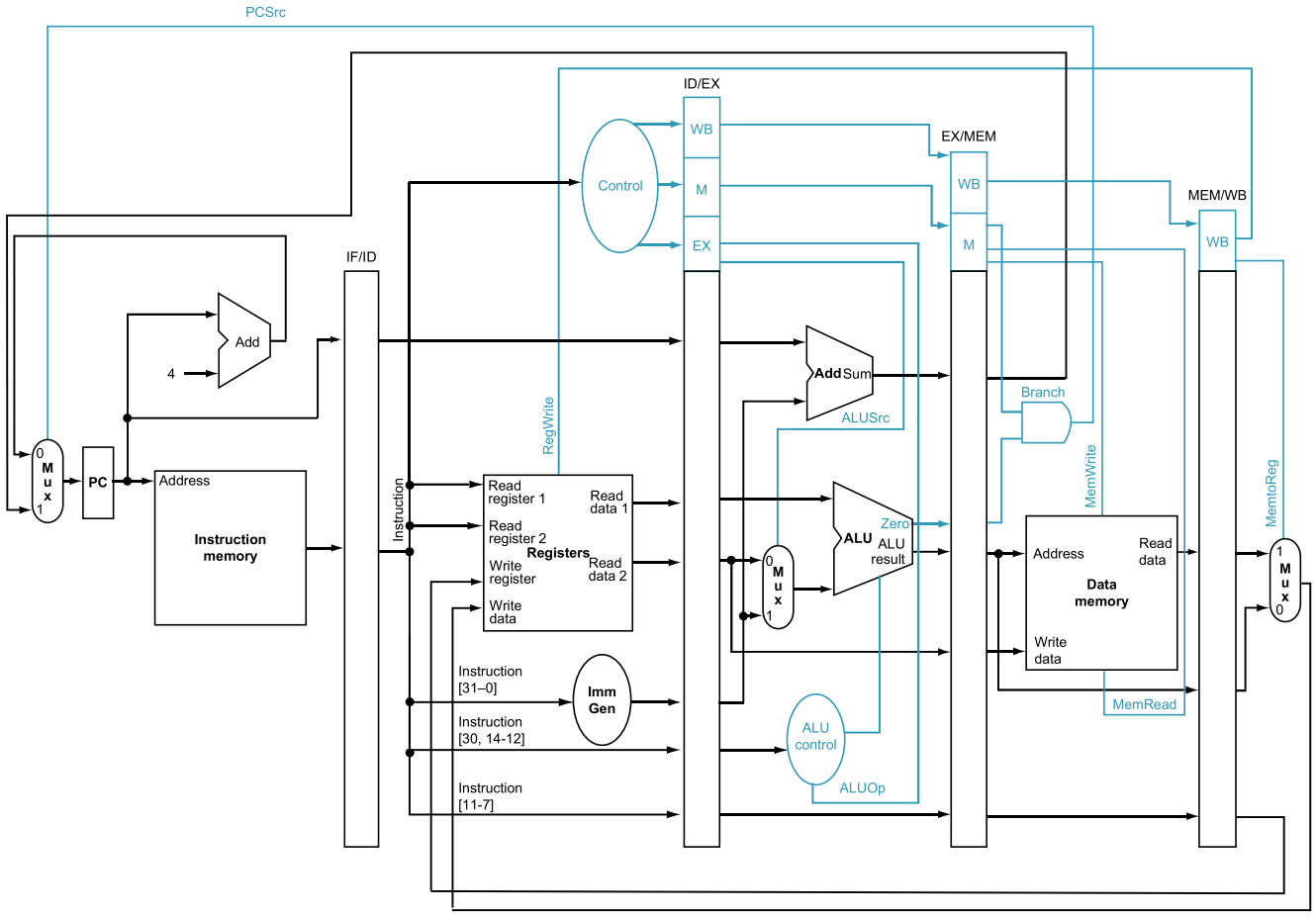


FIGURE 4.53 The pipelined datapath of Figure 4.48, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

How would this sequence perform with our pipeline? Figure 4.54 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 4.54 shows the value of register `x2`, which changes during the middle of clock cycle 5, when the `sub` instruction writes its result.

The potential of `add` 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 4.67 shows that the values read for register `x2` would *not* be the result of the `sub` instruction unless the read occurred during clock cycle 5 or later. Thus, the instructions that would get the correct value of −20 are `add` and `sw`; the `and` and `or` instructions would get the incorrect value 10! Using this style of drawing, such problems become apparent when a dependence line goes backward in time.

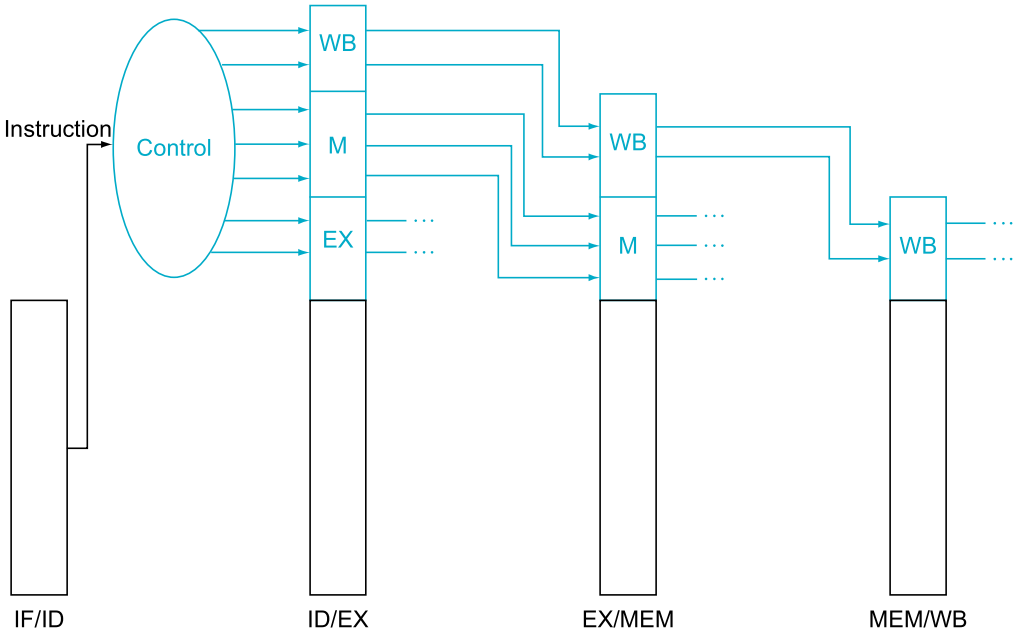


FIGURE 4.52 The seven control lines for the final three stages. Note that two of the seven control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

4.8 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 RISC-V instructions in Figures 4.45 through 4.47 were independent; none of them used the results calculated by any of the others. Yet, in Section 4.6, we saw that data hazards are obstacles to pipelined execution.

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

```
sub   x2, x1, x3    // Register z2 written by sub
and   x12, x2, x5   // 1st operand(x2) depends on sub
or    x13, x6, x2   // 2nd operand(x2) depends on sub
add   x14, x2, x2   // 1st(x2) & 2nd(x2) depend on sub
sw    x15, 100(x2)  // Base (x2) depends on sub
```

The last four instructions are all dependent on the result in register `x2` of the first instruction. If register `x2` 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 `x2`.

Instruction	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

FIGURE 4.49 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 opcodes for the R-type instruction.

Signal name	Effect when deasserted	Effect when asserted
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, 12 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 4.50 A copy of Figure 4.20. The function of each of six control signals is defined. The ALU control lines (ALUOp) are defined in the second column of Figure 4.49. When a 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Note that PCSrc is controlled by an AND gate in Figure 4.48. If the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a `beq` instruction; otherwise, PCSrc is set to 0.

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

FIGURE 4.51 The values of the control lines are the same as in Figure 4.22, but they have been shuffled into three groups corresponding to the last three pipeline stages.

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:* The two source registers are always in the same location in the RISC-V instruction formats, so there is nothing special to control in this pipeline stage.
3. *Execution/address calculation:* The signals to be set are ALUOp and ALUSrc (see Figures 4.49 and 4.50). The signals select the ALU operation and either Read data 2 or a sign-extended immediate as inputs to the ALU.
4. *Memory access:* The control lines set in this stage are Branch, MemRead, and MemWrite. The branch if equal, load, and store instructions set these signals, respectively. Recall that PCSrc in Figure 4.50 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 4.51 has the same values as in Section 4.4, but now the seven control lines are grouped by pipeline stage.

Implementing control means setting the seven control lines to these values in each stage for each instruction.

Since the rest of the control lines starts with the EX stage, we can create the control information during instruction decode for the later stages. The simplest way to pass these control signals is to extend the pipeline registers to include control information. Figure 4.52 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 4.43. Figure 4.53 shows the full datapath with the extended pipeline registers and with the control lines connected to the proper stage. (**Section 4.14** gives more examples of RISC-V code executing on pipelined hardware using single-clock diagrams, if you would like to see more details.)

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

Pipelined Control

Just as we added control to the single-cycle datapath in Section 4.4, 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 4.61 shows those lines. We borrow as much as we can from the control for the simple datapath in Figure 4.21. In particular, we use the same ALU control logic, branch logic, and control lines. These functions are defined in Figures 4.12, 4.20, and 4.22. We reproduce the key information in Figures 4.49 through 4.51 on in two pages in this section to make the following discussion easier to absorb.

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.

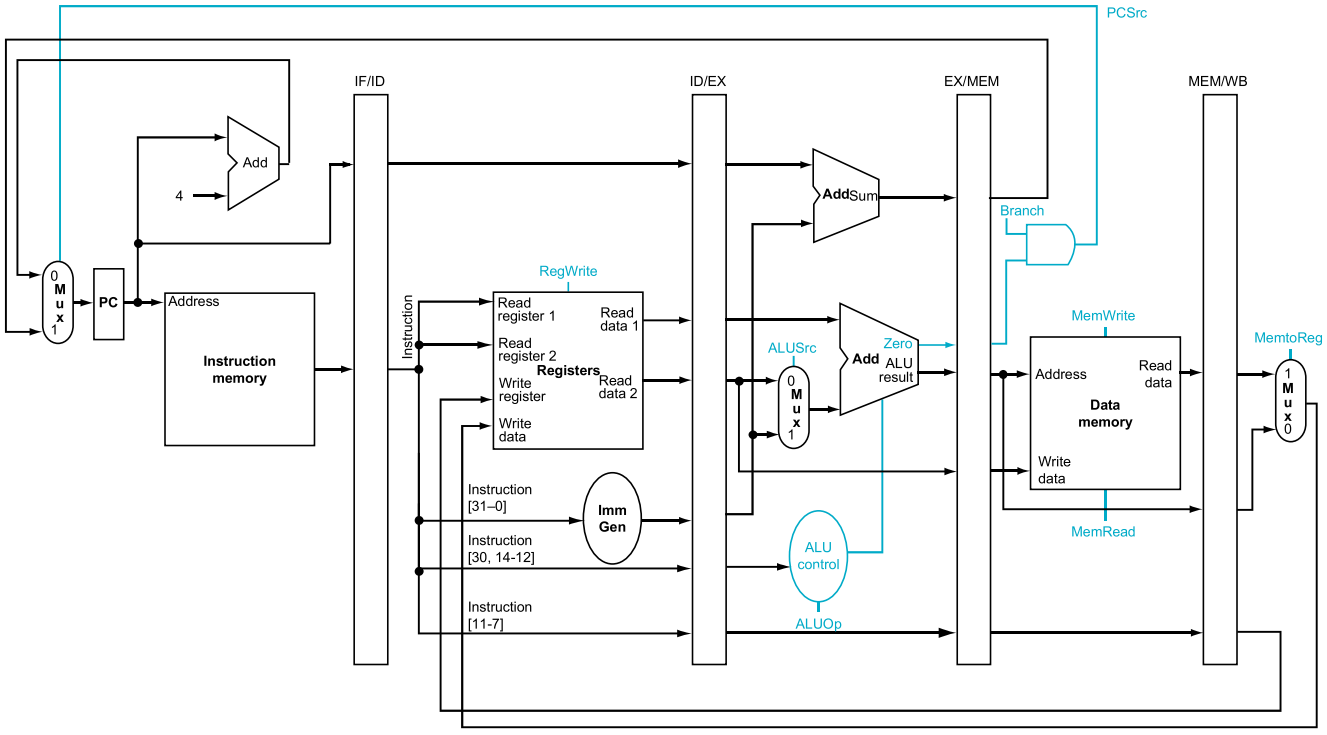


FIGURE 4.48 The pipelined datapath of Figure 4.43 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Section 4.4. Note that we now need funct fields of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register.

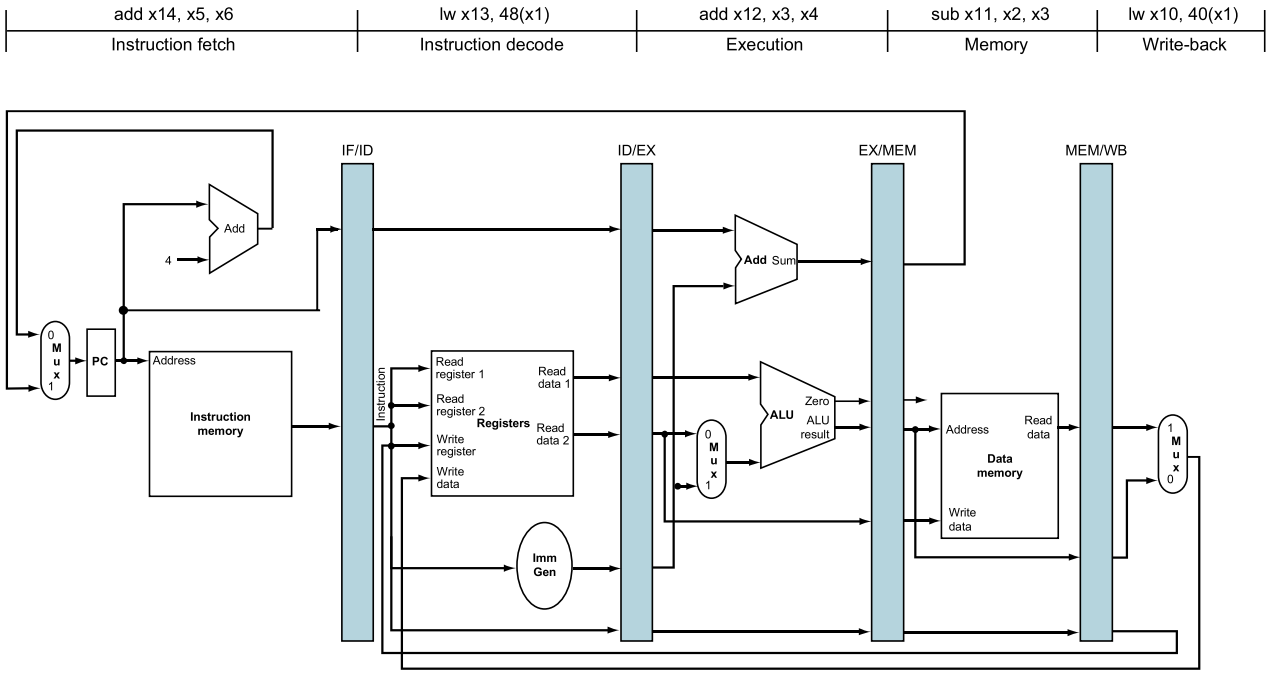


FIGURE 4.47 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.45 and 4.46. As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

single-clock-cycle diagram corresponding to clock cycle 5 of Figures 4.45 and 4.46. 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?

1. Allowing 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 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.

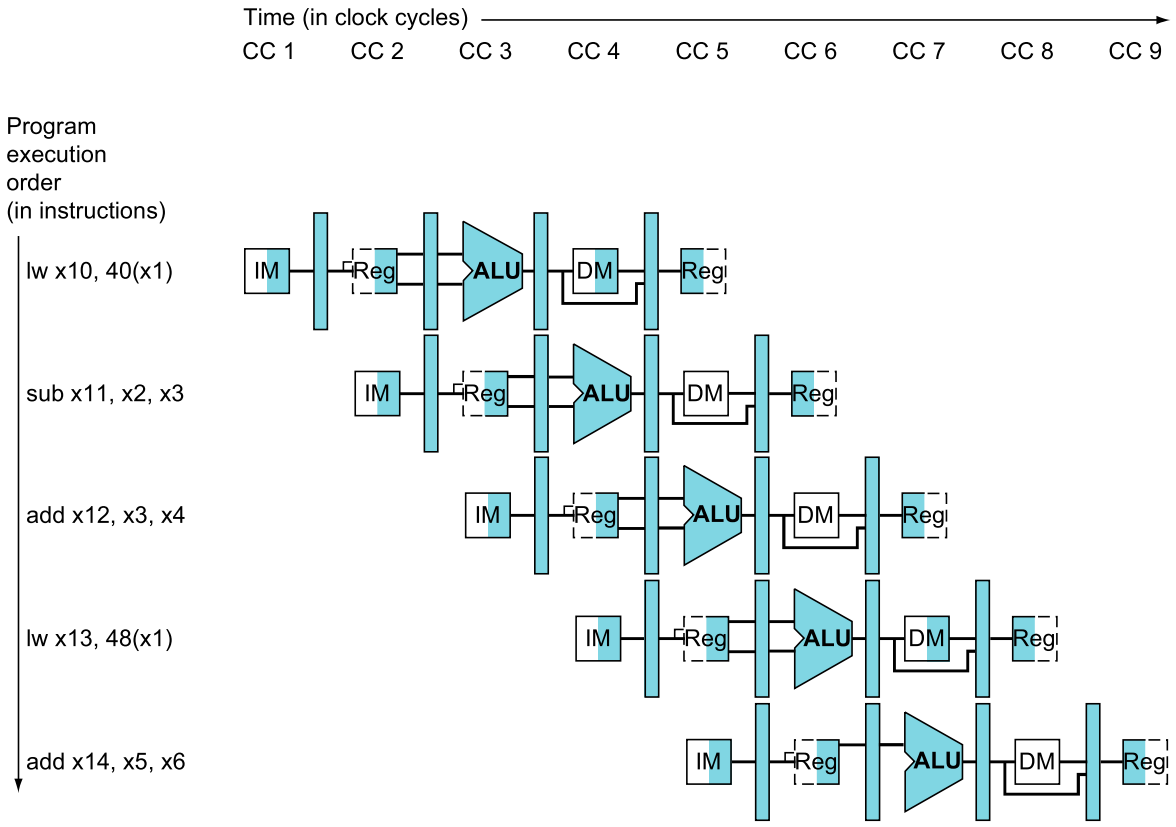


FIGURE 4.45 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 4.26, here we show the pipeline registers between each stage. Figure 4.59 shows the traditional way to draw this diagram.

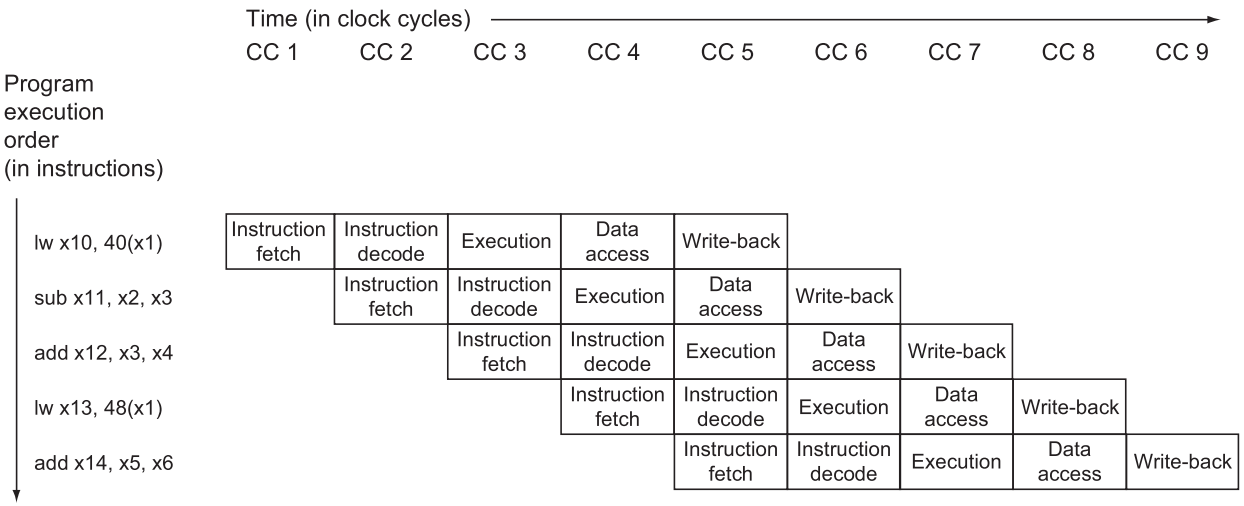


FIGURE 4.46 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.45.

```
lw      x10, 40(x1)
sub     x11, x2, x3
add     x12, x3, x4
lw      x13, 48(x1)
add     x14, x5, x6
```

Figure 4.45 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 4.27. 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 4.46 shows the more traditional version of the multiple-clock-cycle pipeline diagram. Note that Figure 4.45 shows the physical resources used at each stage, while Figure 4.46 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, 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 4.14** gives more illustrations of single-clock diagrams if you would like to see more details about Figure 4.45.) A single-clock-cycle diagram represents a vertical slice of one clock cycle 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 4.47 shows the

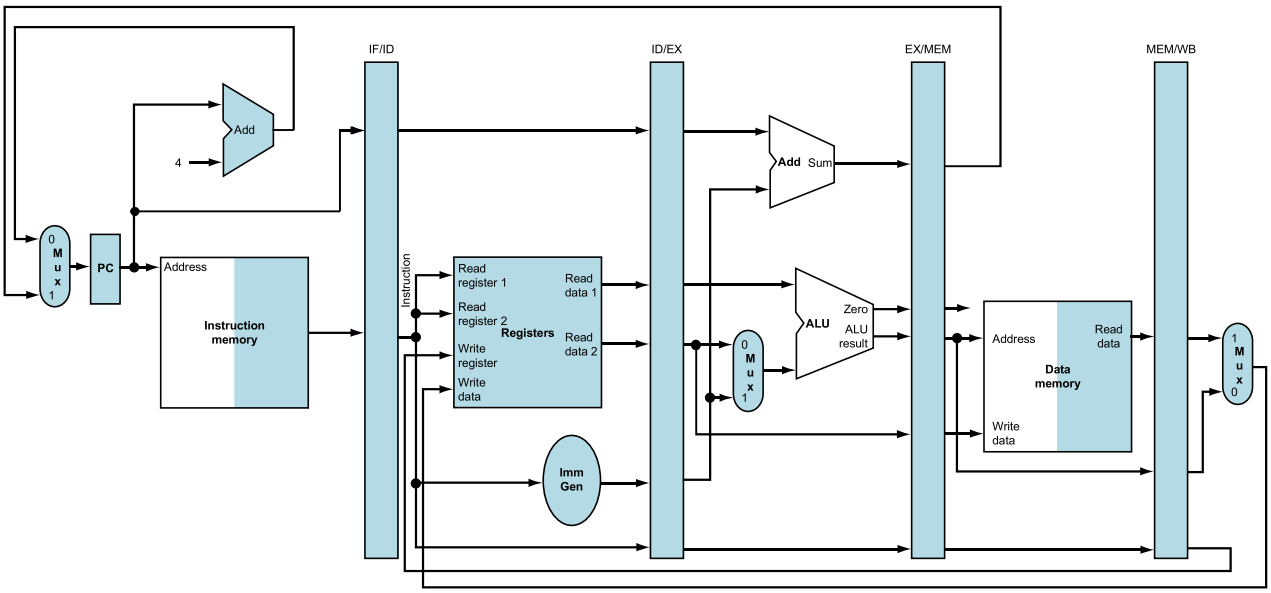


FIGURE 4.44 The portion of the datapath in Figure 4.43 that is used in all five stages of a load instruction.

Hence, we need to preserve the destination register number in the load instruction. Just as store passed the register *value* 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 4.43 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 4.44 is a single drawing of the corrected datapath, highlighting the hardware used in all five stages of the load register instruction in Figures 4.38 through 4.40. See Section 4.9 for an explanation of how to make the branch instruction work as expected.

Graphically Representing Pipelines

Pipelining can be difficult to master, since many instructions are simultaneously executing in a single datapath in every clock cycle. To aid understanding, there are two basic styles of pipeline figures: *multiple-clock-cycle pipeline diagrams*, such as Figure 4.36 on page 298, and *single-clock-cycle pipeline diagrams*, such as Figures 4.38 through 4.42. The multiple-clock-cycle diagrams are simpler but do not contain all the details. For example, consider the following five-instruction sequence:

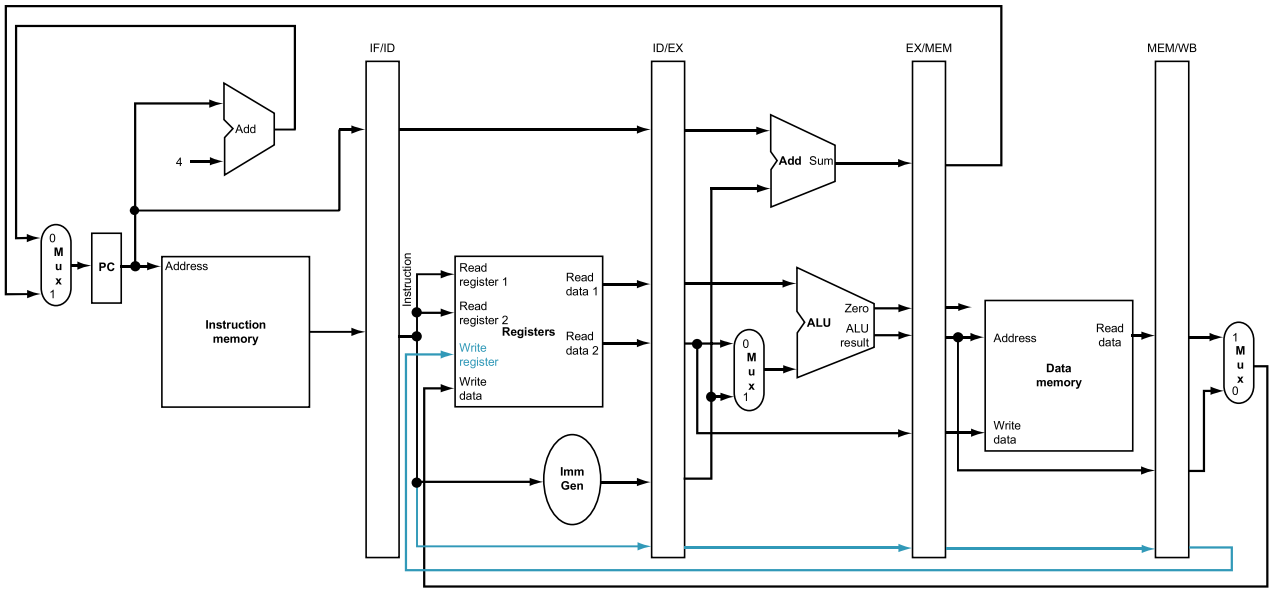


FIGURE 4.43 The corrected pipelined datapath to handle the load instruction properly. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits to the last three pipeline registers. This new path is shown in color.

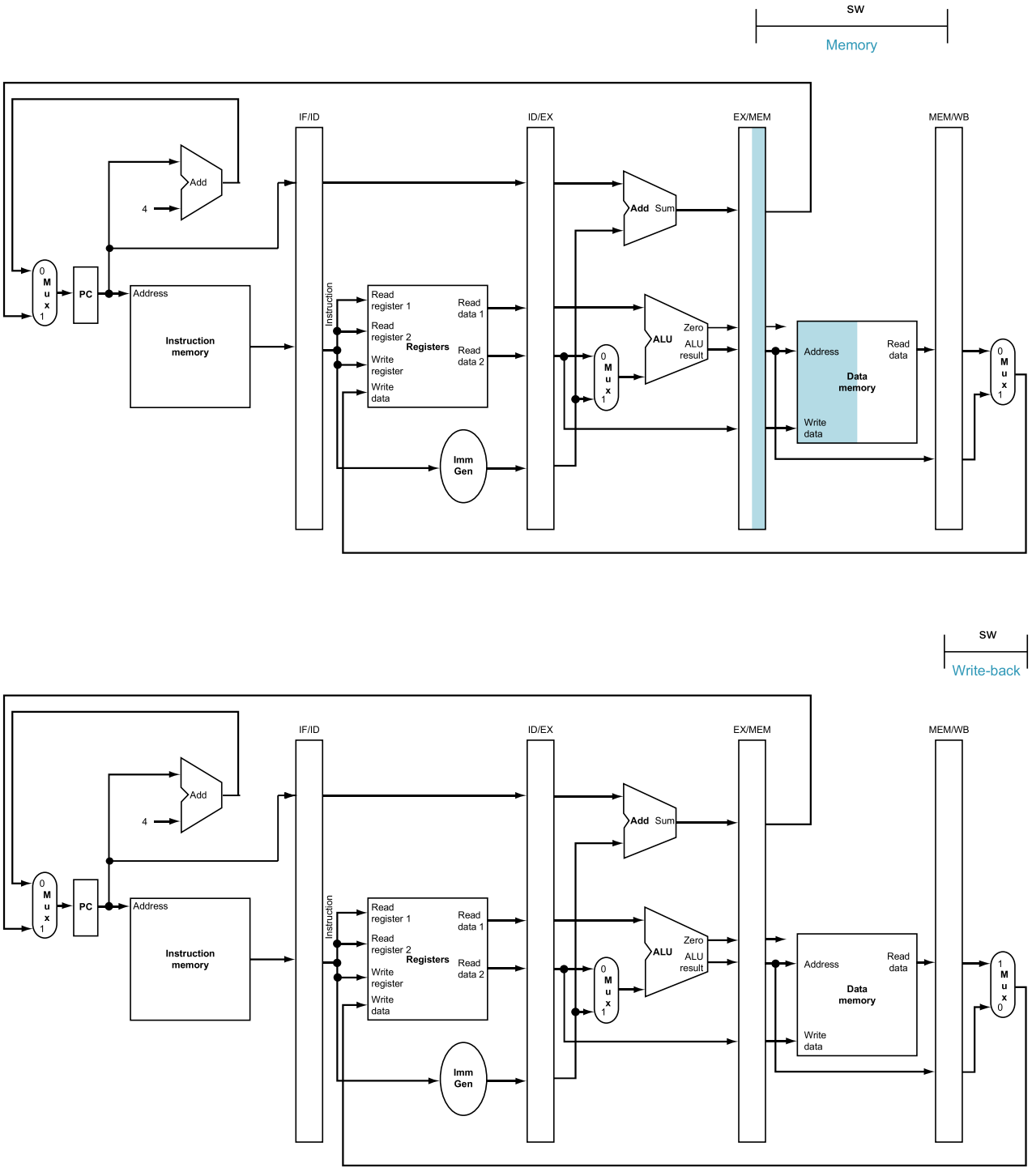


FIGURE 4.42 MEM and WB: The fourth and fifth pipe stages of a store instruction. In the fourth stage, the data are written into data memory for the store. Note that the data come from the EX/MEM pipeline register and that nothing is changed in the MEM/WB pipeline register. Once the data are written in memory, there is nothing left for the store instruction to do, so nothing happens in stage 5.

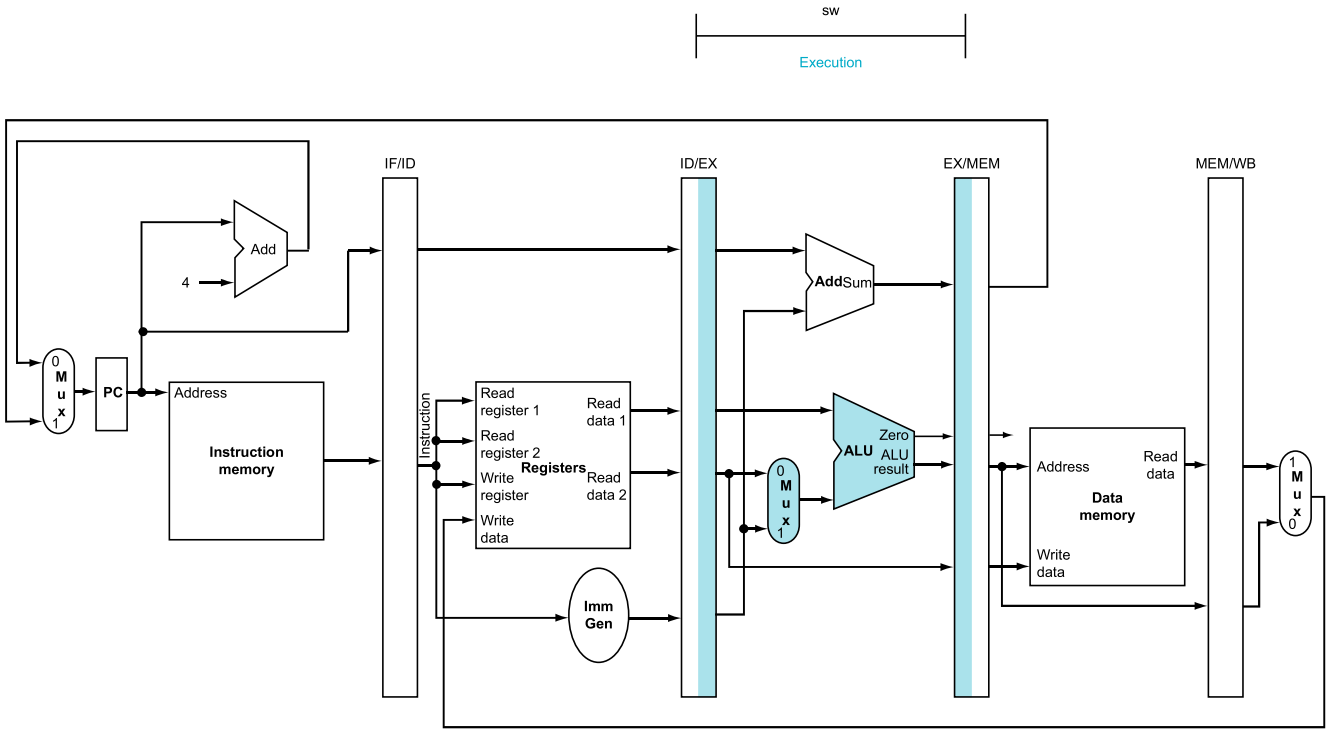


FIGURE 4.41 EX: The third pipe stage of a store instruction. Unlike the third stage of the load instruction in Figure 4.39, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.

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 were 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 287). Hence, these components, and their control, can be associated with a single pipeline stage.

Now we can uncover a bug in the design of the load instruction. Did you see it? Which register is changed in the final stage of the load? More specifically, which instruction supplies the write register number? The instruction in the IF/ID pipeline register supplies the write register number, yet this instruction occurs considerably *after* the load instruction!

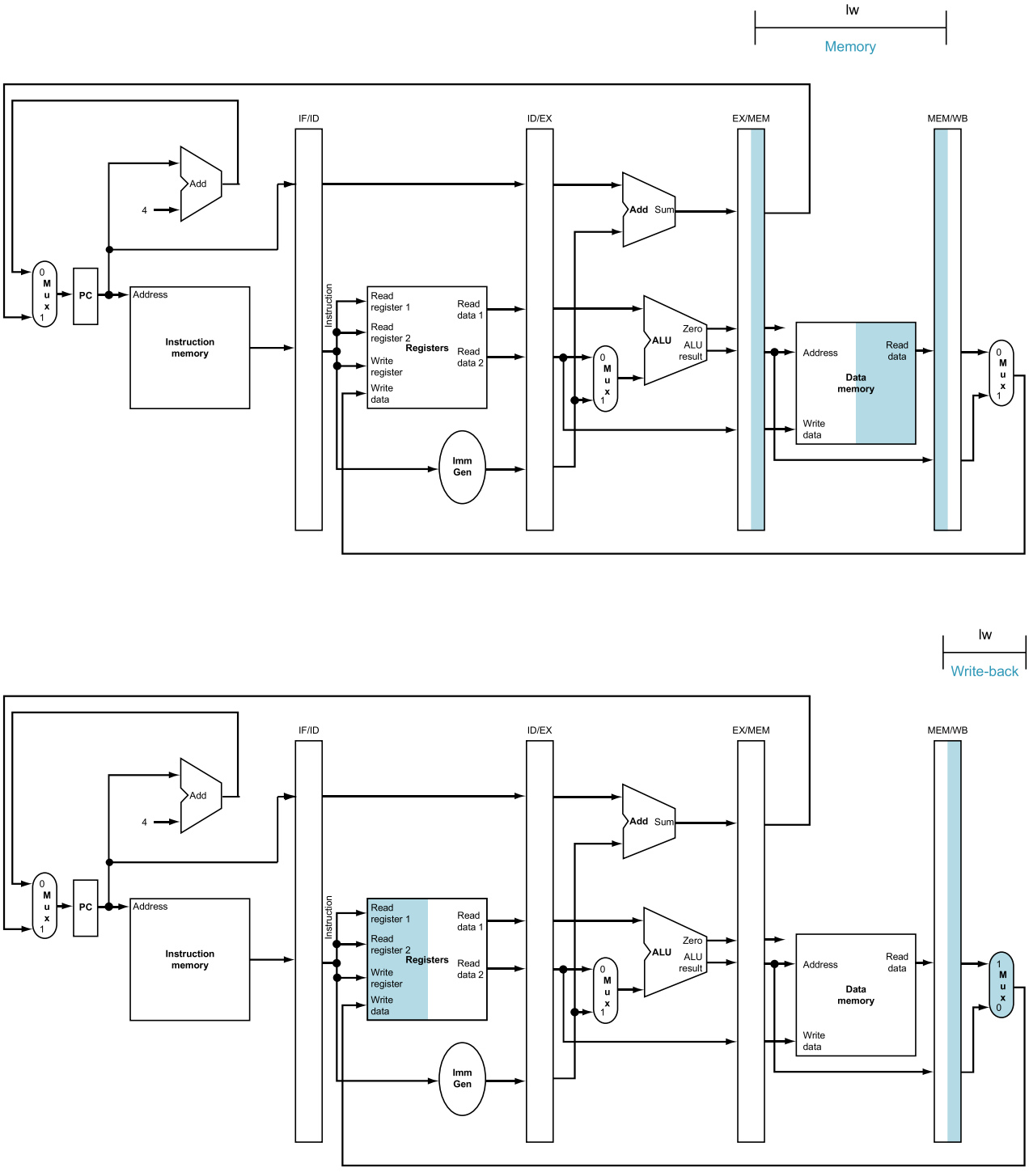


FIGURE 4.40 MEM and WB: The fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 4.37 used in this pipe stage. Data memory is read using the address in the EX/MEM pipeline registers, and the data are placed in the MEM/WB pipeline register. Next, data are 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 4.43.

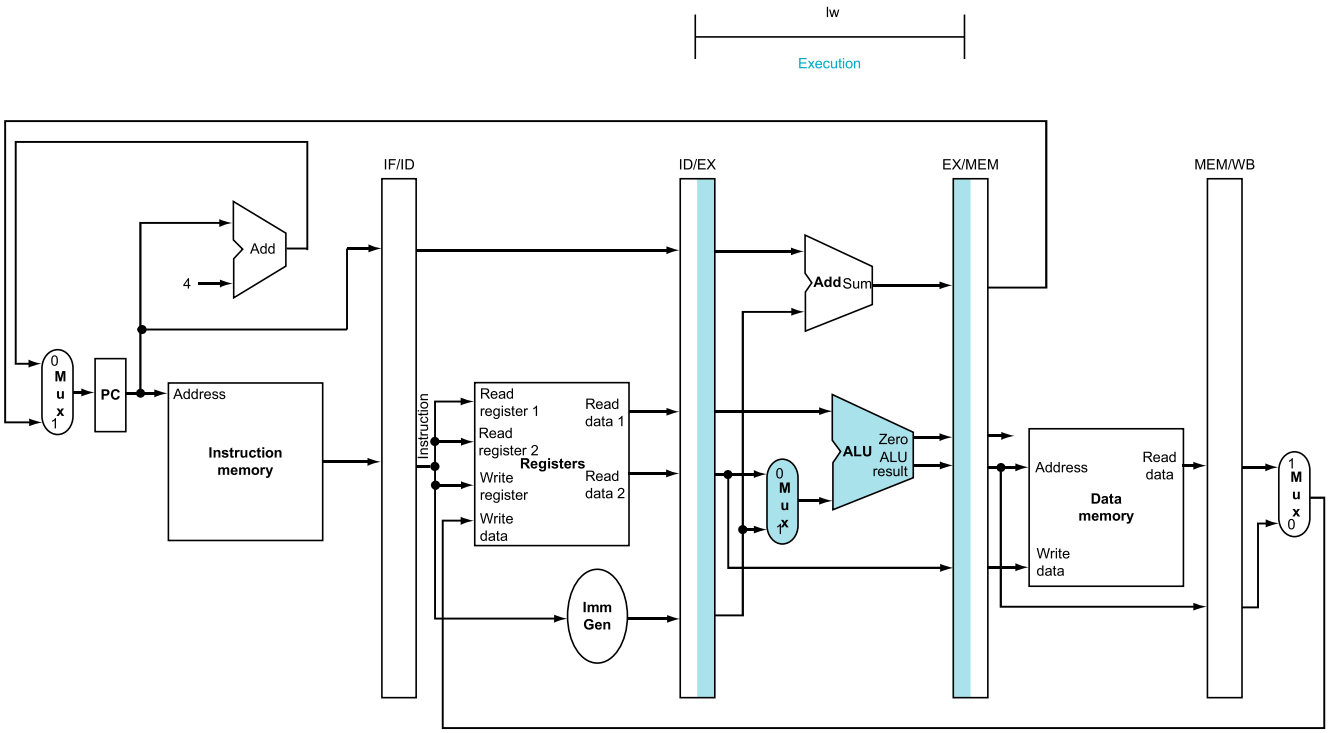


FIGURE 4.39 EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 4.37 used in this pipe stage. The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.

2. *Instruction decode and register file read:* The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the immediate operand. These three 64-bit values are all stored in the ID/EX pipeline register. The bottom portion of Figure 4.38 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. (While the store instruction uses the rs2 field to read the second register in this pipe stage, that detail is not shown in this pipeline diagram, so we can use the same figure for both.)
3. *Execute and address calculation:* Figure 4.41 shows the third step; the effective address is placed in the EX/MEM pipeline register.
4. *Memory access:* The top portion of Figure 4.42 shows the data being written to memory. Note that the register containing the data to be stored was read in an earlier stage and stored in ID/EX. The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.
5. *Write-back:* The bottom portion of Figure 4.42 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

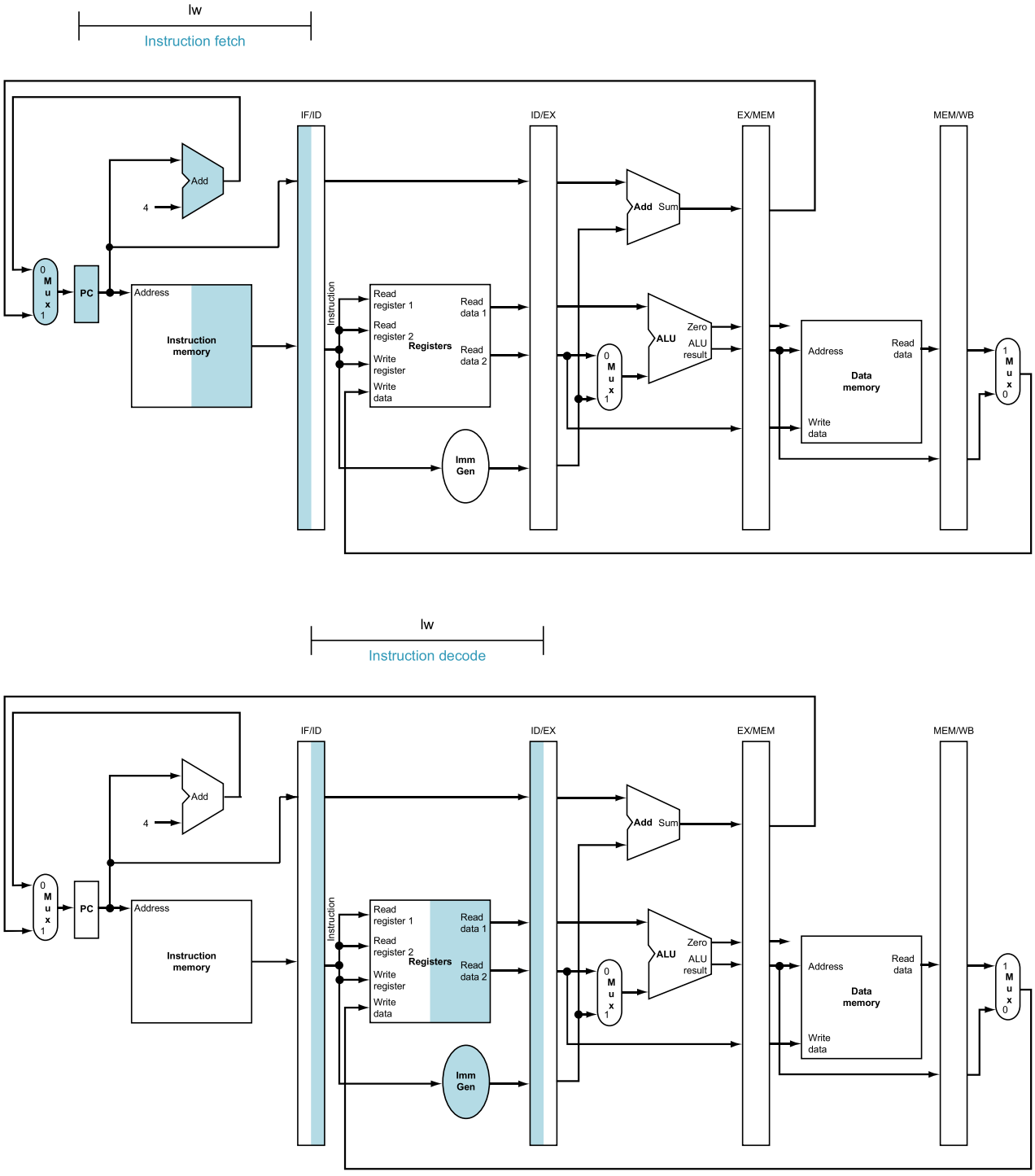


FIGURE 4.38 IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in Figure 4.37 highlighted. The highlighting convention is the same as that used in Figure 4.30. As in Section 4.2, there is no confusion when reading and writing registers, because the contents change only on the clock edge. Although the load needs only the top register in stage 2, it doesn't hurt to do potentially extra work, so it sign-extends the 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.

Figures 4.38 through 4.41, 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 4.30 through 4.32, 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 `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 4.38 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 PC 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 4.38 shows the instruction portion of the IF/ID pipeline register supplying the immediate field, which is sign-extended to 64 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.

3. *Execute or address calculation:* Figure 4.39 shows that the load instruction reads the contents of a register 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 4.40 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 4.40 shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

This walk-through of the load instruction shows that any information needed in a later pipe stage must be passed to that stage via a pipeline register. Walking through a store instruction shows the similarity of instruction execution, as well as passing the information for later stages. Here are the five pipe stages of the store instruction:

1. *Instruction fetch:* The instruction is read from memory using the address in the PC and then is placed in the IF/ID pipeline register. This stage occurs before the instruction is identified, so the top portion of Figure 4.38 works for store as well as load.

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 one 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 4.37, however, the PC is part of the visible architectural state; its contents must be saved when an exception occurs, while the contents of the pipeline registers can be discarded. In the laundry analogy, you could think of the PC as corresponding to the basket that holds the load of dirty clothes before the wash step.

To show how the pipelining works, throughout this chapter we show sequences of figures to demonstrate operation over time. These extra pages would seem to require much more time for you to understand. Fear not; the sequences take much less time than it might appear, because you can compare them to see what changes occur in each clock cycle. Section 4.8 describes what happens when there are data hazards between pipelined instructions; ignore them for now.

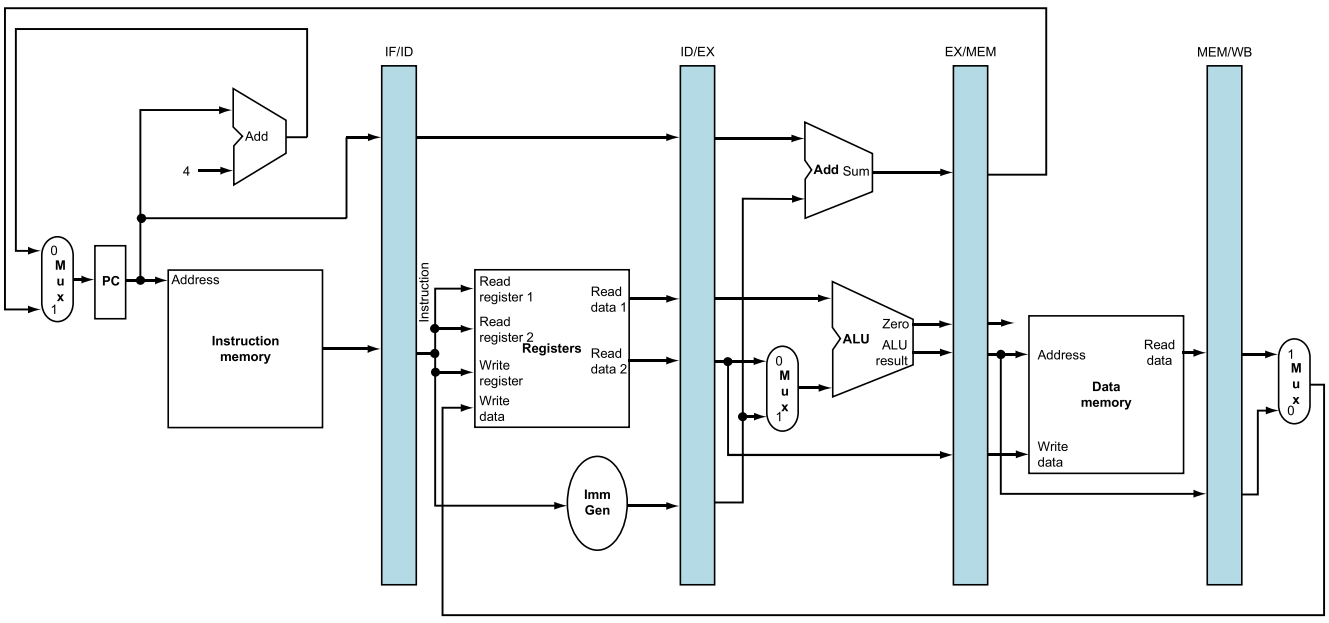


FIGURE 4.37 The pipelined version of the datapath in Figure 4.35. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The 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 96 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 64-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 256, 193, and 128 bits, respectively.

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 4.36 shows the execution of the instructions in Figure 4.29 by displaying their private datapaths on a common timeline. We use a stylized version of the datapath in Figure 4.35 to show the relationships in Figure 4.36.

Figure 4.36 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 4.36 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 4.35. Returning to our laundry analogy, we might have a basket between each pair of stages to hold the clothes for the next step.

Figure 4.37 shows the pipelined datapath with the pipeline registers highlighted. All instructions advance during each clock cycle from one pipeline register

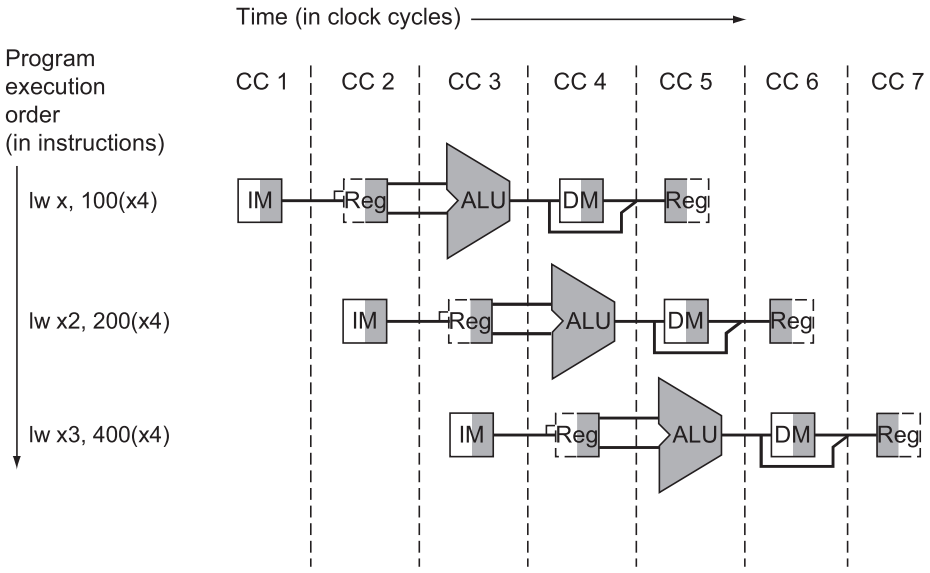


FIGURE 4.36 Instructions being executed using the single-cycle datapath in Figure 4.35, assuming pipelined execution. Similar to Figures 4.30 through 4.32, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 4.48. *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.

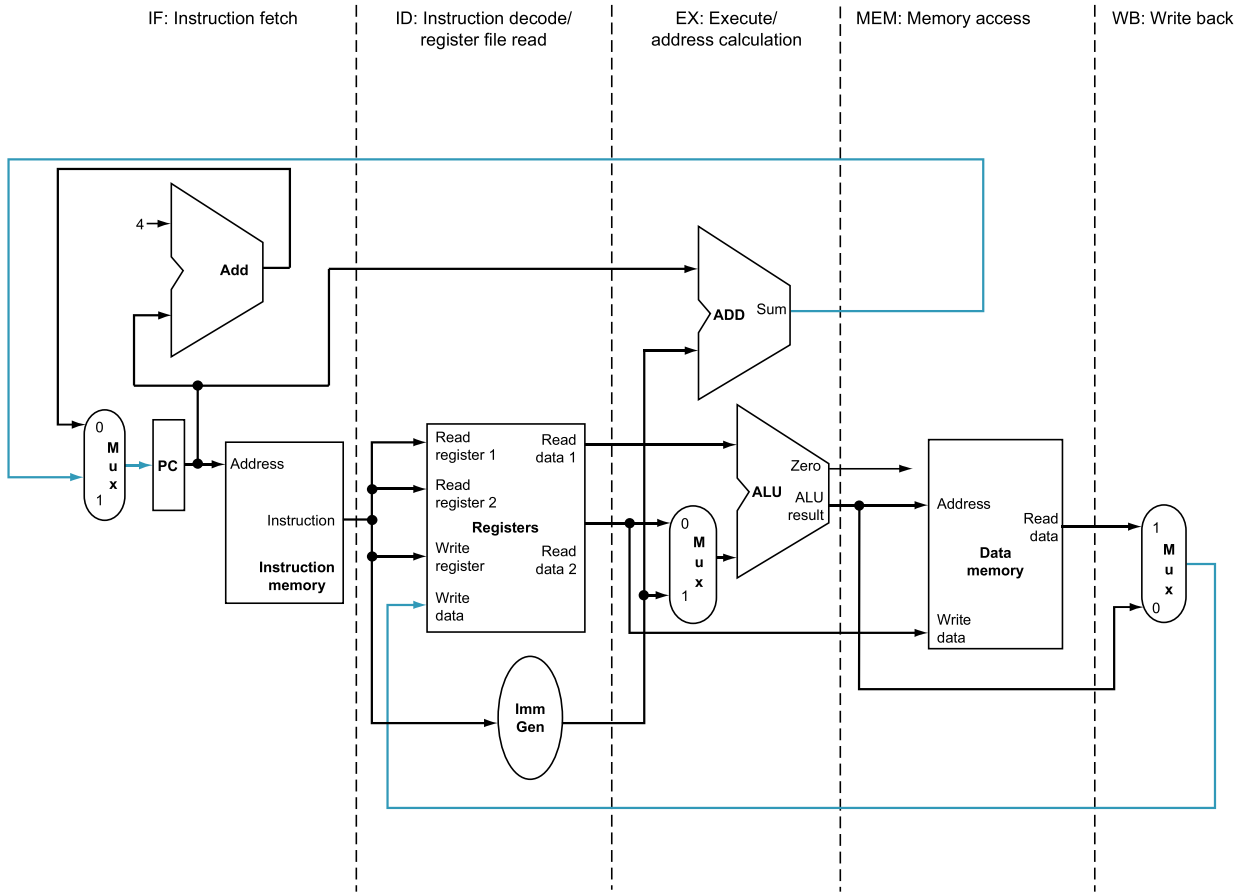


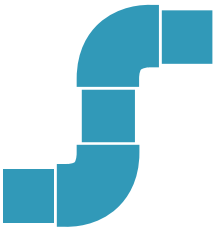
FIGURE 4.35 The single-cycle datapath from Section 4.4 (similar to Figure 4.21). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory 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 do 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.



PIPELINING

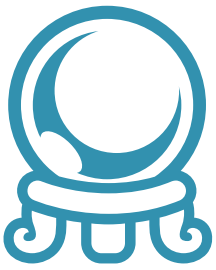
performance bottlenecks in both integer and floating-point programs. Often it is easier to deal with data hazards in floating-point programs because the lower conditional branch frequency and more regular memory access patterns allow the compiler to try to schedule instructions to avoid hazards. It is more difficult to perform such optimizations in integer programs that have less regular memory accesses, involving more use of pointers. As we will see in Section 4.11, 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 five 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*.

Instruction sets can either make life harder or simpler 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.

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



PREDICTION

4.7 Pipelined Datapath and Control

Figure 4.35 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 4.35, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the

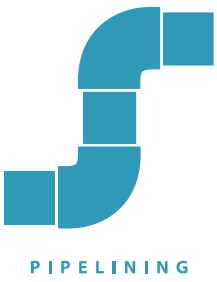
There is less in this than meets the eye.

Tallulah Bankhead, remark to Alexander Woollcott, 1922

Pipeline Overview Summary

Pipelining is a technique that exploits **parallelism** between the instructions in a sequential instruction stream. It has the substantial advantage that, unlike programming a multiprocessor (see Chapter 6), it is fundamentally invisible to the programmer.

In the next few sections of this chapter, we cover the concept of pipelining using the RISC-V 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 4.11. Section 4.11 introduces advanced pipelining concepts, such as superscalar and dynamic scheduling, and Section 4.12 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 4.7. You can then use this understanding to explore the implementation of forwarding and stalls in Section 4.8. You can next read Section 4.9 to learn more about solutions to branch hazards, and finally see how exceptions are handled in Section 4.10.

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.

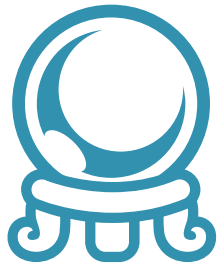
Sequence 1	Sequence 2	Sequence 3
`lw  x10, 0(x10)`	`add  x11, x10, x10`	`addi  x11, x10, 1`
`add  x11, x10, x10`	`addi  x12, x10, 5`	`addi  x12, x10, 2`
	`addi  x14, x11, 5`	`addi  x13, x10, 3`
		`addi  x14, x10, 4`
		`addi  x15, x10, 5`

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 4.11, 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 conditional branch frequencies as well as less predictable branches. Data hazards can be

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



A more sophisticated version of **branch prediction** would have some conditional 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 conditional branches that branch back to the top of the loop. Since they are likely to be taken and they branch backward, we could always predict taken for conditional branches that branch 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 conditional branch and may change predictions for a conditional 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 conditional branches is keeping a history for each conditional 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 conditional branches with more than 90% accuracy (see Section 4.9). When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed conditional 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 4.9.

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

Called the *delayed branch* in computers, this is the solution actually used by the MIPS architecture. The delayed branch always executes the next sequential instruction, with the branch taking place *after* that one instruction delay. It is hidden from the MIPS assembly language programmer because the assembler can automatically arrange the instructions to get the branch behavior desired by the programmer. MIPS software will place an instruction immediately after the delayed branch instruction that is not affected by the branch, and a taken branch changes the address of the instruction that *follows* this safe instruction. In our example, the `add` instruction before the branch in Figure 4.33 does not affect the branch and can be moved after the branch to hide the branch delay fully. Since delayed branches are useful when the branches are short, it is rare to see a processor with a delayed branch of more than one cycle. For longer branch delays, hardware-based branch prediction is usually used.

ANSWER

Figure 3.22 in Chapter 3 shows that conditional branches are 10% of the instructions executed in SPECint2006. Since the other instructions run have a CPI of 1, and conditional branches took one extra clock cycle for the stall, then we would see a CPI of 1.10 and hence a slowdown of 1.10 versus the ideal case.

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 conditional 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 sure you have the right formula to wash uniforms, then just *predict* that it will work and wash the second load while waiting for the first load to dry.

This option does not slow down the pipeline when you are correct. When you are wrong, however, you need to redo the load that was washed while guessing the decision.

Computers do indeed use **prediction** to handle conditional branches. One simple approach is to predict always that conditional branches will be untaken. When you're right, the pipeline proceeds at full speed. Only when conditional branches are taken does the pipeline stall. Figure 4.34 shows such an example:

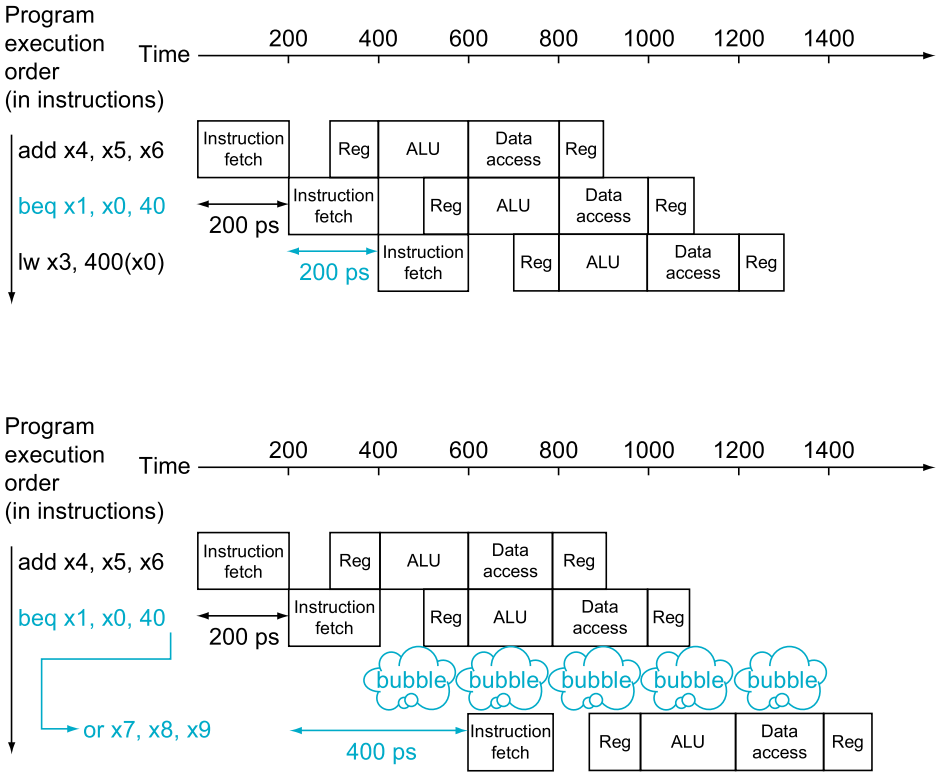


FIGURE 4.34 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 4.33, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. Section 4.9 will reveal the details.

pipeline, we have to wait until the second stage to examine the dry uniform to see if we need to change the washer setup or not. What to do?

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

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

This conservative option certainly works, but it is slow.

The equivalent decision task in a computer is the conditional branch instruction. Notice that we must begin fetching the instruction following the branch on the following 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 a register, calculate the branch address, and update the PC during the second stage of the pipeline (see Section 4.9 for details). Even with this added hardware, the pipeline involving conditional branches would look like Figure 4.33. The instruction to be 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.

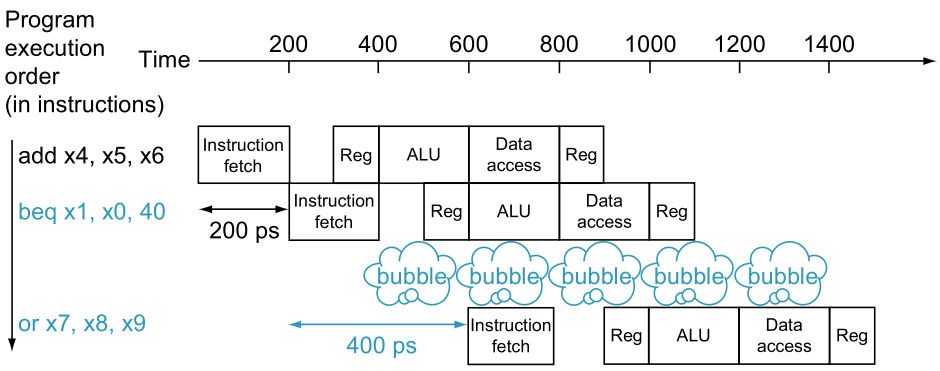


FIGURE 4.33 Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the `or` instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 4.9. The effect on performance, however, is the same as would occur if a bubble were inserted.

```
lw      x1, 0(x31)   // Load b
lw      x2, 8(x31)   // Load e
add     x3, x1, x2   // b + e
sw      x3, 24(x31)  // Store a
ld      x4, 16(x31)  // Load f
add     x5, x1, x4   // b + f
sw      x5, 32(x31)  // Store c
```

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 previous `lw` instruction. Notice that forwarding 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    x1, 0(x31)
lw    x2, 4(x31)
lw    x4, 8(x31)
add   x3,  x1, x2
sw    x3, 12(x31)
add   x5, x1, x4
sw    x5, 16(x31)
```

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 RISC-V architecture, in addition to the three mentioned on page 287. Each RISC-V 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 are strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner. In our laundry

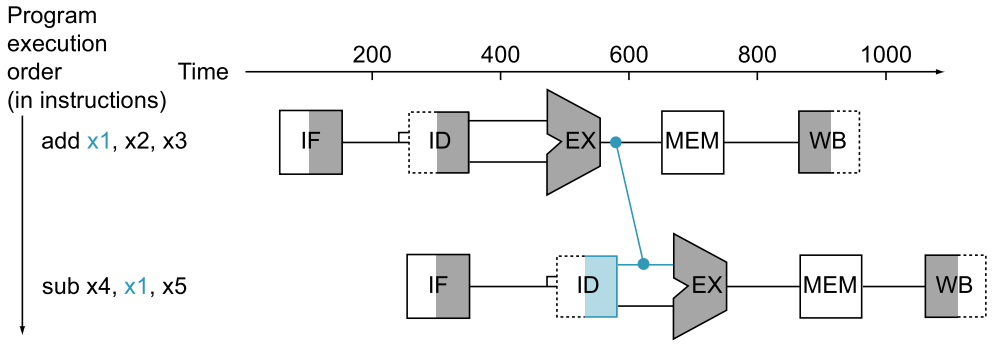


FIGURE 4.31 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `x1` read in the second stage of `sub`.

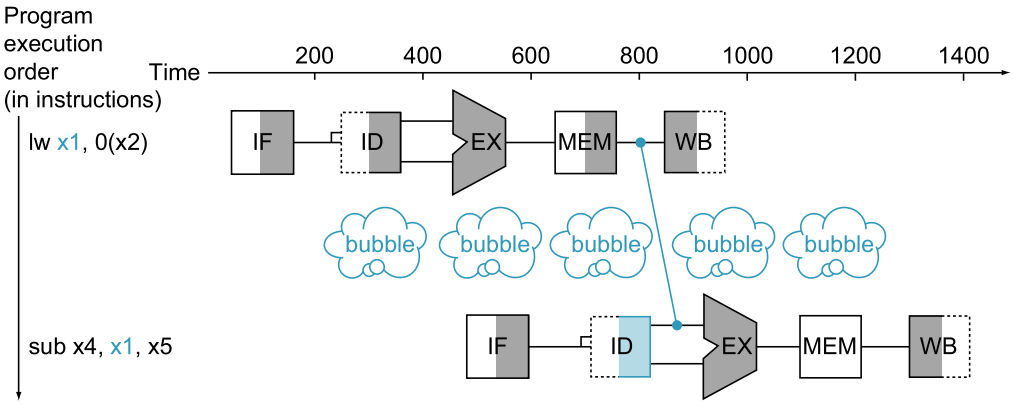


FIGURE 4.32 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since 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.

EXAMPLE

Reordering Code to Avoid Pipeline Stalls

Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

Here is the generated RISC-V code for this segment, assuming all variables are in memory and are addressable as offsets from `x31`:

Forwarding with Two Instructions

EXAMPLE

For the two instructions above, show what pipeline stages would be connected by forwarding. Use the drawing in Figure 4.30 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 4.27.

ANSWER

Figure 4.31 shows the connection to forward the value in `x1` after the execution stage of the `add` instruction as input to the execution stage of the `sub` instruction.

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

Forwarding works very well and is described in detail in Section 4.8. It cannot prevent all pipeline stalls, however. For example, suppose the first instruction was a load of `x1` instead of an add. As we can imagine from looking at Figure 4.31, the desired data would be available only *after* the fourth stage of the first instruction in the dependence, which is too late for the *input* of the third stage of `sub`. Hence, even with forwarding, we would have to stall one stage for a **load-use data hazard**, as Figure 4.32 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 4.8 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.

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

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

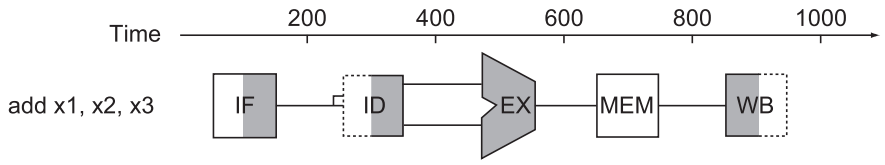


FIGURE 4.30 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.27. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction 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.

Structural Hazard

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.

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

As we said above, the RISC-V 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 4.29 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 are needed to execute the instruction are not yet available.

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Suppose you found a sock at the folding station for which no match existed. One possible strategy is to run down to your room and search through your clothes bureau to see if you can find the match. Obviously, while you are doing the search, loads that have completed drying are ready to fold and those that have finished washing 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 that sum (`x19`):

```
add  x19, x0, x1
sub  x2, x19, x3
```

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 far 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**.

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.

However, 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:

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

Pipelining improves performance by *increasing instruction throughput, in contrast 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 RISC-V instruction set, which was designed for pipelined execution.

First, all RISC-V 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. Modern implementations of the x86 architecture actually translate x86 instructions into simple operations that look like RISC-V instructions and then pipeline the simple operations rather than the native x86 instructions! (See Section 4.11.)

Second, RISC-V has just a few instruction formats, with the source and destination register fields being located in the same place in each instruction.

Third, memory operands only appear in loads or stores in RISC-V. 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. We will shortly see the downside of longer pipelines.

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.

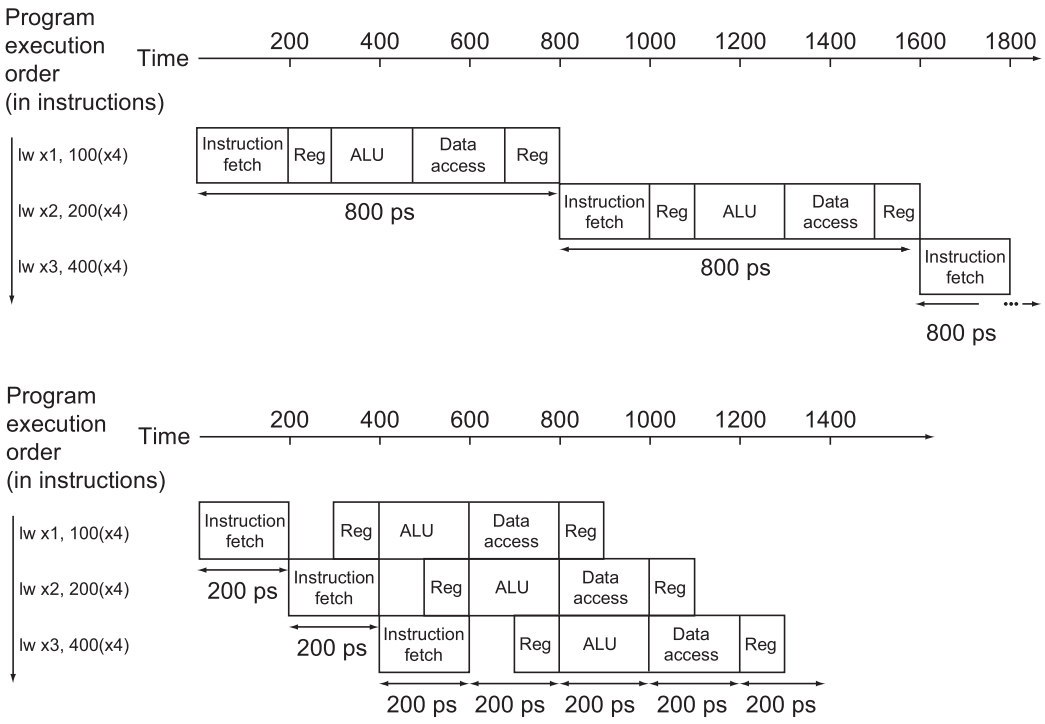


FIGURE 4.29 Single-cycle, nonpipelined execution (top) versus pipelined execution (bottom). Both use the same hardware components, whose time is listed in Figure 4.28. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.27. 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.

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

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

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.

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

Single-Cycle versus Pipelined Performance

EXAMPLE

To make this discussion concrete, let's create a pipeline. In this example, and in the rest of this chapter, we limit our attention to seven instructions: load word (`lw`), store word (`sw`), add (`add`), subtract (`sub`), AND (`and`), OR (`or`), and branch if equal (`beq`).

Contrast the average time between instructions of a single-cycle implementation, in which all instructions take one clock cycle, to a pipelined implementation. Assume that the operation times for the major functional units in this example are 200 ps for memory access for instructions or data, 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 4.28 shows the time required for each of the seven instructions. The single-cycle design must allow for the slowest instruction—in Figure 4.28 it is `lw`—so the time required for every instruction is 800 ps. Similarly to Figure 4.27, Figure 4.29 compares nonpipelined and pipelined execution of three load register instructions. Thus, the time between the first and fourth instructions in the nonpipelined design is 3×800 ps or 2400 ps.

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

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

FIGURE 4.28 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.

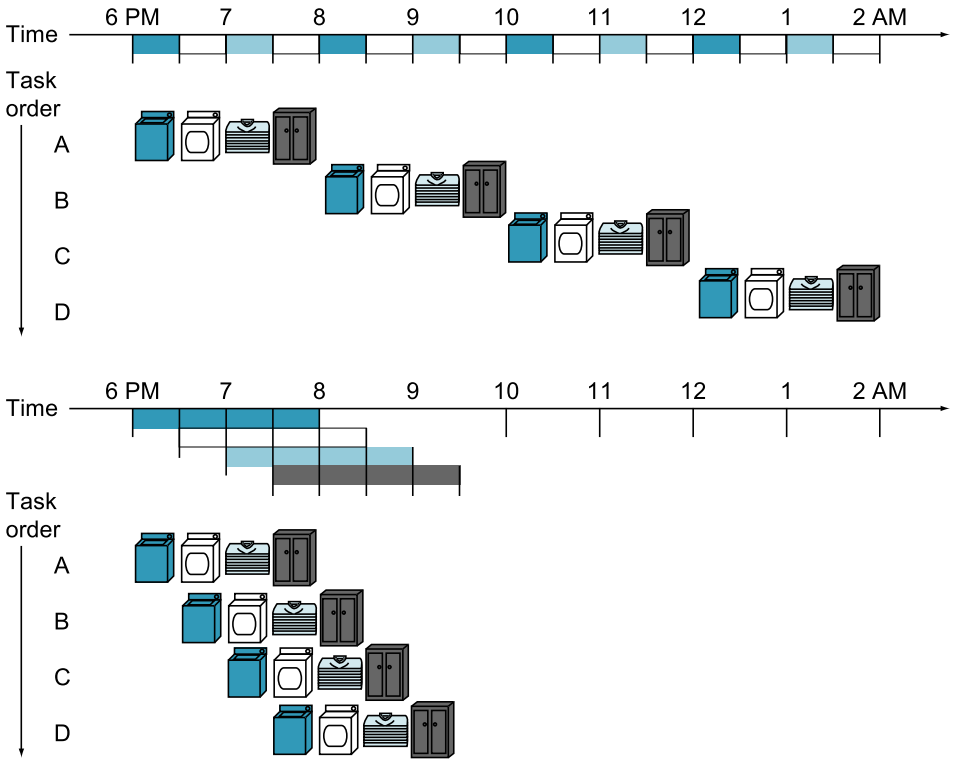


FIGURE 4.27 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for four loads of washing, while pipelined laundry 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.

only show four loads. Notice that at the beginning and end of the workload in the pipelined version in Figure 4.27, 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 four, then the stages will be full most of the time and the increase in throughput will be very close to four.

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

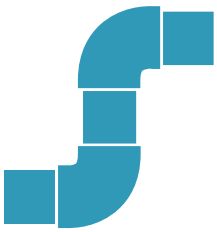
1. Fetch instruction from memory.
2. Read registers and decode the instruction.
3. Execute the operation or calculate an address.
4. Access an operand in data memory (if necessary).
5. Write the result into a register (if necessary).

instructors see pedagogic advantages to explaining multicycle implementation before pipelining, so we off er this implementation option online.

4.6 An Overview of Pipelining

Never waste time.
American proverb

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



PIPELINING

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

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 4.11 and 4.12 to see an introduction to the advanced pipelining techniques used in recent processors such as the Intel Core i7 and ARM Cortex-A53. If you are curious about exploring the anatomy of a pipelined computer, this section is a good introduction to Sections 4.7 through 4.10.

Anyone who has done a lot of laundry has intuitively used pipelining. The *nonpipelined* 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 this load is done, start over with the next dirty load.

The *pipelined* approach takes much less time, as Figure 4.27 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 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 five times as long as one load, while 20 loads of sequential laundry takes 20 times as long as one load. It's only 2.3 times faster in Figure 4.27, because we