computers with very simple instruction sets did use this implementation technique. However, if we tried to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design wouldn't work well at all.

COMMON CASE FAST

Because we must assume that the clock cycle is equal to the worst-case delay for all instructions, it's useless to try implementation techniques that reduce the delay of the common case but do not improve the worst-case cycle time. A single-cycle implementation thus violates the great idea from Chapter 1 of making the **common case fast**.

In next section, we'll look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath but is much more efficient by having a much higher throughput. Pipelining improves efficiency by executing multiple instructions simultaneously.

Check Yourself

Look at the control signals in Figure 4.22. Can you combine any together? Can any control signal output in the figure be replaced by the inverse of another? (Hint: take into account the don't cares.) If so, can you use one signal for the other without adding an inverter?

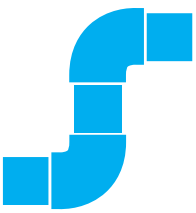
4.5 An Overview of Pipelining

Never waste time.
American proverb

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, **pipelining** is nearly universal.

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

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 4.10 and 4.11 to see an introduction to the advanced pipelining techniques used in recent processors such as the Intel Core i7 and ARM Cortex-A8. If you are interested in exploring the anatomy of a pipelined computer, this section is a good introduction to Sections 4.6 through 4.9.



PIPELINING

Anyone who has done a lot of laundry has intuitively used pipelining. The *non-pipelined* approach to laundry would be as follows:

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, start over with the next dirty load.

The *pipelined* approach takes much less time, as Figure 4.25 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 put 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. 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 speed-up due to pipelining is equal to the number of stages in the

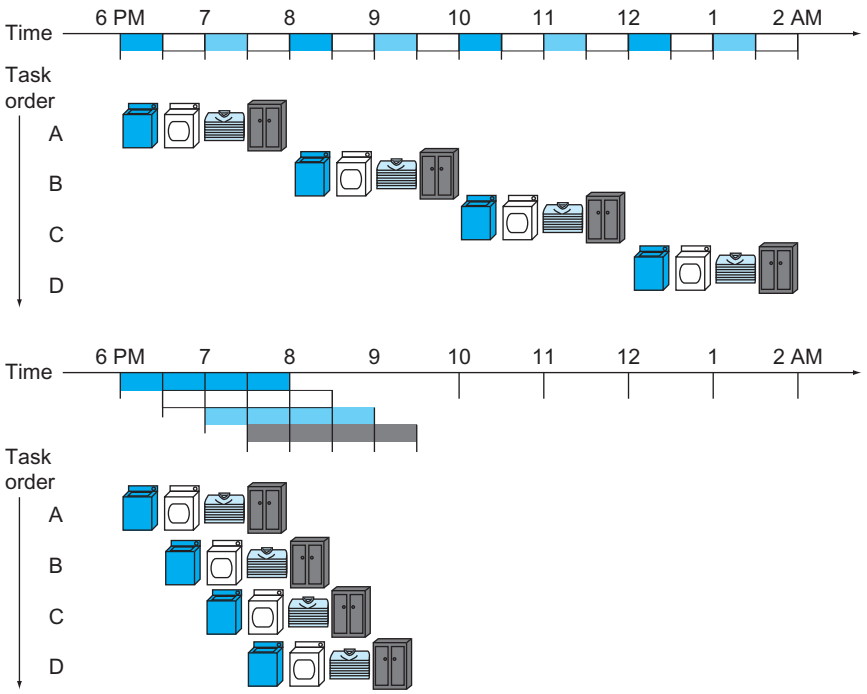


FIGURE 4.25 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes 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.

pipeline, in this case four: washing, drying, folding, and putting away. Therefore, pipelined laundry is potentially four times faster than nonpipelined: 20 loads would take about 5 times as long as 1 load, while 20 loads of sequential laundry takes 20 times as long as 1 load. It's only 2.3 times faster in Figure 4.25, because we only show 4 loads. Notice that at the beginning and end of the workload in the pipelined version in Figure 4.25, 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 regular 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 one 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. In the single-cycle model, every instruction takes exactly one clock cycle, so the clock cycle must be stretched to accommodate the slowest instruction.

ANSWER

Figure 4.26 shows the time required for each of the eight instructions. The single-cycle design must allow for the slowest instruction—in Figure 4.26 it is `lw`—so the time required for every instruction is 800 ps. Similarly

to Figure 4.25, Figure 4.27 compares nonpipelined and pipelined execution of three load word instructions. Thus, the time between the first and fourth instructions in the nonpipelined design is 3 × 800 ns or 2400 ps.

All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation. Just as the single-cycle design must take the worst-case clock cycle of 800 ps, even though some instructions can be as fast as 500 ps, the pipelined execution clock cycle must have the worst-case clock cycle of 200 ps, even though some stages take only 100 ps. Pipelining still offers a fourfold performance improvement: the time between the first and fourth instructions is 3 × 200 ps or 600 ps.

We can turn the pipelining speed-up 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

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

Under ideal conditions and with a large number of instructions, the speed-up 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. Moreover, pipelining involves some overhead, the source of which will be clearer shortly. Thus, the time per instruction in the pipelined processor will exceed the minimum possible, and speed-up will be less than the number of pipeline stages.

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 4.26 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.

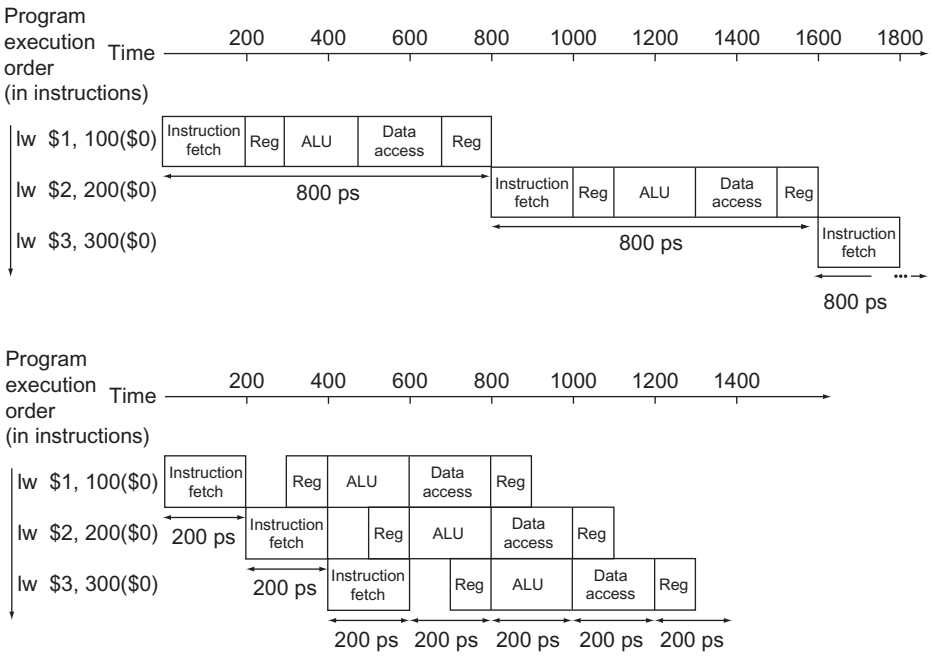


FIGURE 4.27 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 4.26. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.25. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also 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.

Moreover, even our claim of fourfold improvement for our example is not reflected in the total execution time for the three instructions: it's 1400 ps versus 2400 ps. Of course, this is because the number of instructions is not large. What would happen if we increased the number of instructions? We could extend the previous figures to 1,000,003 instructions. We would add 1,000,000 instructions in the pipelined example; each instruction adds 200 ps to the total execution time. The total execution time would be 1,000,000 × 200 ps + 1400 ps, or 200,001,400 ps. In the nonpipelined example, we would add 1,000,000 instructions, each taking 800 ps, so total execution time would be 1,000,000 × 800 ps + 2400 ps, or 800,002,400 ps. Under these conditions, the ratio of total execution times for real programs on nonpipelined to pipelined processors is close to the ratio of times between instructions:

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

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 x86, where instructions vary from 1 byte to 15 bytes, pipelining is considerably more challenging. Recent implementations of the x86 architecture actually translate x86 instructions into simple operations that look like MIPS instructions and then pipeline the simple operations rather than the native x86 instructions! (See Section 4.10.)

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 x86, 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.

Hazards

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

structural hazard When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

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 4.27 had a fourth instruction, we would see that in the same clock cycle the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipeline could have a structural hazard.

Data Hazards

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

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

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 waste three clock cycles in the pipeline.

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.

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**.

EXAMPLE

Forwarding with Two Instructions

For the two instructions above, show what pipeline stages would be connected by forwarding. Use the drawing in Figure 4.28 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 4.25.

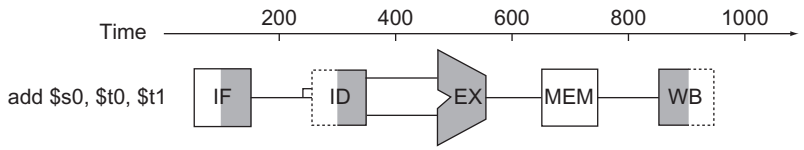


FIGURE 4.28 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.25. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write-back stage, with the drawing showing the register file being written. The shading indicates the element is used by the instruction. Hence, MEM has a white background because `add` does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded in the fifth stage because the register file is written.

Figure 4.29 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.

ANSWER

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

Forwarding works very well and is described in detail in Section 4.7. It cannot prevent all pipeline stalls, however. For example, suppose the first instruction was a load of `$s0` instead of an add. As we can imagine from looking at Figure 4.29, the

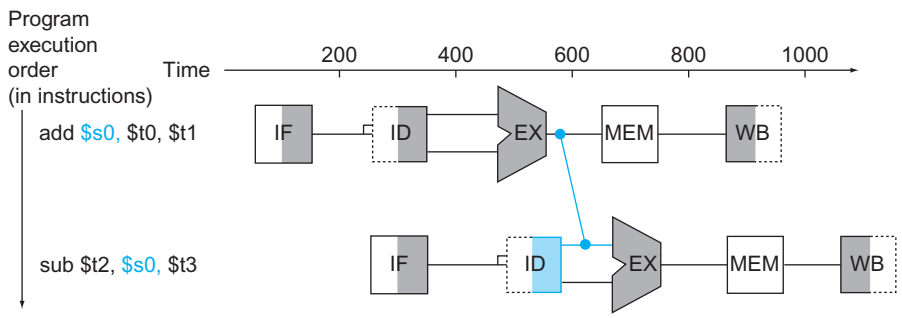


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `$s0` read in the second stage of `sub`.

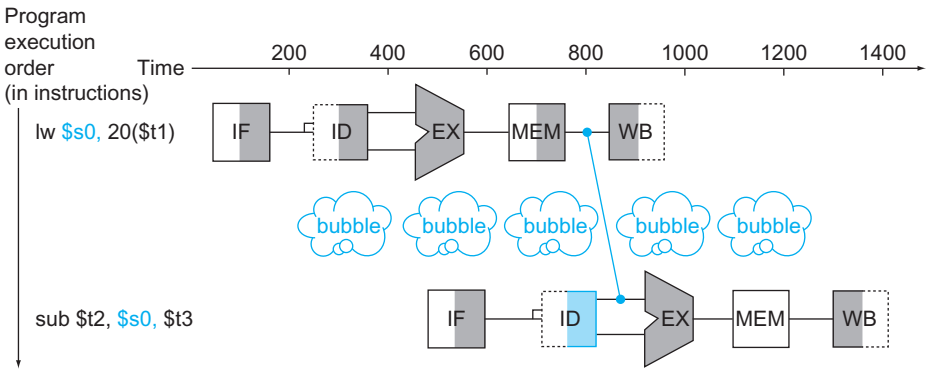


FIGURE 4.30 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 4.7 shows the details of what really happens in the case of a hazard.

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

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

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 4.30 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 4.7 shows how we can handle hard cases like these, using either hardware detection and stalls or software that reorders code to try to avoid load-use pipeline stalls, as this example illustrates.

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 preceding code segment and reorder the instructions to avoid any pipeline stalls.

ANSWER

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

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
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 277. Each MIPS instruction writes at most one result and does this in the last stage of the pipeline. Forwarding is harder if there are multiple results to forward per instruction or if there is a 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 around the register file to the desired unit.

Control Hazards

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

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

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

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

Stall: Just operate sequentially until the first batch is dry and then repeat until you have the right formula.

This conservative option certainly works, but it is slow.

The equivalent decision task in a computer is the branch instruction. Notice that we must begin fetching the instruction following the branch on the very next clock cycle. Nevertheless, 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 4.8 for details). Even with this extra hardware, the pipeline involving conditional branches would look like Figure 4.31. The `lw` instruction, executed if the branch fails, is stalled one extra 200 ps clock cycle before starting.

EXAMPLE

Performance of "Stall on Branch"

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

ANSWER

Figure 3.27 in Chapter 3 shows that branches are 17% of the instructions executed in SPECint2006. Since the other instructions run have a CPI of 1, and branches took one extra clock cycle for the stall, then we would see a CPI of 1.17 and hence a slowdown of 1.17 versus the ideal case.

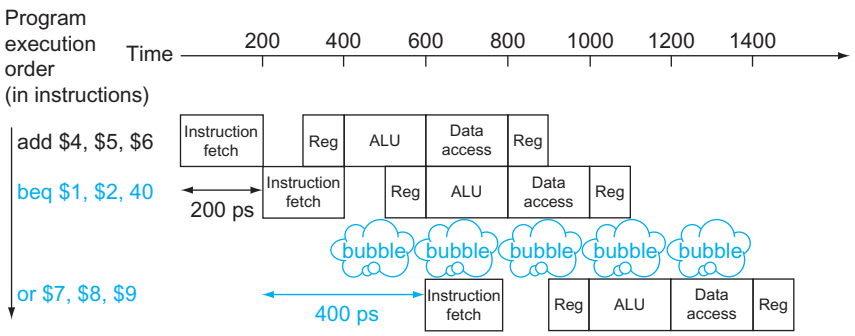
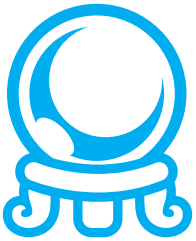


FIGURE 4.31 Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the `OR` instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 4.8. 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 using one of our great ideas from Chapter 1:

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.



PREDICTION

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 predict always 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 4.32 shows such an example.

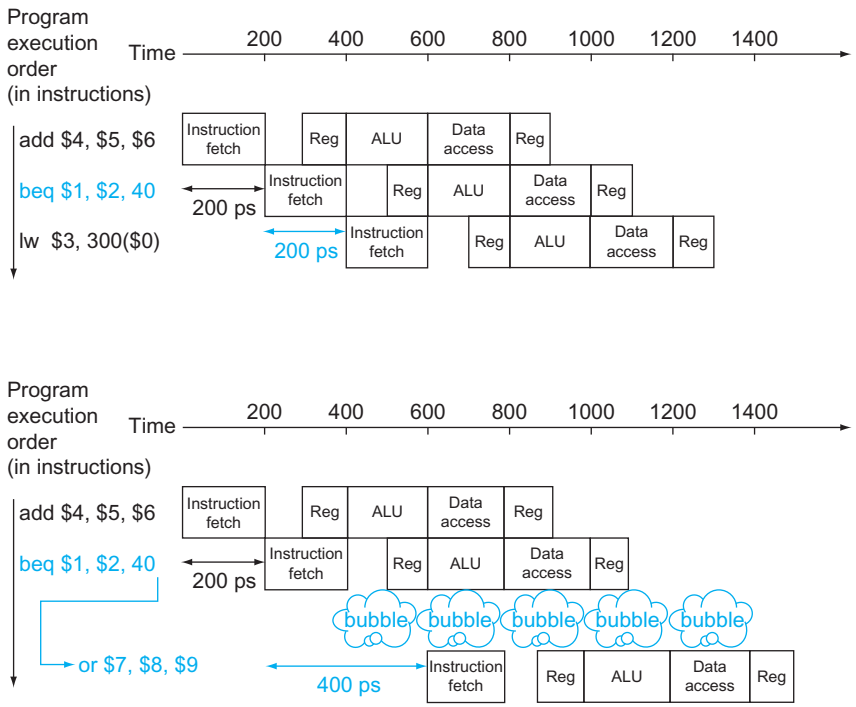
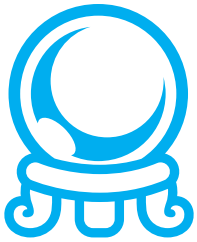


FIGURE 4.32 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 4.31, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. Section 4.8 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.



PREDICTION

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. In the case of programming, 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 backward, 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 **prediction** 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 more than 90% accuracy (see Section 4.8). 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 4.8.

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, and mentioned above, 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 4.31 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 one 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 programming a multiprocessor, it is fundamentally invisible to the programmer.

In the next few sections of this chapter, we cover the concept of pipelining using the MIPS instruction subset from the single-cycle implementation in Section 4.4 and show 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 4.10. Section 4.10 introduces advanced pipelining concepts, such as superscalar and dynamic scheduling, and Section 4.11 examines the pipelines of recent microprocessors.

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 and the basic control, explained in Section 4.6. You can then use this understanding to explore the implementation of forwarding and stalls in Section 4.7. You can then read Section 4.8 to learn more about solutions to branch hazards, and then see how exceptions are handled in Section 4.9.

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.

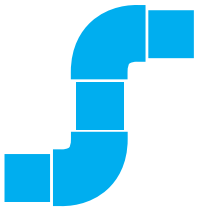
Sequence 1	Sequence 2	Sequence 3

`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`

Understanding Program Performance

Outside 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 4.10, 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 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



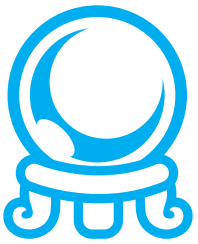
PIPELINING

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 memory 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 memory access, involving more use of pointers. As we will see in Section 4.10, there are more ambitious compiler and hardware techniques for reducing data dependences through scheduling.

The BIG Picture

Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed. Pipelining does not reduce the time it takes to complete an individual instruction, 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 1, pipelining improves instruction *throughput* rather than individual instruction *execution time* or *latency*.

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



PREDICTION

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

4.6 Pipelined Datapath and Control

There is less in this than meets the eye.

Tallulah Bankhead, remark to Alexander Woollcott, 1922

Figure 4.33 shows the single-cycle datapath from Section 4.4 with the pipeline stages identified. The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus, we must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

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 4.33, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the

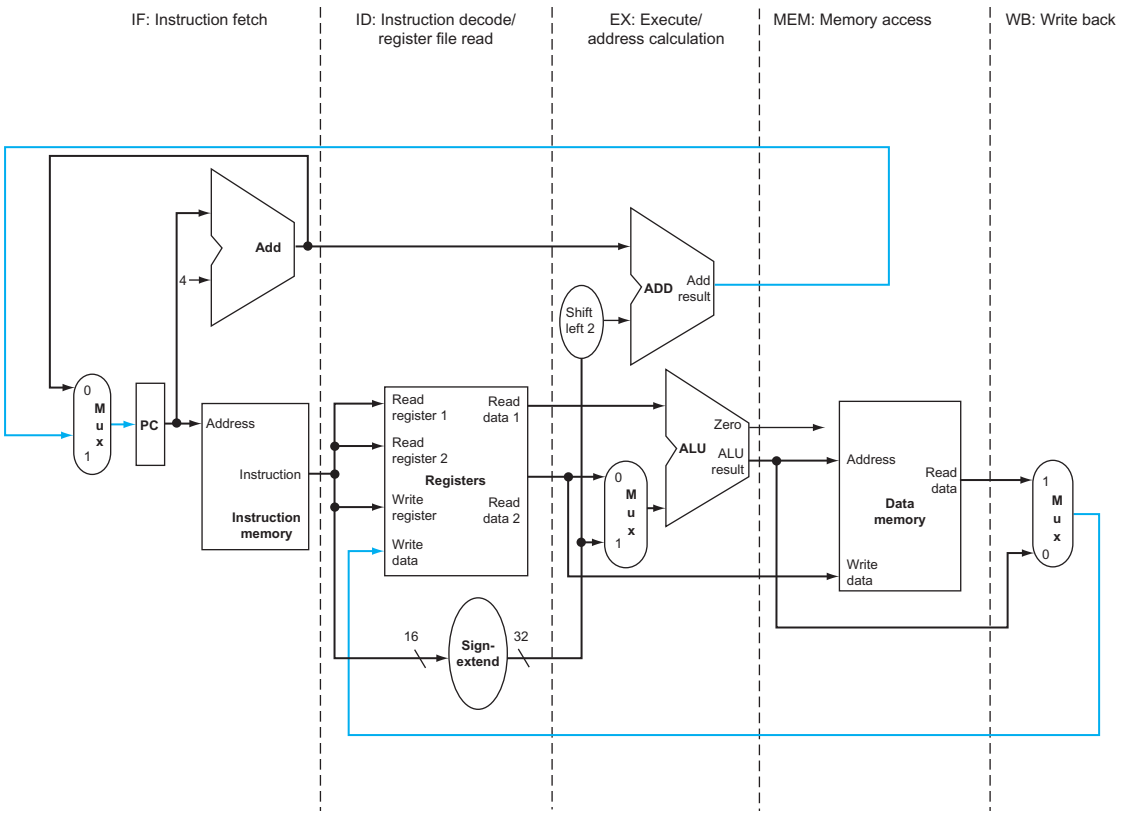


FIGURE 4.33 The single-cycle datapath from Section 4.4 (similar to Figure 4.17). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

five stages as they complete execution. Returning to our laundry analogy, clothes get cleaner, drier, and more organized as they move through the line, and they never move backward.

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; these reverse data movements influence only later instructions in the pipeline. Note that

the first right-to-left flow of data 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 timeline to show their relationship. Figure 4.34 shows the execution of the instructions in Figure 4.27 by displaying their private datapaths on a common timeline. We use a stylized version of the datapath in Figure 4.33 to show the relationships in Figure 4.34.

Figure 4.34 seems to suggest that three instructions need three datapaths. Instead, we add registers to hold data so that portions of a single datapath can be shared during instruction execution.

For example, as Figure 4.34 shows, the instruction memory is used during only one of the five stages of an instruction, allowing it to be shared by following 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 4.33. Returning to our laundry analogy, we might have a basket between each pair of stages to hold the clothes for the next step.

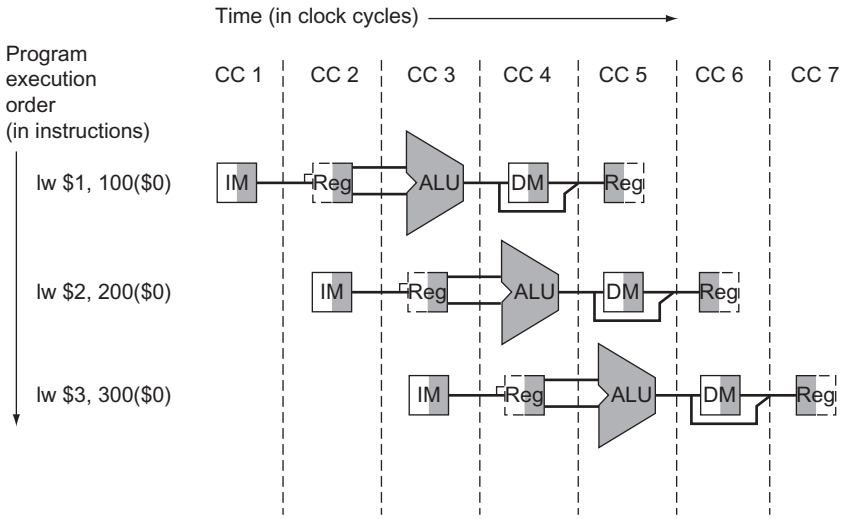


FIGURE 4.34 Instructions being executed using the single-cycle datapath in Figure 4.33, assuming pipelined execution. Similar to Figures 4.28 through 4.30, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 4.33. *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.

Figure 4.35 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 registers in Figure 4.35, 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

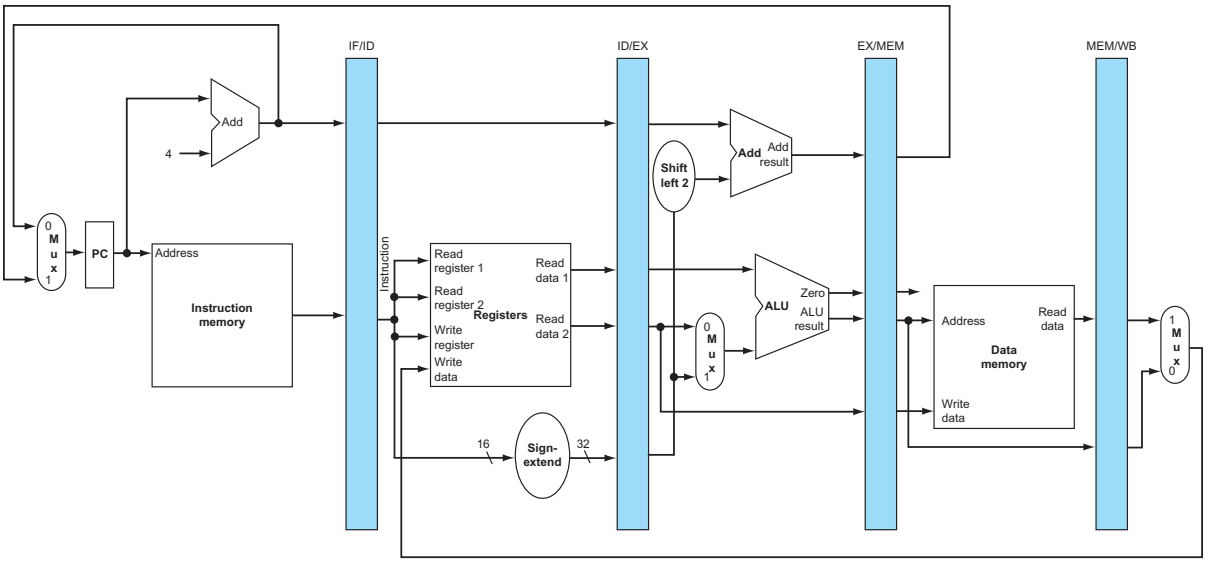


FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The 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.

less time than it might appear, because you can compare them to see what changes occur in each clock cycle. Section 4.7 describes what happens when there are data hazards between pipelined instructions; ignore them for now.

Figures 4.36 through 4.38, our first sequence, show the active portions of the datapath highlighted as a load instruction goes through the five stages of pipelined execution. We show a load first because it is active in all five stages. As in Figures 4.28 through 4.30, we highlight the *right half* of registers or memory when they are being *read* and highlight the *left half* when they are being *written*.

We show the instruction abbreviation `lw` with the name of the pipe stage that is active in each figure. The five stages are the following:

1. *Instruction fetch:* The top portion of Figure 4.36 shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register. 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.

2. *Instruction decode and register file read:* The bottom portion of Figure 4.36 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 4.37 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 4.38 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.

5. *Write-back:* The bottom portion of Figure 4.38 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:

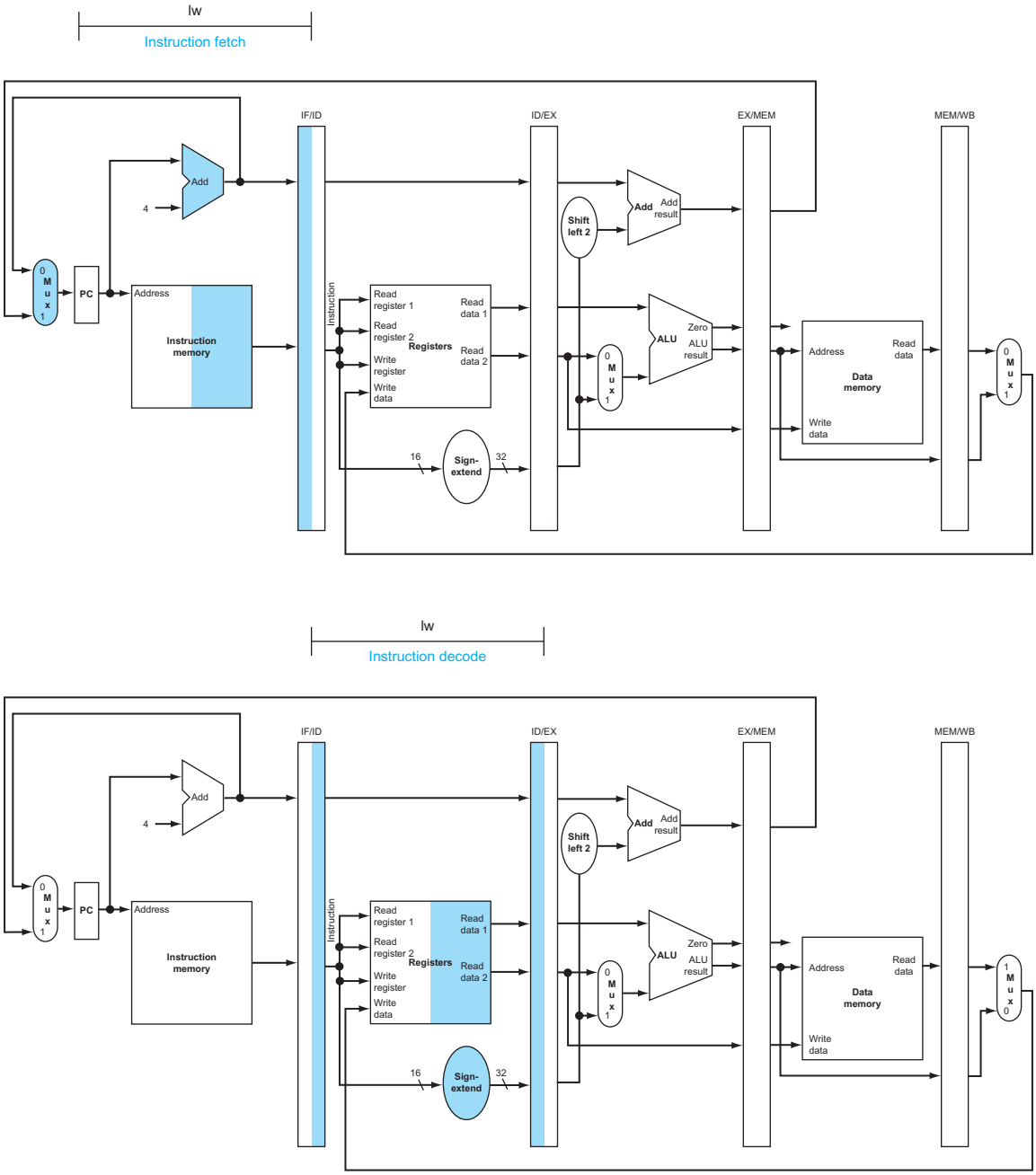


FIGURE 4.36 IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in Figure 4.35 highlighted. The highlighting convention is the same as that used in Figure 4.28. As in Section 4.2, there is no confusion when reading and writing registers, because the contents 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.

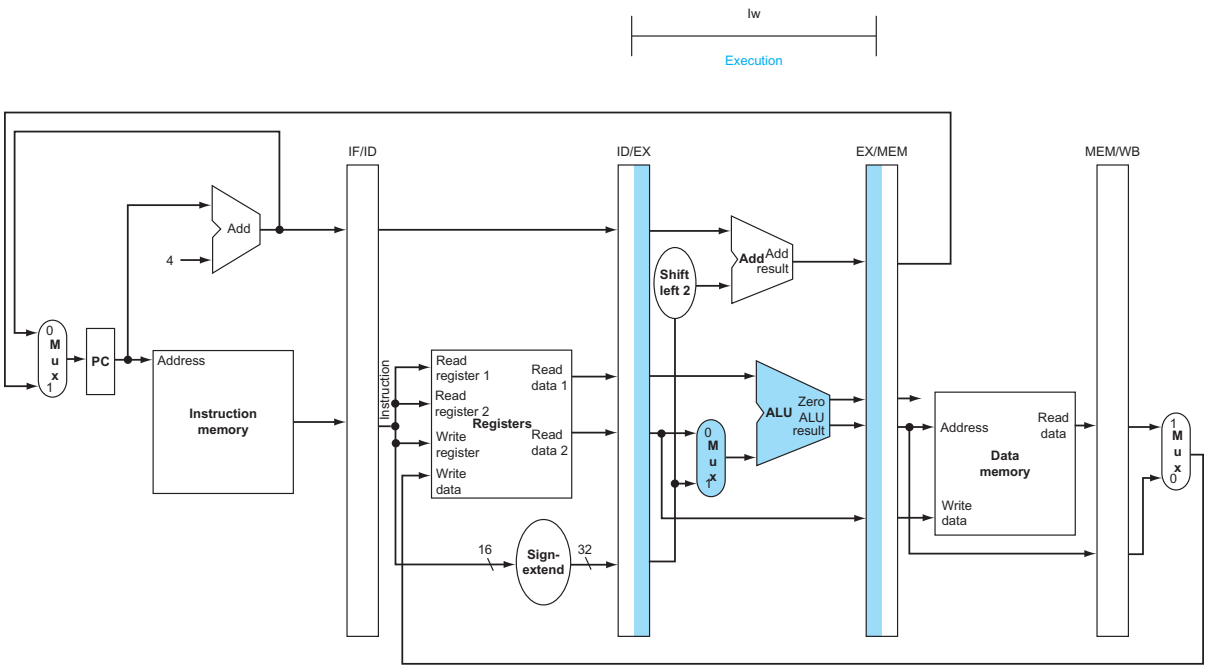


FIGURE 4.37 EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 4.35 used in this pipe stage. The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.

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 4.36 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 4.36 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 4.39 shows the third step; the effective address is placed in the EX/MEM pipeline register.
4. *Memory access:* The top portion of Figure 4.40 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.

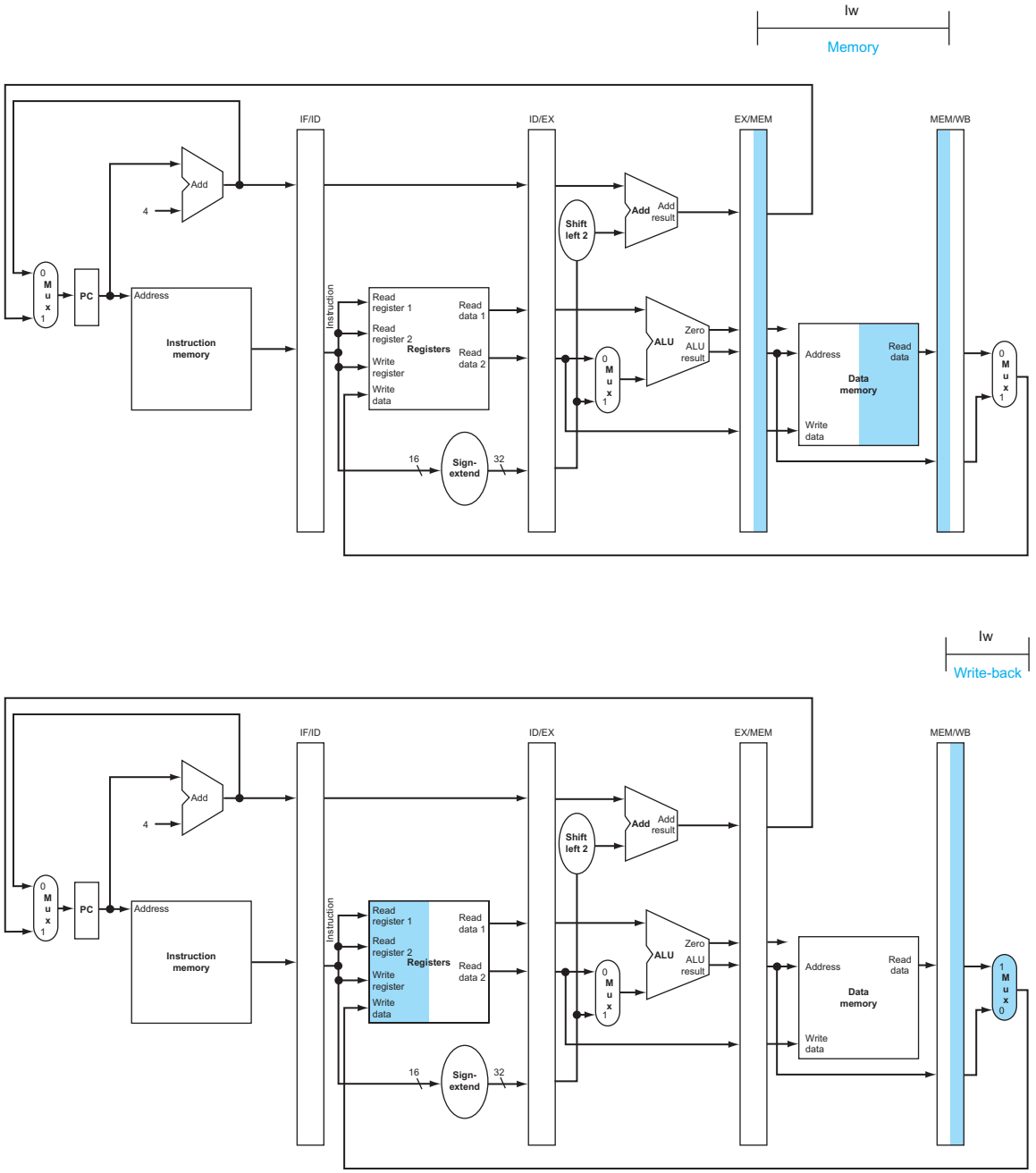


FIGURE 4.38 MEM and WB: The fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 4.35 used in this pipe stage. Data memory is read using the address in the EX/MEM pipeline registers, and the data is placed in the MEM/WB pipeline register. Next, data is read from the MEM/WB pipeline register and written into the register file in the middle of the datapath. Note: there is a bug in this design that is repaired in Figure 4.41.

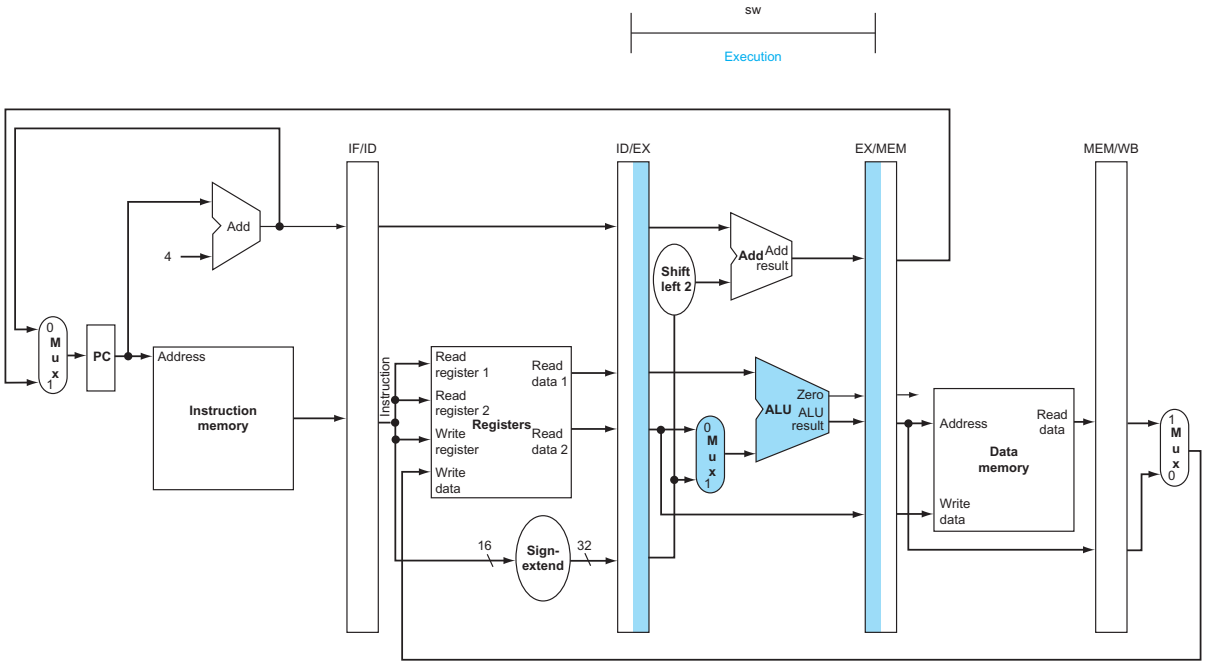


FIGURE 4.39 EX: The third pipe stage of a store instruction. Unlike the third stage of the load instruction in Figure 4.37, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.

5. *Write-back:* The bottom portion of Figure 4.40 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.

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 277). 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

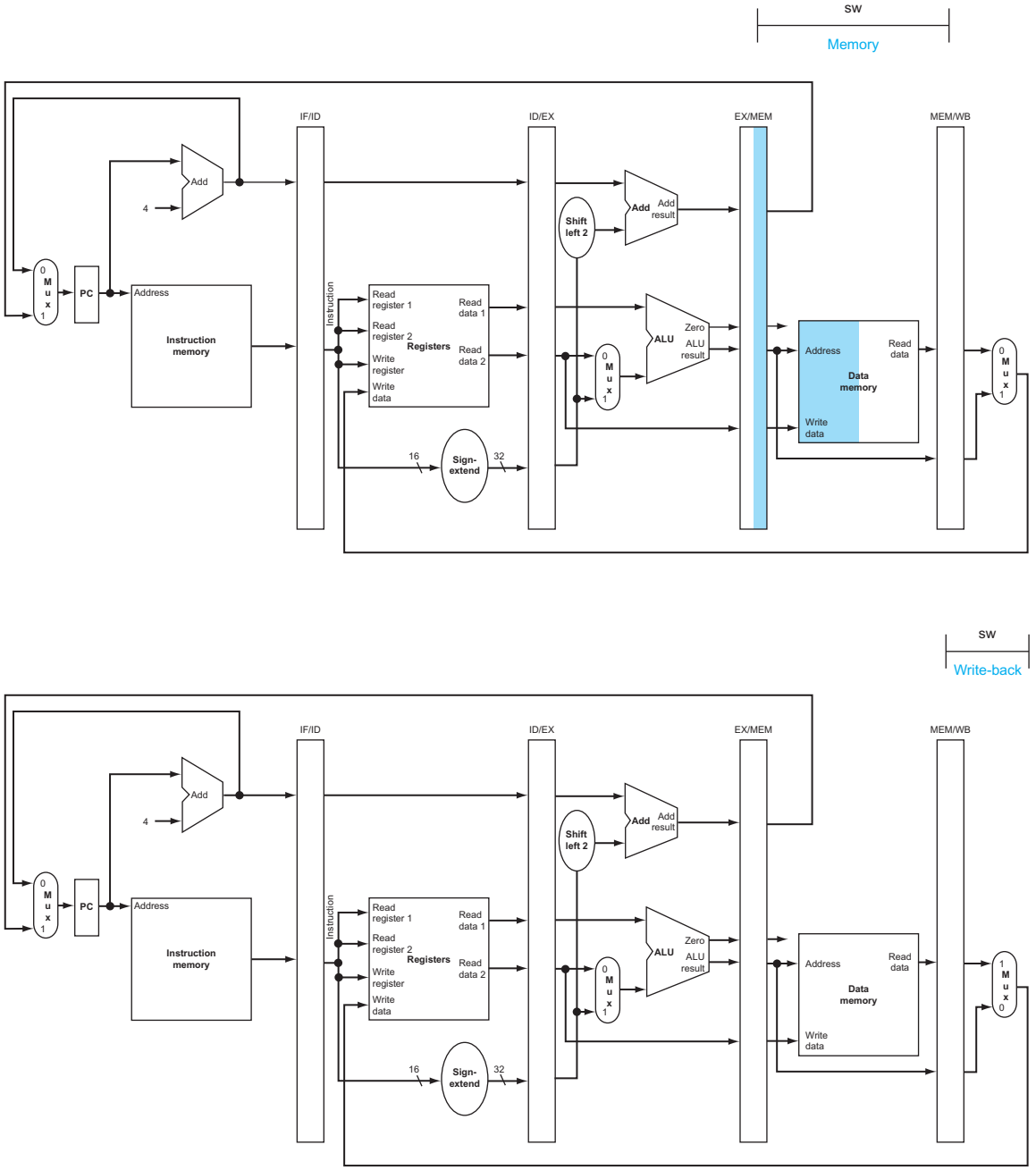


FIGURE 4.40 MEM and WB: The fourth and fifth pipe stages of a store instruction. In the fourth stage, the data is written into data memory for the store. Note that the data comes from the EX/MEM pipeline register and that nothing is changed in the MEM/WB pipeline register. Once the data is written in memory, there is nothing left for the store instruction to do, so nothing happens in stage 5.

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 the WB stage. Another way to think about the passing of the register number is that to share the pipelined datapath, we need 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 4.41 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 4.42 is a single drawing of the corrected datapath, highlighting the hardware used in all five stages of the load word instruction in Figures 4.36 through 4.38. See Section 4.8 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

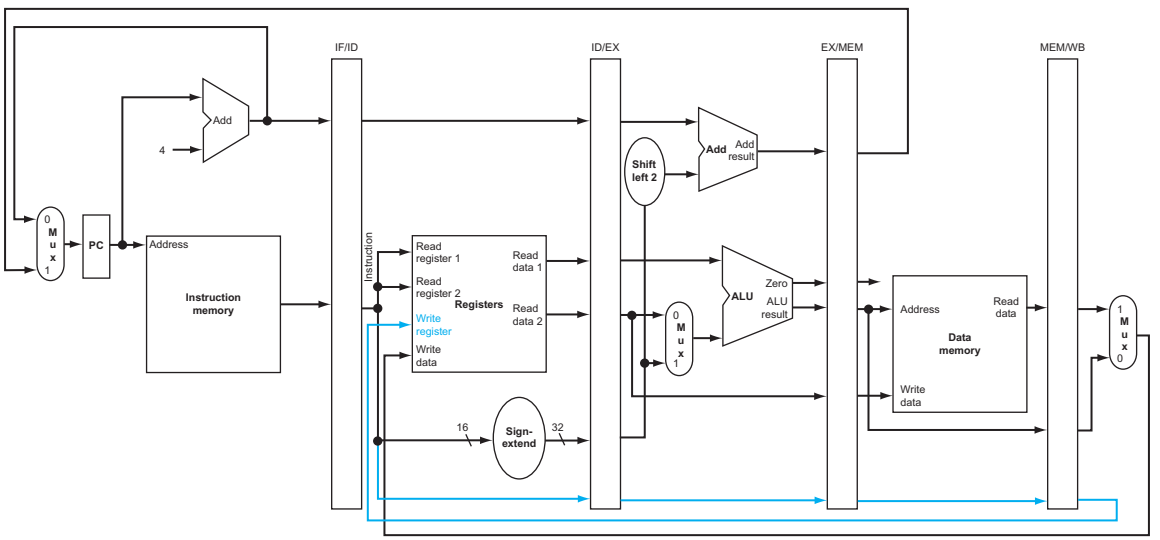


FIGURE 4.41 The corrected pipelined datapath to handle the load instruction properly. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits to the last three pipeline registers. This new path is shown in color.

two basic styles of pipeline figures: *multiple-clock-cycle pipeline diagrams*, such as Figure 4.34 on page 288, and *single-clock-cycle pipeline diagrams*, such as Figures 4.36 through 4.40. The multiple-clock-cycle diagrams are simpler but do not contain all the details. For example, consider the following five-instruction sequence:

```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```

Figure 4.43 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 4.25. 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 graphically, but a rectangle naming each pipe stage works just as well. Figure 4.44 shows the more traditional version of the multiple-clock-cycle pipeline diagram. Note that Figure 4.43 shows the physical resources used at each stage, while Figure 4.44 uses the *name* of each stage.

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,

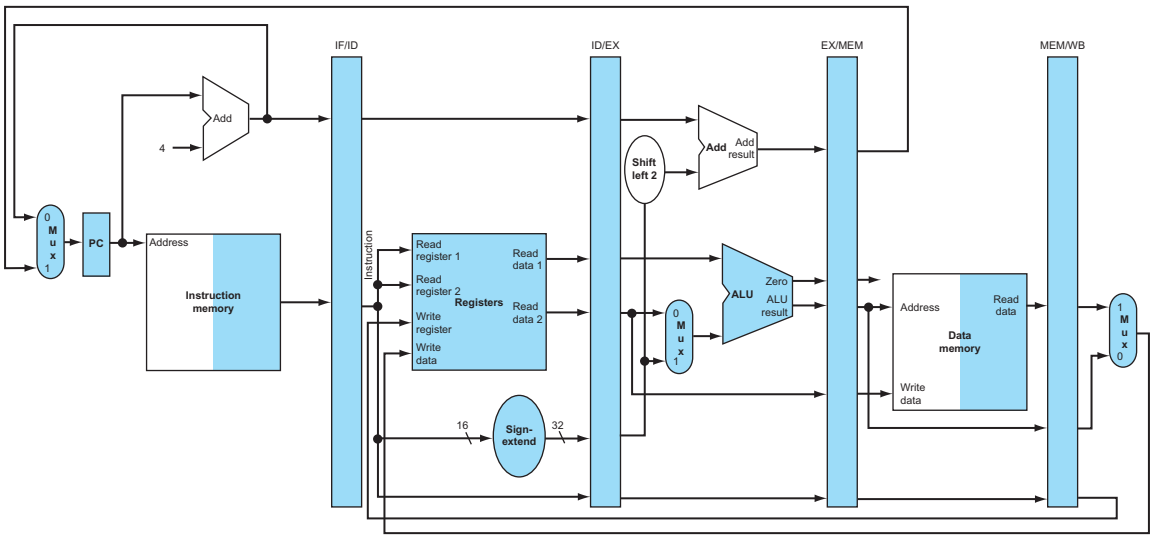


FIGURE 4.42 The portion of the datapath in Figure 4.41 that is used in all five stages of a load instruction.

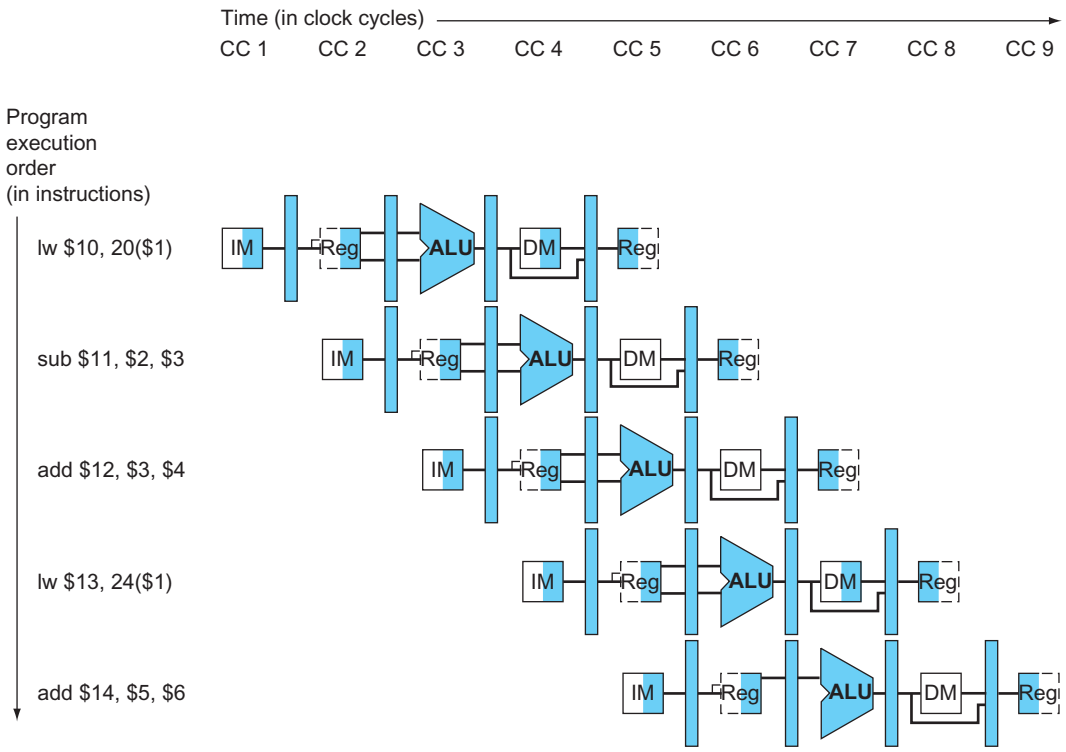


FIGURE 4.43 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 4.28, here we show the pipeline registers between each stage. Figure 4.44 shows the traditional way to draw this diagram.

the drawings appear in groups to show pipeline operation over a sequence of clock cycles. We use multiple-clock-cycle diagrams to give overviews of pipelining situations. (🌐 **Section 4.13** gives more illustrations of single-clock diagrams if you would like to see more details about Figure 4.43.) A single-clock-cycle diagram represents a vertical slice through a set of multiple-clock-cycle diagrams, showing the usage of the datapath by each of the instructions in the pipeline at the designated clock cycle. For example, Figure 4.45 shows the single-clock-cycle diagram corresponding to clock cycle 5 of Figures 4.43 and 4.44. Obviously, the single-clock-cycle diagrams have more detail and take significantly more space to show the same number of clock cycles. The exercises ask you to create such diagrams for other code sequences.

Check Yourself A group of students were 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 four statements. Which ones are correct?

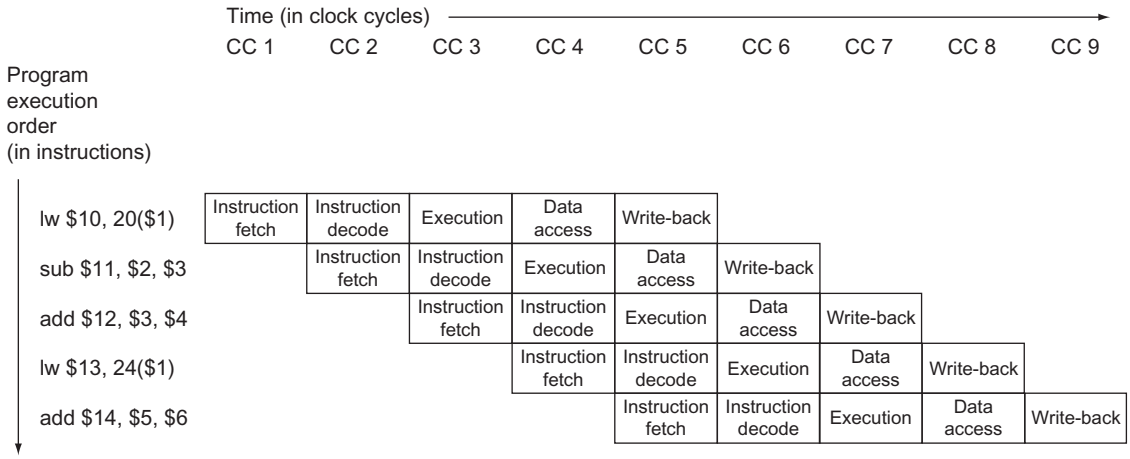


FIGURE 4.44 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.43.

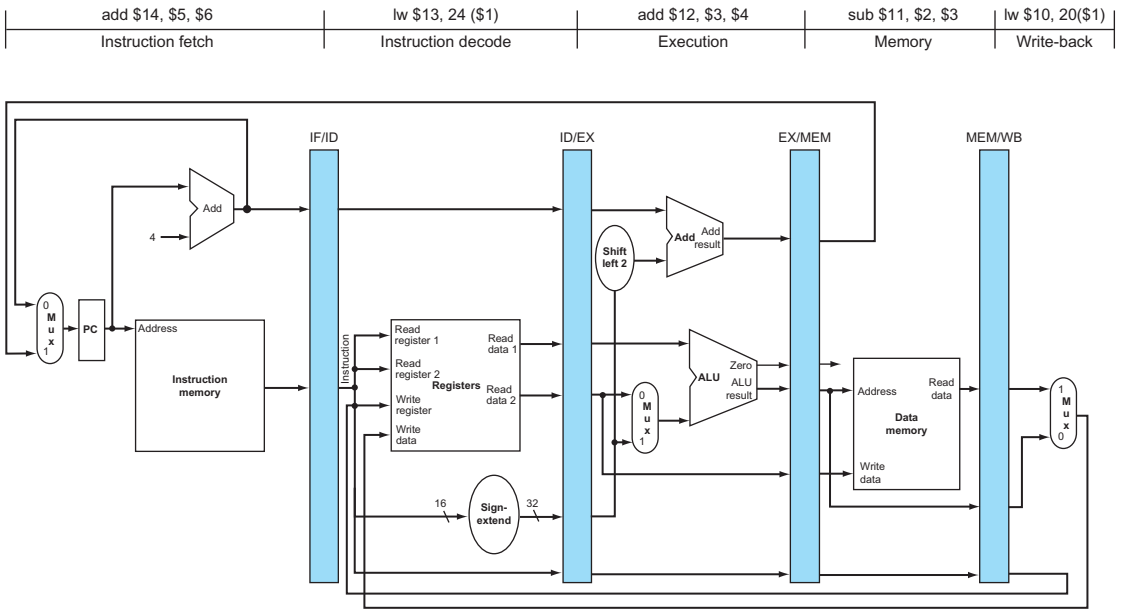


FIGURE 4.45 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.43 and 4.44. As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

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. 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.
4. 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.

Pipelined Control

In the 6600 Computer, perhaps even more than in any previous computer, the control system is the difference.

James Thornton, *Design of a Computer: The Control Data 6600,* 1970

Just as we added control to the single-cycle datapath in Section 4.3, we now add control to the pipelined datapath. We start with a simple design that views the problem through rose-colored glasses.

The first step is to label the control lines on the existing datapath. Figure 4.46 shows those lines. We borrow as much as we can from the control for the simple datapath in Figure 4.17. In particular, we use the same ALU control logic, branch logic, destination-register-number multiplexor, and control lines. These functions are defined in Figures 4.12, 4.16, and 4.18. We reproduce the key information in Figures 4.47 through 4.49 on a single page to make the following discussion easier to follow.

As was the case for the single-cycle implementation, we assume that the PC is written on each clock cycle, so there is no separate write signal for the 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.
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 4.47 and 4.48). The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.

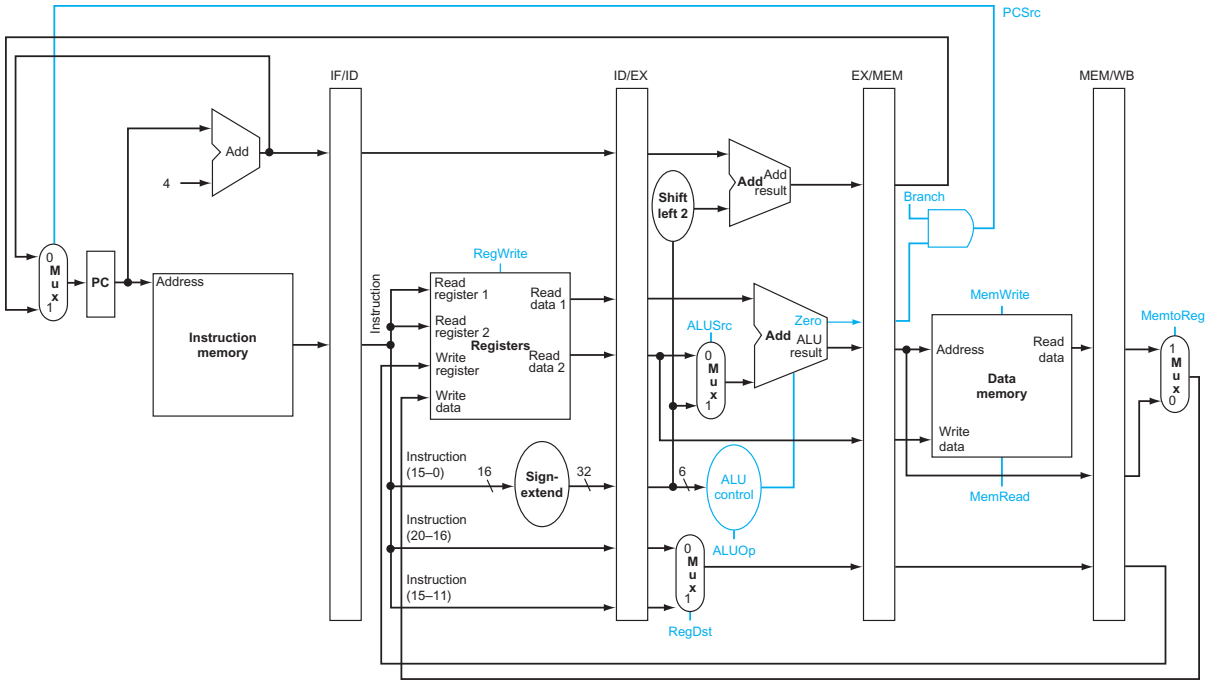


FIGURE 4.46 The pipelined datapath of Figure 4.41 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Section 4.4. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as 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.

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 4.47 A copy of Figure 4.12. 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 4.48 A copy of Figure 4.16. The function of each of seven control signals is defined. The ALU control lines (ALUOp) are defined in the second column of Figure 4.47. When a 1-bit control to a 2-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 4.46. 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.

	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-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 4.49 The values of the control lines are the same as in Figure 4.18, but they have been shuffled into three groups corresponding to the last three pipeline stages.

4. *Memory access:* The control lines set in this stage are Branch, MemRead, and MemWrite. The branch equal, load, and store instructions set these signals, respectively. Recall that PCSrc in Figure 4.48 selects the next sequential address unless control asserts Branch and the ALU result was 0.

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. Figure 4.49 has the same values as in Section 4.4, but now the nine control lines are grouped by pipeline stage.

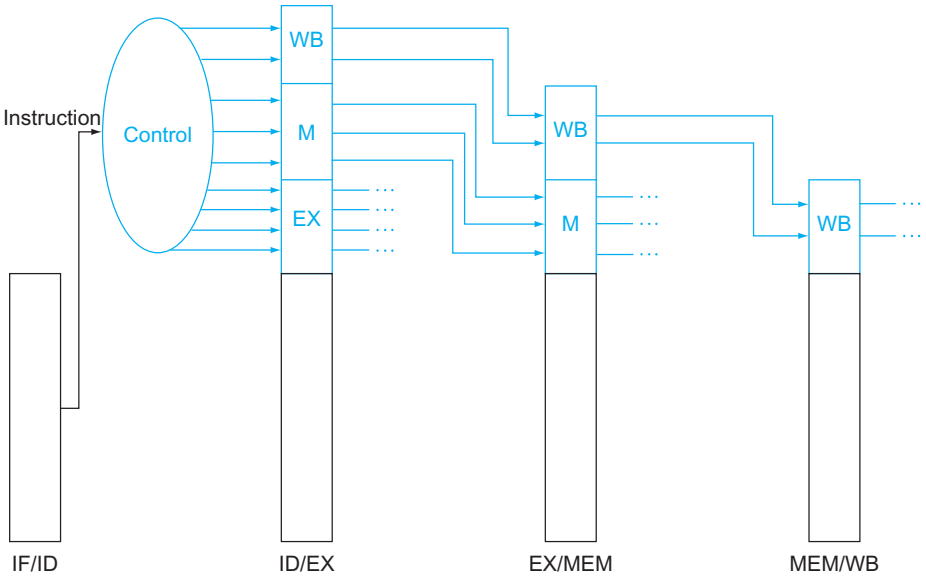


FIGURE 4.50 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

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 4.50 above 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 4.41. Figure 4.51 shows the full datapath with the extended pipeline registers and with the control lines connected to the proper stage. (🌐 **Section 4.13** gives more examples of MIPS code executing on pipelined hardware using single-clock diagrams, if you would like to see more details.)

4.7 Data Hazards: Forwarding versus Stalling

What do you mean, why's it got to be built? It's a bypass. You've got to build bypasses.

Douglas Adams, *The Hitchhiker's Guide to the Galaxy,* 1979

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 4.43 through 4.45 were independent; none of them used the results calculated by any of the others. Yet in Section 4.5, we saw that data hazards are obstacles to pipelined execution.

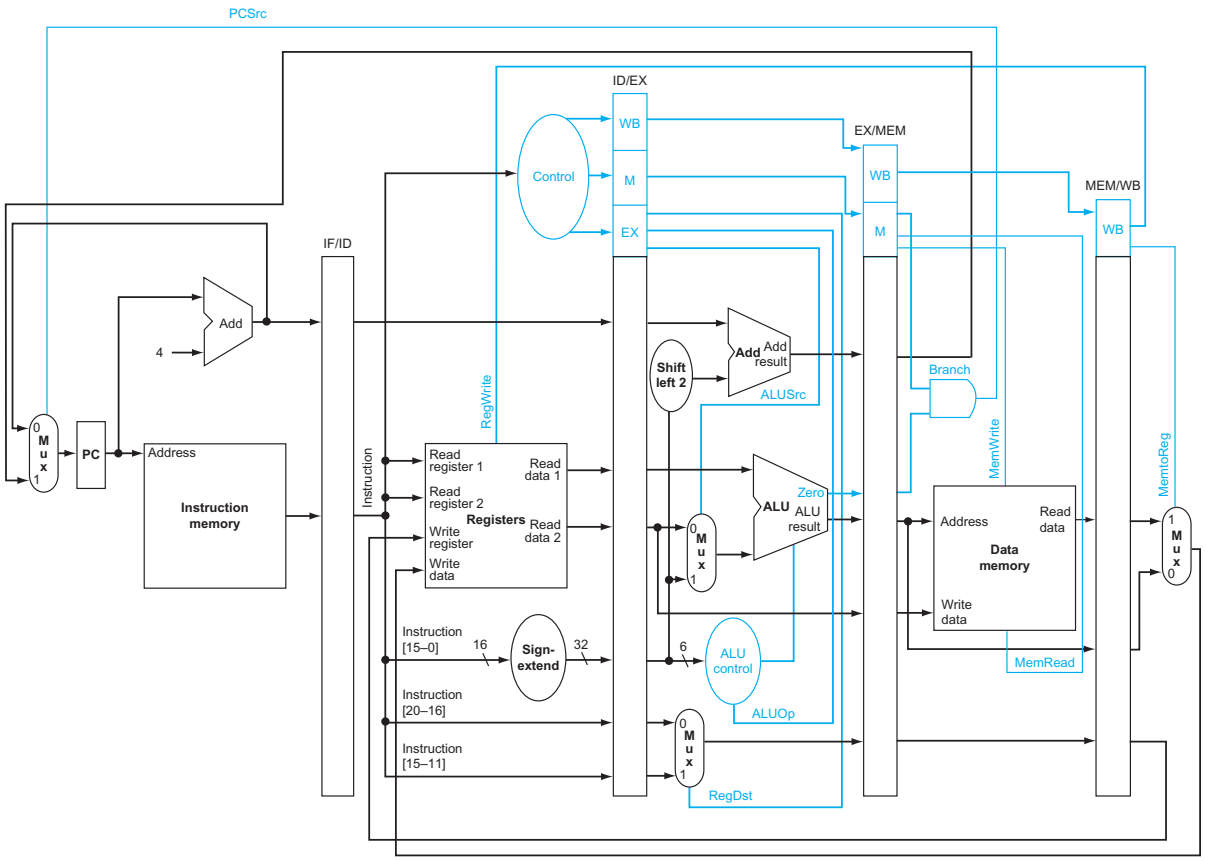


FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Let's look at a sequence with many dependences, shown in color:

```
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 4.52 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 4.52 shows the value of register `$2`, which changes during the middle of clock cycle 5, when the `sub` instruction writes its result.

The last 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 4.52 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

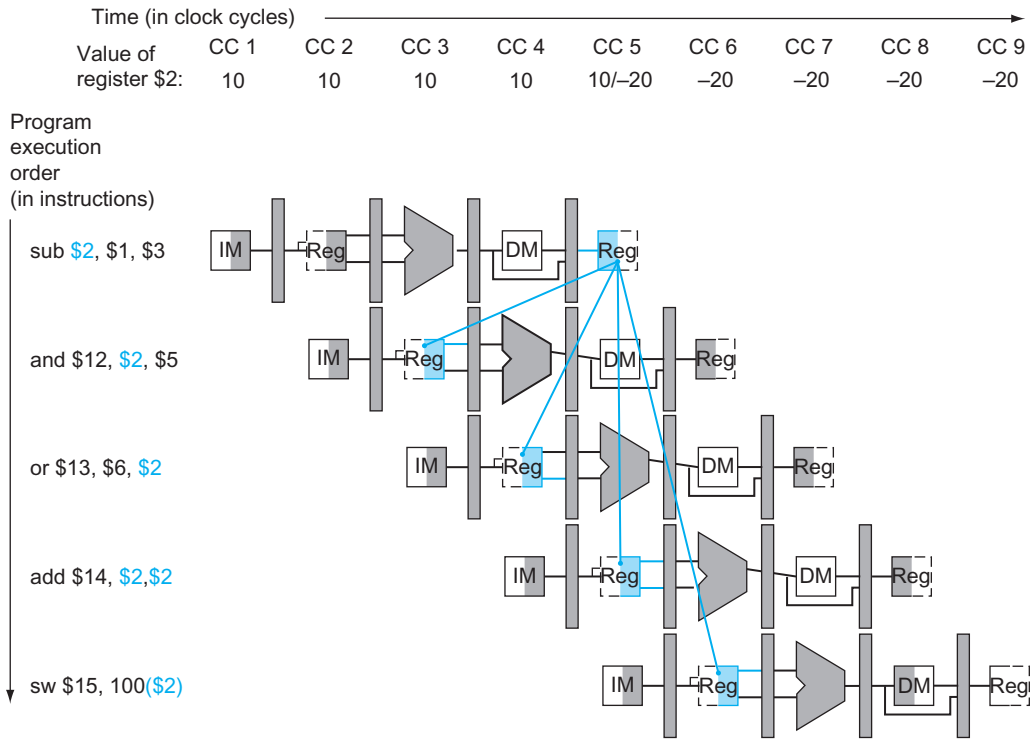


FIGURE 4.52 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into `$2`, and all the following instructions read `$2`. 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 backward in time are *pipeline data hazards*.

OR instructions would get the incorrect value 10! Using this style of drawing, such problems become apparent when a dependence line goes backward in time.

As mentioned in Section 4.5, the desired 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 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 304 is on register $2, between the result of sub $2,$1,$3 and the first read operand of and $12,$2,$5. This hazard can be detected when the and instruction is in the EX stage and the prior instruction is in the MEM stage, so this is hazard 1a:

EX/MEM.RegisterRd = ID/EX.RegisterRs = $2

EXAMPLE

Dependence Detection

Classify the dependences in this sequence from page 304:

```
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 as follows:

- The `sub-or` is a type 2b hazard:

  MEM/WB.RegisterRd = ID/EX.RegisterRt = `$2`

- The two dependences on `sub-add` are not hazards because the register file supplies the proper data during the ID stage of `add`.
- There is no data hazard between `sub` and `sw` because `sw` reads `$2` the clock cycle *after* `sub` writes `$2`.

Because some instructions do not write registers, this policy is inaccurate; sometimes it would forward when it shouldn't. 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 whether RegWrite is asserted. Recall that MIPS requires that every use of `$0` as an operand must yield an operand value of 0. In the event that an instruction in the pipeline has `$0` as its destination (for example, `sll $0, $1, 2`), we want to avoid forwarding its possibly nonzero result value. Not forwarding results destined for `$0` frees the assembly programmer and the compiler of any requirement to avoid using `$0` as a destination. The conditions above thus work properly as long we add EX/MEM.RegisterRd ≠ 0 to the first hazard condition and MEM/WB.RegisterRd ≠ 0 to the second.

Now that we can detect hazards, half of the problem is resolved—but we must still forward the proper data.

Figure 4.53 shows the dependences between the pipeline registers and the inputs to the ALU for the same code sequence as in Figure 4.52. 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 4.54 shows a close-up of the ALU and pipeline register before and after adding forwarding. Figure 4.55 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.

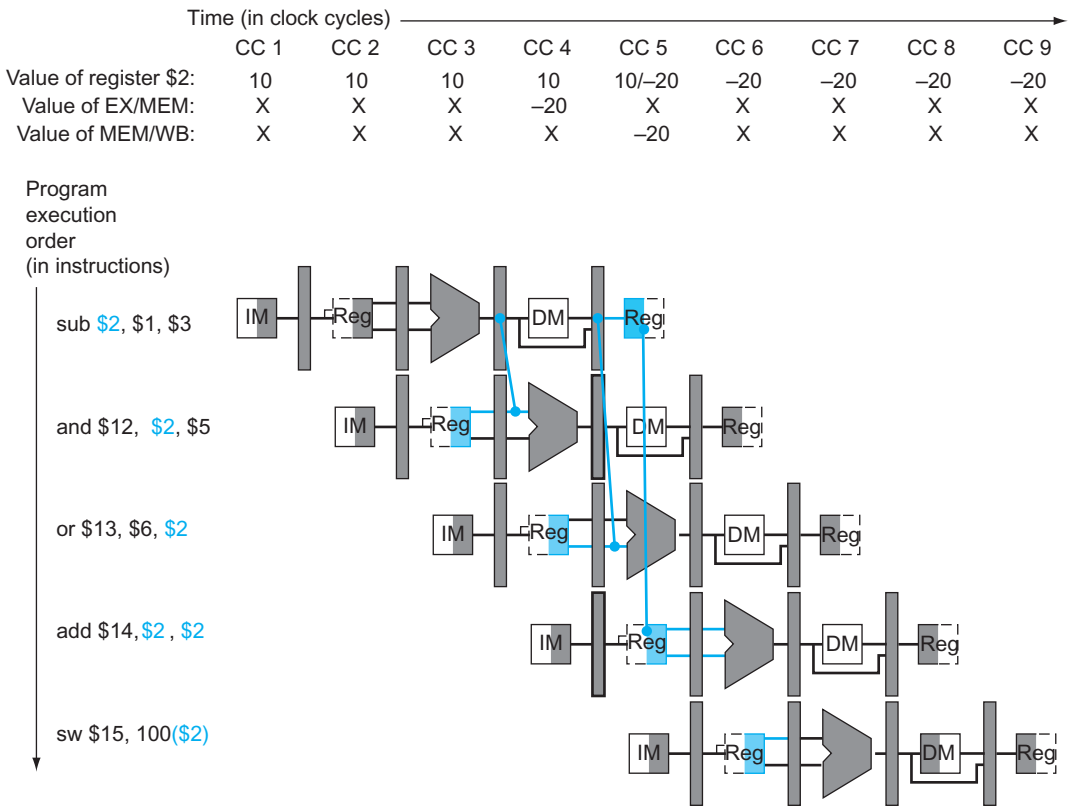


FIGURE 4.53 The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the `AND` instruction and `OR` instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired 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.

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

1. *EX hazard:*

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

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

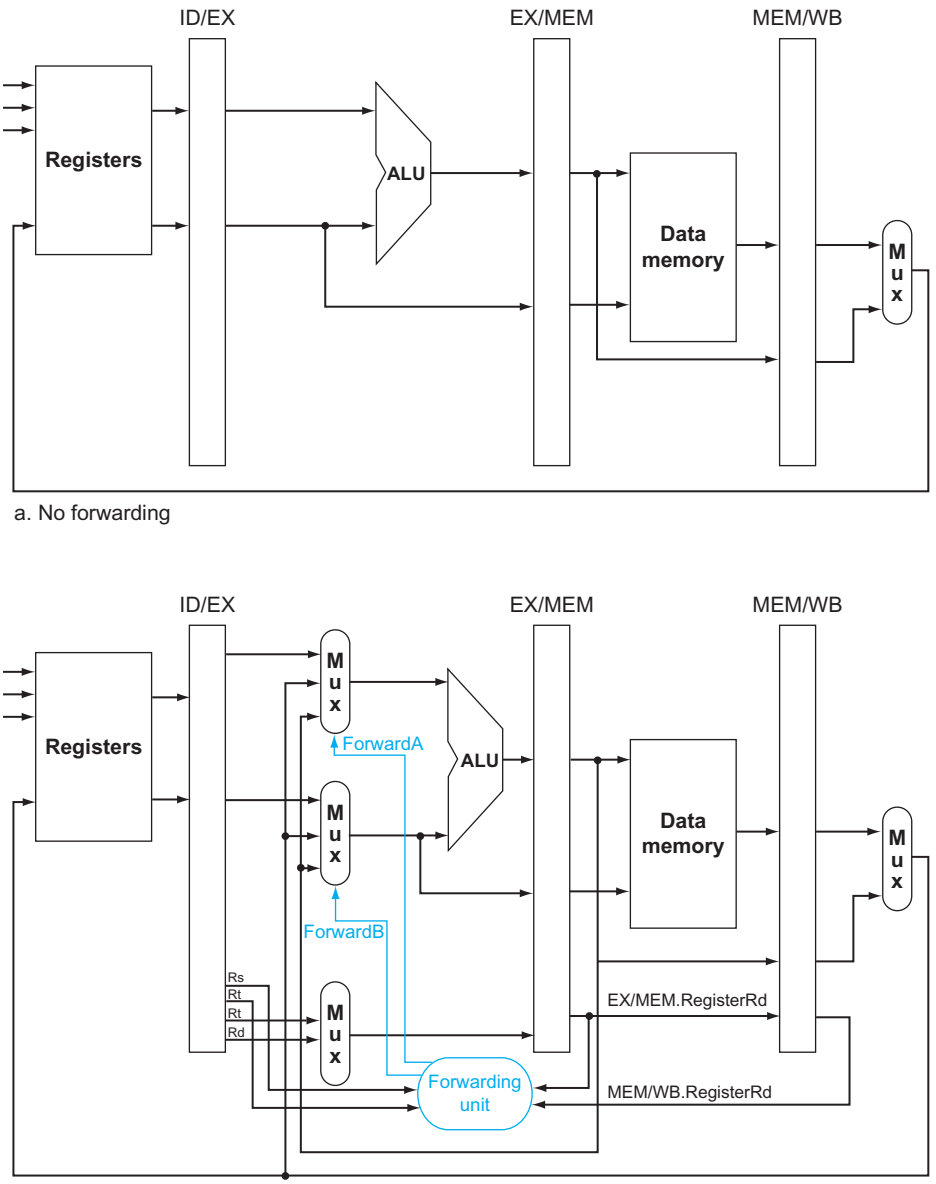


FIGURE 4.54 On the top are the ALU and pipeline registers before adding forwarding. On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from 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. Also note that this mechanism works for `slt` instructions as well.

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 4.55 The control values for the forwarding multiplexors in Figure 4.54. The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

Note that the EX/MEM.RegisterRd field is the register destination for either an ALU instruction (which comes from the Rd field of the instruction) or a load (which comes from the Rt field).

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.

2. *MEM hazard:*

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

As mentioned above, there is no hazard in the WB stage, because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage. Such a register file performs another form of forwarding, but it occurs within the register file.

One complication is potential data hazards between the result of the instruction in the WB stage, the result of the instruction in the MEM stage, and the source operand of the instruction in the ALU stage. For example, when summing a vector of numbers in a single register, a sequence of instructions will all read and write to the same register:

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

In this case, the result is forwarded from the MEM stage because the result in the MEM stage is the more recent result. Thus, the control for the MEM hazard would be (with the additions highlighted):

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
       and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
       and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Figure 4.56 shows the hardware necessary to support forwarding for operations that use results during the EX stage. Note that the EX/MEM.RegisterRd field is the register destination for either an ALU instruction (which comes from the Rd field of the instruction) or a load (which comes from the Rt field).

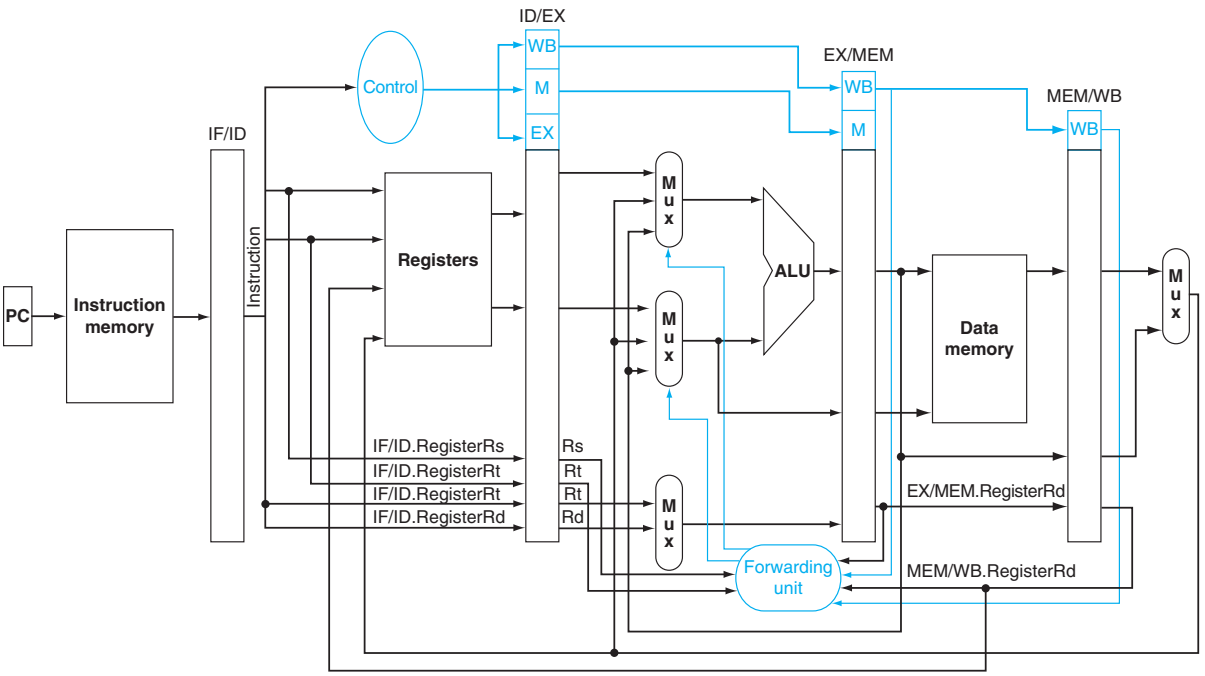


FIGURE 4.56 The datapath modified to resolve hazards via forwarding. Compared with the datapath in Figure 4.51, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath, such as the branch hardware and the sign extension hardware.

Section 4.13 shows two pieces of MIPS code with hazards that cause forwarding, if you would like to see more illustrated examples using single-cycle pipeline drawings.

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. However, consider loads immediately followed by stores, useful when performing memory-to-memory copies in the MIPS architecture. Since copies are frequent, we need to add more forwarding hardware to make them run faster. If we were to redraw Figure 4.53, replacing the `sub` and `AND` instructions with `lw` and `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 to the reader.

In addition, the signed-immediate input to the ALU, needed by loads and stores, is missing from the datapath in Figure 4.56. Since central control decides between register and immediate, and since the forwarding unit chooses the pipeline register for a register

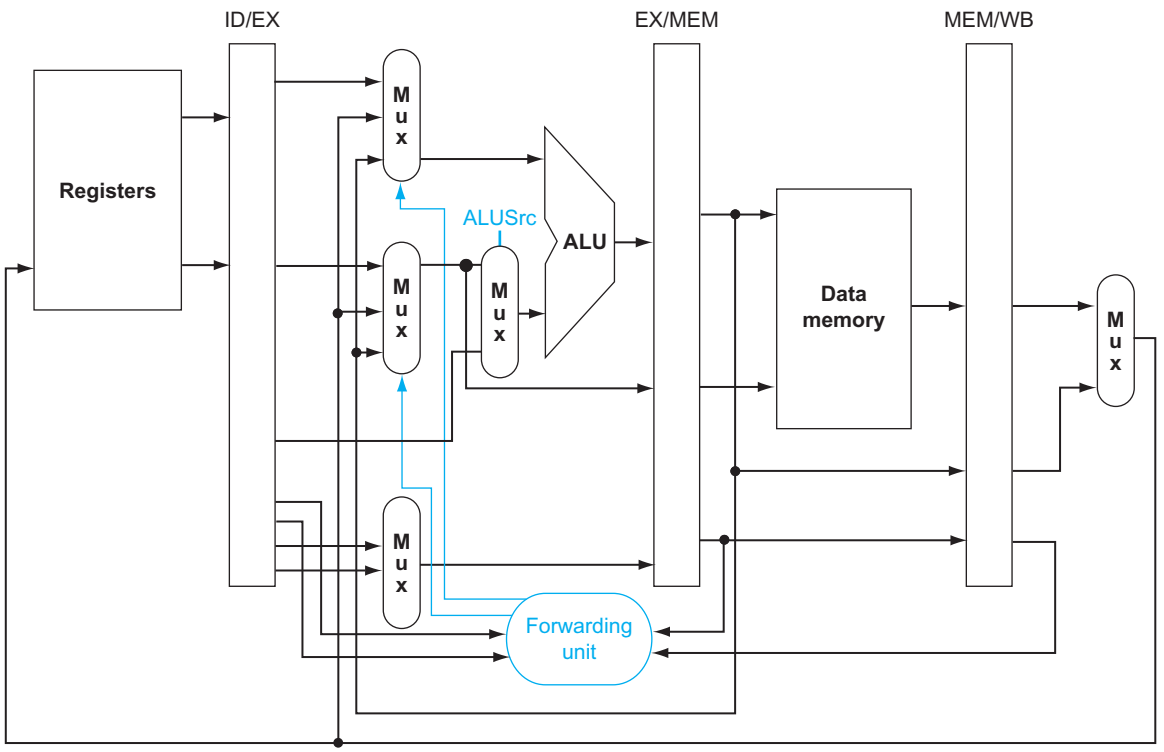


FIGURE 4.57 A close-up of the datapath in Figure 4.54 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.

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 4.57 shows this addition.

Data Hazards and Stalls

If at first you don't succeed, redefine success.

Anonymous

As we said in Section 4.5, 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 4.58 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

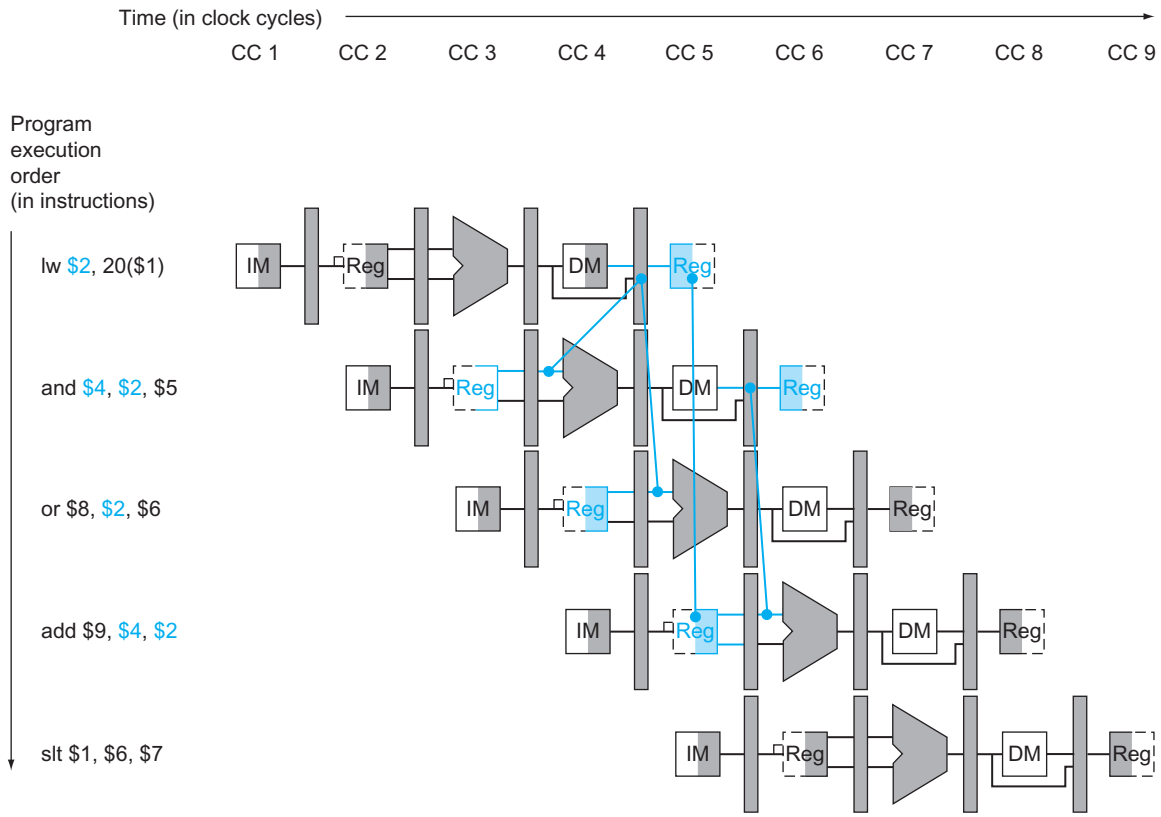


FIGURE 4.58 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (`and`) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

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 one 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 4.58 would need another stall cycle.)

If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled; otherwise, we would lose the fetched instruction. Preventing these two instructions from making progress is accomplished simply by preventing the PC register and the IF/ID pipeline register from changing. Provided these registers are preserved, the instruction in the IF stage will continue to be read using the same PC, and the registers in the ID stage will continue to be read using the same instruction fields in the IF/ID pipeline register. Returning to our favorite analogy, it's as if you restart the washer with the same clothes and let the dryer continue tumbling empty. Of course, like the dryer, the back half of the pipeline starting with the EX stage must be doing something; what it is doing is executing instructions that have no effect: **nops**.

nop An instruction that does no operation to change state.

How can we insert these nops, which act like bubbles, into the pipeline? In Figure 4.49, 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 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 4.59 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. 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. In this example, 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.

Figure 4.60 highlights the pipeline connections for both the hazard detection unit and the forwarding unit. As before, the forwarding unit controls the ALU

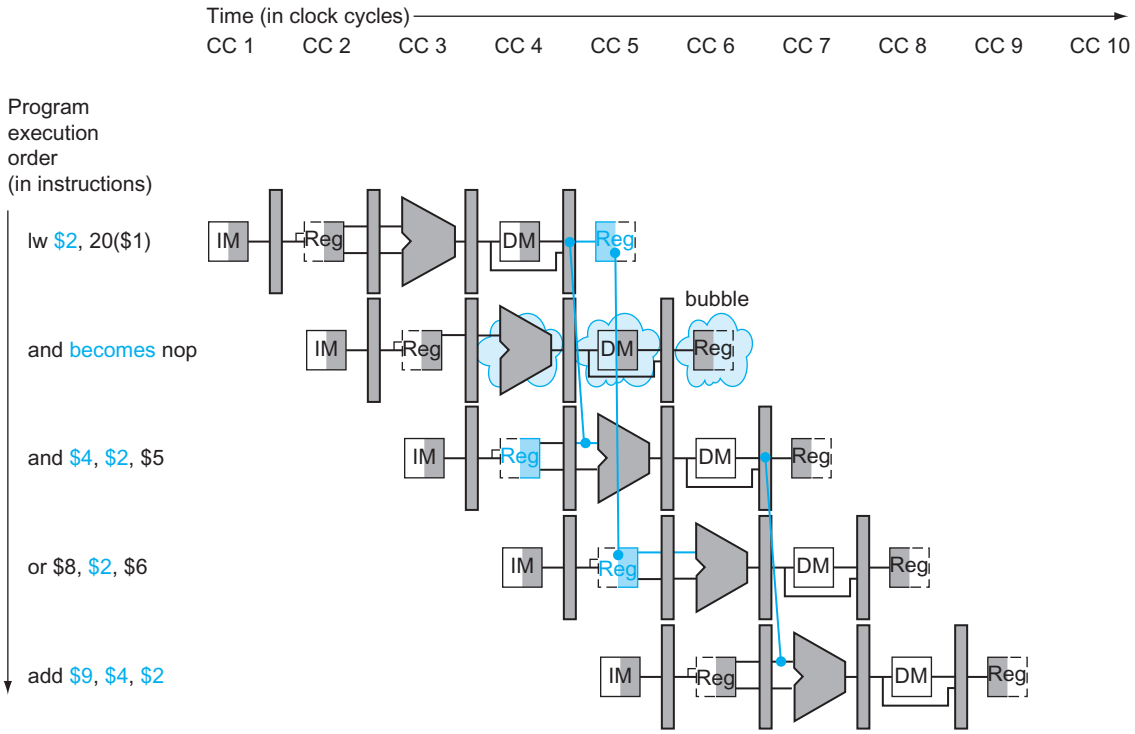


FIGURE 4.59 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the `and` instruction to a nop. Note that the `and` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise the `OR` instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

multiplexors to replace the value from a general-purpose register with the value 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. Section 4.13 gives an example of MIPS code with hazards that causes stalling, illustrated using single-clock pipeline diagrams, if you would like to see more details.

The BIG Picture

Although the compiler generally relies upon the hardware to resolve hazards and thereby 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.

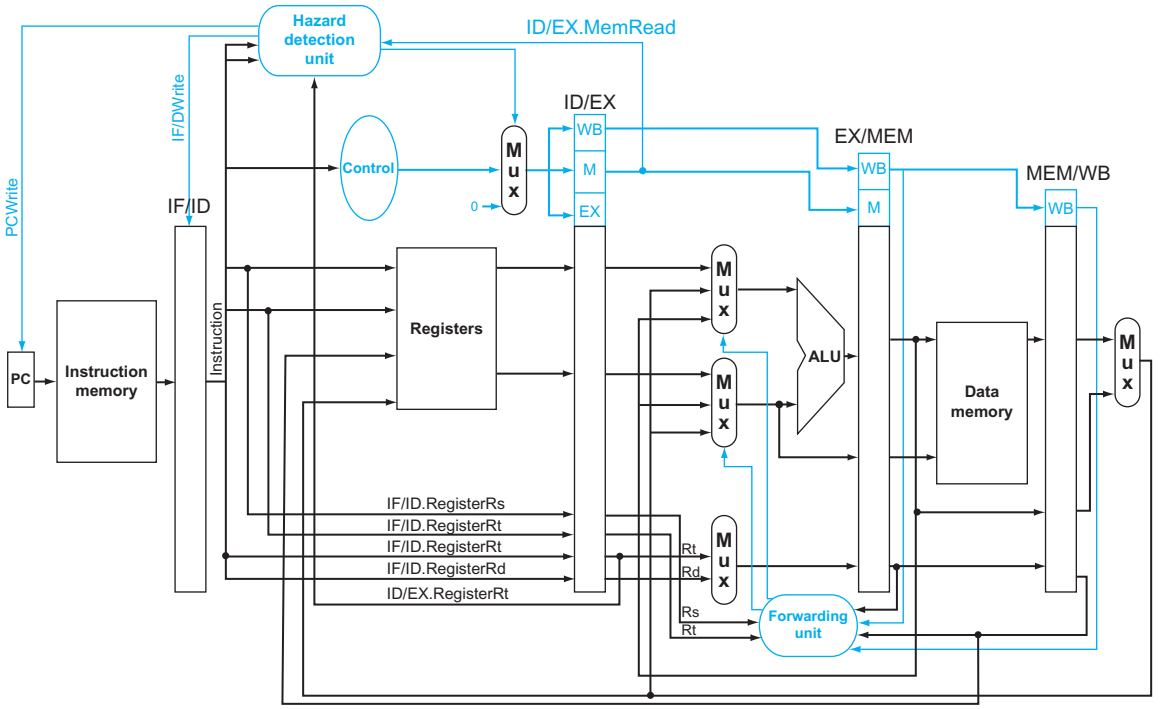


FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

Elaboration: Regarding the remark earlier about setting control lines to 0 to avoid writing registers or memory: only the signals RegWrite and MemWrite need be 0, while the other control signals can be don't cares.

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

Henry David Thoreau, *Walden*, 1854

4.8 Control Hazards

Thus far, we have limited our concern to hazards involving arithmetic operations and data transfers. However, as we saw in Section 4.5, there are also pipeline hazards involving branches. Figure 4.61 shows a sequence of instructions and indicates 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 4.5,

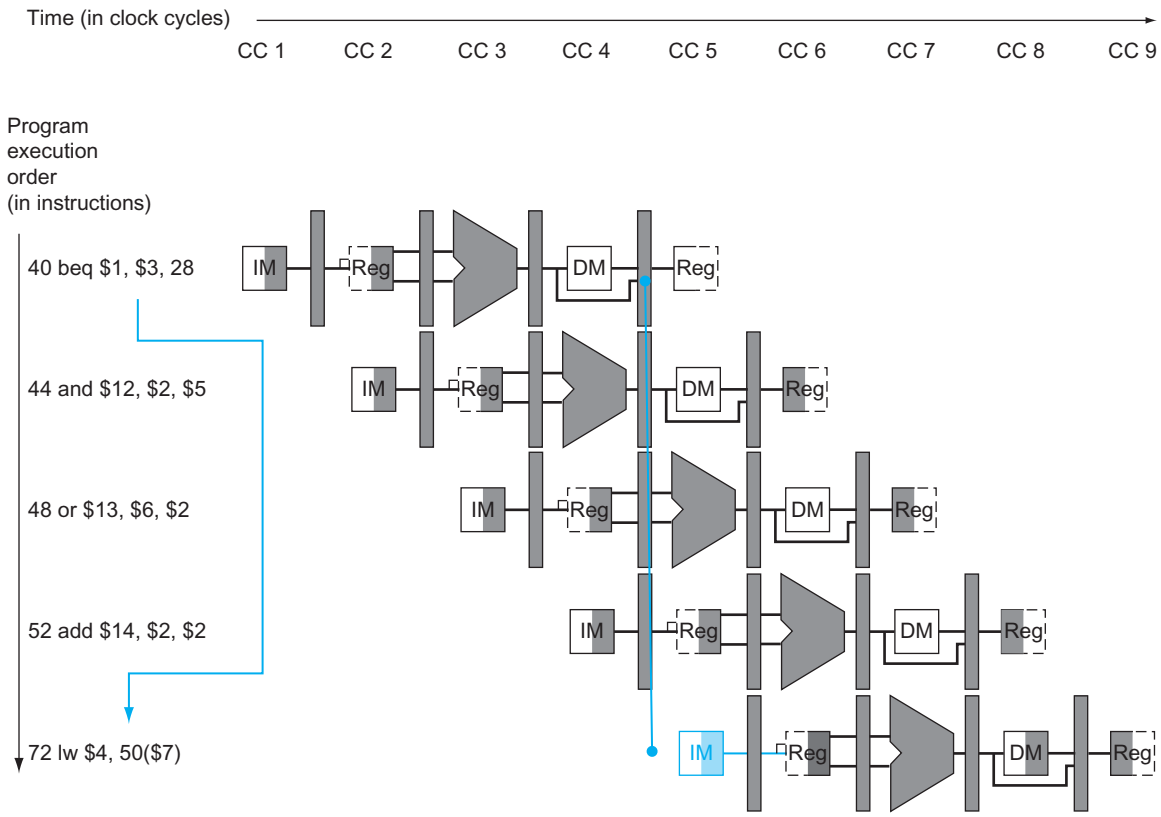
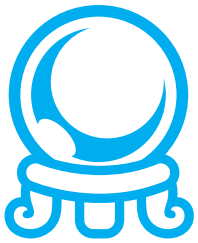


FIGURE 4.61 The impact of the pipeline on the branch instruction. The numbers to the left of the instruction (40, 44, …) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the `beq` instruction above—the three sequential instructions 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 4.31 assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

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 against data hazards. Hence, we use simpler schemes. We look at two schemes for resolving control hazards and one optimization to improve these schemes.



PREDICTION

flush To discard instructions in a pipeline, usually due to an unexpected event.

Assume Branch Not Taken

As we saw in Section 4.5, stalling until the branch is complete is too slow. One improvement over branch stalling is to **predict** 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 change 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.

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 (see Chapter 2).

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. By extension, if a load is immediately followed by a conditional branch that is on the load result, two stall cycles will be needed, as the result from the load appears at the end of the MEM cycle but is needed at the beginning of ID for the branch.

Despite these difficulties, moving the branch execution to the ID stage is an improvement, because it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched. The exercises explore the details of implementing the forwarding path and detecting the hazard.

To flush instructions in the IF stage, we add a control line, called IF.Flush, that zeros the instruction field of the IF/ID pipeline register. Clearing the register transforms the fetched instruction into a `nop`, an instruction that has no action and changes no state.

Pipelined Branch **EXAMPLE**

Show what happens when the branch is taken in this instruction sequence, assuming the pipeline is optimized for branches that are not taken and that we moved the branch execution to the ID stage:

```
36 sub $10, $4, $8
40 beq $1, $3, 7 # PC-relative branch to 40+4+7*4=72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```

ANSWER

Figure 4.62 shows what happens when a branch is taken. Unlike Figure 4.61, there is only one pipeline bubble on a taken branch.

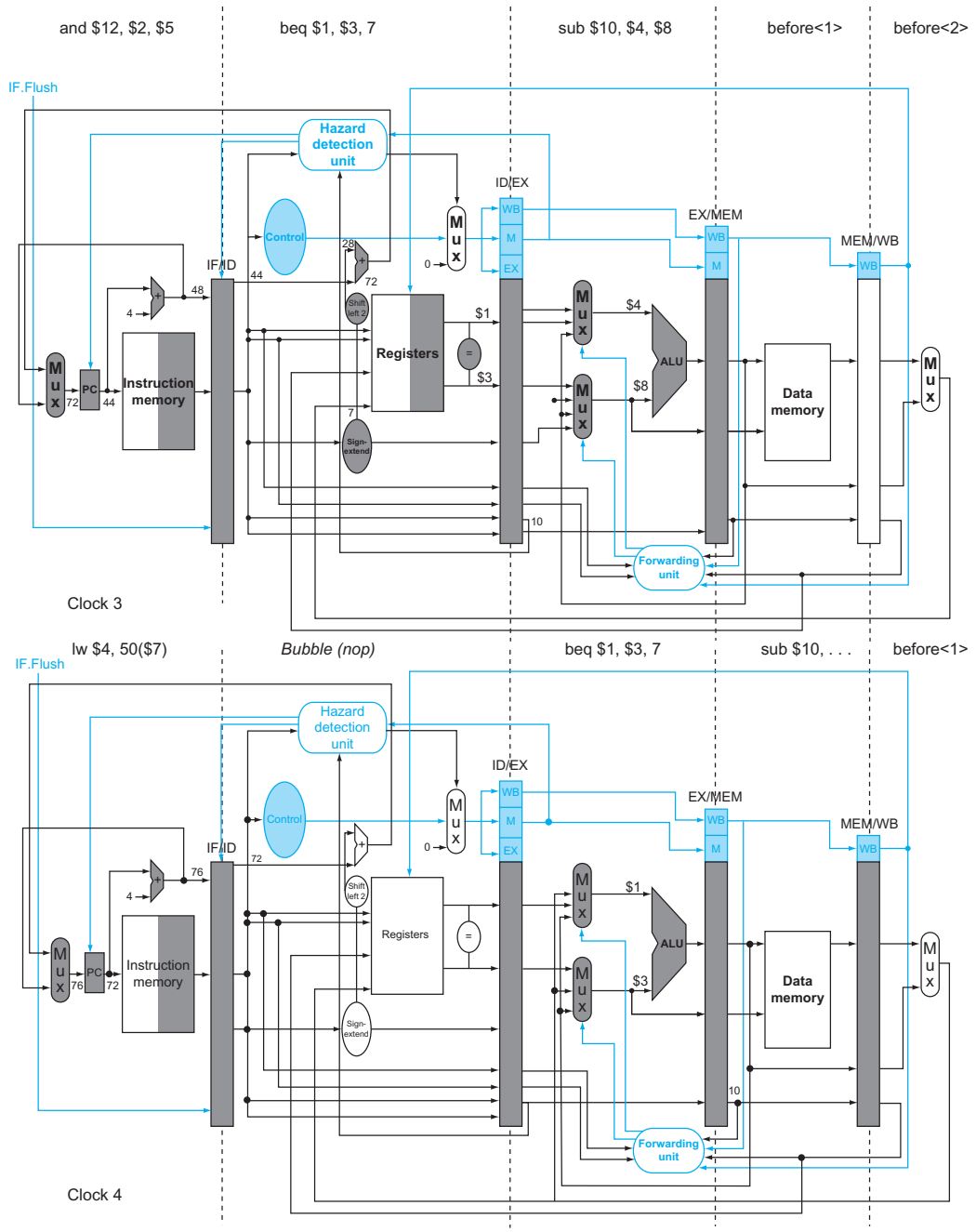
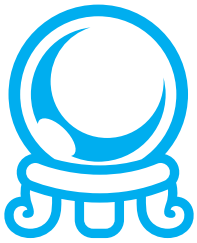


FIGURE 4.62 The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle. Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or `nop` instruction in the pipeline 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 (see Section 4.10), 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 4.5, with more hardware it is possible to try to **predict** branch behavior during program execution.



PREDICTION

dynamic branch prediction Prediction of branches at runtime using runtime 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.

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. However, this doesn't affect correctness. Prediction is just a hint that we hope is 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 can predict incorrectly twice, rather than once, when it is not taken. The following example shows this dilemma.

Loops and Prediction

EXAMPLE

Consider a loop branch that branches nine times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?

ANSWER

The steady-state prediction behavior will mispredict on the first and last loop iterations. Mispredicting the last iteration is inevitable since the prediction bit will indicate taken, as 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 4.63 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 318, it can be as early as the ID stage. Otherwise, sequential fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed as shown in Figure 4.63.

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

Elaboration: As we described in Section 4.5, 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 4.64 shows the three ways in which the branch delay slot can be scheduled.

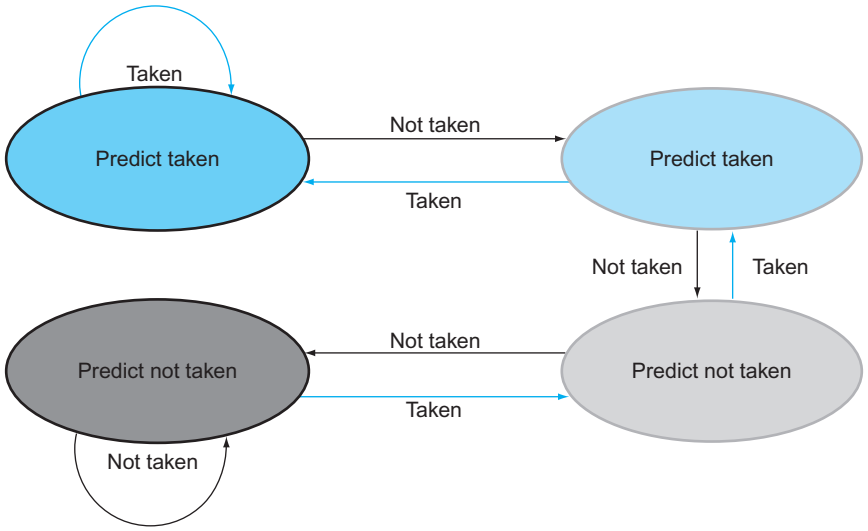


FIGURE 4.63 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The 2-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the mid-point of its range as the division between taken and not taken.

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 4.10), 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 due to **Moore's Law** made dynamic prediction relatively cheaper.

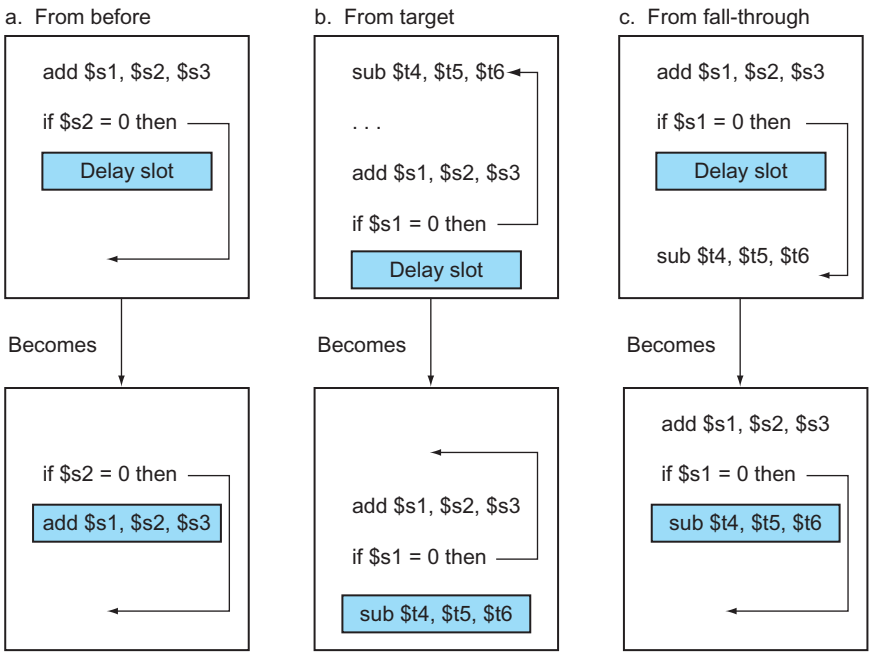


FIGURE 4.64 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) 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.

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 one cycle, meaning that taken branches will have a 1-cycle penalty. Delayed branches are one approach to eliminate that penalty. Another approach is to use a cache to hold the destination program counter or destination instruction using a **branch target buffer**.

branch target buffer A structure that caches the destination PC or destination instruction for a branch. It is usually organized as a cache with tags, making it more costly than a simple prediction buffer.

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 based on whether the last executed branch was taken or not taken. Thus, the global branch behavior can be thought of as adding additional index bits for the prediction lookup.

correlating predictor A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.

A more recent innovation in branch prediction is the use of tournament predictors. A **tournament predictor** uses multiple predictors, tracking, for each branch, which predictor yields the best results. A typical tournament predictor might contain two predictions for each branch index: one based on local information and one based on global branch behavior. A selector would choose which predictor to use for any given prediction. The selector can operate similarly to a 1- or 2-bit predictor, favoring whichever of the two predictors has been more accurate. Some recent microprocessors use such elaborate predictors.

tournament branch predictor A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

Elaboration: One way to reduce the number of conditional branches is to add *conditional move* instructions. Instead of changing the PC with a conditional branch, the instruction conditionally changes the destination register of the move. If the condition fails, the move acts as a `nop`. For example, one version of the MIPS instruction set architecture has two new instructions called `movn` (move if not zero) and `movz` (move if zero). Thus, `movn $8, $11, $4` copies the contents of register 11 into register 8, provided that the value in register 4 is nonzero; otherwise, it does nothing.

The ARMv7 instruction set has a condition field in most instructions. Hence, ARM programs could have fewer conditional branches than in MIPS programs.

Pipeline Summary

We 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 4.65 shows the final evolved datapath and control. We now are ready for yet another control hazard: the sticky issue of exceptions.

Check Yourself Consider three branch prediction schemes: predict not taken, predict taken, and dynamic prediction. Assume that they all have zero penalty when they predict correctly and two cycles when they are wrong. Assume that the average predict

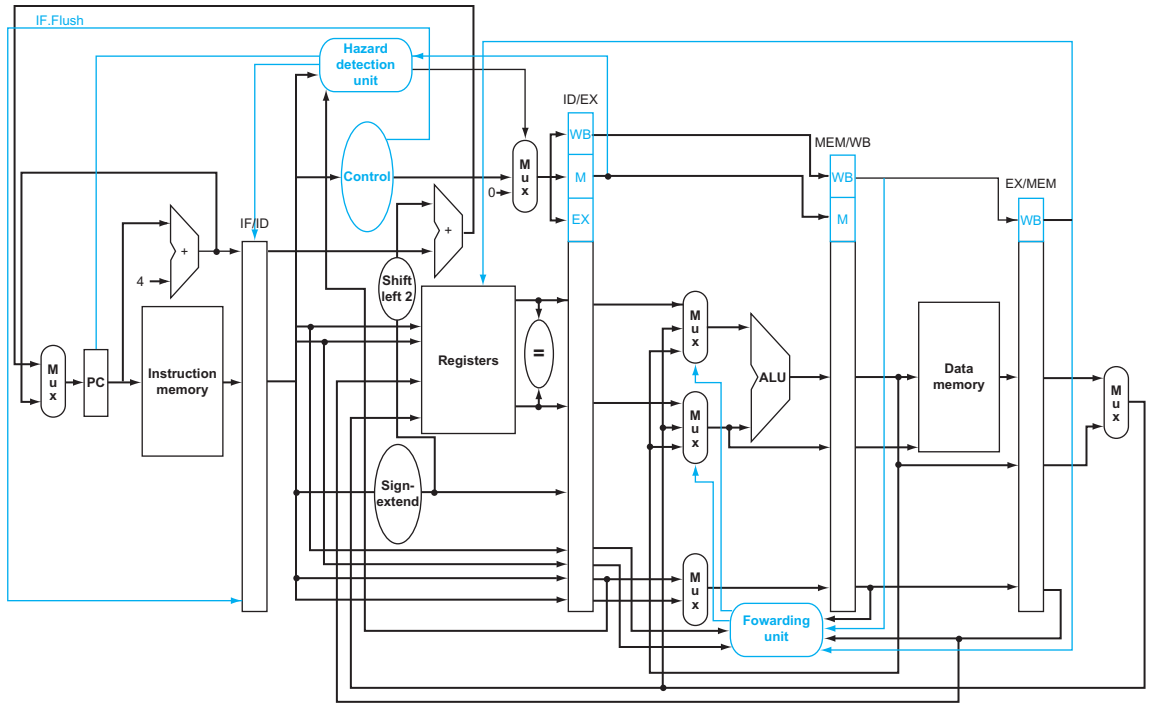


FIGURE 4.65 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.57 and the multiplexor controls from Figure 4.51.

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

4.9 Exceptions

To make a computer with automatic program-interruption facilities behave [sequentially] was not an easy matter, because the number of instructions in various stages of processing when an interrupt signal occurs may be large.

Fred Brooks, Jr., *Planning a Computer System: Project Stretch*, 1962

Control is the most challenging aspect of processor design: it is both the hardest part to get right and the hardest part to make fast. One of the hardest parts of

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

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

control is implementing **exceptions** and **interrupts**—events other than branches or jumps that change the normal flow of instruction execution. They were initially created to handle unexpected events from within the processor, like arithmetic overflow. The same basic mechanism was extended for I/O devices to communicate with the processor, as we will see in Chapter 5.

Many architectures and authors do not distinguish between interrupts and exceptions, often using the older name *interrupt* to refer to both types of events. For example, the Intel x86 uses interrupt. We follow the MIPS convention, using the term *exception* to refer to *any* unexpected change in control flow without distinguishing whether the cause is internal or external; we use the term *interrupt* only when the event is externally caused. Here are five examples showing whether the situation is internally generated by the processor or externally generated:

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

Many of the requirements to support exceptions come from the specific situation that causes an exception to occur. Accordingly, we will return to this topic in Chapter 5, when we will better understand the motivation for additional capabilities in the exception mechanism. In this section, we deal with the control implementation for detecting two types of exceptions that arise from the portions of the instruction set and implementation that we have already discussed.

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

How Exceptions Are Handled in the MIPS Architecture

The two types of exceptions that our current implementation can generate are execution of an undefined instruction and an arithmetic overflow. We'll use arithmetic overflow in the instruction `add $1, $2, $1` as the example exception in the next few pages. The basic action that the processor must perform when an exception occurs is to save the address of the offending instruction in the *exception program counter* (EPC) and then transfer control to the operating system at some specified address.

The operating system can then take the appropriate action, which may involve providing some service to the user program, taking some predefined action in

response to an overflow, or stopping the execution of the program and reporting an error. After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program. In Chapter 5, we will look more closely at the issue of restarting the execution.

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

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

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

Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

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

We can perform the processing required for exceptions by adding a few extra registers and control signals to our basic implementation and by slightly extending control. Let's assume that we are implementing the exception system used in the MIPS architecture, with the single entry point being the address $8000\ 0180_{hex}$. (Implementing vectored exceptions is no more difficult.) We will need to add two additional registers to our current MIPS implementation:

- *EPC:* A 32-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)
- *Cause:* A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused. Assume there is a five-bit field that encodes the two possible exception sources mentioned above, with 10 representing an undefined instruction and 12 representing arithmetic overflow.

Exceptions in a Pipelined Implementation

A pipelined implementation treats exceptions as another form of control hazard. For example, suppose there is an arithmetic overflow in an add instruction. Just as

we did for the taken branch in the previous section, we must flush the instructions that follow the `add` instruction from the pipeline and begin fetching instructions from the new address. We will use the same mechanism we used for taken branches, but this time the exception causes the deasserting of control lines.

When we dealt with branch mispredict, we saw how to flush the instruction in the IF stage by turning it into a `nop`. To flush instructions in the ID stage, we use the multiplexor already in the ID stage that zeros control signals for stalls. A new control signal, called ID.Flush, is ORed with the stall signal from the hazard detection unit to flush during ID. To flush the instruction in the EX phase, we use a new signal called EX.Flush to cause new multiplexors to zero the control lines. To start fetching instructions from location 8000 0180_{hex}, which is the MIPS exception address, we simply add an additional input to the PC multiplexor that sends 8000 0180_{hex} to the PC. Figure 4.66 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 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). In reality, we save the address +4, so the exception handling the software routine must first subtract 4 from the saved value. Figure 4.66 shows a stylized version of the datapath, including the branch hardware and necessary accommodations to handle exceptions.

EXAMPLE

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)
...
```

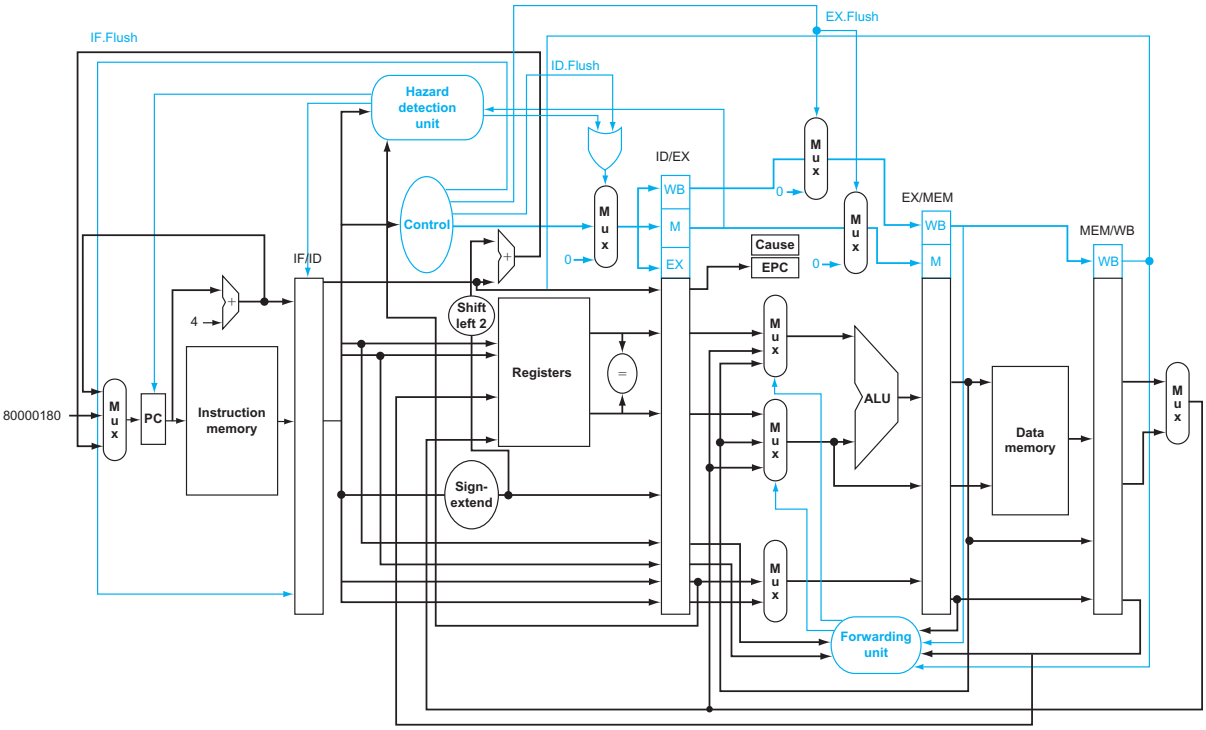


FIGURE 4.66 The datapath with controls to handle exceptions. The key additions include a new input with the value 8000 0180_{hex} in the multiplexor that supplies the new PC value; a Cause register to record the cause of the exception; and an Exception PC register to save the address of the instruction that caused the exception. The 8000 0180_{hex} input to the multiplexor is the initial address to begin fetching instructions in the event of an exception. Although not shown, the ALU overflow signal is an input to the control unit.

assume the instructions to be invoked on an exception begin like this:

```
80000180hex   sw     $26, 1000($0)
80000184hex   sw     $27, 1004($0)
...
```

Show what happens in the pipeline if an overflow exception occurs in the `add` instruction.

ANSWER

Figure 4.67 shows the events, starting with the add instruction in the EX stage. The overflow is detected during that phase, and 8000 0180_{hex} is forced into the PC. Clock cycle 7 shows that the `add` and following instructions are flushed, and the first instruction of the exception code is fetched. Note that the address of the instruction *following* the `add` is saved: $4C_{hex} + 4 = 50_{hex}$.

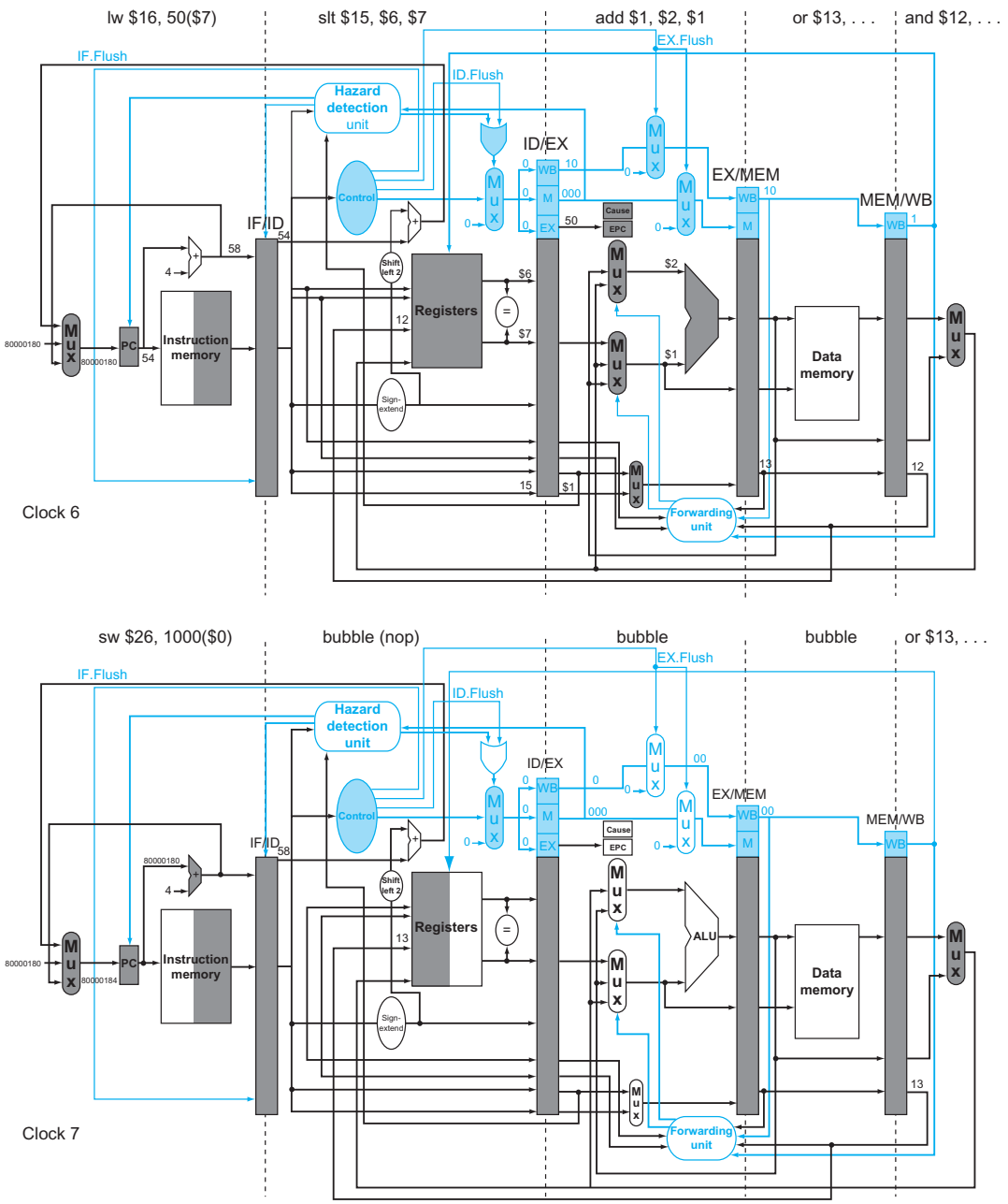


FIGURE 4.67 The result of an exception due to arithmetic overflow in the add instruction. The overflow is detected during the EX stage of clock 6, saving the address following the add in the EPC register ($4C + 4 = 50_{hex}$). Overflow causes all the Flush signals to be set near the end of this clock cycle, deasserting control values (setting them to 0) for the `add`. Clock cycle 7 shows the instructions converted to bubbles in the pipeline plus the fetching of the first instruction of the exception routine—`sw $25,1000($0)`—from instruction location $8000\ 0180_{hex}$. 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.

We mentioned five examples of exceptions on page 326, and we will see others in Chapter 5. 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 solution is to prioritize the exceptions so that it is easy to determine which is serviced first. 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, 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 in a pending exception field so that the hardware can interrupt based on later exceptions, once the earliest one has been serviced.

Hardware/Software Interface

The hardware and the operating system must work in conjunction so that exceptions behave as you would expect. The hardware contract is normally to stop the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address. The operating system contract is to look at the cause of the exception and act appropriately. For an undefined instruction, hardware failure, or arithmetic overflow exception, the operating system normally kills the program and returns an indicator of the reason. For an I/O device request or an operating system service call, the operating system saves the state of the program, performs the desired task, and, at some point in the future, restores the program to continue execution. In the case of I/O device requests, we may often choose to run another task before resuming the task that requested the I/O, since that task may often not be able to proceed until the I/O is complete. Exceptions are 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 5 describes these exceptions and their handling in more detail.

imprecise interrupt Also called **imprecise exception**. Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.

Elaboration: The difficulty of always associating the correct exception with the correct instruction in pipelined computers has led some computer designers to relax this requirement in noncritical cases. Such processors are said to have **imprecise interrupts** or **imprecise exceptions**. In the example above, PC would normally have 58_{hex} at the start of the clock cycle after the exception is detected, even though the offending instruction

precise interrupt Also called **precise exception**. An interrupt or exception that is always associated with the correct instruction in pipelined computers.

is at address $4C_{hex}$. A processor with imprecise exceptions might put 58_{hex} into EPC and leave it up to the operating system to determine which instruction caused the problem. MIPS and the vast majority of computers today support **precise interrupts** or **precise exceptions**. (One reason is to support virtual memory, which we shall see in Chapter 5.)

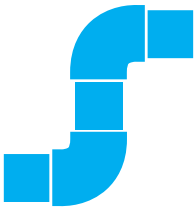
Elaboration: Although MIPS uses the exception entry address $8000\ 0180_{hex}$ for almost all exceptions, it uses the address $8000\ 0000_{hex}$ to improve performance of the exception handler for TLB-miss exceptions (see Chapter 5).

Check Yourself

Which exception should be recognized first in this sequence?

1. `add $1, $2, $1` # arithmetic overflow
2. `XXX $1, $2, $1` # undefined instruction
3. `sub $1, $2, $1` # hardware error

4.10 Parallelism via Instructions



instruction-level parallelism The parallelism among instructions.

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

Be forewarned: this section is a brief overview of fascinating but advanced topics. If you want to learn more details, you should consult our more advanced book, *Computer Architecture: A Quantitative Approach*, fifth edition, where the material covered in these 13 pages is expanded to almost 200 pages (including appendices)!

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 was longer than the others were, 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 speed-up, 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.