



# 6

## Enhancing Performance with Pipelining

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

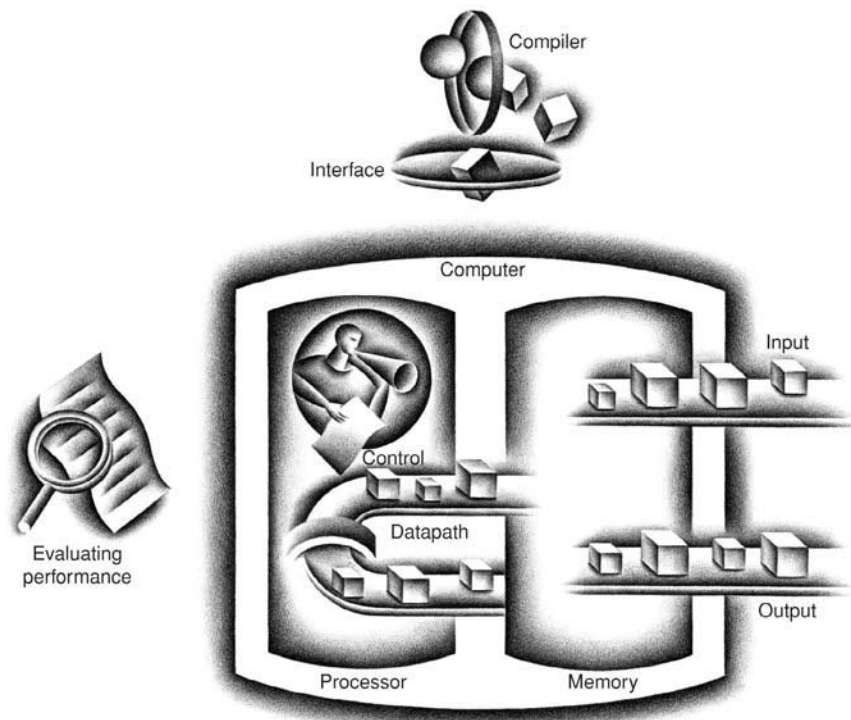
**Robert Herrick**

*Hesperides: Ceremonies for Christmas Eve, 1648*

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

---

## The Five Classic Components of a Computer



*Never waste time.*  
American proverb

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

## 6.1

## An Overview of Pipelining

**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 *non-pipelined* approach to laundry would be

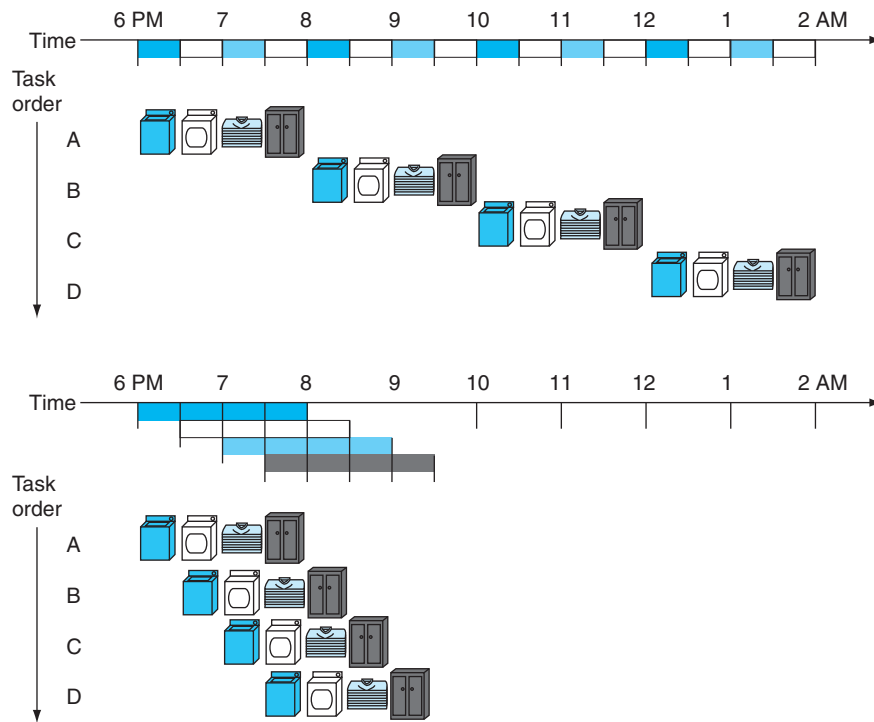
1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

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.

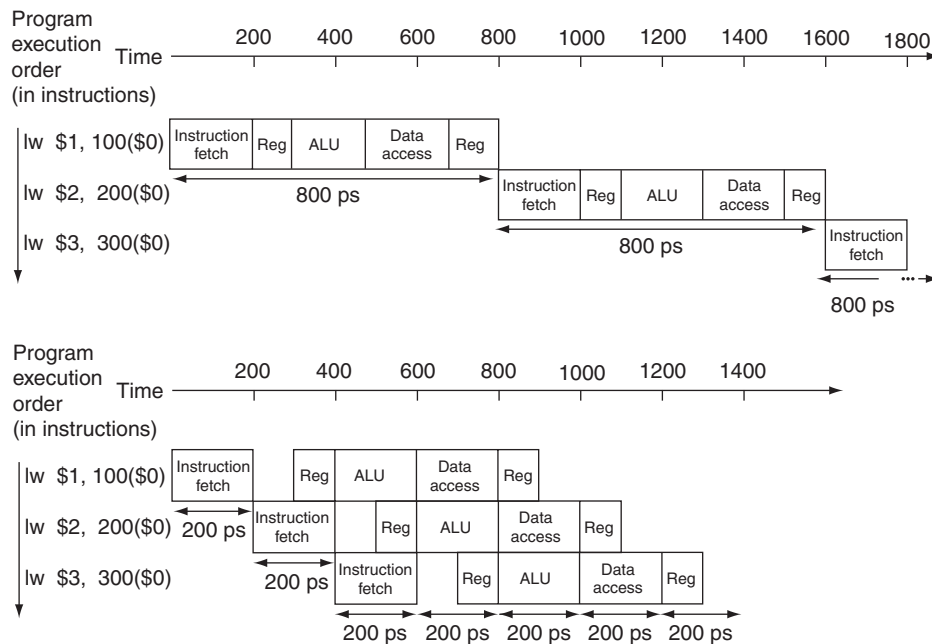
## ANSWER

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  $3 \times 800$  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  $3 \times 200$  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  $1,000,000 \times 200 \text{ ps} + 1400 \text{ ps}$ , or 200,001,400 ps. In the nonpipelined example, we would add 1,000,000 instructions, each taking 800 ps, so total execution time would be  $1,000,000 \times 800 \text{ ps} + 2400 \text{ ps}$ , 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.

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.

**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.



**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.

**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.

## EXAMPLE

## ANSWER

### Data Hazards

**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.

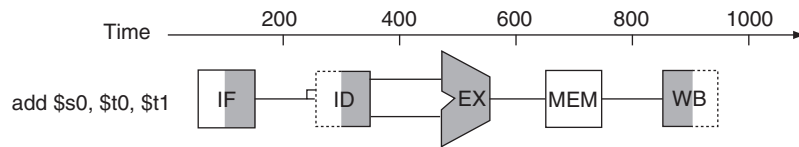
The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve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

### Forwarding with Two Instructions

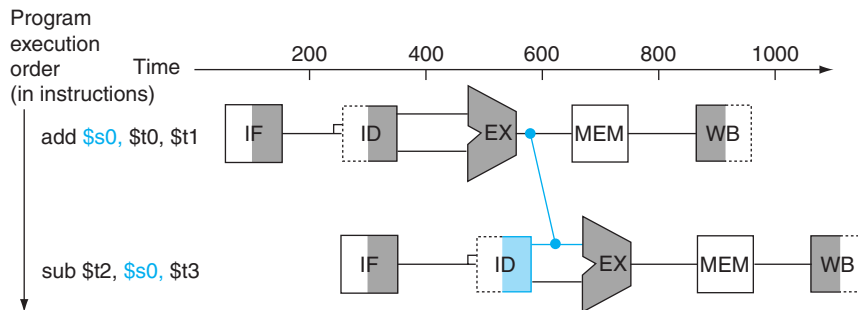
For the two instructions above, show what pipeline stages would be connected by forwarding. Use the drawing in Figure 6.4 to represent the datapath during the five stages of the pipeline. Align a copy of the datapath for each instruction, similar to the laundry pipeline in Figure 6.1.

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 read 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.



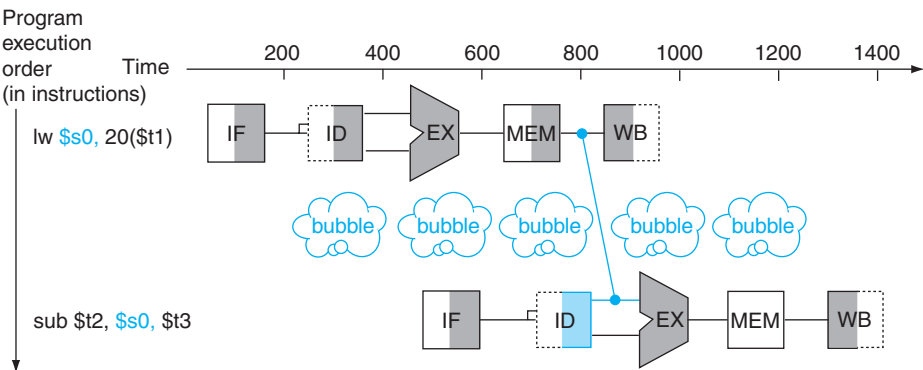
**FIGURE 6.5 Graphical representation of forwarding.** The connection shows the forwarding path from the output of the *EX* stage of *add* to the input of the *EX* stage for *sub*, replacing the value from register *\$s0* read in the second stage of *sub*.

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    $t4, 8($t0)  
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.

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      $t4, 8($t0)
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.

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.

**ANSWER**

**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.

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.

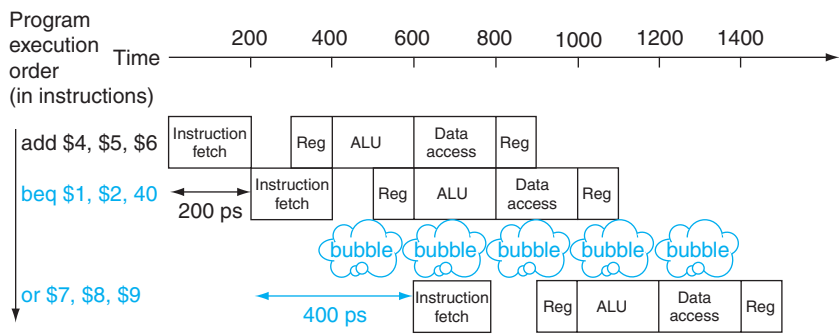
EXAMPLE

ANSWER

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.

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.



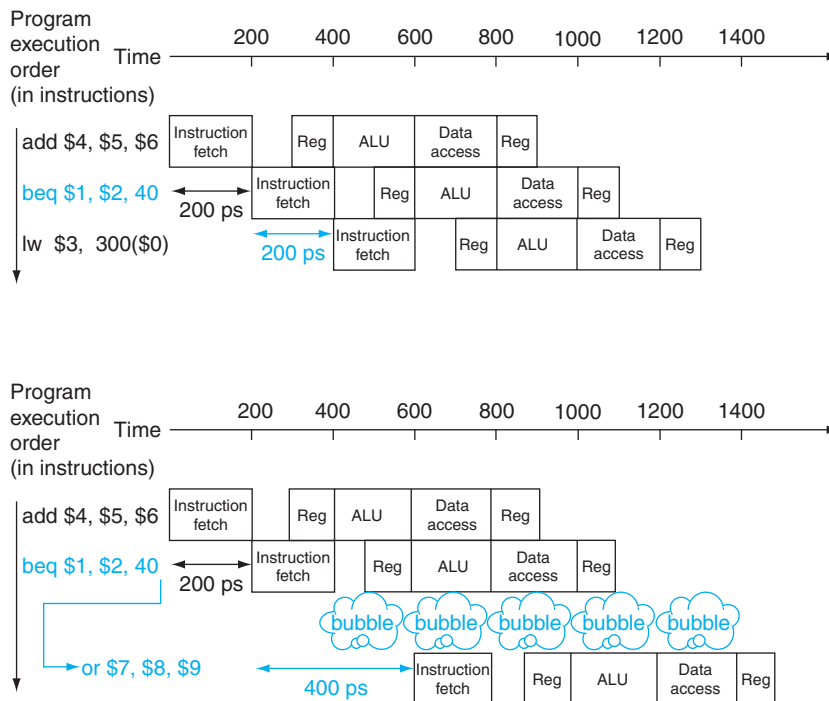
**FIGURE 6.7 Pipeline showing stalling on every conditional branch as solution to control hazards.** There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 6.6. The effect on performance, however,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

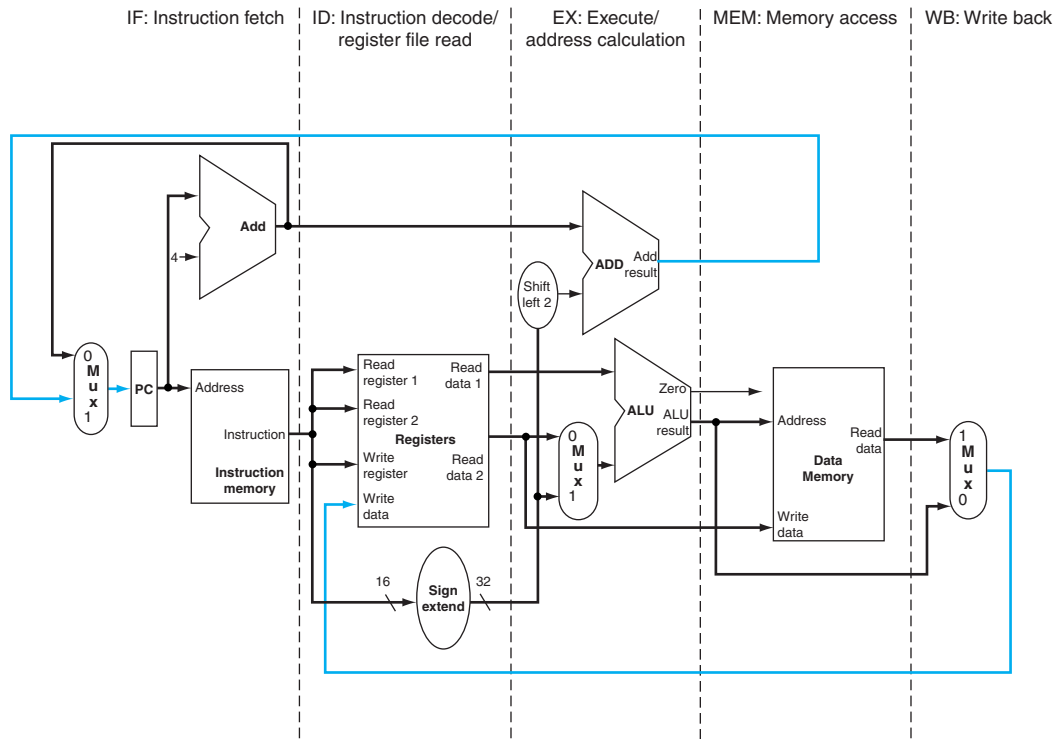
*There is less in this than meets the eye.*

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.

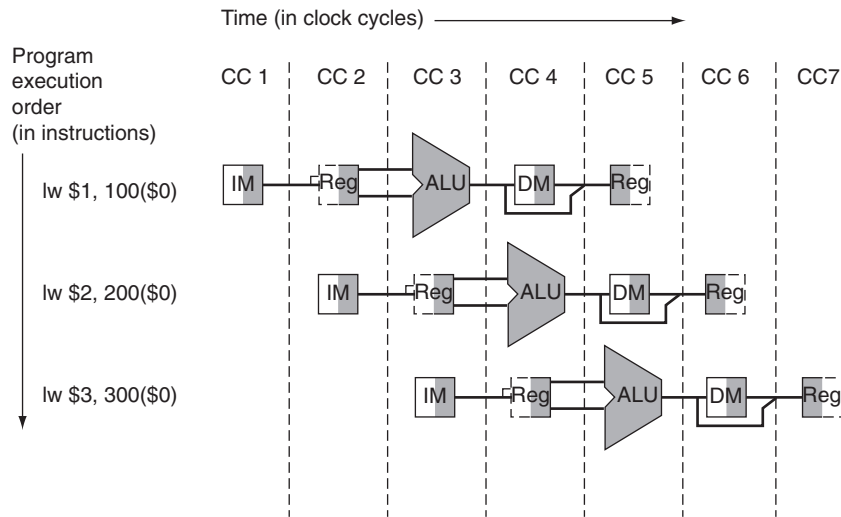
Figure 6.10 seems to suggest that three instructions need three datapaths. In Chapter 5, we added registers to hold data so that portions of the datapath could be shared during instruction execution; we use the same technique here to share the multiple datapaths. For example, as Figure 6.10 shows, the 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-

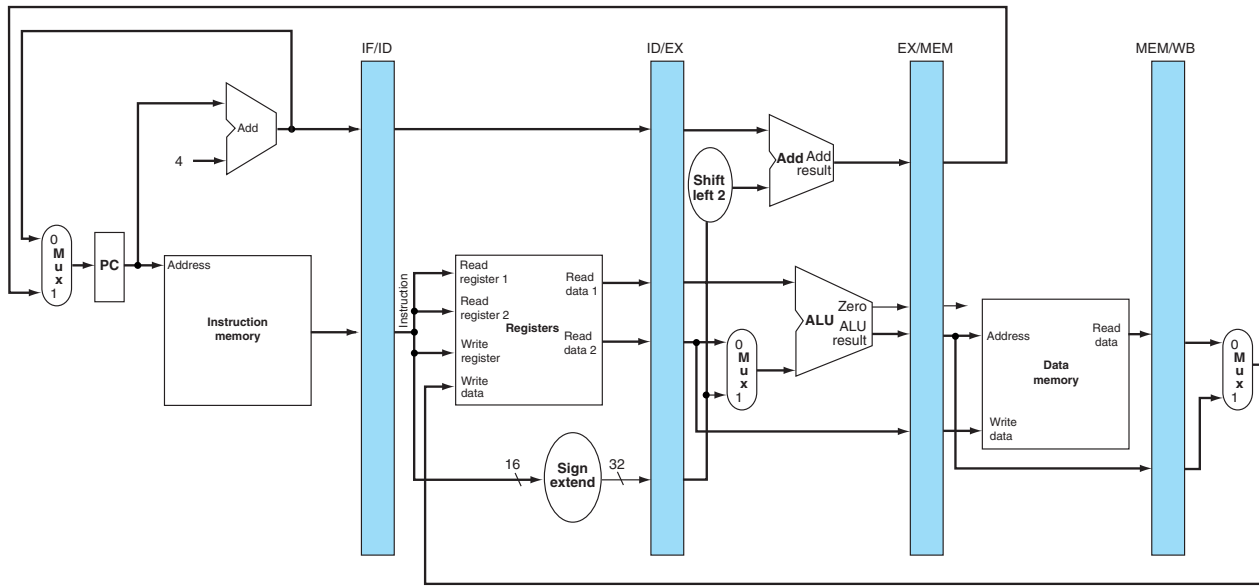


**FIGURE 6.10** Instructions being executed using the single-cycle datapath in Figure 6.9, assuming pipelined execution. Similar to Figures 6.4 through 6.6, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled 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.

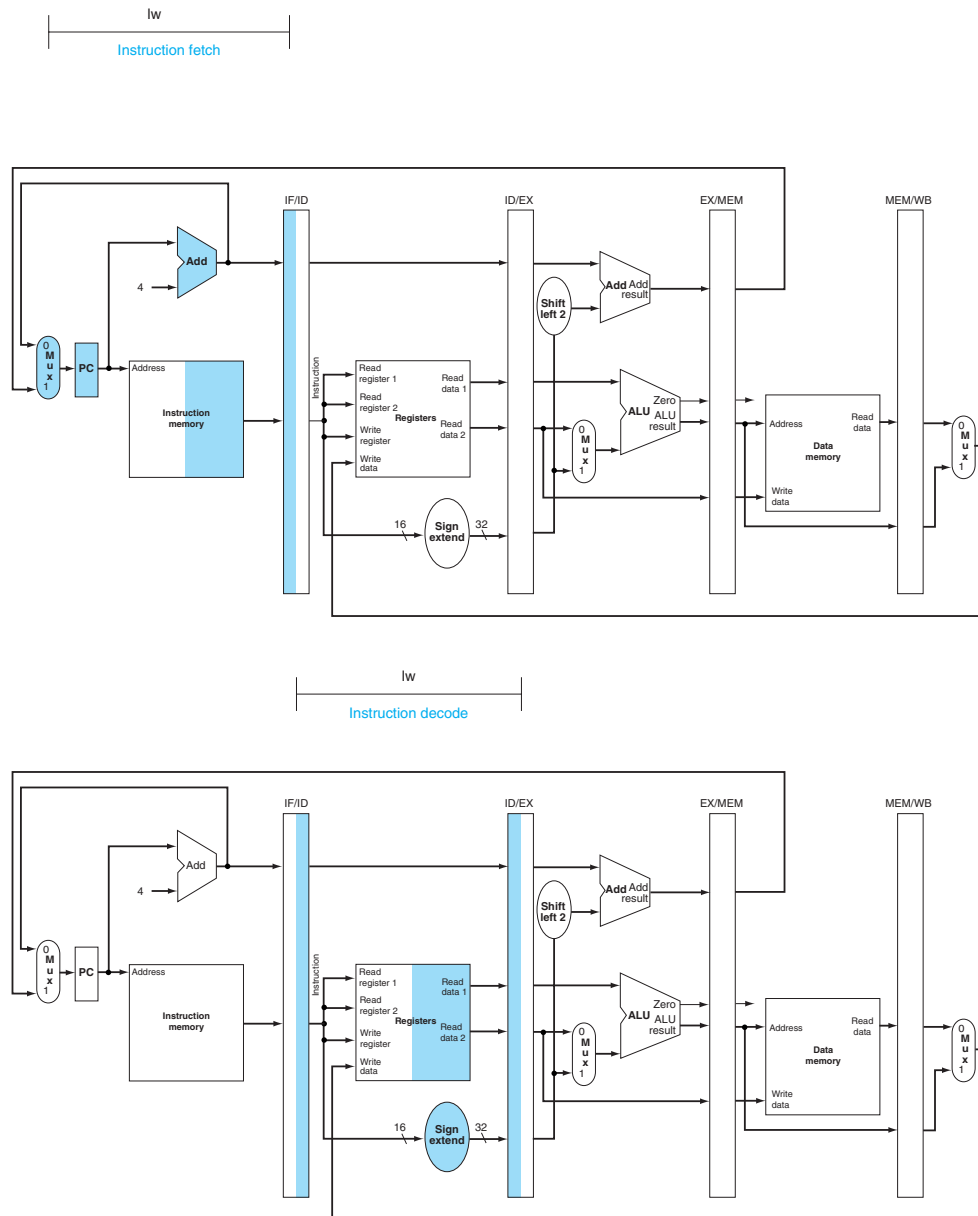
Figures 6.12 through 6.14, our first sequence, show the active portions of the datapath highlighted as a load instruction goes through the five stages of pipelined



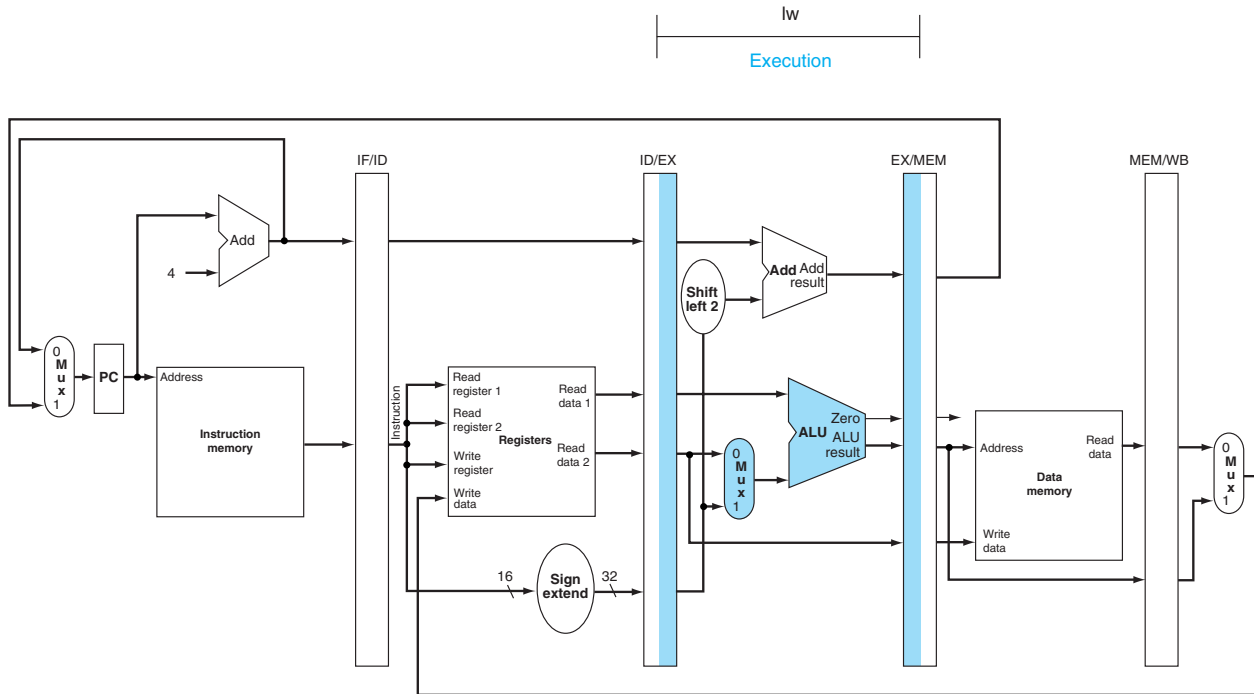
**FIGURE 6.11** The pipelined version of the datapath in Figure 6.9. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The 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  $\text{lw}$  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.

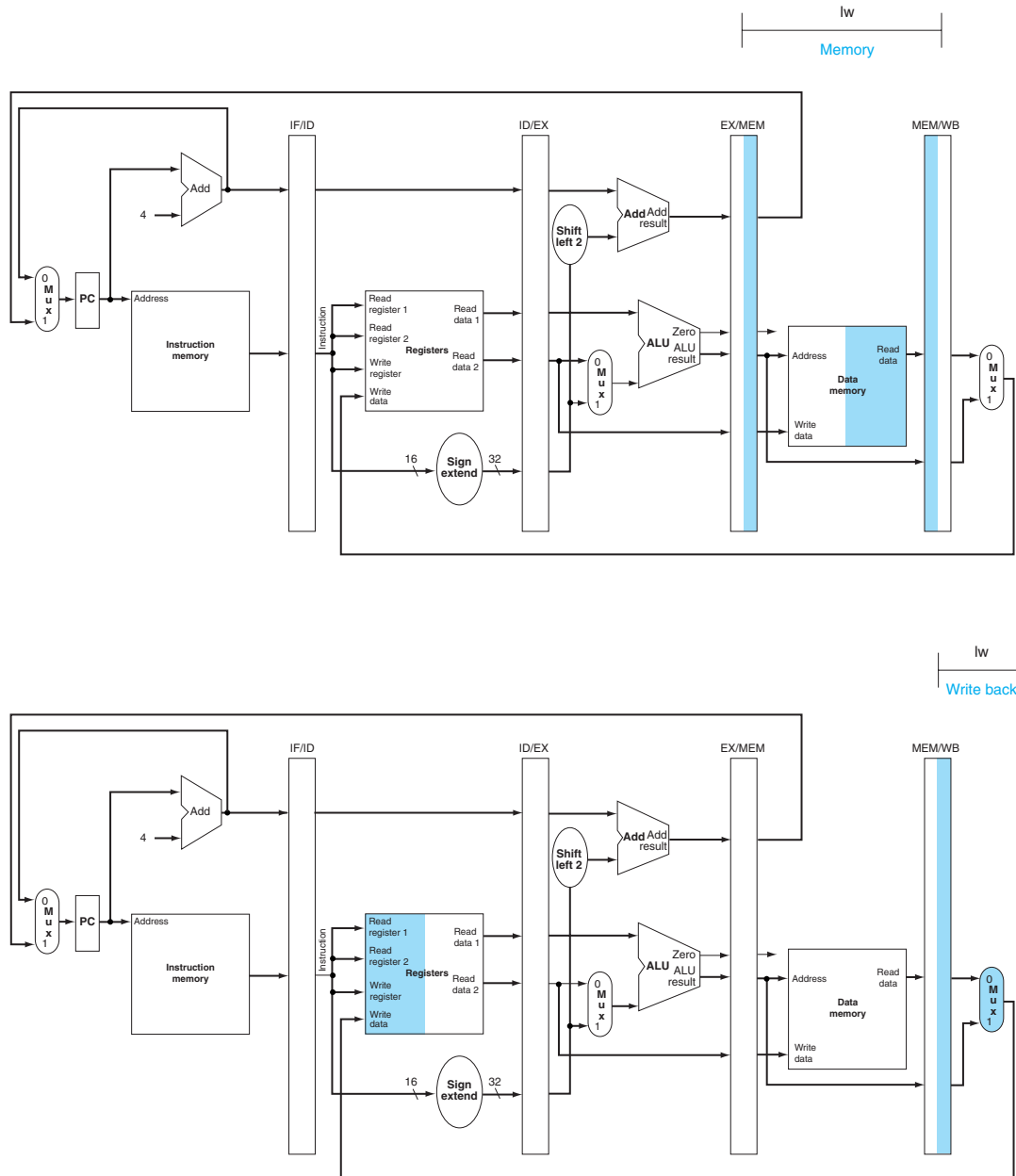


**FIGURE 6.12 IF and ID: first and second pipe stages of an instruction, with the active portions of the datapath in Figure 6.11 highlighted.** The highlighting convention is the same as that used in Figure 6.4. As in Chapter 5, there is no confusion when reading and writing registers because the contents 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.

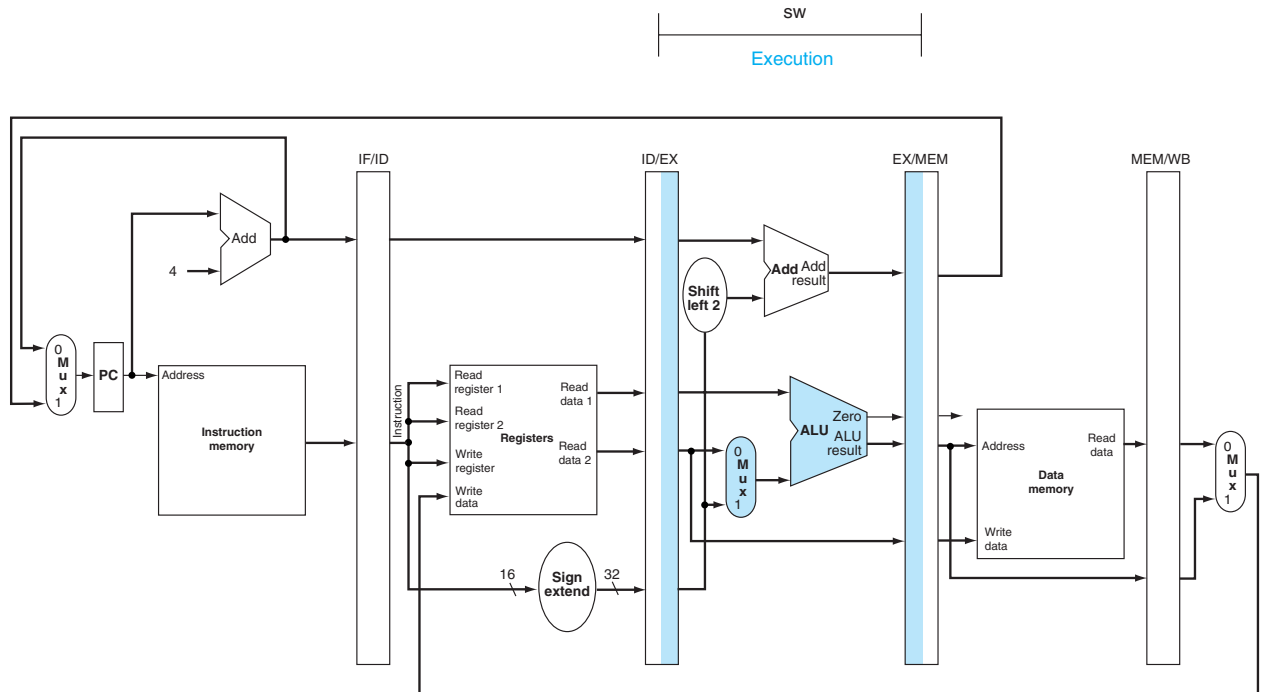


5. *Write back:* The bottom portion of Figure 6.14 shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

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.

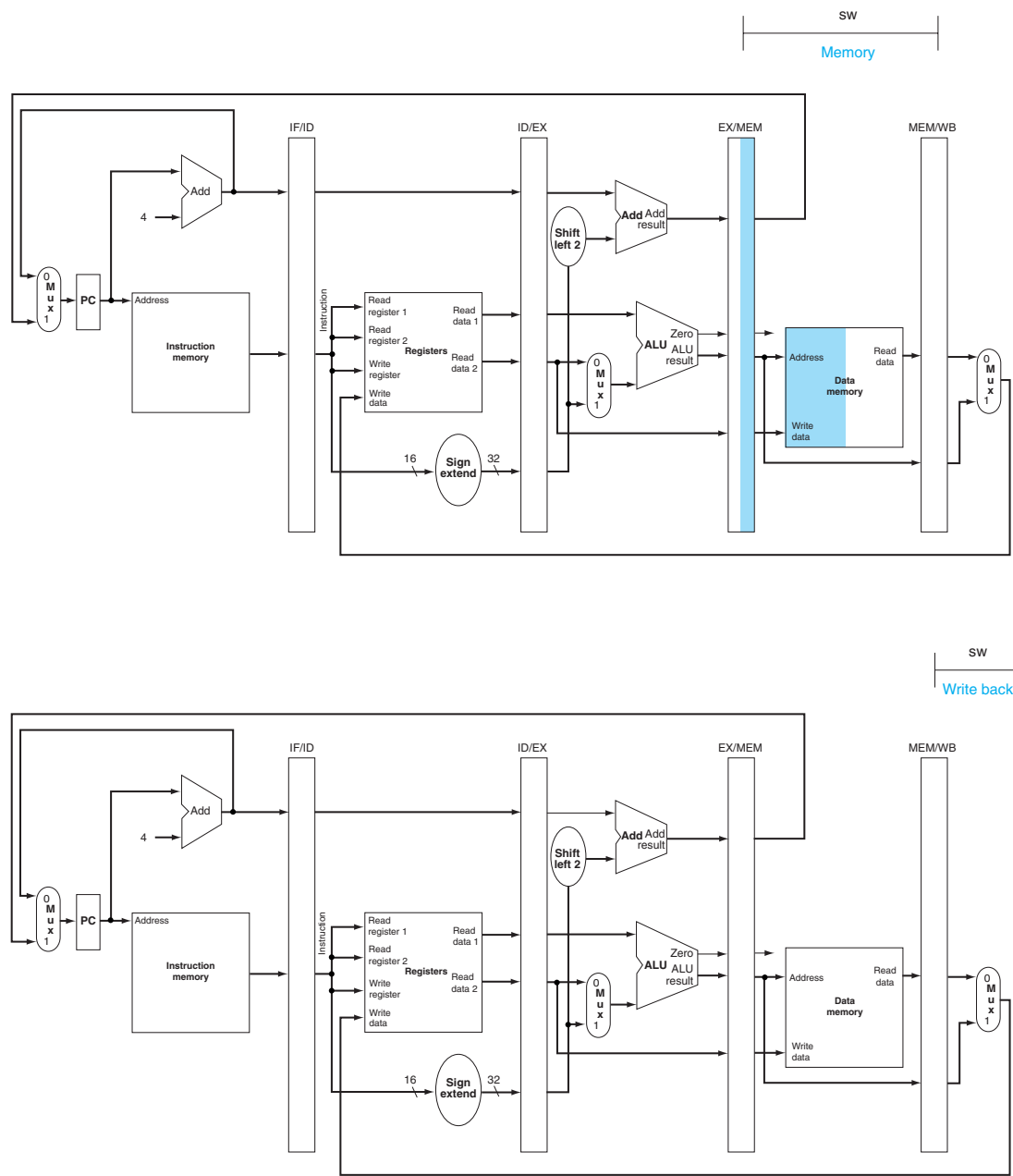


**FIGURE 6.15 EX: the third pipe stage of a store instruction.** Unlike the third stage of the load instruction in Figure 6.13, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.

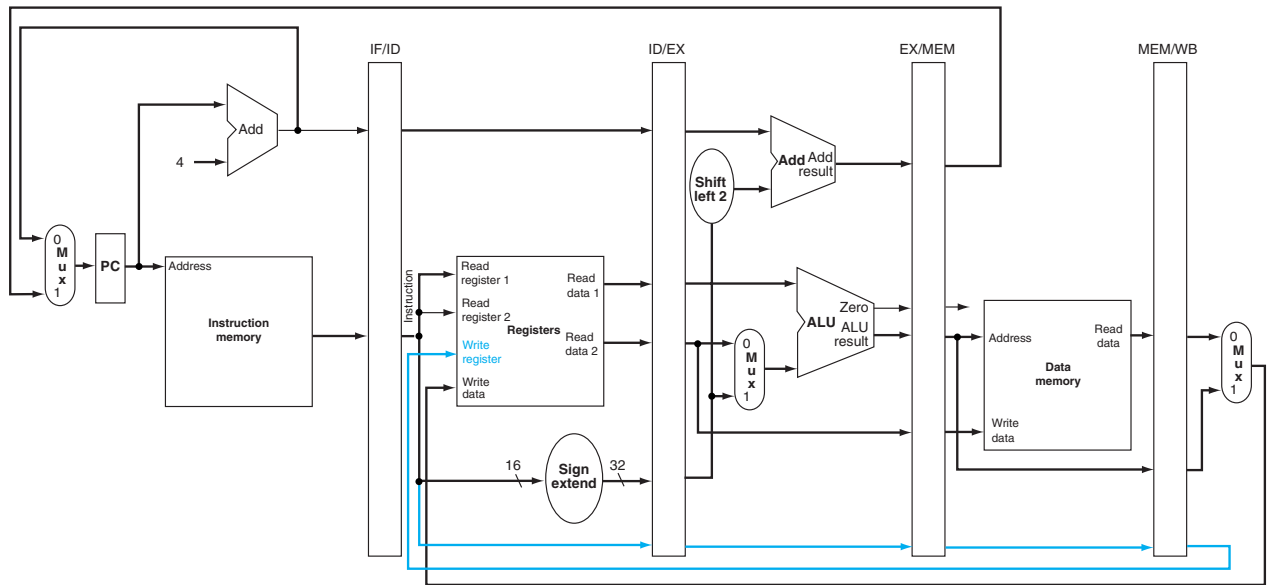
Load and store illustrate a second key point: each logical component of the datapath—such as instruction memory, register read ports, ALU, data memory, and register write port—can be used only within a *single* pipeline stage. Otherwise we would have a *structural hazard* (see page 375). Hence these 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.



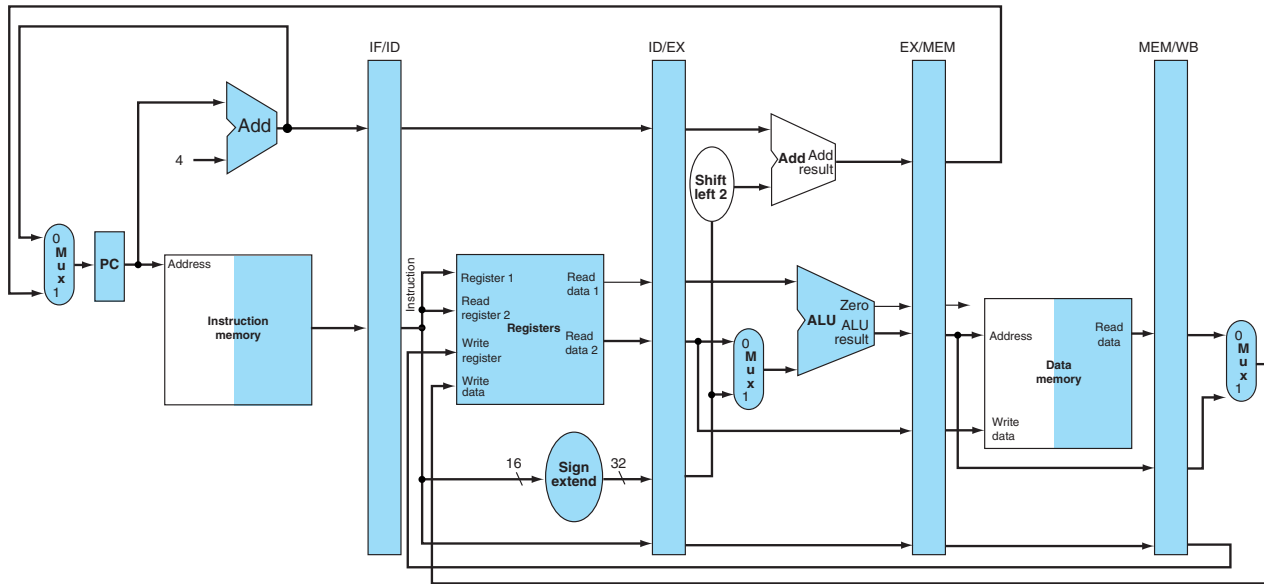
**FIGURE 6.17** The corrected pipelined datapath to properly handle the load instruction. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding 5 more bits to 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 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*,

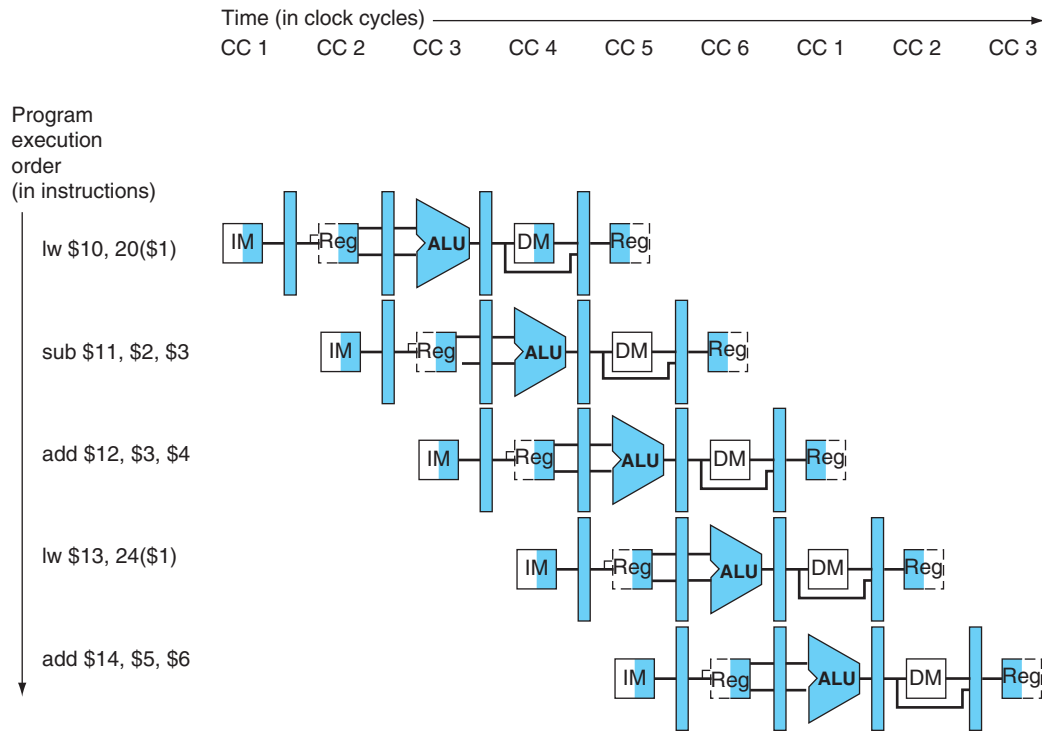


**FIGURE 6.18** The portion of the datapath in Figure 6.17 that is used in all five stages of a load instruction.

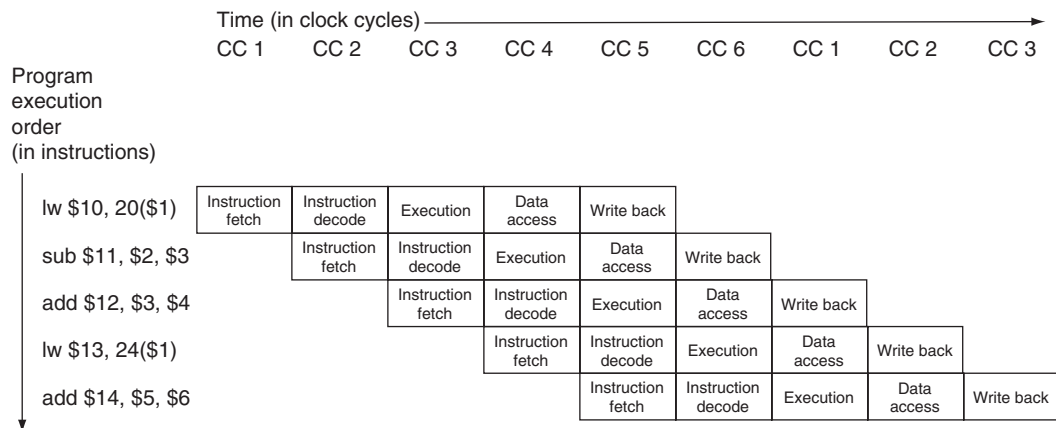
such as Figures 6.12 through 6.16. The multiple-clock-cycle diagrams are simpler but do not contain all the details. For example, consider the following five-instruction sequence:

```
lw    $t0, 20($t1)
sub    $t1, $t2, $t3
add    $t2, $t3, $t4
lw    $t3, 24($t1)
add    $t4, $t5, $t6
```

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.

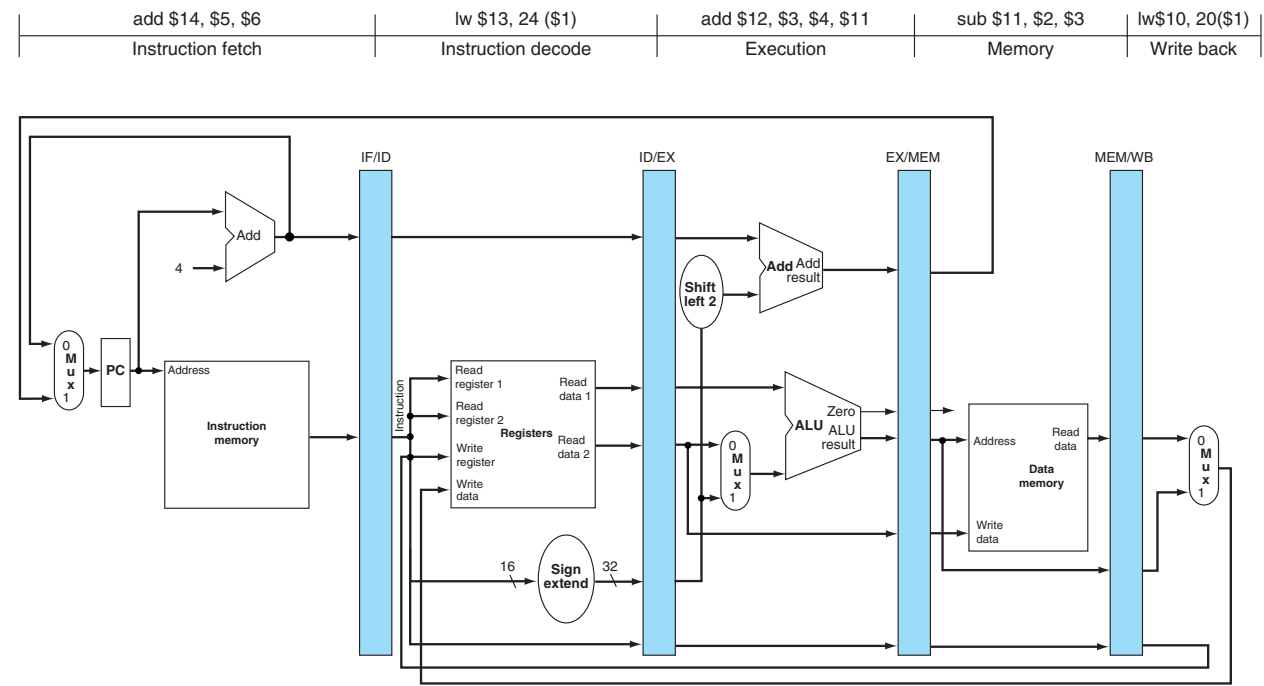


**FIGURE 6.19 Multiple-clock-cycle pipeline diagram of five instructions.** This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike 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.

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

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.

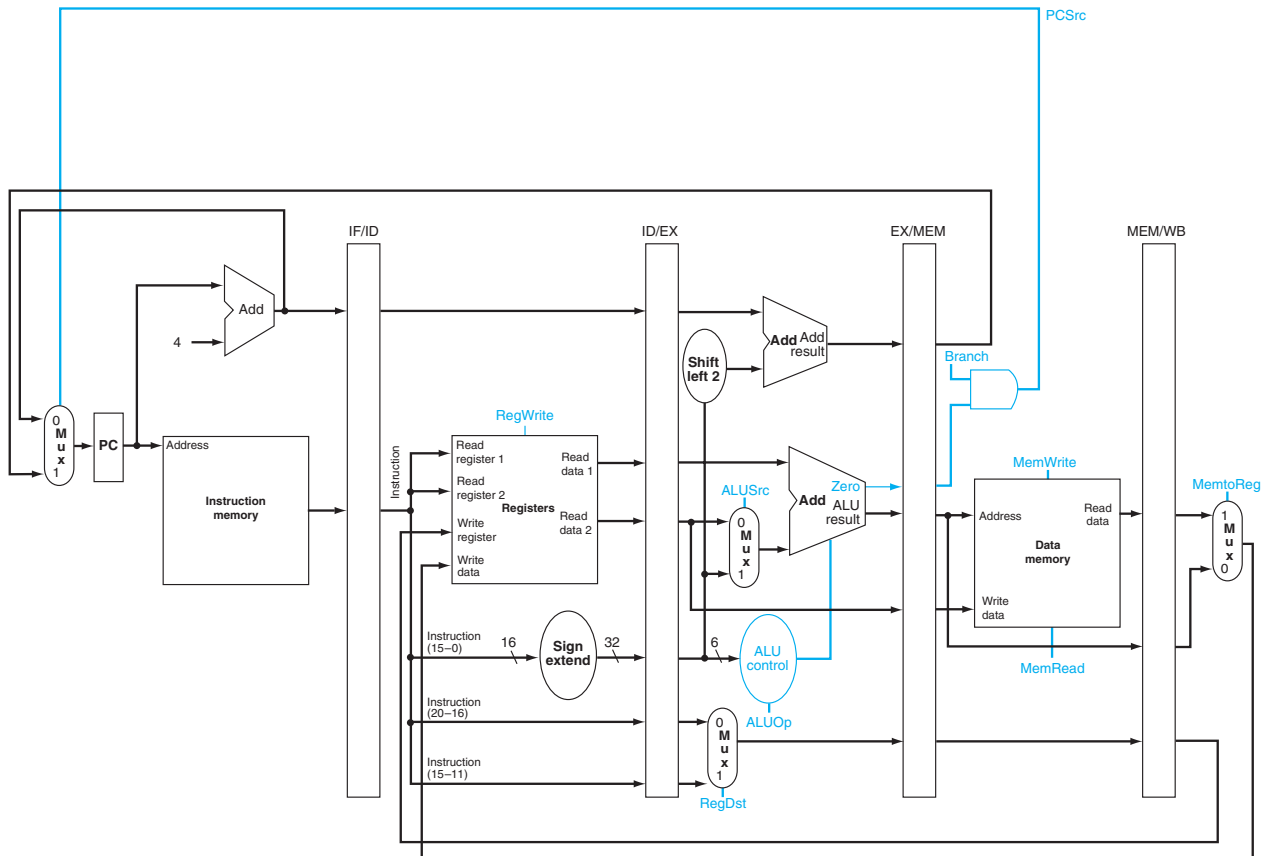
As for the single-cycle implementation discussed in Chapter 5, we assume that the PC is written on each clock cycle, so there is no separate write signal for the

### Check Yourself

*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*





**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.*

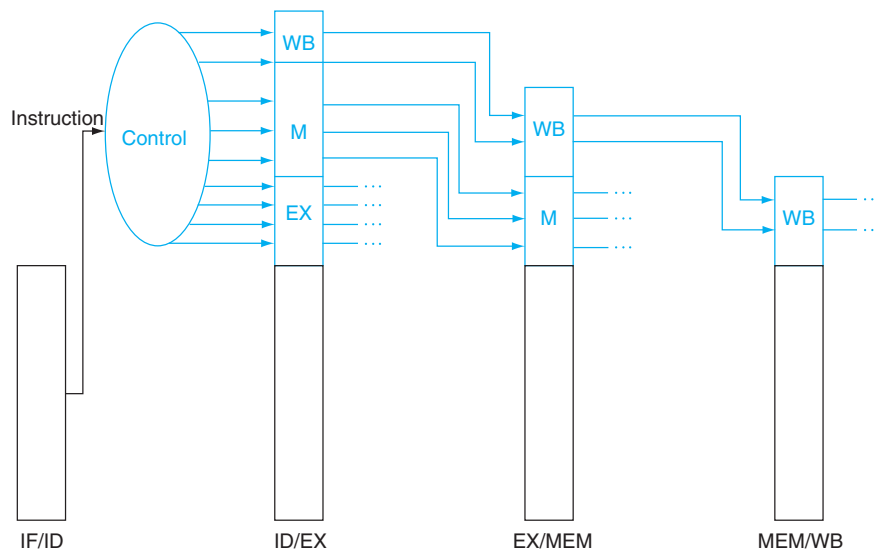
Douglas Adams, *Hitchhikers Guide to the Galaxy*, 1979

## 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.

```

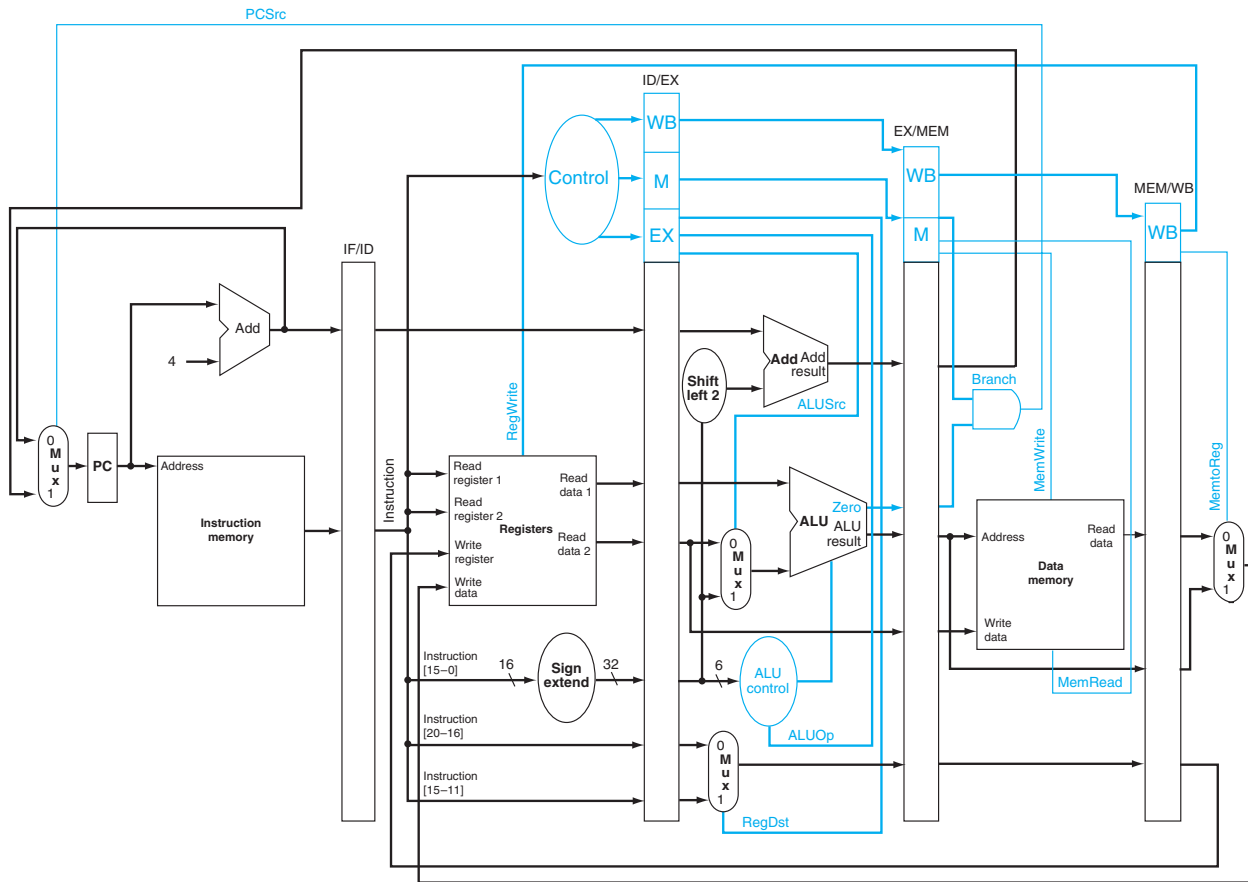
sub    $2, $1,$3    # Register $2 written by sub
and    $12,$2,$5    # 1st operand($2) depends on sub
or     $13,$6,$2    # 2nd operand($2) depends on sub
add    $14,$2,$2    # 1st($2) & 2nd($2) depend on sub
sw     $15,100($2)  # Base ($2) depends on sub

```

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.

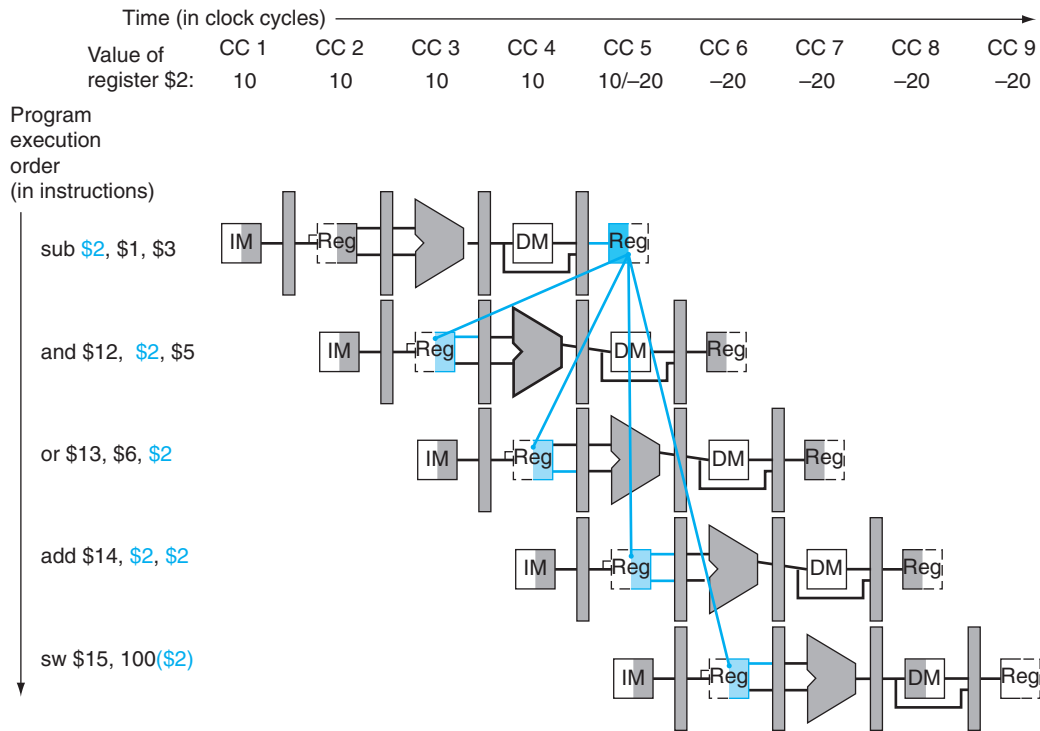
One potential hazard can be resolved by the design of the register file hardware: what happens when a register is read and written in the same clock cycle? We assume that the write is in the first half of the clock cycle and the read is in the second half, so the read delivers what is written. As is 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:

$$\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs} = \$2$$

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:  
$$\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt} = \$2$$
- 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, `sl $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  $\neq$  0 to the first hazard condition and MEM/WB.RegisterRd  $\neq$  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.

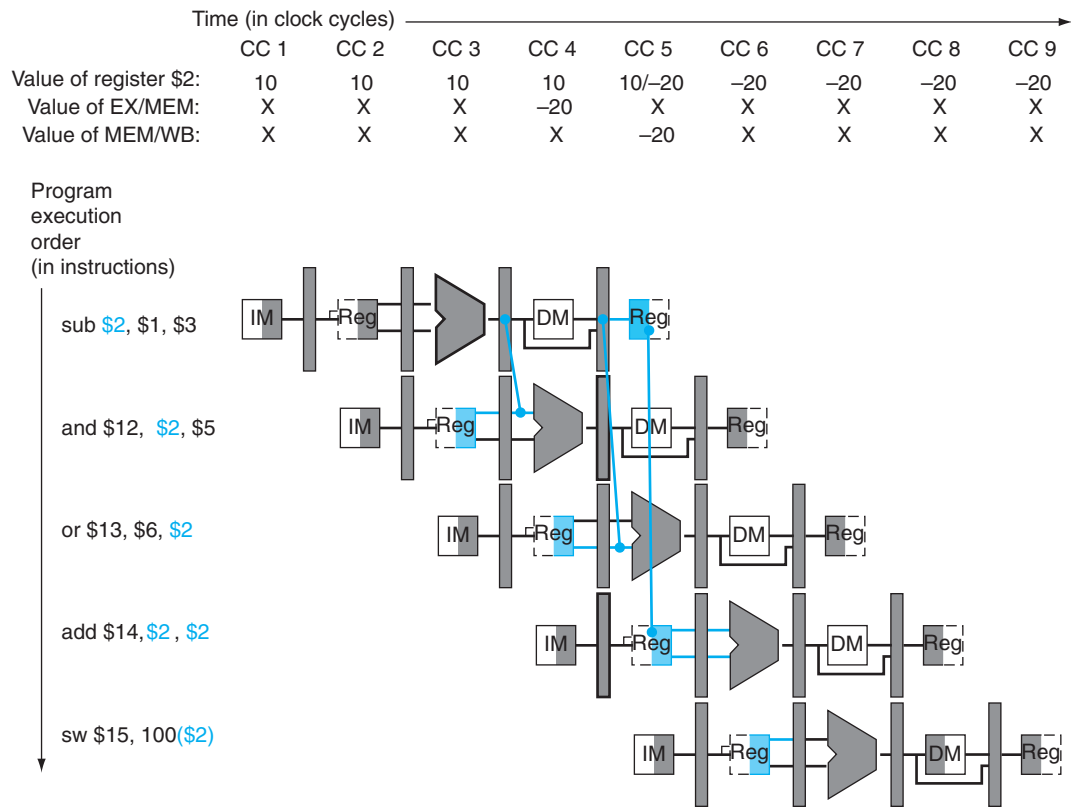
Let's now write both the conditions for detecting hazards and the control signals to resolve them:

1. EX hazard:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd  $\neq$  0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

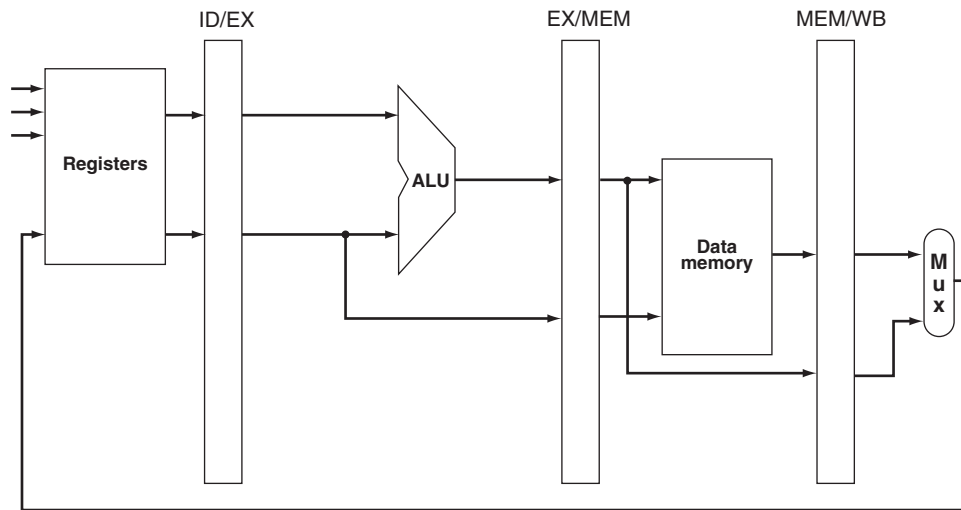
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd  $\neq$  0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```



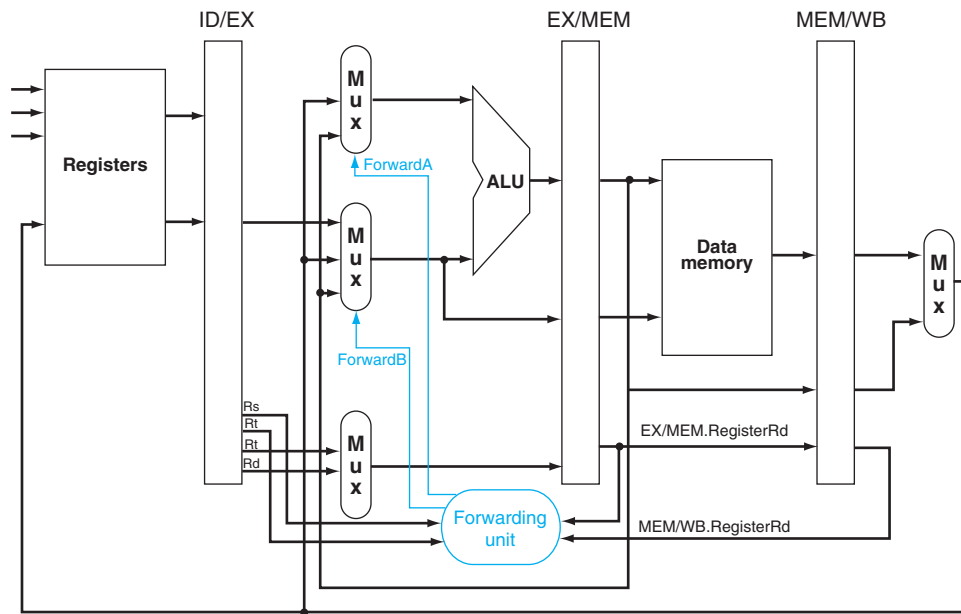


**FIGURE 6.29** The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the and instruction and or instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired 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.



a. No forwarding



b. With forwarding

**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:

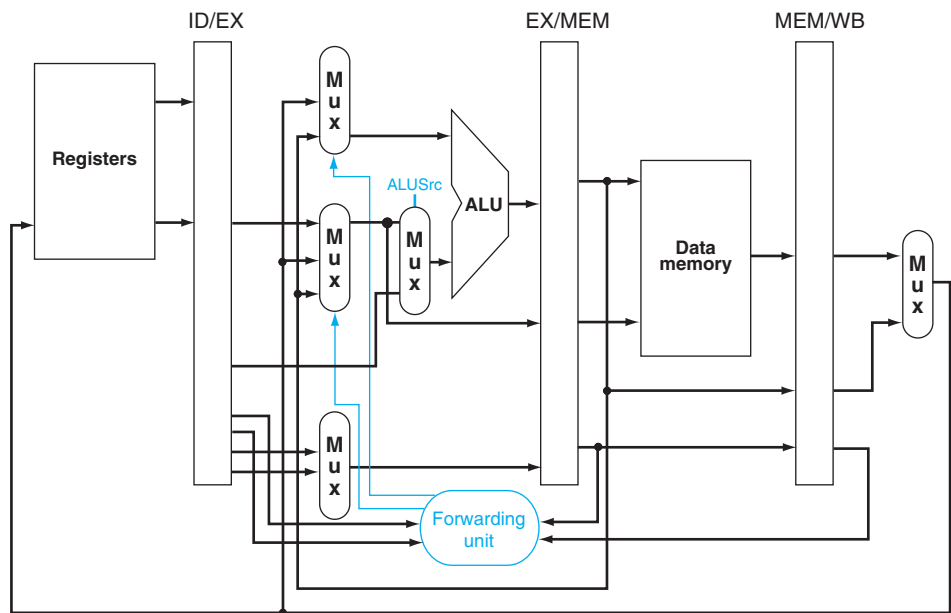
```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
. . .
```

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)



**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:

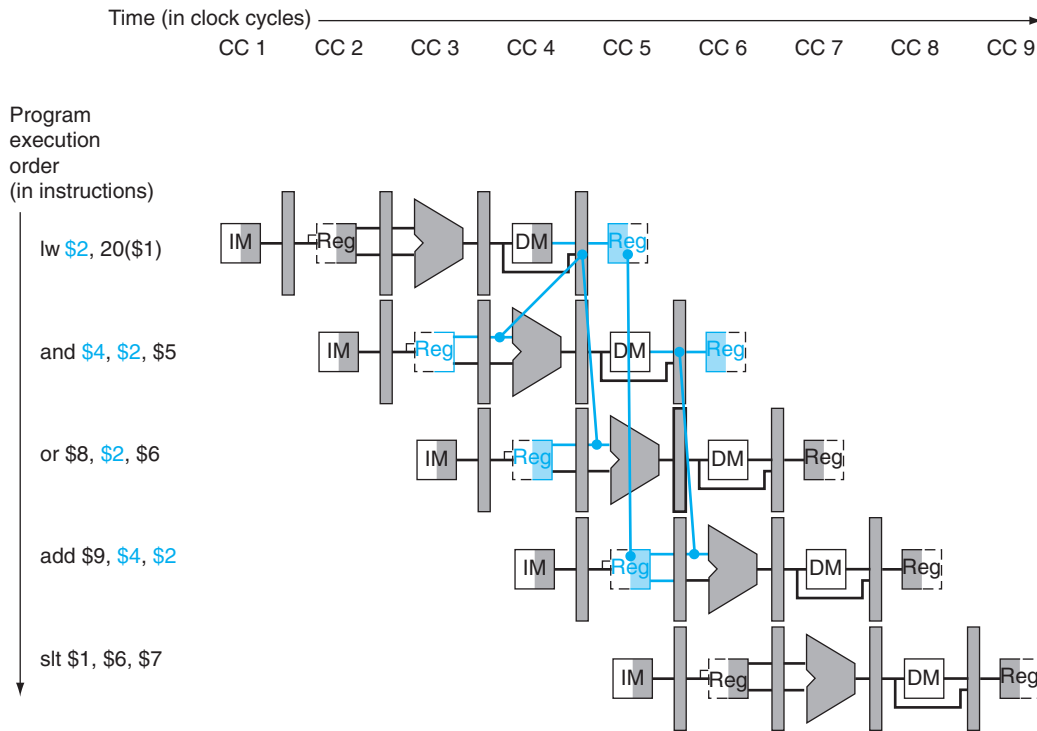
```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
    stall the pipeline
```

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**.

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

**nop** An instruction that does no operation to change state.

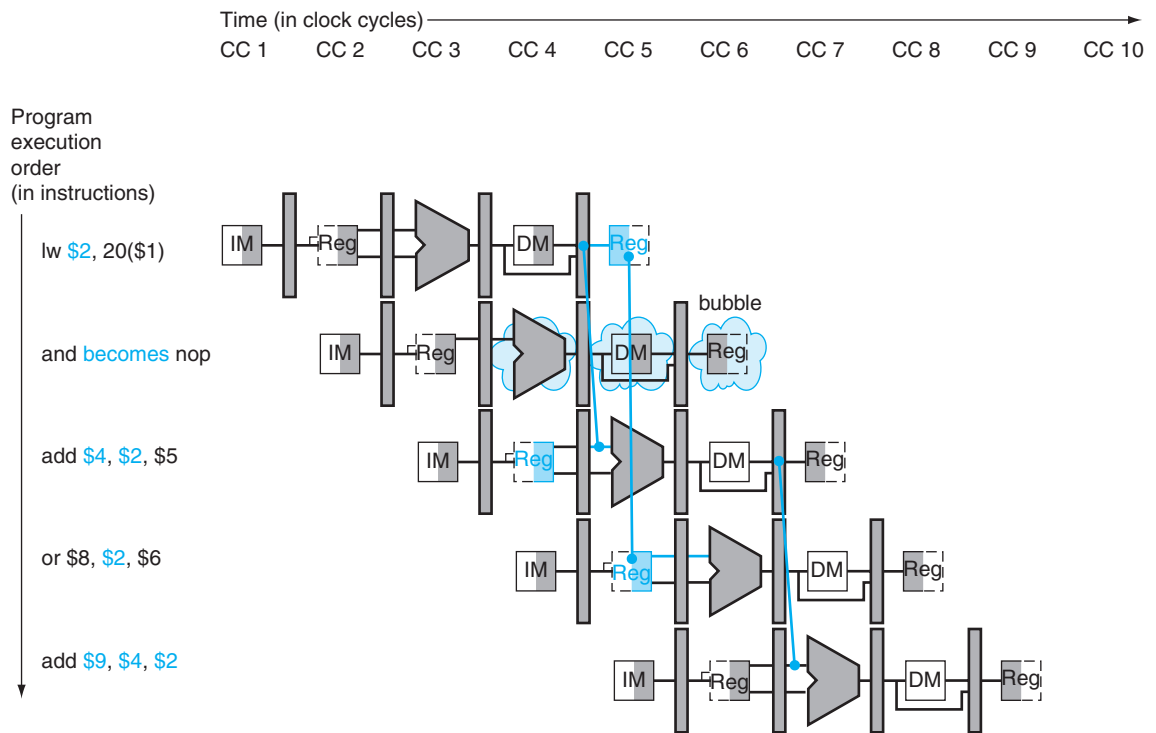


**FIGURE 6.34 A pipelined sequence of instructions.** Since the dependence between the load and the following instruction (*and*) goes backwards in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

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



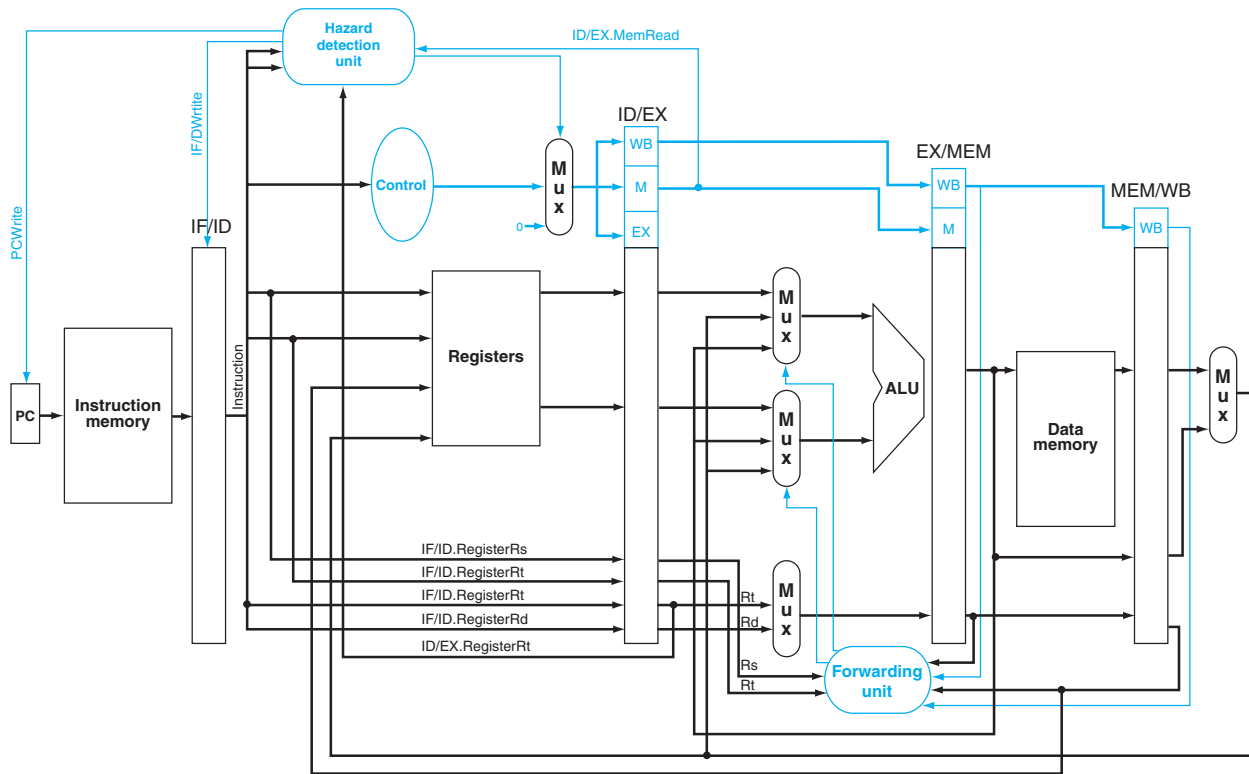
**FIGURE 6.35 The way stalls are really inserted into the pipeline.** A bubble is inserted beginning in clock cycle 4, by changing the `and` instruction to a `nop`. Note that the `and` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus 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 dependencies 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 multiplexers 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.

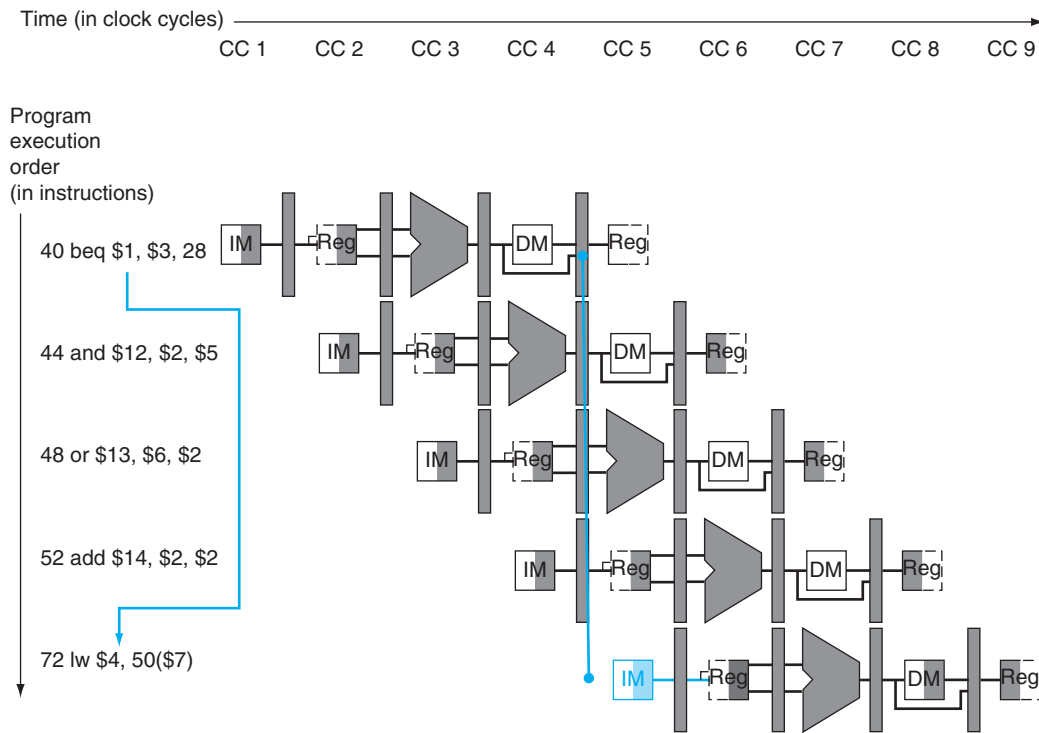
*There are a thousand hacking at the branches of evil to one who is striking at the root.*

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

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:

```

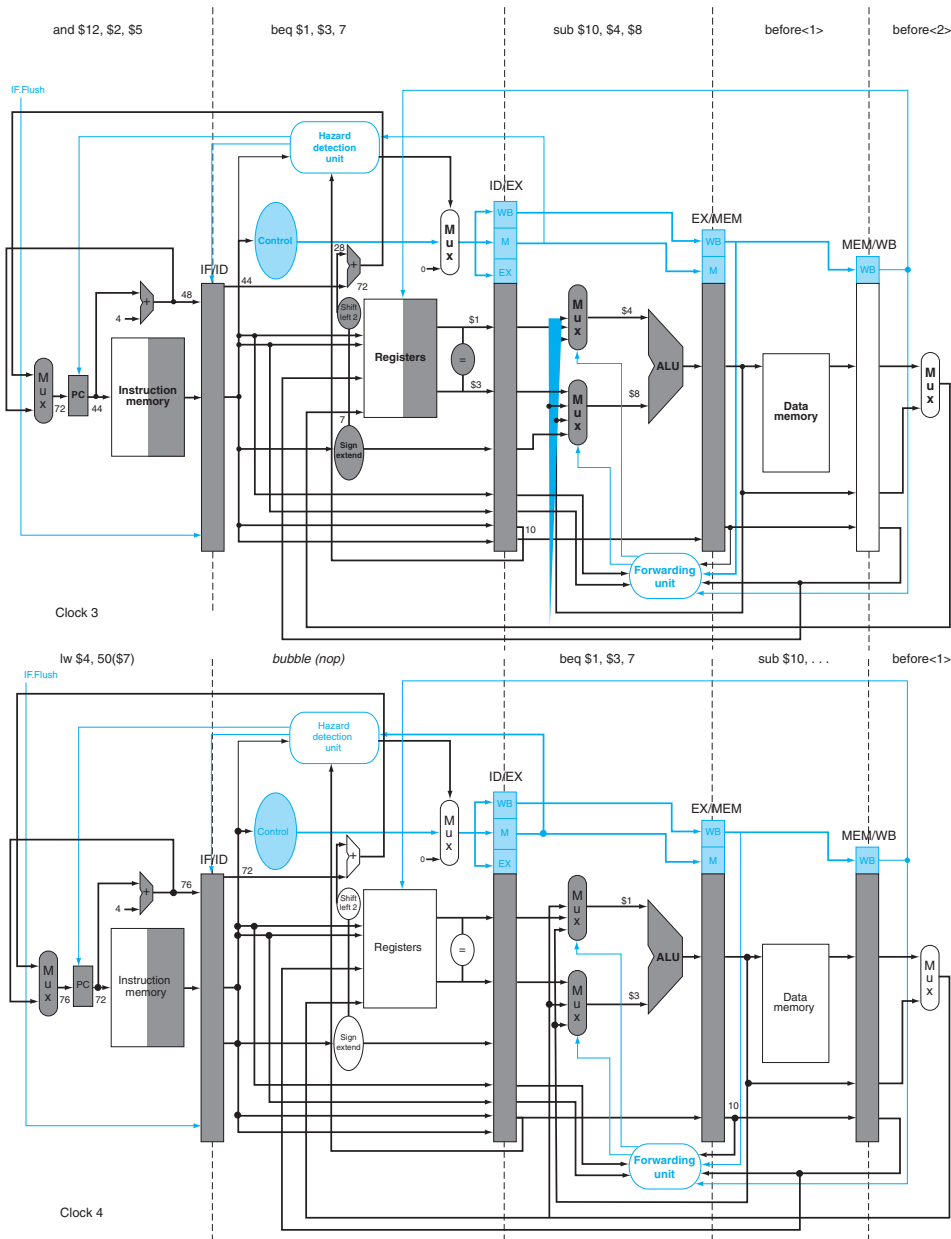
36 sub $t0, $t4, $t8
40 beq $t1, $t3, 7 # PC-relative branch to 40 + 4
   + 7 * 4 = 72
44 and $t2, $t2, $t5
48 or  $t3, $t2, $t6
52 add $t4, $t4, $t2
56 slt $t5, $t6, $t7
   . . .
72 lw  $t4, 50($t7)

```

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.

**EXAMPLE**

**ANSWER**



**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**.

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.

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.

### dynamic branch prediction

Prediction of branches at runtime using run-time information.

### 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.

### Loops and Prediction

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?

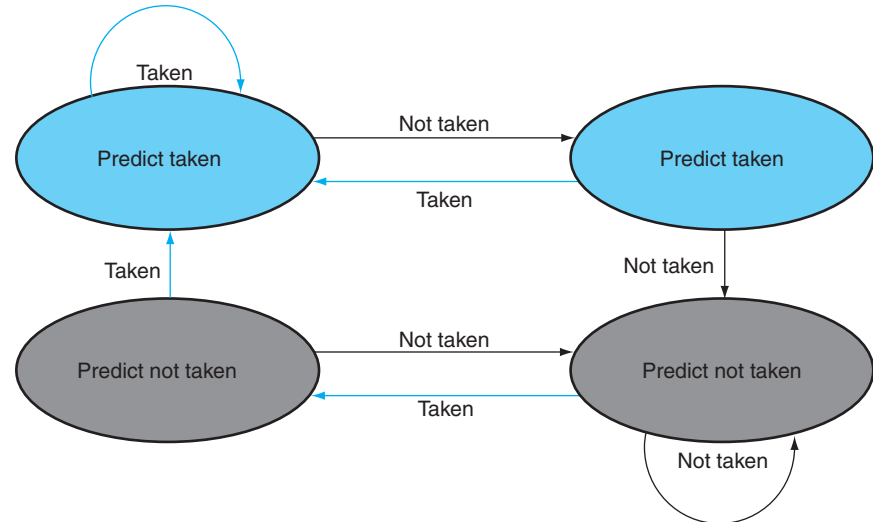
## EXAMPLE

**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.

**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**.

**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.

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.

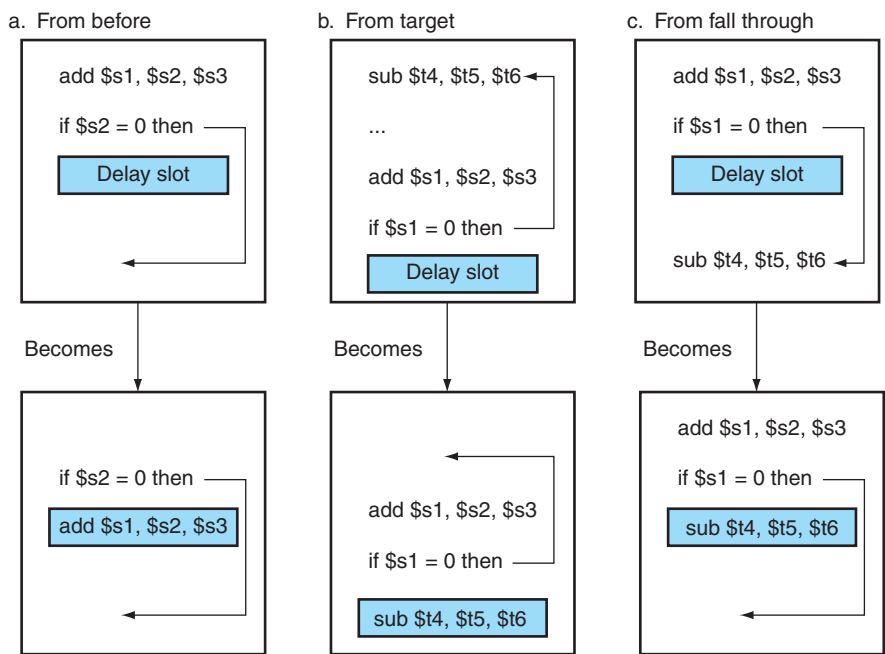
**branch delay slot** The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.

**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.

**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.

**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

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.

#### EXAMPLE

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

$$1.5 \times 25\% + 1 \times 10\% + 1 \times 52\% + 1.25 \times 11\% + 2 \times 2\% = 1.17$$

#### ANSWER

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  $200 \times 4.12 = 824$  ps. For the pipelined design, the average instruction time is  $1.17 \times 200 = 234$  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

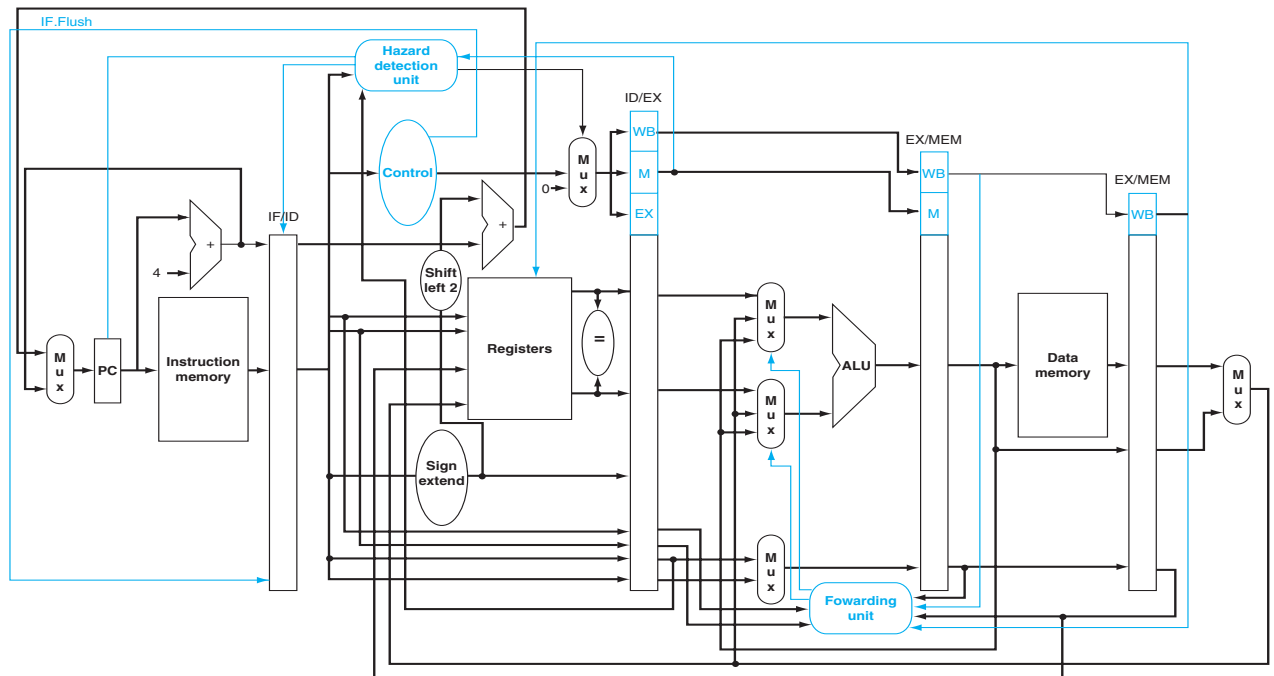
Consider three branch prediction schemes: branch not taken, predict taken, and dynamic prediction. Assume that they all have zero penalty when they predict correctly and 2 cycles when they are wrong. Assume that the average predict accuracy of the dynamic predictor is 90%. Which predictor is the best 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



## 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

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.

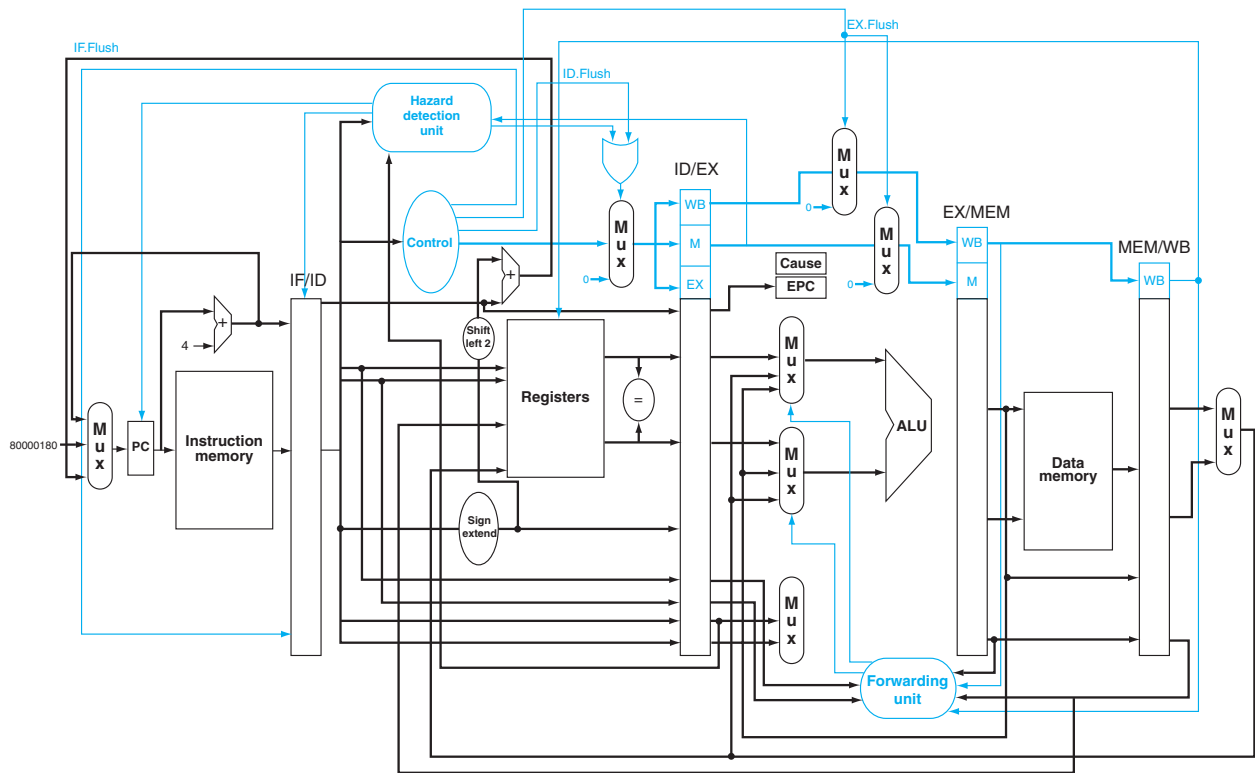
When we dealt with branch mispredict, we saw how to flush the instruction in the IF stage by turning it into a `nop`. To flush instructions in the ID stage, we use the multiplexor already in the ID stage that zeros control signals for stalls. A new

*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

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  $8000\ 0180_{\text{hex}}$ , which is the exception location for an arithmetic overflow, we simply add an additional input to the PC multiplexor that sends  $8000\ 0180_{\text{hex}}$  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  $8000\ 0180_{\text{hex}}$ , 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  $8000\ 0180_{\text{hex}}$  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

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.

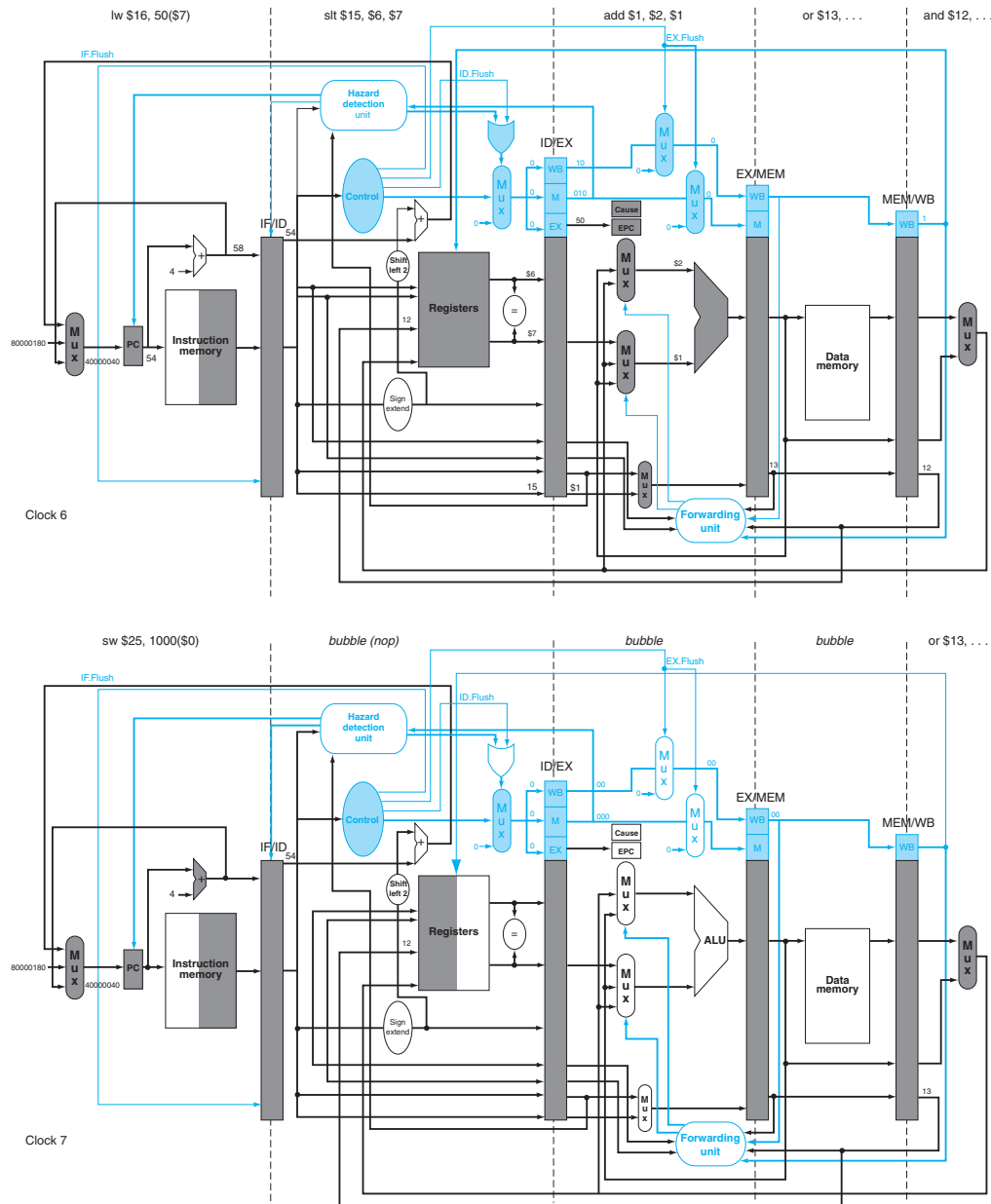
Figure 6.43 shows the events, starting with the add instruction in the EX stage. The overflow is detected during that phase, and 4000 0040<sub>hex</sub> is forced into the PC. Clock cycle 7 shows that the add and following instructions are flushed, and the first instruction of the exception code is fetched. Note that the address of the instruction *following* the add is saved: 4C<sub>hex</sub> + 4 = 50<sub>hex</sub>.

### EXAMPLE

### ANSWER

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 ( $4C + 4 = 50_{\text{hex}}$ ). 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  $4000\ 0040_{\text{hex}}$ . 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.

---

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.

---

## Hardware Software Interface



**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 always associated with the correct instruction in pipelined computers.

### Check Yourself

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  $58_{\text{hex}}$  at the start of the clock cycle after the exception is detected, even though the offending instruction is at address  $4C_{\text{hex}}$ . A processor with imprecise exceptions might put  $58_{\text{hex}}$  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.)

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.

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.

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 dependencies arise.

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.

#### **instruction-level parallelism**

The parallelism among instructions.

**multiple issue** A scheme whereby multiple instructions are launched in 1 clock cycle.

**static multiple issue** An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.

**dynamic multiple issue** An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.

**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.

There are two primary and distinct responsibilities that must be dealt with in a multiple-issue pipeline:

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.

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.

**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.

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.

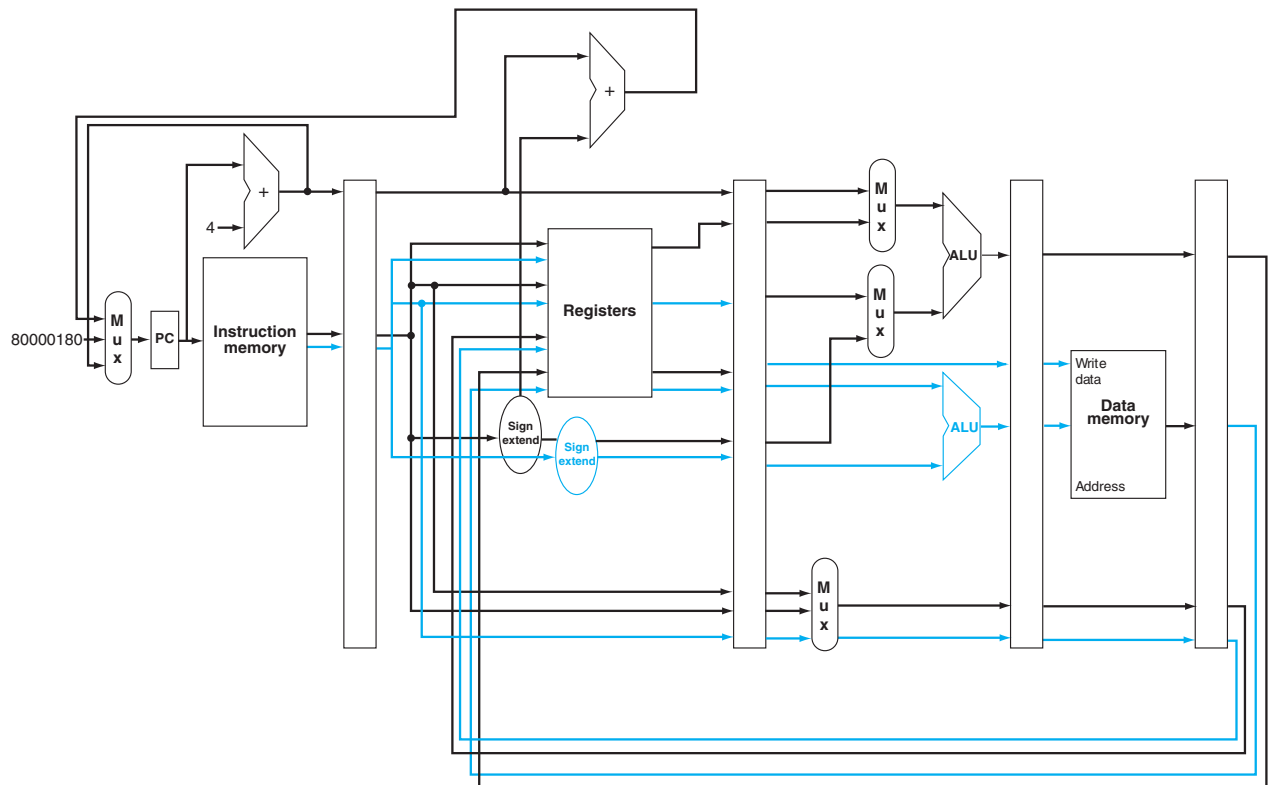
Static multiple-issue processors vary in how they deal with potential data and control hazards. In some designs, the compiler takes full responsibility for removing *all* hazards, scheduling the code and inserting no-ops so that the code executes without any need for hazard detection or hardware-generated 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

See how well loop unrolling and scheduling work in the example above. Assume that the loop index is a multiple of four, for simplicity.

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.

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.

### EXAMPLE

### ANSWER

**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 than by a true dependence that carries a value between two



	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.

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.

**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.

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.

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.

**predication** A technique to make instructions dependent on predicates rather than on branches.

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.

**poison** A result generated when a speculative load yields an exception, or an instruction uses a poisoned operand.

**advanced load** In IA-64, a speculative load instruction with support to check for aliases that could invalidate the load.

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

**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.

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.

Dynamic Multiple-Issue Processors

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:

```
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.

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

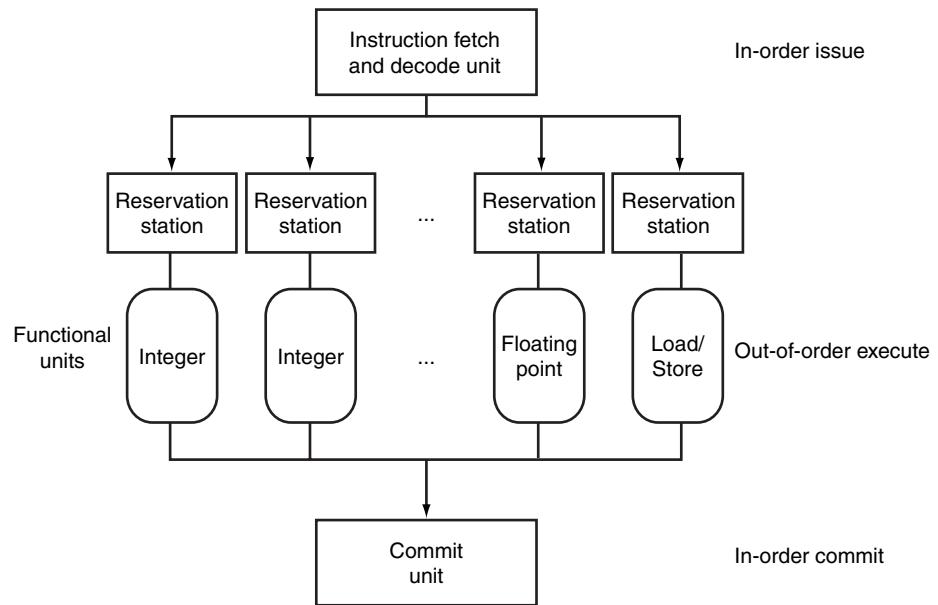
#### dynamic pipeline scheduling

Hardware support for reordering the order of instruction execution so as to avoid stalls.

**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.



**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.

**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.

**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.

**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.

**out-of-order execution** A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait.

## 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.

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.

## Understanding Program Performance

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.

## Check Yourself

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.

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.

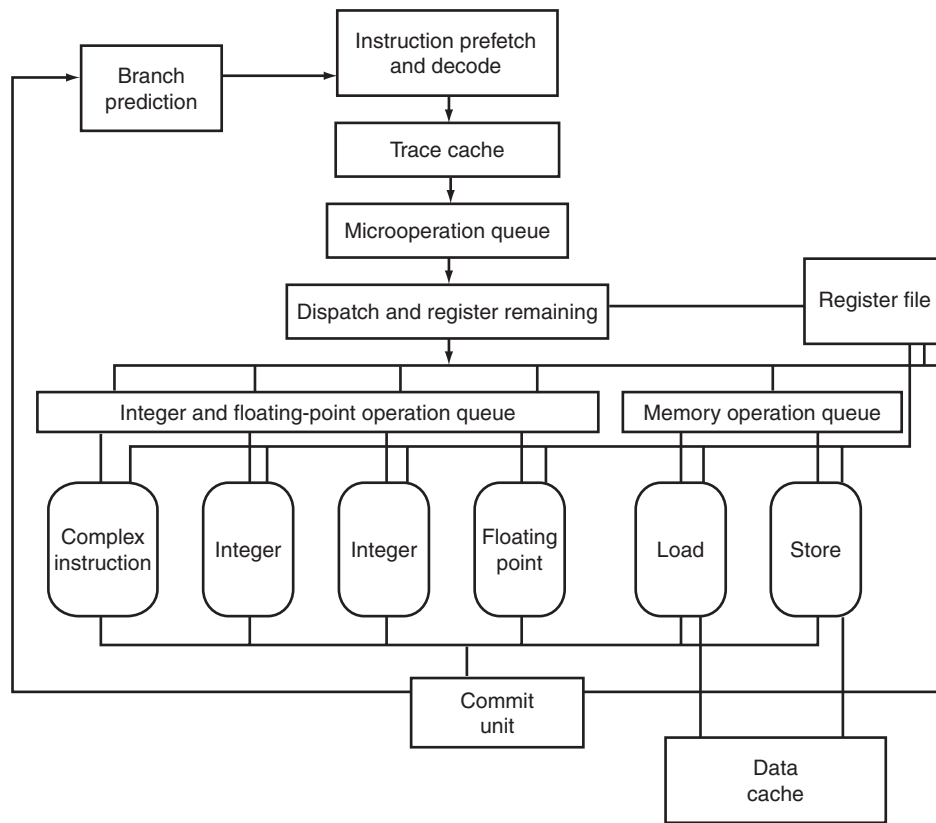
The Pentium 4, and the earlier Pentium III and Pentium Pro, all use the technique of decoding IA-32 instructions into microoperations and executing those microoperations using a speculative pipeline with multiple functional units. In fact, the basic microarchitecture is similar, and all these processors 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

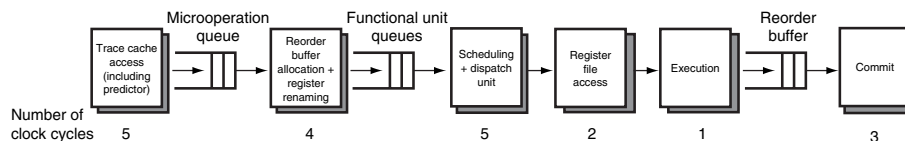
**microarchitecture** The organization of the processor, including the major functional units, their interconnection, and control.

**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 than 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

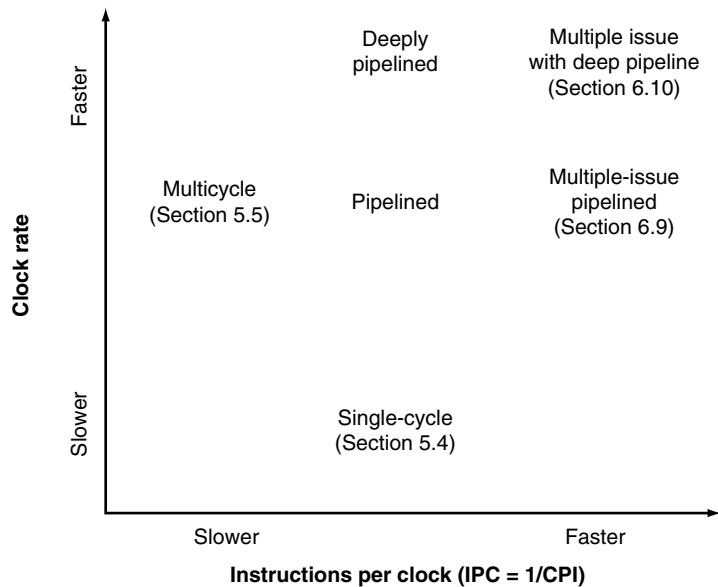
Pipelining improves the average execution time per instruction. Depending on whether you start with a single-cycle or multiple-cycle datapath, this reduction can be thought of as decreasing the clock cycle time or as decreasing the number of clock cycles per instruction (CPI). We started with the simple 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**.

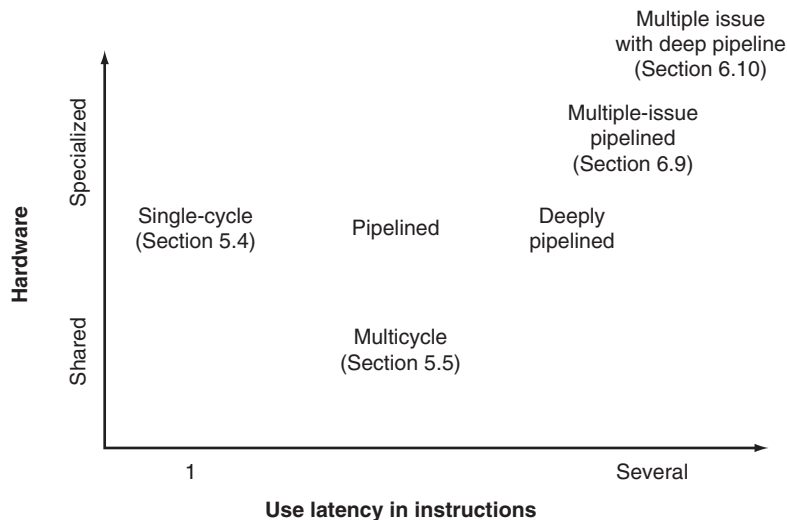
Pipelining and multiple issue both attempt to exploit instruction-level parallelism. The presence of data and control dependences, which can become hazards, are the primary limitations on how much parallelism can be exploited. Scheduling and speculation, both in hardware and software, are the primary 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.

**instruction latency** The inherent execution time for an instruction.




**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.



## 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.



## 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);

- Will it affect the speedup obtained from pipelining? If yes, by how much? Otherwise, why?
- 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.

- How long does it take to execute this program core on a nonpipelined processor?
- 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?
- 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:

- The PC has the value  $100_{\text{ten}}$ , the address of the `sub_instruction`.
- Every register has the initial value  $10_{\text{ten}}$  plus the register number (e.g., register \$8 has the initial value  $18_{\text{ten}}$ ).
- Every memory word accessed as data has the initial value  $1000_{\text{ten}}$  plus the byte address of the word (e.g., `Memory[8]` has the initial value  $1008_{\text{ten}}$ ).

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:

- What would be the actual CPI?
- 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 the Verilog code in 6.7.3 on page 6.7-6–6.7-7 to implement a delayed branch strategy.

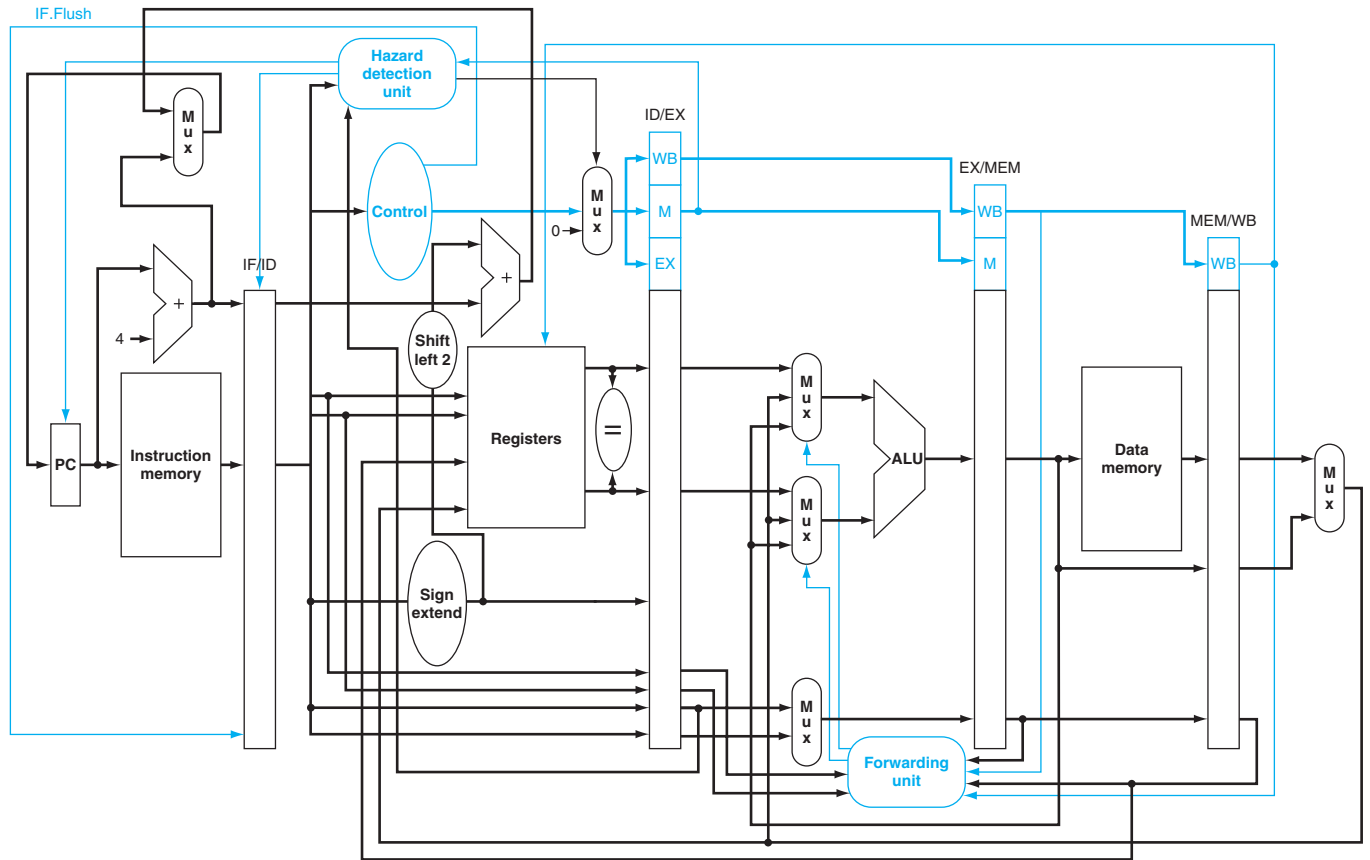
**6.45** [20] <§§6.6, 6.7> Rewrite the verilog code in Figure 6.7.3 on page 6.7-6–6.7-7 to implement a branch target buffer. Assume the buffer is implemented with a module with the following definition:

```

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.

**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 latest processors from Intel or AMD as a starting point.

§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.

## Answers to Check Yourself



# 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.

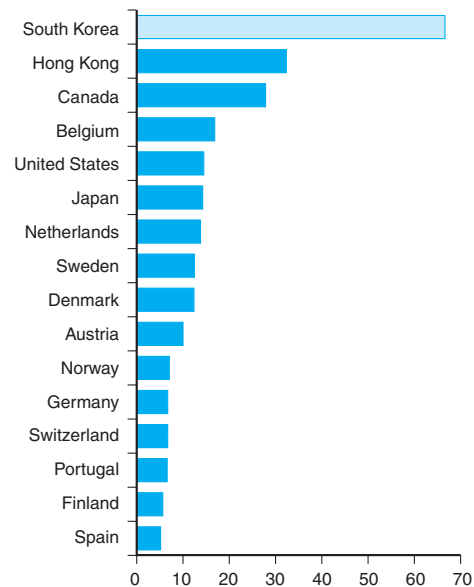
The Internet holds the promise of allowing citizens to communicate without the information first being interpreted by traditional mass media like television, newspapers, and magazines. To see what the future might be, we could look at countries that have widespread, high-speed Internet access.

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](http://www.ohmynews.com)

Google News, [www.news.google.com](http://www.news.google.com)

New York Times Front Page	Google News
<p><b>Judge Rules Out a Death Penalty for 9/11 Suspect</b> Rebuke for Justice Dept.</p> <p><b>Poll Shows Drop in Confidence on Bush Skill in Handling Crises</b> Country on Wrong Track, Says Solid Majority</p> <p><b>Revised Admission for High Schools</b> City Says Students Will Get First Preference</p> <p><b>No Illicit Arms Found in Iraq, U.S. Inspector Tells Congress</b></p> <p><b>U.S. Practice How to Down Hijacked Jets</b> Coetzee, Writer of Apartheid as Bleak Mirror, Wins Nobel</p> <p><b>Sexual Accusations Lead to an Apology by Schwarzenegger</b></p> <p><b>Interim Chief Accepts Stock Exchange Shift</b></p> <p><b>Yankees Even with Twins</b> Agency Warns of Fake Drugs Limbaugh Fallback Position</p>	<p>Top Stories <u>More than 1000 rally behind Schwarzenegger</u> AP - 5 minutes ago <u>Maria Shriver defends husband CNN</u> <u>Can accusations hurt Arnold's campaign?</u> KESQ and 1252 related</p> <p><u>Bush: Hussein 'A Danger to the World' ABC</u> news - 5 hours ago <u>Bush Stands By Decision Voice of America</u> <u>Hunt for weapons yields no evidence The</u> Canberra Times and 598 related</p> <p>World Stories <u>Defiant UN chief announces rival blueprint for Iraq</u> The Times (UK) - 2 hours ago <u>France, Russia Assail US Draft on Iraq</u> Reuters and 782 related</p>

**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.