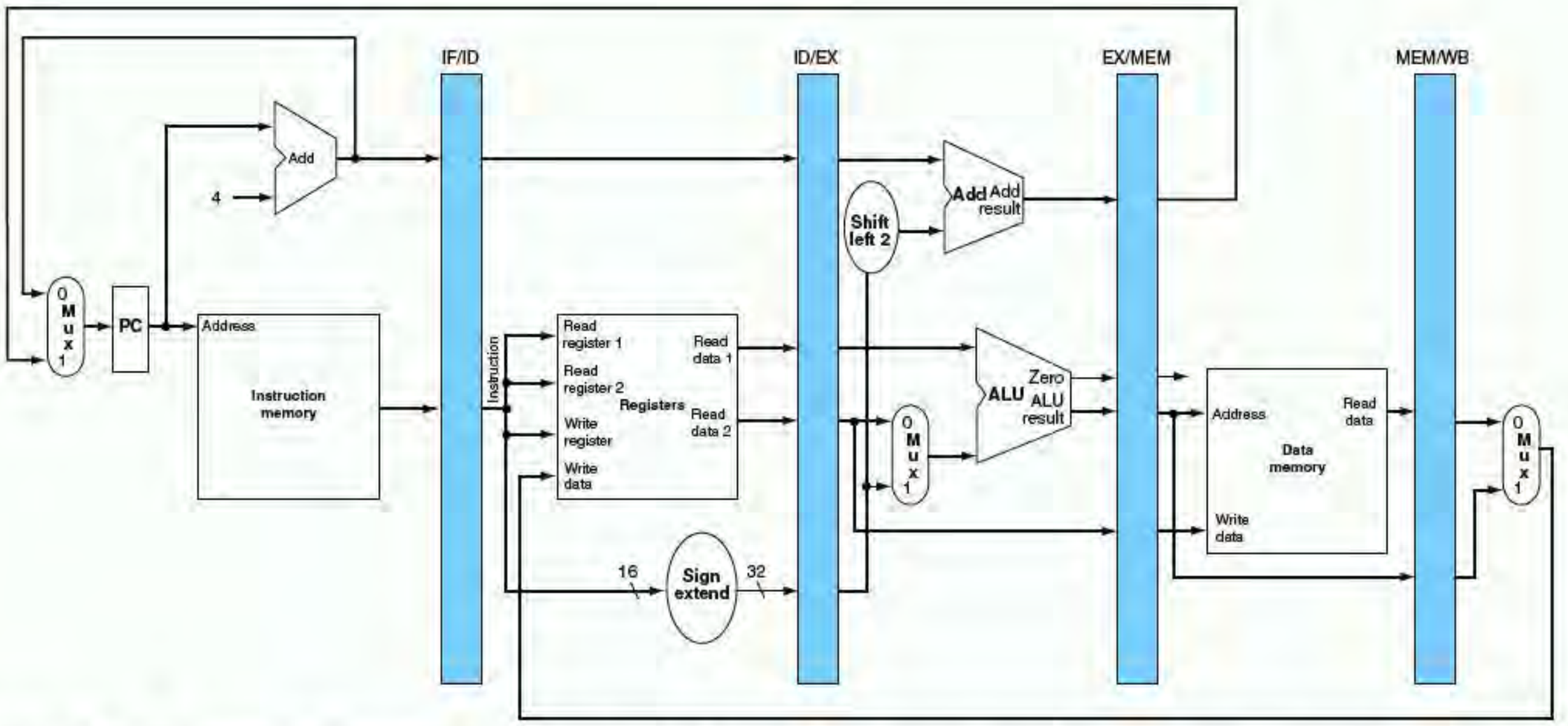


FIGURE 6.11 The pipelined version of the datapath in Figure 6.9. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

execution. We show a load first because it is active in all five stages. As in Figures 6.4 through 6.11, we highlight the *right half* of registers or memory when they are being *read* and highlight the *left half* when they are being *written*. We show the instruction abbreviation `lw` with the name of the pipe stage that is active in each figure. The five stages are the following:

1. *Instruction fetch:* The top portion of Figure 6.12 shows the instruction being read from memory using the address in the PC and then placed in the IF/ID pipeline register. The IF/ID pipeline register is similar to the Instruction register in Figure 5.26 on page 320. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as `beq`. The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

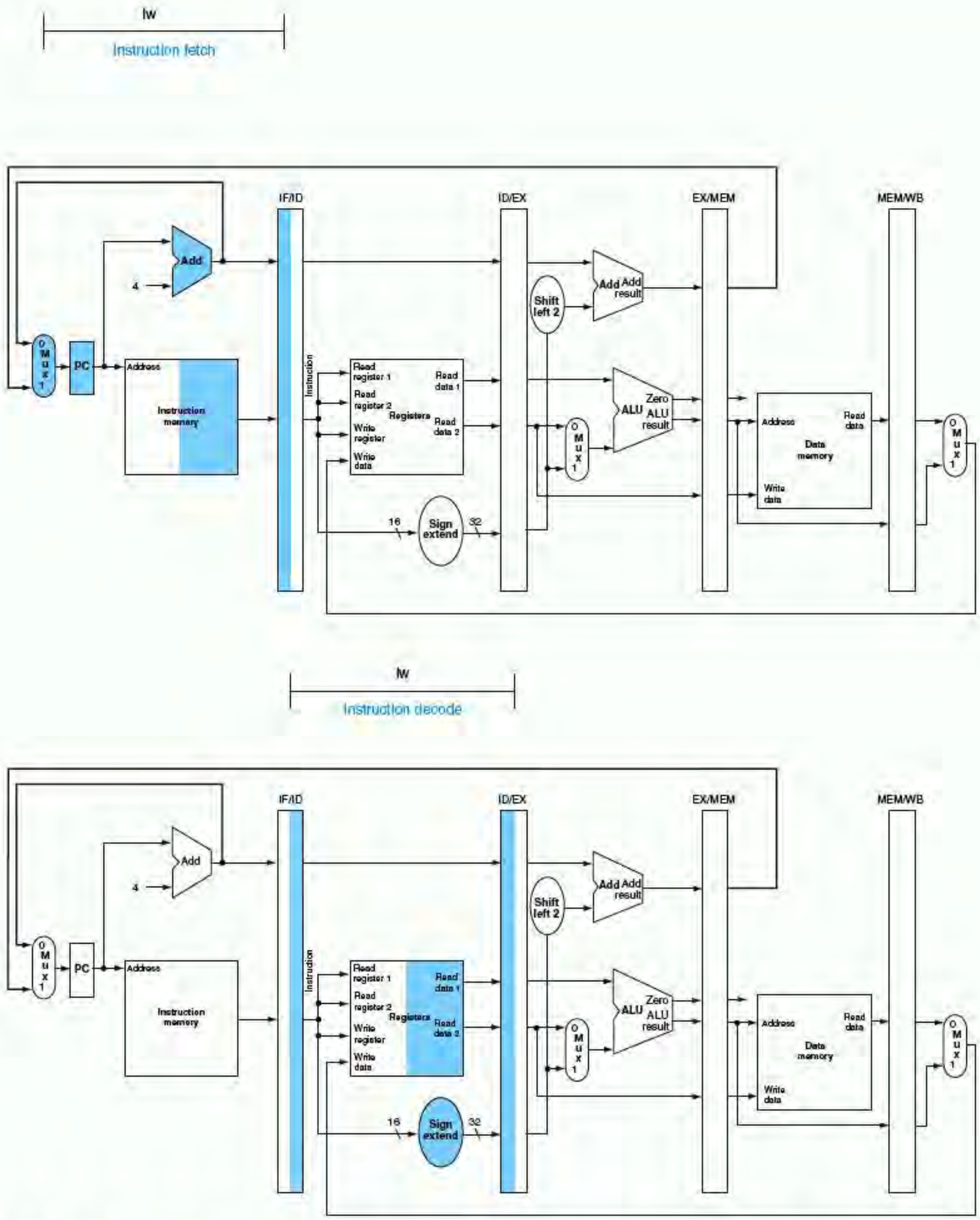


FIGURE 6.12 IF and ID: first and second pipe stages of an instruction, with the active portions of the datapath in Figure 6.11 highlighted. The highlighting convention is the same as that used in Figure 6.4. As in Chapter 5, there is no confusion when reading and writing registers because the contents change only on the clock edge. Although the load needs only the top register in stage 2, the processor doesn't know what instruction is being decoded, so it sign-extends the 16-bit constant and reads both registers into the ID/EX pipeline register. We don't need all three operands, but it simplifies control to keep all three.

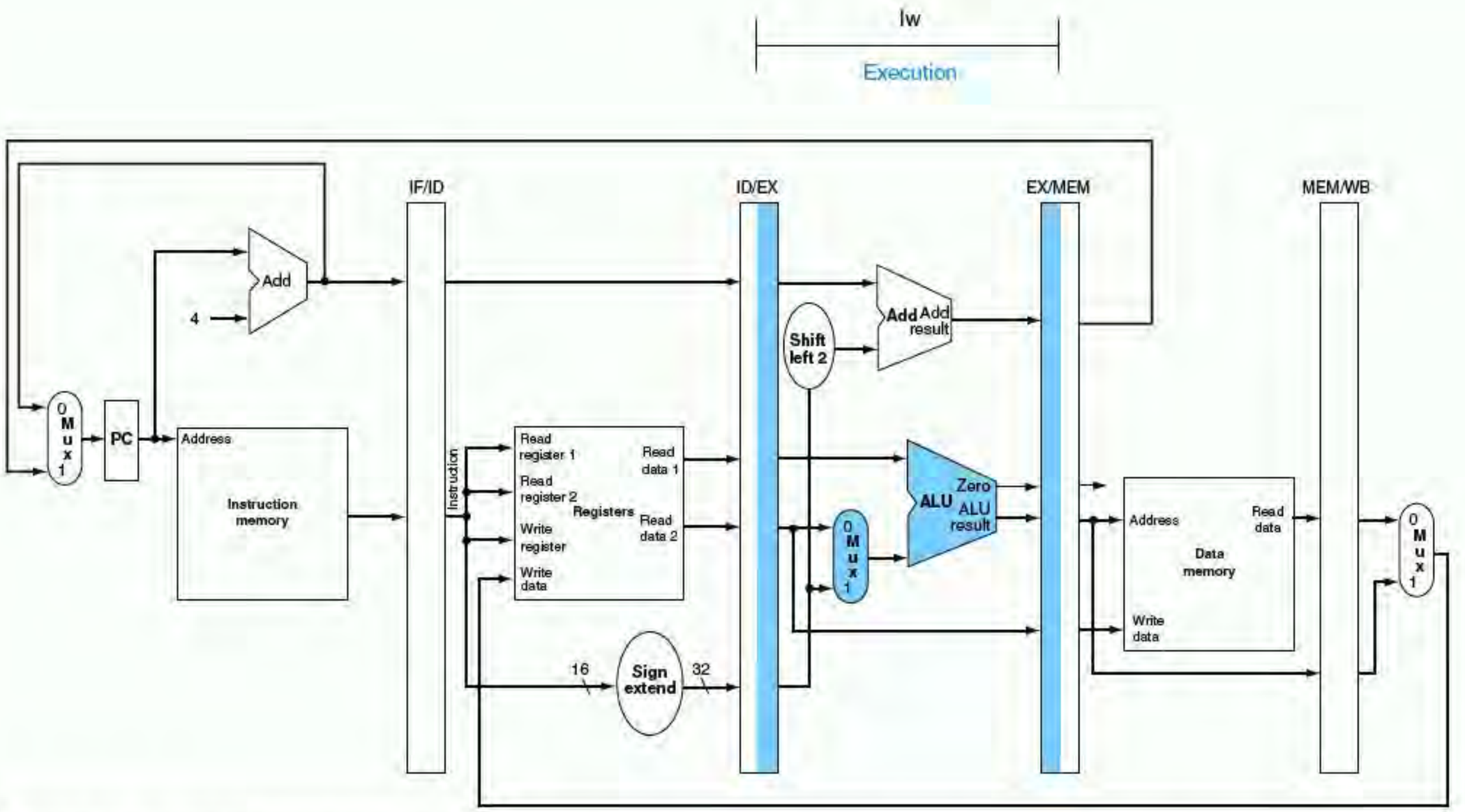
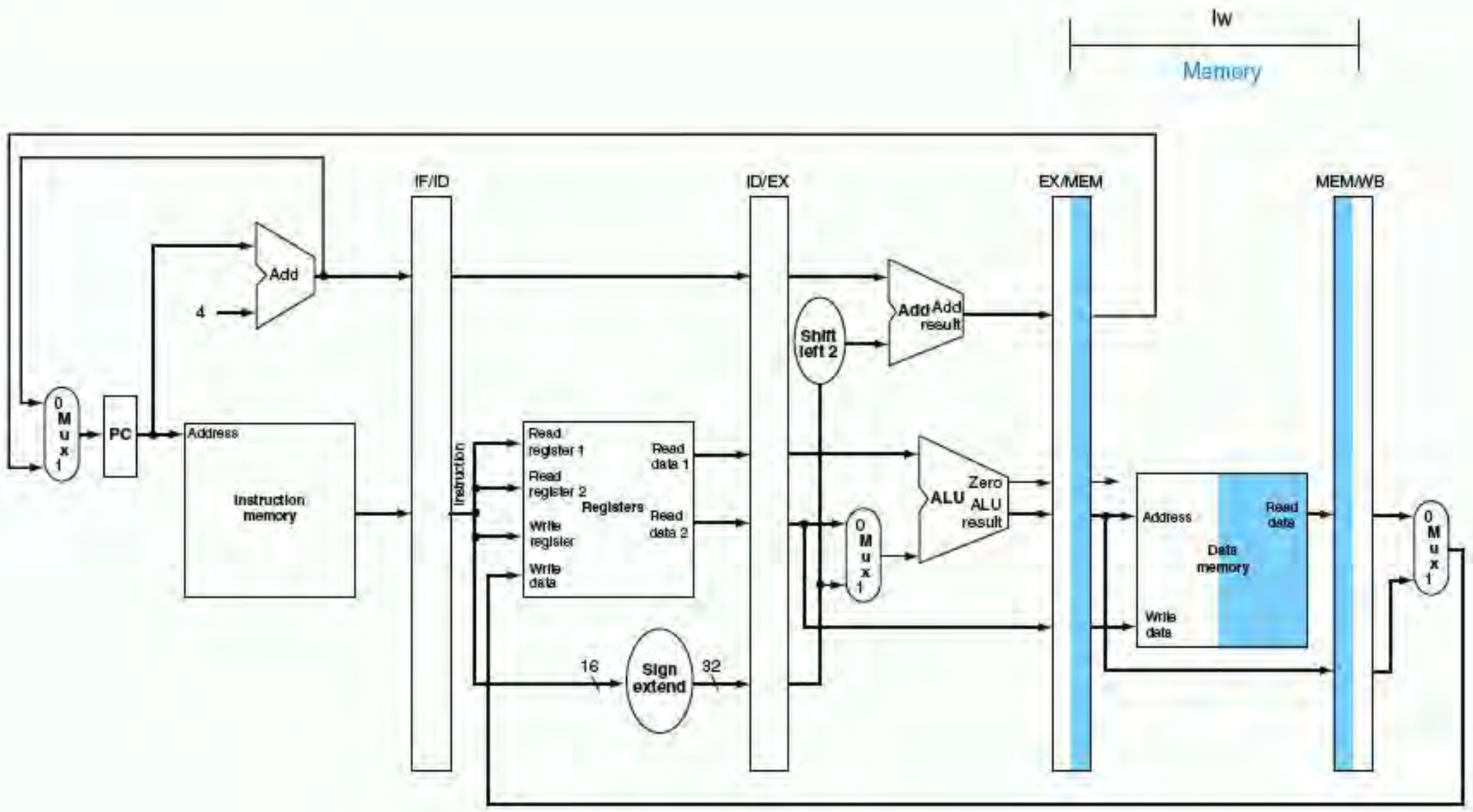


FIGURE 6.13 EX: the third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 6.11 used in this pipe stage. The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.

2. *Instruction decode and register file read:* The bottom portion of Figure 6.12 shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.
3. *Execute or address calculation:* Figure 6.13 shows that the load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register.
4. *Memory access:* The top portion of Figure 6.14 shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.



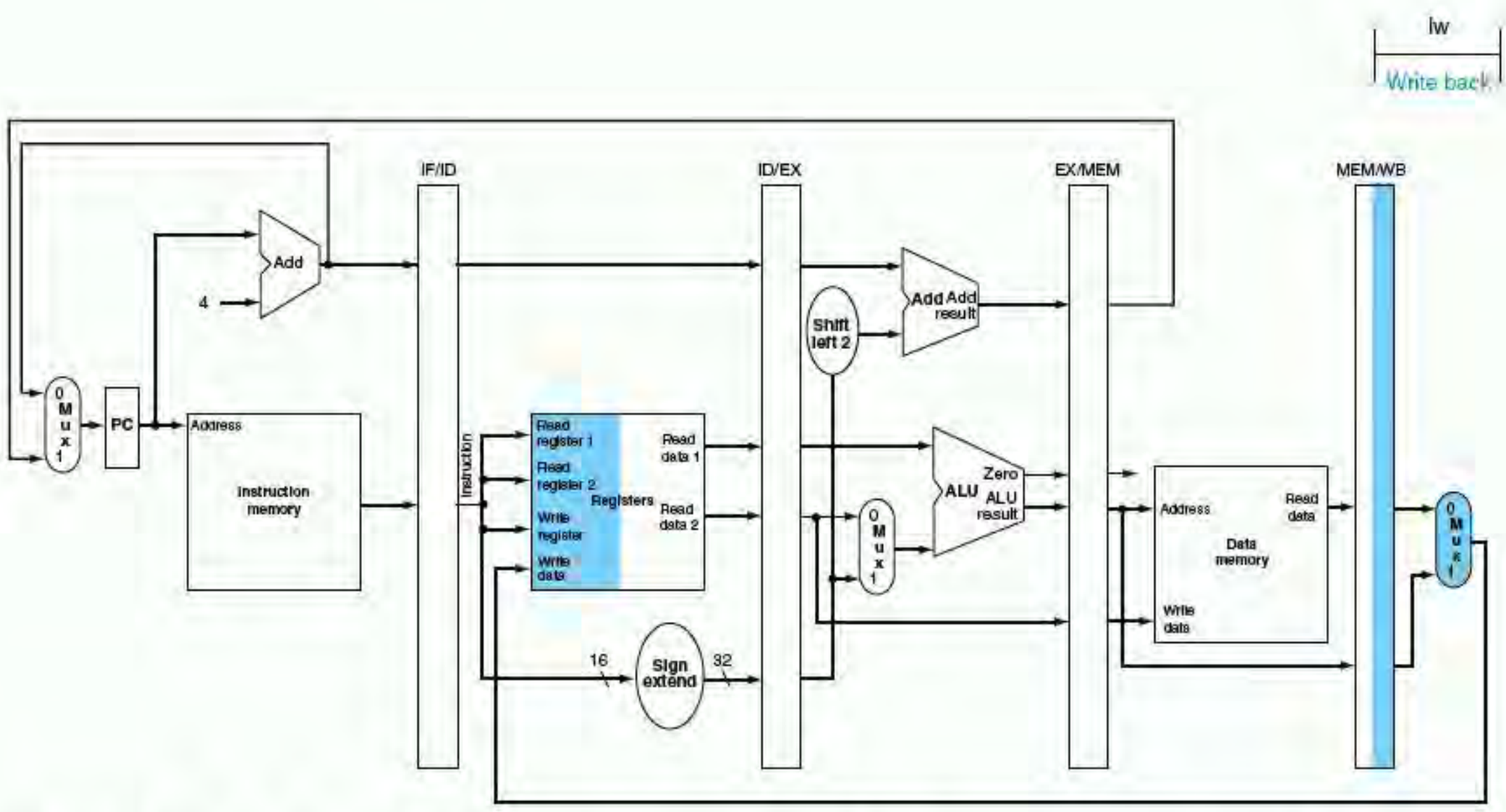


FIGURE 6.14 MEM and WB: the fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 6.11 used in this pipe stage. Data memory is read using the address in the EX/MEM pipeline registers, and the data is placed in the MEM/WB pipeline register. Next, data is read from the MEM/WB pipeline register and written into the register file in the middle of the datapath.

5. *Write back:* The bottom portion of Figure 6.14 shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

This walk-through of the load instruction shows that any information needed in a later pipe stage must be passed to that stage via a pipeline register. Walking through a store instruction shows the similarity of instruction execution, as well as passing the information for later stages. Here are the five pipe stages of the store instruction:

1. *Instruction fetch:* The instruction is read from memory using the address in the PC and then is placed in the IF/ID pipeline register. This stage occurs before the instruction is identified, so the top portion of Figure 6.12 works for store as well as load.
2. *Instruction decode and register file read:* The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the 16-bit immediate. These three 32-bit values are all stored in the ID/EX pipeline register. The bottom portion of Figure 6.12 for load instructions also shows the operations of the second stage for stores. These first two stages are executed by all instructions, since it is too early to know the type of the instruction.
3. *Execute and address calculation:* Figure 6.15 shows the third step; the effective address is placed in the EX/MEM pipeline register.
4. *Memory access:* The top portion of Figure 6.16 shows the data being written to memory. Note that the register containing the data to be stored was read in an earlier stage and stored in ID/EX. The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.
5. *Write back:* The bottom portion of Figure 6.16 shows the final step of the store. For this instruction, nothing happens in the write-back stage. Since every instruction behind the store is already in progress, we have no way to accelerate those instructions. Hence, an instruction passes through a stage even if there is nothing to do because later instructions are already progressing at the maximum rate.

The store instruction again illustrates that to pass something from an early pipe stage to a later pipe stage, the information must be placed in a pipeline register; otherwise, the information is lost when the next instruction enters that pipeline stage. For the store instruction we needed to pass one of the registers read in the ID stage to the MEM stage, where it is stored in memory. The data was first placed in the ID/EX pipeline register and then passed to the EX/MEM pipeline register.

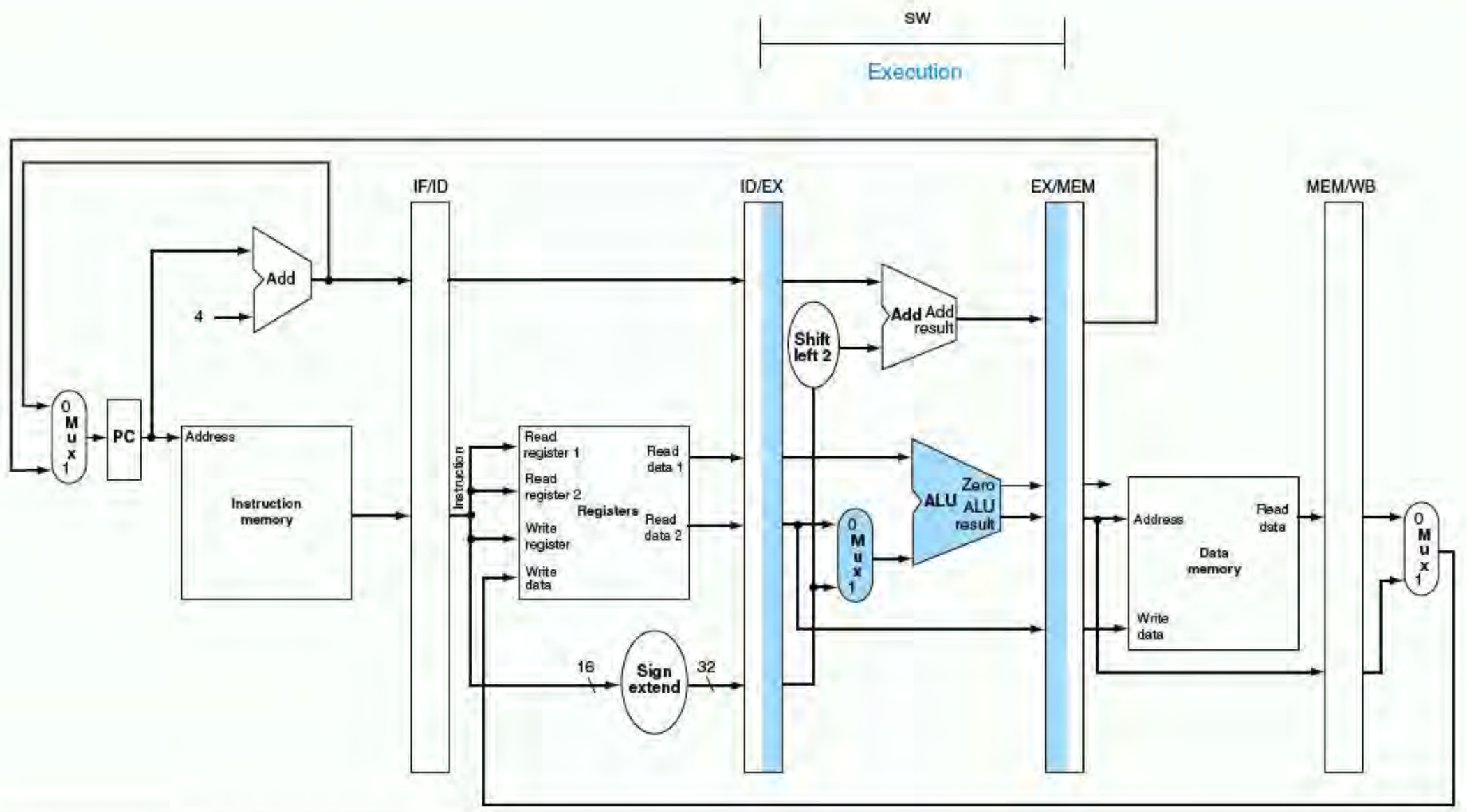
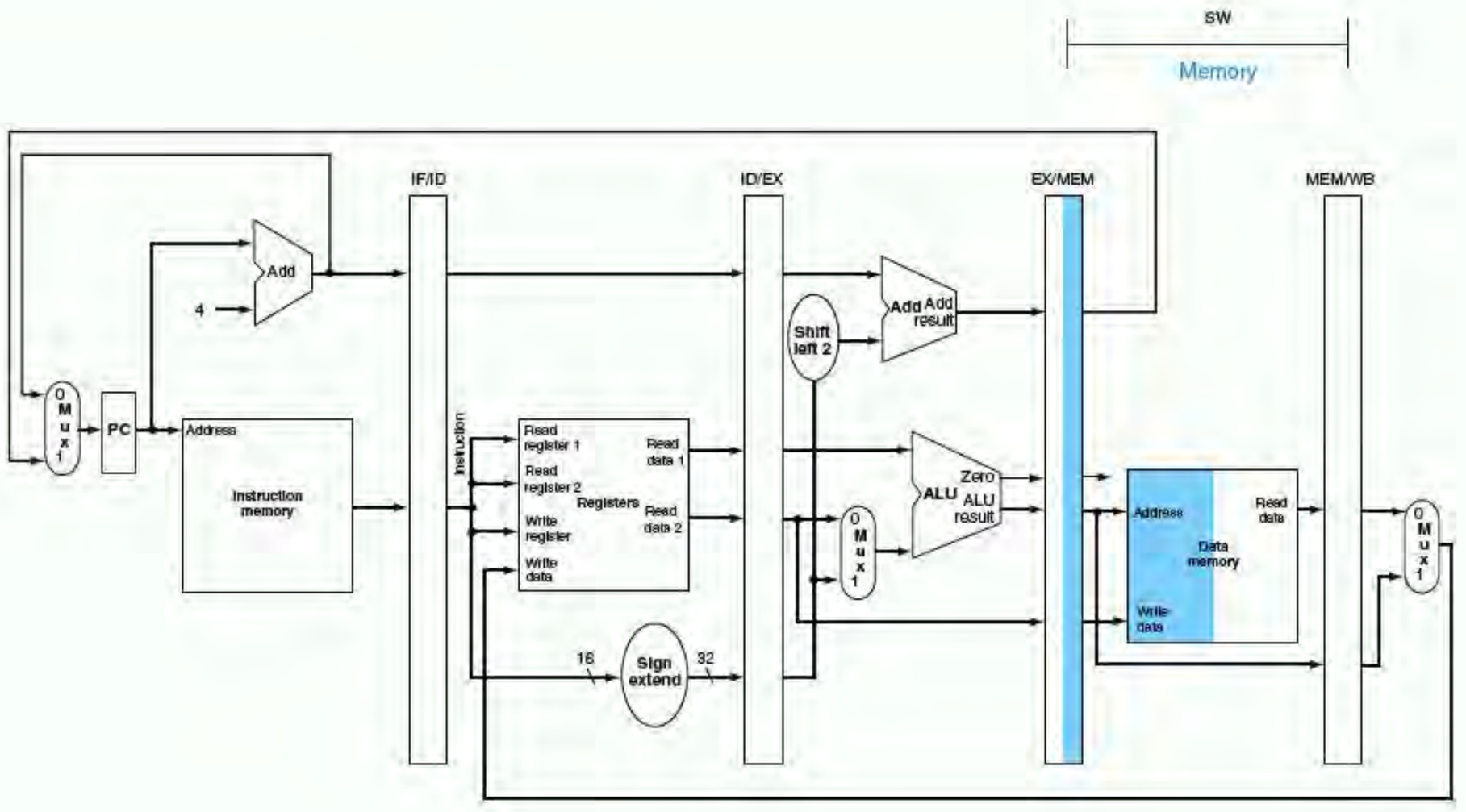


FIGURE 6.15 EX: the third pipe stage of a store instruction. Unlike the third stage of the load instruction in Figure 6.13, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.

Load and store illustrate a second key point: each logical component of the datapath—such as instruction memory, register read ports, ALU, data memory, and register write port—can be used only within a *single* pipeline stage. Otherwise we would have a *structural hazard* (see page 375). Hence these components, and their control, can be associated with a single pipeline stage.

Now we can uncover a bug in the design of the load instruction. Did you see it? Which register is changed in the final stage of the load? More specifically, which instruction supplies the write register number? The instruction in the IF/ID pipeline register supplies the write register number, yet this instruction occurs considerably *after* the load instruction!

Hence, we need to preserve the destination register number in the load instruction. Just as store passed the register *contents* from the ID/EX to the EX/MEM pipeline registers for use in the MEM stage, load must pass the register *number* from the ID/EX through EX/MEM to the MEM/WB pipeline register for use in



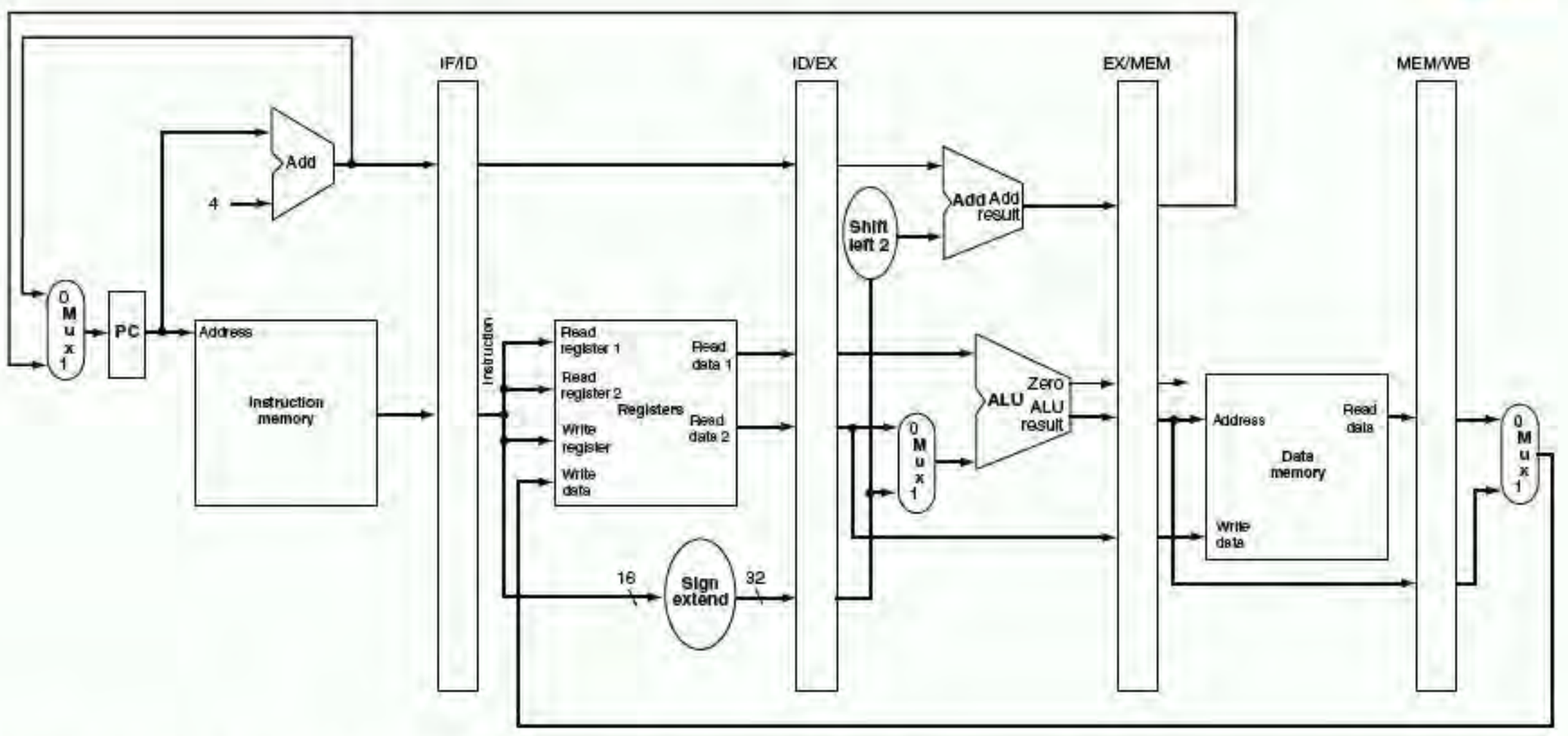


FIGURE 6.16 MEM and WB: the fourth and fifth pipe stage of a store instruction. In the fourth stage, the data is written into data memory for the store. Note that the data comes from the EX/MEM pipeline register and that nothing is changed in the MEM/WB pipeline register. Once the data is written in memory, there is nothing left for the store instruction to do, so nothing happens in stage 5.

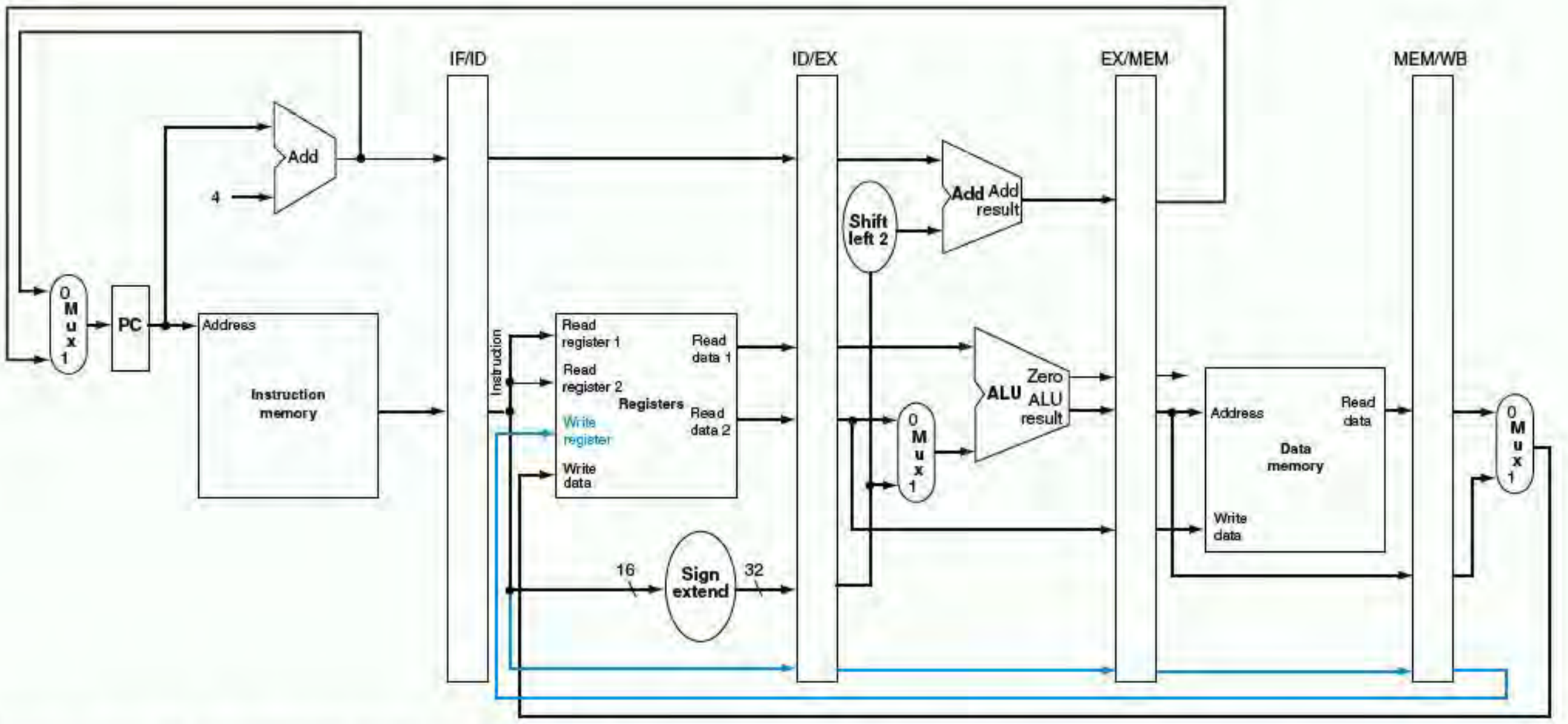


FIGURE 6.17 The corrected pipelined datapath to properly handle the load instruction. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding 5 more bits to the last three pipeline registers. This new path is shown in color.

the WB stage. Another way to think about the passing of the register number is that, in order to share the pipelined datapath, we needed to preserve the instruction read during the IF stage, so each pipeline register contains a portion of the instruction needed for that stage and later stages.

Figure 6.17 shows the correct version of the datapath, passing the write register number first to the ID/EX register, then to the EX/MEM register, and finally to the MEM/WB register. The register number is used during the WB stage to specify the register to be written. Figure 6.18 is a single drawing of the corrected datapath, highlighting the hardware used in all five stages of the load word instruction in Figures 6.12 through 6.14. See Section 6.6 for an explanation of how to make the branch instruction work as expected.

Graphically Representing Pipelines

Pipelining can be difficult to understand, since many instructions are simultaneously executing in a single datapath in every clock cycle. To aid understanding, there are two basic styles of pipeline figures: *multiple-clock-cycle pipeline diagrams*, such as Figure 6.10 on page 387, and *single-clock-cycle pipeline diagrams*,

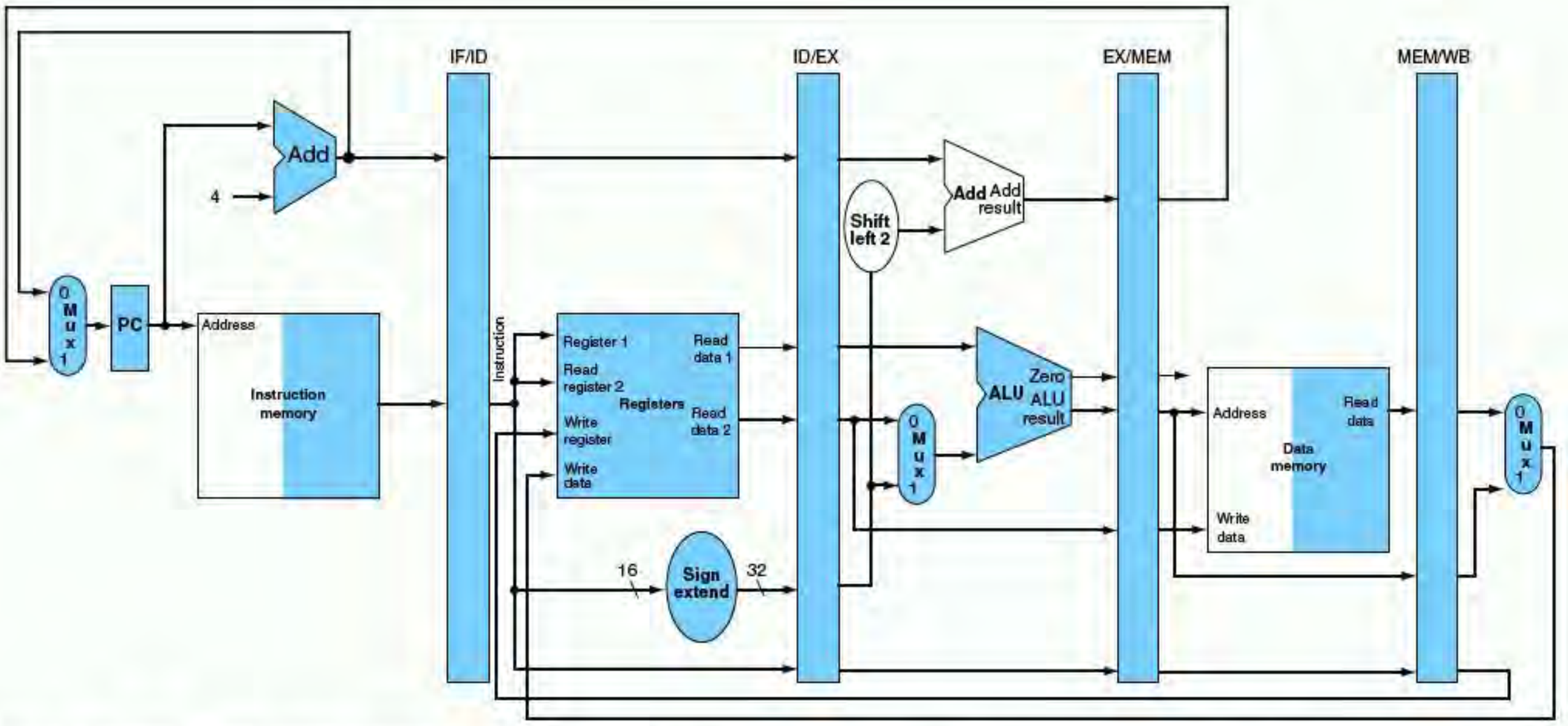


FIGURE 6.18 The portion of the datapath in Figure 6.17 that is used in all five stages of a load instruction.

such as Figures 6.12 through 6.16. The multiple-clock-cycle diagrams are simpler but do not contain all the details. For example, consider the following five-instruction sequence:

```
lw     $10, 20($1)
sub    $11, $2, $3
add    $12, $3, $4
lw     $13, 24($1)
add    $14, $5, $6
```

Figure 6.19 shows the multiple-clock-cycle pipeline diagram for these instructions. Time advances from left to right across the page in these diagrams, and instructions advance from the top to the bottom of the page, similar to the laundry pipeline in Figure 6.1 on page 371. A representation of the pipeline stages is placed in each portion along the instruction axis, occupying the proper clock cycles. These stylized datapaths represent the five stages of our pipeline, but a rectangle naming each pipe stage works just as well. Figure 6.20 shows the more traditional version of the multiple-clock-cycle pipeline diagram. Note that Figure 6.19 shows the physical resources used at each stage, while Figure 6.20 uses the *name* of each stage. We use multiple-clock-cycle diagrams to give overviews of pipelining situations.

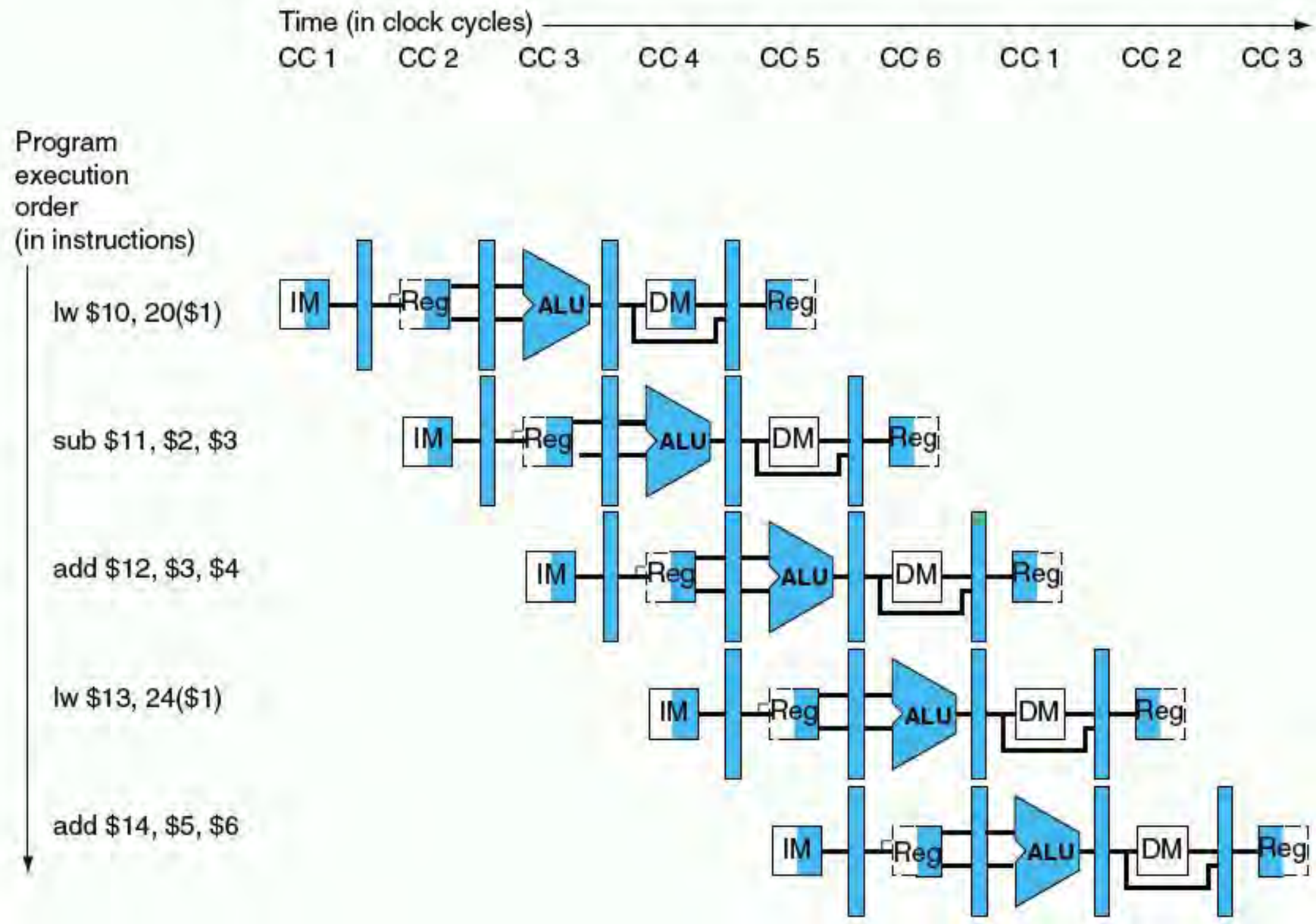


FIGURE 6.19 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 6.4, here we show the pipeline registers between each stage. Figure 6.20 shows the traditional way to draw this diagram.

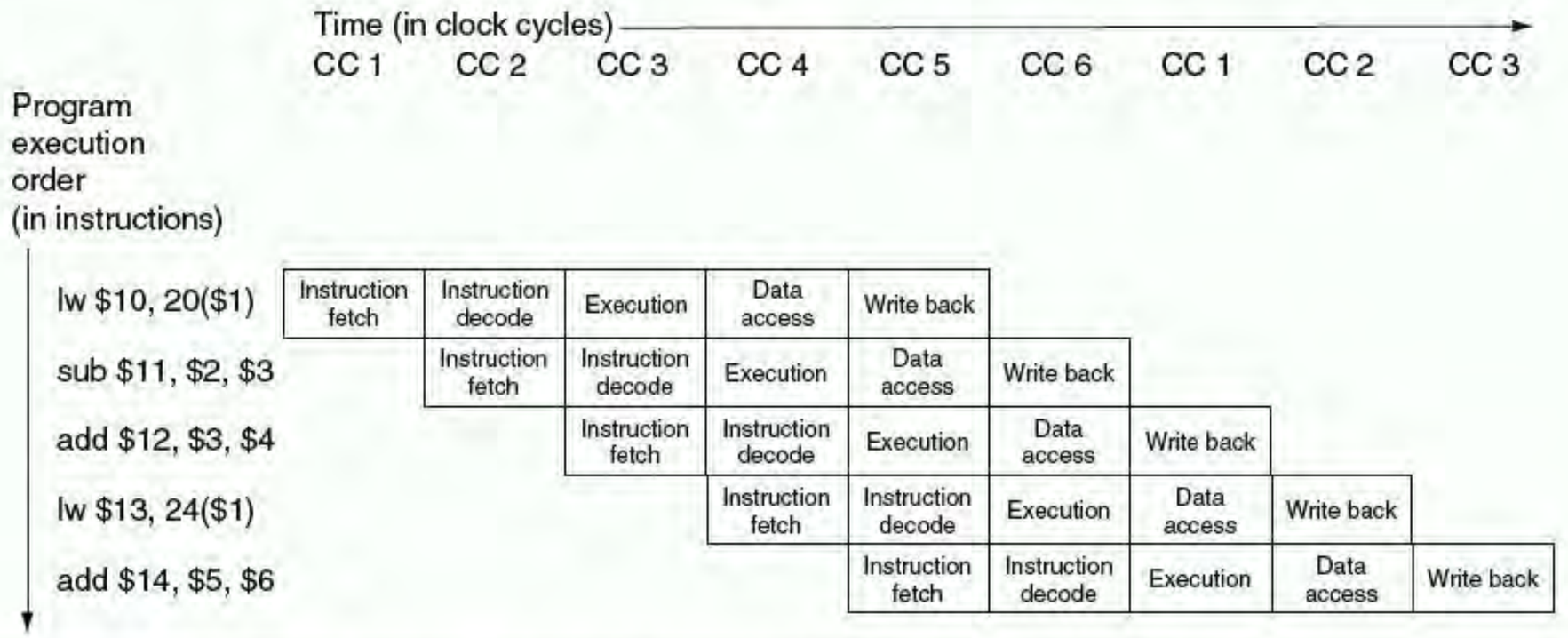


FIGURE 6.20 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 6.19.

Single-clock-cycle pipeline diagrams show the state of the entire datapath during a single clock cycle, and usually all five instructions in the pipeline are identified by labels above their respective pipeline stages. We use this type of figure to show the details of what is happening within the pipeline during each clock cycle; typically, the drawings appear in groups to show pipeline operation over a sequence of clock cycles. A single-clock-cycle diagram represents a vertical slice through a set of multiple-clock-cycle diagram, showing the usage of the datapath by each of the instructions in the pipeline at the designated clock cycle. For example, Figure 6.21 shows the single-clock-cycle diagram corresponding to clock cycle 5 of Figures 6.19 and 6.20. Obviously, the single-clock-cycle diagrams have more detail and take significantly more space to show the same number of clock cycles. The For More Practice section included on the CD includes the corresponding single-clock-cycle diagrams for these two instructions as well as exercises asking you to create such diagrams for another code sequence.

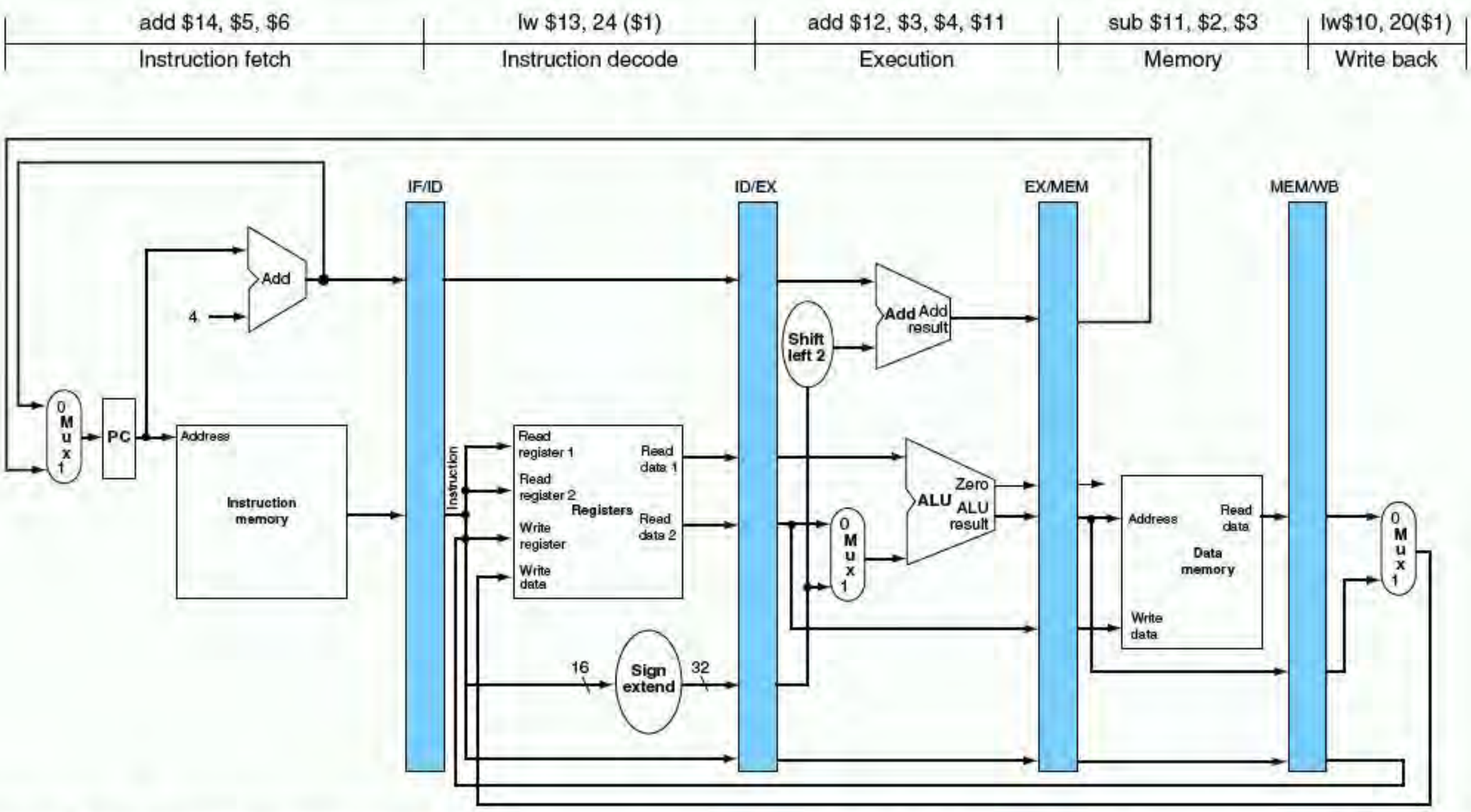


FIGURE 6.21 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 6.19 and 6.20. As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

Check Yourself

A group of students have been debating the efficiency of the five-stage pipeline when one student pointed out that not all instructions are active in every stage of the pipeline. After deciding to ignore the effects of hazards, they made the following five statements. Which ones are correct?

1. Allowing jumps, branches, and ALU instructions to take fewer stages than the five required by the load instruction will increase pipeline performance under all circumstances.
2. Trying to allow some instructions to take fewer cycles does not help, since the throughput is determined by the clock cycle; the number of pipe stages per instruction affects latency, not throughput.
3. Allowing jumps, branches, and ALU operations to take fewer cycles only helps when no loads or stores are in the pipeline, so the benefits are small.
4. You cannot make ALU instructions take fewer cycles because of the write-back of the result, but branches and jumps can take fewer cycles, so there is some opportunity for improvement.
5. Instead of trying to make instructions take fewer cycles, we should explore making the pipeline longer, so that instructions take more cycles, but the cycles are shorter. This could improve performance.

6.3 Pipelined Control

In the 6600 Computer, perhaps even more than in any previous computer, the control system is the difference.

James Thornton,
Design of a Computer: The Control Data 6600, 1970

Just as we added control to the simple datapath in Section 5.4, we now add control to the pipelined datapath. We start with a simple design that views the problem through rose-colored glasses; in Sections 6.4 through 6.8, we remove these glasses to reveal the hazards of the real world.

The first step is to label the control lines on the existing datapath. Figure 6.22 shows those lines. We borrow as much as we can from the control for the simple datapath in Figure 5.17 on page 307. In particular, we use the same ALU control logic, branch logic, destination-register-number multiplexor, and control lines. These functions are defined in Figure 5.12 on page 302, Figure 5.16 on page 306, and Figure 5.18 on page 308. We reproduce the key information in Figures 6.23 through 6.25 to make the remaining text easier to follow.

As for the single-cycle implementation discussed in Chapter 5, we assume that the PC is written on each clock cycle, so there is no separate write signal for the

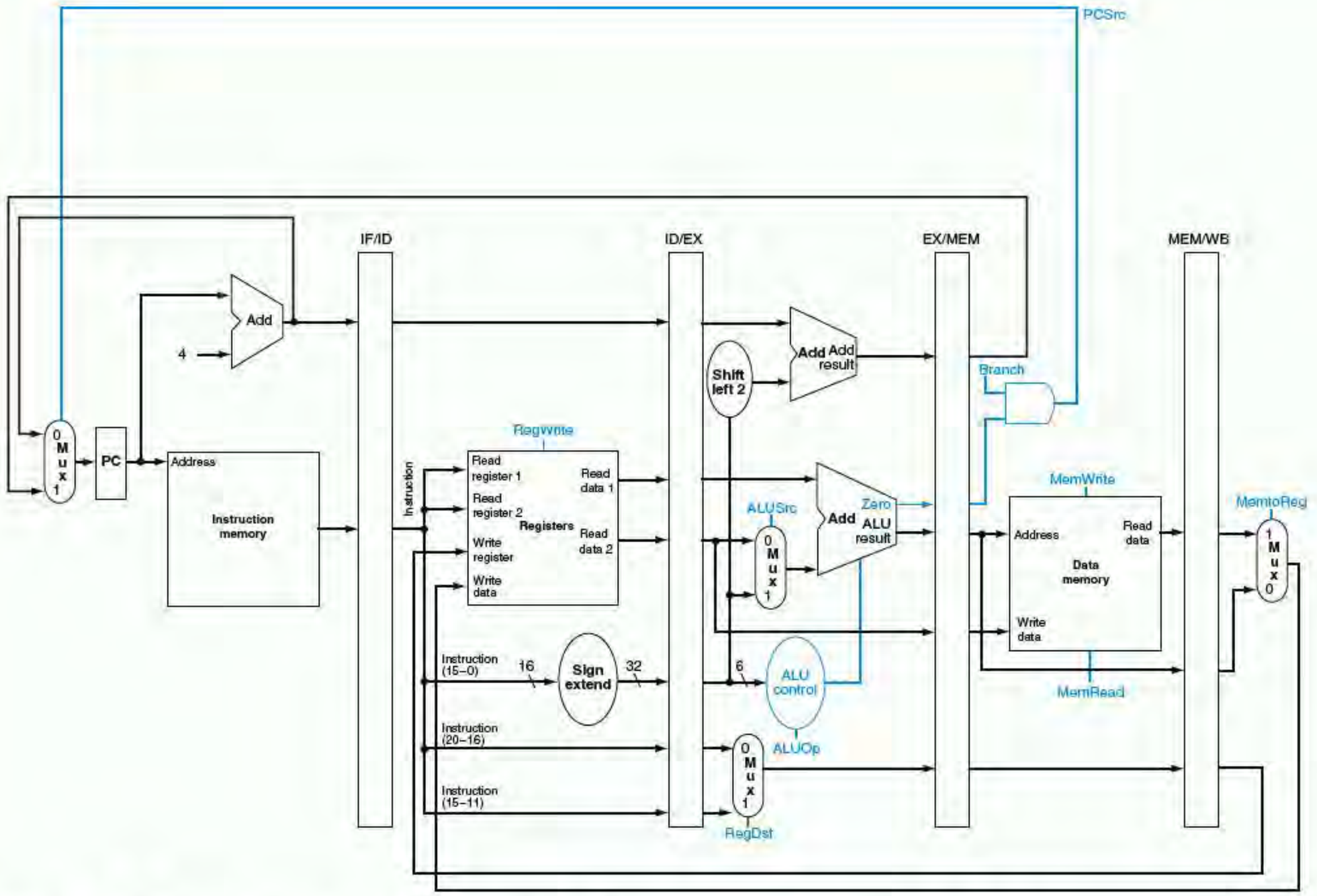


FIGURE 6.22 The pipelined datapath of Figure 6.17 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Chapter 5. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register. Recall that these 6 bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.

PC. By the same argument, there are no separate write signals for the pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB), since the pipeline registers are also written during each clock cycle.

To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.

1. *Instruction fetch:* The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 6.23 A copy of Figure 5.12 on page 302. This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different function codes for the R-type instruction.

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 6.24 A copy of Figure 5.16 on page 306. The function of each of seven control signals is defined. The ALU control lines (ALUOp) are defined in the second column of Figure 6.23. When a 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Note that PCSrc is controlled by an AND gate in Figure 6.22. If the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a `beq` instruction; otherwise, PCSrc is set to 0.

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

FIGURE 6.25 The values of the control lines are the same as in Figure 5.18 on page 308, but they have been shuffled into three groups corresponding to the last three pipeline stages.

2. *Instruction decode/register file read:* As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.
3. *Execution/address calculation:* The signals to be set are RegDst, ALUOp, and ALUSrc (see Figures 6.23 and 6.24). The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.
4. *Memory access:* The control lines set in this stage are Branch, MemRead, and MemWrite. These signals are set by the branch equal, load, and store instructions, respectively. Recall that PCSrc in Figure 6.24 selects the next sequential address unless control asserts Branch and the ALU result was zero.
5. *Write back:* The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and RegWrite, which writes the chosen value.

Since pipelining the datapath leaves the meaning of the control lines unchanged, we can use the same control values as before. Figure 6.25 has the same values as in Chapter 5, but now the nine control lines are grouped by pipeline stage.

Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information.

Since the control lines start with the EX stage, we can create the control information during instruction decode. Figure 6.26 shows that these control signals are then used in the appropriate pipeline stage as the instruction moves down the pipeline, just as the destination register number for loads moves down the pipeline in Figure 6.17 on page 395. Figure 6.27 shows the full datapath with the extended pipeline registers and with the control lines connected to the proper stage.

What do you mean, why's it got to be built? It's a bypass. You've got to build bypasses.

Douglas Adams, *Hitchhikers Guide to the Galaxy*, 1979

6.4 Data Hazards and Forwarding

The examples in the previous section show the power of pipelined execution and how the hardware performs the task. It's now time to take off the rose-colored glasses and look at what happens with real programs. The instructions in Figures 6.19 through 6.21 were independent; none of them used the results calculated by any of the others. Yet in Section 6.1 we saw that data hazards are obstacles to pipelined execution.

Let's look at a sequence with many dependences, shown in color:

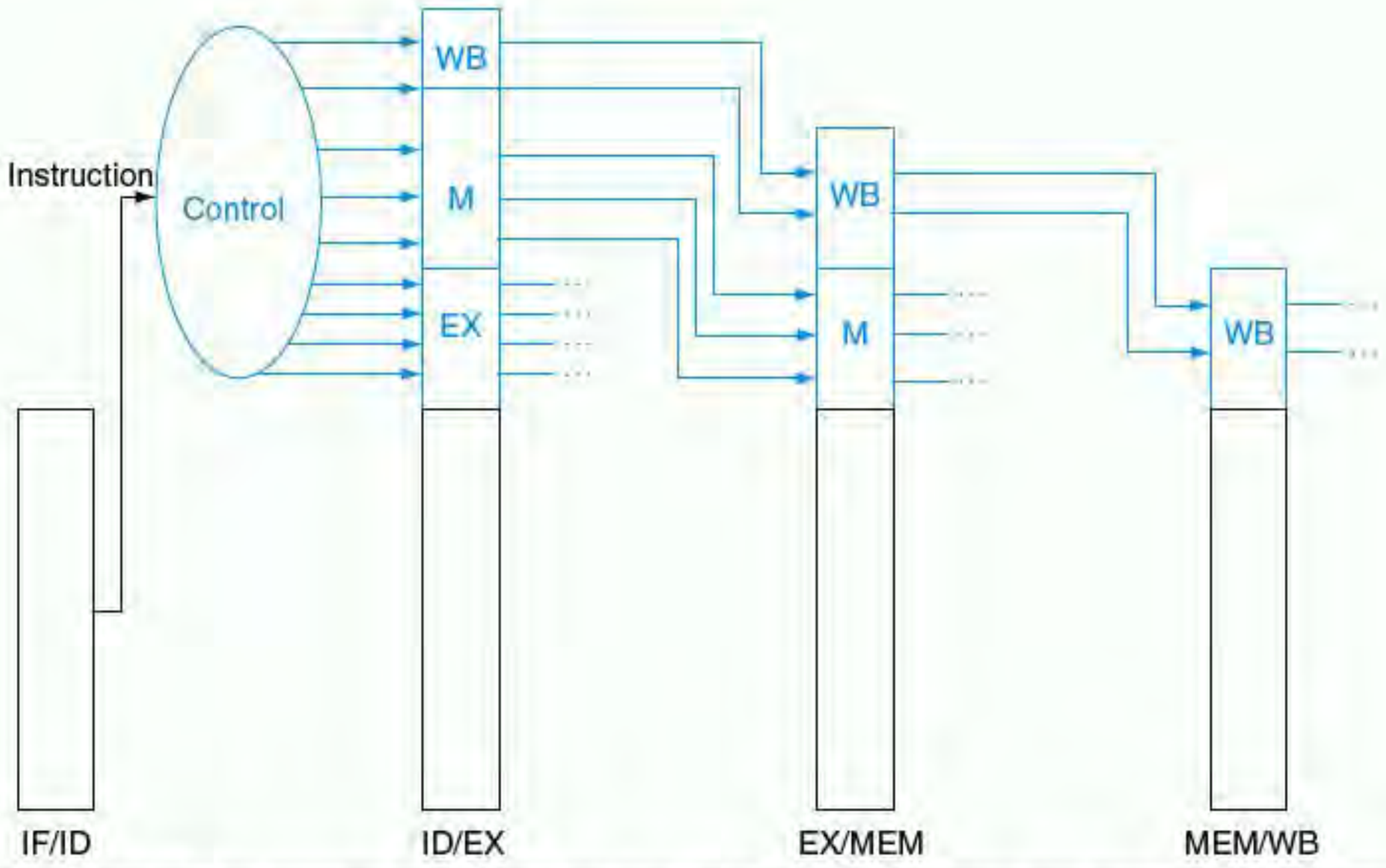


FIGURE 6.26 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

```
sub   $2, $1,$3     # Register $2 written by sub
and   $12,$2,$5     # 1st operand($2) depends on sub
or    $13,$6,$2     # 2nd operand($2) depends on sub
add   $14,$2,$2     # 1st($2) & 2nd($2) depend on sub
sw    $15,100($2)   # Base ($2) depends on sub
```

The last four instructions are all dependent on the result in register $2 of the first instruction. If register $2 had the value 10 before the subtract instruction and –20 afterwards, the programmer intends that –20 will be used in the following instructions that refer to register $2.

How would this sequence perform with our pipeline? Figure 6.28 illustrates the execution of these instructions using a multiple-clock-cycle pipeline representation. To demonstrate the execution of this instruction sequence in our current pipeline, the top of Figure 6.28 shows the value of register $2, which changes during the middle of clock cycle 5, when the sub instruction writes its result.

One potential hazard can be resolved by the design of the register file hardware: what happens when a register is read and written in the same clock cycle? We assume that the write is in the first half of the clock cycle and the read is in the second half, so the read delivers what is written. As is the case for many implementations of register files, we have no data hazard in this case.

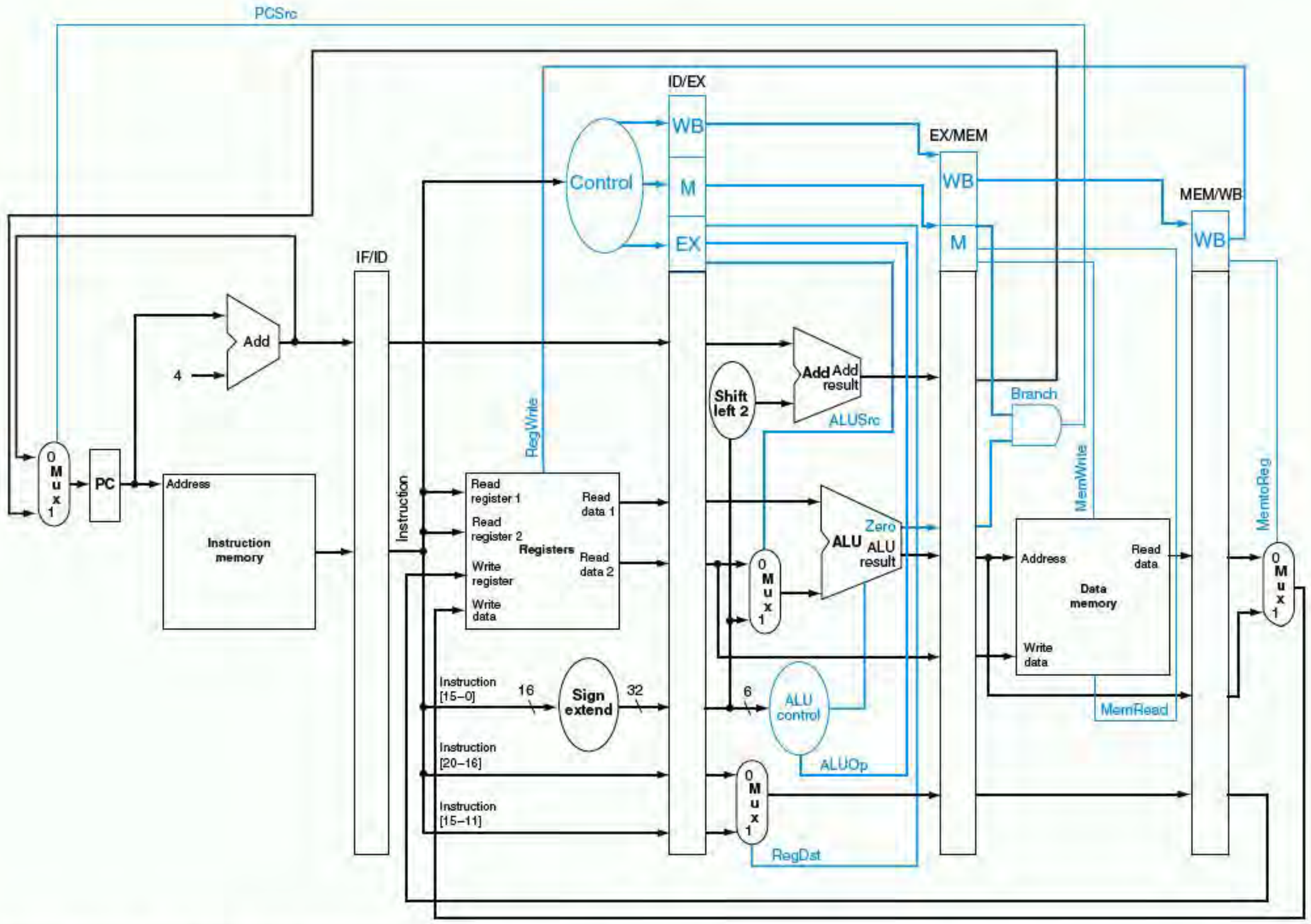


FIGURE 6.27 The pipelined datapath of Figure 6.22, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Figure 6.28 shows that the values read for register `$2` would *not* be the result of the `sub` instruction unless the read occurred during clock cycle 5 or later. Thus, the instructions that would get the correct value of –20 are `add` and `sw`; the `and` and `or` instructions would get the incorrect value 10! Using this style of drawing, such problems become apparent when a dependence line goes backwards in time.

But, look carefully at Figure 6.28: When is the data from the `sub` instruction actually produced? The result is available at the end of the EX stage or clock cycle 3. When is the data actually needed by the `and` and `or` instructions? At the beginning of the EX stage, or clock cycles 4 and 5, respectively. Thus, we can execute

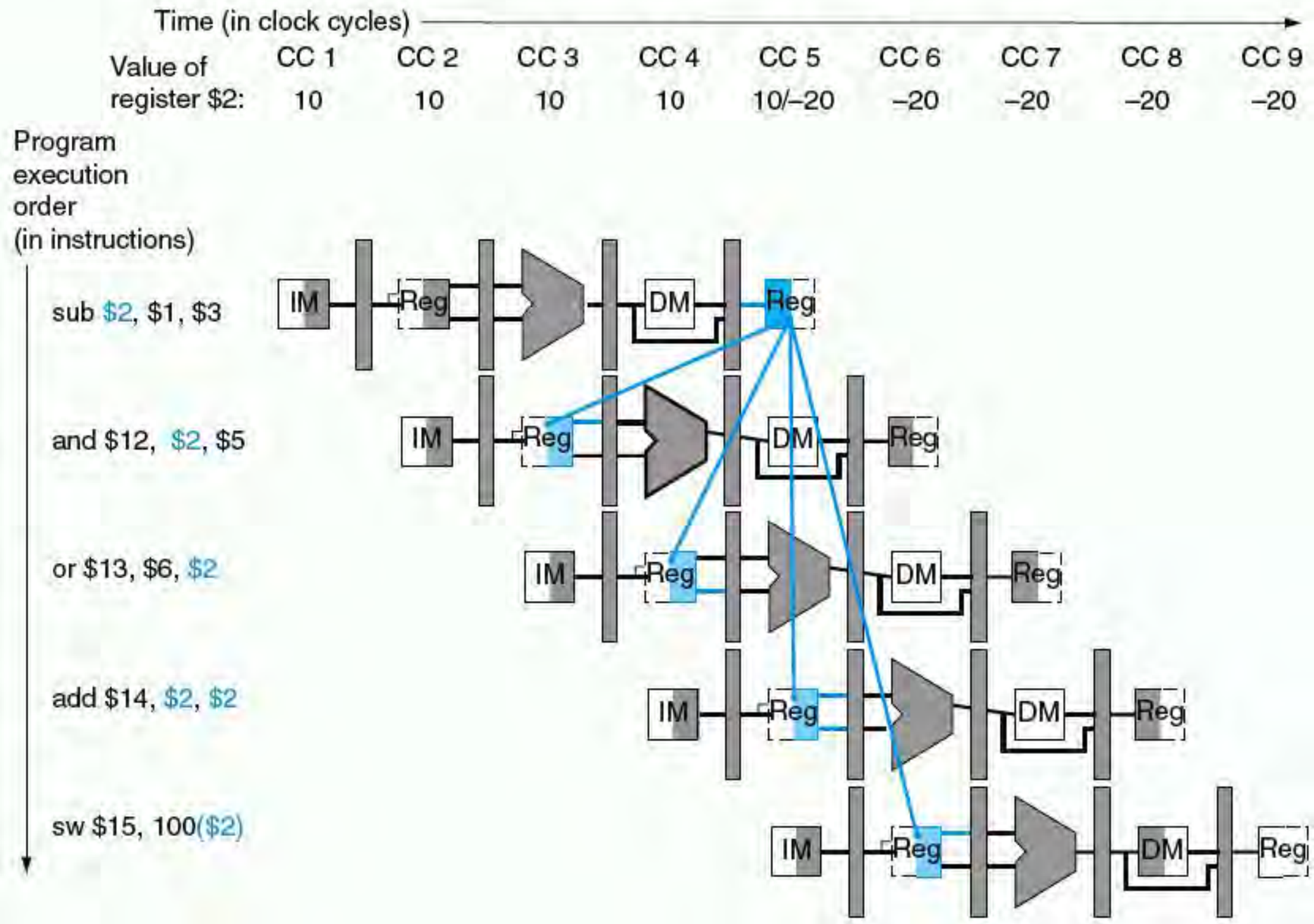


FIGURE 6.28 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC *i*" at the top of the figure means clock cycle *i*. The first instruction writes into $2, and all the following instructions read $2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backwards in time are *pipeline data hazards*.

this segment without stalls if we simply *forward* the data as soon as it is available to any units that need it before it is available to read from the register file.

How does forwarding work? For simplicity in the rest of this section, we consider only the challenge of forwarding to an operation in the EX stage, which may be either an ALU operation or an effective address calculation. This means that when an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage, we actually need the values as inputs to the ALU.

A notation that names the fields of the pipeline registers allows for a more precise notation of dependences. For example, "ID/EX.RegisterRs" refers to the number of one register whose value is found in the pipeline register ID/EX; that is, the one from the first read port of the register file. The first part of the name, to the left of the period, is the name of the pipeline register; the second part is the name

of the field in that register. Using this notation, the two pairs of hazard conditions are

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

The first hazard in the sequence on page 403 is on register `$2`, between the result of `sub $2,$1,$3` and the first read operand of `and $12,$2,$5`. This hazard can be detected when the `and` instruction is in the EX stage and the prior instruction is in the MEM stage, so this is hazard 1a:

EX/MEM.RegisterRd = ID/EX.RegisterRs = `$2`

EXAMPLE

Dependence Detection

Classify the dependences in this sequence from page 403:

```
sub     $2,   $1, $3  # Register $2 set by sub
and     $12,  $2, $5  # 1st operand($2) set by sub
or      $13,  $6, $2  # 2nd operand($2) set by sub
add     $14,  $2, $2  # 1st($2) & 2nd($2) set by sub
sw      $15,  100($2) # Index($2) set by sub
```

ANSWER

As mentioned above, the `sub-and` is a type 1a hazard. The remaining hazards are

- The `sub-or` is a type 2b hazard:

  MEM/WB.RegisterRd = ID/EX.RegisterRt = `$2`

- The two dependences on `sub-add` are not hazards because the register file supplies the proper data during the ID stage of `add`.
- There is no data hazard between `sub` and `sw` because `sw` reads `$2` the clock cycle *after* `sub` writes `$2`.

Because some instructions do not write registers, this policy is inaccurate; sometimes it would forward when it was unnecessary. One solution is simply to check to see if the RegWrite signal will be active: examining the WB control field

of the pipeline register during the EX and MEM stages determines if RegWrite is asserted. Also, MIPS requires that every use of `$0` as an operand must yield an operand value of zero. In the event that an instruction in the pipeline has `$0` as its destination (for example, `sll $0, $1, 2`), we want to avoid forwarding its possibly nonzero result value. Not forwarding results destined for `$0` frees the assembly programmer and the compiler of any requirement to avoid using `$0` as a destination. The conditions above thus work properly as long we add EX/MEM.RegisterRd ≠ 0 to the first hazard condition and MEM/WB.RegisterRd ≠ 0 to the second.

Now that we can detect hazards, half of the problem is resolved—but we must still forward the proper data.

Figure 6.29 shows the dependences between the pipeline registers and the inputs to the ALU for the same code sequence as in Figure 6.28. The change is that the dependence begins from a *pipeline* register rather than waiting for the WB stage to write the register file. Thus the required data exists in time for later instructions, with the pipeline registers holding the data to be forwarded.

If we can take the inputs to the ALU from *any* pipeline register rather than just ID/EX, then we can forward the proper data. By adding multiplexors to the input of the ALU and with the proper controls, we can run the pipeline at full speed in the presence of these data dependences.

For now, we will assume the only instructions we need to forward are the four R-format instructions: `add`, `sub`, `and`, and `or`. Figure 6.30 shows a close-up of the ALU and pipeline register before and after adding forwarding. Figure 6.31 shows the values of the control lines for the ALU multiplexors that select either the register file values or one of the forwarded values.

This forwarding control will be in the EX stage because the ALU forwarding multiplexors are found in that stage. Thus, we must pass the operand register numbers from the ID stage via the ID/EX pipeline register to determine whether to forward values. We already have the rt field (bits 20–16). Before forwarding, the ID/EX register had no need to include space to hold the rs field. Hence, rs (bits 25–21) is added to ID/EX.

Let's now write both the conditions for detecting hazards and the control signals to resolve them:

1. EX hazard:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

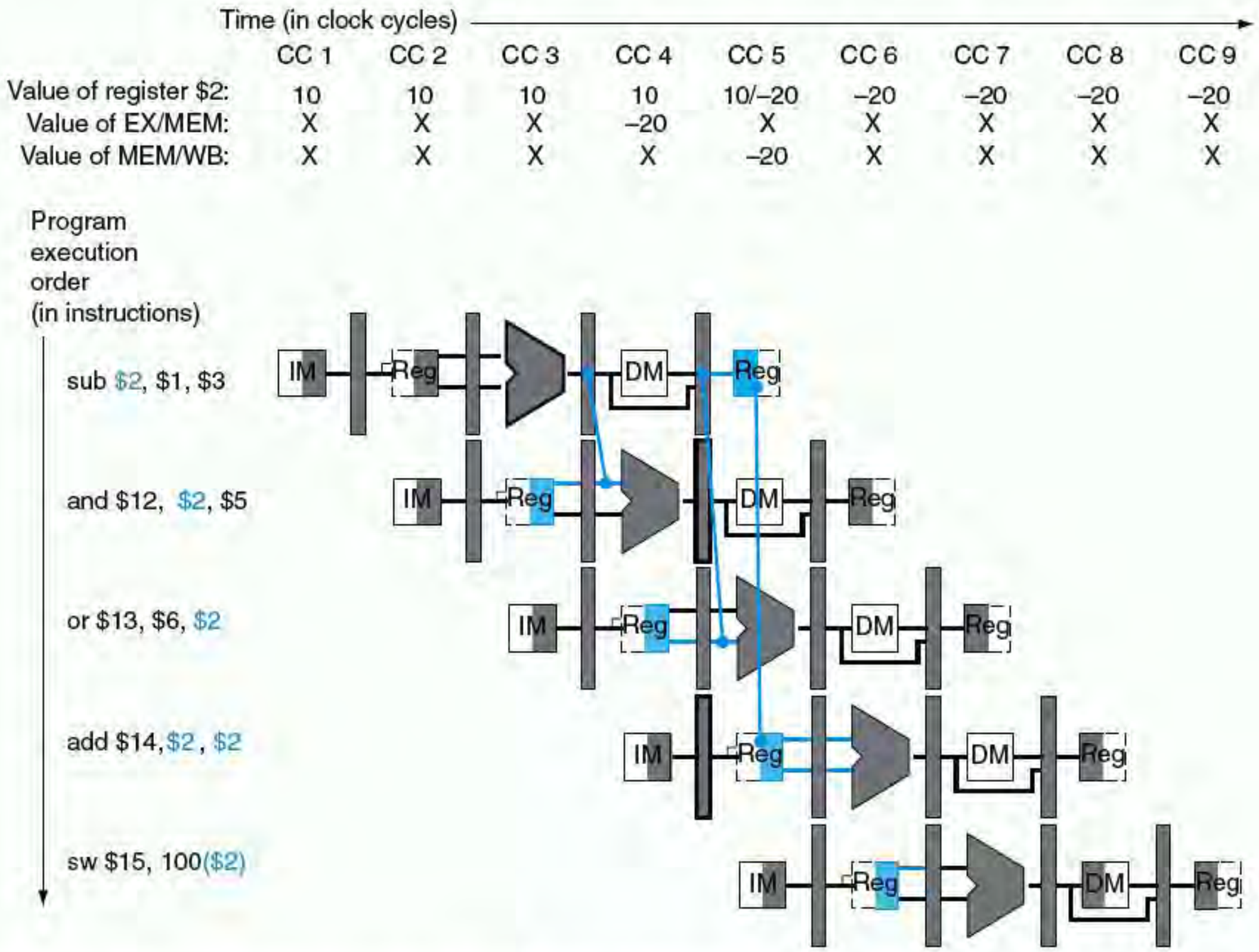


FIGURE 6.29 The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the `and` instruction and `or` instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the `add` does not stall, but the values come from the register file instead of a pipeline register. Register file "forwarding"—that is, the read gets the value of the write in that clock cycle—is why clock cycle 5 shows register $2 having the value 10 at the beginning and –20 at the end of the clock cycle. As in the rest of this section, we handle all forwarding except for the value to be stored by a store instruction.

This case forwards the result from the previous instruction to either input of the ALU. If the previous instruction is going to write to the register file and the write register number matches the read register number of ALU inputs A or B, provided it is not register 0, then steer the multiplexor to pick the value instead from the pipeline register EX/MEM.

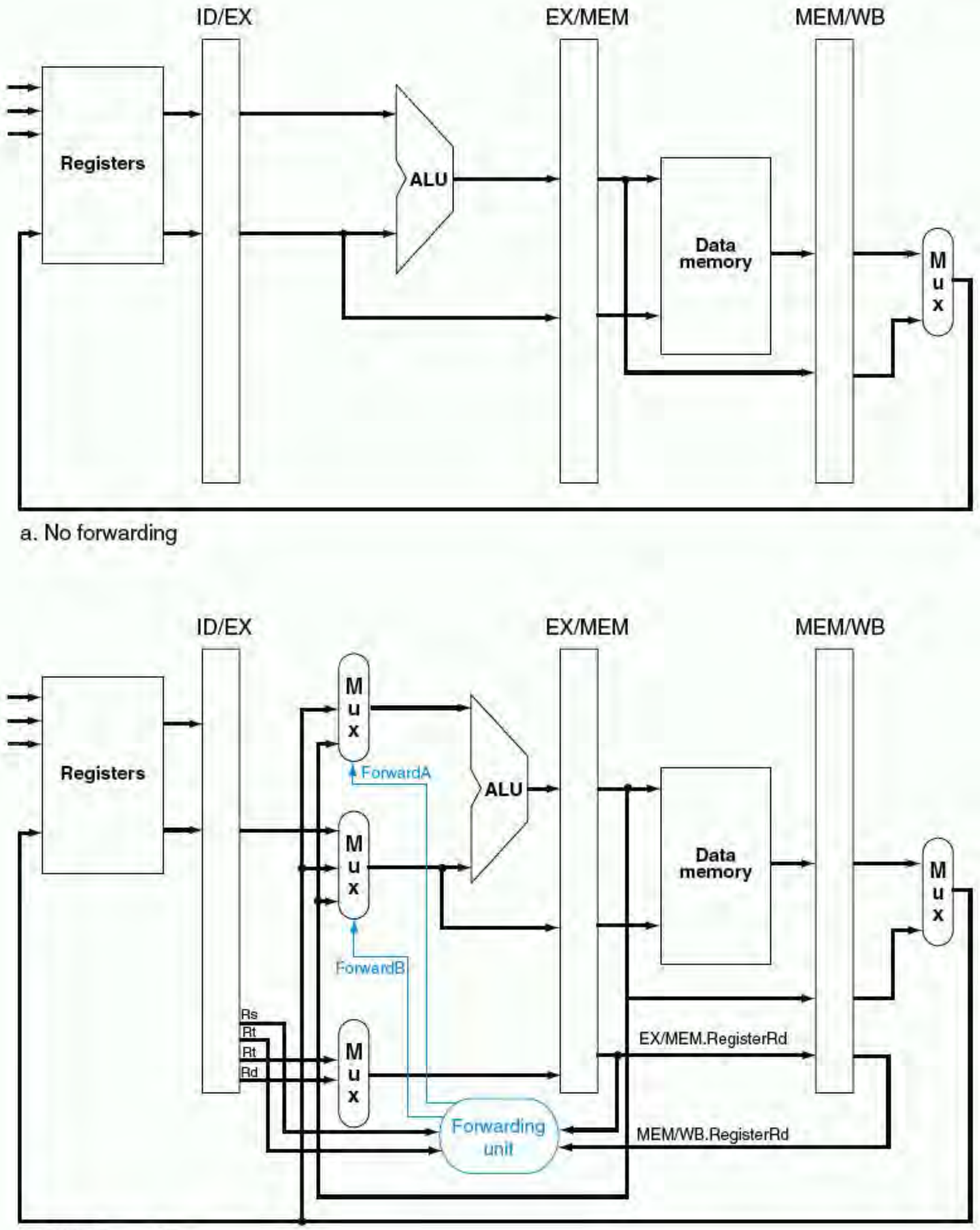


FIGURE 6.30 On the top are the ALU and pipeline registers before adding forwarding. On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware. Note that the ID/EX.RegisterRt field is shown twice, once to connect to the mux and once to the forwarding unit, but it is a single signal. As in the earlier discussion, this ignores forwarding of a store value to a store instruction.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

FIGURE 6.31 The control values for the forwarding multiplexors in Figure 6.30. The signed immediate that is another input to the ALU is described in the elaboration at the end of this section.

2. MEM hazard:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

As mentioned above, there is no hazard in the WB stage because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage. Such a register file performs another form of forwarding, but it occurs within the register file.

One complication is potential data hazards between the result of the instruction in the WB stage, the result of the instruction in the MEM stage, and the source operand of the instruction in the ALU stage. For example, when summing a vector of numbers in a single register, a sequence of instructions will all read and write to the same register:

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
. . .
```

In this case, the result is forwarded from the MEM stage because the result in the MEM stage is the more recent result. Thus the control for the MEM hazard would be (with the additions highlighted)

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Figure 6.32 shows the hardware necessary to support forwarding for operations that use results during the EX stage.

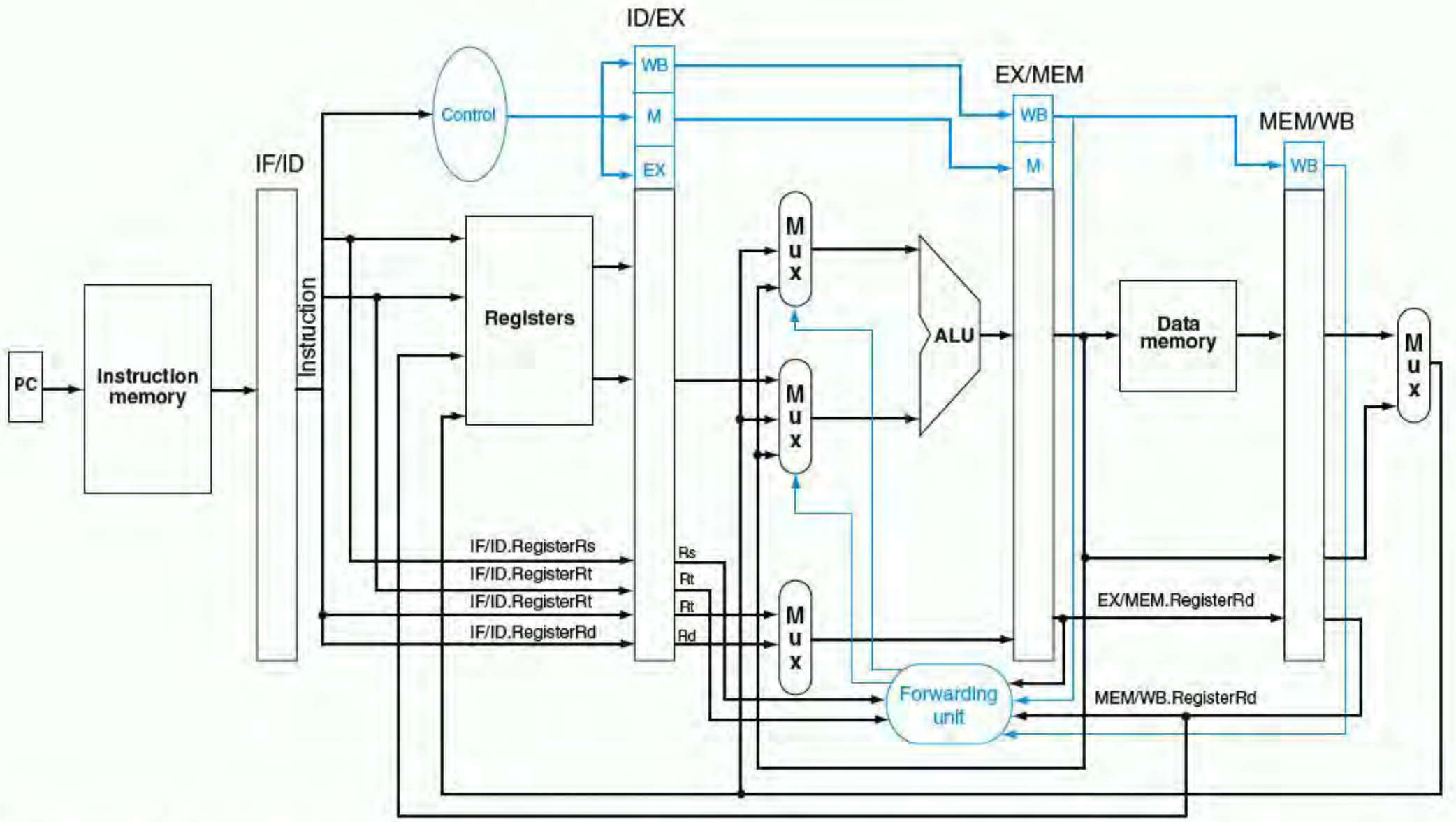


FIGURE 6.32 The datapath modified to resolve hazards via forwarding. Compared with the datapath in Figure 6.27 on page 404, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath such as the branch hardware and the sign extension hardware.

Elaboration: Forwarding can also help with hazards when store instructions are dependent on other instructions. Since they use just one data value during the MEM stage, forwarding is easy. But consider loads immediately followed by stores. We need to add more forwarding hardware to make memory-to-memory copies run faster. If we were to redraw Figure 6.29 on page 408, replacing the `sub` and `and` instructions by `lw` and an `sw`, we would see that it is possible to avoid a stall, since the data exists in the MEM/WB register of a load instruction in time for its use in the MEM stage of a store instruction. We would need to add forwarding into the memory access stage for this option. We leave this modification as an exercise.

In addition, the signed-immediate input to the ALU, needed by loads and stores, is missing from the datapath in Figure 6.32 on page 411. Since central control decides between register and immediate, and since the forwarding unit chooses the pipeline register for a register input to the ALU, the easiest solution is to add a 2:1 multiplexor that chooses between the ForwardB multiplexor output and the signed immediate. Figure 6.33 shows this addition. Note that this solution differs from what we learned in Chapter 5, where the multiplexor controlled by line ALUSrcB was expanded to include the immediate input.

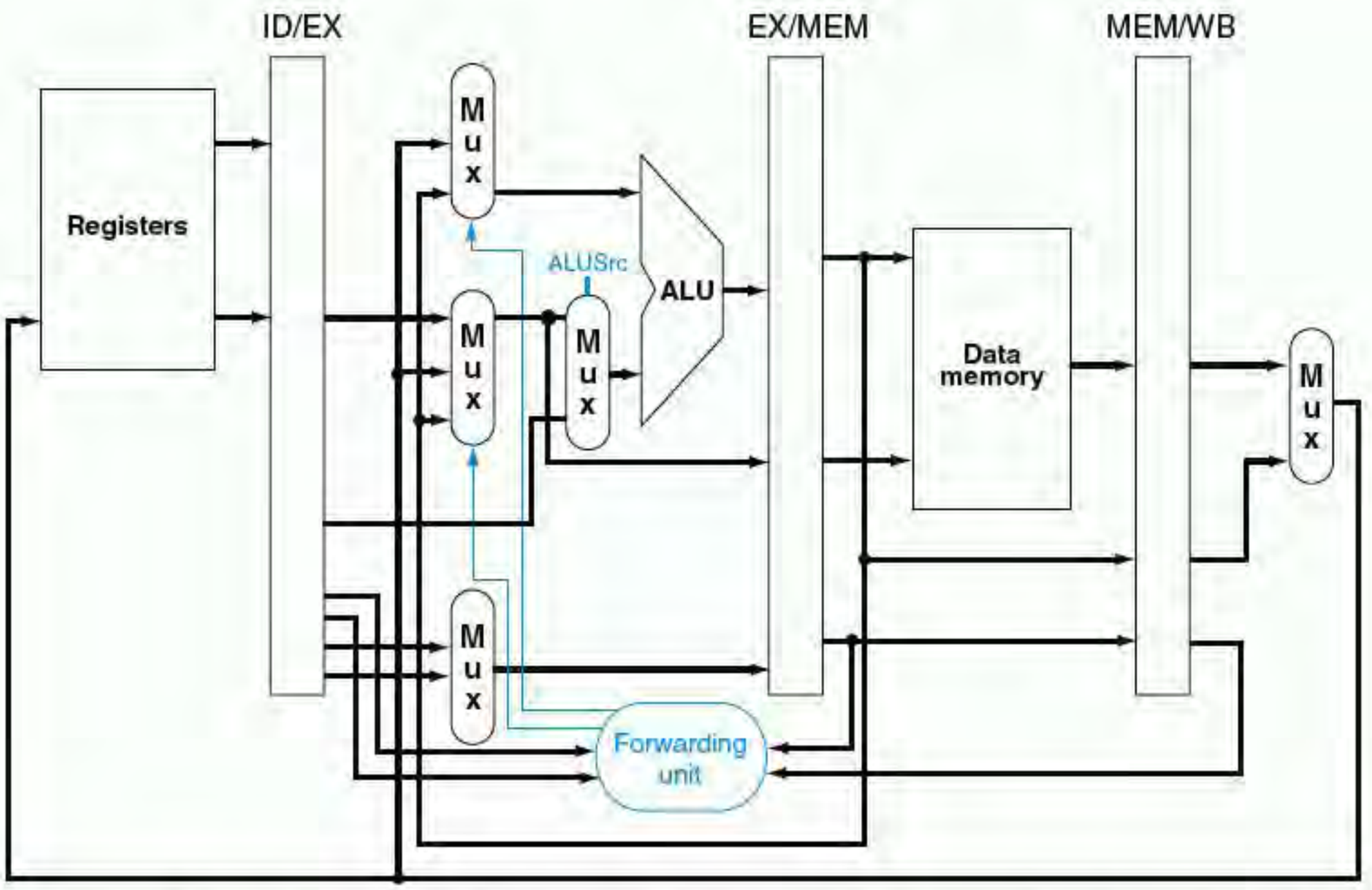


FIGURE 6.33 A close-up of the datapath in Figure 6.30 on page 409 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.

6.5 Data Hazards and Stalls

If at first you don't succeed, redefine success.

Anonymous

As we said in Section 6.1, one case where forwarding cannot save the day is when an instruction tries to read a register following a load instruction that writes the same register. Figure 6.34 illustrates the problem. The data is still being read from memory in clock cycle 4 while the ALU is performing the operation for the following instruction. Something must stall the pipeline for the combination of load followed by an instruction that reads its result.

Hence, in addition to a forwarding unit, we need a *hazard detection unit.* It operates during the ID stage so that it can insert the stall between the load and its use. Checking for load instructions, the control for the hazard detection unit is this single condition:

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
      stall the pipeline
```

The first line tests to see if the instruction is a load: the only instruction that reads data memory is a load. The next two lines check to see if the destination register field of the load in the EX stage matches either source register of the instruction in the ID stage. If the condition holds, the instruction stalls 1 clock cycle. After this 1-cycle stall, the forwarding logic can handle the dependence and execution proceeds. (If there were no forwarding, then the instructions in Figure 6.34 would need another stall cycle.)

If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled; otherwise, we would lose the fetched instruction. Preventing these two instructions from making progress is accomplished simply by preventing the PC register and the IF/ID pipeline register from changing. Provided these registers are preserved, the instruction in the IF stage will continue to be read using the same PC, and the registers in the ID stage will continue to be read using the same instruction fields in the IF/ID pipeline register. Returning to our favorite analogy, it's as if you restart the washer with the same clothes and let the dryer continue tumbling empty. Of course, like the dryer, the back half of the pipeline starting with the EX stage must be doing something; what it is doing is executing instructions that have no effect: **nops**.

nop An instruction that does no operation to change state.

How can we insert these nops, which act like bubbles, into the pipeline? In Figure 6.25 on page 401, we see that deasserting all nine control signals (setting them to 0) in the EX, MEM, and WB stages will create a "do nothing" or nop instruction. By identifying the hazard in the ID stage, we can insert a bubble into

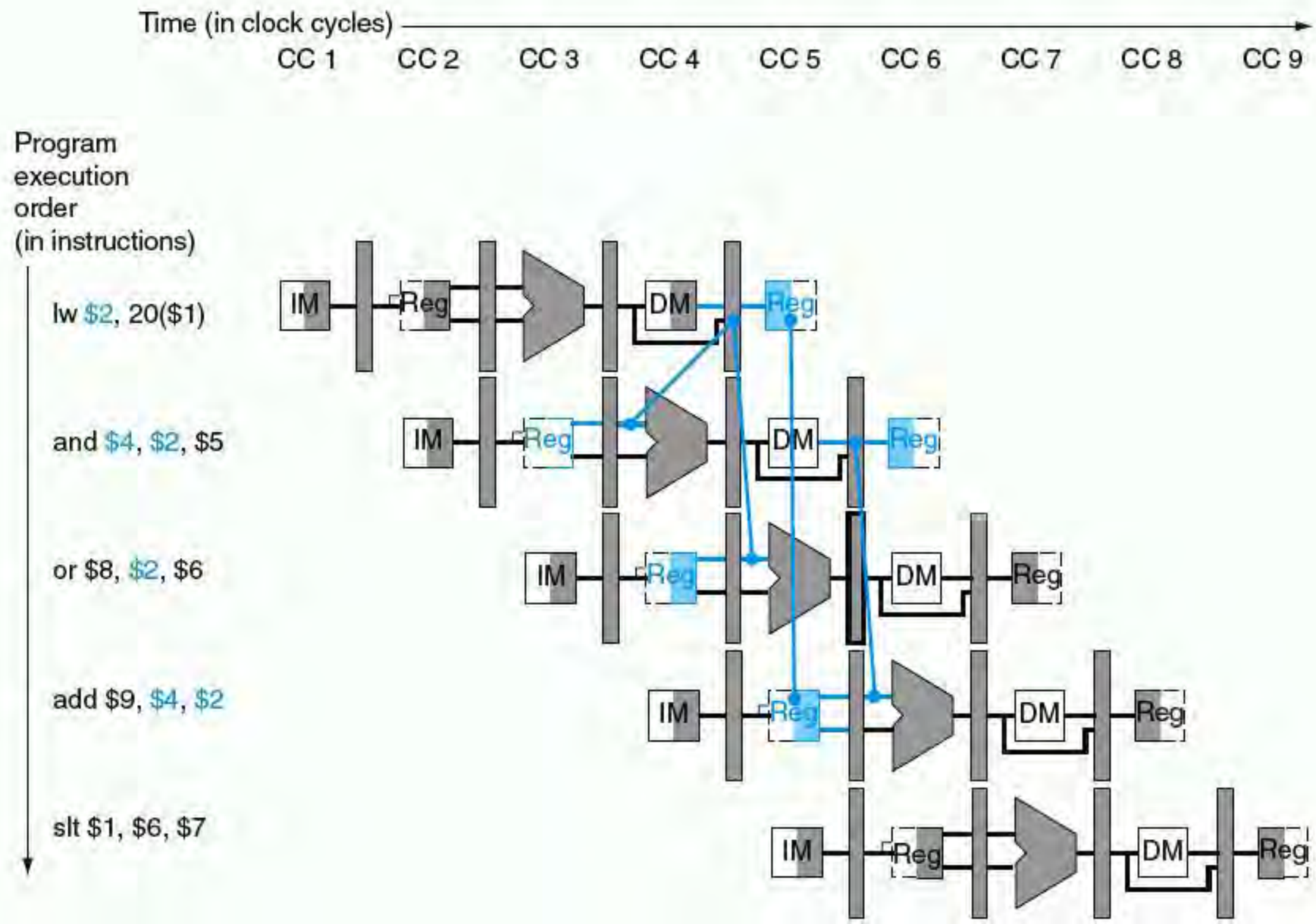


FIGURE 6.34 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (`and`) goes backwards in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

the pipeline by changing the EX, MEM, and WB control fields of the ID/EX pipeline register to 0. These benign control values are percolated forward at each clock cycle with the proper effect: no registers or memories are written if the control values are all 0.

Figure 6.35 shows what really happens in the hardware: the pipeline execution slot associated with the `and` instruction is turned into a nop and all instructions beginning with the `and` instruction are delayed one cycle. The hazard forces the `and` and `or` instructions to repeat in clock cycle 4 what they did in clock cycle 3: `and` reads registers and decodes, and `or` is refetched from instruction memory. Such repeated work is what a stall looks like, but its effect is to stretch the time of the `and` and `or` instructions and delay the fetch of the `add` instruction. Like an air bubble in a water pipe, a stall bubble delays everything behind it and proceeds down the instruction pipe one stage each cycle until it exits at the end.

Figure 6.36 highlights the pipeline connections for both the hazard detection unit and the forwarding unit. As before, the forwarding unit controls the ALU multiplexors to replace the value from a general-purpose register with the value

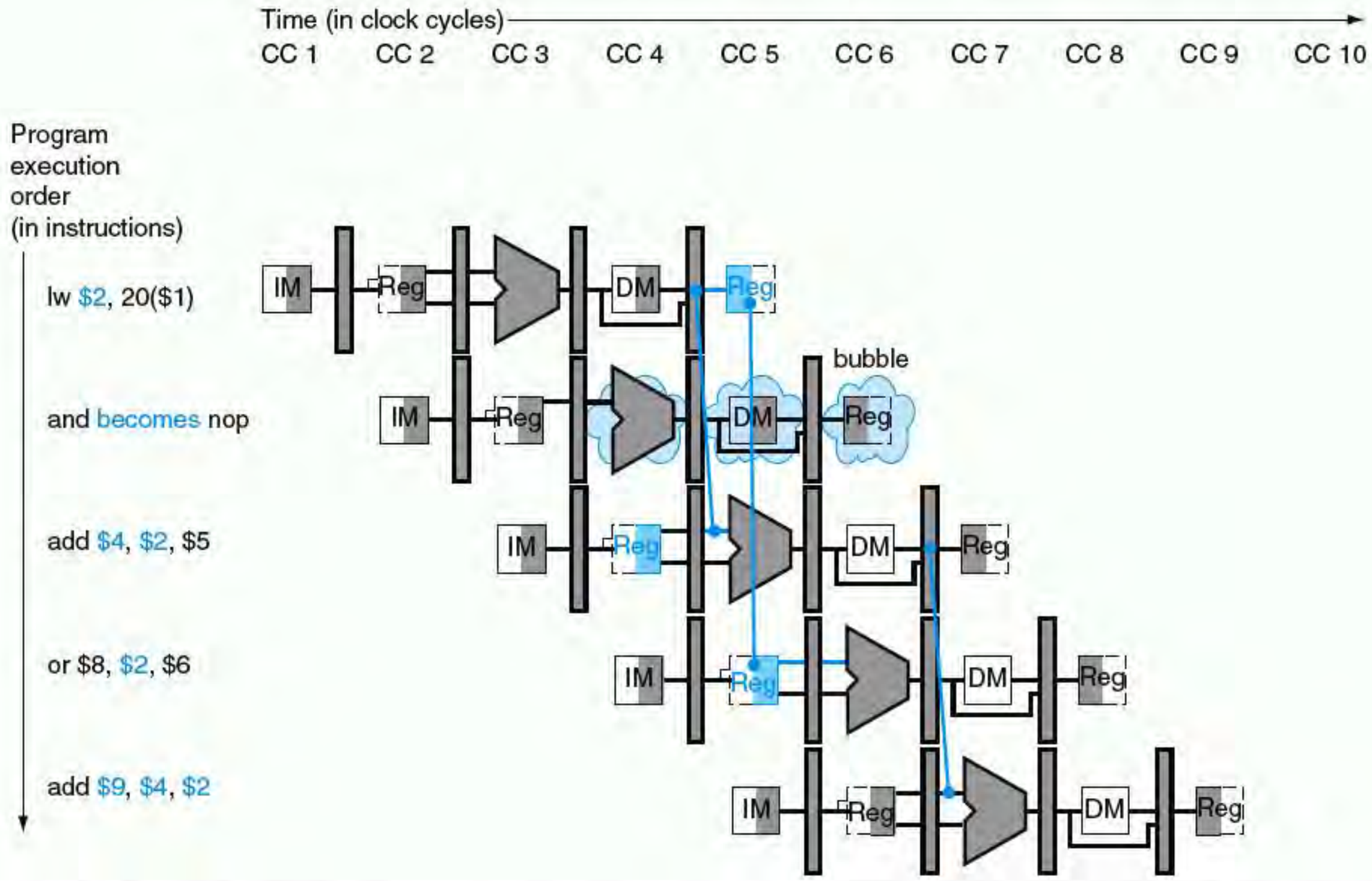


FIGURE 6.35 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the `and` instruction to a nop. Note that the `and` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise the `or` instruction is fetched in clock cycle 3, but its IF stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

from the proper pipeline register. The hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexor that chooses between the real control values and all 0s. The hazard detection unit stalls and deasserts the control fields if the load-use hazard test above is true. We show the single-clock-cycle diagrams in the For More Practice section on the CD.

Although the hardware may or may not rely on the compiler to resolve hazard dependences to ensure correct execution, the compiler must understand the pipeline to achieve the best performance. Otherwise, unexpected stalls will reduce the performance of the compiled code.

The BIG Picture

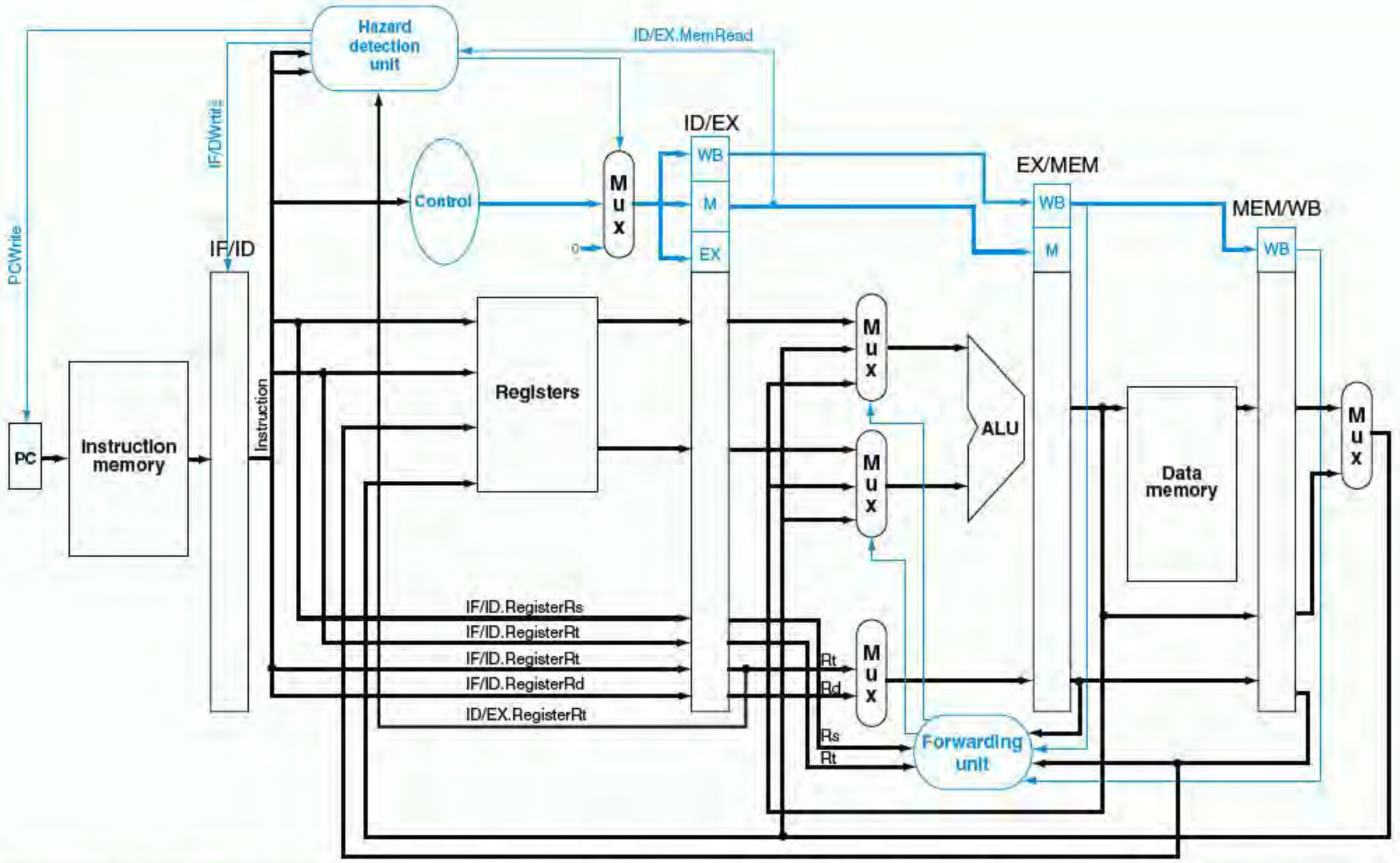


FIGURE 6.36 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

Elaboration: Regarding the remark earlier about setting control lines to 0 to avoid writing registers or memory: only the signals RegWrite and MemWrite need be 0, while the other control signals can be don't cares.

There are a thousand hacking at the branches of evil to one who is striking at the root.

Henry David Thoreau, *Walden*, 1854

6.6 Branch Hazards

Thus far we have limited our concern to hazards involving arithmetic operations and data transfers. But as we saw in Section 6.1, there are also pipeline hazards involving branches. Figure 6.37 shows a sequence of instructions and indicates

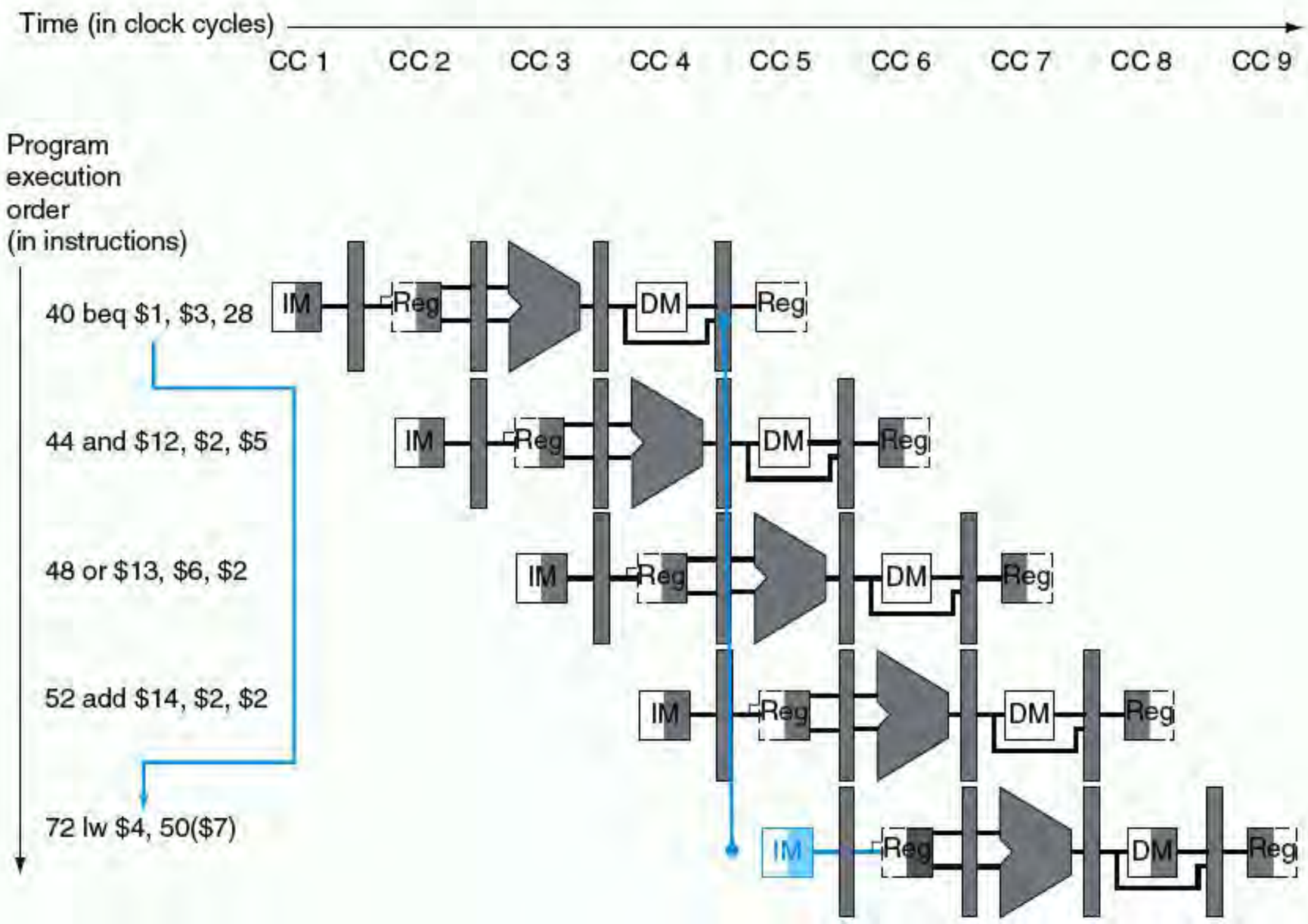


FIGURE 6.37 The impact of the pipeline on the branch instruction. The numbers to the left of the instruction (40, 44, . . .) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the `beq` instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before `beq` branches to `lw` at location 72. (Figure 6.7 on page 380 assumed extra hardware to reduce the control hazard to 1 clock cycle; this figure uses the nonoptimized datapath.)

when the branch would occur in this pipeline. An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to branch doesn't occur until the MEM pipeline stage. As mentioned in Section 6.1, this delay in determining the proper instruction to fetch is called a *control hazard* or *branch hazard*, in contrast to the *data hazards* we have just examined.

This section on control hazards is shorter than the previous sections on data hazards. The reasons are that control hazards are relatively simple to understand, they occur less frequently than data hazards, and there is nothing as effective against control hazards as forwarding is for data hazards. Hence, we use simpler schemes. We look at two schemes for resolving control hazards and one optimization to improve these schemes.

Assume Branch Not Taken

As we saw in Section 6.1, stalling until the branch is complete is too slow. A common improvement over branch stalling is to assume that the branch will not be taken and thus continue execution down the sequential instruction stream. If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

To discard instructions, we merely change the original control values to 0s, much as we did to stall for a load-use data hazard. The difference is that we must also change the three instructions in the IF, ID, and EX stages when the branch reaches the MEM stage; for load-use stalls, we just changed control to 0 in the ID stage and let them percolate through the pipeline. Discarding instructions, then, means we must be able to **flush instructions** in the IF, ID, and EX stages of the pipeline.

flush (instructions) To discard instructions in a pipeline, usually due to an unexpected event.

Reducing the Delay of Branches

One way to improve branch performance is to reduce the cost of the taken branch. Thus far we have assumed the next PC for a branch is selected in the MEM stage, but if we move the branch execution earlier in the pipeline, then fewer instructions need be flushed. The MIPS architecture was designed to support fast single-cycle branches that could be pipelined with a small branch penalty. The designers observed that many branches rely only on simple tests (equality or sign, for example) and that such tests do not require a full ALU operation but can be done with at most a few gates. When a more complex branch decision is required, a separate instruction that uses an ALU to perform a comparison is required—a situation that is similar to the use of condition codes for branches.

Moving the branch decision up requires two actions to occur earlier: computing the branch target address and evaluating the branch decision. The easy part of this change is to move up the branch address calculation. We already have the PC value and the immediate field in the IF/ID pipeline register, so we just move the branch adder from the EX stage to the ID stage; of course, the branch target address calculation will be performed for all instructions, but only used when needed.

The harder part is the branch decision itself. For branch equal, we would compare the two registers read during the ID stage to see if they are equal. Equality can be tested by first exclusive ORing their respective bits and then ORing all the results. Moving the branch test to the ID stage implies additional forwarding and hazard detection hardware, since a branch dependent on a result still in the pipeline must still work properly with this optimization. For example, to implement branch-on-equal (and its inverse), we will need to forward results to the equality test logic that operates during ID. There are two complicating factors:

1. During ID, we must decode the instruction, decide whether a bypass to the equality unit is needed, and complete the equality comparison so that if the instruction is a branch, we can set the PC to the branch target address. Forwarding for the operands of branches was formerly handled by the ALU forwarding logic, but the introduction of the equality test unit in ID will require new forwarding logic. Note that the bypassed source operands of a branch can come from either the ALU/MEM or MEM/WB pipeline latches.
2. Because the values in a branch comparison are needed during ID but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed. For example, if an ALU instruction immediately preceding a branch produces one of the operands for the comparison in the branch, a stall will be required, since the EX stage for the ALU instruction will occur after the ID cycle of the branch.

Despite these difficulties, moving the branch execution to the ID stage is an improvement since it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched. The exercises explore the details of implementing the forwarding path and detecting the hazard.

To flush instructions in the IF stage, we add a control line, called IF.Flush, that zeros the instruction field of the IF/ID pipeline register. Clearing the register transforms the fetched instruction into a `nop`, an instruction that has no action and changes no state.

Pipelined Branch

EXAMPLE

Show what happens when the branch is taken in this instruction sequence, assuming the pipeline is optimized for branches that are not taken and that we moved the branch execution to the ID stage:

```
36 sub $10, $4, $8
40 beq  $1, $3, 7 # PC-relative branch to 40 + 4
+ 7 * 4 = 72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw  $4,  50($7)
```

ANSWER

Figure 6.38 shows what happens when a branch is taken. Unlike Figure 6.37, there is only one pipeline bubble on a taken branch.

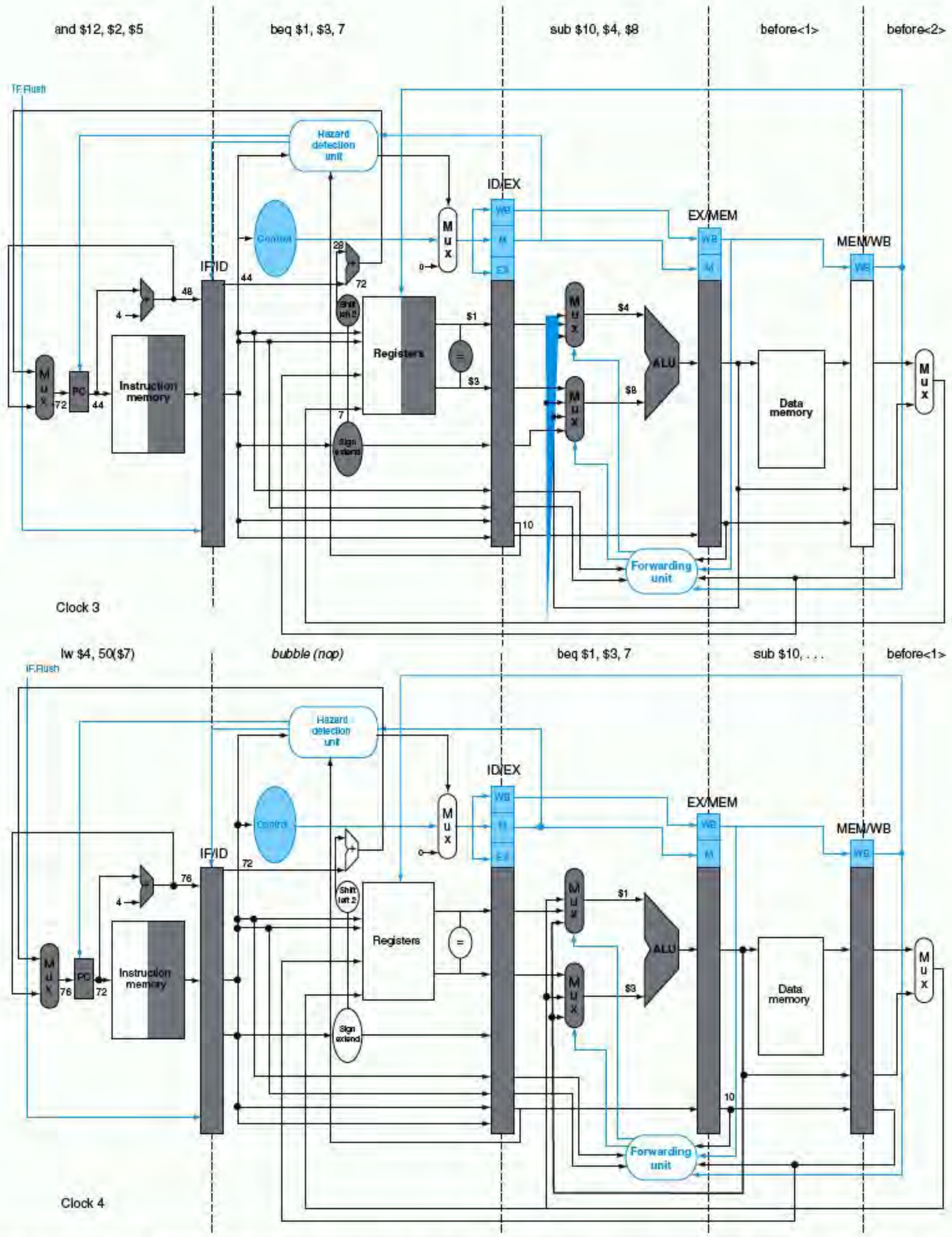


FIGURE 6.38 The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle. Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or `nop` instruction in the pipeline as a result of the taken branch. (Since the `nop` is really `sll $0, $0, 0`, it's arguable whether or not the ID stage in clock 4 should be highlighted.)

Dynamic Branch Prediction

Assuming a branch is not taken is one simple form of *branch prediction*. In that case, we predict that branches are untaken, flushing the pipeline when we are wrong. For the simple five-stage pipeline, such an approach, possibly coupled with compiler-based prediction, is probably adequate. With deeper pipelines, the branch penalty increases when measured in clock cycles. Similarly, with multiple issue, the branch penalty increases in terms of instructions lost. This combination means that in an aggressive pipeline, a simple static prediction scheme will probably waste too much performance. As we mentioned in Section 6.1, with more hardware it is possible to try to predict branch behavior during program execution.

One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and, if so, to begin fetching new instructions from the same place as the last time. This technique is called **dynamic branch prediction**.

dynamic branch prediction Prediction of branches at runtime using runtime information.

One implementation of that approach is a **branch prediction buffer** or **branch history table**. A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

branch prediction buffer Also called **branch history table**. A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not.

This is the simplest sort of buffer; we don't know, in fact, if the prediction is the right one—it may have been put there by another branch that has the same low-order address bits. But this doesn't affect correctness. Prediction is just a hint that is assumed to be correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken. The following example shows this dilemma.

Loops and Prediction

EXAMPLE

Consider a loop branch that branches nine times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?

ANSWER

The steady-state prediction behavior will mispredict on the first and last loop iterations. Mispredicting the last iteration is inevitable since the prediction bit will say taken: the branch has been taken nine times in a row at that point. The misprediction on the first iteration happens because the bit is flipped on prior execution of the last iteration of the loop, since the branch was not taken on that exiting iteration. Thus, the prediction accuracy for this branch that is taken 90% of the time is only 80% (two incorrect predictions and eight correct ones).

Ideally, the accuracy of the predictor would match the taken branch frequency for these highly regular branches. To remedy this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must be wrong twice before it is changed. Figure 6.39 shows the finite state machine for a 2-bit prediction scheme.

A branch prediction buffer can be implemented as a small, special buffer accessed with the instruction address during the IF pipe stage. If the instruction is predicted as taken, fetching begins from the target as soon as the PC is known; as mentioned on page 418, it can be as early as the ID stage. Otherwise, sequential

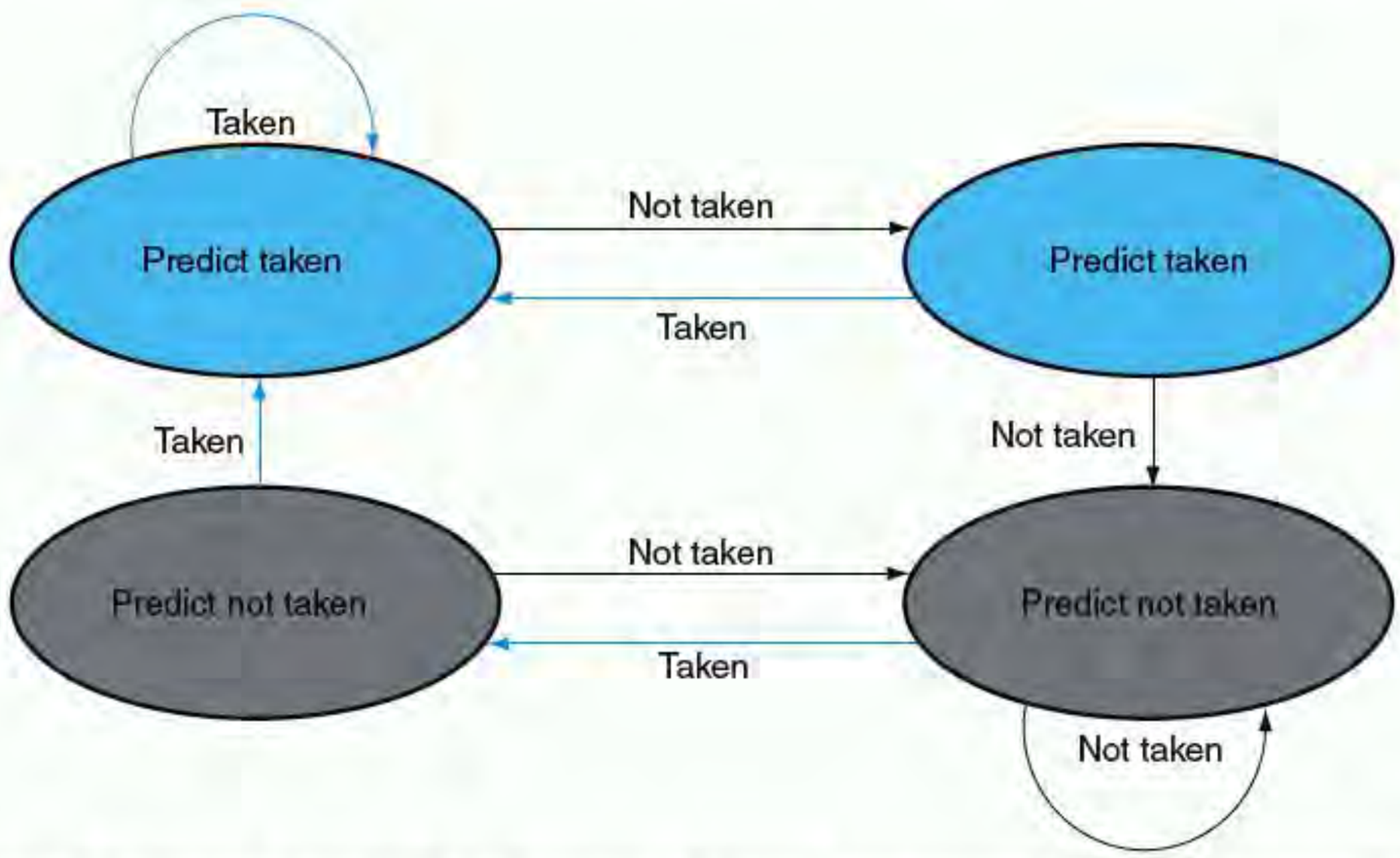


FIGURE 6.39 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The two-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the midpoint of its range as the division between taken and not taken.

fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed as shown in Figure 6.39.

Elaboration: As we described in Section 6.1, in a five-stage pipeline we can make the control hazard a feature by redefining the branch. A delayed branch always executes the following instruction, but the second instruction following the branch will be affected by the branch.

Compilers and assemblers try to place an instruction that always executes after the branch in the branch delay slot. The job of the software is to make the successor instructions valid and useful. Figure 6.40 shows the three ways in which the branch delay slot can be scheduled.

The limitations on delayed-branch scheduling arise from (1) the restrictions on the instructions that are scheduled into the delay slots and (2) our ability to predict at compile time whether a branch is likely to be taken or not.

Delayed branching was a simple and effective solution for a five-stage pipeline issuing one instruction each clock cycle. As processors go to both longer pipelines and issuing multiple instructions per clock cycle (see Section 6.9), the branch delay becomes longer and a single delay slot is insufficient. Hence, delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches. Simultaneously, the growth in available transistors per chip has made dynamic prediction relatively cheaper.

branch delay slot The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.

Elaboration: A branch predictor tells us whether or not a branch is taken, but still requires the calculation of the branch target. In the five-stage pipeline, this calculation takes 1 cycle, meaning that taken branches will have a 1-cycle penalty. Delayed branches are one approach to eliminate that penalty. Another approach is to use a cache to hold the destination program counter or destination instruction, using a branch target buffer.

branch target buffer A structure that caches the destination PC or destination instruction for a branch. It is usually organized as a cache with tags, making it more costly than a simple prediction buffer.

Elaboration: The 2-bit dynamic prediction scheme uses only information about a particular branch. Researchers noticed that using information about both a local branch and the global behavior of recently executed branches together yields greater prediction accuracy for the same number of prediction bits. Such predictors are called correlating predictors. A typical correlating predictor might have two 2-bit predictors for each branch with the choice between predictors made on the basis of whether the last executed branch was taken or not taken. Thus, the global branch behavior can be thought of as adding additional index bits for the prediction lookup.

correlating predictor A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.

A more recent innovation in branch prediction is the use of tournament predictors. A tournament predictor uses multiple predictors, tracking, for each branch, which predictor yields the best results. A typical tournament predictor might contain two predictions for each branch index: one based on local information and one based on global branch behavior. A selector would choose which predictor to use for any given prediction. The selector can operate similarly to a 1- or 2-bit predictor favoring whichever of the two predictors has been more accurate. Many recent advanced microprocessors make use of such elaborate predictors.

tournament branch predictor A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

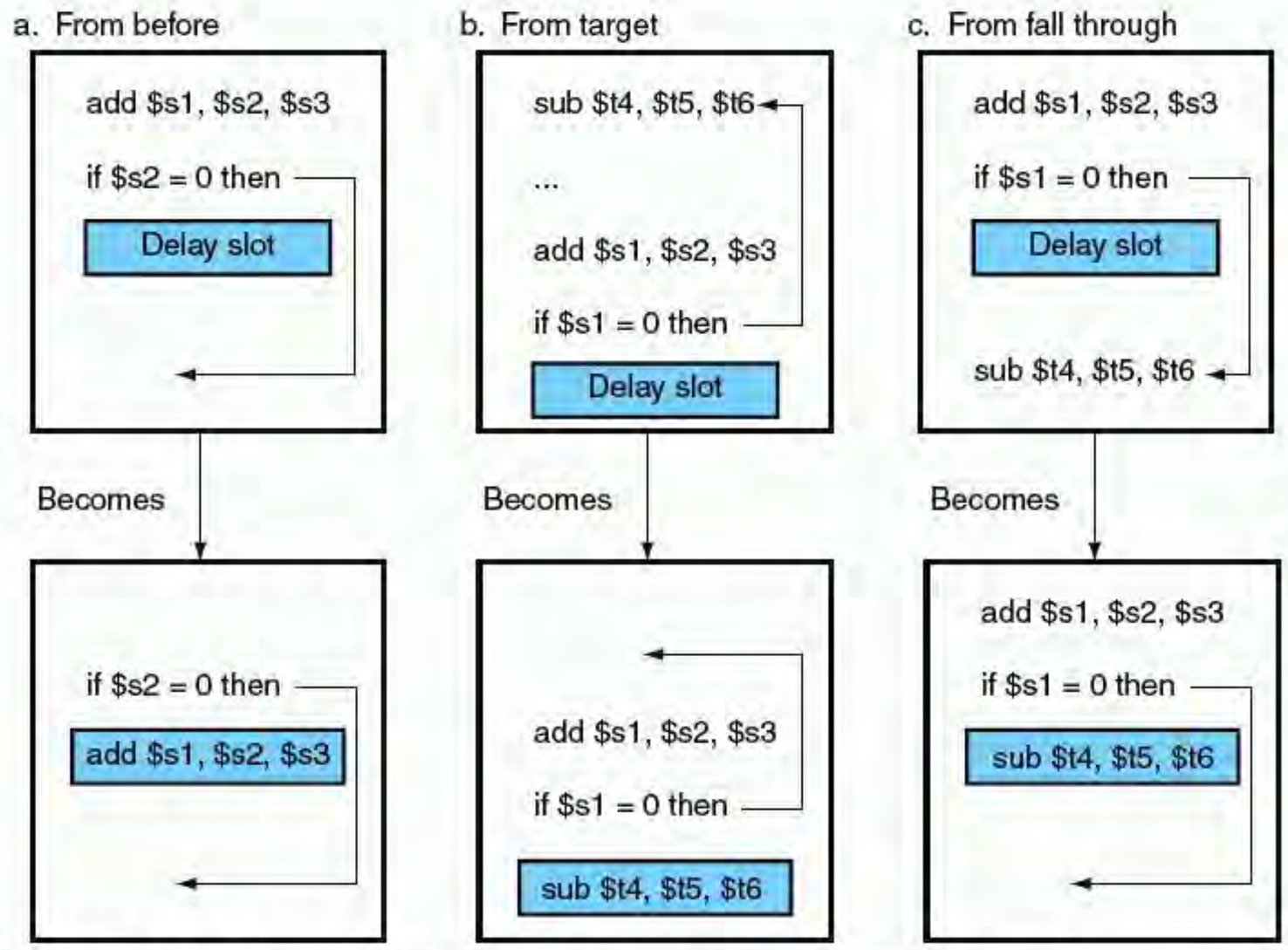


FIGURE 6.40 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of `$s1` in the branch condition prevents the `add` instruction (whose destination is `$s1`) from being moved into the branch delay slot. In (b) the branch-delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the `sub` instruction when the branch goes in the unexpected direction. By "OK" we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, if `$t4` were an unused temporary register when the branch goes in the unexpected direction.

Pipeline Summary

Thus far, we have seen three models of execution: single cycle, multicycle, and pipelined. Pipelined control strives for 1 clock cycle per instruction, like single cycle, but also for a fast clock cycle, like multicycle. Let's revisit the example comparison of single-cycle and multicycle processors.

Comparing Performance of Several Control Schemes

EXAMPLE

Compare performance for single-cycle, multicycle, and pipelined control using the SPECint2000 instruction mix (see examples on pages 315 and 330) and assuming the same cycle times per unit as the example on page 315. For pipelined execution, assume that half of the load instructions are immediately followed by an instruction that uses the result, that the branch delay on misprediction is 1 clock cycle, and that one-quarter of the branches are mispredicted. Assume that jumps always pay 1 full clock cycle of delay, so their average time is 2 clock cycles. Ignore any other hazards.

ANSWER

From the example on page 315 (Performance of Single-Cycle Machines), we get the following functional unit times:

- 200 ps for memory access
- 100 ps for ALU operation
- 50 ps for register file read or write

For the single-cycle datapath, this leads to a clock cycle of

$$200 + 50 + 100 + 200 + 50 = 600 \text{ ps}$$

The example on page 330 (CPI in a Multicycle CPU) has the following instruction frequencies:

- 25% loads
- 10% stores
- 11% branches
- 2% jumps
- 52% ALU instructions

Furthermore, the example on page 330 showed that the CPI for the multiple design was 4.12. The clock cycle for the multicycle datapath and the pipelined design must be the same as the longest functional unit: 200 ps.

For the pipelined design, loads take 1 clock cycle when there is no load-use dependence and 2 when there is. Hence, the average clock cycles per load instruction is 1.5. Stores take 1 clock cycle, as do the ALU instructions. Branches take 1 when predicted correctly and 2 when not, so the average clock cycles per branch instruction is 1.25. The jump CPI is 2. Hence the average CPI is

$$1.5 \times 25\% + 1 \times 10\% + 1 \times 52\% + 1.25 \times 11\% + 2 \times 2\% = 1.17$$

Let's compare the three designs by the average instruction time. For the single-cycle design, it is fixed at 600 ps. For the multicycle design, it is 200 x 4.12 = 824 ps. For the pipelined design, the average instruction time is 1.17 x 200 = 234 ps, making it almost twice as fast as either approach.

The clever reader will notice that the long cycle time of the memory is a performance bottleneck for both the pipelined and multicycle designs. Breaking memory accesses into two clock cycles and thereby allowing the clock cycle to be 100 ps would improve the performance in both cases. We explore this in the exercises.

This chapter started in the laundry room, showing principles of pipelining in an everyday setting. Using that analogy as a guide, we explained instruction pipelining step-by-step, starting with the single-cycle datapath and then adding pipeline registers, forwarding paths, data hazard detection, branch prediction, and flushing instructions on exceptions. Figure 6.41 shows the final evolved datapath and control.

Check Yourself

Consider three branch prediction schemes: branch not taken, predict taken, and dynamic prediction. Assume that they all have zero penalty when they predict correctly and 2 cycles when they are wrong. Assume that the average predict accuracy of the dynamic predictor is 90%. Which predictor is the best choice for the following branches?

1. A branch that is taken with 5% frequency
2. A branch that is taken with 95% frequency
3. A branch that is taken with 70% frequency



6.7 Using a Hardware Description Language to Describe and Model a Pipeline

This section, which appears on the CD, provides a behavioral model in Verilog of the MIPS five-stage pipeline. The initial model ignores hazards, and additions to the model highlight the changes for forwarding, data hazards, and branch hazards.

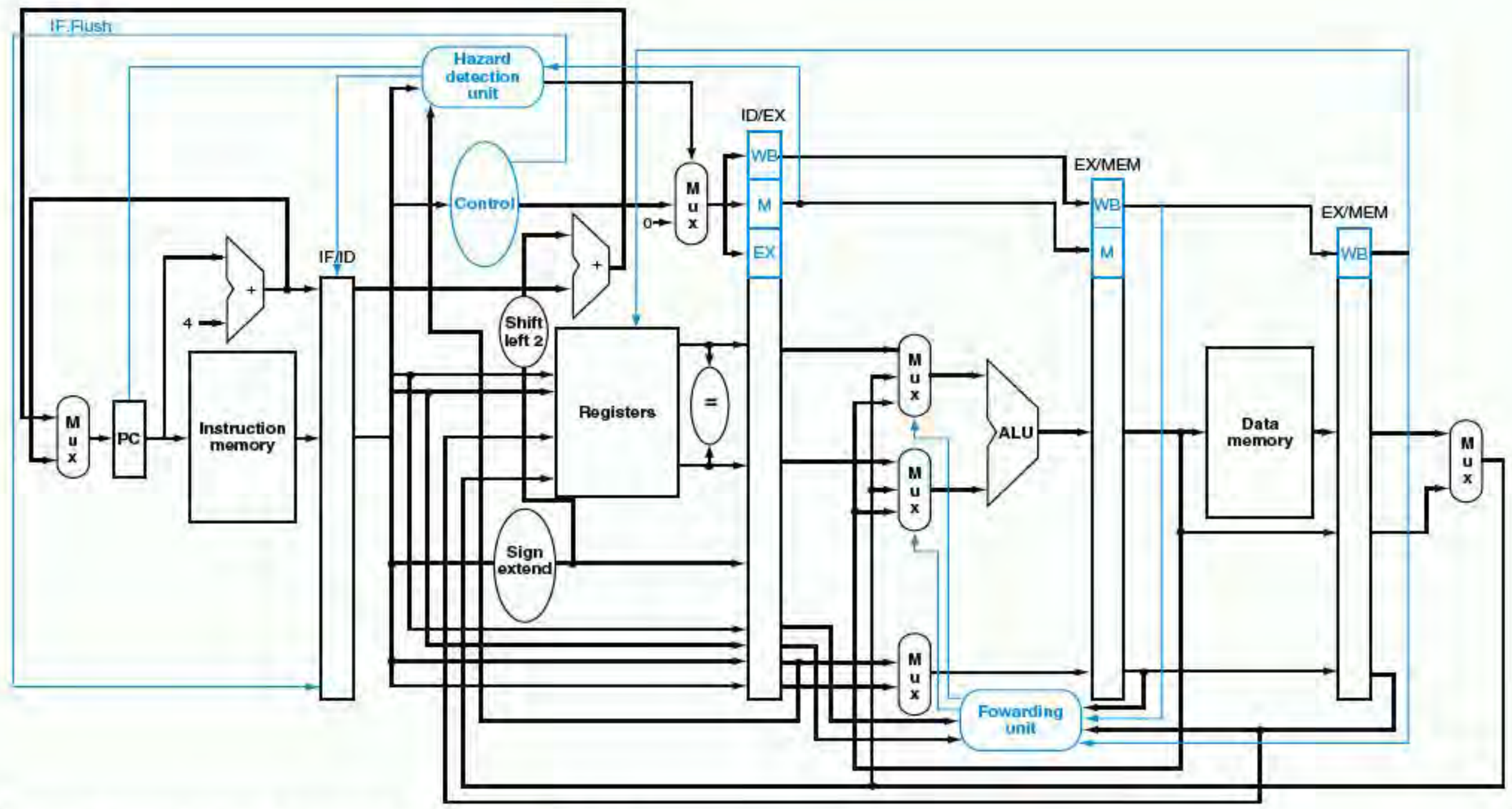


FIGURE 6.41 The final datapath and control for this chapter.

6.8 Exceptions

To make a computer with automatic program-interruption facilities behave [sequentially] was not an easy matter, because the number of instructions in various stages of processing when an interrupt signal occurs may be large.

Fred Brooks Jr., *Planning a Computer System: Project Stretch*, 1962

Another form of control hazard involves exceptions. For example, suppose the following instruction

```
add  $1,$2,$1
```

has an arithmetic overflow. We need to transfer control to the exception routine immediately after this instruction because we wouldn't want this invalid value to contaminate other registers or memory locations.

Just as we did for the taken branch in the previous section, we must flush the instructions that follow the `add` instruction from the pipeline and begin fetching instructions from the new address. We will use the same mechanism we used for taken branches, but this time the exception causes the deasserting of control lines.

When we dealt with branch mispredict, we saw how to flush the instruction in the IF stage by turning it into a `nop`. To flush instructions in the ID stage, we use the multiplexor already in the ID stage that zeros control signals for stalls. A new

control signal, called ID.Flush, is ORed with the stall signal from the Hazard Detection Unit to flush during ID. To flush the instruction in the EX phase, we use a new signal called EX.Flush to cause new multiplexors to zero the control lines. To start fetching instructions from location 8000 0180_{hex}, which is the exception location for an arithmetic overflow, we simply add an additional input to the PC multiplexor that sends 8000 0180_{hex} to the PC. Figure 6.42 shows these changes.

This example points out a problem with exceptions: If we do not stop execution in the middle of the instruction, the programmer will not be able to see the original value of register $1 that helped cause the overflow because it will be clobbered as the destination register of the `add` instruction. Because of careful planning, the overflow exception is detected during the EX stage; hence, we can use the EX.Flush signal to prevent the instruction in the EX stage from writing its result in the WB stage. Many exceptions require that we eventually complete the instruction that caused the exception as if it executed normally. The easiest way to do this

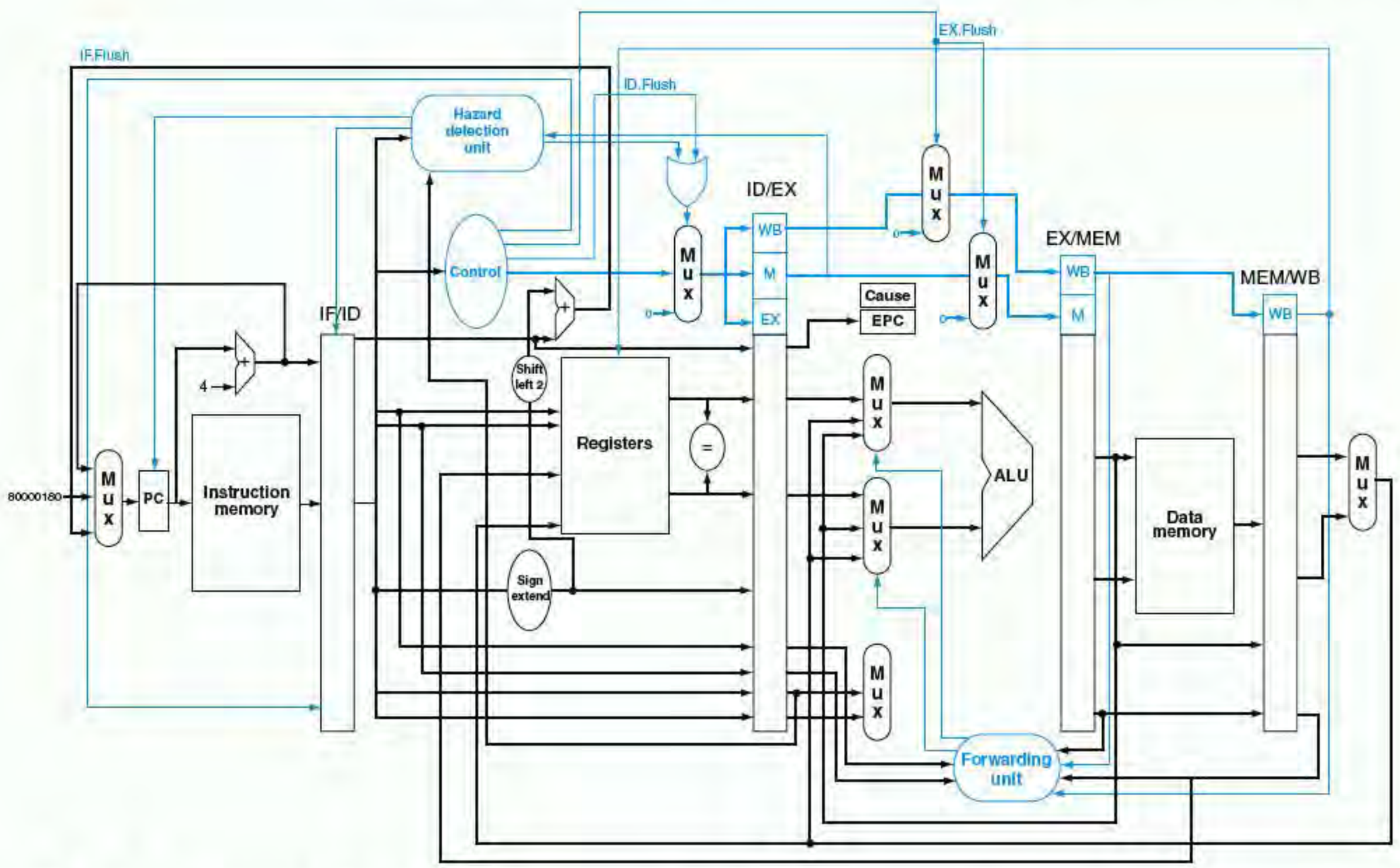


FIGURE 6.42 The datapath with controls to handle exceptions. The key additions include a new input, with the value 8000 0180_{hex}, in the multiplexor that supplies the new PC value; a Cause register to record the cause of the exception; and an Exception PC register to save the address of the instruction that caused the exception. The 8000 0180_{hex} input to the multiplexor is the initial address to begin fetching instructions in the event of an exception. Although not shown, the ALU overflow signal is an input to the control unit.

is to flush the instruction and restart it from the beginning after the exception is handled.

The final step is to save the address of the offending instruction in the Exception Program Counter (EPC), as we did in Chapter 5. In reality, we save the address + 4, so the exception handling routine must first subtract 4 from the saved value. Figure 6.42 shows a stylized version of the datapath, including the branch hardware and necessary accommodations to handle exceptions.

Exception in a Pipelined Computer

EXAMPLE

Given this instruction sequence,

```
40hex      sub   $11, $2, $4
44hex      and   $12, $2, $5
48hex      or    $13, $2, $6
4Chex      add    $1, $2, $1
50hex      slt   $15, $6, $7
54hex      lw    $16, 50($7)
...
```

assume the instructions to be invoked on an exception begin like this:

```
40000040hex      sw      $25, 1000($0)
40000044hex      sw      $26, 1004($0)
...
```

Show what happens in the pipeline if an overflow exception occurs in the `add` instruction.

ANSWER

Figure 6.43 shows the events, starting with the `add` instruction in the EX stage. The overflow is detected during that phase, and 4000 0040_{hex} is forced into the PC. Clock cycle 7 shows that the `add` and following instructions are flushed, and the first instruction of the exception code is fetched. Note that the address of the instruction *following* the `add` is saved: $4C_{hex} + 4 = 50_{hex}$.

Chapter 5 lists some other causes of exceptions:

- I/O device request
- Invoking an operating system service from a user program
- Using an undefined instruction
- Hardware malfunction

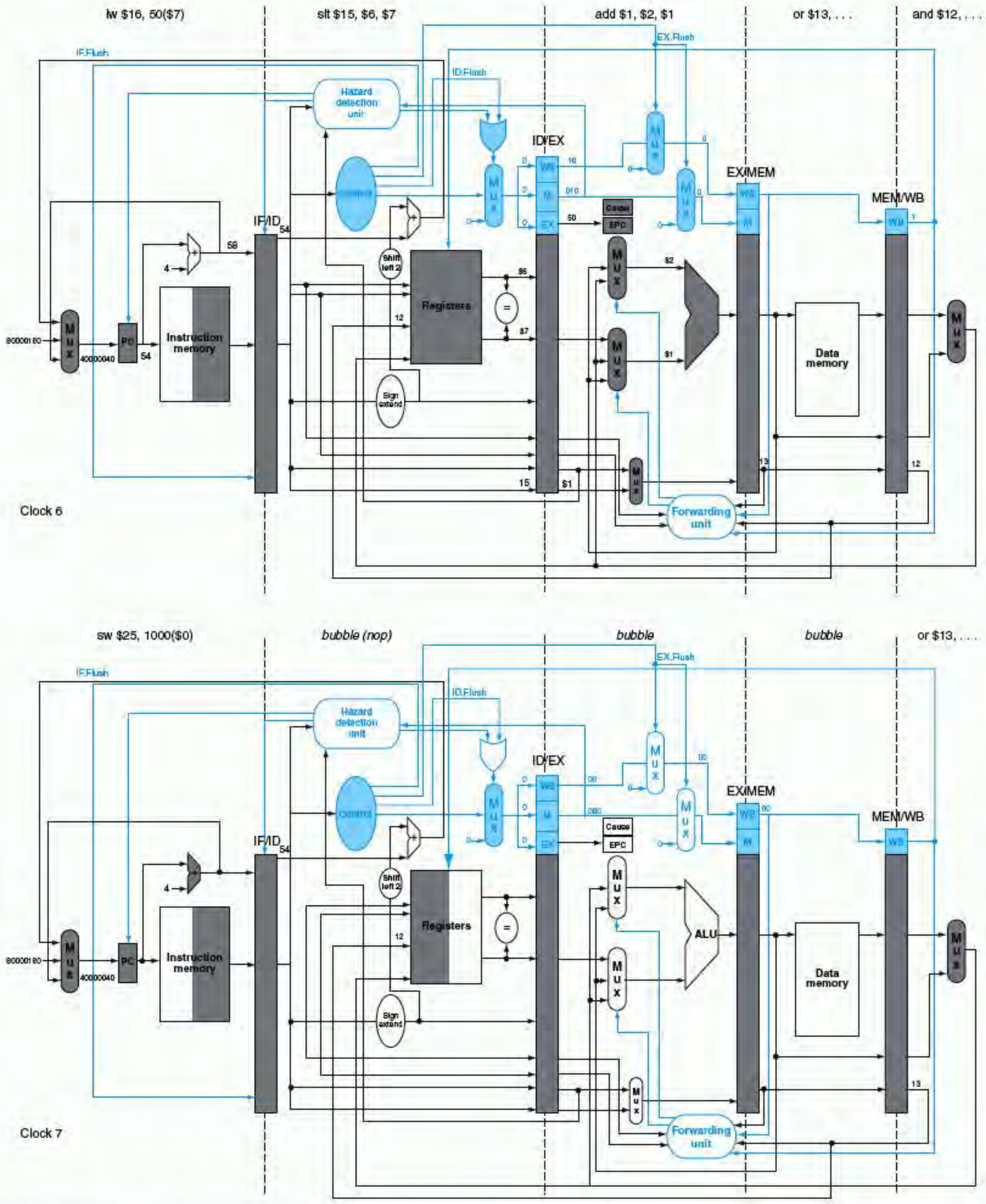


FIGURE 6.43 The result of an exception due to arithmetic overflow in the add instruction. The overflow is detected during the EX stage of clock 6, saving the address following the add in the EPC register ($4C + 4 = 50_{hex}$). Overflow causes all the Flush signals to be set near the end of this clock cycle, deasserting control values (setting them to 0) for the `add`. Clock cycle 7 shows the instructions converted to bubbles in the pipeline plus the fetching of the first instruction of the exception routine—`sw $25,1000($0)`—from instruction location $4000\ 0040_{hex}$. Note that the `and` and `or` instructions, which are prior to the `add`, still complete. Although not shown, the ALU overflow signal is an input to the control unit.

With five instructions active in any clock cycle, the challenge is to associate an exception with the appropriate instruction. Moreover, multiple exceptions can occur simultaneously in a single clock cycle. The normal solution is to prioritize the exceptions so that it is easy to determine which is serviced first; this strategy works for pipelined processors as well. In most MIPS implementations, the hardware sorts exceptions so that the earliest instruction is interrupted.

I/O device requests and hardware malfunctions are not associated with a specific instruction, so the implementation has some flexibility as to when to interrupt the pipeline. Hence, using the mechanism used for other exceptions works just fine.

The EPC captures the address of the interrupted instructions, and the MIPS Cause register records all possible exceptions in a clock cycle, so the exception software must match the exception to the instruction. An important clue is knowing in which pipeline stage a type of exception can occur. For example, an undefined instruction is discovered in the ID stage, and invoking the operating system occurs in the EX stage. Exceptions are collected in the Cause register so that the hardware can interrupt based on later exceptions, once the earliest one has been serviced.

Hardware Software Interface

The hardware and the operating system must work in conjunction so that exceptions behave as you would expect. The hardware contract is normally to stop the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address. The operating system contract is to look at the cause of the exception and act appropriately. For an undefined instruction, hardware failure, or arithmetic overflow exception, the operating system normally kills the program and returns an indicator of the reason. For an I/O device request or an operating system service call, the operating system saves the state of the program, performs the desired task, and, at some point in the future, restores the program to continue execution. In the case of I/O device requests, we may often choose to run another task before resuming the task that requested the I/O, since that task may often not be able to proceed until the I/O is complete. This is why the ability to save and restore the state of any task is critical. One of the most important and frequent uses of exceptions is handling page faults and TLB exceptions; Chapter 7 describes these exceptions and their handling in more detail.

imprecise interrupt Also called imprecise exception. Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.

precise interrupt Also called **precise exception**. An interrupt or exception that is alway associated with the correct instruction in pipelined computers.

The difficulty of always associating the correct exception with the correct instruction in pipelined computers has led some computer designers to relax this requirement in noncritical cases. Such processors are said to have **imprecise interrupts** or **imprecise exceptions**. In the example above, PC would normally have 58_{hex} at the start of the clock cycle after the exception is detected, even though the offending instruction is at address $4C_{hex}$. A processor with imprecise exceptions might put 58_{hex} into EPC and leave it up to the operating system to determine which instruction caused the problem. MIPS and the vast majority of computers today support **precise interrupts** or **precise exceptions**. (One reason is to support virtual memory, which we shall see in Chapter 7.)

Check Yourself

The MIPS designers wanted the integer multiply and divide instructions to operate in parallel with other integer instructions. Since multiply and divide take multiple clock cycles, a group of students is arguing over whether it is possible to implement precise exceptions. Which of the following arguments are completely accurate?

1. It is impossible to implement precise exceptions, since a multiply or divide can raise an exception after instructions that follow it.
2. It is trivial to implement precise exceptions since multiply and divide cannot raise an exception once they start, and so the timing of all exceptions is obviously precise.
3. It does not matter whether multiply or divide can raise an exception. The fact that they could still be executing and not completed when some other instruction raised an exception makes it impossible to implement precise exceptions.
4. Although it is true that a multiply or divide could still be executing, it is guaranteed to complete shortly, and when it does, any exception raised for an instruction following a multiply or divide will then be precise.

6.9 Advanced Pipelining: Extracting More Performance

Be forewarned that Sections 6.9 and 6.10 are brief overviews of fascinating but advanced topics. If you want to learn more details, you should consult our more

advanced book, *Computer Architecture: A Quantitative Approach*, third edition, where the material covered in the next 18 pages is expanded to over 200 pages!

Pipelining exploits the potential parallelism among instructions. This parallelism is called **instruction-level parallelism** (ILP). There are two primary methods for increasing the potential amount of instruction-level parallelism. The first is increasing the depth of the pipeline to overlap more instructions. Using our laundry analogy and assuming that the washer cycle were longer than the others, we could divide our washer into three machines that perform the wash, rinse, and spin steps of a traditional washer. We would then move from a four-stage to a six-stage pipeline. To get the full speedup, we need to rebalance the remaining steps so they are the same length, in processors or in laundry. The amount of parallelism being exploited is higher, since there are more operations being overlapped. Performance is potentially greater since the clock cycle can be shorter.

instruction-level parallelism The parallelism among instructions.

Another approach is to replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage. The general name for this technique is **multiple issue**. A multiple-issue laundry would replace our household washer and dryer with, say, three washers and three dryers. You would also have to recruit more assistants to fold and put away three times as much laundry in the same amount of time. The downside is the extra work to keep all the machines busy and transferring the loads to the next pipeline stage.

multiple issue A scheme whereby multiple instructions are launched in 1 clock cycle.

Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate or, stated alternatively, for the CPI to be less than 1. It is sometimes useful to flip the metric, and use *IPC*, or *instructions per clock cycle*, particularly as values become less than 1! Hence, a 6 GHz four-way multiple-issue microprocessor can execute a peak rate of 24 billion instructions per second and have a best case CPI of 0.25, or IPC of 4. Assuming a five-stage pipeline, such a processor would have 20 instructions in execution at any given time. Today's high-end microprocessors attempt to issue from three to eight instructions in every clock cycle. There are typically, however, many constraints on what types of instructions may be executed simultaneously and what happens when dependences arise.

static multiple issue An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.

There are two major ways to implement a multiple-issue processor, with the major difference being the division of work between the compiler and the hardware. Because the division of work dictates whether decisions are being made statically (that is, at compile time) or dynamically (that is, during execution), the approaches are sometimes called **static multiple issue** and **dynamic multiple issue**. As we will see, both approaches have other, more commonly used names, which may be less precise or more restrictive.

dynamic multiple issue An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.

There are two primary and distinct responsibilities that must be dealt with in a multiple-issue pipeline:

issue slots The positions from which instructions could issue in a given clock cycle; by analogy these correspond to positions at the starting blocks for a sprint.

1. Packaging instructions into issue slots: How does the processor determine how many instructions and which instructions can be issued in a given clock cycle? In most static issue processors, this process is at least partially handled by the compiler; in dynamic issue designs, it is normally dealt with at runtime by the processor, although the compiler will often have already tried to help improve the issue rate by placing the instructions in a beneficial order.
2. Dealing with data and control hazards: In static issue processors, some or all of the consequences of data and control hazards are handled statically by the compiler. In contrast, most dynamic issue processors attempt to alleviate at least some classes of hazards using hardware techniques operating at execution time.

Although we describe these as distinct approaches, in reality techniques from one approach are often borrowed by the other, and neither approach can claim to be perfectly pure.

The Concept of Speculation

speculation An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

One of the most important methods for finding and exploiting more ILP is speculation. Speculation is an approach that allows the compiler or the processor to "guess" about the properties of an instruction, so as to enable execution to begin for other instructions that may depend on the speculated instruction. For example, we might speculate on the outcome of a branch, so that instructions after the branch could be executed earlier. Or, we might speculate that a store that precedes a load does not refer to the same address, which would allow the load to be executed before the store. The difficulty with speculation is that it may be wrong. So, any speculation mechanism must include both a method to check if the guess was right and a method to unroll or back out the effects of the instructions that were executed speculatively. The implementation of this back-out capability adds complexity to any processor supporting speculation.

Speculation may be done in the compiler or by the hardware. For example, the compiler can use speculation to reorder instructions, moving an instruction across a branch or a load across a store. The processor hardware can perform the same transformation at runtime using techniques we discuss later in this section.

The recovery mechanisms used for incorrect speculation are rather different. In the case of speculation in software, the compiler usually inserts additional instructions that check the accuracy of the speculation and provide a fix-up routine to

use when the speculation was incorrect. In hardware speculation, the processor usually buffers the speculative results until it knows they are no longer speculative. If the speculation was correct, the instructions are completed by allowing the contents of the buffers to be written to the registers or memory. If the speculation was incorrect, the hardware flushes the buffers and reexecutes the correct instruction sequence.

Speculation introduces one other possible problem: speculating on certain instructions may introduce exceptions that were formerly not present. For example, suppose a load instruction is moved in a speculative manner, but the address it uses is not legal when the speculation is incorrect. The result would be that an exception that should not have occurred will occur. The problem is complicated by the fact that if the load instruction were not speculative, then the exception must occur! In compiler-based speculation, such problems are avoided by adding special speculation support that allows such exceptions to be ignored until it is clear that they really should occur. In hardware-based speculation, exceptions are simply buffered until it is clear that the instruction causing them is no longer speculative and is ready to complete; at that point the exception is raised, and normal exception handling proceeds.

Since speculation can improve performance when done properly and decrease performance when done carelessly, significant effort goes into deciding when it is appropriate to speculate. Later in this section, we will examine both static and dynamic techniques for speculation.

Static Multiple Issue

Static multiple-issue processors all use the compiler to assist with packaging instructions and handling hazards. In a static issue processor, you can think of the set of instructions that issue in a given clock cycle, which is called an **issue packet**, as one large instruction with multiple operations. This view is more than an analogy. Since a static multiple-issue processor usually restricts what mix of instructions can be initiated in a given clock cycle, it is useful to think of the issue packet as a single instruction allowing several operations in certain predefined fields. This view led to the original name for this approach: Very Long Instruction Word (VLIW). The Intel IA-64 architecture uses this approach, which it calls by its own name: Explicitly Parallel Instruction Computer (EPIC). The Itanium and Itanium 2 processors, available in 2000 and 2002, respectively, are the first implementations of the IA-64 architecture.

issue packet The set of instructions that issues together in 1 clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

Most static issue processors also rely on the compiler to take on some responsibility for handling data and control hazards. The compiler's responsibilities may include static branch prediction and code scheduling to reduce or prevent all hazards.

Let's look at a simple static issue version of a MIPS processor, before we describe the use of these techniques in more aggressive processors. After using this simple example to review the comments, we discuss the highlights of the Intel IA-64 architecture.

An Example: Static Multiple Issue with the MIPS ISA

To give a flavor of static multiple issue, we consider a simple two-issue MIPS processor, where one of the instructions can be an integer ALU operation or branch, and the other can be a load or store. Such a design is like that used in some embedded MIPS processors. Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions. In many static multiple-issue processors, and essentially all VLIW processors, the layout of simultaneously issuing instructions is restricted to simplify the decoding and instruction issue. Hence, we will require that the instructions be paired and aligned on a 64-bit boundary, with the ALU or branch portion appearing first. Furthermore, if one instruction of the pair cannot be used, we require that it be replaced with a no-op. Thus, the instructions always issue in pairs, possibly with a nop in one slot. Figure 6.44 shows how the instructions look as they go into the pipeline in pairs.

Static multiple-issue processors vary in how they deal with potential data and control hazards. In some designs, the compiler takes full responsibility for removing *all* hazards, scheduling the code and inserting no-ops so that the code executes without any need for hazard detection or hardware-generated stalls. In others, the hardware detects data hazards and generates stalls between two issue packets, while requiring that the compiler avoid all dependences within an instruction pair. Even so, a hazard generally forces the entire issue packet containing the dependent instruction to stall. Whether the software must handle all hazards or

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

FIGURE 6.44 Static two-issue pipeline in operation. The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

only try to reduce the fraction of hazards between separate issue packets, the appearance of having a large single instruction with multiple operations is reinforced. We will assume the second approach for this example.

To issue an ALU and a data transfer operation in parallel, the first need for additional hardware—beyond the usual hazard detection and stall logic—is extra ports in the register file (see Figure 6.45). In 1 clock cycle we may need to read two registers for the ALU operation and two more for a store, and also one write port for an ALU operation and one write port for a load. Since the ALU is tied up for the ALU operation, we also need a separate adder to calculate the effective address for data transfers. Without these extra resources, our two-issue pipeline would be hindered by structural hazards.

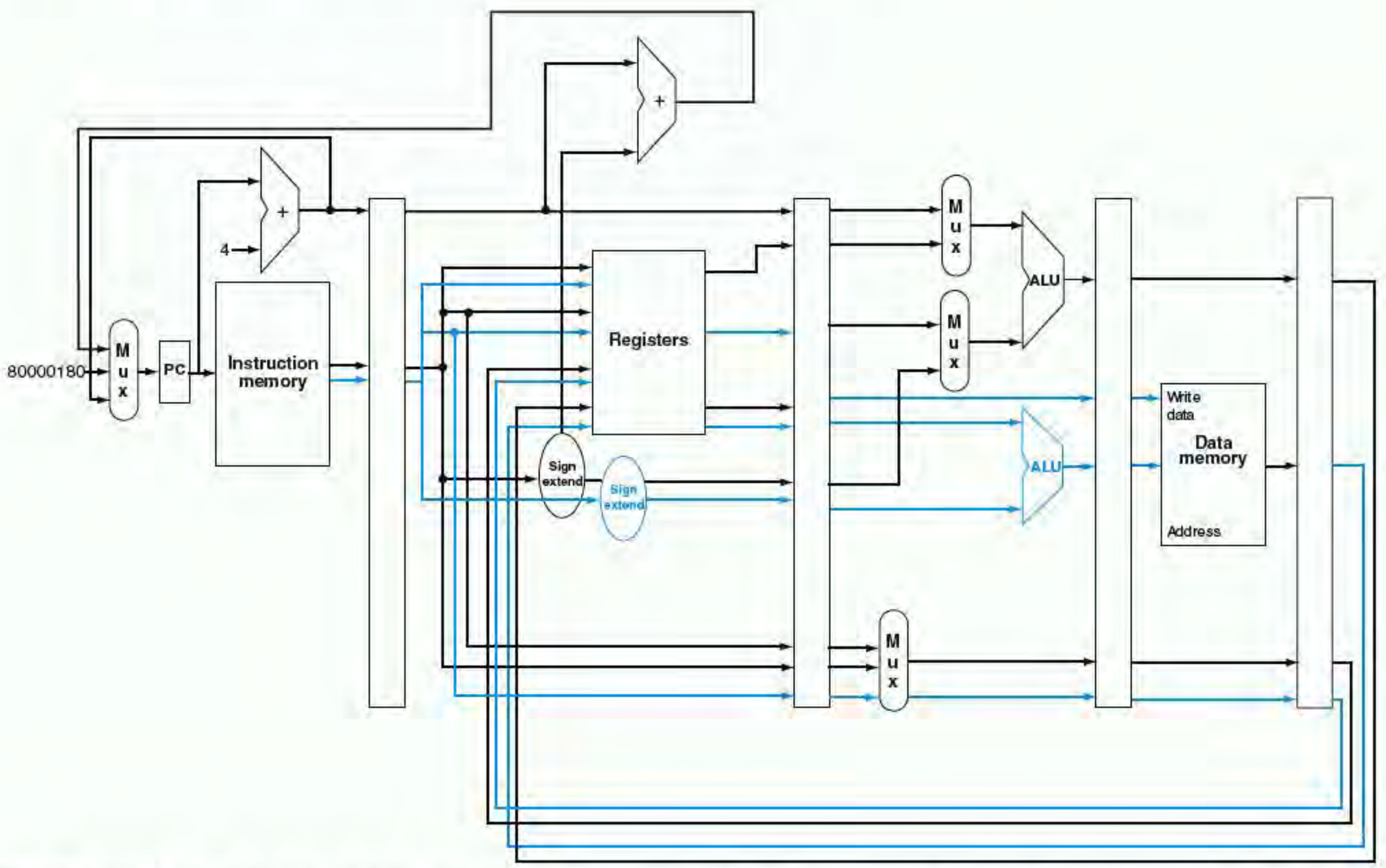


FIGURE 6.45 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

Clearly, this two-issue processor can improve performance by up to a factor of 2. Doing so, however, requires that twice as many instructions be overlapped in execution, and this additional overlap increases the relative performance loss from data and control hazards. For example, in our simple five-stage pipeline, loads have a use latency of 1 clock cycle, which prevents one instruction from using the result without stalling. In the two-issue, five-stage pipeline, the result of a load instruction cannot be used on the next *clock cycle*. This means that the next *two* instructions cannot use the load result without stalling. Furthermore, ALU instructions that had no use latency in the simple five-stage pipeline, now have a one-instruction use latency, since the results cannot be used in the paired load or store. To effectively exploit the parallelism available in a multiple-issue processor, more ambitious compiler or hardware scheduling techniques are needed, and static multiple issue requires that the compiler takes on this role.

EXAMPLE

Simple Multiple-Issue Code Scheduling

How would this loop be scheduled on a static two-issue pipeline for MIPS?

```
Loop:   lw     $t0, 0($s1)     # $t0=array element
        addu   $t0,$t0,$s2     # add scalar in $s2
        sw     $t0, 0($s1)     # store result
        addi   $s1,$s1,-4      # decrement pointer
        bne    $s1,$zero,Loop  # branch $s1!=0
```

Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

ANSWER

The first three instructions have data dependences, and so do the last two. Figure 6.46 shows the best schedule for these instructions. Notice that just one pair of instructions has both issue slots used. It takes 4 clocks per loop iteration; at 4 clocks to execute 5 instructions, we get the disappointing CPI of 0.8 versus the best case of 0.5., or an IPC of 1.25 versus 2.0. Notice that in computing CPI or IPC, we do not count any nops executed as useful instructions. Doing so would improve CPI, but not performance!

loop unrolling A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

An important compiler technique to get more performance from loops is **loop unrolling**, a technique where multiple copies of the loop body are made. After unrolling, there is more ILP available by overlapping instructions from different iterations.

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		`lw $t0, 0($s1)`	1
	`addi $s1,$s1,-4`		2
	`addu $t0,$t0,$s2`		3
	`bne $s1,$zero,Loop`	`sw $t0, 4($s1)`	4

FIGURE 6.46 The scheduled code as it would look on a two-issue MIPS pipeline. The empty slots are nops.

Loop Unrolling for Multiple-Issue Pipelines

EXAMPLE

See how well loop unrolling and scheduling work in the example above. Assume that the loop index is a multiple of four, for simplicity.

ANSWER

To schedule the loop without any delays, it turns out that we need to make four copies of the loop body. After unrolling and eliminating the unnecessary loop overhead instructions, the loop will contain four copies each of `lw`, `add`, and `sw`, plus one `addi` and one `bne`. Figure 6.47 shows the unrolled and scheduled code.

During the unrolling process, the compiler introduced additional registers (`$t1`, `$t2`, `$t3`). The goal of this process, called **register renaming**, is to eliminate dependences that are not true data dependences, but could either lead to potential hazards or prevent the compiler from flexibly scheduling the code. Consider how the unrolled code would look using only `$t0`. There would be repeated instances of `lw $t0,0($$s1)`, `addu $t0,$t0,$s2` followed by `sw t0,4($s1)`, but these sequences, despite using `$t0`, are actually completely independent—no data values flow between one pair of these instructions and the next pair. This is what is called an **antidependence** or **name dependence**, which is an ordering forced purely by the reuse of a name, rather than a real data dependence.

register renaming The renaming of registers, by the compiler or hardware, to remove antidependences.

antidependence Also called **name dependence**. An ordering forced by the reuse of a name, typically a register, rather then by a true dependence that carries a value between two

Renaming the registers during the unrolling process allows the compiler to subsequently move these independent instructions so as to better schedule the code. The renaming process eliminates the name dependences, while preserving the true dependences.

Notice now that 12 of the 14 instructions in the loop execute as a pair. It takes 8 clocks for four loop iterations, or 2 clocks per iteration, which yields a CPI of 8/14 = 0.57. Loop unrolling and scheduling with dual issue gave us a factor of two improvement, partly from reducing the loop control instructions and partly from dual issue execution. The cost of this performance improvement is using four temporary registers rather than one, as well as a significant increase in code size.

	ALU or branch instruction		Data transfer instruction		Clock cycle
Loop:	addi	$s1,$s1,-16	lw	$t0, 0($s1)	1
			lw	$t1,12($s1)	2
	addu	$t0,$t0,$s2	lw	$t2, 8($s1)	3
	addu	$t1,$t1,$s2	lw	$t3, 4($s1)	4
	addu	$t2,$t2,$s2	sw	$t0, 16($s1)	5
	addu	$t3,$t3,$s2	sw	$t1,12($s1)	6
			sw	$t2, 8($s1)	7
	bne	$s1,$zero,Loop	sw	$t3, 4($s1)	8

FIGURE 6.47 The unrolled and scheduled code of Figure 6.46 as it would look on a static two-issue MIPS pipeline. The empty slots are nops. Since the first instruction in the loop decrements $s1 by 16, the addresses loaded are the original value of $s1, then that address minus 4, minus 8, and minus 12.

The Intel IA-64 Architecture

The IA-64 architecture is a register-register, RISC-style instruction set like the 64-bit version of the MIPS architecture (called MIPS-64), but with several unique features to support explicit, compiler-driven exploitation of ILP. Intel calls the approach EPIC (Explicitly Parallel Instruction Computer). The major differences between IA-64 and the MIPS architecture are the following:

1. IA-64 has many more registers than MIPS, including 128 integer and 128 floating-point registers, as well as 8 special registers for branches and 64 1-bit condition registers. In addition, IA-64 supports register windows in a fashion similar to the original Berkeley RISC and Sun SPARC architectures.
2. IA-64 places instructions into bundles that have a fixed format and explicit designation of dependences.
3. IA-64 includes special instructions and capabilities for speculation and for branch elimination, which increase the amount of ILP that can be exploited.

The IA-64 architecture is designed to achieve the major benefits of a VLIW—implicit parallelism among operations in an instruction and fixed formatting of the operation fields—while maintaining greater flexibility than a VLIW normally allows. The IA-64 architecture uses two different concepts to achieve this flexibility: instruction groups and bundles.

instruction group In IA-64, a sequence of consecutive instructions with no register data dependences among them.

stop In IA-64, an explicit indicator of a break between independent and dependent instructions.

An **instruction group** is a sequence of consecutive instructions with no register data dependences among them. All the instructions in a group could be executed in parallel if sufficient hardware resources existed and if any dependences through memory were preserved. An instruction group can be arbitrarily long, but the compiler must *explicitly* indicate the boundary between one instruction group and another. This boundary is indicated by placing a **stop** between two instructions that belong to different groups.

IA-64 instructions are encoded in *bundles,* which are 128 bits wide. Each bundle consists of a 5-bit template field and three instructions, each 41 bits in length. To simplify the decoding and instruction issue process, the template field of a bundle specifies which of five different execution units each instruction in the bundle requires. The five different execution units are integer ALU, noninteger ALU (includes shifters and multimedia operations), memory unit, floating-point unit, and branch unit.

The 5-bit template field within each bundle describes *both* the presence of any stops associated with the bundle and the execution unit type required by each instruction within the bundle. The bundle formats can specify only a subset of all possible combinations of instruction types and stops.

To enhance the amount of ILP that can be exploited, IA-64 provides extensive support for predication and for speculation (see the Elaboration on page 442). **Predication** is a technique that can be used to eliminate branches by making the execution of an instruction dependent on a predicate, rather than dependent on a branch. As we saw earlier, branches reduce the opportunity to exploit ILP by restricting the movement of code. Loop unrolling works well to eliminate loop branches, but a branch within a loop—arising, for example, from an *if-then-else* statement—cannot be eliminated by loop unrolling. Predication, however, provides a method to eliminate the branch, allowing more flexible exploitation of parallelism.

predication A technique to make instructions dependent on predicates rather than on branches.

For example, suppose we had a code sequence like

```
if (p) {statement 1} else {statement 2}
```

Using normal compilation methods, this segment would compile using two branches: one after the condition branching to the else portion and one after statement 1 branching to the next sequential statement. With predication, it could be compiled as

```
(p) statement 1
(~p) statement 2
```

where the use of `(condition)` indicates that the statement is executed only if `condition` is true, and otherwise becomes a no-op. Notice that predication can be used as way to speculate, as well as a method to eliminate branches.

The IA-64 architecture provides comprehensive support for predication: nearly every instruction in the IA-64 architecture can be predicated by specifying a predicate register, whose identity is placed in the lower 6 bits of an instruction field. One consequence of full predication is that a conditional branch is simply a branch with a guarding predicate!

IA-64 is the most sophisticated example of an instruction set with support for compiler-based exploitation of ILP. Intel's Itanium and Itanium 2 processors implement this architecture. A brief summary of the characteristics of these processors is given in Figure 6.48.

Processor	Maximum instr. issues / clock	Functional units	Maximum ops. per clock	Max. clock rate	Transistors (millions)	Power (watts)	SPEC int2000	SPEC fp2000
Itanium	6	4 integer/media 2 memory 3 branch 2 FP	9	0.8 GHz	25	130	379	701
Itanium 2	6	6 integer/media 4 memory 3 branch 2 FP	11	1.5 Ghz	221	130	810	1427

FIGURE 6.48 A summary of the characteristics of the Itanium and Itanium 2, Intel's first two implementations of the IA-64 architecture. In addition to higher clock rates and more functional units, the Itanium 2 includes an on-chip level 3 cache, versus an off-chip level 3 cache in the Itanium.

Elaboration: Speculation support in the IA-64 architecture consists of separate support for control speculation, which deals with deferring exceptions for speculated instructions, and memory reference speculation, which supports speculation of load instructions. Deferred exception handling is supported by adding speculative load instructions, which, when an exception occurs, tag the result as poison. When a poisoned result is used by an instruction, the result is also poison. The software can then check for a poisoned result when it knows that the execution is no longer speculative.

poison A result generated when a speculative load yields an exception, or an instruction uses a poisoned operand.

In IA-64, we can also speculate on memory references by moving loads earlier than stores on which they may depend. This is done with an advanced load instruction. An advanced load executes normally, but uses a special table to track the address that the processor loaded from. All subsequent stores check that table and generate a flag in the entry if the store address matches the load address. A subsequent instruction must be used to check the status of the entry after the load is no longer speculative. If a store to the same address has intervened, the check instruction specifies a fix-up routine that reexecutes the load and any other dependent instructions before continuing execution; if no such store has occurred, the table entry is simply cleared, indicating that the load is no longer speculative.

advanced load In IA-64, a speculative load instruction with support to check for aliases that could invalidate the load.

Dynamic Multiple-Issue Processors

superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle.

Dynamic multiple-issue processors are also known as superscalar processors, or simply superscalars. In the simplest superscalar processors, instructions issue in-order, and the processor decides whether zero, one, or more instructions can issue in a given clock cycle. Obviously, achieving good performance on such a processor still requires the compiler to try to schedule instructions to move dependences apart and thereby improve the instruction issue rate. Even with such compiler scheduling, there is an important difference between this simple superscalar and a VLIW processor: the code, whether scheduled or not, is guaranteed by the hardware to execute correctly. Furthermore, compiled code will always run correctly

independent of the issue rate or pipeline structure of the processor. In some VLIW designs, this has not been the case, and recompilation was required when moving across different processor models; in other static issue processors, code would run correctly across different implementations, but often so poorly as to make compilation effectively required.

Many superscalars extend the basic framework of dynamic issue decisions to include **dynamic pipeline scheduling**. Dynamic pipeline scheduling chooses which instructions to execute in a given clock cycle while trying to avoid hazards and stalls. Let's start with a simple example of avoiding a data hazard. Consider the following code sequence:

dynamic pipeline scheduling Hardware support for reordering the order of instruction execution so as to avoid stalls.

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

Even though the `sub` instruction is ready to execute, it must wait for the `lw` and `addu` to complete first, which might take many clock cycles if memory is slow. (Chapter 7 explains caches, the reason that memory accesses are sometimes very slow.) Dynamic pipeline scheduling allows such hazards to be avoided either fully or partially.

Dynamic Pipeline Scheduling

Dynamic pipeline scheduling chooses which instructions to execute next, possibly reordering them to avoid stalls. In such processors, the pipeline is divided into three major units: an instruction fetch and issue unit, multiple functional units (10 or more in high-end designs in 2004), and a **commit unit**. Figure 6.49 shows the model. The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution. Each functional unit has buffers, called **reservation stations**, that hold the operands and the operation. (In the next section, we will discuss an alternative to reservation stations used by many recent processors.) As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated. When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit, which buffers the result until it is safe to put the result into the register file or, for a store, into memory. The buffer in the commit unit, often called the **reorder buffer**, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline. Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.

commit unit The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer-visible registers and memory.

reservation station A buffer within a functional unit that holds the operands and the operation.

reorder buffer The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register.

The combination of buffering operands in the reservation stations and results in the reorder buffer provides a form of register renaming, just like that used by

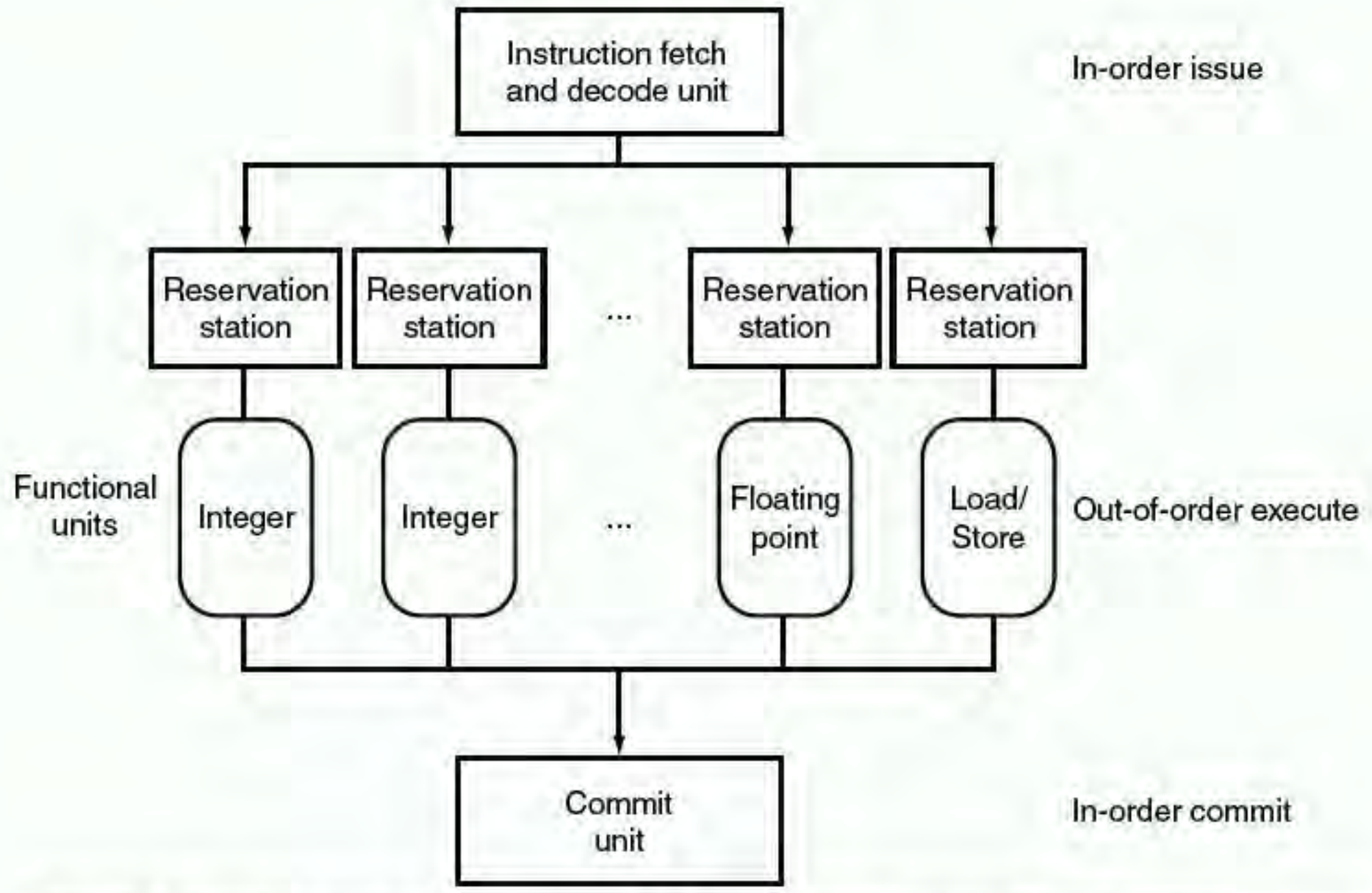


FIGURE 6.49 The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation.

the compiler in our earlier loop unrolling example on page 439. To see how this conceptually works, consider the following steps:

1. When an instruction issues, if either of its operands is in the register file or the reorder buffer, it is copied to the reservation station immediately, where it is buffered until all the operands and an execution unit are available. For the issuing instruction, the register copy of the operand is no longer required, and if a write to that register occurred, the value could be overwritten.
2. If an operand is not in the register file or reorder buffer, it must be waiting to be produced by a functional unit. The name of the functional unit that will produce the result is tracked. When that unit eventually produces the result, it is copied directly into the waiting reservation station from the functional unit bypassing the registers.

These steps effectively use the reorder buffer and the reservation stations to implement register renaming.

Conceptually, you can think of a dynamically scheduled pipeline as analyzing the dataflow structure of a program, as we saw when we discussed dataflow analysis within a compiler in Chapter 2. The processor then executes the instructions in some order that preserves the data flow order of the program. To make programs behave as if they were running on a simple in-order pipeline, the instruction fetch and decode unit is required to issue instructions in order, which allows dependences to be tracked, and the commit unit is required to write results to registers and memory in program execution order. This conservative mode is called in-order completion. Hence, if an exception occurs, the computer can point to the last instruction executed, and the only registers updated will be those written by instructions before the instruction causing the exception. Although, the front end (fetch and issue) and the back end (commit) of the pipeline run in order, the functional units are free to initiate execution whenever the data they need is available. Today, all dynamically scheduled pipelines use in-order completion, although this was not always true.

Dynamic scheduling is often extended by including hardware-based speculation, especially for branch outcomes. By predicting the direction of a branch, a dynamically scheduled processor can continue to fetch and execute instructions along the predicted path. Because the instructions are committed in order, we know whether or not the branch was correctly predicted before any instructions from the predicted path are committed. A speculative, dynamically scheduled pipeline can also support speculation on load addresses, allowing load-store reordering, and using the commit unit to avoid incorrect speculation. In the next section we will look at the use of dynamic scheduling with speculation in the Pentium 4 design.

in-order commit A commit in which the results of pipelined execution are written to the programmer-visible state in the same order that instructions are fetched.

Elaboration: A commit unit controls updates to the register file *and* memory. Some dynamically scheduled processors update the register file immediately during execution using extra registers to implement the renaming function and preserving the older copy of a register until the instruction updating the register is no longer speculative. Other processors buffer the result, typically in a structure called a reorder buffer, and the actual update to the register file occurs later as part of the commit. Stores to memory must be buffered until commit time either in a *store buffer* (see Chapter 7) or in the reorder buffer. The commit unit allows the store to write to memory from the buffer when the buffer has a valid address and valid data, and when the store is no longer dependent on predicted branches.

out-of-order execution A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait.

Elaboration: Memory accesses benefit from *nonblocking caches*, which continue servicing cache accesses during a cache miss (see Chapter 7). Out-of-order execution processors need nonblocking caches to allow instructions to execute during a miss.

Hardware Software Interface

Given that compilers can also schedule code around data dependences, you might ask, Why would a superscalar processor use dynamic scheduling? There are three major reasons. First, not all stalls are predictable. In particular, cache misses (see Chapter 7) cause unpredictable stalls. Dynamic scheduling allows the processor to hide some of those stalls by continuing to execute instructions while waiting for the stall to end.

Second, if the processor speculates on branch outcomes using dynamic branch prediction, it cannot know the exact order of instructions at compile time, since it depends on the predicted and actual behavior of branches. Incorporating dynamic speculation to exploit more ILP without incorporating dynamic scheduling would significantly restrict the benefits of such speculation.

Third, as the pipeline latency and issue width change from one implementation to another, the best way to compile a code sequence also changes. For example, how to schedule a sequence of dependent instructions is affected by both issue width and latency. The pipeline structure affects both the number of times a loop must be unrolled to avoid stalls as well as the process of compiler-based register renaming. Dynamic scheduling allows the hardware to hide most of these details. Thus, users and software distributors do not need to worry about having multiple versions of a program for different implementations of the same instruction set. Similarly, old legacy code will get much of the benefit of a new implementation without the need for recompilation.

The BIG Picture

Both pipelining and multiple-issue execution increase peak instruction throughput and attempt to exploit ILP. Data and control dependences in programs, however, offer an upper limit on sustained performance because the processor must sometimes wait for a dependence to be resolved. Software-centric approaches to exploiting ILP rely on the ability of the compiler to find and reduce the effects of such dependences, while hardware-centric approaches rely on extensions to the pipeline and issue mechanisms. Speculation, performed by the compiler or the hardware, can increase the amount of ILP that can be exploited, although care must be taken since speculating incorrectly is likely to reduce performance.

Understanding Program Performance

Modern, high-performance microprocessors are capable of issuing several instructions per clock; unfortunately, sustaining that issue rate is very difficult. For example, despite the existence of processors with four to six issues per clock, very few applications can sustain more than two instructions per clock. There are two primary reasons for this.

First, within the pipeline, the major performance bottlenecks arise from dependences that cannot be alleviated, thus reducing the parallelism among instructions and the sustained issue rate. Although little can be done about true data dependences, often the compiler or hardware does not know precisely whether a dependence exists or not, and so must conservatively assume the dependence exists. For example, code that makes use of pointers, particularly in ways that create more aliasing, will lead to more implied potential dependences. In contrast, the greater regularity of array accesses often allows a compiler to deduce that no dependences exist. Similarly, branches that cannot be accurately predicted whether at runtime or compile time will limit the ability to exploit ILP. Often additional ILP is available, but the ability of the compiler or the hardware to find ILP that may be widely separated (sometimes by the execution of thousands of instructions) is limited.

Second, losses in the memory system (the topic of Chapter 7) also limit the ability to keep the pipeline full. Some memory system stalls can be hidden, but limited amounts of ILP also limit the extent to which such stalls can be hidden.

Check Yourself

State whether the following techniques or components are associated primarily with a software- or hardware-based approach to exploiting ILP. In some cases, the answer may be both.

1. Branch prediction
2. Multiple issue
3. VLIW
4. Superscalar
5. Dynamic scheduling
6. Out-of-order execution
7. Speculation
8. EPIC
9. Reorder buffer
10. Register renaming
11. Predication

6.10 Real Stuff: The Pentium 4 Pipeline

In the last chapter, we discussed how the Pentium 4 fetched and translated IA-32 instructions into microoperations. The microoperations are then executed by a sophisticated, dynamically scheduled, speculative pipeline capable of sustaining an execution rate of three microoperations per clock cycle. This section focuses on that microoperation pipeline. The Pentium 4 combines multiple issue with deep pipelining so as to achieve both a low CPI and a high clock rate.

When we consider the design of sophisticated, dynamically scheduled processors, the design of the functional units, the cache and register file, instruction issue, and overall pipeline control become intermingled, making it difficult to separate out the datapath from the pipeline. Because of this, many engineers and researchers have adopted the term **microarchitecture** to refer to the detailed internal architecture of a processor. Figure 6.50 shows the microarchitecture of the Pentium 4, focusing on the structures for executing the microoperations.

microarchitecture The organization of the processor, including the major functional units, their interconnection, and control.

Another way to look at the Pentium 4 is to see the pipeline stages that a typical instruction goes through. Figure 6.51 shows the pipeline structure and the typical number of clock cycles spent in each; of course, the number of clock cycles varies due to the nature of dynamic scheduling as well as the requirements of individual microoperations.

The Pentium 4, and the earlier Pentium III and Pentium Pro, all use the technique of decoding IA-32 instructions into microoperations and executing those microoperations using a speculative pipeline with multiple functional units. In fact, the basic microarchitecture is similar, and all these processors can complete up to three microoperations per cycle. The Pentium 4 gains its performance advantage over the Pentium III through several enhancements:

1. A pipeline that is roughly twice as deep (approximately 20 cycles versus 10) and can run almost twice as fast in the same technology
2. More functional units (7 versus 5)
3. Support for a larger number of outstanding operations (126 versus 40)
4. The use of a trace cache (see Chapter 7) and a much better branch predictor (4K entries versus 512)
5. Other enhancements to the memory system, which we discuss in Chapter 7

architectural registers The instruction set visible registers of a processor; for example, in MIPS, these are the 32 integer and 16 floating-point registers.

Elaboration: The Pentium 4 uses a scheme for resolving antidependences and incorrect speculation that uses a reorder buffer together with register renaming. Register renaming explicitly renames the architectural registers in a processor (8 in the case of IA-32) to a larger set of physical registers (128 in the Pentium 4). The Pentium 4 uses

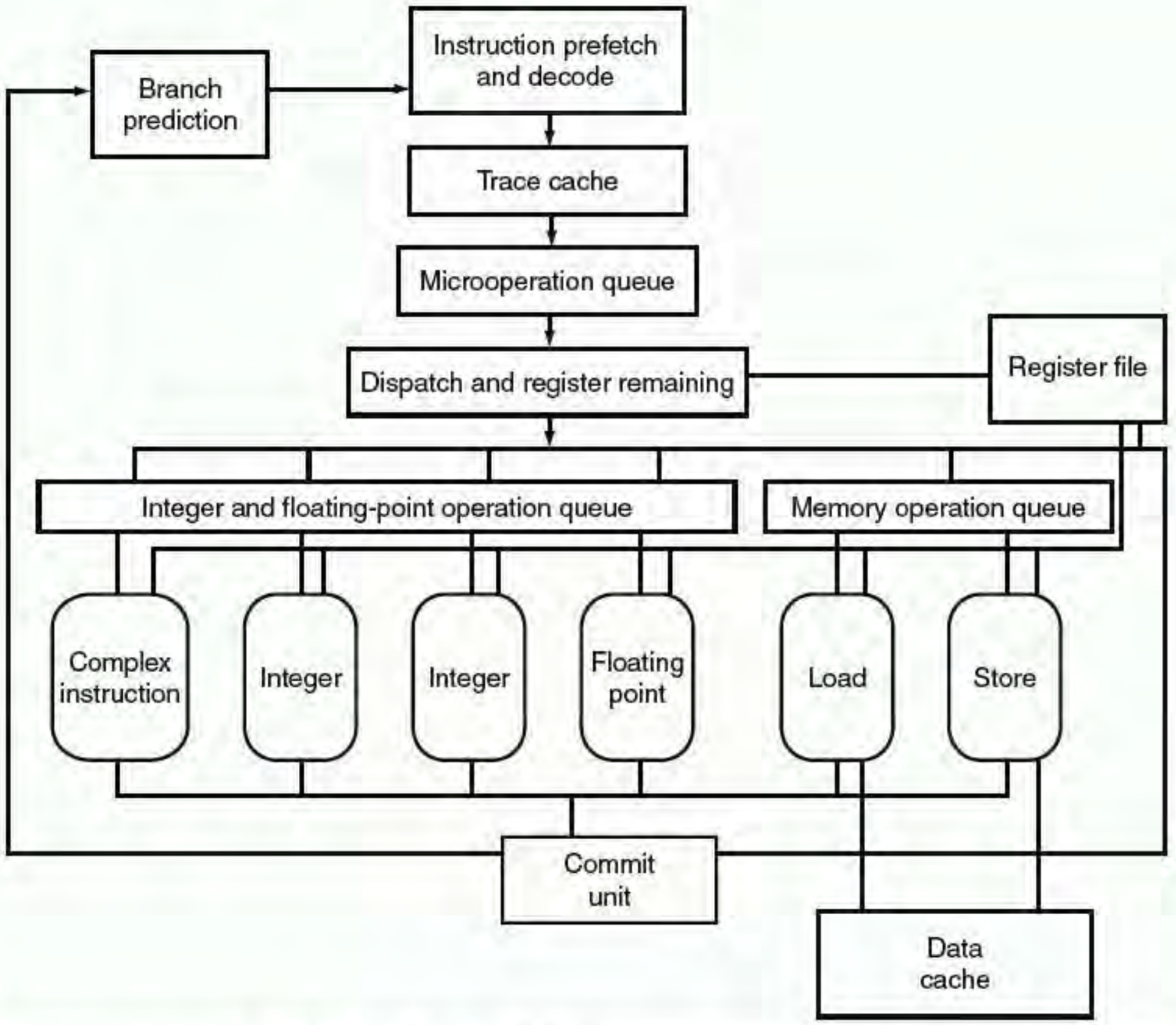


FIGURE 6.50 The microarchitecture of the Intel Pentium 4. The extensive queues allow up to 126 microoperations to be outstanding at any point in time, including 48 loads and 24 stores. There are actually seven functional units, since the FP unit includes a separate dedicated unit for floating-point moves. The load and store units are actually separated into two parts, with the first part handling address calculation and the second part responsible for the actual memory reference. The integer ALUs operate at twice the clock frequency, allowing two integer ALU operations to be completed by each of the two integer units in a single clock cycle. As we described in Chapter 5, the Pentium 4 uses a special cache, called the trace cache, to hold predecoded sequences of microoperations, corresponding to IA-32 instructions. The operation of a trace cache is explained in more detail in Chapter 7. The FP unit also handles the MMX multimedia and SSE2 instructions. There is an extensive bypass network among the functional units; since the pipeline is dynamic rather than static, bypassing is done by tagging results and tracking source operands, so as to allow a match when a result is produced for an instruction in one of the queues that needs the result. Intel is expected to release new versions of the Pentium 4 in 2004, which will probably have changes in the microarchitecture.

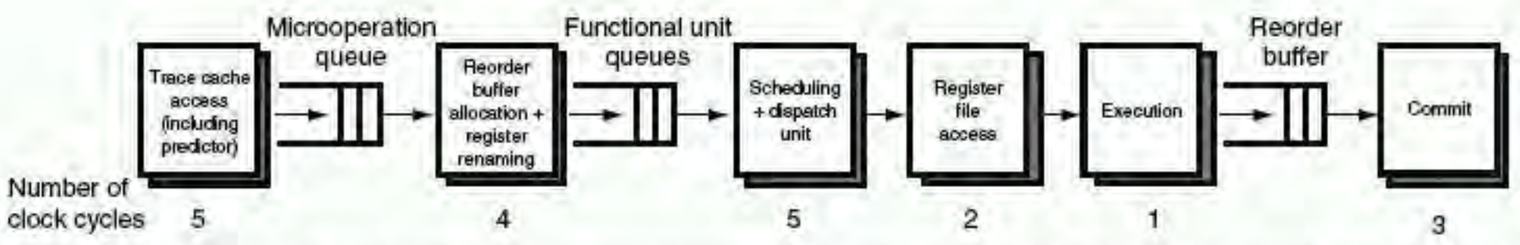


FIGURE 6.51 The Pentium 4 pipeline showing the pipeline flow for a typical instruction and the number of clock cycles for the major steps in the pipeline. The major buffers where instructions wait are also shown.

register renaming to remove antidependences. Register renaming requires the processor to maintain a map between the architectural registers and the physical registers, indicating which physical register is the most current copy of an architectural register. By keeping track of the renamings that have occurred, register renaming offers another approach to recovery in the event of incorrect speculation: simply undo the mappings that have occurred since the first incorrectly speculated instruction. This will cause the state of the processor to return to the last correctly executed instruction, keeping the correct mapping between the architectural and physical registers.

Understanding Program Performance

The Pentium 4 combines a deep pipeline (averaging 20 or more pipe stages per instruction) and aggressive multiple issue to achieve high performance. By keeping the latencies for back-to-back operations low (0 for ALU operations and 2 for loads), the impact of data dependences is reduced. What are the most serious potential performance bottlenecks for programs running on this processor? The following list includes some potential performance problems, the last three of which can apply in some form to any high-performance pipelined processor.

- The use of IA-32 instructions that do not map to three or fewer simple microoperations
- Branches that are difficult to predict, causing misprediction stalls and restarts when speculation fails
- Poor instruction locality, which causes the trace cache not to function effectively
- Long dependences—typically caused by long-running instructions or data cache misses—which lead to stalls

Performance delays arising in accessing memory (see Chapter 7) that cause the processor to stall

Check Yourself

Are the following statements true or false?

1. The Pentium 4 can issue more instructions per clock then the Pentium III.
2. The Pentium 4 multiple-issue pipeline directly executes IA-32 instructions.
3. The Pentium 4 uses dynamic scheduling but no speculation.
4. The Pentium 4 microarchitecture has many more registers than IA-32 requires.
5. The Pentium 4 pipeline has fewer stages than the Pentium III.
6. The trace cache in the Pentium 4 is exactly the same as an instruction cache.

6.11 Fallacies and Pitfalls

Fallacy: Pipelining is easy.

Our books testify to the subtlety of correct pipeline execution. Our advanced book had a pipeline bug in its first edition, despite its being reviewed by more than 100 people and being class-tested at 18 universities. The bug was uncovered only when someone tried to build the computer in that book. The fact that the Verilog to describe a pipeline like that in the Pentium 4 will be thousands of lines is an indication of the complexity. Beware!

Fallacy: Pipelining ideas can be implemented independent of technology.

When the number of transistors on-chip and speed of transistors made a five-stage pipeline the best solution, then the delayed branch (see the Elaboration on page 423) was a simple solution to control hazards. With longer pipelines, superscalar execution, and dynamic branch prediction, it is now redundant. In the early 1990s, dynamic pipeline scheduling took too many resources and was not required for high performance, but as transistor budgets continued to double and logic became much faster than memory, then multiple functional units and dynamic pipelining made more sense. Today, all high-end processors use multiple issue, and most choose to implement aggressive speculation as well.

Pitfall: Failure to consider instruction set design can adversely impact pipelining.

Many of the difficulties of pipelining arise because of instruction set complications. Here are some examples:

- Widely variable instruction lengths and running times can lead to imbalance among pipeline stages and severely complicate hazard detection in a design pipelined at the instruction set level. This problem was overcome, initially in the DEC VAX 8500 in the late 1980s, using the micropipelined scheme that the Pentium 4 employs today. Of course, the overhead of translation and maintaining correspondence between the microoperations and the actual instructions remains.
- Sophisticated addressing modes can lead to different sorts of problems. Addressing modes that update registers, such as update addressing (see Chapter 3), complicate hazard detection. Other addressing modes that require multiple memory accesses substantially complicate pipeline control and make it difficult to keep the pipeline flowing smoothly.

Perhaps the best example is the DEC Alpha and the DEC NVAX. In comparable technology, the newer instruction set architecture of the Alpha allowed an implementation whose performance is more than twice as fast as NVAX. In another

example, Bhandarkar and Clark [1991] compared the MIPS M/2000 and the DEC VAX 8700 by counting clock cycles of the SPEC benchmarks; they concluded that, although the MIPS M/2000 executes more instructions, the VAX on average executes 2.7 times as many clock cycles, so the MIPS is faster.

Nine-tenths of wisdom consists of being wise in time.
American proverb

6.12 Concluding Remarks

Pipelining improves the average execution time per instruction. Depending on whether you start with a single-cycle or multiple-cycle datapath, this reduction can be thought of as decreasing the clock cycle time or as decreasing the number of clock cycles per instruction (CPI). We started with the simple single-cycle datapath, so pipelining was presented as reducing the clock cycle time of the simple datapath. Multiple issue, in comparison, clearly focuses on reducing CPI (or increasing IPC). Figure 6.52 shows the effect on CPI and clock rate for each of the microarchitectures from Chapters 5 and 6. Performance is increased by moving up and to the right, since it is the product of IPC and clock rate that determines performance for a given instruction set.

Pipelining improves throughput, but not the inherent execution time, or *latency*, of instructions; the latency is similar in length to the multicycle approach. Unlike that approach, which uses the same hardware repeatedly during instruction execution, pipelining starts an instruction every clock cycle by having dedicated hardware. Similarly, multiple issue adds additional datapath hardware to allow multiple instructions to begin every clock cycle, but at an increase in effective latency. Figure 6.53 shows the datapaths from Figure 6.52 placed according to the amount of sharing of hardware and **instruction latency**.

instruction latency The inherent execution time for an instruction.

Pipelining and multiple issue both attempt to exploit instruction-level parallelism. The presence of data and control dependences, which can become hazards, are the primary limitations on how much parallelism can be exploited. Scheduling and speculation, both in hardware and software, are the primary techniques used to reduce the performance impact of dependences.

The switch to longer pipelines, multiple instruction issue, and dynamic scheduling in the mid-1990s has helped sustain the 60% per year processor performance increase that we have benefited from since the early 1980s. In the past, it appeared that the choice was between the highest clock rate processors and the most sophisticated superscalar processors. As we have seen, the Pentium 4 combines both and achieves remarkable performance.

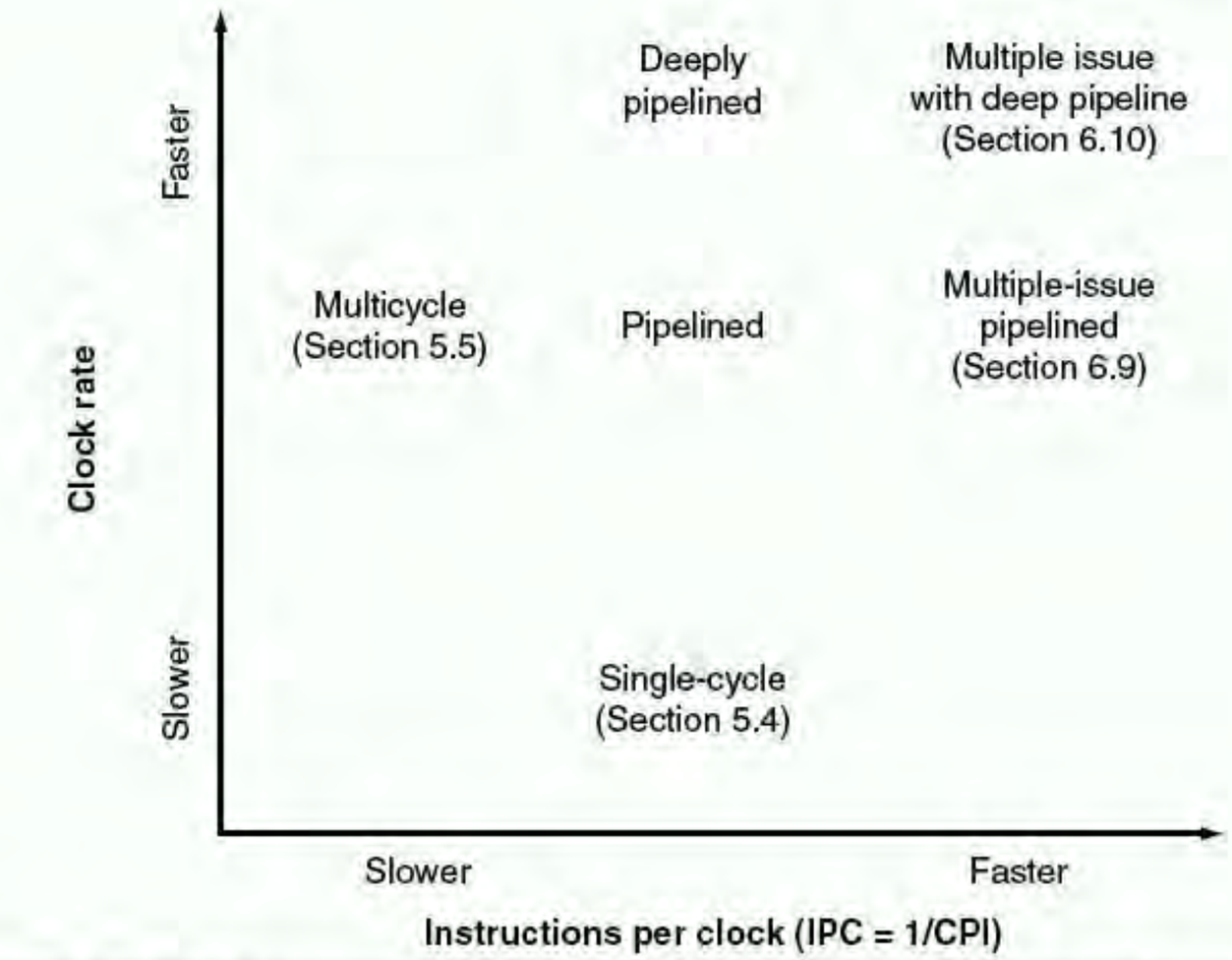


FIGURE 6.52 The performance consequences of simple (single-cycle) datapath and multicycle datapath from Chapter 5 and the pipelined execution model in Chapter 6. Remember that CPU performance is a function of IPC times clock rate, and hence moving to the upper right increases performance. Although the instructions per clock cycle is slightly larger in the simple datapath, the pipelined datapath is close, and it uses a clock rate as fast as the multicycle datapath.

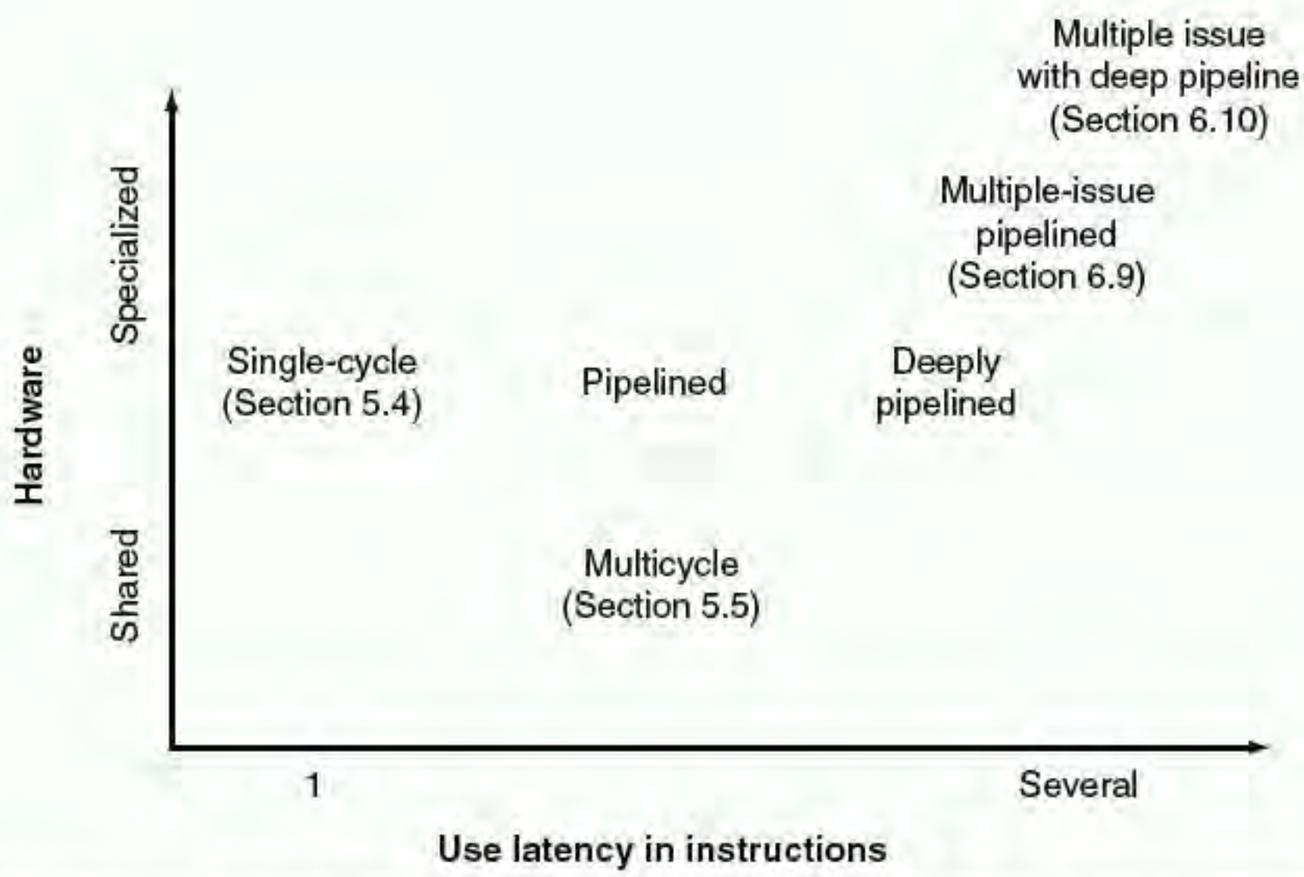


FIGURE 6.53 The basic relationship between the datapaths in Figure 6.52. Notice that the *x*-axis is use latency in instructions, which is what determines the ease of keeping the pipeline full. The pipelined datapath is shown as multiple clock cycles for instruction latency because the execution time of an instruction is not shorter; it's the instruction throughput that is improved.

With remarkable advances in processing, Amdahl's law suggests that another part of the system will become the bottleneck. That bottleneck is the topic of the next chapter: the memory system.

An alternative to pushing uniprocessors to automatically exploit parallelism at the instruction level is trying multiprocessors, which exploit parallelism at much coarser levels. Parallel processing is the topic of Chapter 9, which appears on the CD.

6.13 Historical Perspective and Further Reading

This section, which appears on the CD, discusses the history of the first pipelined processors, the earliest superscalars, the development of out-of-order and speculative techniques, as well as important developments in the accompanying compiler technology.

6.14 Exercises

6.1 [5] <§6.1> If the time for an ALU operation can be shortened by 25% (compared to the description in Figure 6.2 on page 373);

a. Will it affect the speedup obtained from pipelining? If yes, by how much? Otherwise, why?

b. What if the ALU operation now takes 25% more time?

6.2 [10] <§6.1> A computer architect needs to design the pipeline of a new microprocessor. She has an example workload program core with 10^6 instructions. Each instruction takes 100 ps to finish.

a. How long does it take to execute this program core on a nonpipelined processor?

b. The current state-of-the-art microprocessor has about 20 pipeline stages. Assume it is perfectly pipelined. How much speedup will it achieve compared to the nonpipelined processor?

c. Real pipelining isn't perfect, since implementing pipelining introduces some overhead per pipeline stage. Will this overhead affect instruction latency, instruction throughput, or both?

6.3 [5] <§6.1> Using a drawing similar to Figure 6.5 on page 377, show the forwarding paths needed to execute the following four instructions:

```
add $3, $4, $6
sub $5, $3, $2
lw  $7, 100($5)
add $8, $7, $2
```

6.4 [10] <§6.1> Identify all of the data dependencies in the following code. Which dependencies are data hazards that will be resolved via forwarding? Which dependencies are data hazards that will cause a stall?

```
add $3, $4, $2
sub $5, $3, $1
lw  $6, 200($3)
add $7, $3, $6
```

6.5 [5] <§6.1> For More Practice: Delayed Branches

6.6 [10] <§6.2> Using Figure 6.22 on page 400 as a guide, use colored pens or markers to show which portions of the datapath are active and which are inactive in each of the five stages of the `sw` instruction. We suggest that you use five photocopies of Figure 6.22 to answer this exercise. (We hereby grant you permission to violate the Copyright Protection Act in doing the exercises in Chapters 5 and 6!) Be sure to include a legend to explain your color scheme.

6.7 [5] <§6.2> For More Practice: Understanding Pipelines by Drawing Them

6.8 [5] <§6.2> For More Practice: Understanding Pipelines by Drawing Them

6.9 [15] <§6.2> For More Practice: Understanding Pipelines by Drawing Them

6.10 [5] <§6.2> For More Practice: Pipeline Registers

6.11 [15] <§§4.8, 6.2> For More Practice: Pipelining Floating Point

6.12 [15] <§6.3> Figure 6.37 on page 417 and Figure 6.35 on page 415 are two styles of drawing pipelines. To make sure you understand the relationship between these two styles, draw the information in Figures 6.31 through 6.35 on pages 410 through 415 using the style of Figure 6.37 on page 417. Highlight the active portions of the data paths in the figure.

6.13 [20] <§6.3> Figure 6.14.10 is similar to Figure 6.14.7 on page 6.14-9 in the For More Practice section, but the instructions are unidentified. Determine as much as you can about the five instructions in the five pipeline stages. If you cannot fill in a field of an instruction, state why. For some fields it will be easier to decode the machine instructions into assembly language, using Figure 3.18 on

page 205 and Figure A.10.2 on page A-50 as references. For other fields it will be easier to look at the values of the control signals, using Figures 6.26 through 6.28 on pages 403 and 405 as references. You may need to carefully examine Figures 6.14.5 through 6.14.9 to understand how collections of control values are presented (i.e., the leftmost bit in one cycle will become the uppermost bit in another cycle). For example, the EX control value for the subtract instruction, 1100, computed during the ID stage of cycle 3 in Figure 6.14.6, becomes three separate values specifying RegDst (1), ALUOp (10), and ALUSrc (0) in cycle 4.

6.14 [40] <§6.3> The following piece of code is executed using the pipeline shown in Figure 6.30 on page 409:

```
lw  $5, 40($2)
add $6, $3, $2
or  $7, $2, $1
and $8, $4, $3
sub $9, $2, $1
```

At cycle 5, right before the instructions are executed, the processor state is as follows:

a. The PC has the value 100_{ten}, the address of the `sub_instruction`.

b. Every register has the initial value 10_{ten} plus the register number (e.g., register $8 has the initial value 18_{ten}).

c. Every memory word accessed as data has the initial value 1000_{ten} plus the byte address of the word (e.g., Memory[8] has the initial value 1008_{ten}).

Determine the value of every field in the four pipeline registers in cycle 5.

6.15 [20] <§6.3> For More Practice: Labeling Pipeline Diagrams with Control

6.16 [20] <§6.4> For More Practice: Illustrating Diagrams with Forwarding

6.17 [5] <§§6.4, 6.5> Consider executing the following code on the pipelined datapath of Figure 6.36 on page 416:

```
add   $2,  $3,  $1
sub   $4,  $3,  $5
add   $5,  $3,  $7
add   $7,  $6,  $1
add   $8,  $2,  $6
```

At the end of the fifth cycle of execution, which registers are being read and which register will be written?

6.18 [5] <§§6.4, 6.5> With regard to the program in Exercise 6.17, explain what the forwarding unit is doing during the fifth cycle of execution. If any comparisons are being made, mention them.

6.19 [5] <§§6.4, 6.5> With regard to the program in Exercise 6.17, explain what the hazard detection unit is doing during the fifth cycle of execution. If any comparisons are being made, mention them.

6.20 [20] <§§6.4, 6.5> For More Practice: Forwarding in Memory

6.21 [5] <§6.5> We have a program of 10^3 instructions in the format of "`lw, add, lw, add, ...`" The `add` instruction depends (and only depends) on the `lw` instruction right before it. The `lw` instruction also depends (and only depends) on the `add` instruction right before it. If the program is executed on the pipelined datapath of Figure 6.36 on page 416:

a. What would be the actual CPI?

b. Without forwarding, what would be the actual CPI?

6.22 [5] <§§6.4, 6.5> Consider executing the following code on the pipelined datapath of Figure 6.36 on page 416:

```
lw   $4, 100($2)
sub  $6, $4, $3
add  $2, $3, $5
```

How many cycles will it take to execute this code? Draw a diagram like that of Figure 6.34 on page 414 that illustrates the dependencies that need to be resolved, and provide another diagram like that of Figure 6.35 on page 415 that illustrates how the code will actually be executed (incorporating any stalls or forwarding) so as to resolve the identified problems.

6.23 [15] <§6.5> List all the inputs and outputs of the forwarding unit in Figure 6.36 on page 416. Give the names, the number of bits, and brief usage for each input and output.

6.24 [20] <§6.5> For More Practice: Illustrating Diagrams with Forwarding and Stalls

6.25 [20] <§6.5> For More Practice: Impact on Forwarding of Moving It to ID Stage

6.26 [15] <§§6.2–6.5> For More Practice: Impact of Memory Addressing Mode on Pipeline

6.27 [10] <§§6.2–6.5> For More Practice: Impact of Arithmetic Operations with Memory Operands on Pipeline

6.28 [30] <§6.5, Appendix C> For More Practice: Forwarding Unit Hardware Design

6.29 [1 week] <§§6.4, 6.5> Using the simulator provided with this book, collect statistics on data hazards for a C program (supplied by either the instructor or with the software). You will write a subroutine that is passed the instruction to be executed, and this routine must model the five-stage pipeline in this chapter. Have your program collect the following statistics:

- Number of instructions executed.
- Number of data hazards not resolved by forwarding and number resolved by forwarding.
- If the MIPS C compiler that you are using issues `nop` instructions to avoid hazards, count the number of `nop` instructions as well.

Assuming that the memory accesses always take 1 clock cycle, calculate the average number of clock cycles per instruction. Classify `nop` instructions as stalls inserted by software, then subtract them from the number of instructions executed in the CPI calculation.

6.30 [7] <§§6.4, 6.5> In the example on page 425, we saw that the performance advantage of the multicycle design was limited by the longer time required to access memory versus use the ALU. Suppose the memory access became 2 clock cycles long. Find the relative performance of the single-cycle and multicycle designs. In the next few exercises, we extend this to the pipelined design, which requires lots more work!

6.31 [10] <§6.6> For More Practice: Coding with Conditional Moves

6.32 [10] <§6.6> For More Practice: Performance Advantage of Conditional Move

6.33 [20] <§§6.2–6.6> In the example on page 425, we saw that the performance advantage of both the multicycle and the pipelined designs was limited by the longer time required to access memory versus use the ALU. Suppose the memory access became 2 clock cycles long. Draw the modified pipeline. List all the possible new forwarding situations and all possible new hazards and their length.

6.34 [20] <§§6.2–6.6> Redo the example on page 425 using the restructured pipeline of Exercise 6.33 to compare the single-cycle and multicycle. For branches, assume the same prediction accuracy, but increase the penalty as appropriate. For loads, assume that the subsequent instructions depend on the load with a probability of 1/2, 1/4, 1/8, 1/16, and so on. That is, the instruction following a load by two has a 25% probability of using the load result as one of its sources. Ignoring any other data hazards, find the relative performance of the pipelined design to the single-cycle design with the restructured pipeline.

6.35 [10] <§§6.4–6.6> As pointed out on page 418, moving the branch comparison up to the ID stage introduces an opportunity for both forwarding and hazards that cannot be resolved by forwarding. Give a set of code sequences that show the possible

forwarding paths required and hazard cases that must be detected, considering only one of the two operands. The number of cases should equal the maximum length of the hazard if no forwarding existed.

6.36 [15] <§6.6> We have a program core consisting of five conditional branches. The program core will be executed thousands of times. Below are the outcomes of each branch for one execution of the program core (T for taken, N for not taken).

Branch 1: T-T-T
Branch 2: N-N-N-N
Branch 3: T-N-T-N-T-N
Branch 4: T-T-T-N-T
Branch 5: T-T-N-T-T-N-T

Assume the behavior of each branch remains the same for each program core execution. For dynamic schemes, assume each branch has its own prediction buffer and each buffer initialized to the same state before each execution. List the predictions for the following branch prediction schemes:

a. Always taken
b. Always not taken
c. 1-bit predictor, initialized to predict taken
d. 2-bit predictor, initialized to weakly predict taken

What are the prediction accuracies?

6.37 [10] <§§6.4–6.6> Sketch all the forwarding paths for the branch inputs and show when they must be enabled (as we did on page 407).

6.38 [10] <§§6.4–6.6> Write the logic to detect any hazards on the branch sources, as we did on page 410.

6.39 [10] <§§6.4–6.6> The example on page 378 shows how to *maximize* performance on our pipelined datapath with forwarding and stalls on a use following a load. Rewrite the following code to *minimize* performance on this datapath—that is, reorder the instructions so that this sequence takes the *most* clock cycles to execute while still obtaining the same result.

```
lw    $2, 100($6)
lw    $3, 200($7)
add   $4, $2, $3
add   $6, $3, $5
sub   $8, $4, $6
lw    $7, 300($8)
beq   $7, $8, Loop
```

6.40 [20] <§6.6> Consider the pipelined datapath in Figure 6.54 on page 461. Can an attempt to flush and an attempt to stall occur simultaneously? If so, do they result in conflicting actions and/or cooperating actions? If there are any cooperating actions, how do they work together? If there are any conflicting actions, which should take priority? Is there a simple change you can make to the datapath to ensure the necessary priority? You may want to consider the following code sequence to help you answer this question:

```
beq $1, $2, TARGET   # assume that the branch is taken
lw  $3, 40($4)
add $2, $3, $4
sw  $2, 40($4)
TARGET:  or  $1, $1, $2
```

6.41 [15] <§§ 6.4, 6.7> The Verilog for implementing forwarding in Figure 6.7.2 on page 6.7-4–6.7-5 did not consider forwarding of a result as the value to be stored by a SW instruction. Add this to the Verilog code.

6.42 [5] <§§6.5, 6.7> The Verilog for implementing stalls in Figure 6.7.3 on page 6.7-6–6.7-7 did not consider forwarding of a result to use in an address calculation. Make this simple addition to the Verilog code.

6.43 [15] <§§6.6, 6.7> The Verilog code for implementing branch hazard detection and stalls in Figure 6.7.3 on page 6.7-6–6.7-7 does not detect the possibility of data hazards for the two source registers of a `BEQ` instruction. Extend the Verilog in Figure 6.7.3 on page 6.7-6–6.7-7 to handle all data hazards for branch operands. Write both the forwarding and stall logic needed for completing branches during ID.

6.44 [10] <§§6.6, 6.7> Rewrite theVerilog code in 6.7.3 on page 6.7-6–6.7-7 to implement a delayed branch strategy.

6.45 [20] <§§6.6, 6.7> Rewrite the verilog code in Figure 6.7.3 on page 6.7-6–6.7-7 to implement a branch target buffer. Assume the buffer is implemented with a module with the following definition:

```
module PredictPC (currentPC,nextPC,miss,update,destination);
   input currentPC,
      update, // true if previous prediction was unavailable or incorrect
      destination; / used with update to correct a prediction
   output nextPC, // returns the next PC if prediction is accurate
      miss; // true means no prediction in buffer
endmodule;
```

Make sure you accomodate all three possibilities: a correct prediction, a miss in the buffer (that is, miss = true), and an incorrect prediction. In the last two cases, you must also update the prediction.

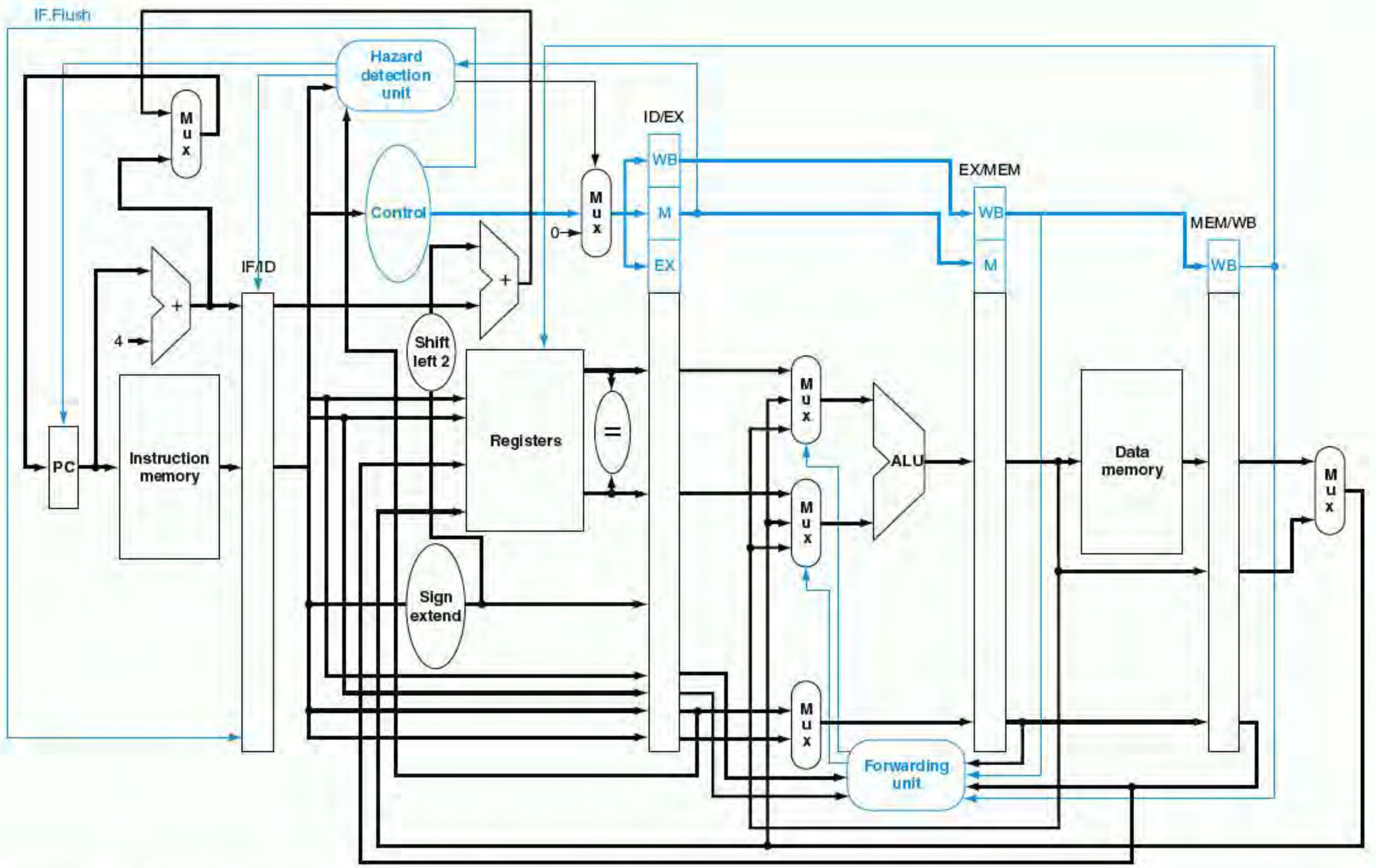


FIGURE 6.54 Datapath for branch, including hardware to flush the instruction that follows the branch. This optimization moves the branch decision from the fourth pipeline stage to the second; only one instruction that follows the branch will be in the pipe at that time. The control line IF.Flush turns the fetched instruction into a nop by zeroing the IF/ID pipeline register. Although the flush line is shown coming from the control unit in this figure, in reality it comes from hardware that determines if a branch is taken, labeled with an equal sign to the right of the registers in the ID stage. The forwarding muxes and paths must also be added to this stage, but are not shown to simplify the figure.

(page 461)

6.46 [1 month] <§§5.4, 6.3–6.8> If you have access to a simulation system such as Verilog or ViewLogic, first design the single-cycle datapath and control from Chapter 5. Then evolve this design into a pipelined organization, as we did in this chapter. Be sure to run MIPS programs at each step to ensure that your refined design continues to operate correctly.

6.47 [10] <§6.9> The following code has been unrolled once but not yet scheduled. Assume the loop index is a multiple of two (i.e., `$10` is a multiple of eight):

```
Loop:   lw   $2,  0($10)
        sub  $4,  $2, $3
        sw   $4,  0($10)
        lw   $5,  4($10)
        sub  $6,  $5, $3
        sw   $6,  4($10)
        addi $10, $10, 8
        bne  $10, $30, Loop
```

Schedule this code for fast execution on the standard MIPS pipeline (assume that it supports `addi` instruction). Assume initially `$10` is 0 and `$30` is 400 and that branches are resolved in the MEM stage. How does the scheduled code compare against the original unscheduled code?

6.48 [20] <§6.9> This exercise is similar to Exercise 6.47, except this time the code should be unrolled twice (creating three copies of the code). However, it is not known that the loop index is a multiple of three, and thus you will need to invent a means of ensuring that the code still executes properly. (Hint: Consider adding some code to the beginning or end of the loop that takes care of the cases not handled by the loop.)

6.49 [20] <§6.9> Using the code in Exercise 6.47, unroll the code four times and schedule it for the static multiple-issue version of the MIPS processor described on pages 436–439. You may assume that the loop executes for a multiple of four times.

6.50 [10] <§§6.1–6.9> As technology leads to smaller feature sizes, the wires become relatively slower (as compared to the logic). As logic becomes faster with the shrinking feature size and clock rates increase, wire delays consume more clock cycles. That is why the Pentium 4 has several pipeline stages dedicated to transferring data along wires from one part of the pipeline to another. What are the drawbacks to having to add pipe stages for wire delays?

6.51 [30] <§6.10> New processors are introduced more quickly than new versions of textbooks. To keep your textbook current, investigate some of the latest developments in this area and write a one-page elaboration to insert at the end of Section 6.10. Use the World-Wide Web to explore the characteristics of the lastest processors from Intel or AMD as a starting point.

Answers to Check Yourself

§6.1, page 384: 1. Stall on the LW result. 2. Bypass the ADD result. 3. No stall or bypass required.
§6.2, page 399: Statements 2 and 5 are correct; the rest are incorrect.
§6.6, page 426: 1. Predict not taken. 2. Predict taken. 3. Dynamic prediction.
§6.7, page 6.7-3: Statements 1 and 3 are both true.
§6.7, page 6.7-7: Only statement #3 is completely accurate.
§6.8, page 432: Only #4 is totally accurate. #2 is partially accurate.
§6.9, page 447: Speculation: both; reorder buffer: hardware; register renaming: both; out-of-order execution: hardware; predication: software; branch prediction: both; VLIW: software; superscalar: hardware; EPIC: both, since there is substantial hardware support; multiple issue: both; dynamic scheduling: hardware.
§6.10, page 450: All the statements are false.

Computers in the Real World

Mass Communication without Gatekeepers

Problem to solve: Offer society sources of news and opinion beyond those found in the traditional mass media.

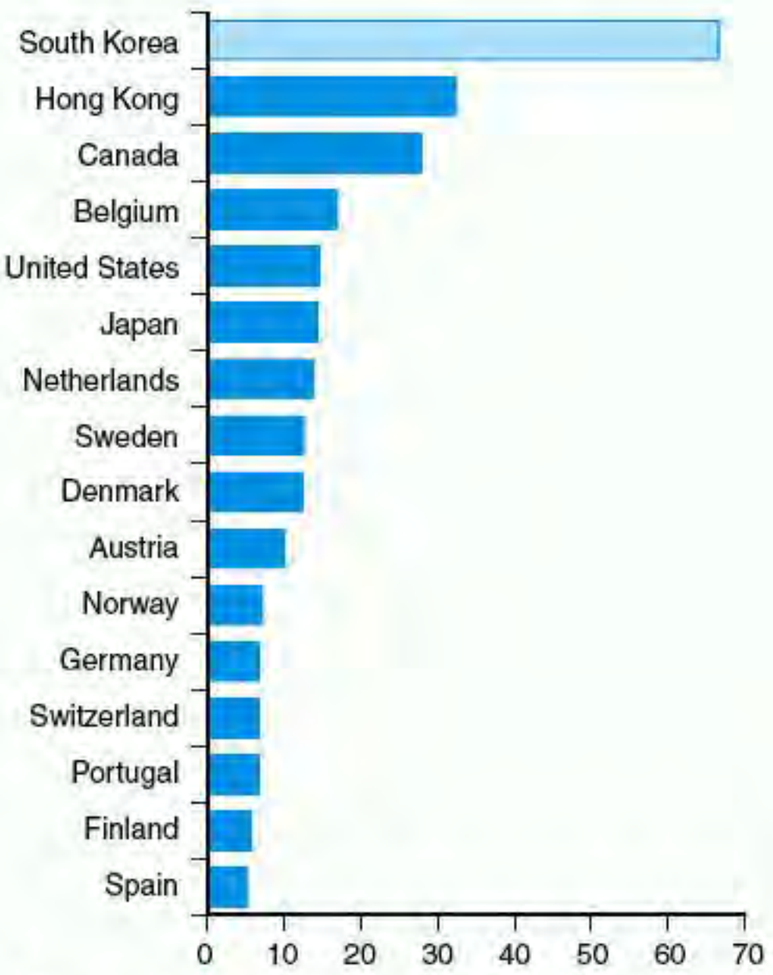
Solution: Use the Internet and World Wide Web to select and publish nontraditional and nonlocal news sources.

The Internet holds the promise of allowing citizens to communicate without the information first being interpreted by traditional mass media like television, newspapers, and magazines. To see what the future might be, we could look at countries that have widespread, high-speed Internet access.

One place is South Korea. In 2002, 68% of South Korean households had broadband access, compared to 15% in the United States and 8% in Western Europe. (Broadband is generally digital subscriber loop or cable speeds, about 300 to 1000 Kbps.) The main reason for the greater penetration is that 70% of households are in large cities and almost half are found in apartments. Hence, the Korean telecommunications industry could afford to quickly offer broadband to 90% of the households.

What was the impact of widespread high-speed access on Korean society? Internet news sites became extremely popular. One example is OhMyNews, which publishes articles from *anyone* after first checking that the facts in the article are correct.

Many believe that Internet news services influenced the outcome of the 2002 Korean presidential election. First, they encouraged more young people to vote. Second, the winning candidate advocated politics that were



Percentage of households with broadband connections by country in 2002.
Source: The Yankee Group, Boston.

closer to those popular on the Internet news services. Together they overcame the disadvantage that most major media organizations endorsed his opponent.

Google News is another example of nontraditional access to news that goes beyond the mass media of one country. It searches international news services for topics, and then summarizes and displays them by popularity. Rather than leaving the decision of what articles should be on the front page to local newspaper editors, the worldwide media decides. In addition, by providing links to stories from many countries, the reader gets an international perspective rather than a local one. It also is updated many times a day unlike a daily newspaper. The figure below compares the *New York Times* front page to the Google News Web site on the same day.

The widespread impact of these technologies reminds us that computer engineers have responsibilities to their communities. We must be aware of societal values concerning privacy, security, free speech, and so on to ensure that new technological innovations enhance those values rather than inadvertently compromising them.

To learn more see these references on the library

- "Seriously wired," *The Economist*, April 17, 2003.
- OhMyNews, www.ohmynews.com
- Google News, www.news.google.com

New York Times Front Page	Google News
Judge Rules Out a Death Penalty for 9/11 Suspect Rebuke for Justice Dept. **Poll Shows Drop In Confidence on Bush Skill In Handling Crises** Country on Wrong Track, Says Solid Majority **Revised Admission for High Schools** City Says Students Will Get First Preference **No Illicit Arms Found In Iraq, U.S. Inspector Tells Congress** **U.S. Practice How to Down Hijacked Jets** **Coetzee, Writer of Apartheid as Bleak Mirror, Wins Nobel** **Sexual Accusations Lead to an Apology by Schwarzenegger** **Interim Chief Accepts Stock Exchange Shift** **Yankees Even with Twins** Agency Warns of Fake Drugs Limbaugh Fallback Position	Top Stories More than 1000 rally behind Schwarzenegger AP - 5 minutes ago Maria Shriver defends husband CNN Can accusations hurt Arnold's campaign? KESQ and 1252 related Bush: Hussein 'A Danger to the World' ABC news - 5 hours ago Bush Stands By Decision Voice of America Hunt for weapons yields no evidence The Canberra Times and 598 related World Stories Defiant UN chief announces rival blueprint for Iraq The Times (UK) - 2 hours ago France, Russia Assail US Draft on Iraq Reuters and 782 related

***New York Times* versus Google News on october 3, 2003 at 6 PM PT.** The newspaper front page headlines must balance big stories with national news, local news, and sports. Google News has many stories per headline from around the world, with links the reader can follow. Google stories vary by time of day and hence are more recent.

7

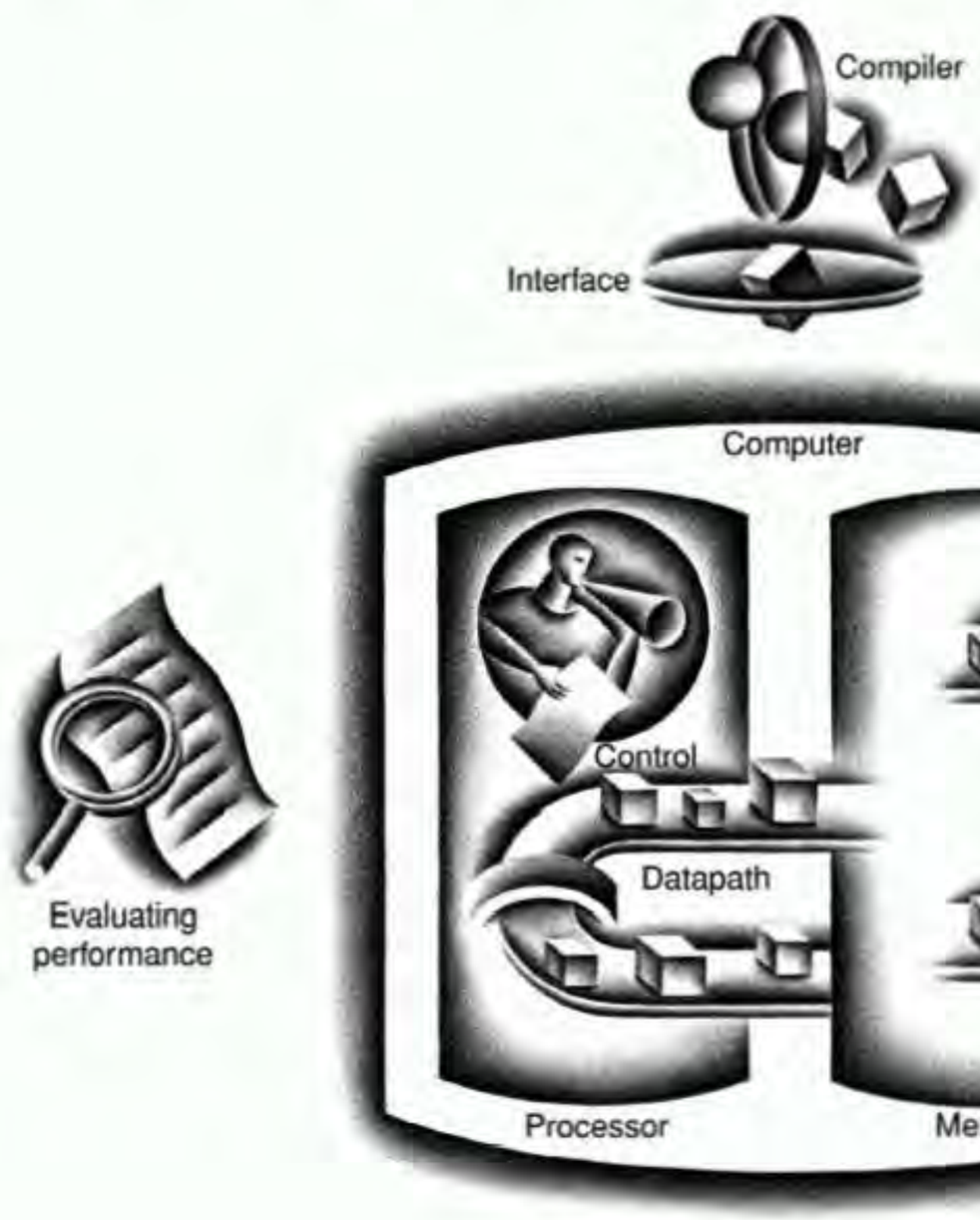
Large and Fast: Exploiting Memory Hierarchy

Ideally one would desire an indefinitely large memory capacity such that any particular . . . word would be immediately available. . . . We are . . . forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

A. W. Burks, H. H. Goldstine, and J. von Neumann
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, 1946

7.1 **Introduction** 468
7.2 **The Basics of Caches** 473
7.3 **Measuring and Improving Cache Performance** 492
7.4 **Virtual Memory** 511
7.5 **A Common Framework for Memory Hierarchies** 538
7.6 **Real Stuff: The Pentium P4 and the AMD Opteron Memory Hierarchies** 546
7.7 **Fallacies and Pitfalls** 550
7.8 **Concluding Remarks** 552
7.9 **Historical Perspective and Further Reading** 555
7.10 **Exercises** 555

The Five Classic Components of a Computer



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 aid 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 a library with a collection of books that you have pulled from the shelves and are examining. You find that several of the important computers that you need to write about are described in the books you have, but there is nothing about the EDSAC. Therefore, 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 most of your 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 memory in computers, just as it would be impossible for you to fit all the library books on your desk and still find 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 The principle stating that if a data location is referenced then it will tend to be referenced again soon.

spatial locality The locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.

- **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 English computers 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 too 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 faster memories are more expensive per bit than the slower memories and thus smaller.

memory hierarchy A structure that uses multiple levels of memories; as the distance from the CPU increases, the size of the memories and the access time both increase.

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 processor (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.8 of Appendix B. The third 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 2004:

Memory technology	Typical access time	$ per GB in 2004
SRAM	0.5–5 ns	$4000–$10,000
DRAM	50–70 ns	$100–$200
Magnetic disk	5,000,000–20,000,000 ns	$0.50–$2

Because of these differences in cost and access time, it is advantageous to build memory as a hierarchy of levels. Figure 7.1 shows the faster memory is close to the processor and the slower, less expensive memory is below it. 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.

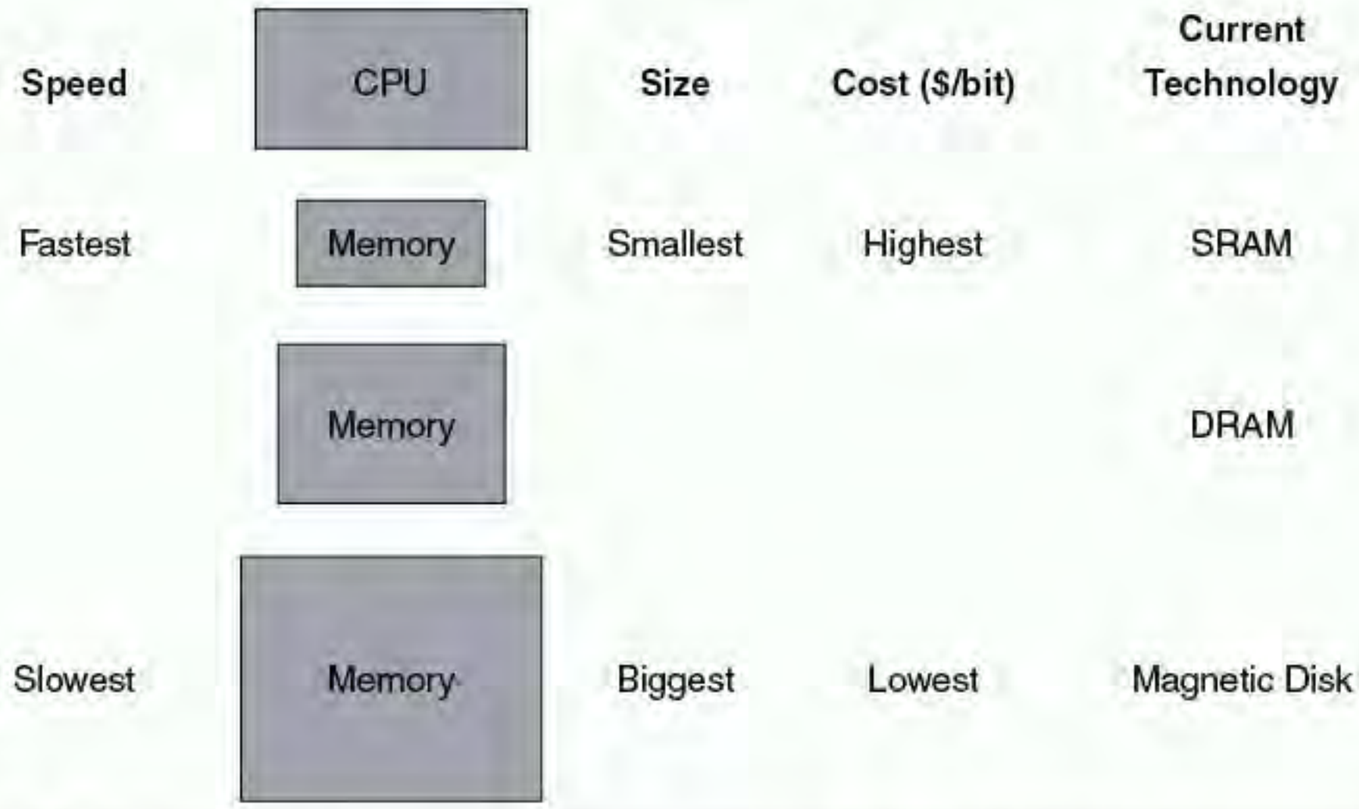


FIGURE 7.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory.

The memory system is organized as a hierarchy: a level closer to the processor is generally a subset of any level further away, and all the data is stored at the lowest level. By analogy, 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. Figure 7.2 shows that the minimum unit of information that can be either present or not present in the two-level hierarchy is called a **block** or a *line*; in our library analogy, a block of information is one book.

block The minimum unit of information that can be either present or not present in the two-level hierarchy.

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 go from your desk to the shelves to find the desired book.) 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 perfor-

hit rate The fraction of memory accesses found in a cache.

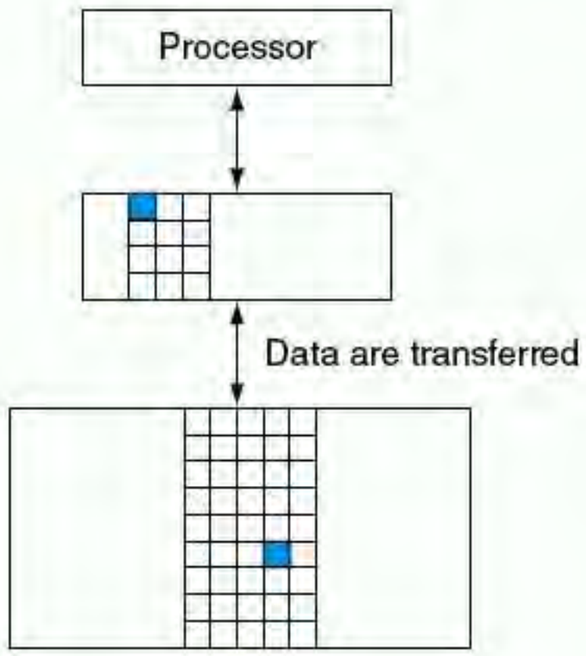


FIGURE 7.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a *block*. Usually we transfer an entire block when we copy something between levels.

mance of the memory hierarchy. The **miss rate** (1 – hit rate) is the fraction of memory accesses not found in the upper level.

Since performance is the major reason for having a memory hierarchy, the time to service 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 get a new book from 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 computer. 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

miss rate The fraction of memory accesses not found in a level of the memory hierarchy.

hit time The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.

miss penalty The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, and insert it in the level that experienced the miss.

memory hierarchies to get good performance. We show how important this understanding is with two examples.

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 dozens of recent doctoral theses have demonstrated.

Check Yourself

Which of the following statements are generally true?

1. Caches take advantage of temporal locality.
2. On a read, the value returned depends on which blocks are in the cache.
3. Most of the cost of the memory hierarchy is at the highest level.

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.

Figure 7.3 shows that a memory hierarchy uses smaller and faster memory technologies close to the processor. 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 also present in level $i + 1$.

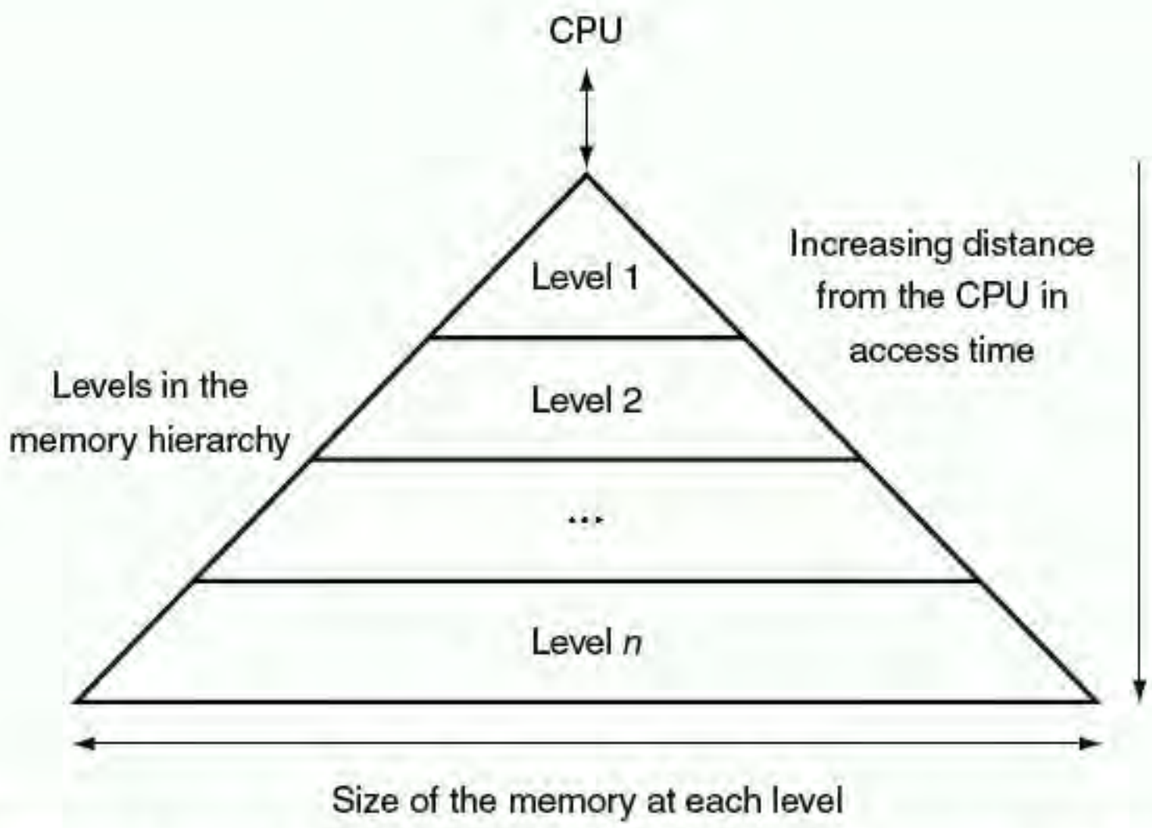


FIGURE 7.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size. This structure with the appropriate operating mechanisms allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level *n*. Maintaining this illusion is the subject of this chapter. Although the local disk is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy.

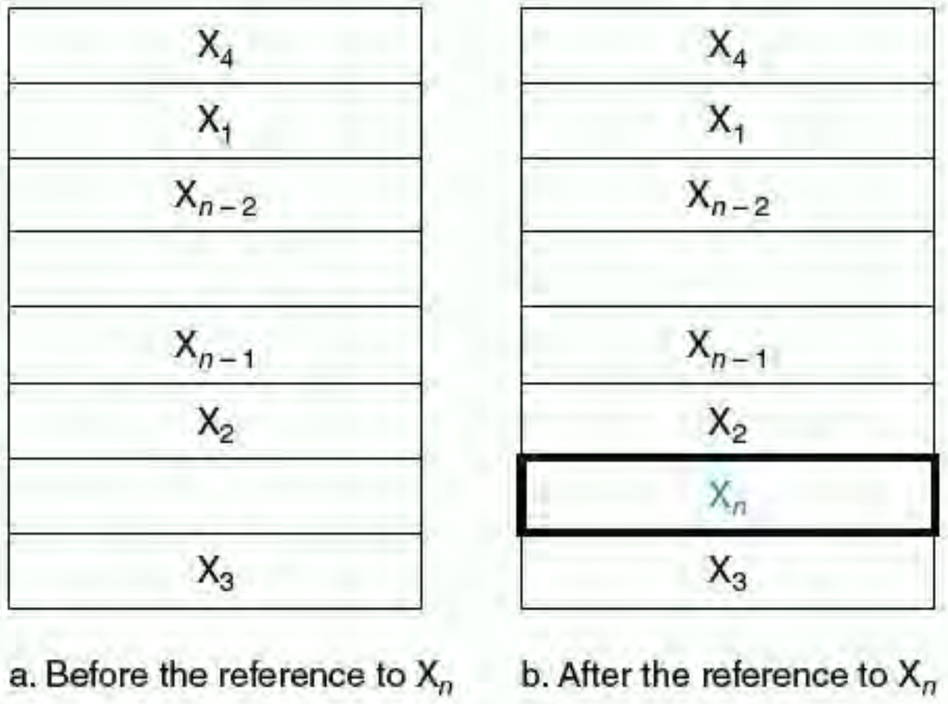
7.2 The Basics of Caches

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 processor and main memory in the first commercial computer 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 computers in the early 1960s and in production computers later in that same decade; every general-purpose computer built today, from servers to low-power embedded processors, includes caches.

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. (Readers already familiar with cache basics may want to skip to Section 7.3 on page 492.)



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

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

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 $X_1, X_2, \ldots, X_{n-1}$, and the processor requests a word X_n that is not in the cache. This request results in a miss, and the word X_n is brought from memory into cache.

In looking at the scenario in Figure 7.4, there are two questions to answer: How do we know if a data item is in the cache? Moreover, if it is, how do we find it? The answers to these two questions are related. If each word can go in exactly one place in the cache, then it is straightforward to find the word if it is in the cache. The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the *address* of the word in memory. This cache structure is called **direct mapped**, since each memory location is mapped directly to exactly one location in the cache. The typical mapping between addresses and cache locations for a direct-mapped cache is usually simple. For example, almost all direct-mapped caches use the mapping

direct-mapped cache A cache structure in which each memory location is mapped to exactly one location in the cache.

(Block address) modulo (Number of cache blocks in the cache)

This mapping is attractive because if the number of entries in the cache is a power of two, then modulo can be computed simply by using the low-order $\log_2$ (cache size in blocks) bits of the address; hence the cache may be accessed directly with the low-order bits. For example, Figure 7.5 shows how the memory addresses

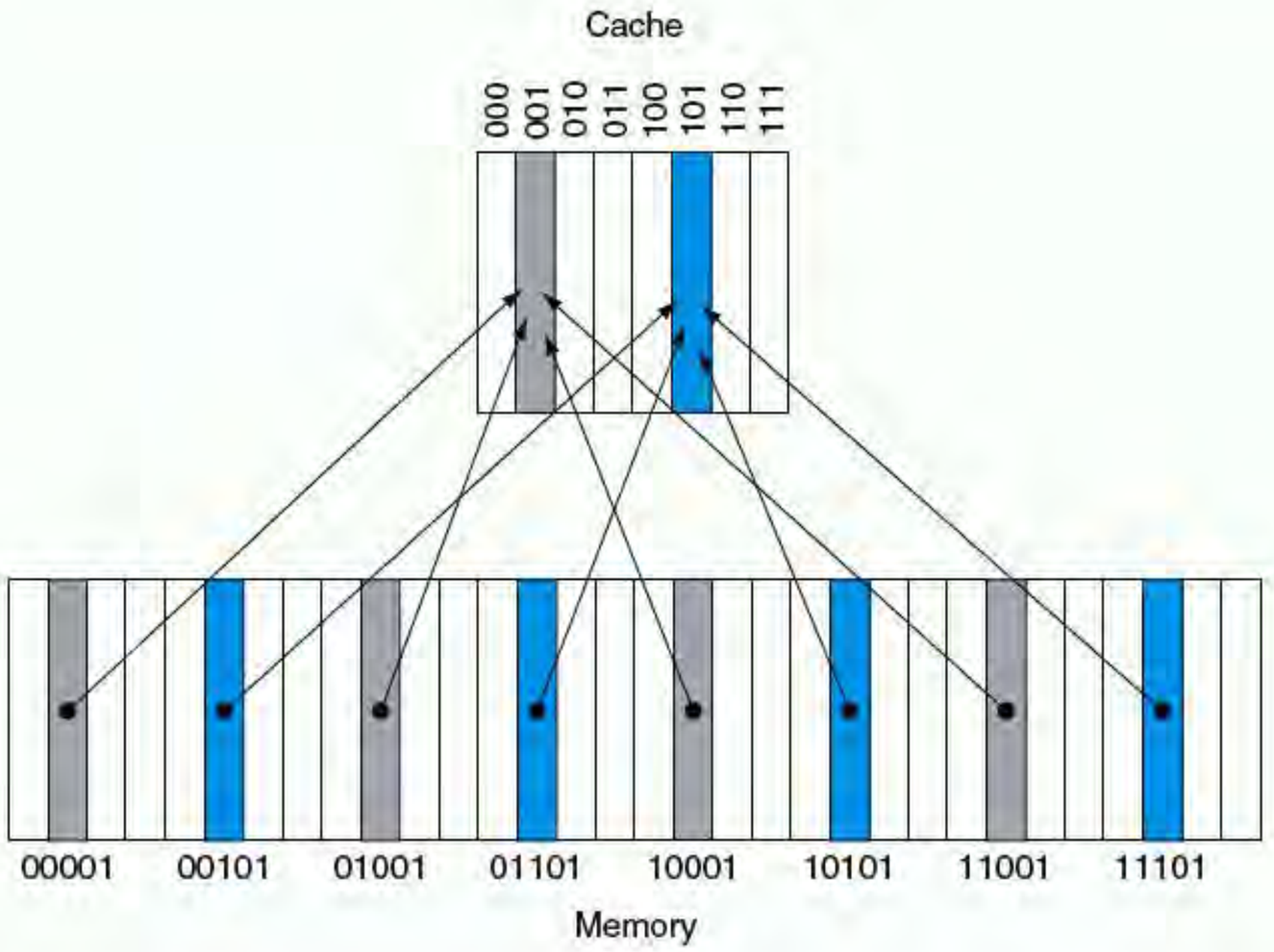


FIGURE 7.5 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the cache word X modulo 8. That is, the low-order $\log_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001_{two}, 01001_{two}, 10001_{two}, and 11001_{two} all map to entry 001_{two} of the cache, while addresses 00101_{two}, 01101_{two}, 10101_{two}, and 11101_{two} all map to entry 101_{two} of the cache.

between 1_{ten} (00001_{two}) and 29_{ten} (11101_{two}) map to locations 1_{ten} (001_{two}) and 5_{ten} (101_{two}) in a direct-mapped cache of eight words.

Because each cache location can contain the contents of a number of different memory locations, how do we know whether the data in the cache corresponds to a requested word? That is, how do we know whether a requested word is in the cache or not? We answer this question by adding a set of **tags** to the cache. The tags contain the address information required to identify whether a word in the cache corresponds to the requested word. The tag needs only to contain the upper portion of the address, corresponding to the bits that are not used as an index into the cache. For example, in Figure 7.5 we need only have the upper 2 of the 5 address bits in the tag, since the lower 3-bit index field of the address selects the block. We exclude the index bits because they are redundant, since by definition the index field of every address must have the same value.

tag A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word.

We also need a way to recognize that a cache block does not have valid information. For instance, when a processor starts up, the cache does not have good

data, and the tag fields will be meaningless. Even after executing many instructions, some of the cache entries may still be empty, as in Figure 7.4. Thus, we need to know that the tag should be ignored for such entries. The most common method is to add a **valid bit** to indicate whether an entry contains a valid address. If the bit is not set, there cannot be a match for this block.

valid bit A field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains valid data.

For the rest of this section, we will focus on explaining how reads work in a cache and how the cache control works for reads. In general, handling reads is a little simpler than handling writes, since reads do not have to change the contents of the cache. After seeing the basics of how reads work and how cache misses can be handled, we'll examine the cache designs for real computers and detail how these caches handle writes.

Accessing a Cache

Figure 7.6 shows the contents of an eight-word direct-mapped cache as it responds to a series of requests from the processor. Since there are eight blocks in the cache, the low-order 3 bits of an address give the block number. Here is the action for each reference:

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (7.6b)	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	miss (7.6c)	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	miss (7.6d)	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	miss (7.6e)	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	miss (7.6f)	$(10010_{two} \bmod 8) = 010_{two}$

When the word at address 18 (10010_{two}) is brought into cache block 2 (010_{two}), the word at address 26 (11010_{two}), which was in cache block 2 (010_{two}), must be replaced by the newly requested data. This behavior allows a cache to take advantage of temporal locality: recently accessed words replace less recently referenced words. This situation is directly analogous to needing a book from the shelves and having no more space on your desk—some book already on your desk must be returned to the shelves. In a direct-mapped cache, there is only one place to put the newly requested item and hence only one choice of what to replace.

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

FIGURE 7.6 The cache contents are shown after each reference request that *misses*, with the index and tag fields shown in binary. The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), and 10010_{two} (miss). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss. The tag field will contain only the upper portion of the address. The full address of a word contained in cache block i with tag field j for this cache is $j \times 8 + i$, or equivalently the concatenation of the tag field j and the index i. For example, in cache f above, index 010 has tag 10 and corresponds to address 10010.

We know where to look in the cache for each possible address: the low-order bits of an address can be used to find the unique cache entry to which the address could map. Figure 7.7 shows how a referenced address is divided into

- a cache index, which is used to select the block
- a tag field, which is used to compare with the value of the tag field of the cache

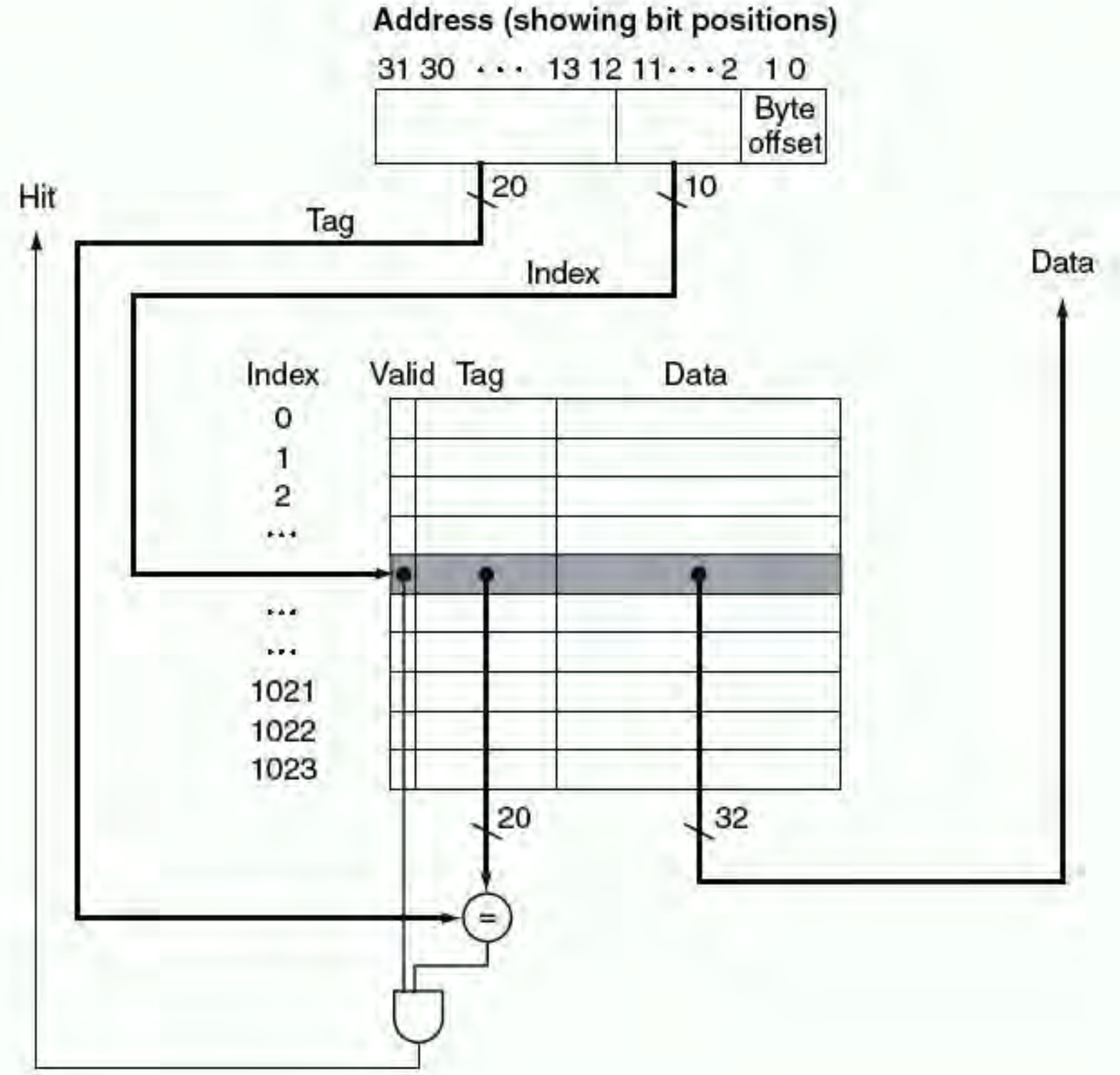


FIGURE 7.7 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words and a block size of 1 word, 10 bits are used to index the cache, leaving 32 – 10 – 2 = 20 bits to be compared against the tag. If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

The index of a cache block, together with the tag contents of that block, uniquely specifies the memory address of the word contained in the cache block. Because the index field is used as an address to access the cache and because an n-bit field has 2^n values, the total number of entries in a direct-mapped cache must be a power of two. In the MIPS architecture, since words are aligned to multiples of 4 bytes, the least significant 2 bits of every address specify a byte within a word and hence are ignored when selecting the word in the block.

The total number of bits needed for a cache is a function of the cache size and the address size because the cache includes both the storage for the data and the tags. The size of the block above was one word, but normally it is several. Assuming the 32-bit byte address, a direct-mapped cache of size 2^n blocks with 2^m-word (2^{m+2}-byte) blocks will require a tag field whose size is $32 - (n + m + 2)$ bits because n bits are used for the index, m bits are used for the word within the block, and 2 bits are used for the byte part of the address. The total number of bits in a direct-mapped cache is $2^n \times$ (block size + tag size + valid field size). Since the block size is 2^m words (2^{m+5} bits) and the address size is 32 bits, the number of bits in such a cache is $2^n \times (m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (m \times 32 + 31 - n - m)$. However, the naming convention is to excludes the size of the tag and valid field and to count only the size of the data.

Bits in a Cache

EXAMPLE

How many total bits are required for a direct-mapped cache with 16 KB of data and 4-word blocks, assuming a 32-bit address?

ANSWER

We know that 16 KB is 4K words, which is 2^{12} words, and, with a block size of 4 words (2^2), 2^{10} blocks. Each block has $4\times$ 32 or 128 bits of data plus a tag, which is $32 - 10 - 2 - 2$ bits, plus a valid bit. Thus, the total cache size is

$$2^{10} \times (128 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ Kbits}$$

or 18.4 KB for a 16 KB cache. For this cache, the total number of bits in the cache is about 1.15 times as many as needed just for the storage of the data.

EXAMPLE

Mapping an Address to a Multiword Cache Block

Consider a cache with 64 blocks and a block size of 16 bytes. What block number does byte address 1200 map to?

ANSWER

We saw the formula on page 474. The block is given by

$$\text{(Block address) modulo (Number of cache blocks)}$$

where the address of the block is

$$\frac{\text{Byte address}}{\text{Bytes per block}}$$

Notice that this block address is the block containing all addresses between

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block}$$

and

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block} + (\text{Bytes per block} - 1)$$

Thus, with 16 bytes per block, byte address 1200 is block address

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75$$

which maps to cache block number (75 modulo 64) = 11. In fact, this block maps all addresses between 1200 and 1215.

Larger blocks exploit spatial locality to lower miss rates. As Figure 7.8 shows, increasing the block size usually decreases the miss rate. The miss rate may go up eventually if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the cache will become small, and there will be a great deal of competition for those blocks. As a result, a block will be bumped out of the cache before many of its words are accessed. Stated alternatively, spatial locality among the words in a block decreases with a very large block; consequently, the benefits in the miss rate become smaller.

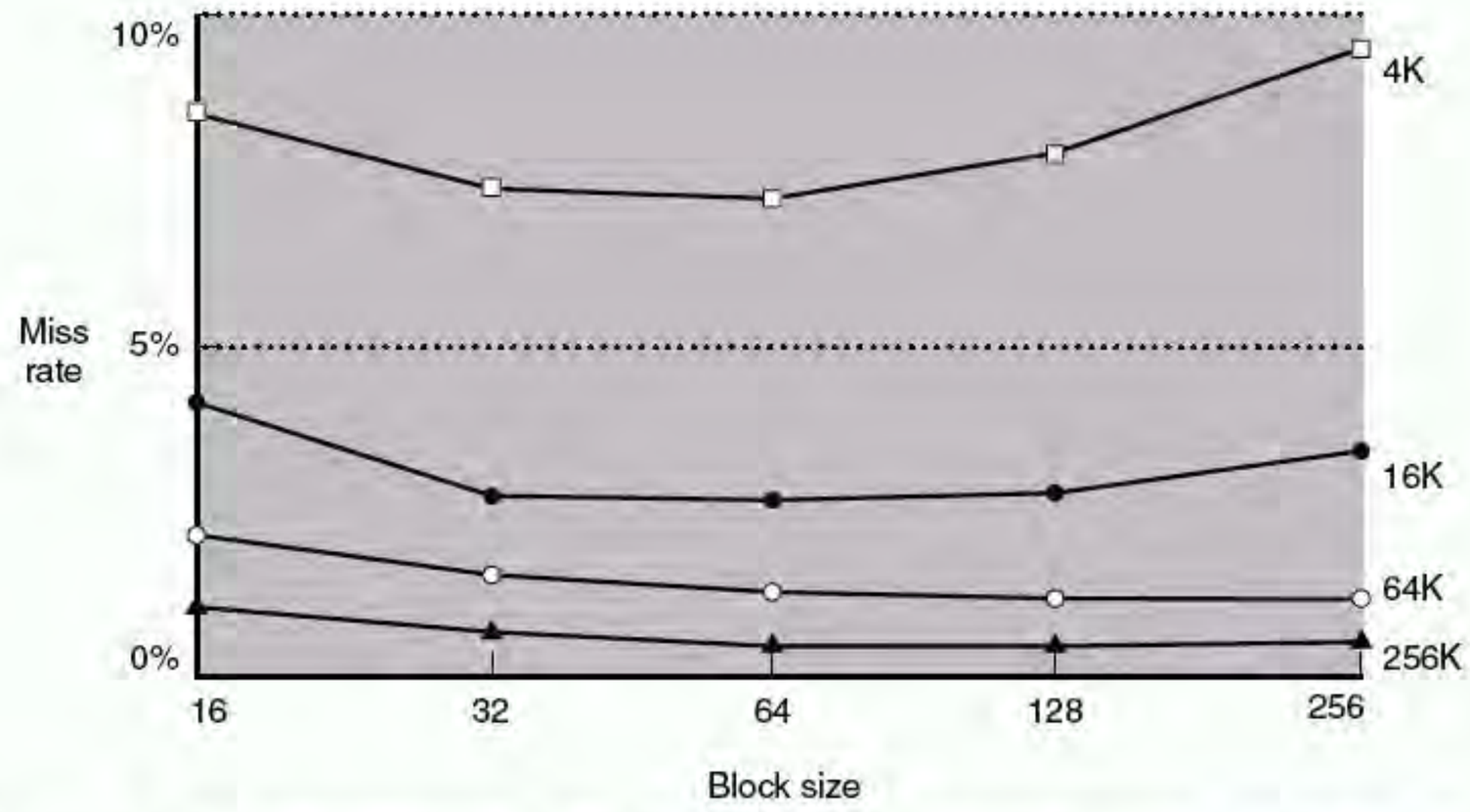


FIGURE 7.8 Miss rate versus block size. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.) Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92.

A more serious problem associated with just increasing the block size is that the cost of a miss increases. The miss penalty is determined by the time required to fetch the block from the next lower level of the hierarchy and load it into the cache. The time to fetch the block has two parts: the latency to the first word and the transfer time for the rest of the block. Clearly, unless we change the memory system, the transfer time—and hence the miss penalty—will increase as the block size grows. Furthermore, the improvement in the miss rate starts to decrease as the blocks become larger. The result is that the increase in the miss penalty overwhelms the decrease in the miss rate for large blocks, and cache performance thus decreases. Of course, if we design the memory to transfer larger blocks more efficiently, we can increase the block size and obtain further improvements in cache performance. We discuss this topic in the next section.

Elaboration: The major disadvantage of increasing the block size is that the cache miss penalty increases. Although it is hard to do anything about the latency component of the miss penalty, we may be able to hide some of the transfer time so that the miss penalty is effectively smaller. The simplest method for doing this, called *early restart*, is simply to resume execution as soon as the requested word of the block is returned, rather than wait for the entire block. Many processors use this technique for instruction

access, where it works best. Instruction accesses are largely sequential, so if the memory system can deliver a word every clock cycle, the processor may be able to restart operation when the requested word is returned, with the memory system delivering new instruction words just in time. This technique is usually less effective for data caches because it is likely that the words will be requested from the block in a less predictable way, and the probability that the processor will need another word from a different cache block before the transfer completes is high. If the processor cannot access the data cache because a transfer is ongoing, then it must stall.

An even more sophisticated scheme is to organize the memory so that the requested word is transferred from the memory to the cache first. The remainder of the block is then transferred, starting with the address after the requested word and wrapping around to the beginning of the block. This technique, called *requested word first*, or *critical word first*, can be slightly faster than early restart, but it is limited by the same properties that limit early restart.

Handling Cache Misses

cache miss A request for data from the cache that cannot be filled because the data is not present in the cache.

Before we look at the cache of a real system, let's see how the control unit deals with **cache misses**. The control unit must detect a miss and process the miss by fetching the requested data from memory (or, as we shall see, a lower-level cache). If the cache reports a hit, the computer continues using the data as if nothing had happened. Consequently, we can use the same basic control that we developed in Chapter 5 and enhanced to accommodate pipelining in Chapter 6. The memories in the datapath in Chapters 5 and 6 are simply replaced by caches.

Modifying the control of a processor to handle a hit is trivial; misses, however, require some extra work. The cache miss handling is done with the processor control unit and with a separate controller that initiates the memory access and refills the cache. The processing of a cache miss creates a stall, similar to the pipeline stalls discussed in Chapter 6, as opposed to an interrupt, which would require saving the state of all registers. For a cache miss, we can stall the entire processor, essentially freezing the contents of the temporary and programmer-visible registers, while we wait for memory. In contrast, pipeline stalls, discussed in Chapter 6, are more complex because we must continue executing some instructions while we stall others.

Let's look a little more closely at how instruction misses are handled for either the multicycle or pipelined datapath; the same approach can be easily extended to handle data misses. If an instruction access results in a miss, then the content of the Instruction register is invalid. To get the proper instruction into the cache, we must be able to instruct the lower level in the memory hierarchy to perform a read. Since the program counter is incremented in the first clock cycle of execution in both the pipelined and multicycle processors, the address of the instruction that generates an instruction cache miss is equal to the value of the program counter minus 4. Once we have the address, we need to instruct the main memory

to perform a read. We wait for the memory to respond (since the access will take multiple cycles), and then write the words into the cache.

We can now define the steps to be taken on an instruction cache miss:

1. Send the original PC value (current PC – 4) to the memory.
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.
4. Restart the instruction execution at the first step, which will refetch the instruction, this time finding it in the cache.

The control of the cache on a data access is essentially identical: on a miss, we simply stall the processor until the memory responds with the data.

Handling Writes

Writes work somewhat differently. Suppose on a store instruction, we wrote the data into only the data cache (without changing main memory); then, after the write into the cache, memory would have a different value from that in the cache. In such a case, the cache and memory are said to be *inconsistent*. The simplest way to keep the main memory and the cache consistent is to always write the data into both the memory and the cache. This scheme is called **write-through**.

write-through A scheme in which writes always update both the cache and the memory, ensuring that data is always consistent between the two.

The other key aspect of writes is what occurs on a write miss. We first fetch the words of the block from memory. After the block is fetched and placed into the cache, we can overwrite the word that caused the miss into the cache block. We also write the word to main memory using the full address.

Although this design handles writes very simply, it would not provide very good performance. With a write-through scheme, every write causes the data to be written to main memory. These writes will take a long time, likely at least 100 processor clock cycles, and could slow down the processor considerably. For the SPEC2000 integer benchmarks, for example, 10% of the instructions are stores. If the CPI without cache misses was 1.0, spending 100 extra cycles on every write would lead to a CPI of $1.0 + 100 \times 10\% = 11$, reducing performance by more than a factor of 10.

One solution to this problem is to use a **write buffer**. A write buffer stores the data while it is waiting to be written to memory. After writing the data into the cache and into the write buffer, the processor can continue execution. When a write to main memory completes, the entry in the write buffer is freed. If the write buffer is full when the processor reaches a write, the processor must stall until there is an empty position in the write buffer. Of course, if the rate at which the

write buffer A queue that holds data while the data are waiting to be written to memory.

memory can complete writes is less than the rate at which the processor is generating writes, no amount of buffering can help because writes are being generated faster than the memory system can accept them.

The rate at which writes are generated may also be *less* than the rate at which the memory can accept them, and yet stalls may still occur. This can happen when the writes occur in bursts. To reduce the occurrence of such stalls, processors usually increase the depth of the write buffer beyond a single entry.

write-back A scheme that handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced.

The alternative to a write-through scheme is a scheme called write-back. In a write-back scheme, when a write occurs, the new value is written only to the block in the cache. The modified block is written to the lower level of the hierarchy when it is replaced. Write-back schemes can improve performance, especially when processors can generate writes as fast or faster than the writes can be handled by main memory; a write-back scheme is, however, more complex to implement than write-through.

In the rest of this section, we describe caches from real processors, and we examine how they handle both reads and writes. In Section 7.5, we will describe the handling of writes in more detail.

Elaboration: Writes introduce several complications into caches that are not present for reads. Here we discuss two of them: the policy on write misses and efficient implementation of writes in write-back caches.

Consider a miss in a write-through cache. The strategy followed in most write-through cache designs, called *fetch-on-miss*, *fetch-on-write*, or sometimes *allocate-on-miss*, allocates a cache block to the address that missed and fetches the rest of the block into the cache before writing the data and continuing execution. Alternatively, we could either allocate the block in the cache but not fetch the data (called *no-fetch-on-write*), or even not allocate the block (called *no-allocate-on-write*). Another name for these strategies that do not place the written data into the cache is *write-around*, since the data is written around the cache to get to memory. The motivation for these schemes is the observation that sometimes programs write entire blocks of data before reading them. In such cases, the fetch associated with the initial write miss may be unnecessary. There are a number of subtle issues involved in implementing these schemes in multiword blocks, including complicating the handling of write hits by requiring mechanisms similar to those used for write-back caches.

Actually implementing stores efficiently in a cache that uses a write-back strategy is more complex than in a write-through cache. In a write-back cache, we must write the block back to memory if the data in the cache is dirty and we have a cache miss. If we simply overwrote the block on a store instruction before we knew whether the store had hit in the cache (as we could for a write-through cache), we would destroy the contents of the block, which is not backed up in memory. A write-through cache can write the data into the cache and read the tag; if the tag mismatches, then a miss occurs. Because the cache is write-through, the overwriting of the block in the cache is not catastrophic since memory has the correct value.

In a write-back cache, because we cannot overwrite the block, stores either require two cycles (a cycle to check for a hit followed by a cycle to actually perform the write) or require an extra buffer, called a *store buffer*, to hold that data—effectively allowing the store to take only one cycle by pipelining it. When a store buffer is used, the processor does the cache lookup and places the data in the store buffer during the normal cache access cycle. Assuming a cache hit, the new data is written from the store buffer into the cache on the next unused cache access cycle.

By comparison, in a write-through cache, writes can always be done in one cycle. There are some extra complications with multiword blocks, however, since we cannot simply overwrite the tag when we write the data. Instead, we read the tag and write the data portion of the selected block. If the tag matches the address of the block being written, the processor can continue normally, since the correct block has been updated. If the tag does not match, the processor generates a write miss to fetch the rest of the block corresponding to that address. Because it is always safe to overwrite the data, write hits still take one cycle.

Many write-back caches also include write buffers that are used to reduce the miss penalty when a miss replaces a dirty block. In such a case, the dirty block is moved to a write-back buffer associated with the cache while the requested block is read from memory. The write-back buffer is later written back to memory. Assuming another miss does not occur immediately, this technique halves the miss penalty when a dirty block must be replaced.

An Example Cache: The Intrinsity FastMATH processor

The Intrinsity FastMATH is a fast embedded microprocessor that uses the MIPS architecture and a simple cache implementation. Near the end of the chapter, we will examine the more complex cache design of the Intel Pentium 4, but we start with this simple, yet real, example for pedagogical reasons. Figure 7.9 shows the organization of the Intrinsity FastMATH data cache.

This processor has 12-stage pipeline, similar to that discussed in Chapter 6. When operating at peak speed, the processor can request both an instruction word and a data word on every clock. To satisfy the demands of the pipeline without stalling, separate instruction and data caches are used. Each cache is 16 KB, or 4K words, with 16-word blocks.

Read requests for the cache are straightforward. Because there are separate data and instruction caches, separate control signals will be needed to read and write each cache. (Remember that we need to update the instruction cache when a miss occurs.) Thus, the steps for a read request to either cache are as follows:

1. Send the address to the appropriate cache. The address comes either from the PC (for an instruction) or from the ALU (for data).
2. If the cache signals hit, the requested word is available on the data lines. Since there are 16 words in the desired block, we need to select the right

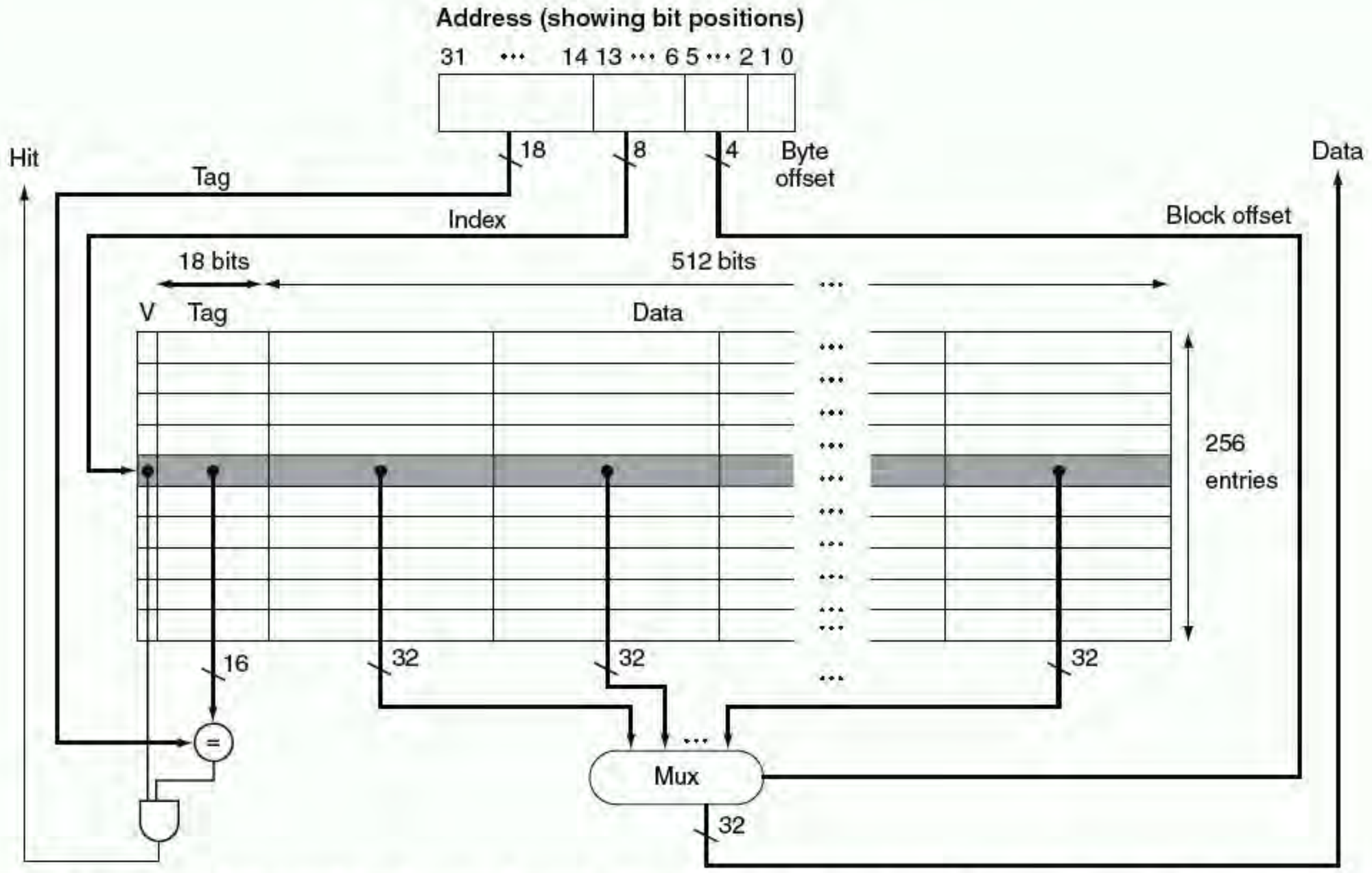


FIGURE 7.9 The 16 KB caches in the Intrinsity FastMATH each contain 256 blocks with 16 words per block. The tag field is 18 bits wide and the index field is 8 bits wide, while a 4-bit field (bits 5–2) is used to index the block and select the word from the block using a 16-to-1 multiplexor. In practice, to eliminate the multiplexor, caches use a separate large RAM for the data and a smaller RAM for the tags, with the block offset supplying the extra address bits for the large data RAM. In this case, the large RAM is 32 bits wide and must have 16 times as many words as blocks in the cache.

one. A block index field is used to control the multiplexor (shown at the bottom of the figure), which selects the requested word from the 16 words in the indexed block.

3. If the cache signals miss, we send the address to the main memory. When the memory returns with the data, we write it into the cache and then read it to fulfill the request.

For writes, the Intrinsity FastMATH offers both write-through and write-back, leaving it up to the operating system to decide which strategy to use for an application. It has a one-entry write buffer.

Instruction miss rate	Data miss rate	Effective combined miss rate
0.4%	11.4%	3.2%

FIGURE 7.10 Approximate instruction and data miss rates for the Intrinsity FastMATH processor for SPEC2000 benchmarks. The combined miss rate is the effective miss rate seen for the combination of the 16 KB instruction cache and 16 KB data cache. It is obtained by weighting the instruction and data individual miss rates by the frequency of instruction and data references.

What cache miss rates are attained with a cache structure like that used by the Intrinsity FastMATH? Figure 7.10 shows the miss rates for the instruction and data caches for the SPEC2000 integer benchmarks. The combined miss rate is the effective miss rate per reference for each program after accounting for the differing frequency of instruction and data accesses.

Although miss rate is an important characteristic of cache designs, the ultimate measure will be the effect of the memory system on program execution time; we'll see how miss rate and execution time are related shortly.

Elaboration: A combined cache with a total size equal to the sum of the two split caches will usually have a better hit rate. This higher rate occurs because the combined cache does not rigidly divide the number of entries that may be used by instructions from those that may be used by data. Nonetheless, many processors use a split instruction and data cache to increase cache *bandwidth*.

Here are miss rates for caches the size of those found in the Intrinsity FastMATH processor, and for a combined cache whose size is equal to the total of the two caches:

- Total cache size: 32 KB
- Split cache effective miss rate: 3.24%
- Combined cache miss rate: 3.18%

The miss rate of the split cache is only slightly worse.

The advantage of doubling the cache bandwidth, by supporting both an instruction and data access simultaneously, easily overcomes the disadvantage of a slightly increased miss rate. This observation is another reminder that we cannot use miss rate as the sole measure of cache performance, as Section 7.3 shows.

split cache A scheme in which a level of the memory hierarchy is composed of two independent caches that operate in parallel with each other with one handling instructions and one handling data.

Designing the Memory System to Support Caches

Cache misses are satisfied from main memory, which is constructed from DRAMs. In Section 7.1, we saw that DRAMs are designed with the primary emphasis on density rather than access time. Although it is difficult to reduce the latency to fetch the first word from memory, we can reduce the miss penalty if we increase the bandwidth from the memory to the cache. This reduction allows

larger block sizes to be used while still maintaining a low miss penalty, similar to that for a smaller block.

The processor is typically connected to memory over a bus. The clock rate of the bus is usually much slower than the processor, by as much as a factor of 10. The speed of this bus affects the miss penalty.

To understand the impact of different organizations of memory, let's define a set of hypothetical memory access times. Assume

- 1 memory bus clock cycle to send the address
- 15 memory bus clock cycles for each DRAM access initiated
- 1 memory bus clock cycle to send a word of data

If we have a cache block of four words and a one-word-wide bank of DRAMs, the miss penalty would be $1 + 4 \times 15 + 4 \times 1 = 65$ memory bus clock cycles. Thus, the number of bytes transferred per bus clock cycle for a single miss would be

$$\frac{4 \times 4}{65} = 0.25$$

Figure 7.11 shows three options for designing the memory system. The first option follows what we have been assuming: memory is one word wide, and all accesses are made sequentially. The second option increases the bandwidth to memory by widening the memory and the buses between the processor and memory; this allows parallel access to all the words of the block. The third option increases the bandwidth by widening the memory but not the interconnection bus. Thus, we still pay a cost to transmit each word, but we can avoid paying the cost of the access latency more than once. Let's look at how much these other two options improve the 65-cycle miss penalty that we would see for the first option (Figure 7.11a).

Increasing the width of the memory and the bus will increase the memory bandwidth proportionally, decreasing both the access time and transfer time portions of the miss penalty. With a main memory width of two words, the miss penalty drops from 65 memory bus clock cycles to $1 + 2 \times 15 + 2 \times 1 = 33$ memory bus clock cycles. With a four-word-wide memory, the miss penalty is just 17 memory bus clock cycles. The bandwidth for a single miss is then 0.48 (almost twice as high) bytes per bus clock cycle for a memory that is two words wide, and 0.94 bytes per bus clock cycle when the memory is four words wide (almost four times higher). The major costs of this enhancement are the wider bus and the potential increase in cache access time due to the multiplexor and control logic between the processor and cache.

Instead of making the entire path between the memory and cache wider, the memory chips can be organized in banks to read or write multiple words in one

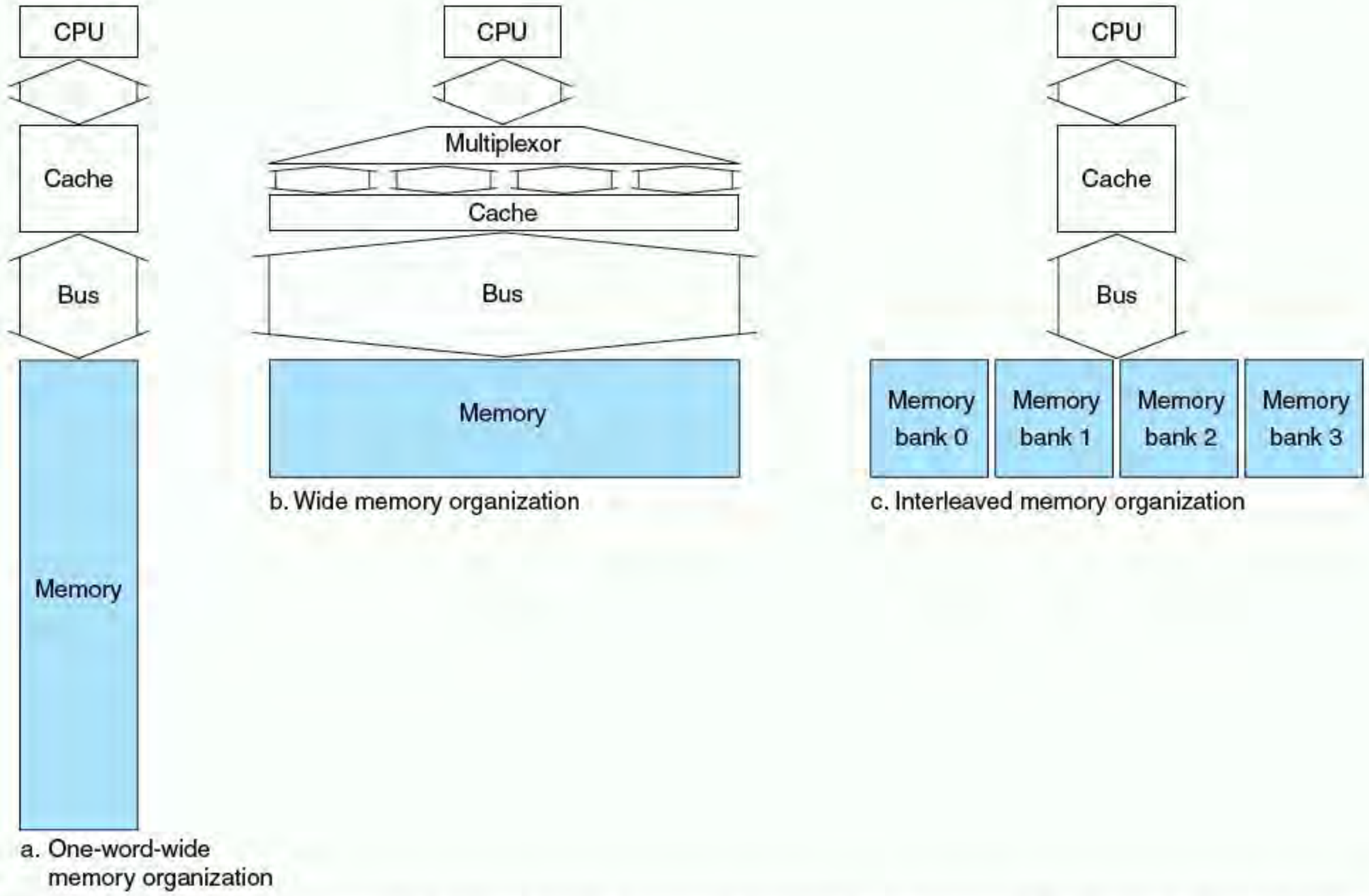


FIGURE 7.11 The primary method of achieving higher memory bandwidth is to increase the physical or logical width of the memory system. In this figure, memory bandwidth is improved two ways. The simplest design, (a), uses a memory where all components are one word wide; (b) shows a wider memory, bus, and cache; while (c) shows a narrow bus and cache with an interleaved memory. In (b), the logic between the cache and processor consists of a multiplexor used on reads and control logic to update the appropriate words of the cache on writes.

access time rather than reading or writing a single word each time. Each bank could be one word wide so that the width of the bus and the cache need not change, but sending an address to several banks permits them all to read simultaneously. This scheme, which is called *interleaving*, retains the advantage of incurring the full memory latency only once. For example, with four banks, the time to get a four-word block would consist of 1 cycle to transmit the address and read request to the banks, 15 cycles for all four banks to access memory, and 4 cycles to send the four words back to the cache. This yields a miss penalty of $1 + 1 \times 15 + 4 \times 1 = 20$ memory bus clock cycles. This is an effective bandwidth per miss of 0.80 bytes per clock, or about three times the bandwidth for the one-word-wide mem-

ory and bus. Banks are also valuable on writes. Each bank can write independently, quadrupling the write bandwidth and leading to fewer stalls in a write-through cache. As we will see, an alternative strategy for writes makes interleaving even more attractive.

Elaboration: Memory chips are organized to produce a number of output bits, usually 4 to 32, with 8 or 16 being the most popular in 2004. We describe the organization of a RAM as $d \times w$, where d is the number of addressable locations (the depth) and w is the output (or width of each location). One path to improving the rate at which we transfer data from the memory to the caches is to take advantage of the structure of DRAMs. DRAMs are logically organized as rectangular arrays, and access time is divided into row access and column access. DRAMs buffer a row of bits inside the DRAM for column access. They also come with optional timing signals that allow repeated accesses to the buffer without a row access time. This capability, originally called *page mode*, has gone through a series of enhancements. In page mode, the buffer acts like an SRAM; by changing column address, random bits can be accessed in the buffer until the next row access. This capability changes the access time significantly, since the access time to bits in the row is much lower. Figure 7.12 shows how the density, cost, and access time of DRAMS have changed over the years.

The newest development is DDR SDRAMs (double data rate synchronous DRAMs). SDRAMs provide for a burst access to data from a series of sequential locations in the DRAM. An SDRAM is supplied with a starting address and a burst length. The data in the burst is transferred under control of a clock signal, which in 2004 can run at up to

Year introduced	Chip size	$ per MB	Total access time to a new row/column	Column access time to existing row
1980	64 Kbit	$1500	250 ns	150 ns
1983	256 Kbit	$500	185 ns	100 ns
1985	1 Mbit	$200	135 ns	40 ns
1989	4 Mbit	$50	110 ns	40 ns
1992	16 Mbit	$15	90 ns	30 ns
1996	64 Mbit	$10	60 ns	12 ns
1998	128 Mbit	$4	60 ns	10 ns
2000	256 Mbit	$1	55 ns	7 ns
2002	512 Mbit	$0.25	50 ns	5 ns
2004	1024 Mbit	$0.10	45 ns	3 ns

FIGURE 7.12 DRAM size increased by multiples of four approximately once every three years until 1996, and thereafter doubling approximately every two years. The improvements in access time have been slower but continuous, and cost almost tracks density improvements, although cost is often affected by other issues, such as availability and demand. The cost per megabyte is not adjusted for inflation.

300 MHz. The two key advantages of SDRAMs are the use of a clock that eliminates the need to synchronize and the elimination of the need to supply successive addresses in the burst. The DDR part of the name means data transfers on both the leading and falling edge of the clock, thereby getting twice as much bandwidth as you might expect based on the clock rate and the data width. To deliver such high bandwidth, the internal DRAM is organized as interleaved memory banks.

The advantage of these optimizations is that they use the circuitry already largely on the DRAMs, adding little cost to the system while achieving a significant improvement in bandwidth. The internal architecture of DRAMs and how these optimizations are implemented are described in Section B.8 of Appendix B.

Summary

We began the previous section by examining the simplest of caches: a direct-mapped cache with a one-word block. In such a cache, both hits and misses are simple, since a word can go in exactly one location and there is a separate tag for every word. To keep the cache and memory consistent, a write-through scheme can be used, so that every write into the cache also causes memory to be updated. The alternative to write-through is a write-back scheme that copies a block back to memory when it is replaced; we'll discuss this scheme further in upcoming sections.

To take advantage of spatial locality, a cache must have a block size larger than one word. The use of a larger block decreases the miss rate and improves the efficiency of the cache by reducing the amount of tag storage relative to the amount of data storage in the cache. Although a larger block size decreases the miss rate, it can also increase the miss penalty. If the miss penalty increased linearly with the block size, larger blocks could easily lead to lower performance. To avoid this, the bandwidth of main memory is increased to transfer cache blocks more efficiently. The two common methods for doing this are making the memory wider and interleaving. In both cases, we reduce the time to fetch the block by minimizing the number of times we must start a new memory access to fetch a block, and, with a wider bus, we can also decrease the time needed to send the block from the memory to the cache.

Check Yourself

The speed of the memory system affects the designer's decision on the size of the cache block. Which of the following cache designer guidelines are generally valid?

1. The shorter the memory latency, the smaller the cache block.
2. The shorter the memory latency, the larger the cache block.
3. The higher the memory bandwidth, the smaller the cache block.
4. The higher the memory bandwidth, the larger the cache block.

7.3 Measuring and Improving Cache Performance

In this section, we begin by looking at how to measure and analyze cache performance; we then explore two different techniques for improving cache performance. One focuses on reducing the miss rate by reducing the probability that two different memory blocks will contend for the same cache location. The second technique reduces the miss penalty by adding an additional level to the hierarchy. This technique, called *multilevel caching*, first appeared in high-end computers selling for over $100,000 in 1990; since then it has become common on desktop computers selling for less than $1000!

CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU spends waiting for the memory system. Normally, we assume that the costs of cache accesses that are hits are part of the normal CPU execution cycles. Thus,

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory-stall clock cycles}) \times \text{Clock cycle time}$$

The memory-stall clock cycles come primarily from cache misses, and we make that assumption here. We also restrict the discussion to a simplified model of the memory system. In real processors, the stalls generated by reads and writes can be quite complex, and accurate performance prediction usually requires very detailed simulations of the processor and memory system.

Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from writes:

$$\text{Memory-stall clock cycles} = \text{Read-stall cycles} + \text{Write-stall cycles}$$

The read-stall cycles can be defined in terms of the number of read accesses per program, the miss penalty in clock cycles for a read, and the read miss rate:

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

Writes are more complicated. For a write-through scheme, we have two sources of stalls: write misses, which usually require that we fetch the block before continuing the write (see the Elaboration on page 484 for more details on dealing with writes), and write buffer stalls, which occur when the write buffer is full when a write occurs. Thus, the cycles stalled for writes equals the sum of these two:

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty}\right) + \text{Write buffer stalls}$$

Because the write buffer stalls depend on the timing of writes, and not just the frequency, it is not possible to give a simple equation to compute such stalls. Fortunately, in systems with a reasonable write buffer depth (e.g., four or more words) and a memory capable of accepting writes at a rate that significantly exceeds the average write frequency in programs (e.g., by a factor of two), the write buffer stalls will be small, and we can safely ignore them. If a system did not meet these criteria, it would not be well designed; instead, the designer should have used either a deeper write buffer or a write-back organization.

Write-back schemes also have potential additional stalls arising from the need to write a cache block back to memory when the block is replaced. We will discuss this more in Section 7.5.

In most write-through cache organizations, the read and write miss penalties are the same (the time to fetch the block from memory). If we assume that the write buffer stalls are negligible, we can combine the reads and writes by using a single miss rate and the miss penalty:

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

We can also factor this as

$$\text{Memory-stall clock cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Let's consider a simple example to help us understand the impact of cache performance on processor performance.

Calculating Cache Performance

EXAMPLE

Assume an instruction cache miss rate for a program is 2% and a data cache miss rate is 4%. If a processor has a CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. Use the instruction frequencies for SPECint2000 from Chapter 3, Figure 3.26, on page 228.

ANSWER

The number of memory miss cycles for instructions in terms of the Instruction count (I) is

$$\text{Instruction miss cycles} = I \times 2\% \times 100 = 2.00 \times I$$

The frequency of all loads and stores in SPECint2000 is 36%. Therefore, we can find the number of memory miss cycles for data references:

$$\text{Data miss cycles} = I \times 36\% \times 4\% \times 100 = 1.44 \times I$$

The total number of memory-stall cycles is 2.00 I + 1.44 I = 3.44 I. This is more than 3 cycles of memory stall per instruction. Accordingly, the CPI with memory stalls is 2 + 3.44 = 5.44. Since there is no change in instruction count or clock rate, the ratio of the CPU execution times is

$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{I \times \text{CPI}_{\text{stall}} \times \text{Clock cycle}}{I \times \text{CPI}_{\text{perfect}} \times \text{Clock cycle}}$$

$$= \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2}$$

The performance with the perfect cache is better by $\frac{5.44}{2} = 2.72$.

What happens if the processor is made faster, but the memory system is not? The amount of time spent on memory stalls will take up an increasing fraction of the execution time; Amdahl's law, which we examined in Chapter 4, reminds us of this fact. A few simple examples show how serious this problem can be. Suppose we speed up the computer in the previous example by reducing its CPI from 2 to 1 without changing the clock rate, which might be done with an improved pipeline. The system with cache misses would then have a CPI of 1 + 3.44 = 4.44, and the system with the perfect cache would be

$$\frac{4.44}{1} = 4.44 \text{ times faster}$$

The amount of execution time spent on memory stalls would have risen from

$$\frac{3.44}{5.44} = 63\%$$

to

$$\frac{3.44}{4.44} = 77\%$$

Similarly, increasing clock rate without changing the memory system also increases the performance lost due to cache misses, as the next example shows.

Cache Performance with Increased Clock Rate

EXAMPLE

Suppose we increase the performance of the computer in the previous example by doubling its clock rate. Since the main memory speed is unlikely to change, assume that the absolute time to handle a cache miss does not change. How much faster will the computer be with the faster clock, assuming the same miss rate as the previous example?

ANSWER

Measured in the faster clock cycles, the new miss penalty will be twice as many clock cycles, or 200 clock cycles. Hence:

$$\text{Total miss cycles per instruction} = (2\% \times 200) + 36\% \times (4\% \times 200) = 6.88$$

Thus, the faster computer with cache misses will have a CPI of 2 + 6.88 = 8.88, compared to a CPI with cache misses of 5.44 for the slower computer.

Using the formula for CPU time from the previous example, we can compute the relative performance as

$$\frac{\text{Performance with fast clock}}{\text{Performance with slow clock}} = \frac{\text{Execution time with slow clock}}{\text{Execution time with fast clock}}$$

$$= \frac{\text{IC} \times \text{CPI}_{\text{slow clock}} \times \text{Clock cycle}}{\text{IC} \times \text{CPI}_{\text{fast clock}} \times \frac{\text{Clock cycle}}{2}}$$

$$= \frac{5.44}{8.88 \times \frac{1}{2}} = 1.23$$

Thus, the computer with the faster clock is about 1.2 times faster rather than 2 times faster, which it would have been if we ignored cache misses.