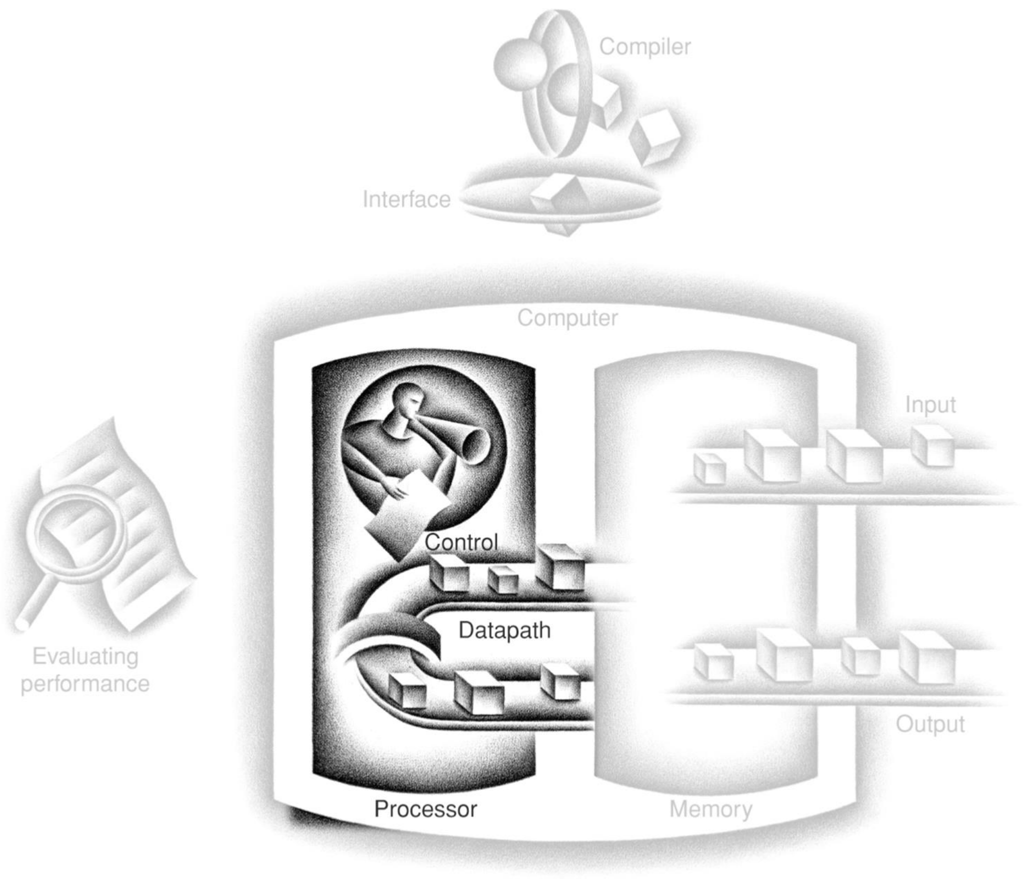
4

The Processor

In a major matter, no details are small.

French Proverb

OUTLINE

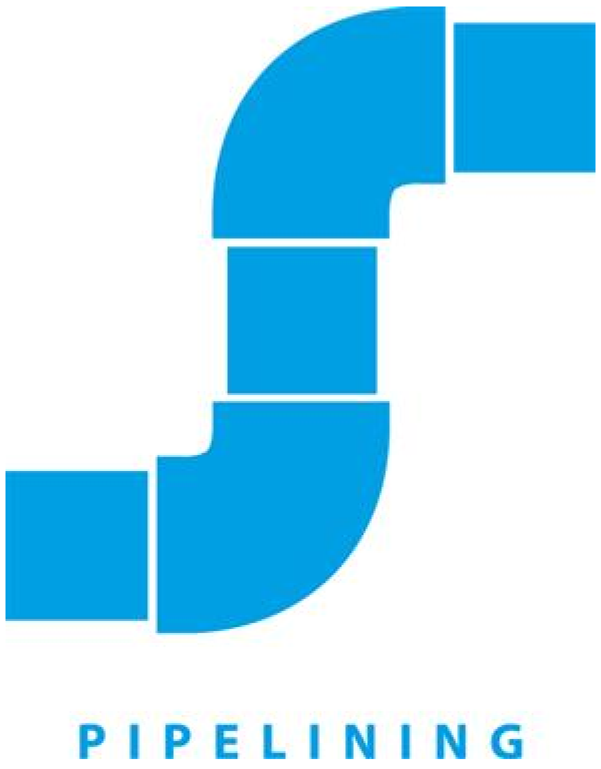


Five Classic Components of a Computer

4.1 Introduction

Chapter 1 explains that the performance of a computer is determined by three key factors: instruction count, clock cycle time, and *clock cycles per instruction* (CPI). Chapter 2 explains that the compiler and the instruction set architecture determine the instruction count required for a given program. However, the implementation of the processor determines both the clock cycle time and the number of clock cycles per instruction. In this chapter, we construct the datapath and control unit for two different implementations of the MIPS instruction set.

This chapter contains an explanation of the principles and techniques used in implementing a processor, starting with a highly abstract and simplified overview in this section. It is followed by a section that builds up a datapath and constructs a simple version of a processor sufficient to implement an instruction set like MIPS. The bulk of the chapter covers a more realistic **pipelined** MIPS implementation, followed by a section that develops the concepts necessary to implement more complex instruction sets, like the x86.



For the reader interested in understanding the high-level interpretation of instructions and its impact on program performance, this initial section and Section 4.6 present the basic concepts of pipelining. Recent trends are covered in Section 4.11, and Section 4.12 describes the recent Intel Core i7 and ARM Cortex-A8 architectures. Section 4.13 shows how to use instruction-level parallelism to more than double the performance of the matrix multiply from Section 3.8. These sections provide enough background to understand the pipeline concepts at a high level.

For the reader interested in understanding the processor and its performance in more depth, Sections 4.3, 4.4, and 4.7 will be useful. Those interested in learning how to build a processor should also cover 4.2, 4.8, 4.9, and 4.10. For readers with an interest in modern hardware design,

Section 4.14 describes how hardware design languages and CAD tools are used to implement hardware, and then how to use a hardware design language to describe a pipelined implementation. It also gives several more illustrations of how pipelining hardware executes.

A Basic MIPS Implementation

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

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

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

In examining the implementation, we will have the opportunity to see how the instruction set architecture determines many aspects of the implementation, and how the choice of various implementation strategies affects the clock rate and CPI for the computer. Many of the key design principles introduced in Chapter 1 can be illustrated by looking at the implementation, such as *Simplicity favors regularity*. In addition, most concepts used to implement the MIPS subset in this chapter are the same basic ideas that are used to construct a broad spectrum of computers, from high-performance servers to general-purpose microprocessors to embedded processors.

An Overview of the Implementation

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

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

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction. The simplicity and regularity of the MIPS instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and branches for comparison. After using the ALU, the actions required to complete various instruction classes differ. A memory-reference instruction will need to access the memory either to read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register. Lastly, for a branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by 4 to get the address of the next instruction.

Figure 4.1 shows the high-level view of a MIPS implementation, focusing on the various functional units and their interconnection. Although this figure shows most of the flow of data through the processor, it omits two important aspects of instruction execution.

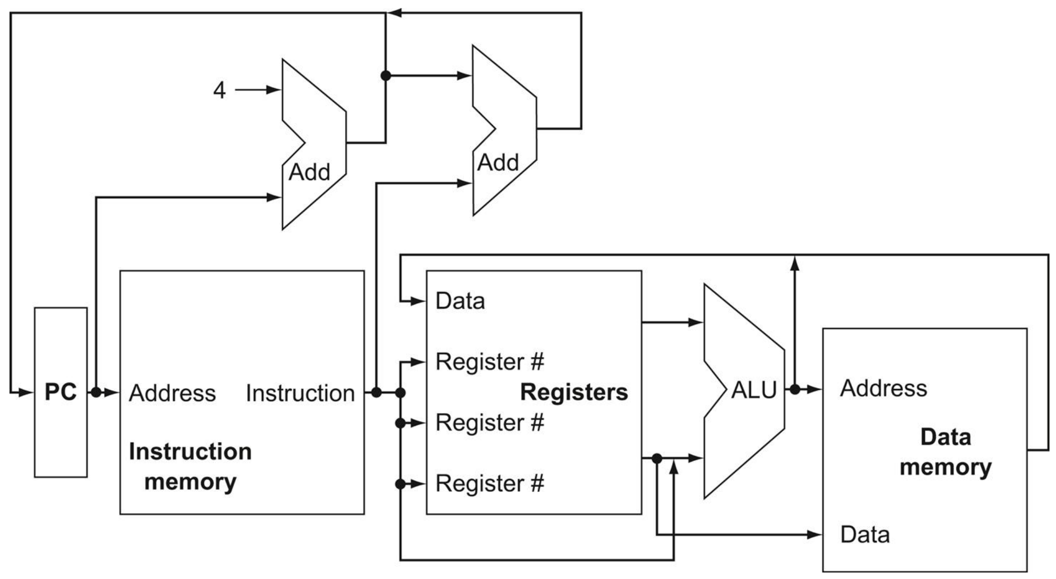


FIGURE 4.1 **An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.**
All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either load a value from memory into the registers or store a value from the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the

ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

First, in several places, Figure 4.1 shows data going to a particular unit as coming from two different sources. For example, the value written into the PC can come from one of two adders, the data written into the register file can come from either the ALU or the data memory, and the second input to the ALU can come from a register or the immediate field of the instruction. In practice, these data lines cannot simply be wired together; we must add a logic element that chooses from among the multiple sources and steers one of those sources to its destination. This selection is commonly done with a device called a *multiplexor*, although this device might better be called a *data selector*. Appendix B describes the multiplexor, which selects from among several inputs based on the setting of its control lines. The control lines are set based primarily on information taken from the instruction being executed.

The second omission in Figure 4.1 is that several of the units must be controlled depending on the type of instruction. For example, the data memory must read on a load and write on a store. The register file must be written only on a load or on an arithmetic-logical instruction. And, of course, the ALU must perform one of several operations. (Appendix B describes the detailed design of the ALU.) Like the multiplexors, control lines are set on the basis of various fields in the instruction to direct these operations.

Figure 4.2 shows the datapath of Figure 4.1 with the three required multiplexors added, as well as control lines for the major functional units. A *control unit*, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors. The third multiplexor, which determines whether PC + 4 or the branch destination address is written into the PC, is set based on the Zero output of the ALU, which is used to perform the comparison of a beq instruction. The regularity and simplicity of the MIPS instruction set means that a simple decoding process can be used to determine how to set the control lines.

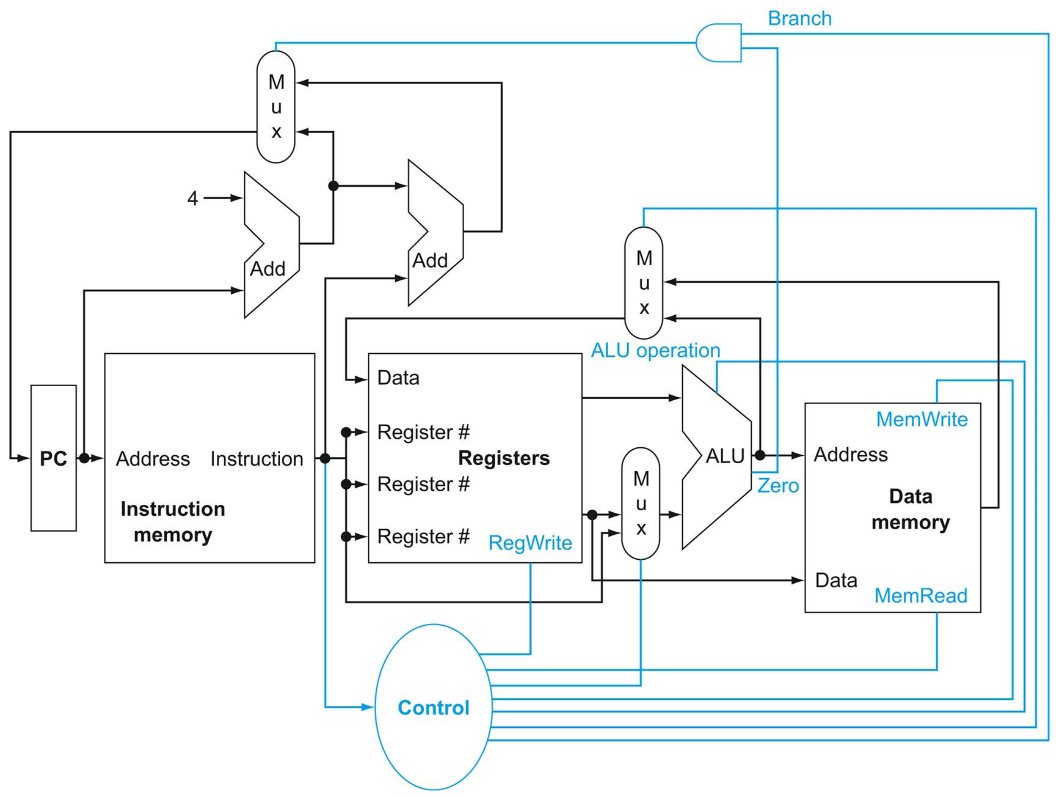


FIGURE 4.2 The basic implementation of the MIPS subset, including the necessary multiplexors and control lines.
The top multiplexor ("Mux") controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that "ANDs" together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store). The added control lines are

straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.

In the remainder of the chapter, we refine this view to fill in the details, which requires that we add further functional units, increase the number of connections between units, and, of course, enhance a control unit to control what actions are taken for different instruction classes. Sections 4.3 and 4.4 describe a simple implementation that uses a single long clock cycle for every instruction and follows the general form of Figures 4.1 and 4.2. In this first design, every instruction begins execution on one clock edge and completes execution on the next clock edge.

While easier to understand, this approach is not practical, since the clock cycle must be severely stretched to accommodate the longest instruction. After designing the control for this simple computer, we will look at faster implementations with all their complexities, including exceptions.

Check Yourself

How many of the five classic components of a computer—shown on page 255—do Figures 4.1 and 4.2 include?

4.2 Logic Design Conventions

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

The datapath elements in the MIPS implementation consist of two different types of logic elements: elements that operate on data values and elements that contain state. The elements that operate on data values are all

combinational, which means that their outputs depend only on the current inputs. Given the same input, a combinational element always produces the same output. The ALU in Figure 4.1 and discussed in **Appendix B** is an example of a combinational element. Given a set of inputs, it always produces the same output because it has no internal storage.

combinational element
An operational element, such as an AND gate or an ALU.

Other elements in the design are not combinational, but instead contain *state*. An element contains state if it has some internal storage. We call these elements **state elements** because, if we pulled the power plug on the computer, we could restart it accurately by loading the state elements with the values they contained before we pulled the plug. Furthermore, if we saved and restored the state elements, it would be as if the computer had never lost power. Thus, these state elements completely characterize the computer. In Figure 4.1, the instruction and data memories, as well as the registers, are all examples of state elements.

state element
A memory element, such as a register or a memory.

A state element has at least two inputs and one output. The required inputs are the data value to be written into the element and the clock, which determines when the data value is written. The output from a state element provides the value that was written in an earlier clock cycle. For example, one of the logically simplest state elements is a D-type flip-flop (see **Appendix B**), which has exactly these two inputs (a value and a clock) and one output. In addition to flip-flops, our MIPS implementation uses two other types of state elements: memories and registers, both of which appear in Figure 4.1. The clock is used to determine when the state element should be written; a state element can be read at any time.

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

Clocking Methodology

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

clocking methodology

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

For simplicity, we will assume an **edge-triggered clocking** methodology. An edge-triggered clocking methodology means that any values stored in a sequential logic element are updated only on a clock edge, which is a quick transition from low to high or *vice versa* (see Figure 4.3). Because only state elements can store a data value, any collection of combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements. The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle.

edge-triggered clocking

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

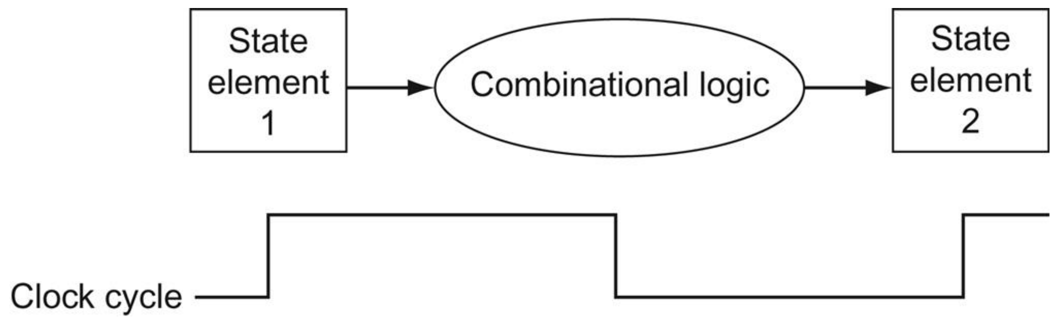


FIGURE 4.3 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements in this chapter, including memory, are assumed to be positive edge-triggered; that is, they change on the rising clock edge.

Figure 4.3 shows the two state elements surrounding a block of combinational logic, which operates in a single clock cycle: all signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle. The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

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

control signal

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

We will use the word **asserted** to indicate a signal that is logically high and *assert* to specify that a signal should be driven logically high, and *deassert* or **deasserted** to represent logically low. We use the terms assert and deassert because when we implement hardware, at times 1 represents logically high and at times it can represent logically low.

asserted

The signal is logically high or true.

deasserted

The signal is logically low or false.

An edge-triggered methodology allows us to read the contents of a register, send the value through some combinational logic, and write that register in the same clock cycle. Figure 4.4 gives a generic example. It doesn't matter whether we assume that all writes take place on the rising clock edge (from low to high) or on the falling clock edge (from high to low), since the inputs to the combinational logic block cannot change except on the chosen clock edge. In this book we use the rising clock edge. With an edge-triggered timing methodology, there is *no* feedback within a single clock cycle, and the logic in Figure 4.4 works correctly. In **Appendix B**, we briefly discuss additional timing constraints (such as setup and hold times) as well as other timing methodologies.

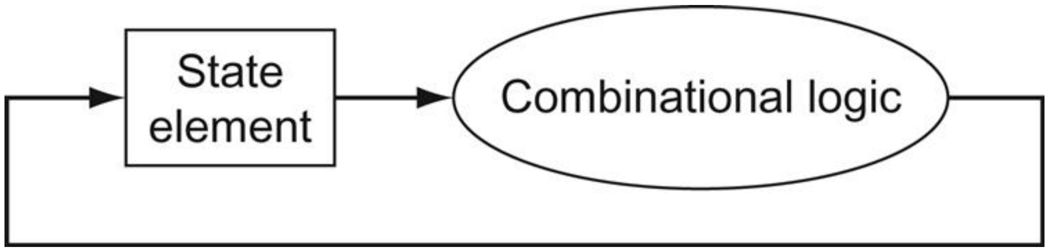


FIGURE 4.4 **An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values.** Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. Feedback cannot occur within one clock cycle because of the edge-triggered update of the state element. If feedback were possible, this design could not work properly. Our designs in this chapter and the next rely on the edge-triggered timing methodology and on structures like the one shown in this figure.

For the 32-bit MIPS architecture, nearly all of these state and logic elements will have inputs and outputs that are 32 bits wide, since that is the width of most of the data handled by the processor. We will make it clear whenever a unit has an input or output that is other than 32 bits in width. The figures will indicate *buses*, which are signals wider than 1 bit, with thicker lines. At times, we will want to combine several buses to form a wider bus; for example, we may want to obtain a 32-bit bus by combining two 16-bit buses. In such cases, labels on the bus lines will make it clear that we are concatenating buses to form a wider bus. Arrows are also added to help clarify the direction of the flow of data between elements. Finally, **color** indicates a control signal as opposed to a signal that carries data; this distinction will become clearer as we proceed through this chapter.

Check Yourself

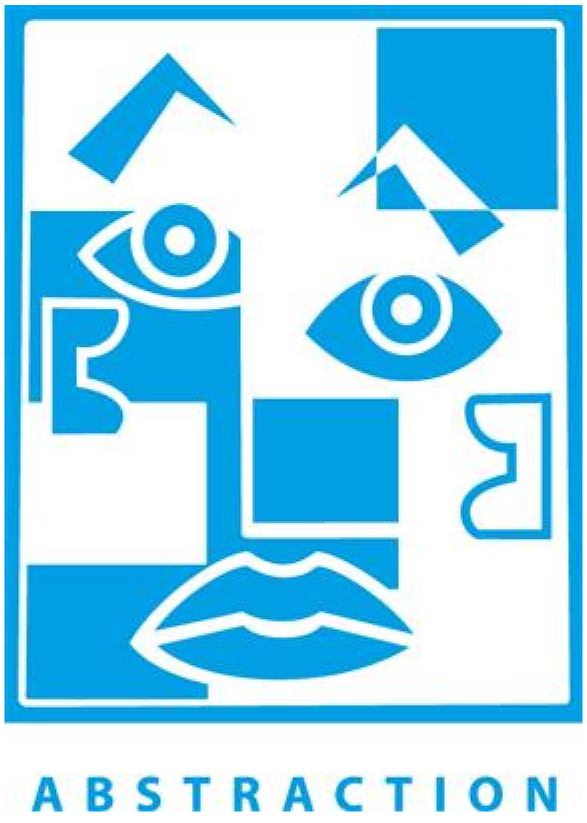
True or false: Because the register file is both read and written on the same clock cycle, any MIPS datapath using edge-triggered writes must have more than one copy of the register file.

Elaboration

There is also a 64-bit version of the MIPS architecture, and, naturally enough, most paths in its implementation would be 64 bits wide.

4.3 Building a Datapath

A reasonable way to start a datapath design is to examine the major components required to execute each class of MIPS instructions. Let's start at the top by looking at which **datapath elements** each instruction needs, and then work our way down through the levels of **abstraction**. When we show the datapath elements, we will also show their control signals. We use abstraction in this explanation, starting from the bottom up.



datapath element

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

Figure 4.5a shows the first element we need: a memory unit to store the instructions of a program and supply instructions given an address. Figure 4.5b also shows the **program counter (PC)**, which as we saw in Chapter 2 is a register that holds the address of the current instruction. Lastly, we will need an adder to increment the PC to the address of the next instruction. This adder, which is combinational, can be built from the ALU described in detail in **Appendix B** simply by wiring the control lines so that the control always specifies an add operation. We will draw such an ALU with the label *Add*, as in Figure 4.5, to indicate that it has been permanently made an adder and cannot perform the other ALU functions.

program counter (PC)

The register containing the address of the instruction in the program being executed.

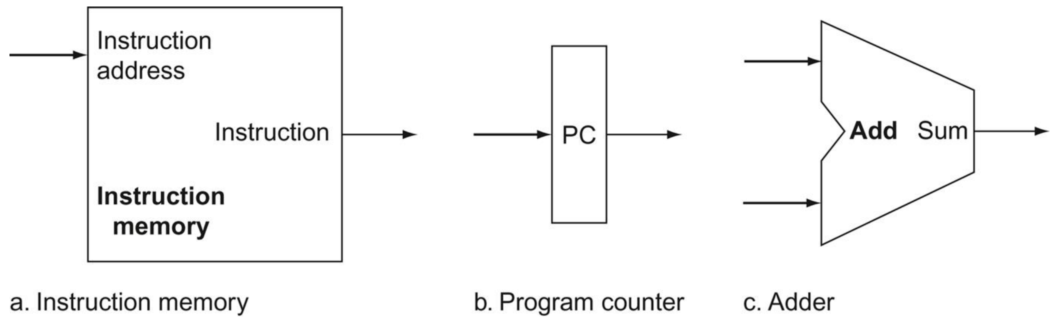


FIGURE 4.5 **Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.**
The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later. Figure 4.6 shows how to combine the three elements from Figure 4.5 to form the portion of a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

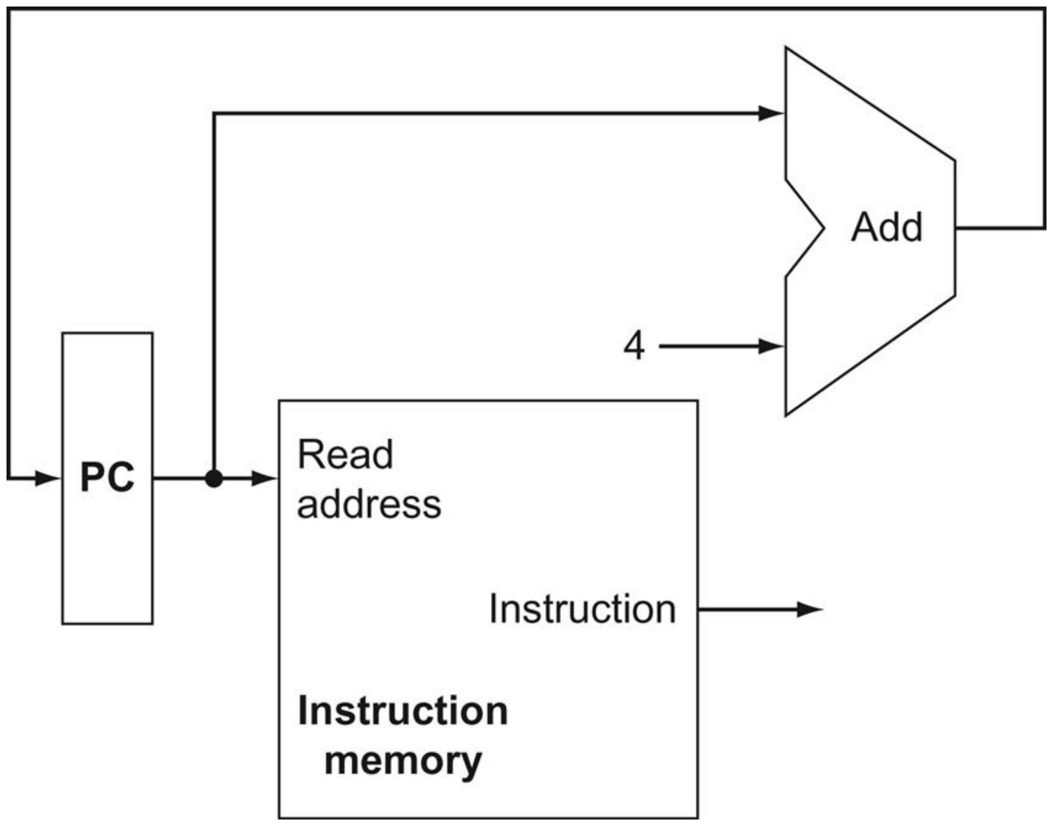


FIGURE 4.6 **A portion of the datapath used for fetching instructions and incrementing the program counter.**
The fetched instruction is used by other parts of the datapath.

Now let's consider the R-format instructions (see Figure 2.20 on page 126). They all read two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions either *R-type instructions* or *arithmetic-logical instructions* (since they perform arithmetic or logical operations). This instruction class includes `add`, `sub`, `AND`, `OR`, and `slt`, which were introduced in Chapter 2.

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

The processor's 32 general-purpose registers are stored in a structure called a **register file**. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer. In addition, we will need an ALU to operate on the values read from the registers.

register file

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

R-format instructions have three register operands, so we will need to read two data words from the register file and write one data word into the register file for each instruction. For each data word to be read from the registers, we need an input to the register file that specifies the *register number* to be read and an output from the register file that will carry the value that has been read from the registers. To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the *data* to be written into the register. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge. Figure 4.7a shows the result; we need a total of four inputs (three for register numbers and one for data) and two outputs (both for data). The register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$), whereas the data input and two data output buses are each 32 bits wide.

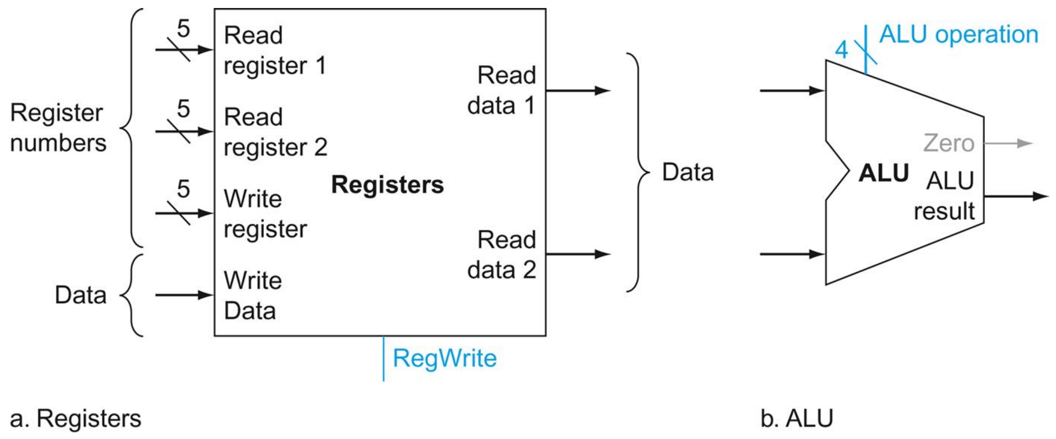


FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section B.8 of **Appendix B**. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU

is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in **Appendix B**. We will use the Zero detection output of the ALU shortly to implement branches. The overflow output will not be needed until Section 4.10, when we discuss exceptions; we omit it until then.

Figure 4.7b shows the ALU, which takes two 32-bit inputs and produces a 32-bit result, as well as a 1-bit signal if the result is 0. The 4-bit control signal of the ALU is described in detail in **Appendix B**; we will review the ALU control shortly when we need to know how to set it.

Next, consider the MIPS load word and store word instructions, which have the general form `lw $t1,offset_value($t2)` or `sw $t1,offset_value ($t2)`. These instructions compute a memory address by adding the base register, which is `$t2`, to the 16-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in `$t1`. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is `$t1`. Thus, we will need both the register file and the ALU from Figure 4.7.

Furthermore, we will need a unit to **sign-extend** the 16-bit offset field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to. The data memory must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory. Figure 4.8 shows these two elements.

sign-extend

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

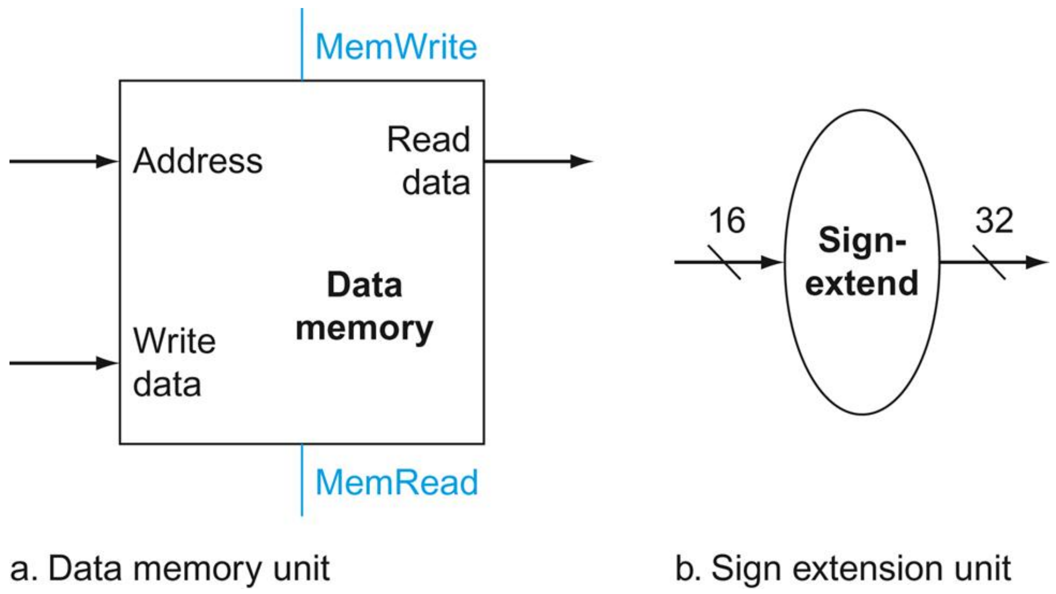


FIGURE 4.8 **The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the sign extension unit.** The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in Chapter 5. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output (see Chapter 2). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See

Section B.8 of **Appendix B** for further discussion of how real memory chips work.

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

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

branch target address

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

To deal with the latter complication, we will need to shift the offset field by 2.

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

branch taken

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

branch not taken or (untaken branch)

A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

Thus, the branch datapath must do two operations: compute the branch target address and compare the register contents. (Branches also affect the instruction fetch portion of the datapath, as we will deal with shortly.) Figure 4.9 shows the structure of the datapath segment that handles branches. To compute the branch target address, the branch datapath includes a sign extension unit, from Figure 4.8 and an adder. To perform the compare, we need to use the register file shown in Figure 4.7a to supply the two register operands (although we will not need to write into the register file). In addition, the comparison can be done using the ALU we designed in **Appendix B**. Since that ALU provides an output signal that indicates whether the result was 0, we can send the two register operands to the ALU with the control set to do a subtract. If the Zero signal out of the ALU unit is asserted, we know that the two values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equal test of branches. Later, we will show exactly how to connect the control signals of the ALU for use in the datapath.

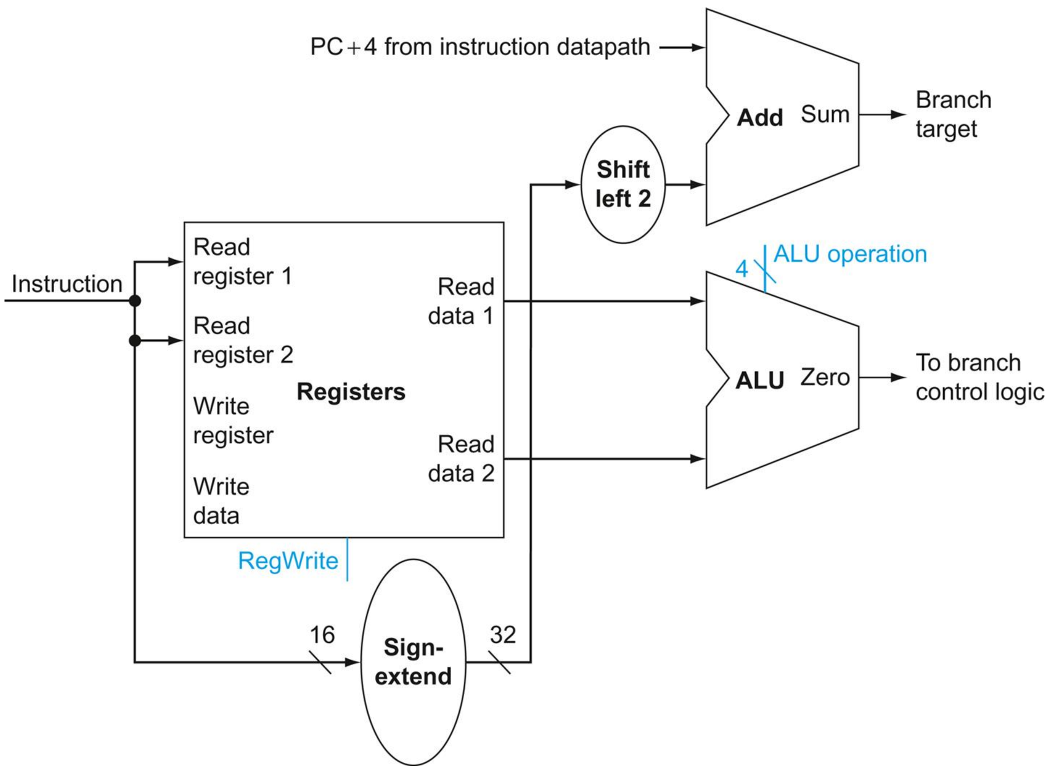


FIGURE 4.9 **The portion of a datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits.** The unit labeled *Shift left 2* is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended offset field; no actual shift hardware is needed, since the amount of the “shift” is constant. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only “sign bits.” Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

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

Elaboration

In the MIPS instruction set, **branches are delayed**, meaning that the instruction immediately following the branch is always executed, *independent* of whether the branch condition is true or false. When the condition is false, the execution looks like a normal branch. When the condition is true, a delayed branch first executes the instruction immediately following the branch in sequential instruction order before jumping to the specified branch target address. The motivation for delayed branches arises from how pipelining affects branches (see Section 4.9). For simplicity, we generally ignore delayed branches in this chapter and implement a nondelayed `beq` instruction.

delayed branch

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

Creating a Single Datapath

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

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

Building a Datapath

Example

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

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

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

Answer

To create a datapath with only a single register file and a single ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus, one multiplexor is placed at the ALU input and another at the data input to the register file. Figure 4.10 shows the operational portion of the combined datapath.

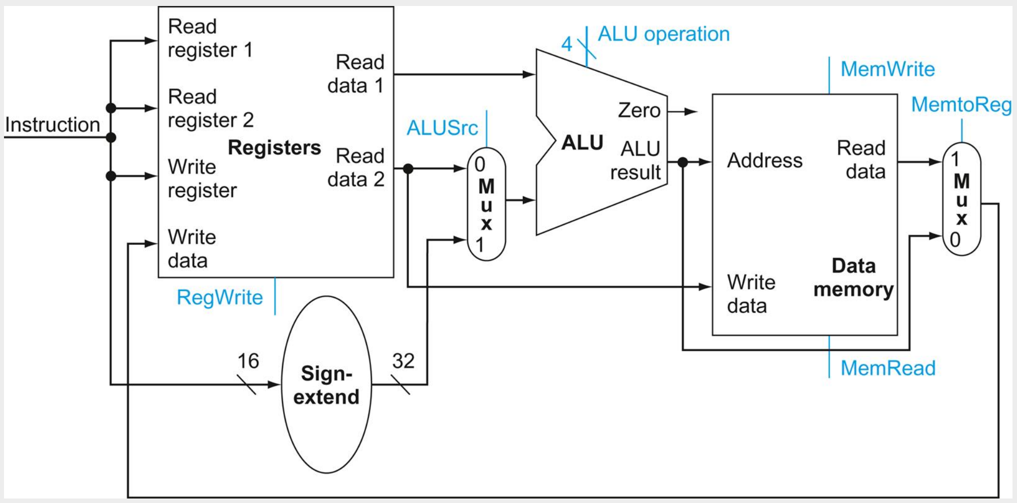


FIGURE 4.10 **The datapath for the memory instructions and the R-type instructions.** This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example.

Now we can combine all the pieces to make a simple datapath for the core MIPS architecture by adding the datapath for instruction fetch (Figure 4.6), the datapath from R-type and memory instructions (Figure 4.10), and the datapath for branches (Figure 4.9). Figure 4.11 shows the datapath we obtain by composing the separate pieces. The branch instruction uses the main ALU for comparison of the register operands, so we must keep the adder from Figure 4.9 for computing the branch target address. An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC.

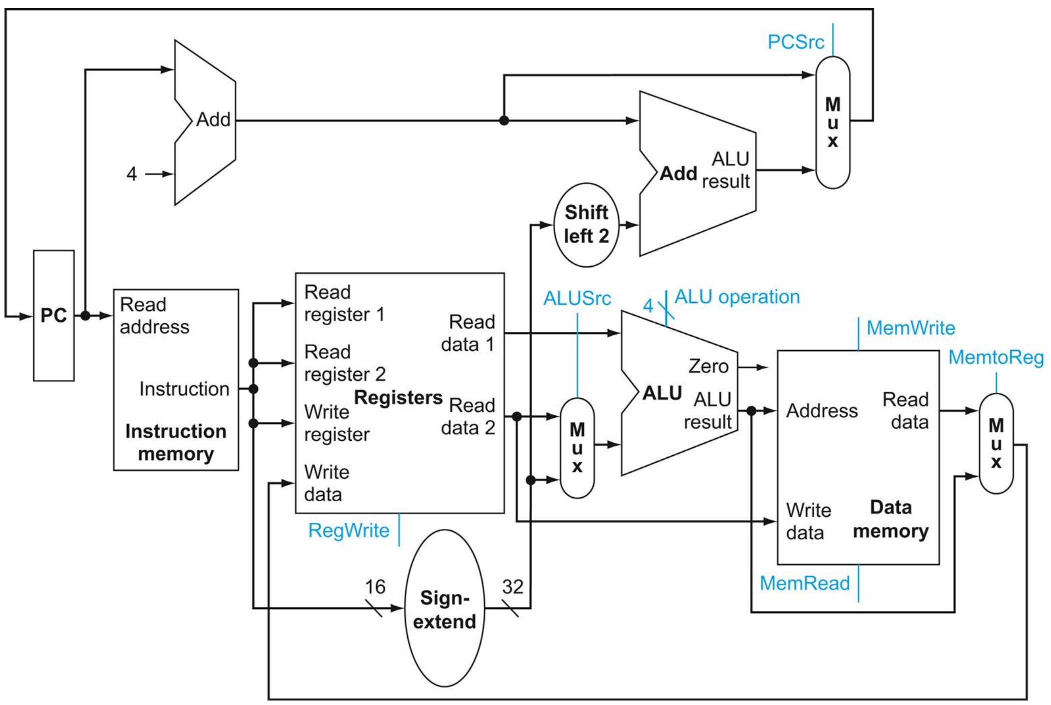


FIGURE 4.11 The simple datapath for the core MIPS architecture combines the elements required by different instruction classes. The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches. The support for jumps will be added later.

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

Check Yourself

I. Which of the following is correct for a load instruction? Refer to Figure 4.10.

a. MemtoReg should be set to cause the data from memory to be sent to the register file.
b. MemtoReg should be set to cause the correct register destination to be sent to the register file.
c. We do not care about the setting of MemtoReg for loads.

II. The single-cycle datapath conceptually described in this section *must* have separate instruction and data memories, because
a. the formats of data and instructions are different in MIPS, and hence different memories are needed.
b. having separate memories is less expensive.
c. the processor operates in one clock cycle and cannot use a single-ported memory for two different accesses within that clock cycle.

4.4 A Simple Implementation Scheme

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

The ALU Control

The MIPS ALU in **Appendix B** defines the 6 following combinations of four control inputs:

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

Depending on the instruction class, the ALU will need to perform one of these first five functions. (NOR is needed for other parts of the MIPS instruction set not found in the subset we are implementing.) For load word and store word instructions, we use the ALU to compute the memory address by addition. For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field in the low-order bits of the instruction (see Chapter 2). For branch equal, the ALU must perform a subtraction.

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

In Figure 4.12, we show how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code. Later in this chapter we will see how the ALUOp bits are generated from the main control unit.

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

FIGURE 4.12 **How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction.**
The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we “don’t care” about the value of the function code, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input.

See **Appendix B**.

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

There are several different ways to implement the mapping from the 2-bit ALUOp field and the 6-bit funct field to the four ALU operation control bits. Because only a small number of the 64 possible values of the function field are of interest and the function field is used only when the ALUOp bits equal 10, we can use a small piece of logic that recognizes the subset of possible values and causes the correct setting of the ALU control bits.

As a step in designing this logic, it is useful to create a *truth table* for the interesting combinations of the function code field and the ALUOp bits, as we've done in Figure 4.13; this **truth table** shows how the 4-bit ALU control is set depending on these two input fields. Since the full truth table is very large (2^8 = 256 entries) and we don't care about the value of the ALU control for many of these input combinations, we show only the truth table entries for which the ALU control must have a specific value. Throughout this chapter, we will use this practice of showing only the truth table entries for outputs that must be asserted and not showing those that are all deasserted or don't care. (This practice has a disadvantage, which we discuss in Section D.2 of **Appendix D**.)

truth table

From logic, a representation of a logical operation by listing all the values of the inputs and then in each case showing what the resulting outputs should be.

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

FIGURE 4.13 **The truth table for the 4 ALU control bits (called Operation).**
The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Note that when the function field is used, the first 2 bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

Because in many instances we do not care about the values of some of the inputs, and because we wish to keep the tables compact, we also include **don't-care terms**. A don't-care term in this truth table (represented by an X in an input column) indicates that the output does not depend on the value of the input corresponding to that column. For example, when the ALUOp bits are 00, as in the first row of Figure 4.13, we always set the ALU control to 0010, independent of the function code. In this case, then, the function code inputs will be don't cares in this line of the truth table. Later, we will see examples of another type of don't-care term. If you are unfamiliar with the concept of don't-care terms, see **Appendix B** for more information.

don't-care term

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

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

Designing the Main Control Unit

Now that we have described how to design an ALU that uses the function code and a 2-bit signal as its control inputs, we can return to looking at the rest of the control. To start this process, let’s identify the fields of an instruction and the control lines that are needed for the datapath we constructed in Figure 4.11. To understand how to connect the fields of an instruction to the datapath, it is useful to review the formats of the three instruction classes: the R-type, branch, and load-store instructions. Figure 4.14 shows these formats.

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

FIGURE 4.14 **The three instruction classes (R-type, load and store, and branch) use two different instruction formats.**
The jump instructions use another format, which we will discuss shortly. (a) Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. The ALU function is in the funct field and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are `add`, `sub`, `AND`, `OR`, and `slt`. The shamt field is used only for shifts; we will ignore it in this chapter. (b) Instruction format for load (opcode = 35_{ten}) and store (opcode = 43_{ten}) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory. (c) Instruction format for branch equal (opcode = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC + 4 to compute the branch target address.

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

- The op field, which as we saw in Chapter 2 is called the **opcode**, is always contained in bits 31:26. We will refer to this field as Op[5:0].
- The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and store.
- The base register for load and store instructions is always in bit positions 25:21 (rs).
- The 16-bit offset for branch equal, load, and store is always in positions 15:0.
- The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd). Thus, we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

opcode

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

The first design principle from Chapter 2—*simplicity favors regularity*—pays off here in specifying control.

Using this information, we can add the instruction labels and extra multiplexor (for the Write register number input of the register file) to the simple datapath. Figure 4.15 shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexors. Since all the multiplexors have two inputs, they each require a single control line.

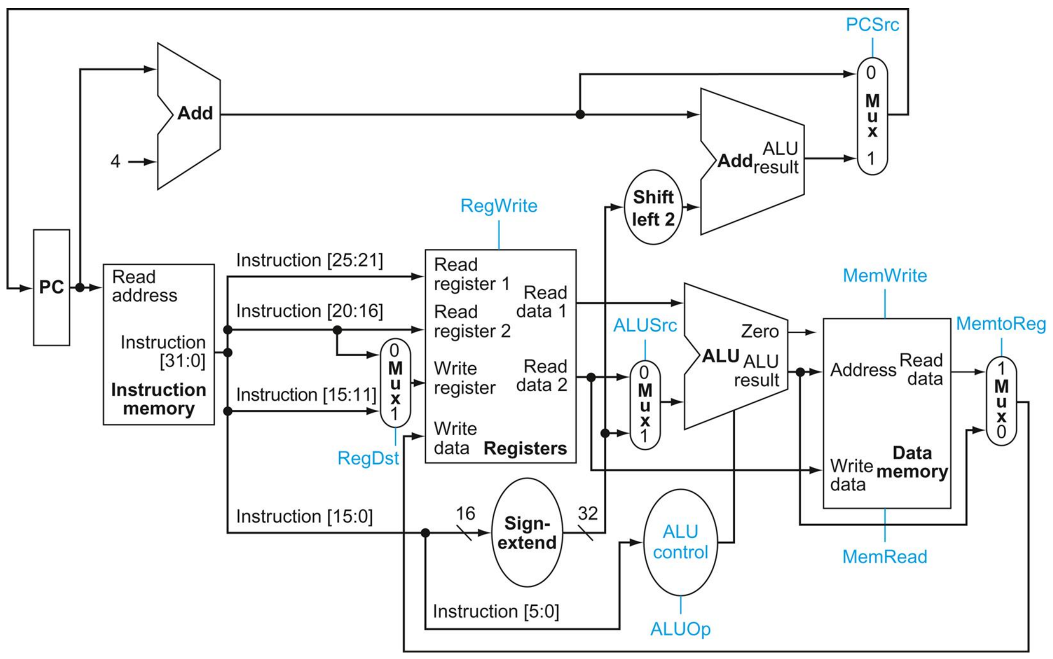


FIGURE 4.15 **The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified.**
The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

Figure 4.15 shows seven single-bit control lines plus the 2-bit ALUOp control signal. We have already defined how the ALUOp control signal works, and it is useful to define what the seven other control signals do informally before we determine how to set these control signals during instruction execution. Figure 4.16 describes the function of these seven control lines.

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

FIGURE 4.16 **The effect of each of the seven control signals.**
When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See **Appendix B** for further discussion of this problem.)

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

These nine control signals (seven from Figure 4.16 and two for ALUOp) can now be set on the basis of six input signals to the control unit, which are the opcode bits 31 to 26. Figure 4.17 shows the datapath with the control unit and the control signals.

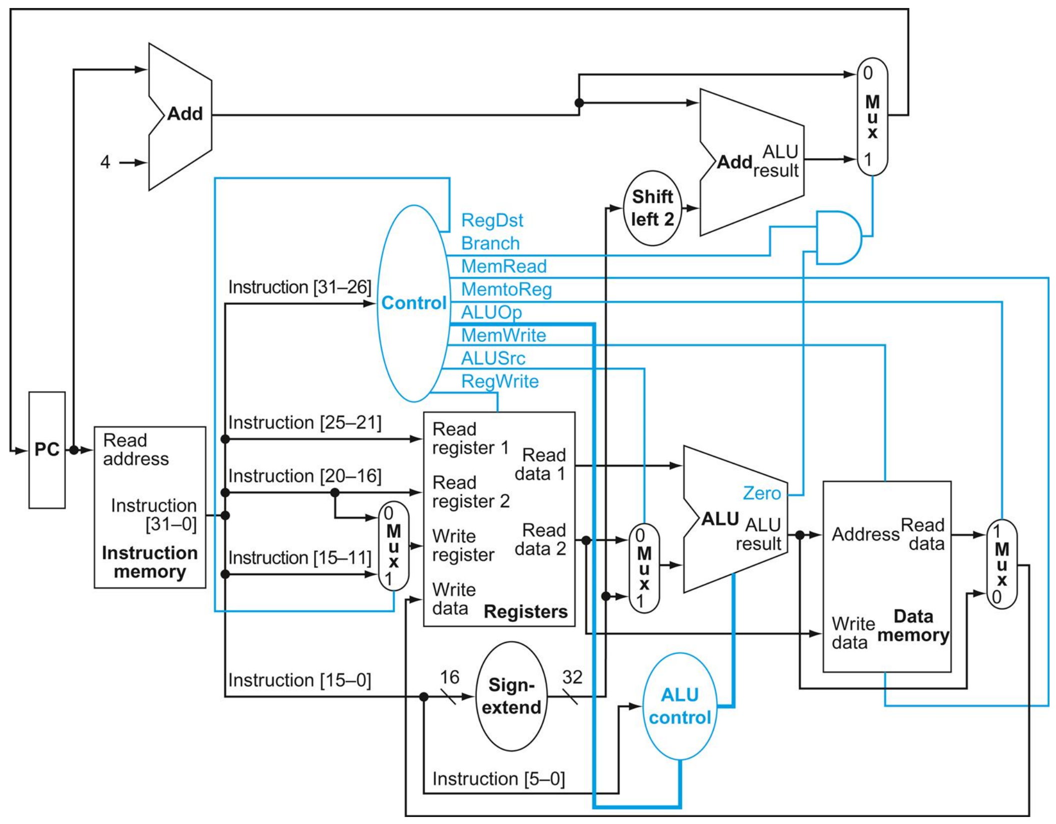


FIGURE 4.17 **The simple datapath with the control unit.**
The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures.

Before we try to write a set of equations or a truth table for the control unit, it will be useful to try to define the control function informally. Because the setting of the control lines depends only on the opcode, we define whether each control signal should be 0, 1, or don’t care (X) for each of the opcode values. Figure 4.18 defines how the control signals should be set for each opcode; this information follows directly from Figures 4.12, 4.16, and 4.17.

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

FIGURE 4.18 **The setting of the control lines is completely determined by the opcode fields of the instruction.**

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

Operation of the Datapath

With the information contained in Figures 4.16 and 4.18, we can design the control unit logic, but before we do that, let’s look at how each instruction uses the datapath. In the next few figures, we show the flow of three different instruction classes through the datapath. The asserted control signals and active datapath elements are highlighted in each of these. Note that a multiplexor whose control is 0 has a definite action, even if its control line is not highlighted. Multiple-bit control signals are highlighted if any constituent signal is asserted.

Figure 4.19 shows the operation of the datapath for an R-type instruction, such as `add $t1,$t2,$t3`. Although everything occurs in one clock cycle, we can think of four steps to execute the instruction; these steps are ordered by the flow of information:

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

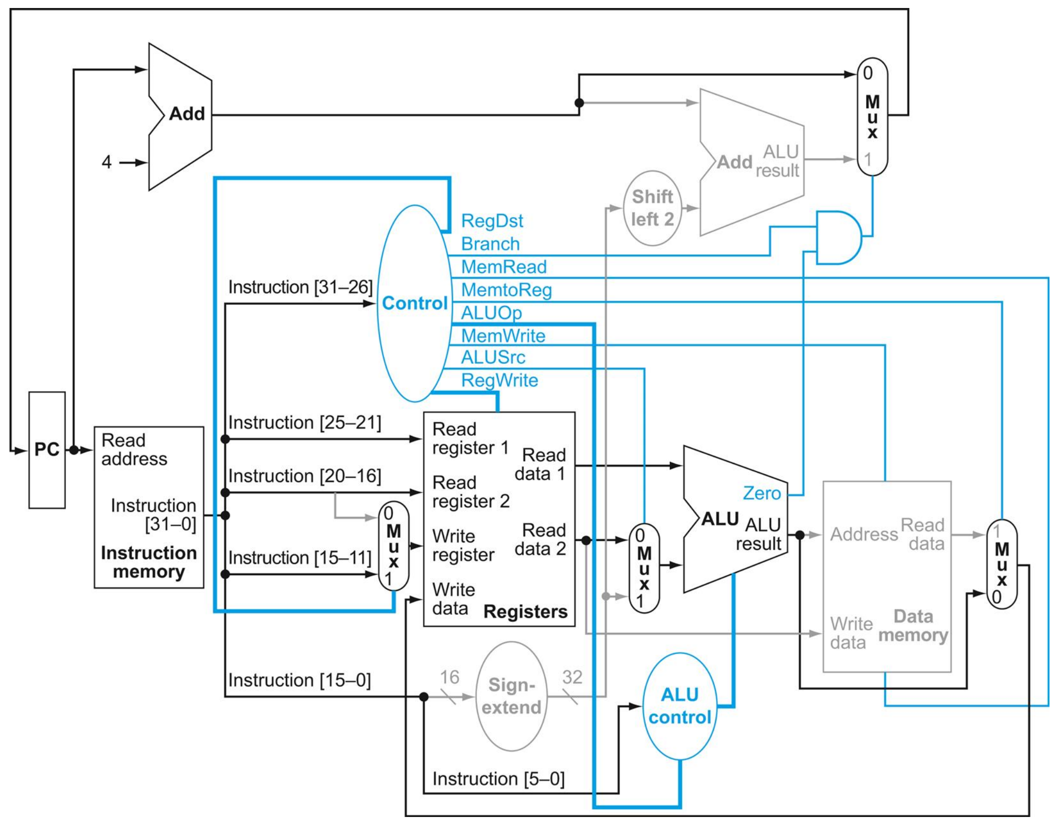


FIGURE 4.19 The datapath in operation for an R-type instruction, such as `add $t1,$t2,$t3`. The control lines, datapath units, and connections that are active are highlighted.

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

lw $t1, offset($t2)

in a style similar to Figure 4.19. Figure 4.20 shows the active functional units and asserted control lines for a load. We can think of a load instruction as operating in five steps (similar to how the R-type executed in four):

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

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

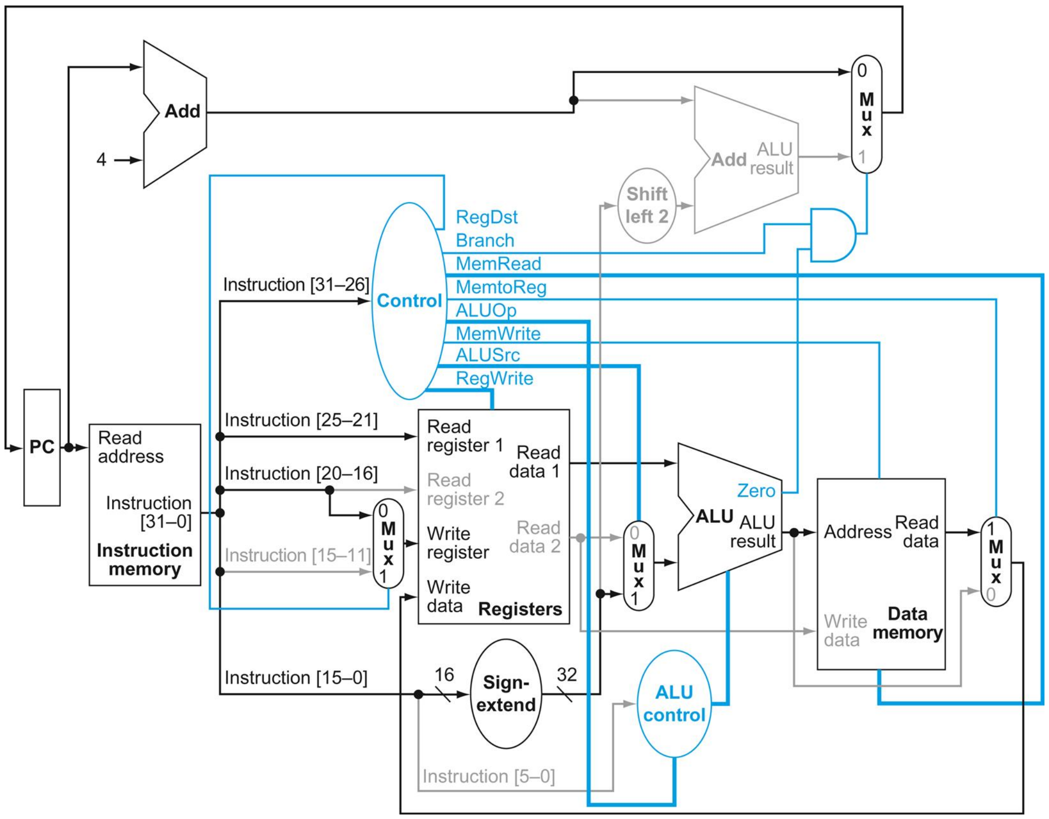


FIGURE 4.20 **The datapath in operation for a load instruction.** The control lines, datapath units, and connections that are active are highlighted. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occur.

Finally, we can show the operation of the branch-on-equal instruction, such as `beq $t1, $t2, offset`, in the same fashion. It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address. Figure 4.21 shows the four steps in execution:

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers, `$t1` and `$t2`, are read from the register file.
3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (`offset`) shifted left by two; the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

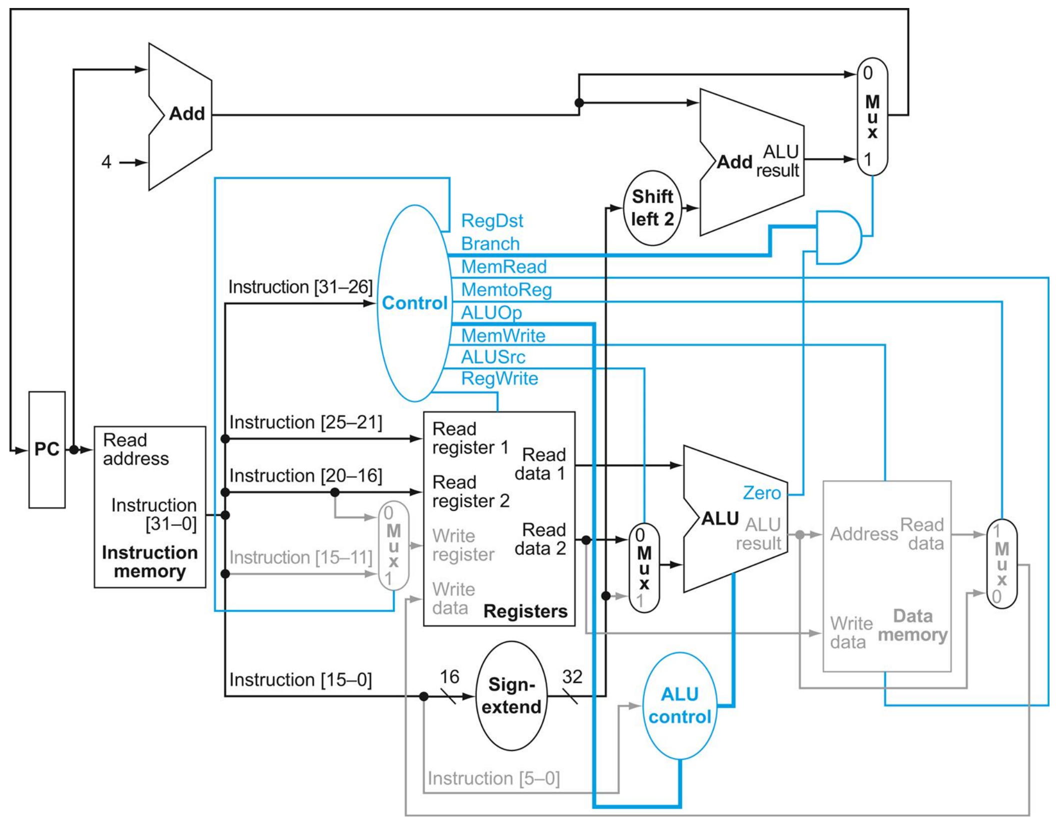


FIGURE 4.21 The datapath in operation for a branch-on-equal instruction. The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.

Finalizing Control

Now that we have seen how the instructions operate in steps, let's continue with the control implementation. The control function can be precisely defined using the contents of Figure 4.18. The outputs are the control lines, and the input is the 6-bit opcode field, Op [5:0]. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.

Figure 4.22 shows the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and we can implement it directly in gates in an automated fashion. We show this final step in Section D.2 in **Appendix D**.

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

FIGURE 4.22 **The control function for the simple single-cycle implementation is completely specified by this truth table.**
The top six rows of the table gives the combinations of input signals that correspond to the four opcodes, one per column, that determine the control output settings. (Remember that Op[5:0] corresponds to bits 31:26 of the instruction, which is the op field.) The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression $\overline{Op5} \cdot \overline{Op2}$, since this is sufficient to distinguish the R-format instructions from `lw`, `sw`, and `beq`. We do not take advantage of this simplification, since the rest of the MIPS opcodes are used in a full implementation.

Now that we have a **single-cycle implementation** of most of the MIPS core instruction set, let's add the jump instruction to show how the basic datapath and control can be extended to handle other instructions in the instruction set.

single-cycle implementation

Also called **single clock cycle implementation**. An implementation in which all instructions are executed in one clock cycle. While easy to understand, it is too slow to be practical.

Implementing Jumps

Example

Figure 4.17 shows the implementation of many of the instructions we looked at in Chapter 2. One class of instructions missing is that of the jump instruction. Extend the datapath and control of Figure 4.17 to include the jump instruction. Describe how to set any new control lines.

Answer

The jump instruction, shown in Figure 4.23, looks somewhat like a branch instruction but computes the target PC differently and is not conditional. Like a branch, the low-order 2 bits of a jump address are always 00_{two}. The next lower 26 bits of this 32-bit address come from the 26-bit immediate field in the instruction. The upper 4 bits of the address that should replace the PC come from the PC of the jump instruction plus 4. Thus, we can implement a jump by storing into the PC the concatenation of

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

Field	000010	address
Bit positions	31:26	25:0

FIGURE 4.23 Instruction format for the jump instruction (opcode = 2).
The destination address for a jump instruction is formed by concatenating the upper 4 bits of the current PC + 4 to the 26-bit address field in the jump instruction and adding 00 as the 2 low-order bits.

Figure 4.24 shows the addition of the control for jump added to Figure 4.17. An additional multiplexor is used to select the source for the new PC value, which is either the incremented PC (PC + 4), the branch target PC, or the jump target PC. One additional control signal is needed for the additional multiplexor. This control signal, called *Jump*, is asserted only when the instruction is a jump—that is, when the opcode is 2.

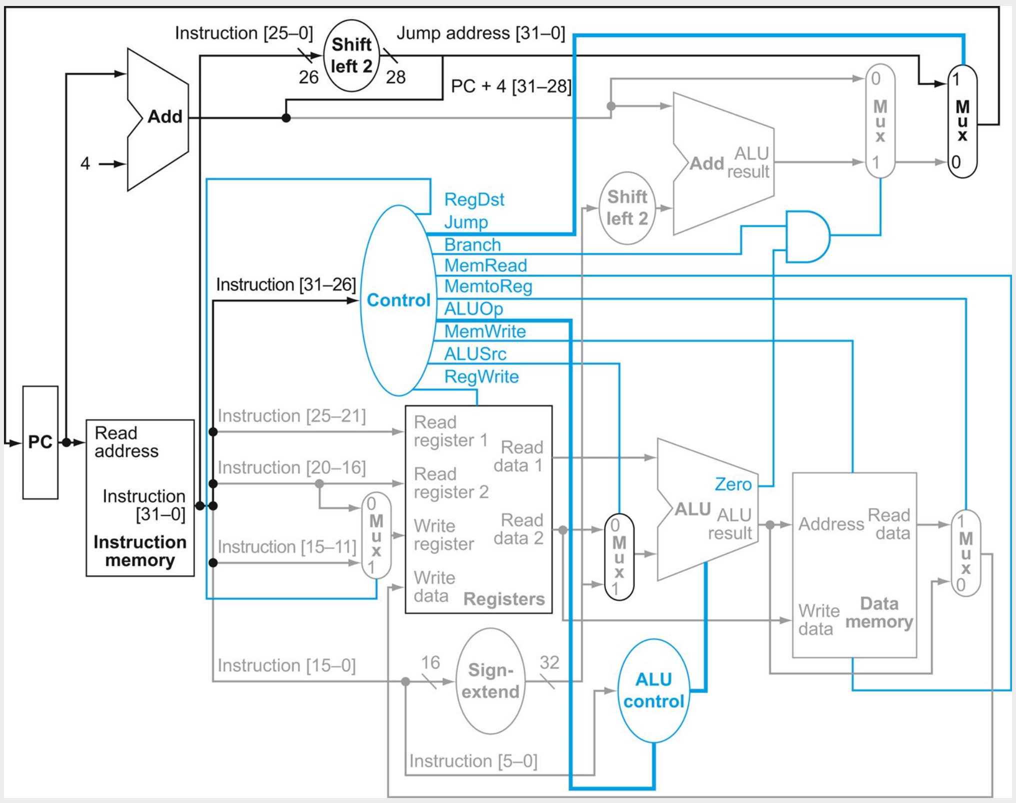


FIGURE 4.24 **The simple control and datapath are extended to handle the jump instruction.** An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

Why a Single-Cycle Implementation Is Not Used Today

Although the single-cycle design will work correctly, it would not be used in modern designs because it is inefficient. To see why this is so, notice that the clock cycle must have the same length for every instruction in this single-cycle design. Of course, the longest possible path in the processor determines the clock cycle. This path is almost certainly a load instruction, which uses five functional units in series: the instruction memory, the register file, the ALU, the data memory, and the register file. Although the CPI is 1 (see Chapter 1), the overall performance of a single-cycle implementation is likely to be poor, since the clock cycle is too long.

The penalty for using the single-cycle design with a fixed clock cycle is significant, but might be considered acceptable for this small instruction set. Historically, early 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.

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 section 4.6, we'll look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath but is much more efficient 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 A Multicycle Implementation

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

Although it could reduce hardware costs, almost all chips today use pipelining instead to increase performance over a single cycle implementation, so some readers may want to skip multicycle and go directly to pipelining. However, some instructors see pedagogic advantages to explaining multicycle implementation before pipelining, so we offer this implementation option online.

4.5 A Multicycle Implementation

In an earlier example, we broke each instruction into a series of steps corresponding to the functional unit operations that were needed. We can use these steps to create a **multicycle implementation**. In a multicycle implementation, each *step* in the execution will take 1 clock cycle. The multicycle implementation allows a functional unit to be used more than once per instruction, as long as it is used on different clock cycles. This sharing can help reduce the amount of hardware required. The ability to allow instructions to take different numbers of clock cycles and the ability to share functional units within the execution of a single instruction are the major advantages of a multicycle design. Figure e4.5.1 shows the abstract version of the multicycle datapath. If we compare Figure e4.5.1 to the datapath for the single-cycle version in Figure 4.15, we can see the following differences:

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

multicycle implementation

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

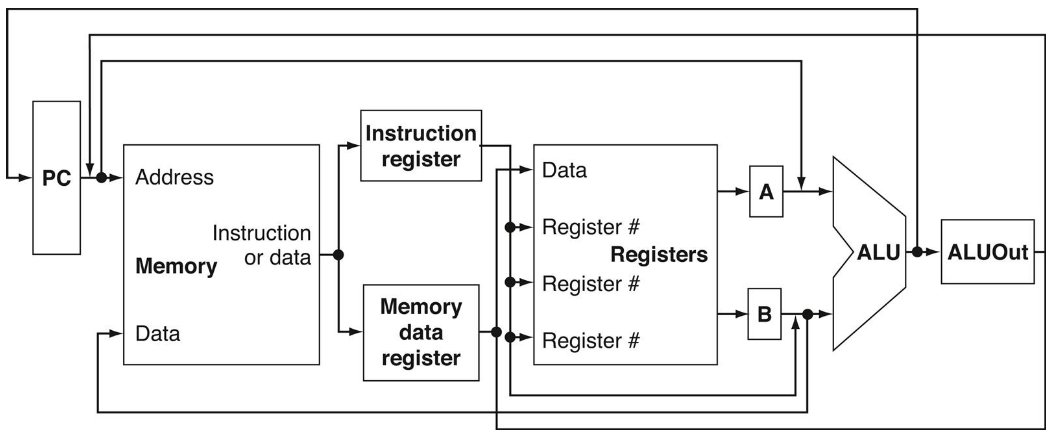


FIGURE E4.5.1 **The high-level view of the multicycle datapath.** This picture shows the key elements of the datapath: a shared memory unit, a single ALU shared among instructions, and the connections among these shared units. The use of shared functional units requires the addition or widening of multiplexors as well as new temporary registers that hold data between clock cycles of the same instruction. The additional registers are the Instruction register (IR), the Memory data register (MDR), A, B, and ALUOut.

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

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

The following temporary registers are added to meet these requirements:

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

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

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

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

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

Figure e4.5.2 shows the details of the datapath with these additional multiplexors. By introducing a few registers and multiplexors, we are able to reduce the number of memory units from two to one and eliminate two adders. Since registers and multiplexors are fairly small compared to a memory unit or ALU, this could yield a substantial reduction in the hardware cost.

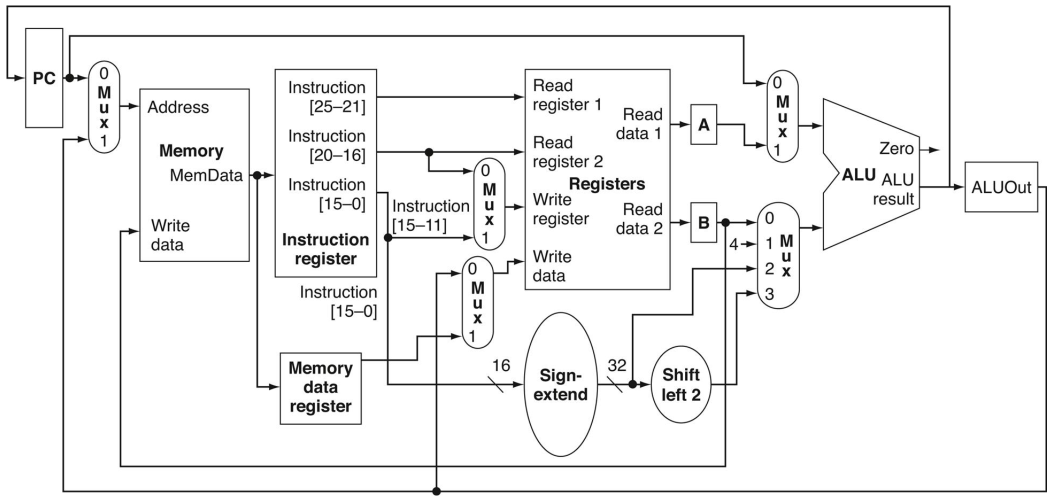


FIGURE E4.5.2 Multicycle datapath for MIPS handles the basic instructions. Although this datapath supports normal incrementing of the PC, a few more connections and a multiplexor will be needed for branches and jumps; we will add these shortly. The additions versus the single-clock datapath include several registers (IR, MDR, A, B, ALUOut), a multiplexor for the memory address, a multiplexor for the top ALU input, and expanding the multiplexor on the bottom ALU input into a four-way selector. These small additions allow us to remove two adders and a memory unit.

Because the datapath shown in Figure e4.5.2 takes multiple clock cycles per instruction, it will require a different set of control signals. The programmer-visible state units (the PC, the memory, and the registers) as well as the IR will need write control signals. The memory will also need a read signal. We can use the ALU control unit from the single-cycle datapath (see Figure 4.13 and Appendix D) to control the ALU here as well. Finally, each of the two-input multiplexors requires a single control line, while the four-input multiplexor requires two control lines. Figure e4.5.3 shows the datapath of Figure e4.5.2 with these control lines added.

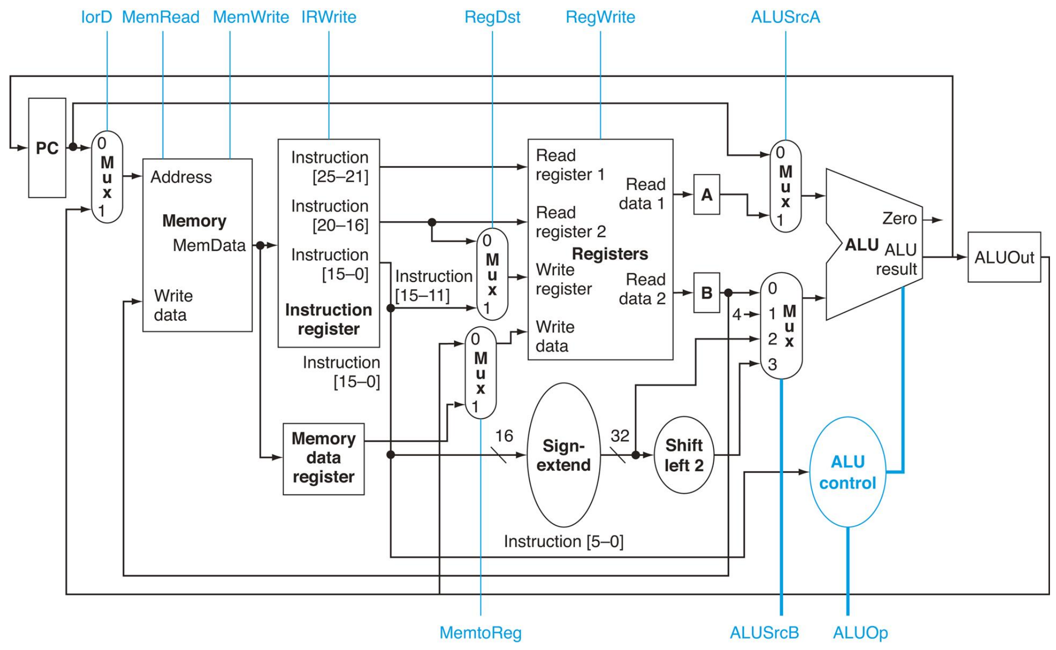


FIGURE E4.5.3 The multicycle datapath from Figure e4.5.2 with the control lines shown. The signals ALUOp and ALUSrcB are 2-bit control signals, while all the other control lines are 1-bit signals. Neither register A nor B requires a write signal, since their contents are only read on the cycle immediately after it is written. The memory data register has been added to hold the data from a load when the data returns from memory. Data from a load returning from memory cannot be written directly into the register file since the clock cycle cannot accommodate the time required for both the memory access and the register file write. The MemRead signal has been moved to the top of the memory unit to simplify the figures. The full set of datapaths and control lines for branches will be added shortly.

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

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

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

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

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

Figure e4.5.4 shows the complete multicycle datapath and control unit, including the additional control signals and multiplexor for implementing the PC updating.

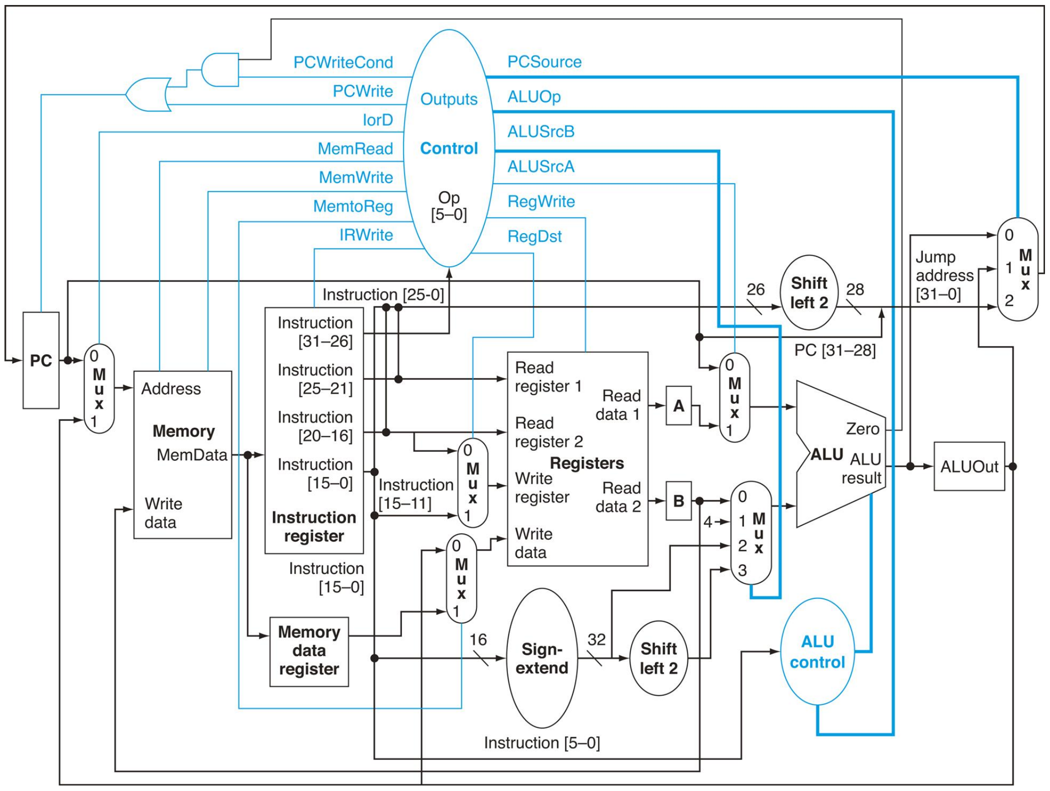


FIGURE E4.5.4 The complete datapath for the multicycle implementation together with the necessary control lines. The control lines of Figure e4.5.3 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure e4.5.3 include the multiplexor used to select the source of a new PC value; gates used to combine the PC write signals; and the control signals PCSource, PCWrite, and PCWriteCond. The PCWriteCond signal is used to decide whether a conditional branch should be taken. Support for jumps is included.

Before examining the steps to execute each instruction, let us informally examine the effect of all the control signals (just as we did for the single-cycle design in Figure 4.16 on page 264). Figure e4.5.5 shows what each control signal does when asserted and deasserted.

Elaboration

To reduce the number of signal lines interconnecting the functional units, designers can use *shared buses*. A shared bus is a set of lines that connect multiple units; in most cases, they include multiple sources that can place data on the bus and multiple readers of the value. Just as we reduced the number of functional units for the datapath, we can reduce the number of buses interconnecting these units by sharing the buses. For example, there are six sources coming to the ALU; however, only two of them are needed at any one time. Thus, a pair of buses can be used to hold values that are being sent to the ALU. Rather than placing a large multiplexor in front of the ALU, a designer can use a shared bus and then ensure that only one of the sources is driving the bus at any point. Although this saves signal lines, the same number of control lines will be needed to control what goes on the bus. The major drawback to using such bus structures is a potential performance penalty, since a bus is unlikely to be as fast as a point-to-point connection.

Actions of the 1-bit control signals

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

Actions of the 2-bit control signals

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

FIGURE E4.5.5 **The action caused by the setting of each control signal in Figure e4.5.4.** The top table describes the 1-bit control signals, while the bottom table describes the 2-bit signals. Only those control lines that affect multiplexors have an action when they are deasserted. This information is similar to that in Figure 4.16 on page 264 for the single-cycle datapath, but adds several new control lines (IRWrite, PCWrite, PCWriteCond, ALUSrcB, and PCSource) and removes control lines that are no longer used or have been replaced (PCSrc, Branch, and Jump).

Breaking the Instruction Execution into Clock Cycles

Given the datapath in Figure e4.5.4, we now need to look at what should happen in each clock cycle of the multicycle execution, since this will determine what additional control signals may be needed, as well as the setting of the control signals. Our goal in breaking the execution into clock cycles should be to maximize performance. We can begin by breaking the execution of any instruction into a series of steps, each taking one clock cycle, attempting to keep the amount of work per cycle roughly equal. For example, we will restrict each step to contain at most one ALU operation, or one register file access, or one memory access. With this restriction, the clock cycle could be as short as the longest of these operations.

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

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

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

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

1 Instruction fetch step

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

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

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

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

2 Instruction decode and register fetch step

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

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

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

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

Operation: Access the register file to read registers rs and rt and store the results into registers A and B. Since A and B are overwritten on every cycle, the register file can be read on every cycle with the values stored into A and B. This step also computes the branch target address and stores the address in ALUOut, where it will be used on the next clock cycle if the instruction is a branch. This requires setting ALUSrcA to 0 (so that the PC is sent to the ALU), ALUSrcB to the value 11 (so that the sign-extended and shifted offset field is sent to the ALU), and ALUOp to 00 (so the ALU adds). The register file accesses and computation of branch target occur in parallel.

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

3 Execution, memory address computation, or branch completion

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

Memory reference:

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

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

Arithmetic-logical instruction (R-type):

ALUOut <= A op B;

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

Branch:

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

Operation: The ALU is used to do the equal comparison between the two registers read in the previous step. The Zero signal out of the ALU is used to determine whether or not to branch. This requires setting ALUSrcA=1 and setting ALUSrcB=00 (so that the register file outputs are the ALU inputs). The ALUOp signals will need to be set to 01 (causing the ALU to subtract) for equality testing. The PCWriteCond signal will need to be asserted to update the PC if the Zero output of the ALU is asserted. By setting PCSource to 01, the value written into the PC will come from ALUOut, which holds the branch target address computed in the previous cycle. For conditional branches that are taken, we actually write the PC twice: once from the output of the ALU (during the Instruction decode/register fetch) and once from ALUOut (during the Branch completion step). The value written into the PC last is the one used for the next instruction fetch.

Jump:

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

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

4 Memory access or R-type instruction completion step

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

Memory reference:

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

or

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

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

Arithmetic-logical instruction (R-type):

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

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

5 Memory read completion step

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

Load:

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

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

This five-step sequence is summarized in Figure e4.5.6. From this sequence we can determine what the control must do on each clock cycle.

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

FIGURE E4.5.6 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

Defining the Control

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

Online Section 4.14 shows how hardware design languages are used to design modern processors with examples of both the multicycle datapath and the finite- state control. In modern digital systems design, the final step of taking a hardware description to actual gates is handled by logic and datapath synthesis tools. Appendix D shows how this process operates by translating the multicycle control unit to a detailed hard- ware implementation. The key ideas of control can be grasped from this chapter without examining the material on hardware description languages or Appendix D. However, if you want to actually do some hardware design, Appendix C can show you what the implementations are likely to look like at the gate level.

CPI in a Multicycle CPU

Example

Using the SPECINT2006 instruction mix shown in Figure 3.24, what is the CPI, assuming that each state in the multicycle CPU requires 1 clock cycle?

ANSWER

The mix is 20% loads, 8% stores 10% branches, and 62% ALU (all the rest of the mix, which we assume to be ALU instructions). From Figure 4.5.6, the number of clock cycles for each instruction class is the following:

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

The CPI is given by the following:

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

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

The ratio

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

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

$$\text{CPI} = 0.20 \times 5 + 0.08 \times 4 + 0.62 \times 4 + 0.10 \times 3 = 4.10$$

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

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

finite-state machine

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

next-state function

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

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

The finite-state control essentially corresponds to the five steps of execution shown on pages 4.5–15 to 4.5–20; each state in the finite-state machine will take 1 clock cycle. The finite-state machine will consist of several parts. Since the first two steps of execution are identical for every instruction, the initial two states of the finite-state machine will be common for all instructions. Steps 3 through 5 differ, depending on the opcode. After the execution of the last step for a particular instruction class, the finite-state machine will return to the initial state to begin fetching the next instruction.

Figure e4.5.7 shows this abstracted representation of the finite-state machine. To fill in the details of the finite-state machine, we will first expand the instruction fetch and decode portion, and then we will show the states (and actions) for the different instruction classes.

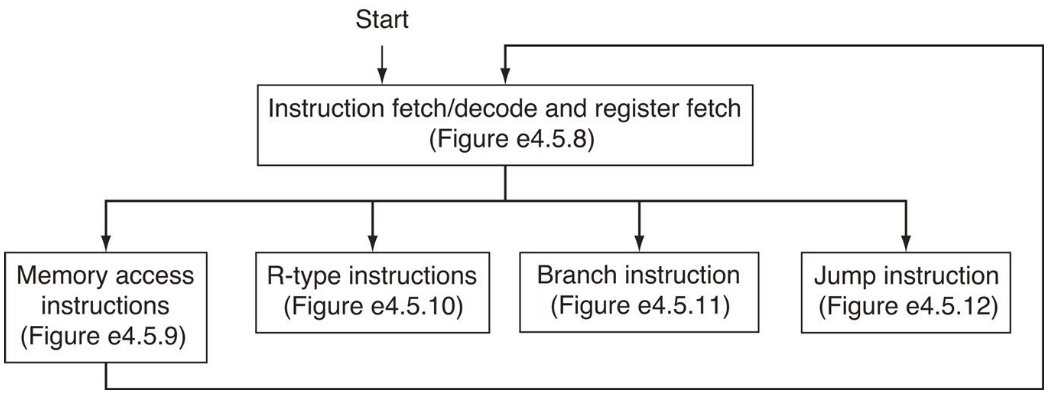

Start
Instruction fetch/decode and register fetch (Figure e4.5.8)
Memory access instructions (Figure e4.5.9)
R-type instructions (Figure e4.5.10)
Branch instruction (Figure e4.5.11)
Jump instruction (Figure e4.5.12)

FIGURE E4.5.7 **The high-level view of the finite-state machine control.** The first steps are independent of the instruction class; then a series of sequences that depend on the instruction opcode are used to complete each instruction class. After completing the actions needed for that instruction class, the control returns to fetch a new instruction. Each box in this figure may represent one to several states. The arc labeled Start marks the state in which to begin when the first instruction is to be fetched.

We show the first two states of the finite-state machine in Figure e4.5.8 using a traditional graphic representation. We number the states to simplify the explanation, though the numbers are arbitrary. State 0, corresponding to step 1, is the starting state of the machine.

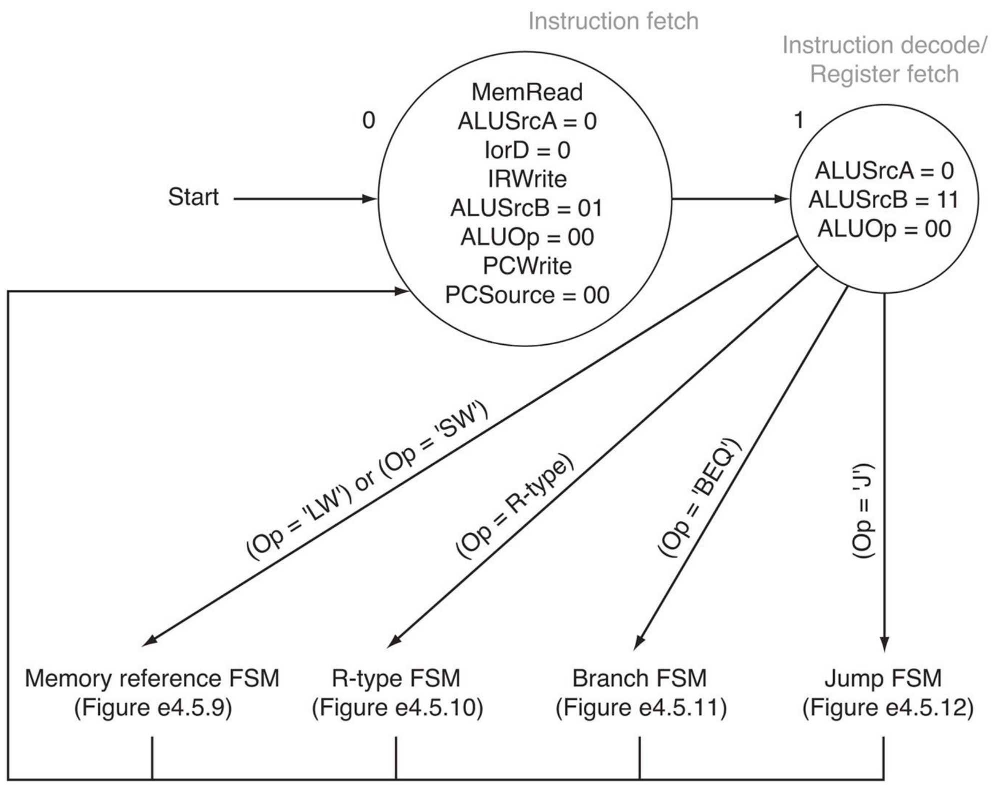

Instruction fetch
Instruction decode/ Register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
Start
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory reference FSM (Figure e4.5.9)
R-type FSM (Figure e4.5.10)
Branch FSM (Figure e4.5.11)
Jump FSM (Figure e4.5.12)

FIGURE E4.5.8 **The instruction fetch and decode portion of every instruction is identical.** These states correspond to the top box in the abstract finite-state machine in Figure e4.5.7. In the first state we assert two signals to cause the memory to read an instruction and write it into the Instruction register (MemRead and IRWrite), and we set IorD to 0 to choose the PC as the address source. The signals ALUSrcA, ALUSrcB, ALUOp, PCWrite, and PCSource are set to compute PC + 4 and store it into the PC. (It will also be stored into ALUOut, but never used from there.) In the next state, we compute the branch target address by setting ALUSrcB to 11 (causing the shifted and sign-extended lower 16 bits of the IR to be sent to the ALU), setting ALUSrcA to 0 and ALUOp to 00; we store the result in the ALUOut register, which is written on every cycle. There are four next states that depend on the class of

the instruction, which is known during this state. The control unit input, called Op, is used to determine which of these arcs to follow. Remember that all signals not explicitly asserted are deasserted; this is particularly important for signals that control writes. For multiplexor controls, lack of a specific setting indicates that we do not care about the setting of the multiplexor.

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

Figure e4.5.9 shows the portion of the finite-state machine needed to implement the memory reference instructions. For the memory reference instructions, the first state after fetching the instruction and registers computes the memory address (state 2). To compute the memory address, the ALU input multiplexors must be set so that the first input is the A register, while the second input is the sign-extended displacement field; the result is written into the ALUOut register. After the memory address calculation, the memory should be read or written; this requires two different states. If the instruction opcode is lw, then state 3 (corresponding to the step Memory access) does the memory read (MemRead is asserted). The output of the memory is always written into MDR. If it is sw, state 5 does a memory write (MemWrite is asserted). In states 3 and 5, the signal IorD is set to 1 to force the memory address to come from the ALU. After performing a write, the instruction sw has completed execution, and the next state is state 0. If the instruction is a load, however, another state (state 4) is needed to write the result from the memory into the register file. Setting the multiplexor controls MemtoReg=1 and RegDst=0 will send the loaded value in the MDR to be written into the register file, using rt as the register number. After this state, corresponding to the Memory read completion step, the next state is state 0.

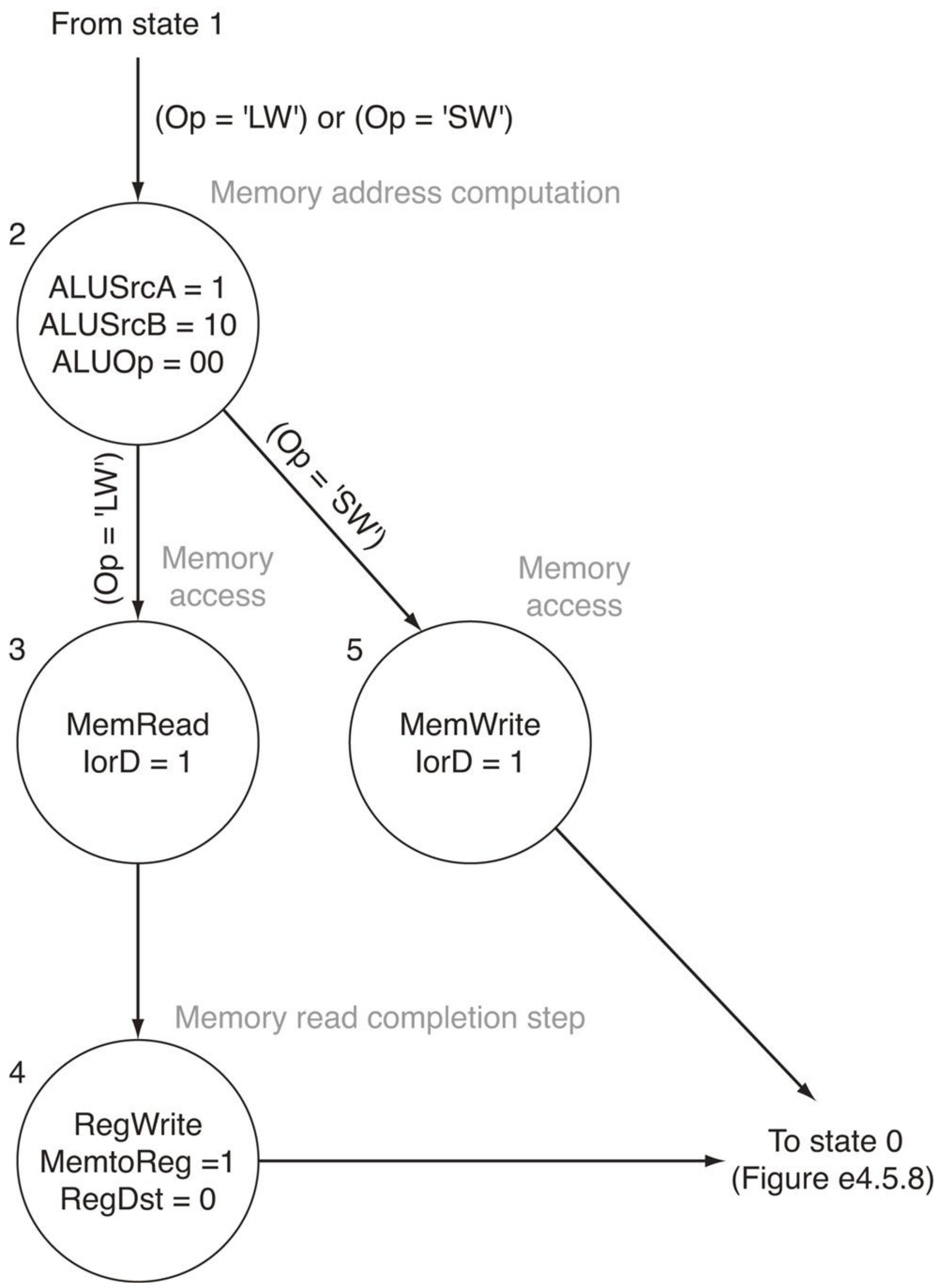


FIGURE E4.5.9 **The finite-state machine for controlling memory reference instructions has four states.**
These states correspond to the box labeled "Memory access instructions" in Figure e4.5.7.

After performing a memory address calculation, a separate sequence is needed for load and for store. The setting of the control signals ALUSrcA, ALUSrcB, and ALUOp is used to cause the memory address computation in state 2. Loads require an extra state to write the result from the MDR (where the result is written in state 3) into the register file.

To implement the R-type instructions requires two states corresponding to steps 3 (Execute) and 4 (R-type completion). Figure e4.5.10 shows this two-state portion of the finite-state machine. State 6 asserts ALUSrcA and sets the ALUSrcB signals to 00; this forces the two registers that were read from the register file to be used as inputs to the ALU. Setting ALUOp to 10 causes the ALU control unit to use the function field to set the ALU control signals. In state 7, RegWrite is asserted to cause the register file to write, RegDst is asserted to cause the rd field to be used as the register number of the destination, and MemtoReg is deasserted to select ALUOut as the source of the value to write into the register file.

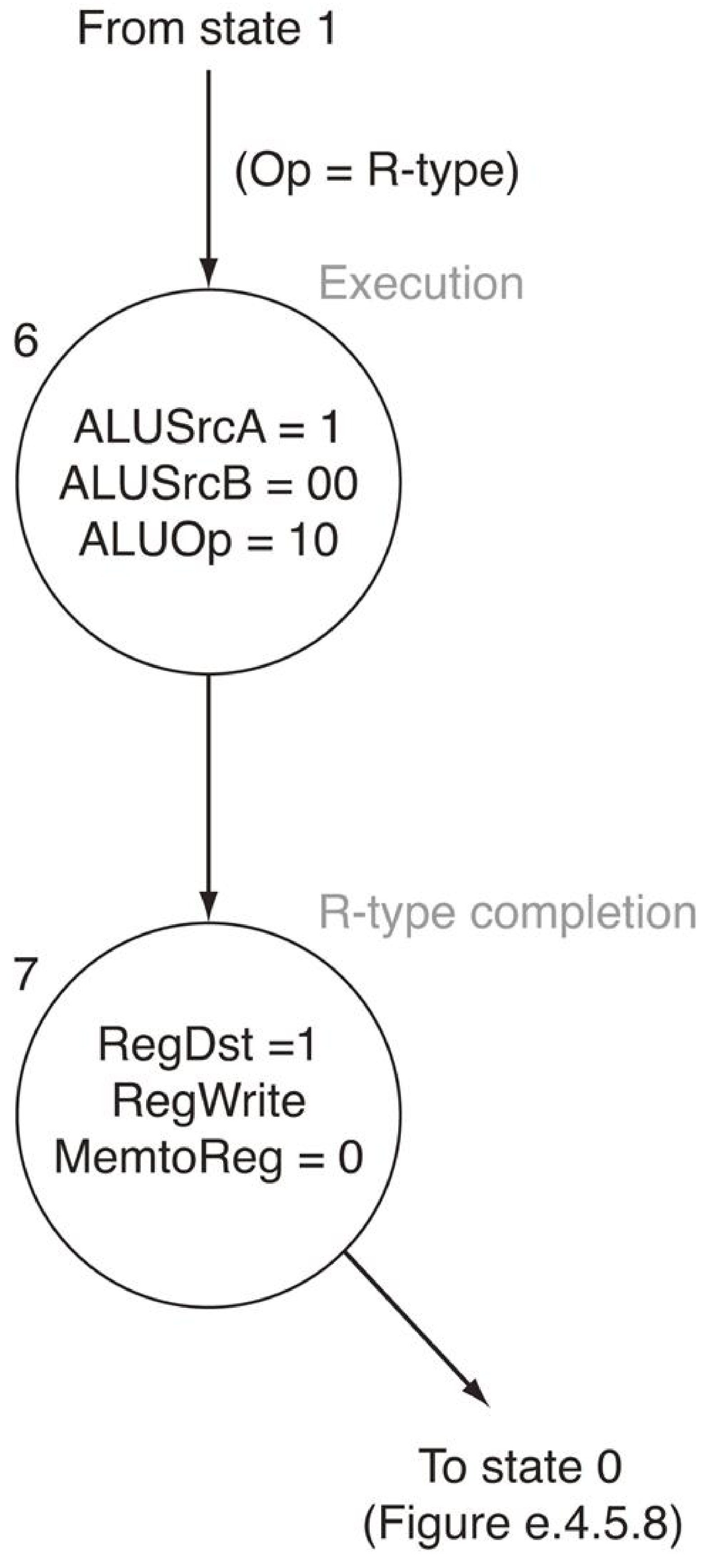

From state 1
(Op = R-type)
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
R-type completion
7
RegDst =1
RegWrite
MemtoReg = 0
To state 0
(Figure e.4.5.8)

FIGURE E4.5.10 **R-type instructions can be implemented with a simple two-state finite-state machine.**
These states correspond to the box labeled “R-type instructions” in Figure e4.5.7. The first state causes the ALU operation to occur, while the second state causes the ALU result (which is in ALUOut) to be written in the register file. The three signals asserted during state 7 cause the contents of ALUOut to be written into the register file in the entry specified by the rd field of the Instruction register.

For branches, only a single additional state is necessary because they complete execution during the third step of instruction execution. During this state, the control signals that cause the ALU to compare the contents of registers A and B must be set, and the signals that cause the PC to be written conditionally with the address in the ALUOut register are also set. To perform the comparison requires that we assert ALUSrcA and set ALUSrcB to 00, and set the ALUOp value to 01 (forcing a subtract). (We use only the Zero output of the ALU, not the result of the subtraction.) To control the writing of the PC, we assert PCWriteCond and set PCSource=01, which will cause the value in the ALUOut register (containing the branch address calculated in state 1, Figure e4.5.8) to be written into the PC if the Zero bit out of the ALU is asserted. Figure e4.5.11 shows this single state.

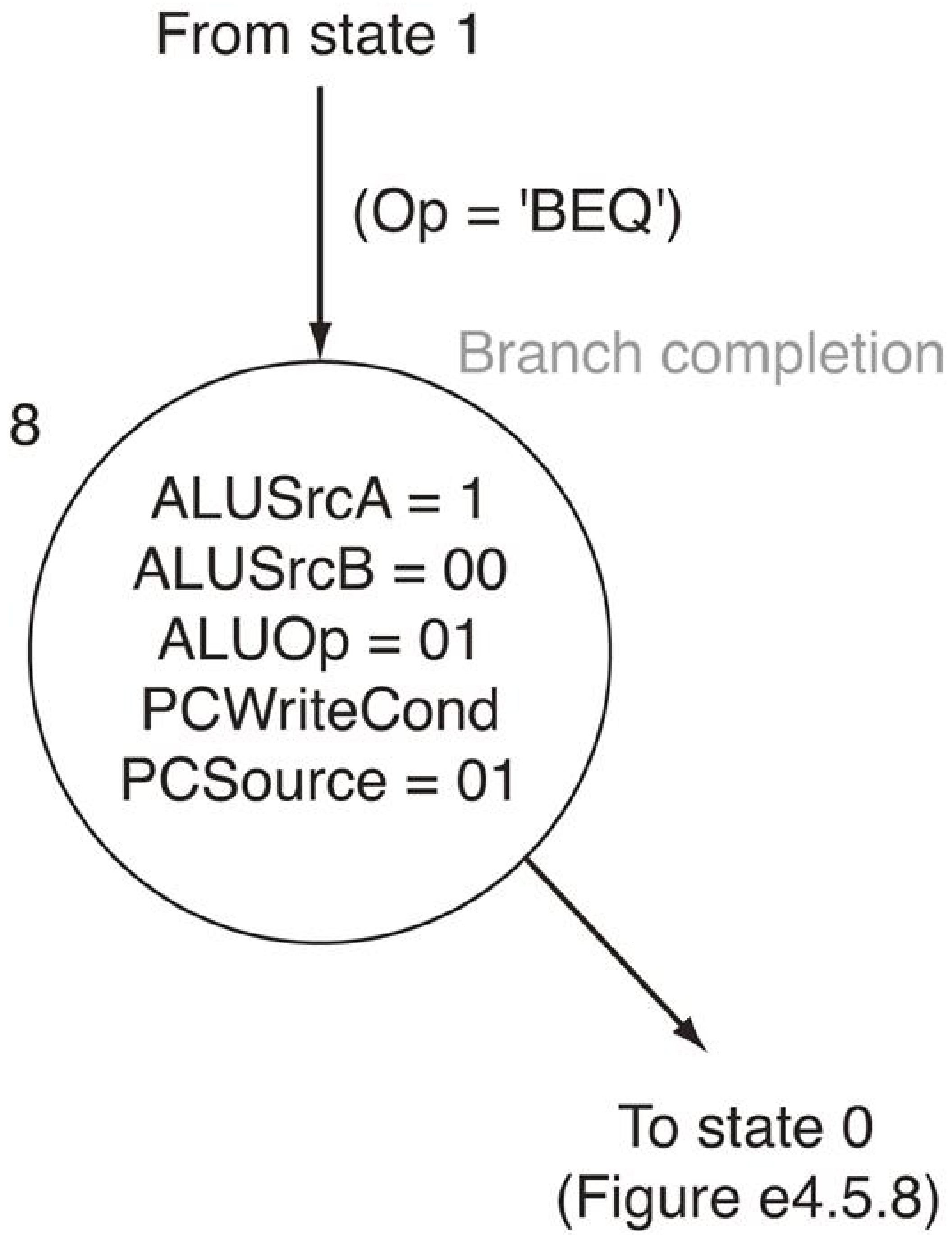


FIGURE E4.5.11 **The branch instruction requires a single state.**
The first three outputs that are asserted cause the ALU to compare the registers (ALUSrcA, ALUSrcB, and ALUOp), while the signals PCSource and PCWriteCond perform the

conditional write if the branch condition is true. Notice that we do not use the value written into ALUOut; instead, we use only the Zero output of the ALU. The branch target address is read from ALUOut, where it was saved at the end of state 1.

The last instruction class is jump; like branch, it requires only a single state (shown in Figure e4.5.12) to complete its execution. In this state, the signal PCWrite is asserted to cause the PC to be written. By setting PCSource to 10, the value supplied for writing will be the lower 26 bits of the Instruction register with 00_{two} added as the low-order bits concatenated with the upper 4 bits of the PC.

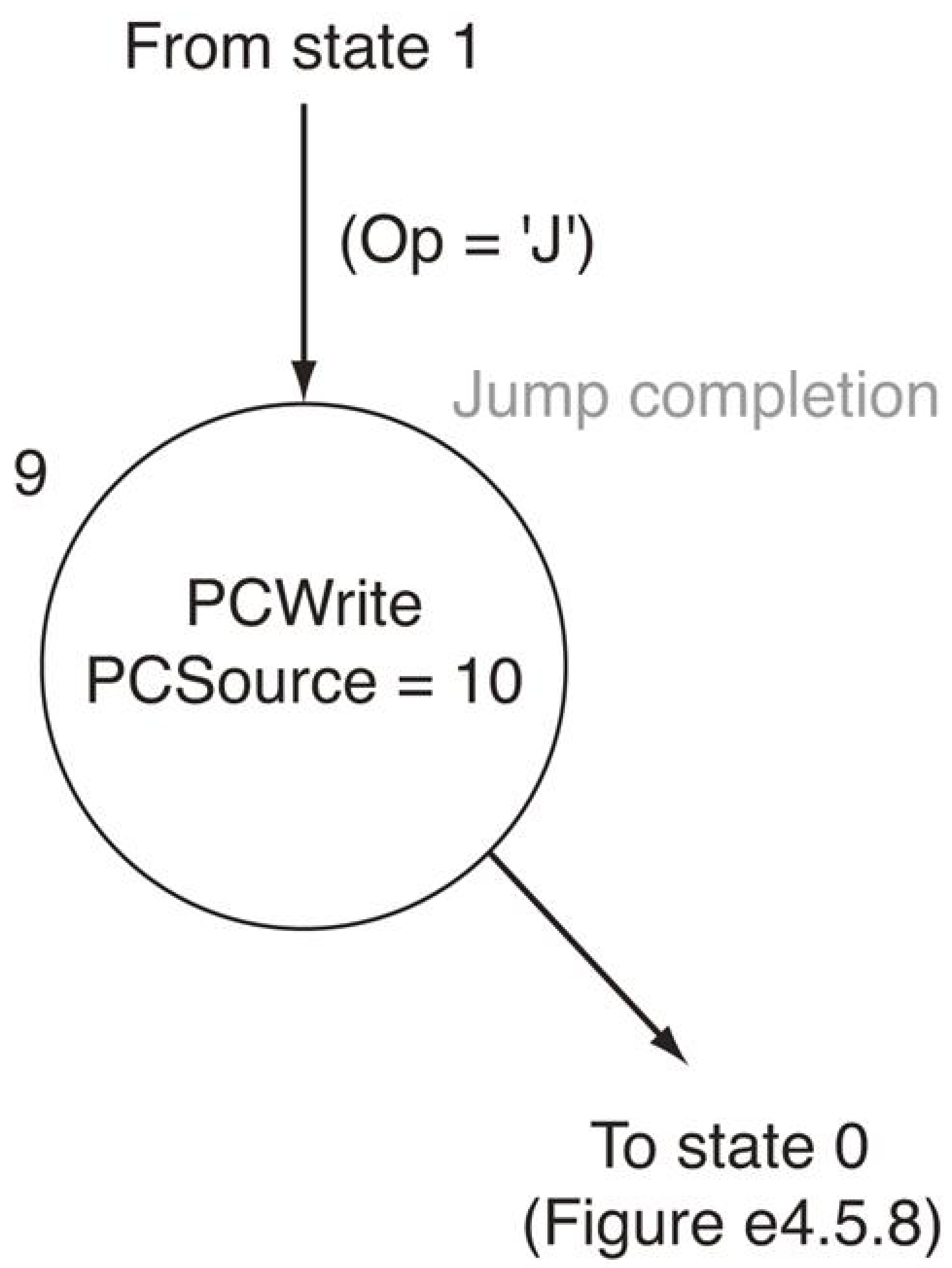


FIGURE E4.5.12 **The jump instruction requires a single state.**
that asserts two control signals to write the PC with the lower 26 bits of the Instruction register shifted left 2 bits and concatenated to the upper 4 bits of the PC of this instruction.

We can now put these pieces of the finite-state machine together to form a specification for the control unit, as shown in Figure e4.5.13. In each state, the signals that are asserted are shown. The next state depends on the opcode bits of the instruction, so we label the arcs with a comparison for the corresponding instruction opcodes.

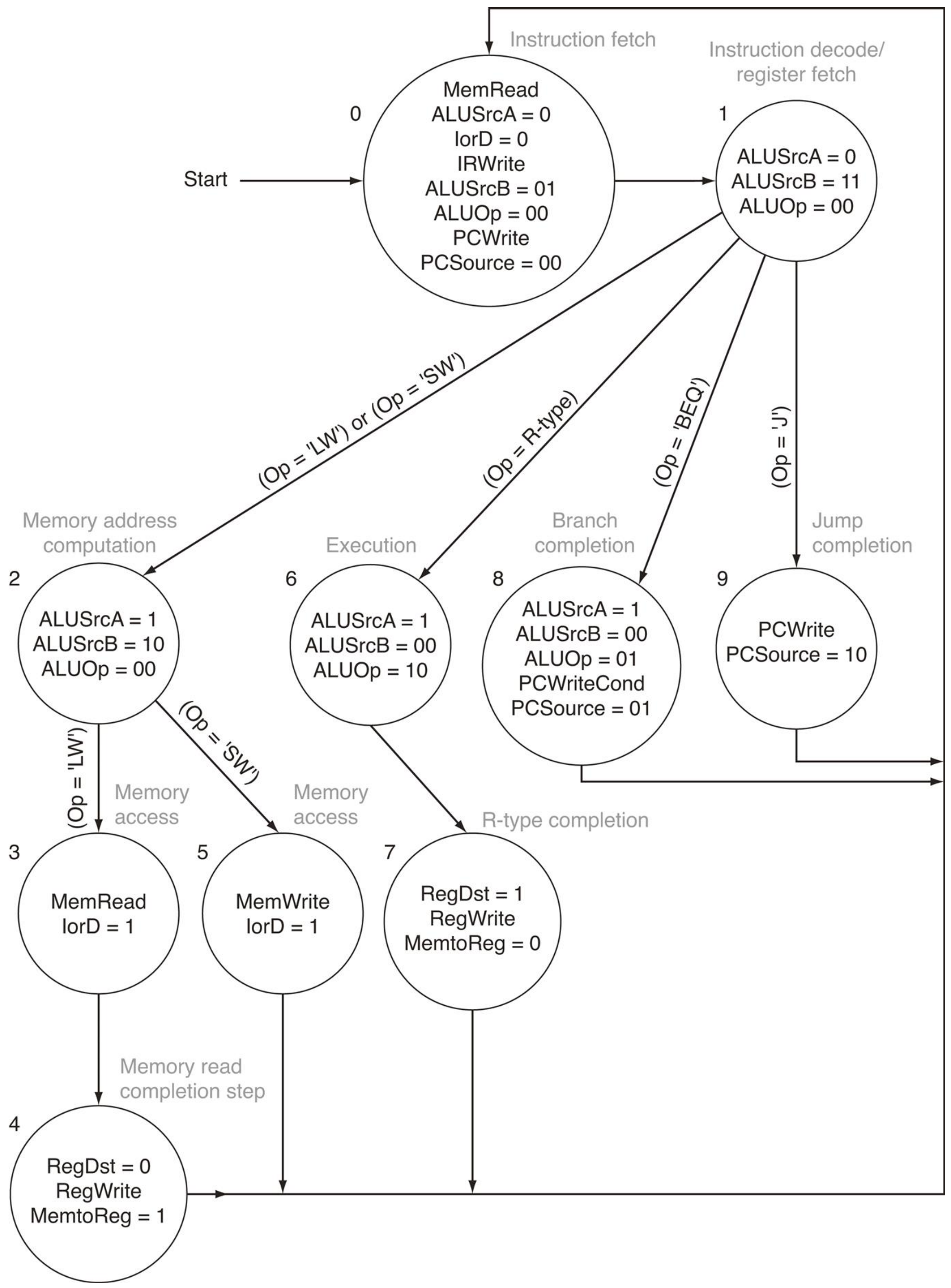


FIGURE E4.5.13 **The complete finite-state machine control for the datapath shown in Figure e4.5.4.**
The labels on the arcs are conditions that are tested to determine which state is the next state; when the next state is unconditional, no label is

given. The labels inside the nodes indicate the output signals asserted during that state; we always specify the setting of a multiplexor control signal if the correct operation requires it. Hence, in some states a multiplexor control will be set to 0.

A finite-state machine can be implemented with a temporary register that holds the current state and a block of combinational logic that determines both the datapath signals to be asserted and the next state. Figure e4.5.14 shows how such an implementation might look. Appendix D describes in detail how the finite-state machine is implemented using this structure. In Section D.3, the combinational control logic for the finite-state machine of Figure e4.5.13 implemented both with a ROM (read-only memory) and a PLA (programmable logic array). (Also see Appendix B for a description of these logic elements.) In the next section of this chapter, we consider another way to represent control. Both of these techniques are simply different representations of the same control information.

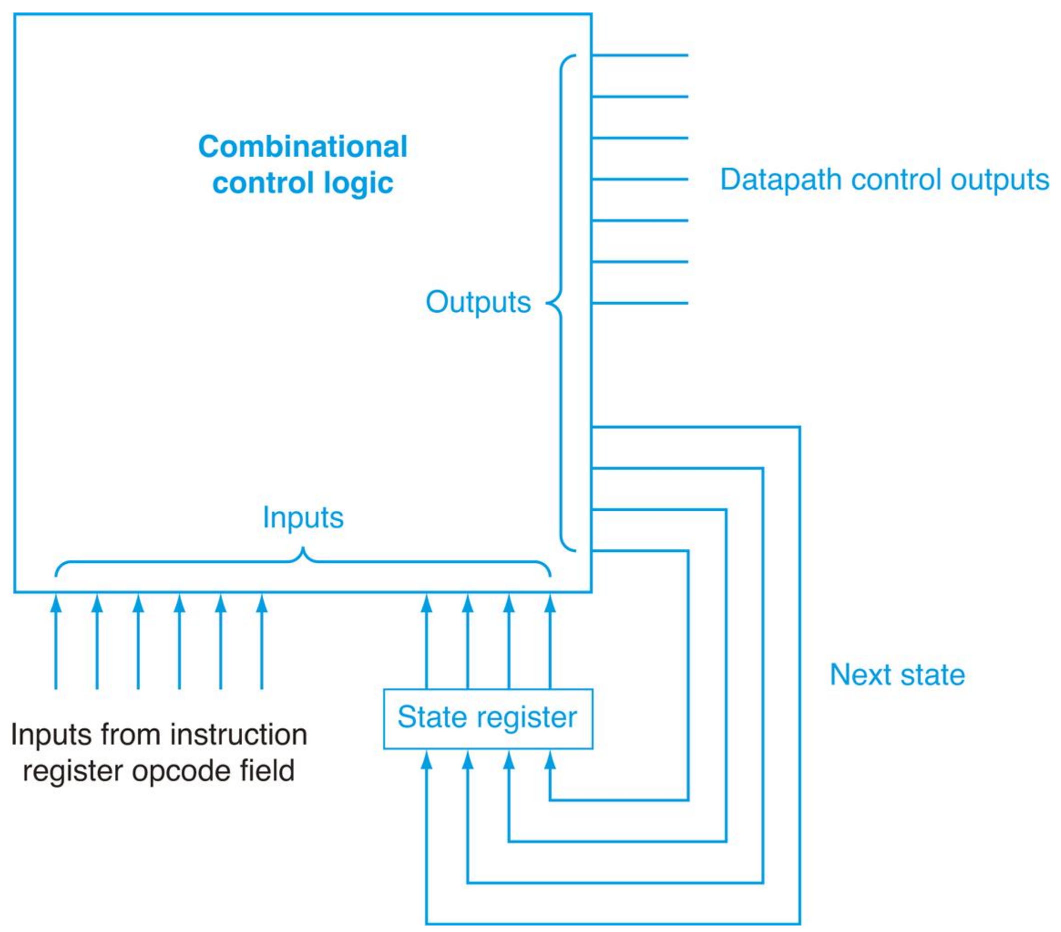

Combinational control logic
Datapath control outputs
Outputs
Inputs
Next state
State register
Inputs from instruction register opcode field

FIGURE E4.5.14 **Finite-state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state.**
The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational logic are the current state and any inputs used to determine the next state. In this case, the inputs are the instruction register opcode bits. Notice that in the finite-state machine used in this chapter, the outputs depend only on the current state, not on the inputs. The elaboration below explains this in more detail.

Pipelining, which is the subject of the next section is almost always used to accelerate the execution of instructions. For simple instructions, pipelining is capable of achieving the higher clock rate of a multicycle design and a single- cycle CPI of a single-clock design. In most pipelined processors, however, some instructions take longer than a single cycle and require multicycle control. Floating- point are one universal example. There are many examples in the IA-32 architecture that require the use of multicycle control.

Elaboration

The style of finite-state machine in Figure e4.5.14 is called a Moore machine, after Edward Moore. Its identifying characteristic is that the output depends only on the current state. For a Moore machine, the box labeled Combinational control logic can be split into two pieces. One piece has the control output and only the state input, while the other has only the next-state output.

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

Understanding Program Performance

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

Check Yourself

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

4.6 An Overview of Pipelining

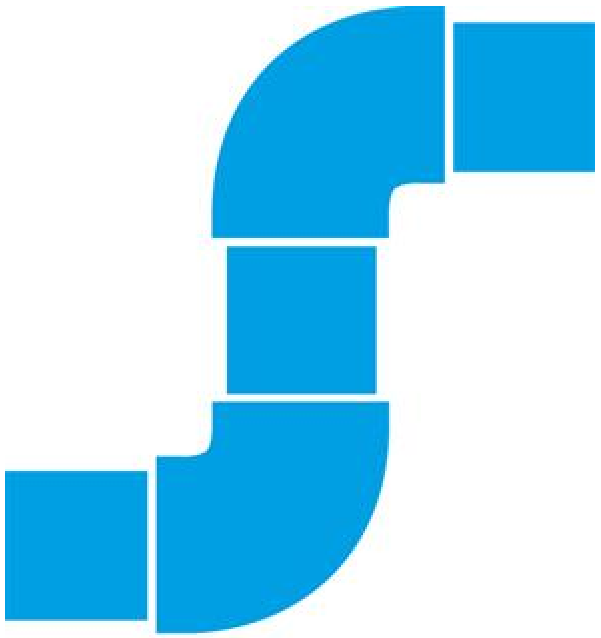
Never waste time.

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.



PIPELINING

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-A8. If you are interested in 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 *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 this load 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.

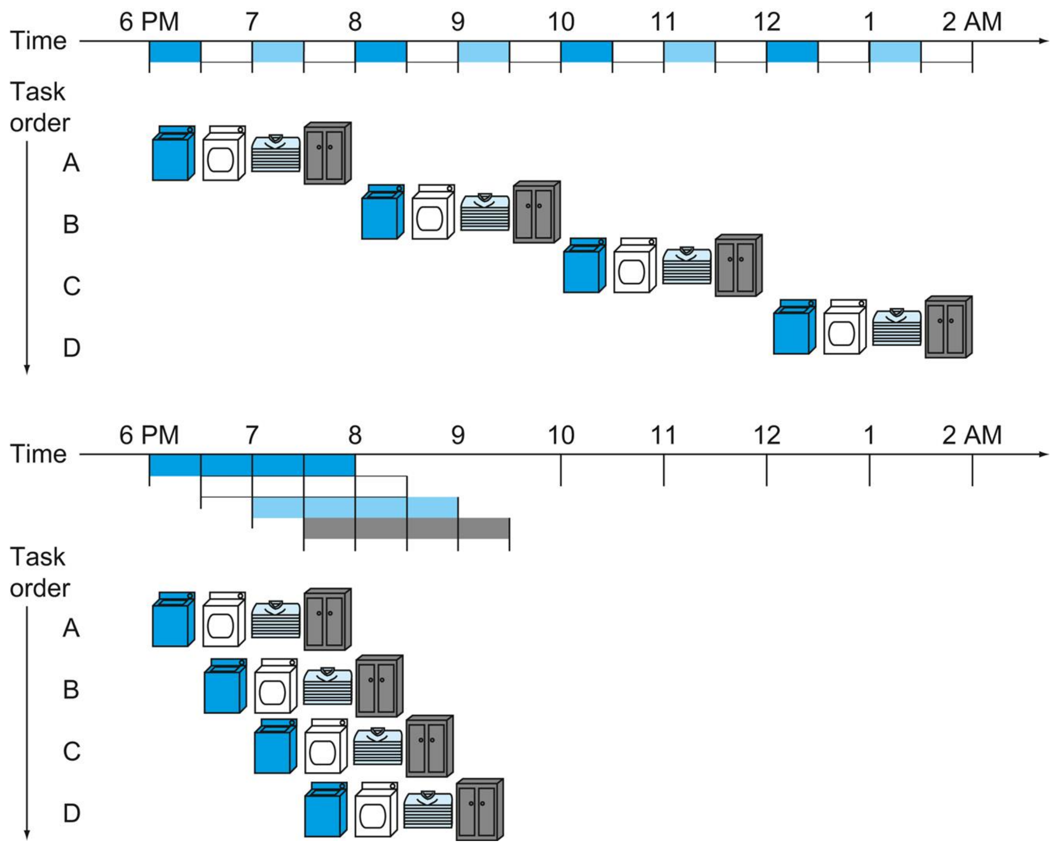


FIGURE 4.25 **The laundry analogy for pipelining.** Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, “folder,” and “storer” each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry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

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 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.

Single-Cycle versus Pipelined Performance

Example

To make this discussion concrete, let's create a pipeline. In this example, and in the rest of this chapter, we limit our attention to eight instructions:

load word (`lw`), store word (`sw`), add (`add`), subtract (`sub`), AND (`and`), OR (`or`), set less than (`slt`), and branch on equal (`beq`).

Compare the average time between instructions of a single-cycle implementation, in which all instructions take 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.

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.

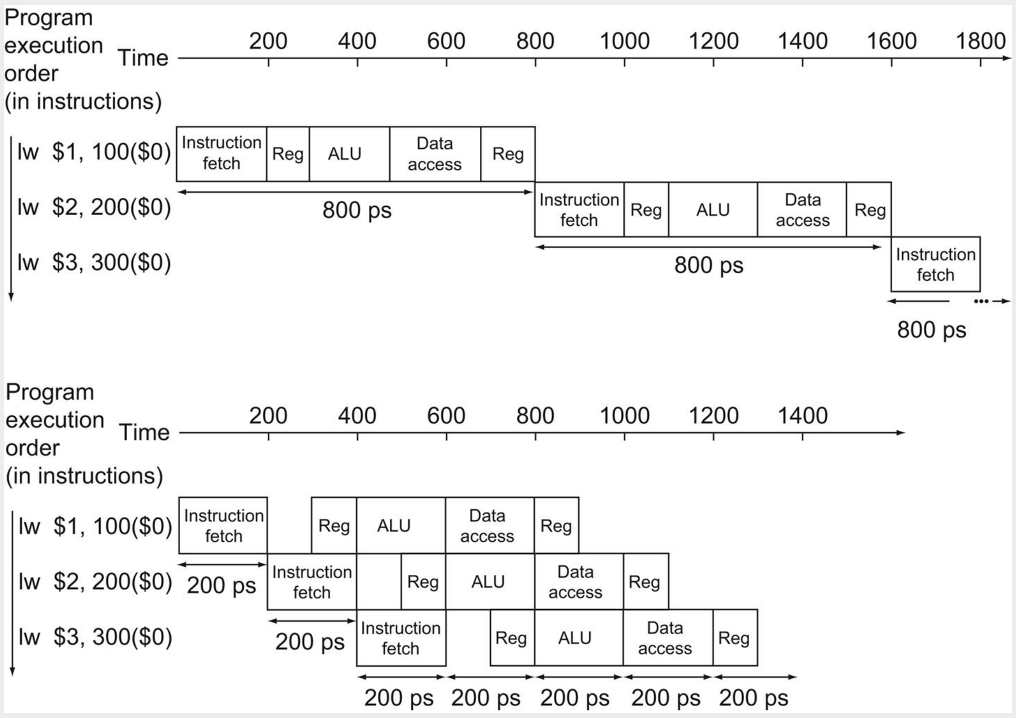


FIGURE 4.27 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom.

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.

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.

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:

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

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. Modern 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 4.11.)

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.

Structural Hazard

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 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.

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.

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.

Although we could try to rely on compilers to remove all such hazards, the results would not be satisfactory. These dependences happen just too often and the delay is just too long to expect the compiler to rescue us from this dilemma.

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

forwarding

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.

Forwarding with Two Instructions

Example

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

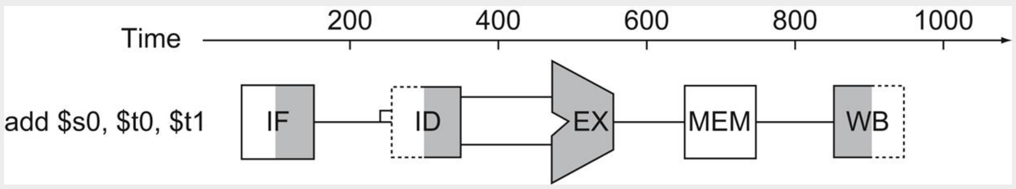


FIGURE 4.28 **Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.25.**
Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the 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.

Answer

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.

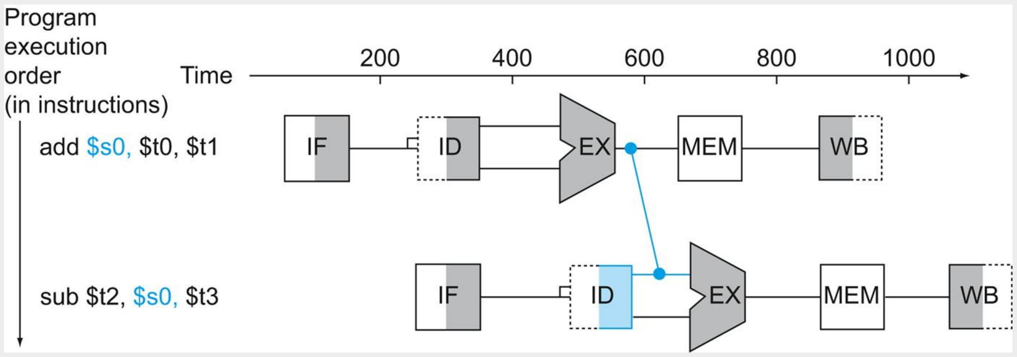


FIGURE 4.29 Graphical representation of forwarding.
The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `$s0` read in the second stage of `sub`.

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 `$s0` instead of an add. As we can imagine from looking at Figure 4.29, 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 4.30 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 4.8 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 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.

Reordering Code to Avoid Pipeline Stalls

Example

Consider the following code segment in C:

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

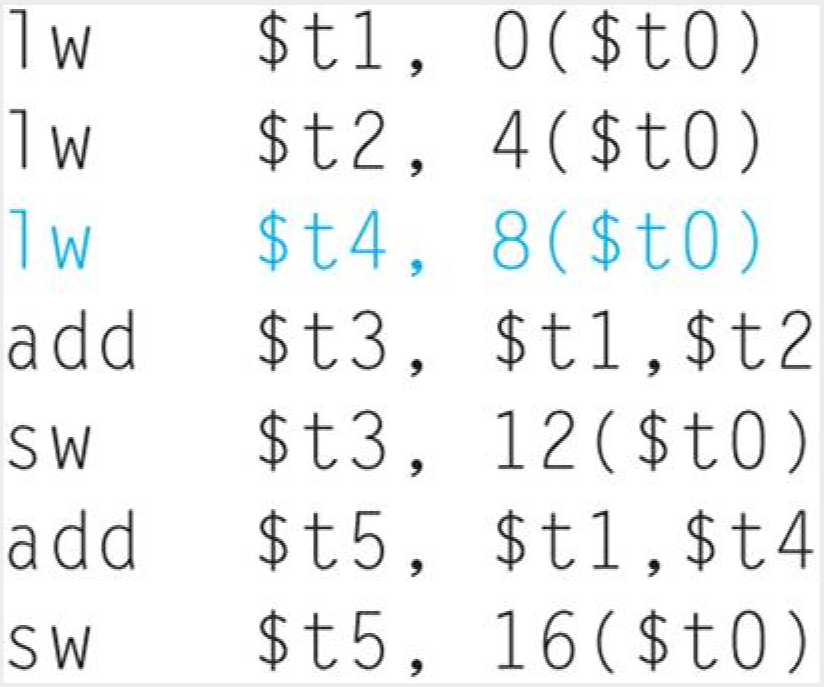
Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from `$t0`:

```
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.

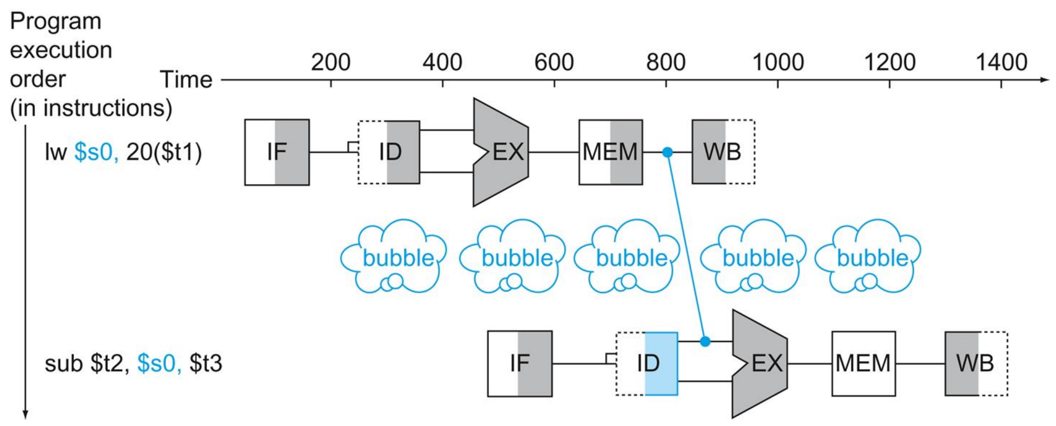


FIGURE 4.30 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 4.8 shows the details of what really happens in the case of a hazard.

Forwarding yields another insight into the MIPS architecture, in addition to the four mentioned on pages 289–290. 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 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 4.9 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.

Performance of “Stall on Branch”

Example

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 3.28 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.

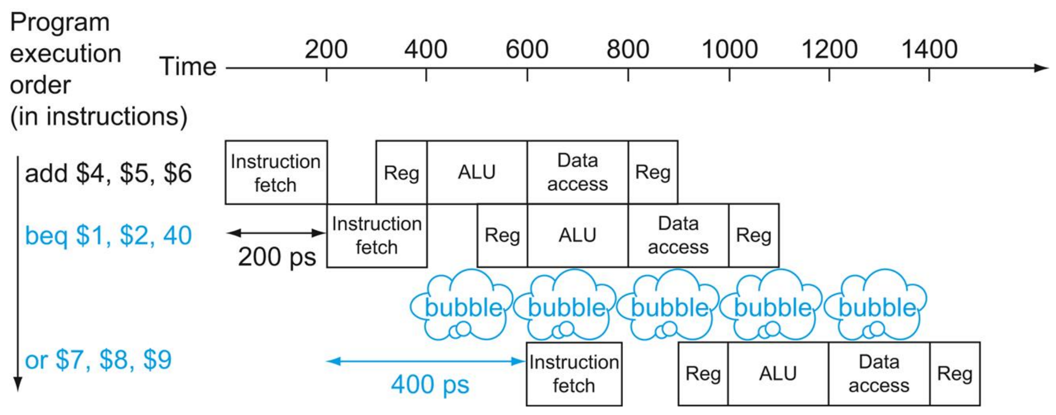


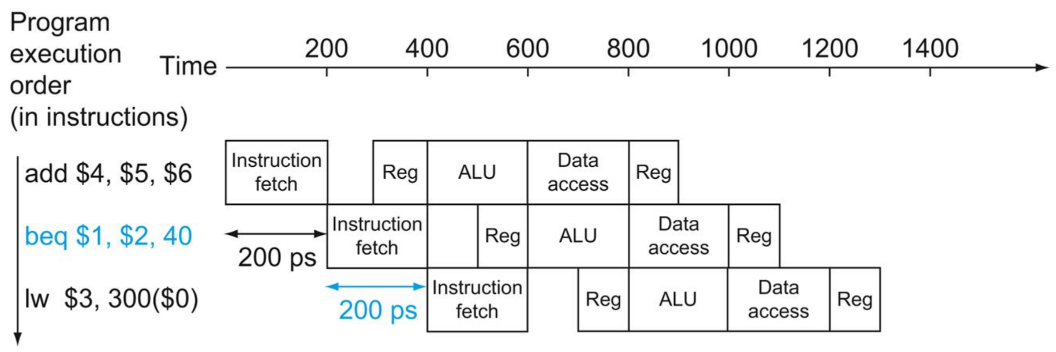
FIGURE 4.31 Pipeline showing stalling on every conditional branch as solution to control hazards.
This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the OR instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 4.9. 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.

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.



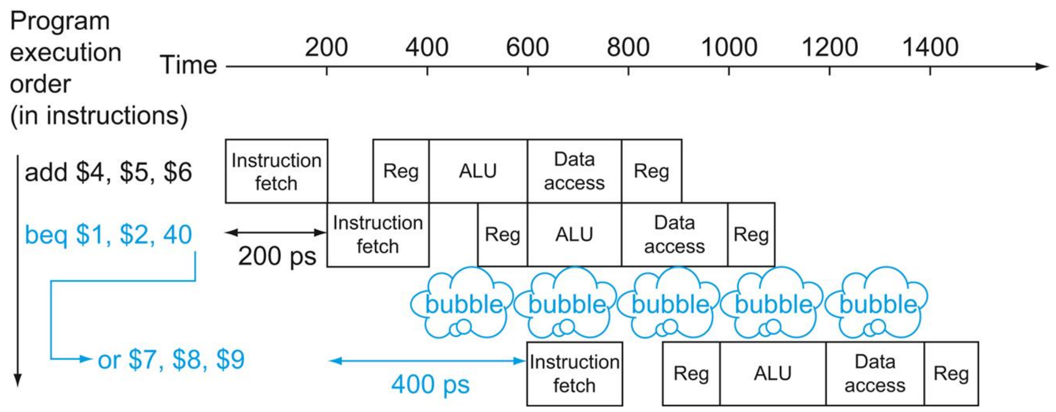


FIGURE 4.32 **Predicting that branches are not taken as a solution to control hazard.** The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 4.31, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. Section 4.9 will reveal the details.

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.

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.

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.

PREDICTION

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 4.9). 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 4.9.

Elaboration

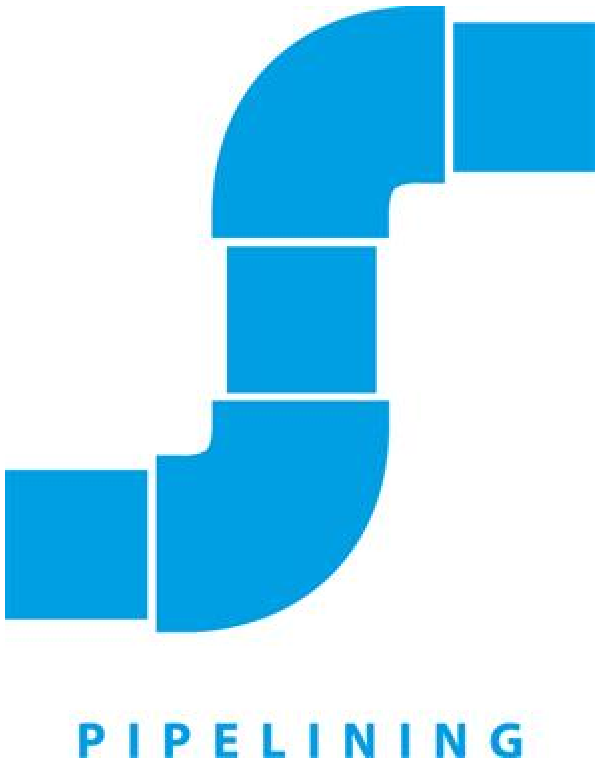
There is a third approach to the control hazard, called *delayed decision*. In our analogy, whenever you are going to make such a decision about laundry, just place a load of nonfootball clothes in the washer while waiting for football uniforms to dry. As long as you have enough dirty clothes that are 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 4.11. 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 then read Section 4.9 to learn more about solutions to branch hazards, and then see how exceptions are handled in Section 4.10.

Check Yourself

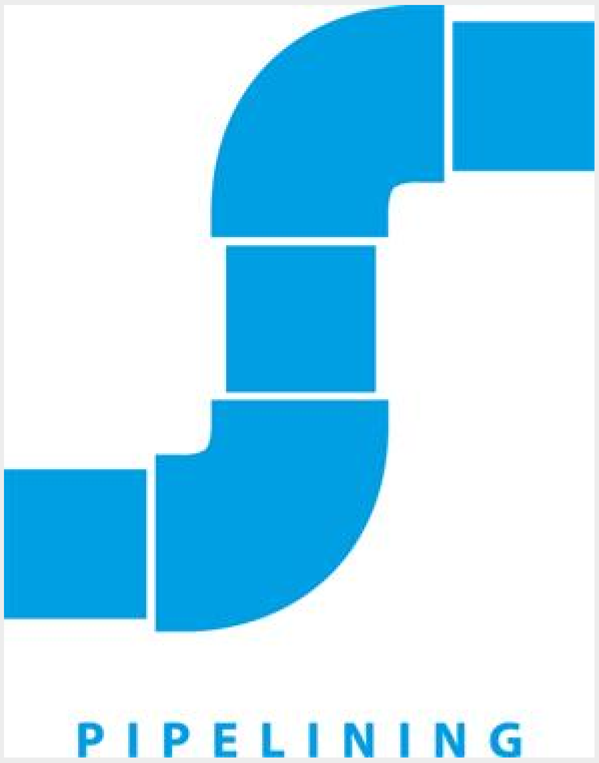
For each code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

Sequence 1	Sequence 2	Sequence 3
`lw $t0,0($t0)`	`add $t1,$t0,$t0`	`addi $t1,$t0,#1`
`add $t1,$t0,$t0`	`addi $t2,$t0,#5`	`addi $t2,$t0,#2`
	`addi $t4,$t1,#5`	`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 4.11, 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 branch frequencies as well as less predictable branches. Data hazards can be performance bottlenecks in both integer and floating-point programs. Often it is easier to deal with data hazards in floating-point programs because the lower branch frequency and more regular 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 4.11, 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 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.

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.

PREDICTION

4.7 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

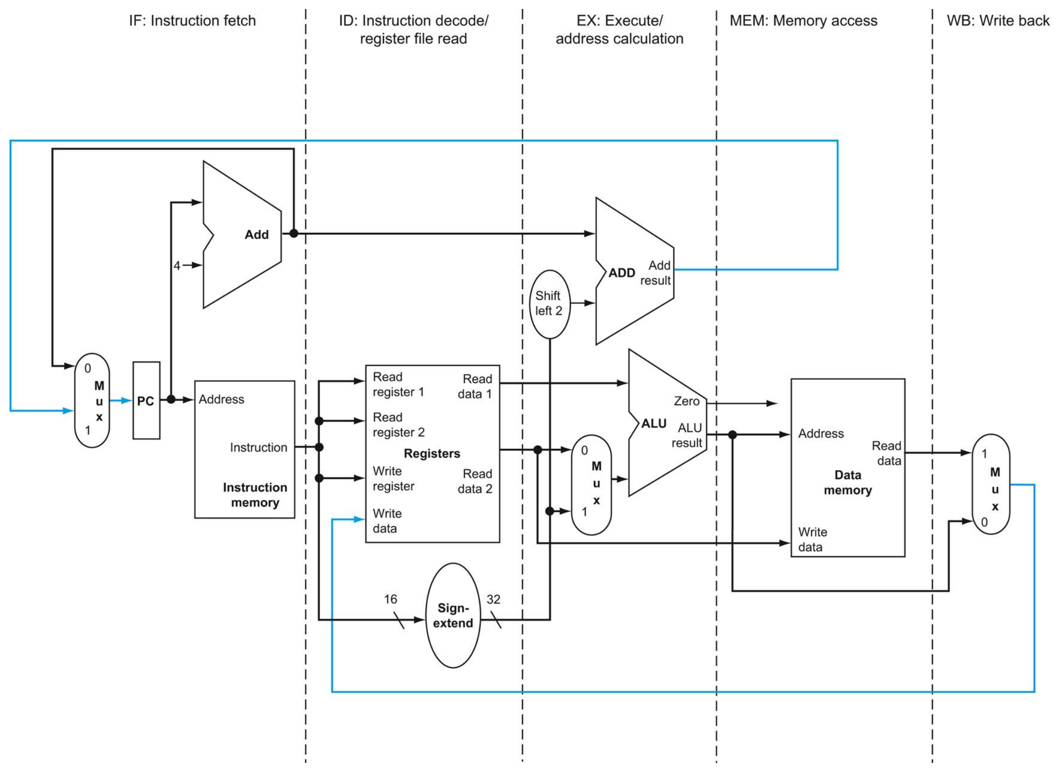


FIGURE 4.33 **The single-cycle datapath from Section 4.4 (similar to Figure 4.17).** Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from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

In Figure 4.33, these five components correspond roughly to the way the data-path is drawn; instructions and data move generally from left to right through the 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.

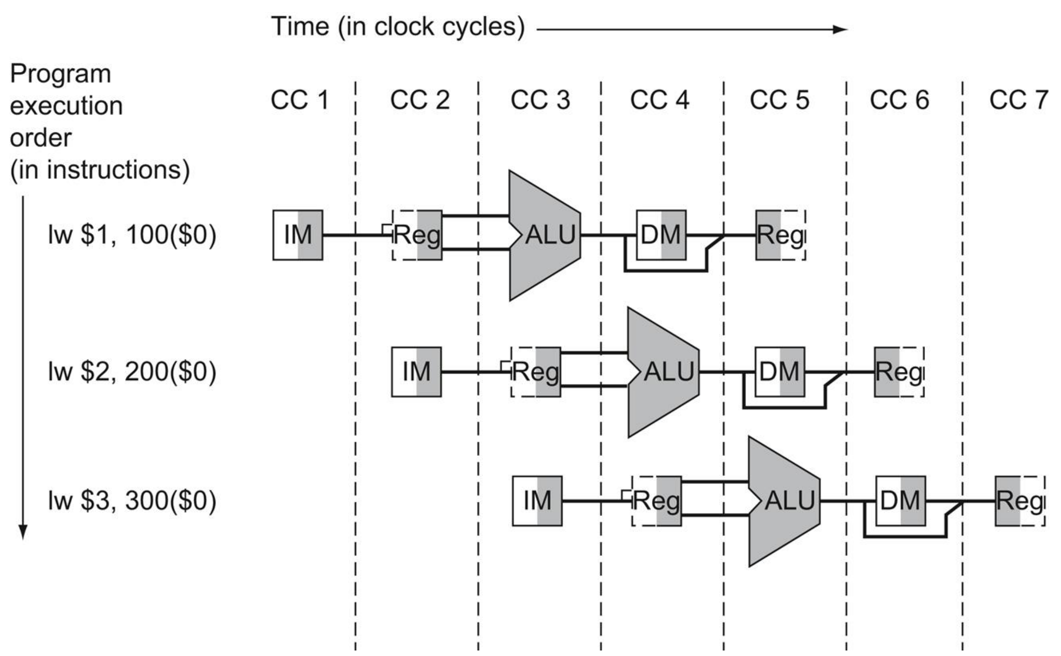


FIGURE 4.34 Instructions being executed using the single-cycle datapath in Figure 4.33, assuming pipelined execution.
Similar to Figures 4.28 through 4.30, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled 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 4.34 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 4.35 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.

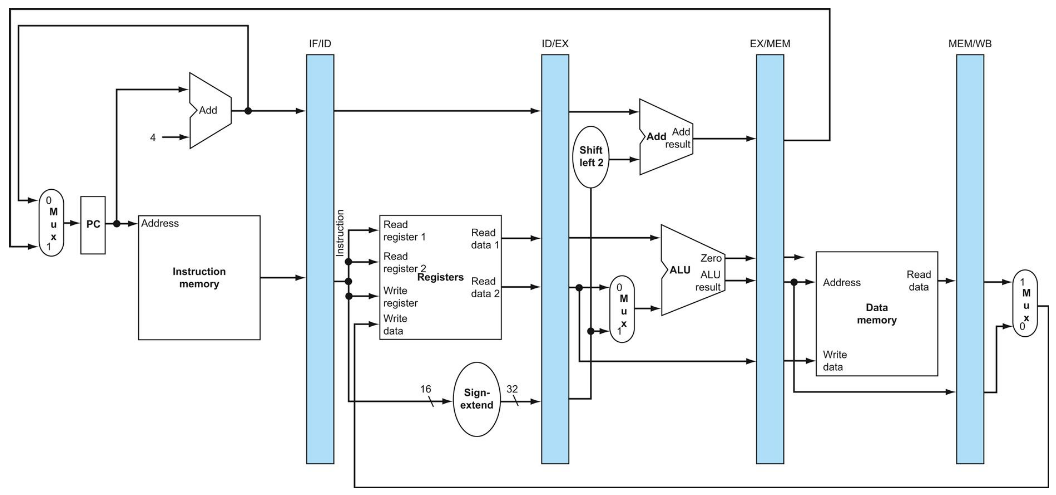


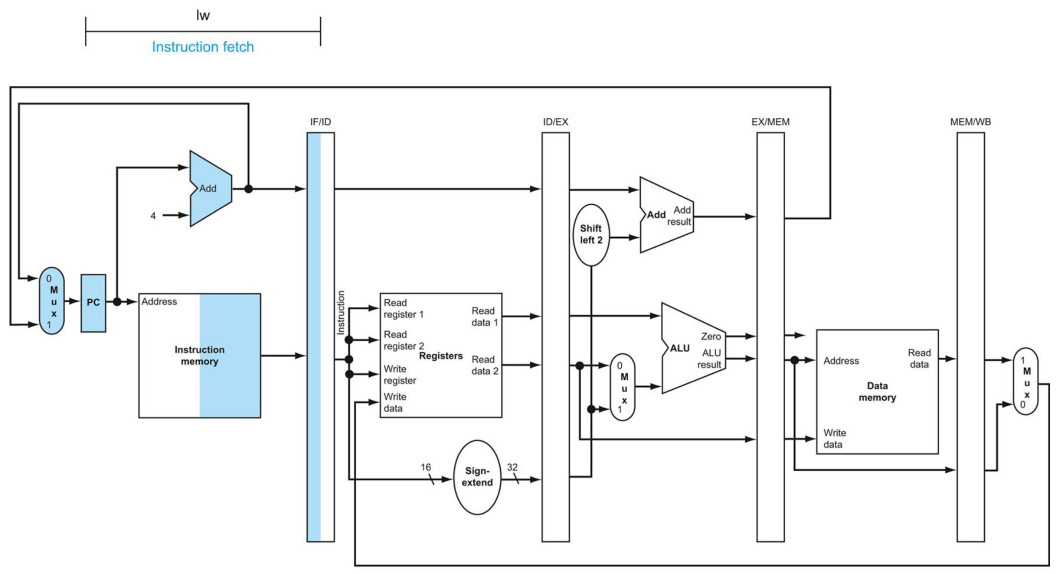
FIGURE 4.35 **The pipelined version of the datapath in Figure 4.33.**
The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages.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

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 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.

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*.



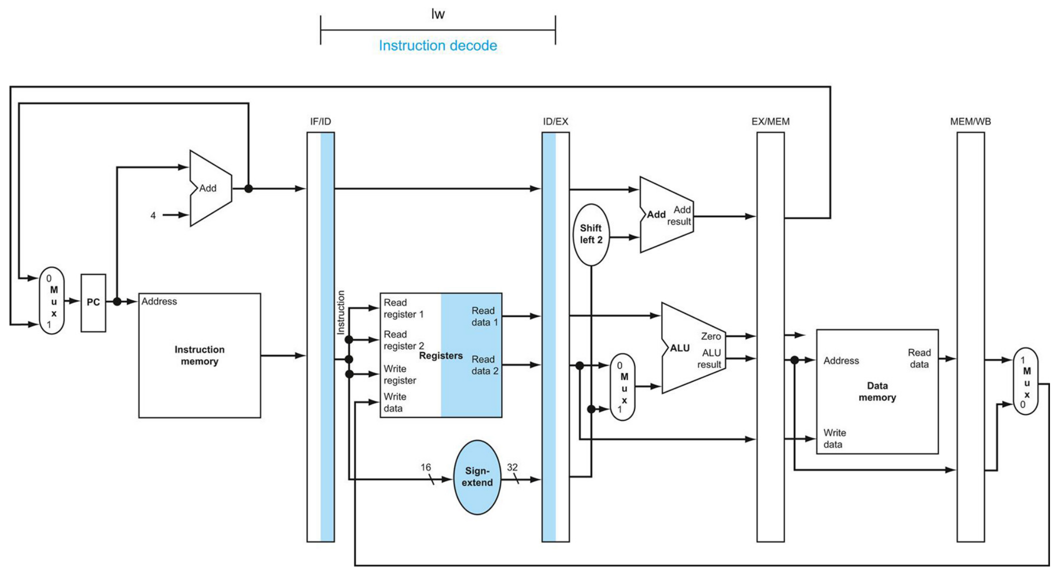


FIGURE 4.36 IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in Figure 4.35 highlighted.
The highlighting convention is the same as that used in Figure 4.28. As in Section 4.2, there is no confusion when reading and writing registers, because the contents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

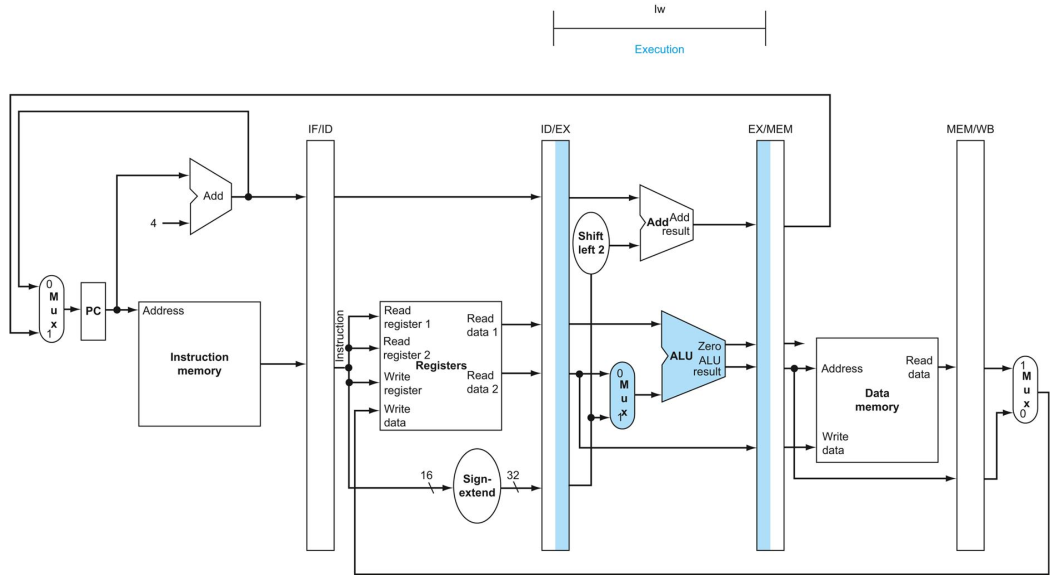
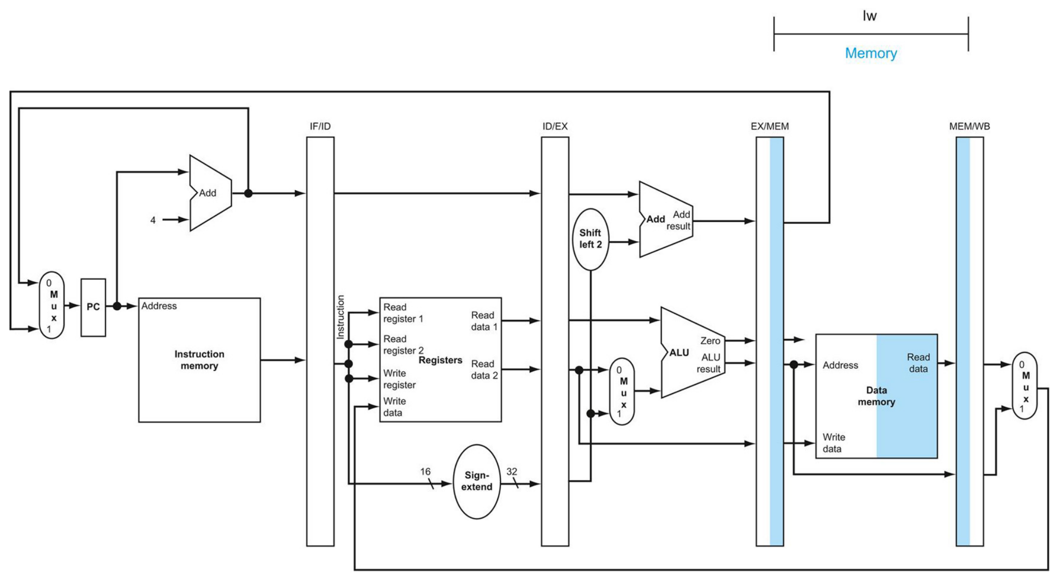


FIGURE 4.37 **EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 4.35 used in this pipe stage.**
The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.



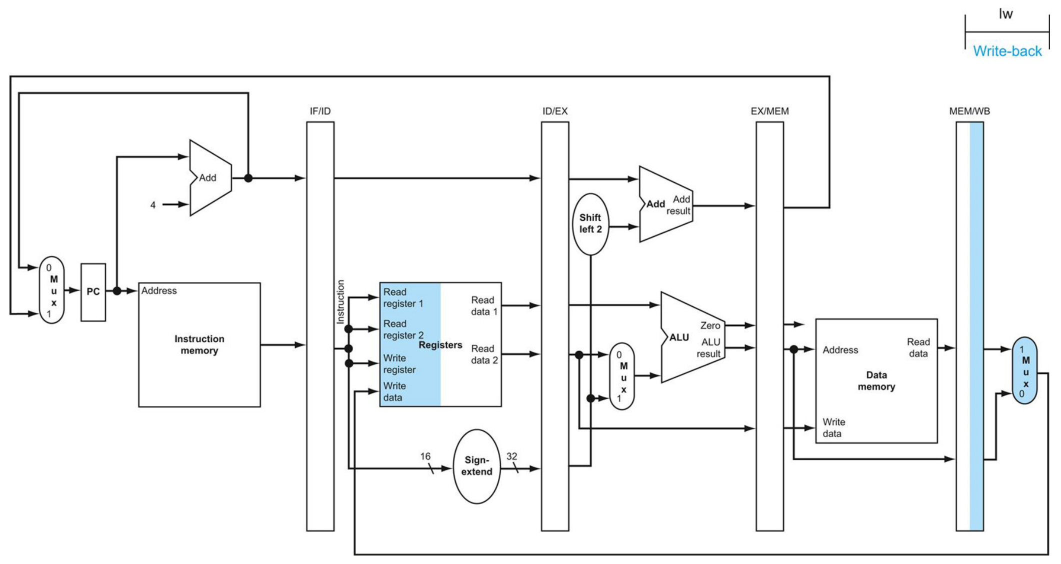


FIGURE 4.38 **MEM and WB: The fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 4.35 used in this pipe stage.**
Data memory is read using the address in the EX/MEM pipeline registers, and the data is placed in the MEM/WB pipeline register. Next, data 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.

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:

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.
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.

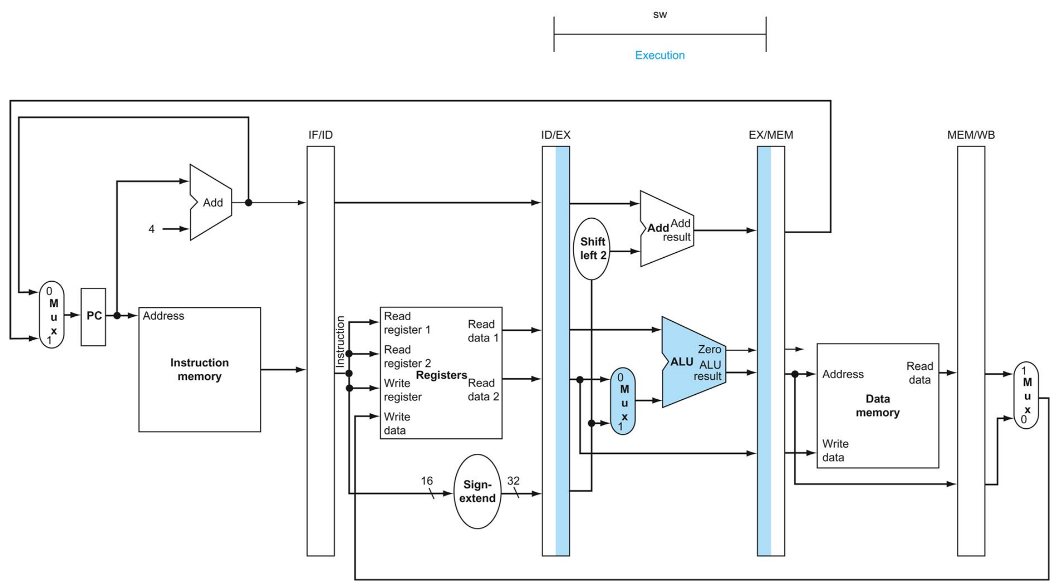
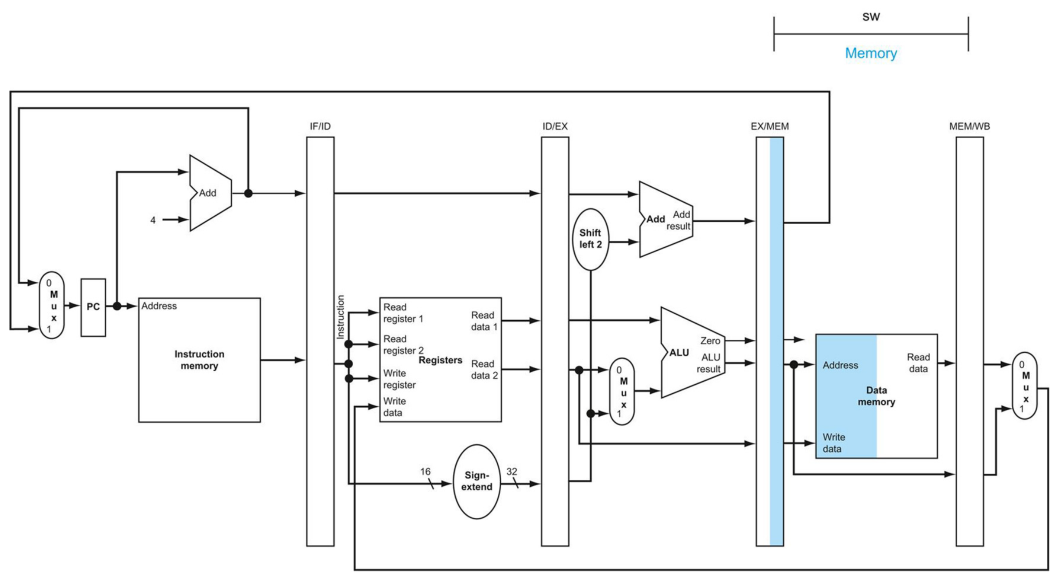


FIGURE 4.39 **EX: The third pipe stage of a store instruction.**
Unlike the third stage of the load instruction in Figure 4.37, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn’t hurt to always write this second register into the EX/MEM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



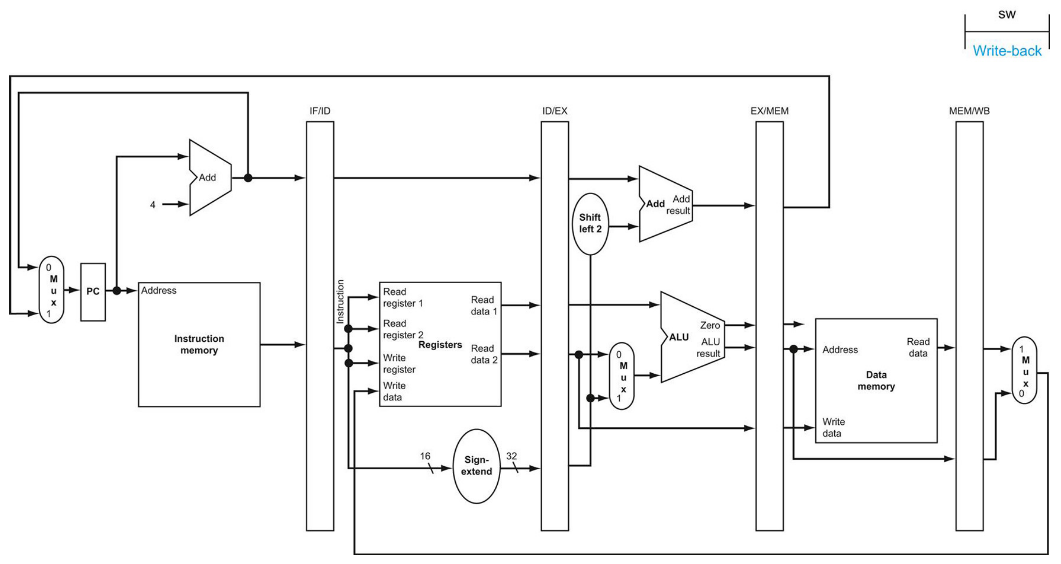


FIGURE 4.40 **MEM and WB: The fourth and fifth pipe stages of a store instruction.**
In the fourth stage, the data is written into data memory for the store. Note that the data comes from the EX/MEM pipeline register and that nothing is changed in the MEM/WB pipeline register. Once the data is written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

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

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

Hence, we need to preserve the destination register number in the load instruction. Just as store passed the register *contents* from the ID/EX to the EX/MEM pipeline registers for use in the MEM stage, load must pass the *register* number from the ID/EX through EX/MEM to the MEM/WB pipeline register for use in 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 4.9 for an explanation of how to make the branch instruction work as expected.

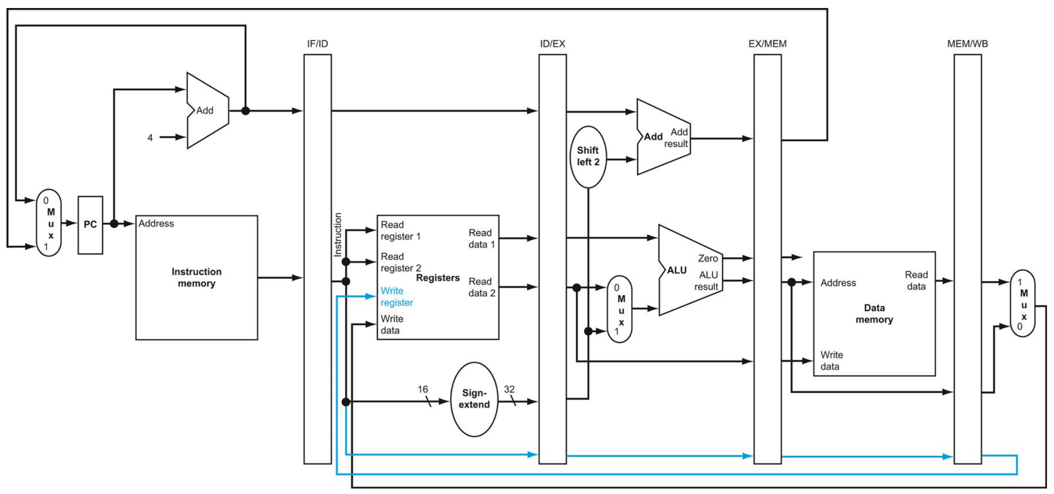


FIGURE 4.41 **The corrected pipelined datapath to handle the load instruction properly.** The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits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

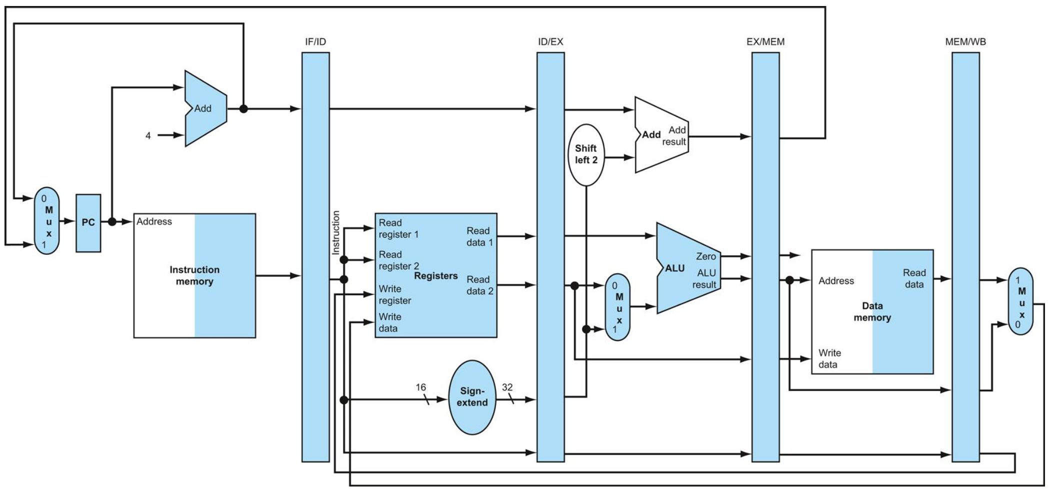


FIGURE 4.42 The portion of the datapath in Figure 4.41 that is used in all five stages of a load instruction.

Graphically Representing Pipelines

Pipelining can be difficult to understand, since many instructions are simultaneously executing in a single datapath in every clock cycle. To aid understanding, there are two basic styles of pipeline figures: *multiple-clock-cycle pipeline diagrams*, such as Figure 4.34 on page 300, 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.

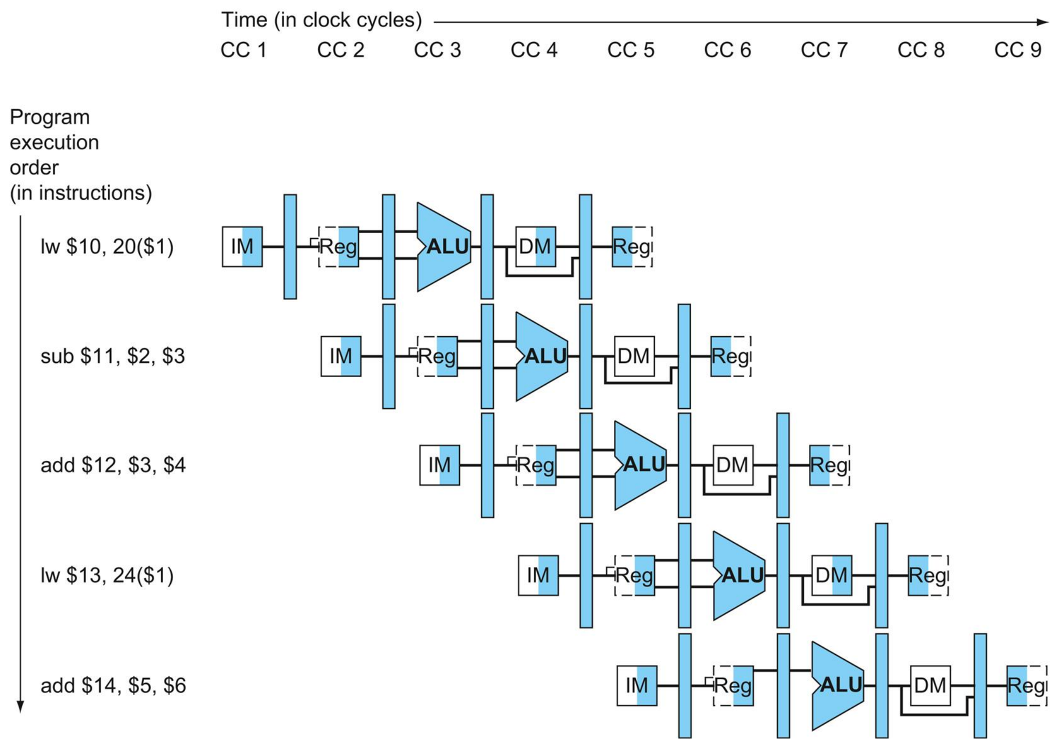


FIGURE 4.43 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure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

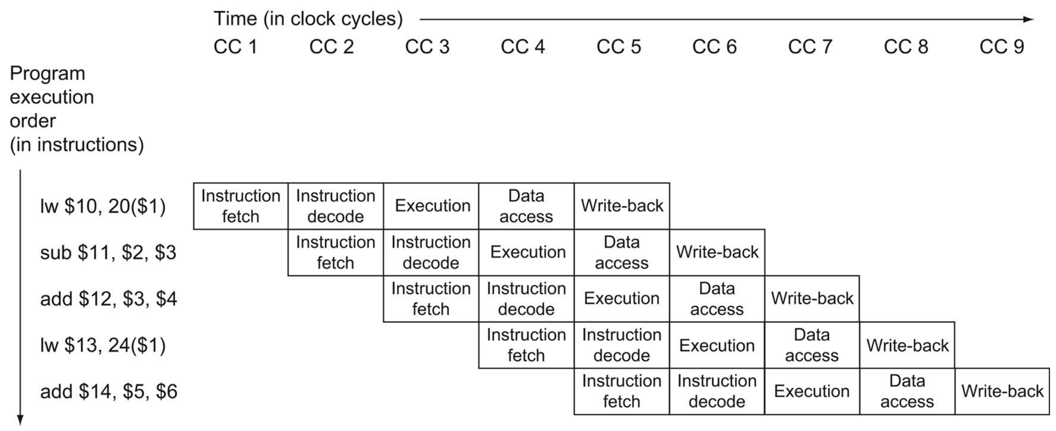


FIGURE 4.44 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.43.

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 4.43.) 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?

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.

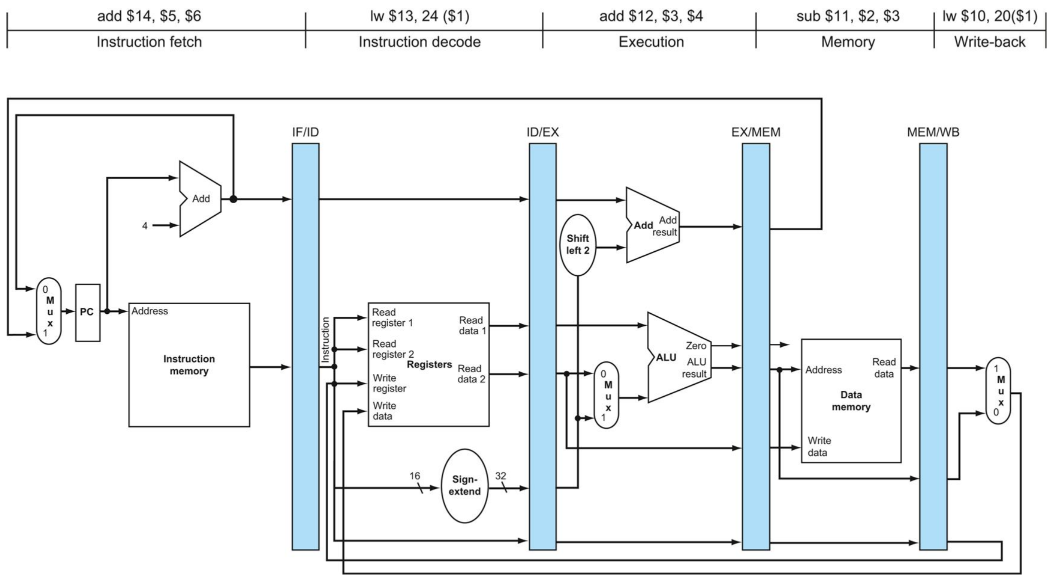


FIGURE 4.45 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.43 and 4.44.
As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

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 in two pages in this section to make the following discussion easier to follow.

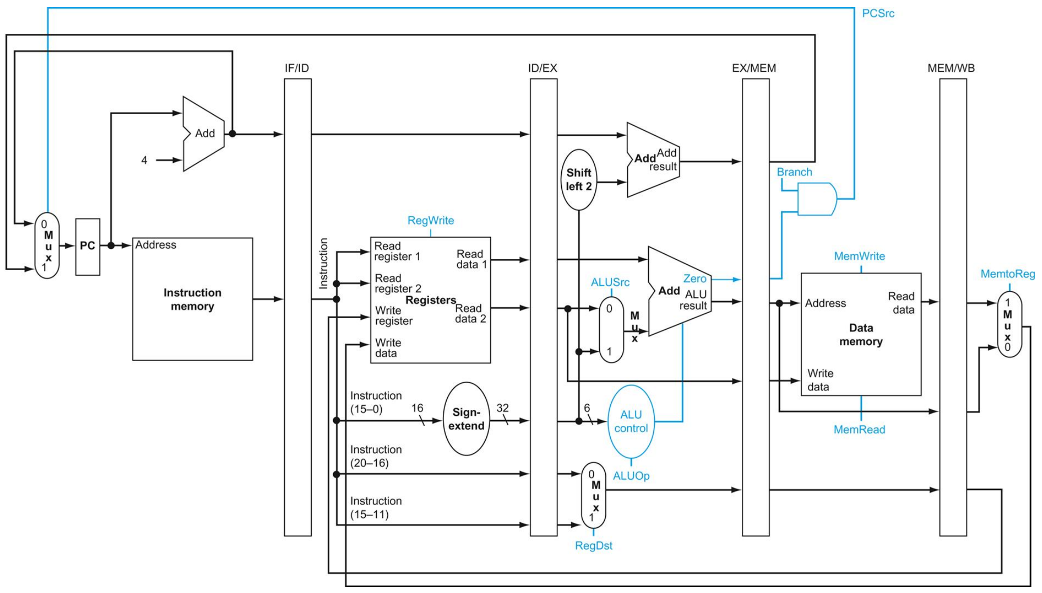


FIGURE 4.46 **The pipelined datapath of Figure 4.41 with the control signals identified.** This datapath borrows the control logic for PC source, register destination number, and ALU control from Section 4.4. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage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

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.
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 Reg-Write, 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.

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 4.14** gives more examples of MIPS code executing on pipelined hardware using single-clock diagrams, if you would like to see more details.)

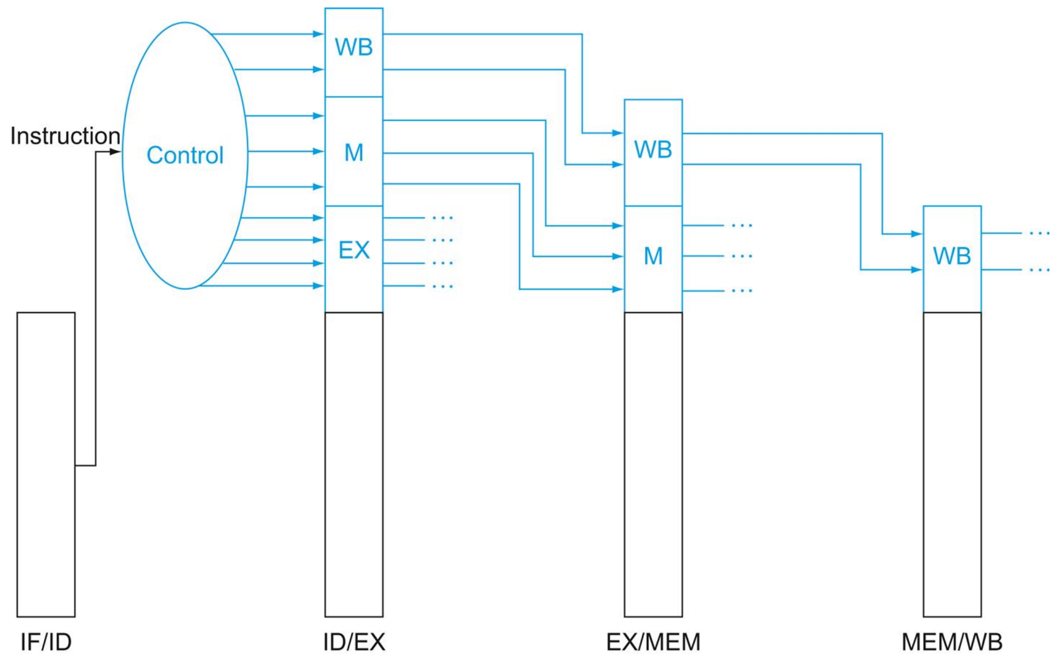


FIGURE 4.50 The control lines for the final three stages.
Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed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

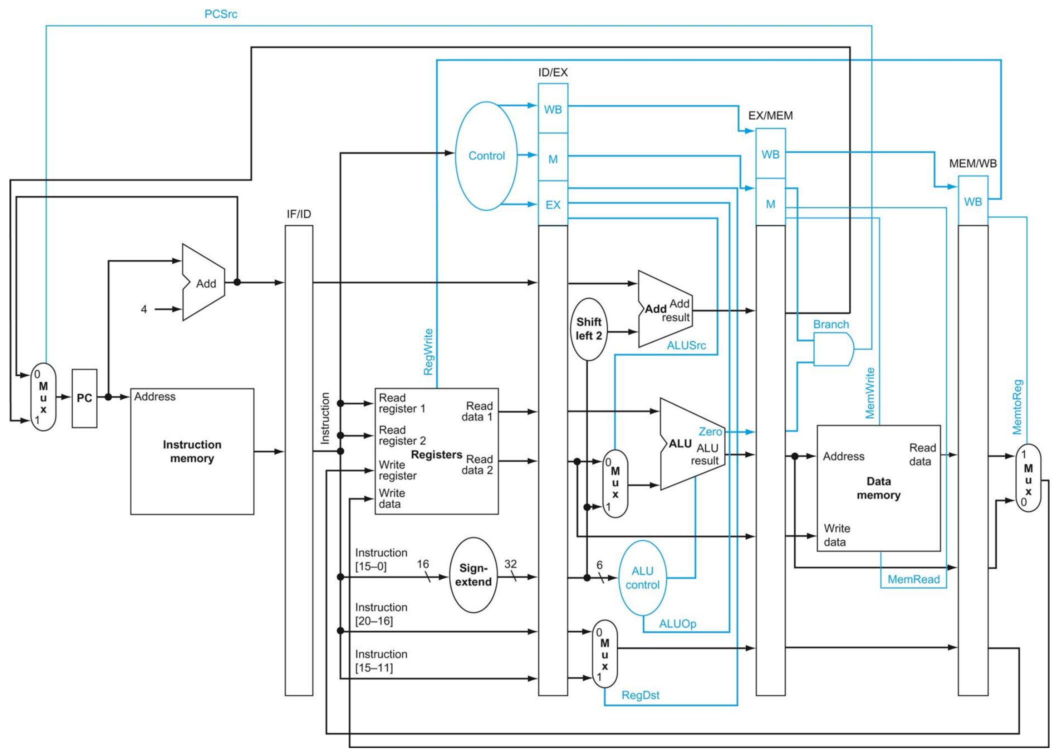


FIGURE 4.51 **The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers.** The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for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

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 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 4.6, we saw that data hazards are obstacles to pipelined execution.

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.

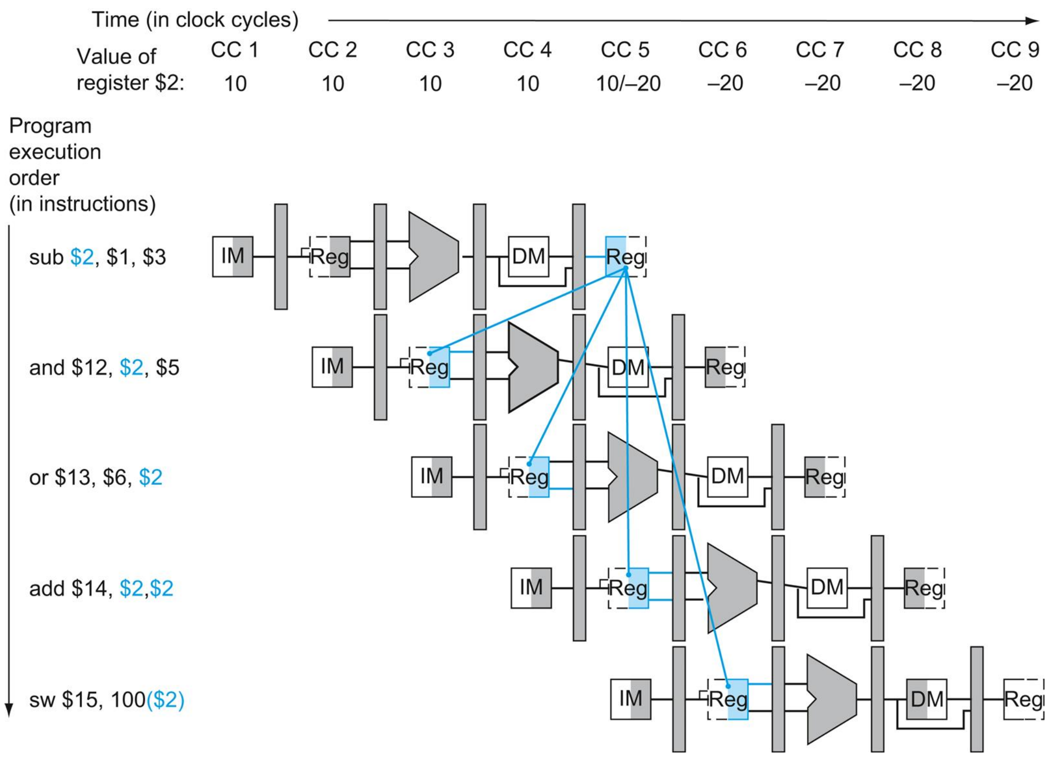


FIGURE 4.52 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into `$2`, and all the following instructions read `$2`.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

The potential hazard of `add` 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 `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 4.6, 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 316 is on register `$2`, between the result of `sub $2,$1,$3` and the first read operand of `and $12,$2,$5`.

This hazard can be detected when the `and` instruction is in the EX stage and the prior instruction is in the MEM stage, so this is hazard 1a:

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

Dependence Detection

Example

Classify the dependences in this sequence from page 316:

```
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:

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

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

Because some instructions do not write registers, this policy is inaccurate; sometimes it would forward when it 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.

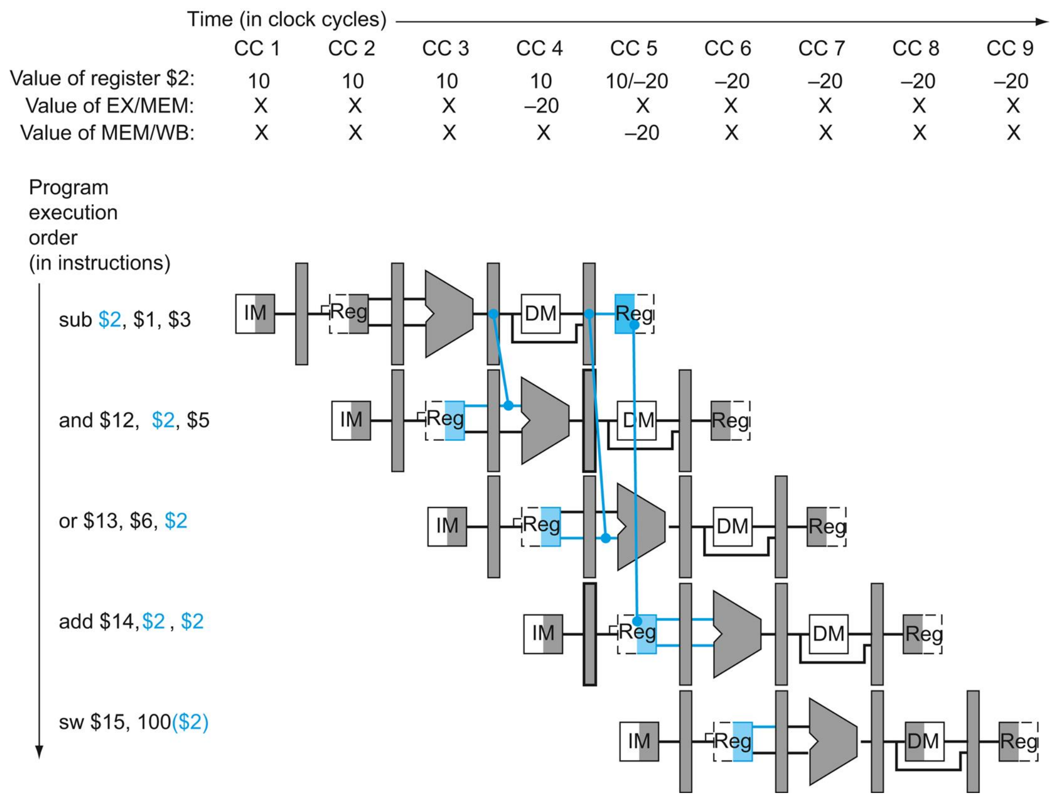


FIGURE 4.53 **The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the `AND` instruction and `OR` instruction by forwarding the results found in the pipeline registers.** The values in the pipeline registers show that the desired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

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.

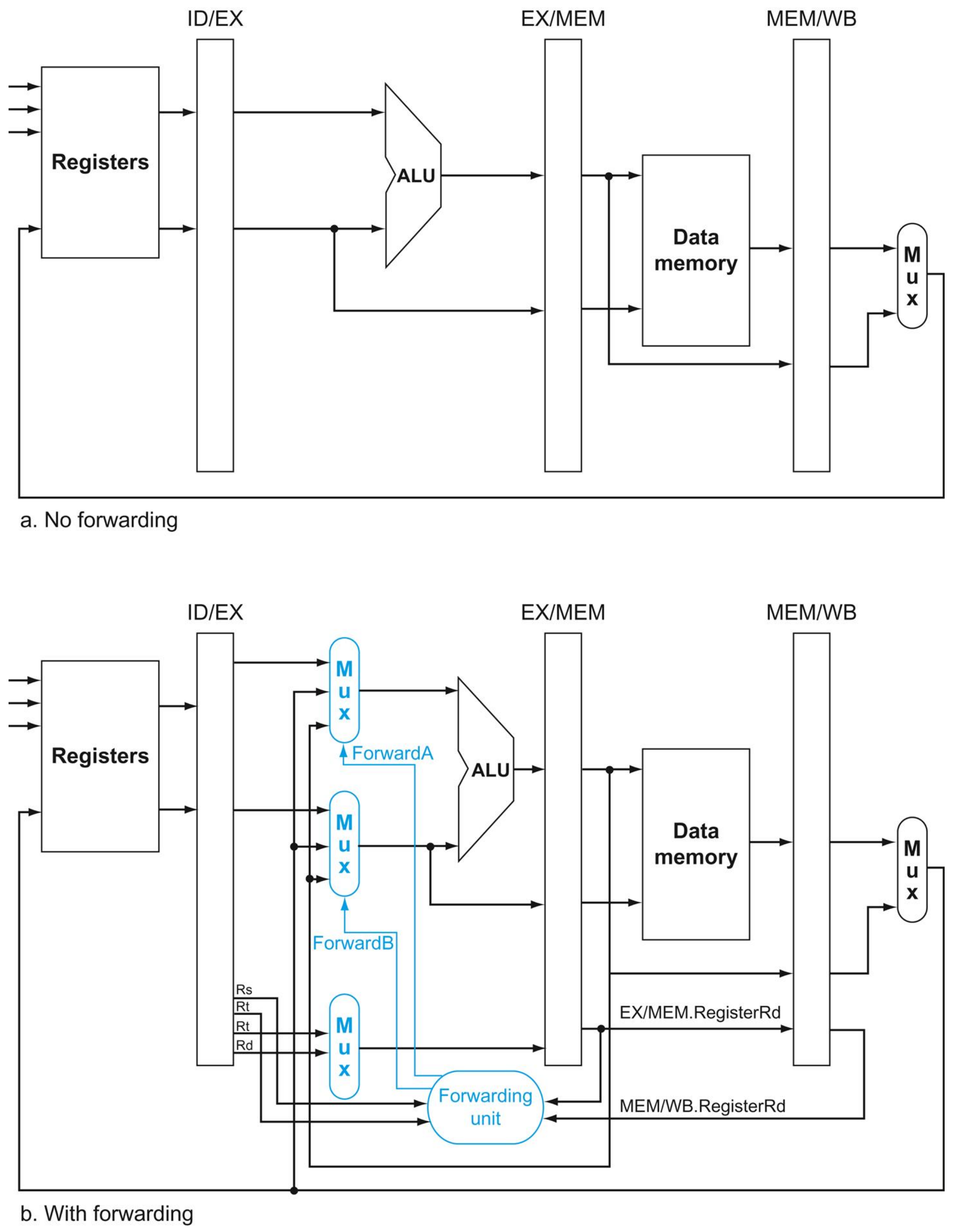


FIGURE 4.54 **On the top are the ALU and pipeline registers before adding forwarding.** On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details 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.

This forwarding control will be in the EX stage, because the ALU forwarding multiplexors are found in that stage. Thus, we must pass the operand register numbers from the ID stage via the ID/EX pipeline register to determine whether to forward values. We already have the rt field (bits 20–16). Before forwarding, the ID/EX register had no need to include space to hold the rs field. Hence, rs (bits 25–21) is added to ID/EX.

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

1. *EX hazard:*

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

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

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,$2add $1,$1,$3add $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).

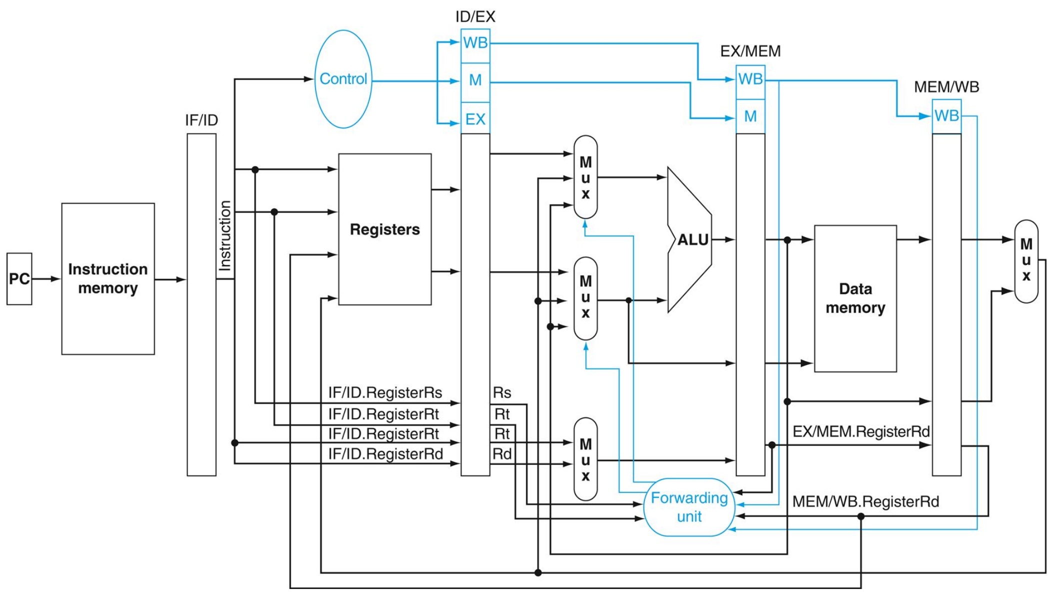


FIGURE 4.56 The datapath modified to resolve hazards via forwarding.
Compared with the datapath in Figure 4.51, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath, such as the branch hardware and the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

Section 4.14 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 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.

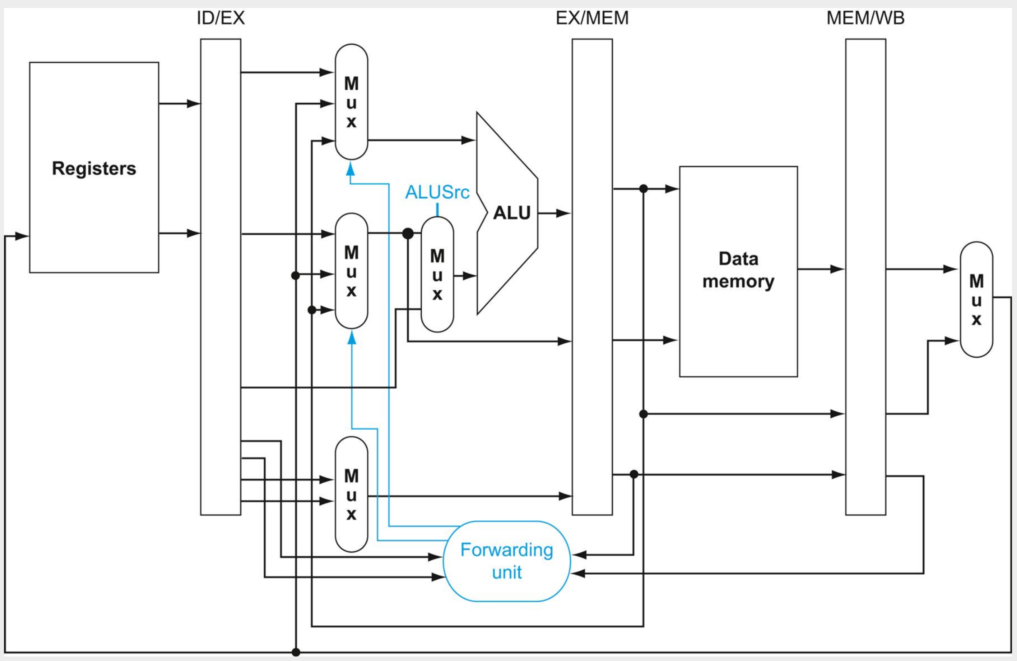


FIGURE 4.57 A close-up of the datapath in Figure 4.54 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.

Data Hazards and Stalls

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

Anonymous

As we said in Section 4.6, 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.

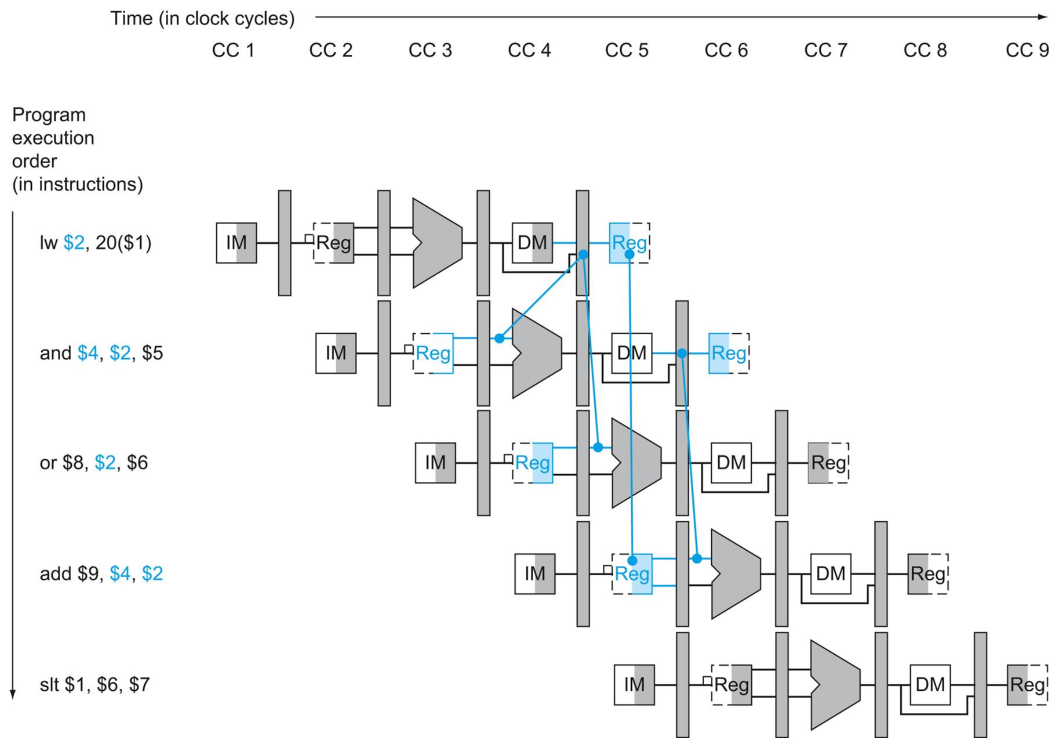


FIGURE 4.58 **A pipelined sequence of instructions.**
Since the dependence between the load and the following instruction (`and`) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

Hence, in addition to a forwarding unit, we need a *hazard detection unit*. It operates during the ID stage so that it can insert the stall between the load and its use. Checking for load instructions, the control for the hazard detection unit is this single condition:

if (ID/EX.MemRead and

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

The first line tests to see if the instruction is a load: the only instruction that reads data memory is a load. The next two lines check to see if the destination register field of the load in the EX stage matches either source register of the instruction in the ID stage. If the condition holds, the instruction stalls 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.

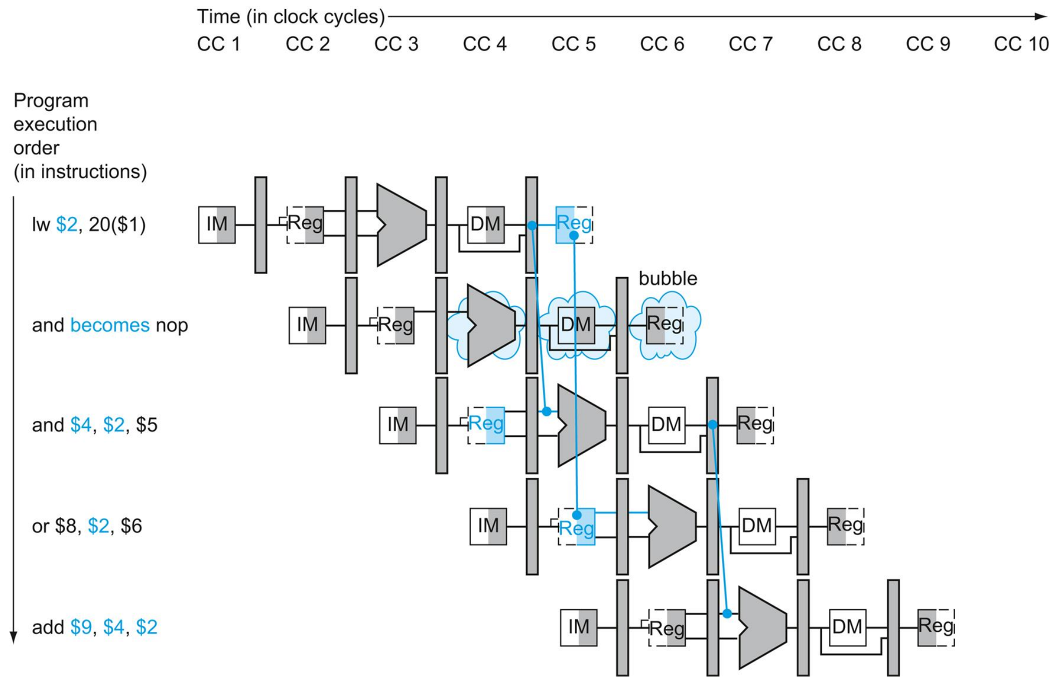


FIGURE 4.59 The way stalls are really inserted into the pipeline.
A bubble is inserted beginning in clock cycle 4, by changing the `and` instruction to a nop. Note that the `and` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus 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.

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 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 4.14** 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.

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.

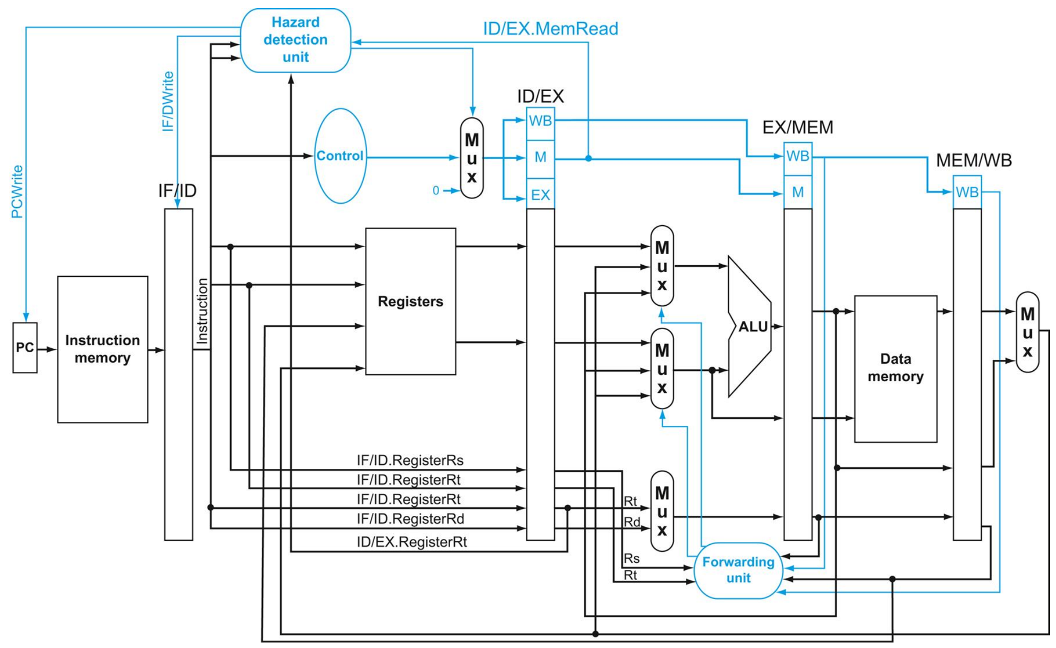


FIGURE 4.60 **Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit.**
Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware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

4.9 Control Hazards

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

Henry David Thoreau, *Walden*, 1854

Thus far, we have limited our concern to hazards involving arithmetic operations and data transfers. However, as we saw in Section 4.6, 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 4.6, 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.

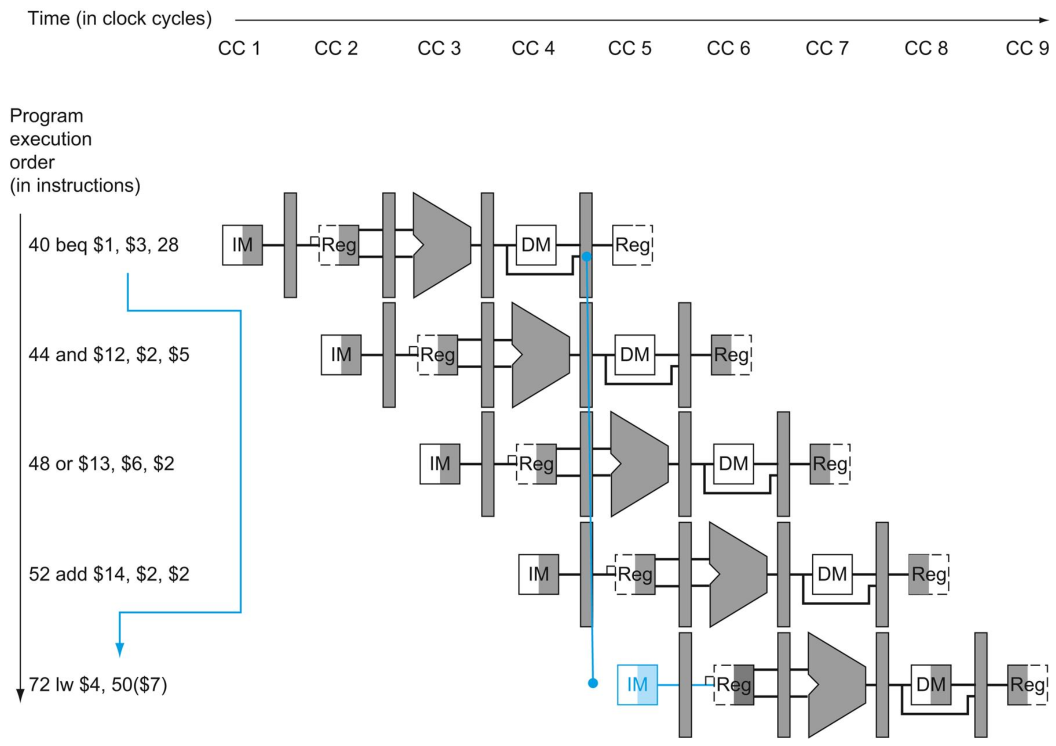


FIGURE 4.61 The impact of the pipeline on the branch instruction.
The numbers to the left of the instruction (40, 44, …) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the `beq` instruction above—the three sequential instructions 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 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.

Assume Branch Not Taken

As we saw in Section 4.6, 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.

flush

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

Reducing the Delay of Branches

One way to improve branch performance is to reduce the cost of the taken branch. Thus far, we have assumed the next PC for a branch is selected in the MEM stage, but if we move the branch execution earlier in the pipeline, then fewer instructions need be flushed. The MIPS architecture was designed to support fast single-cycle branches that could be pipelined with a small branch penalty. The designers observed that many branches rely only on simple tests (equality or sign, for example) and that such tests do not require a full ALU operation but can be done with at most a few gates. When a more complex branch decision is required, a separate instruction that uses an ALU to perform a comparison is required—a situation that is similar to the use of condition codes for branches (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. (A zero output of the OR gate means the two registers are equal.) 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.

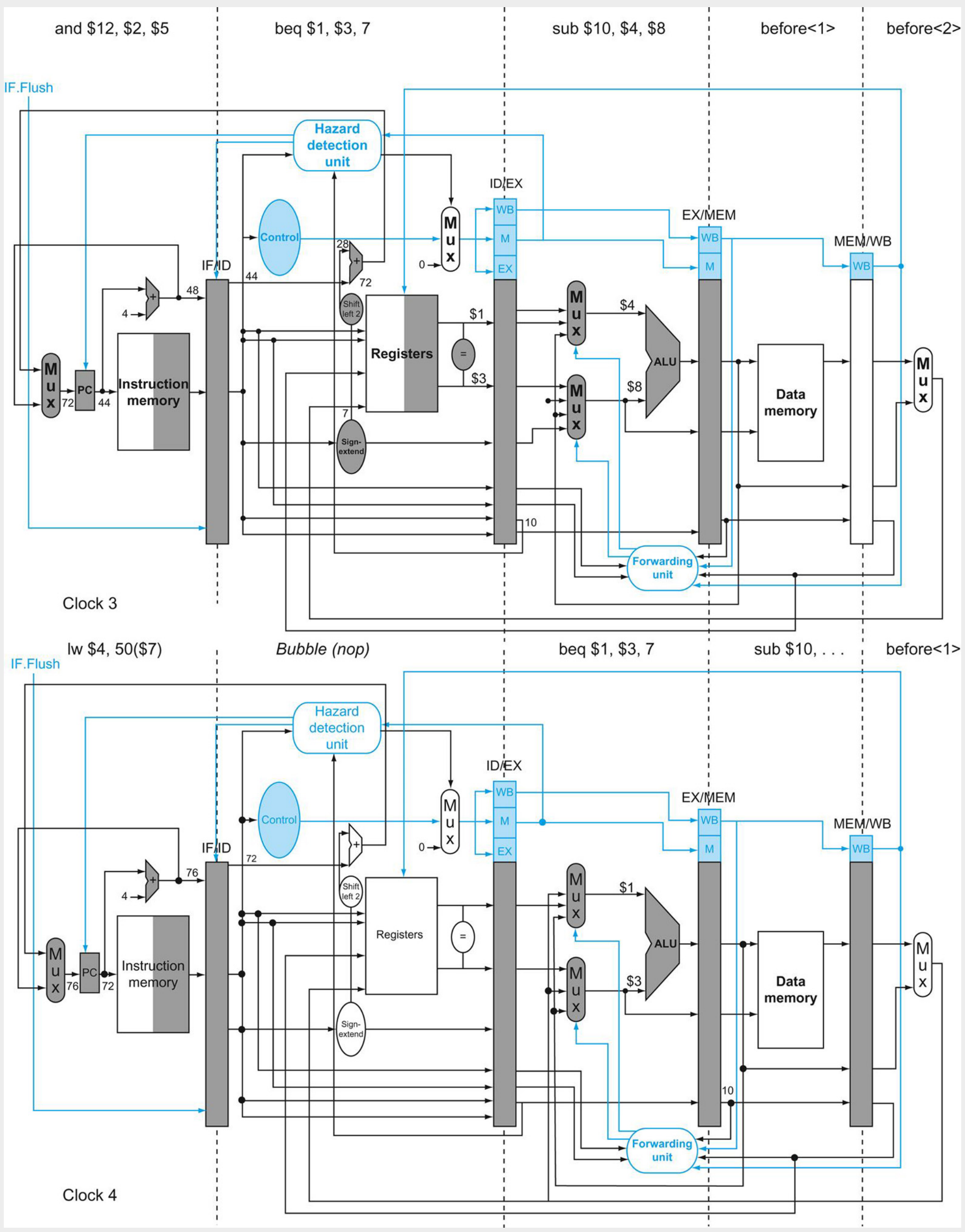


FIGURE 4.62 **The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle.**
Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or `nop` instruction in the 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 4.11), 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 4.6, with more hardware it is possible to try to **predict** branch behavior during program execution.

One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and, if so, to begin fetching new instructions from the same place as the last time. This technique is called **dynamic branch prediction**.

dynamic branch prediction

Prediction of branches at runtime using runtime information.

One implementation of that approach is a **branch prediction buffer** or **branch history table**. A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

branch prediction buffer

Also called **branch history table**. A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not.

This is the simplest sort of buffer; we don't know, in fact, if the prediction is the right one—it may have been put there by another branch that has the same low-order address bits. However, this doesn't affect correctness. Prediction is just a hint that we hope is accurate, 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, we can use more prediction bits. 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.

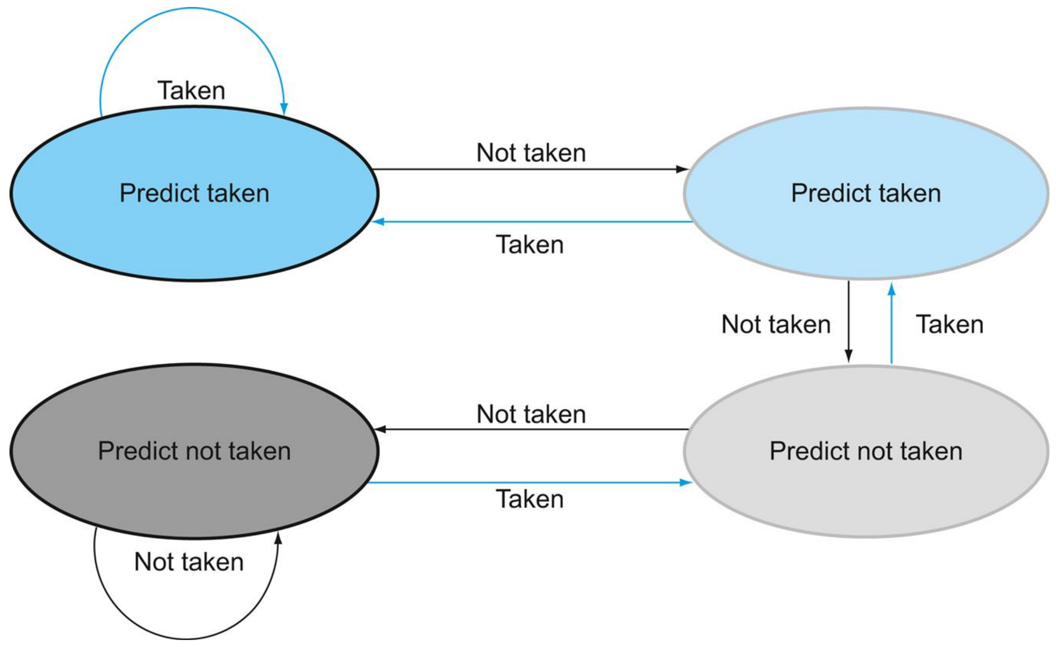


FIGURE 4.63 **The states in a 2-bit prediction scheme.**
By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The 2-bit scheme is a general instance of a counter-based 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.

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 330, 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 Figure 4.63 shows.

Elaboration

As we described in Section 4.6, 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.

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.

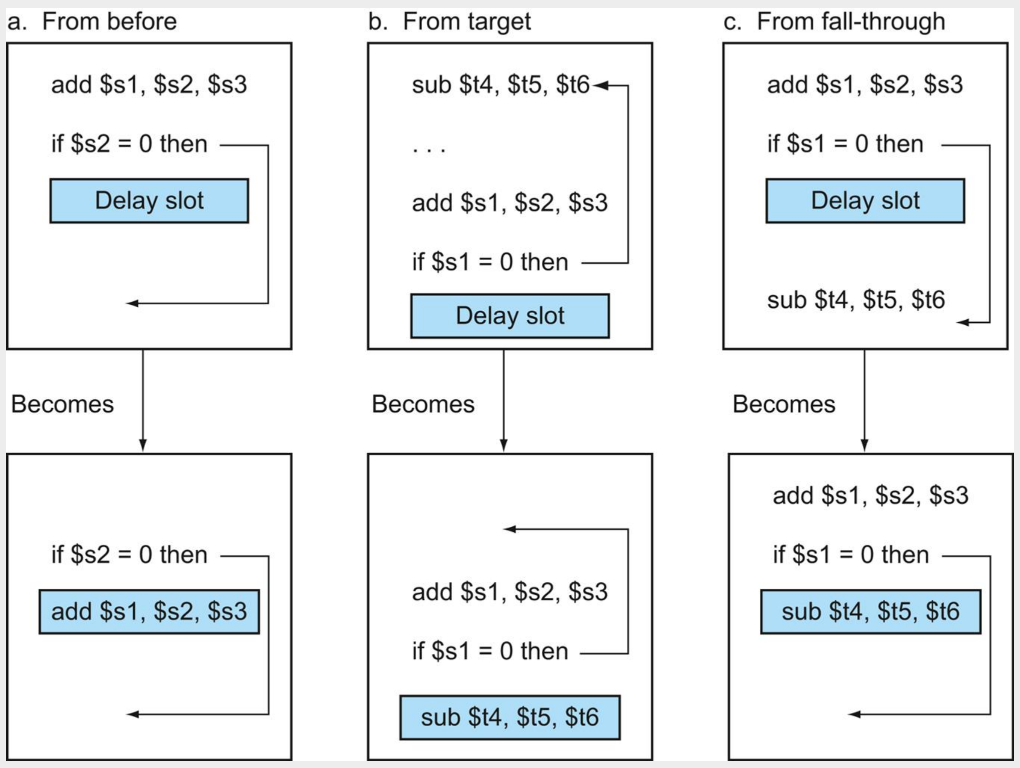


FIGURE 4.64 **Scheduling the branch delay slot.** The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when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

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 4.11), 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. There are more transistors in the branch predictor of modern microprocessors than there were in the whole first MIPS chips!

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 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

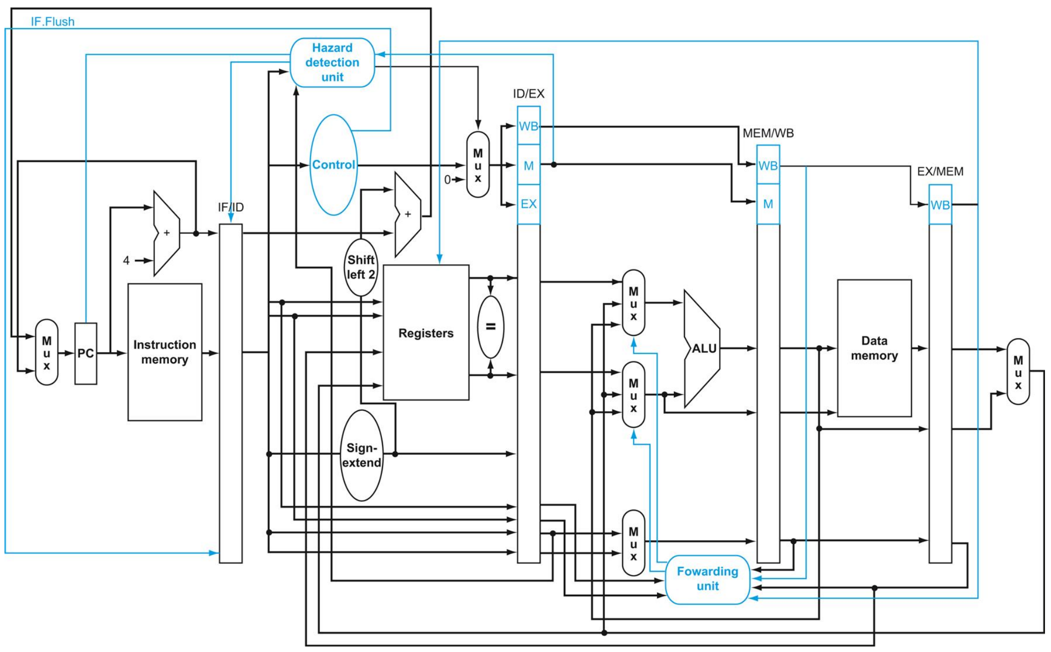


FIGURE 4.65 **The final datapath and control for this chapter.**
Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.57 and the multiplexor controls from Figure 4.51.

4.10 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 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.

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.)

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:

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

vectored interrupt

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

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 $8000\ 0180_{\text{hex}}$. (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.

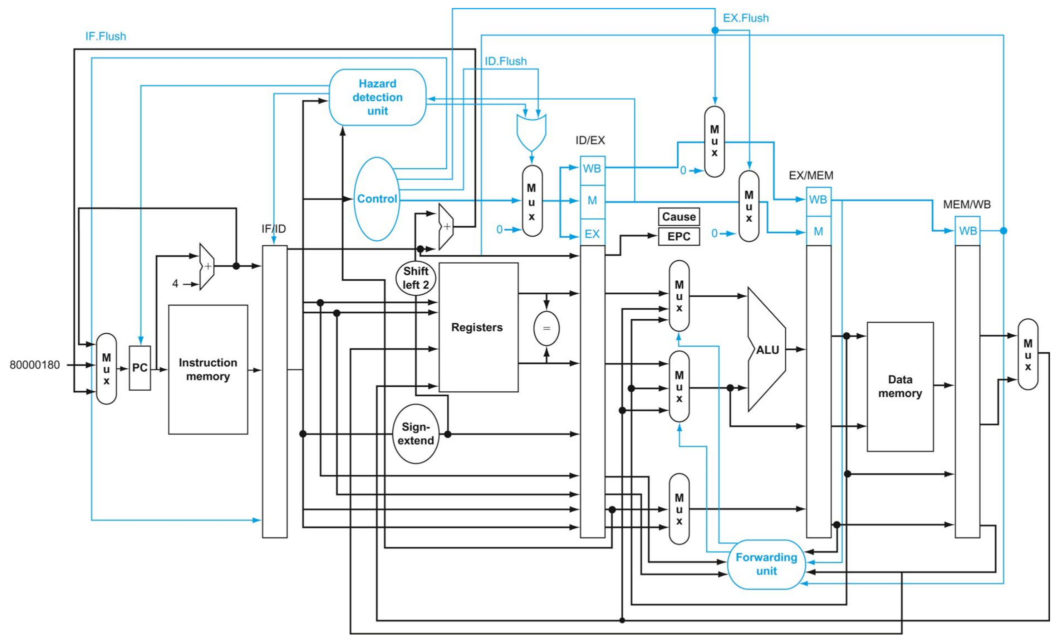


FIGURE 4.66 The datapath with controls to handle exceptions. The key additions include a new input with the value 8000 0180$_{\text{hex}}$ in the multiplexor that supplies the new PC value; a Cause register to record the cause of the exception; and an Exception PC register to save the address of the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

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 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.

Exception in a Pipelined Computer

Example

Given this instruction sequence,

40_{hex}`sub $11, $2, $4`

44_{hex}`and $12, $2, $5`

48_{hex}`or $13, $2, $6`

$4C_{hex}$`add $1, $2, $1`

50_{hex}`slt $15, $6, $7`

54_{hex}`lw $16, 50($7)`

. . .

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

80000180_{hex} `sw $26, 1000($0)`

80000184_{hex} `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: $4C_{hex}+4 = 50_{hex}$.

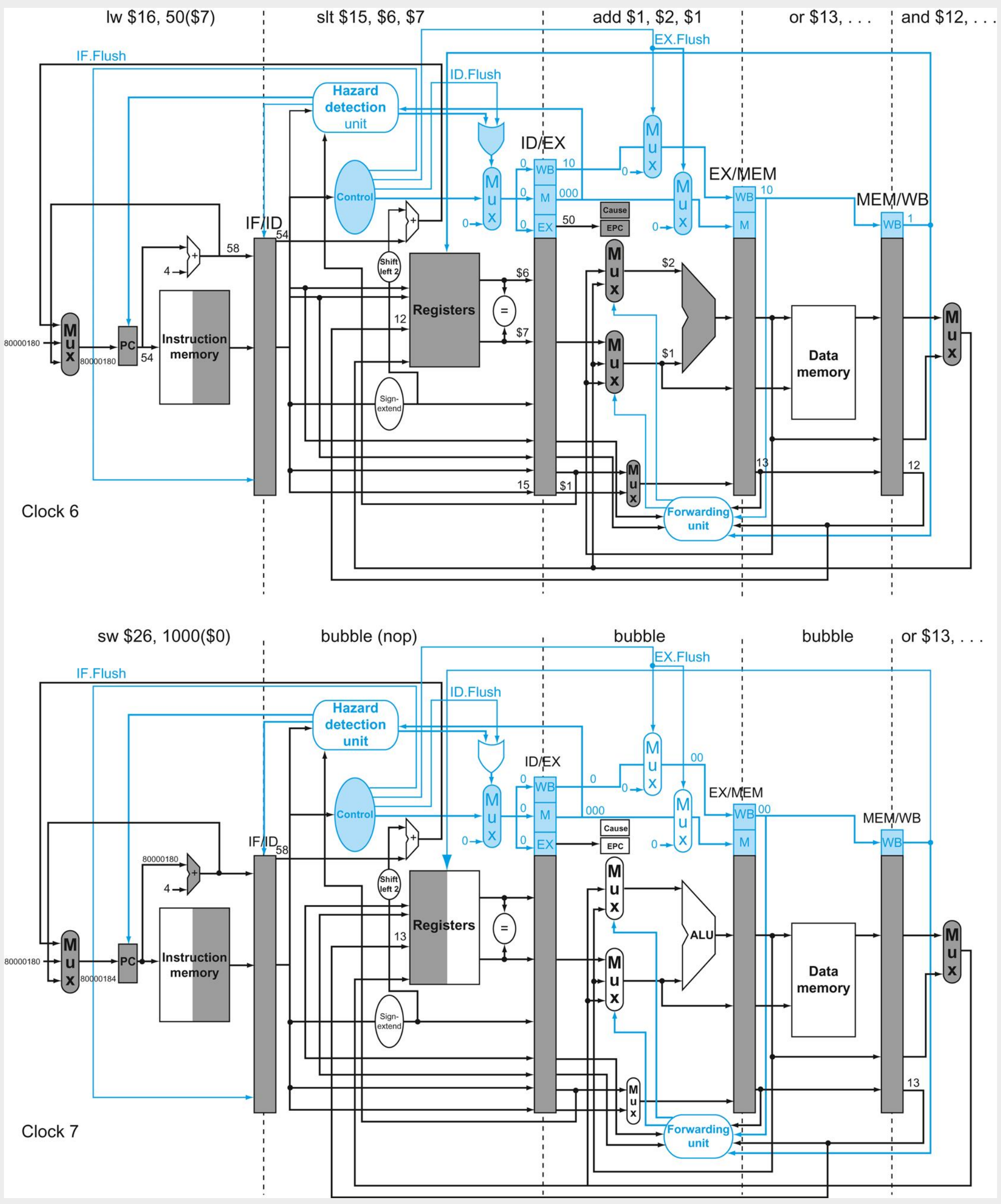


FIGURE 4.67 **The result of an exception due to arithmetic overflow in the add instruction.** The overflow is detected during the EX stage of clock 6, saving the address following the add in the EPC register (4C + 4 = 50_{hex}). Overflow causes all the Flush signals to be set near the end of this clock 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 338, 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 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.)

precise interrupt

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

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

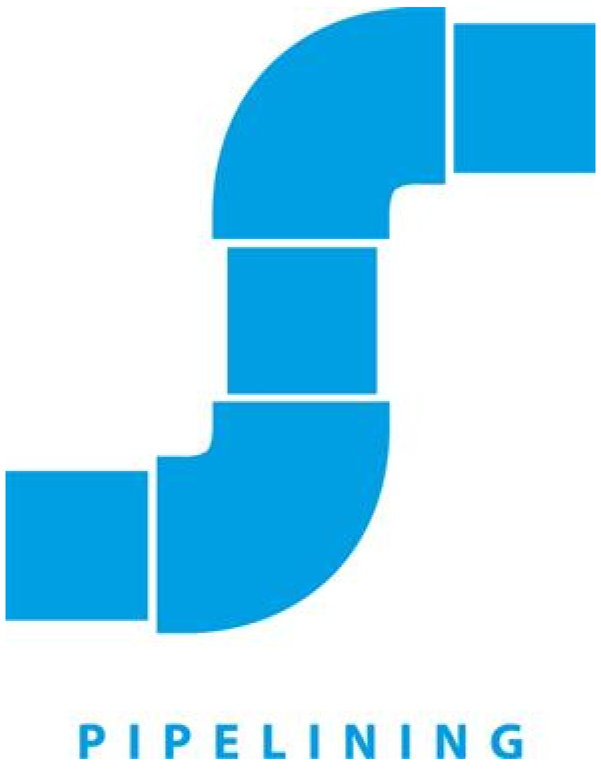
Which exception should be recognized first in this sequence?

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

4.11 Parallelism via Instructions

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*, sixth 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 have the same duration, 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.



PARALLELISM

instruction-level parallelism
The parallelism among instructions.

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

Another approach is to replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage. The general name for this technique is **multiple issue**. A multiple-issue laundry would replace our household washer and dryer with, say, three washers and three dryers. You would also have to recruit more assistants to fold and put away three times as much laundry in the same amount of time. The downside is the extra work to keep all the machines busy and transferring the loads to the next pipeline stage.

Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate or, stated alternatively, the CPI to be less than 1. As mentioned in Chapter 1, it is sometimes useful to flip the metric and use *IPC*, or *instructions per clock cycle*. Hence, a 4 GHz four-way multiple-issue microprocessor can execute a peak rate of 16 billion instructions per second and have a best-case CPI of 0.25, or an IPC of 4. Assuming a five-stage pipeline, such a processor would have 20 instructions in execution at any given time. Today's high-end microprocessors attempt to issue from three to six instructions in every clock cycle. Even moderate designs will aim at an IPC of 2. There are typically, however, many constraints on what types of instructions may be executed simultaneously, and what happens when dependences arise.

There are two major ways to implement a multiple-issue processor, with the major difference being the division of work between the compiler and the hardware. Because the division of work dictates whether decisions are being made statically (that is, at compile time) or dynamically (that is, during execution), the approaches are sometimes called **static multiple issue** and **dynamic multiple issue**. As we will see, both approaches have other, more commonly used names, which may be less precise or more restrictive.

static multiple issue

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

dynamic multiple issue

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

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

1. Packaging instructions into **issue slots**: how does the processor determine how many instructions and which instructions can be issued in a given clock cycle? In most static issue processors, this process is at least partially handled by the compiler; in dynamic issue designs, it is normally dealt with at runtime by the processor, although the compiler will often have already tried to help improve the issue rate by placing the instructions in a beneficial order.

2. Dealing with data and control hazards: in static issue processors, the compiler handles some or all of the consequences of data and control hazards statically. In contrast, most dynamic issue processors attempt to alleviate at least some classes of hazards using hardware techniques operating at execution time.

issue slots

The positions from which instructions could issue in a given clock cycle; by analogy, these correspond to positions at the starting blocks for a sprint.

Although we describe these as distinct approaches, in reality one approach often borrows techniques from the other, and neither approach can claim to be perfectly pure.

The Concept of Speculation

speculation

An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

One of the most important methods for finding and exploiting more ILP is speculation. Based on the great idea of **prediction**, **speculation** is an approach that allows the compiler or the processor to "guess" about the properties of an instruction, so as to enable execution to begin for other instructions that may depend on the speculated instruction. For example, we might speculate on the outcome of a branch, so that instructions after the branch could be executed earlier. Another example is that we might speculate that a store that precedes a load does not refer to the same address, which would allow the load to be executed before the store. The difficulty with speculation is that it may be wrong. So, any speculation mechanism must include both a method to check if the guess was right and a method to unroll or back out the effects of the instructions that were executed speculatively. The implementation of this back-out capability adds complexity.

Speculation may be done in the compiler or by the hardware. For example, the compiler can use speculation to reorder instructions, moving an instruction across a branch or a load across a store. The processor hardware can perform the same transformation at runtime using techniques we discuss later in this section.

The recovery mechanisms used for incorrect speculation are rather different. In the case of speculation in software, the compiler usually inserts additional instructions that check the accuracy of the speculation and provide a fix-up routine to use when the speculation is incorrect. In hardware speculation, the processor usually buffers the speculative results until it knows they are no longer speculative. If the speculation is correct, the instructions are completed by allowing the contents of the buffers to be written to the registers or memory. If the speculation is incorrect, the hardware flushes the buffers and re-executes the correct instruction sequence.

Speculation introduces one other possible problem: speculating on certain instructions may introduce exceptions that were formerly not present. For example, suppose a load instruction is moved in a speculative manner, but the address it uses is not legal when the speculation is incorrect. The result would be an exception that should not have occurred. The problem is complicated by the fact that if the load instruction were not speculative, then the exception must occur! In compiler-based speculation, such problems are avoided by adding special speculation support that allows such exceptions to be ignored until it is clear that they really should occur. In hardware-based speculation, exceptions are simply buffered until it is clear that the instruction causing them is no longer speculative and is ready to complete; at that point the exception is raised, and nor-mal exception handling proceeds.

Since speculation can improve performance when done properly and decrease performance when done carelessly, significant effort goes into deciding when it is appropriate to speculate. Later in this section, we will examine both static and dynamic techniques for speculation.

Static Multiple Issue

issue packet

The set of instructions that issues together in one clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

Static multiple-issue processors all use the compiler to assist with packaging instructions and handling hazards. In a static issue processor, you can think of the set of instructions issued in a given clock cycle, which is called an **issue packet**, as one large instruction with multiple operations. This view is more than an analogy. Since a static multiple-issue processor usually restricts what mix of instructions can be initiated in a given clock cycle, it is useful to think of the issue packet as a single instruction allowing several operations in certain predefined fields. This view led to the original name for this approach: **Very Long Instruction Word (VLIW)**.

Very Long Instruction Word (VLIW)

A style of instruction set architecture that launches many operations that are defined to be independent in a single wide instruction, typically with many separate opcode fields.

Most static issue processors also rely on the compiler to take on some responsibility for handling data and control hazards. The compiler's responsibilities may include static branch prediction and code scheduling to reduce or prevent all hazards. Let's look at a simple static issue version of a MIPS processor, before we describe the use of these techniques in more aggressive processors.

An Example: Static Multiple Issue with the MIPS ISA

To give a flavor of static multiple issue, we consider a simple two-issue MIPS processor, where one of the instructions can be an integer ALU operation or branch and the other can be a load or store. Such a design is like that used in some embedded MIPS processors. Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions. In many static multiple-issue processors, and essentially all VLIW processors, the layout of simultaneously issuing instructions is restricted to simplify the decoding and instruction issue. Hence, we will require that the instructions be paired and aligned on a 64-bit boundary, with the ALU or branch portion appearing first. Furthermore, if one instruction of the pair cannot be used, we require that it be replaced with a `nop`. Thus, the instructions always issue in pairs, possibly with a `nop` in one slot. Figure 4.68 shows how the instructions look as they go into the pipeline in pairs.

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

FIGURE 4.68 **Static two-issue pipeline in operation.**
The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

Static multiple-issue processors vary in how they deal with potential data and control hazards. In some designs, the compiler takes full responsibility for removing *all* hazards, scheduling the code and inserting no-ops so that the code executes without any need for hazard detection or hardware-generated stalls. In others, the hardware detects data hazards and generates stalls between two issue packets, while requiring that the compiler avoid all dependences within an instruction pair. Even so, a hazard generally forces the entire issue packet containing the dependent instruction to stall. Whether the software must handle all hazards or only try to reduce the fraction of hazards between separate issue packets, the appearance of having a large single instruction with multiple operations is reinforced. We will assume the second approach for this example.

To issue an ALU and a data transfer operation in parallel, the first need for additional hardware—beyond the usual hazard detection and stall logic—is extra ports in the register file (see Figure 4.69). In one clock cycle we may need to read two registers for the ALU operation and two more for a store, and also one write port for an ALU operation and one write port for a load. Since the ALU is tied up for the ALU operation, we also need a separate adder to calculate the effective address for data transfers. Without these extra resources, our two-issue pipeline would be hindered by structural hazards.

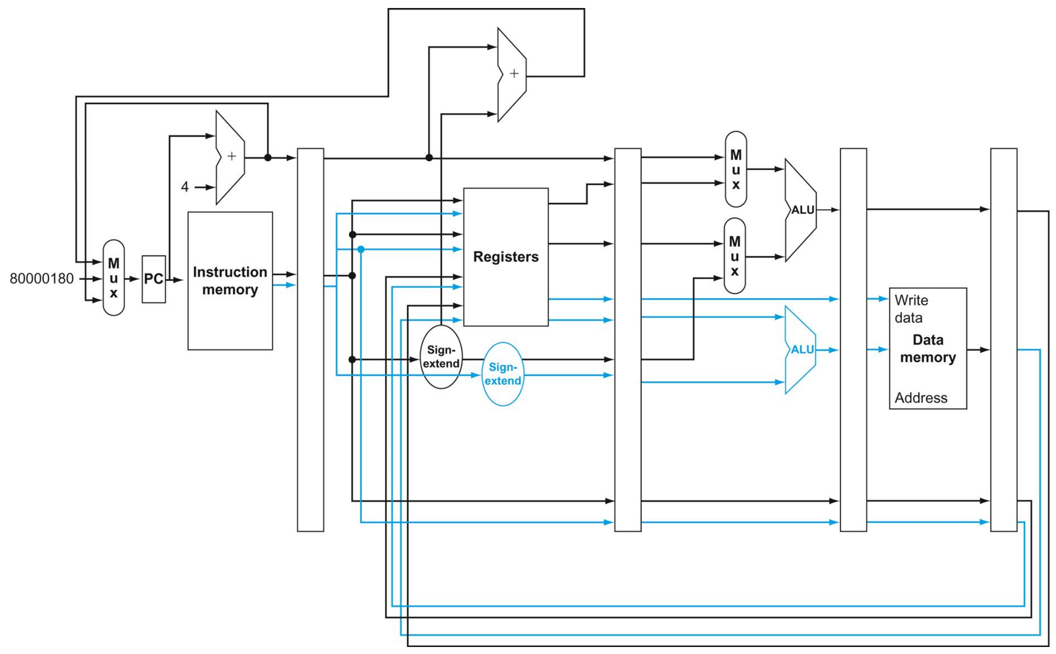


FIGURE 4.69 **A static two-issue datapath.** The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

Clearly, this two-issue processor can improve performance by up to a factor of 2. Doing so, however, requires that twice as many instructions be overlapped in execution, and this additional overlap increases the relative performance loss from data and control hazards. For example, in our simple five-stage pipeline, loads have a **use latency** of one clock cycle, which prevents one instruction from using the result without stalling. In the two-issue, five-stage pipeline the result of a load instruction cannot be used on the next *clock cycle.* This means that the next *two* instructions cannot use the load result without stalling. Furthermore, ALU instructions that had no use latency in the simple five-stage pipeline now have a one-instruction use latency, since the results cannot be used in the paired load or store. To effectively exploit the parallelism available in a multiple-issue processor, more ambitious compiler or hardware scheduling techniques are needed, and static multiple issue requires that the compiler take on this role.

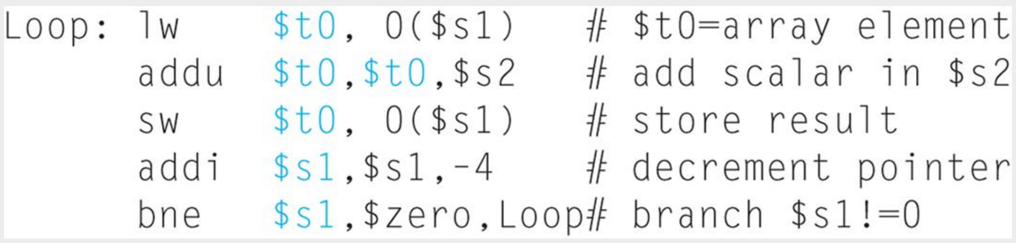
use latency

Number of clock cycles between a load instruction and an instruction that can use the result of the load without stalling the pipeline.

Simple Multiple-Issue Code Scheduling

Example

How would this loop be scheduled on a static two-issue pipeline for MIPS?



```
Loop: lw    $t0, 0($s1)    # $t0=array element
      addu  $t0,$t0,$s2    # add scalar in $s2
      sw    $t0, 0($s1)    # store result
      addi  $s1,$s1,-4     # decrement pointer
      bne   $s1,$zero,Loop# branch $s1!=0
```

Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

Answer

The first three instructions have data dependences, and so do the last two. Figure 4.70 shows the best schedule for these instructions. Notice that just one pair of instructions has both issue slots used. It takes four clocks per loop iteration; at four clocks to execute five instructions, we get the disappointing CPI of 0.8 versus the best case of 0.5., or an IPC of 1.25 versus 2.0. Notice that in computing CPI or IPC, we do not count any nops executed as useful instructions. Doing so would improve CPI, but not performance!

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

FIGURE 4.70 **The scheduled code as it would look on a two-issue MIPS pipeline.**
The empty slots are no-ops.

An important compiler technique to get more performance from loops is **loop unrolling**, where multiple copies of the loop body are made. After unrolling, there is more ILP available by overlapping instructions from different iterations.

loop unrolling

A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

Loop Unrolling for Multiple-Issue Pipelines

Example

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

Answer

To significantly reduce the delays in the loop, we need to make four copies of the loop body. After unrolling and eliminating the unnecessary loop overhead instructions, the loop will contain four copies each of `lw`, `add`, and `sw`, plus one `addi` and one `bne`. Figure 4.71 shows the unrolled and scheduled code.

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi $s1,$s1,-16	lw $t0, 0($s1)	1
		lw $t1,12($s1)	2
	addu $t0,$t0,$s2	lw $t2, 8($s1)	3
	addu $t1,$t1,$s2	lw $t3, 4($s1)	4
	addu $t2,$t2,$s2	sw $t0, 16($s1)	5
	addu $t3,$t3,$s2	sw $t1,12($s1)	6
		sw $t2, 8($s1)	7
	bne $s1,$zero,Loop	sw $t3, 4($s1)	8

FIGURE 4.71 **The unrolled and scheduled code of Figure 4.70 as it would look on a static two-issue MIPS pipeline.**
The empty slots are no-ops. Since the first instruction in the loop decrements $s1 by 16, the addresses loaded are the original value of $s1, then that address minus 4, minus 8, and minus 12.

During the unrolling process, the compiler introduced additional registers (`$t1, $t2, $t3`). The goal of this process, called **register renaming**, is to eliminate dependences that are not true data dependences, but could either lead to potential hazards or prevent the compiler from flexibly scheduling the code. Consider how the unrolled code would look using only `$t0`. There would be repeated instances of `lw $t0,0($$s1)`, addu `$t0,$t0,$s2` followed by `sw t0,4($s1)`, but these sequences, despite using `$t0`, are actually completely independent—no data values flow between one set of these instructions and the next set. This case is what is called an **antidependence** or **name dependence**, which is an ordering forced purely by the reuse of a name, rather than a real data dependence that is also called a true dependence.

register renaming

The renaming of registers by the compiler or hardware to remove antidependences.

antidependence

Also called **name dependence**. An ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two instructions.

Renaming the registers during the unrolling process allows the compiler to move these independent instructions subsequently so as to better schedule the code. The renaming process eliminates the name dependences, while preserving the true dependences.

Notice now that 12 of the 14 instructions in the loop execute as pairs. It takes 8 clocks for 4 loop iterations, or 2 clocks per iteration, which yields a CPI of 8/14 = 0.57 and an IPC of 1.75. Loop unrolling and scheduling with dual issue gave us an improvement factor of almost 2, partly from reducing the loop control instructions and partly from dual issue execution. The cost of this performance improvement is using four temporary registers rather than one, as well as a significant increase in code size.

Dynamic Multiple-Issue Processors

Dynamic multiple-issue processors are also known as **superscalar** processors, or simply superscalars. In the simplest superscalar processors, instructions issue in order, and the processor decides whether zero, one, or more instructions can issue in a given clock cycle. Obviously, achieving good performance on such a processor still requires the compiler to try to schedule instructions to move dependences apart and thereby improve the instruction issue rate. Even with such compiler scheduling, there is an important difference between this simple superscalar and a VLIW processor: the code, whether scheduled or not, is guaranteed by the hardware to execute correctly. Furthermore, compiled code will always run correctly independent of the issue rate or pipeline structure of the processor. In some VLIW designs, this has not been the case, and recompilation was required when moving across different processor models; in other static issue processors, code would run correctly across different implementations, but often so poorly as to make compilation effectively required.

superscalar

An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution.

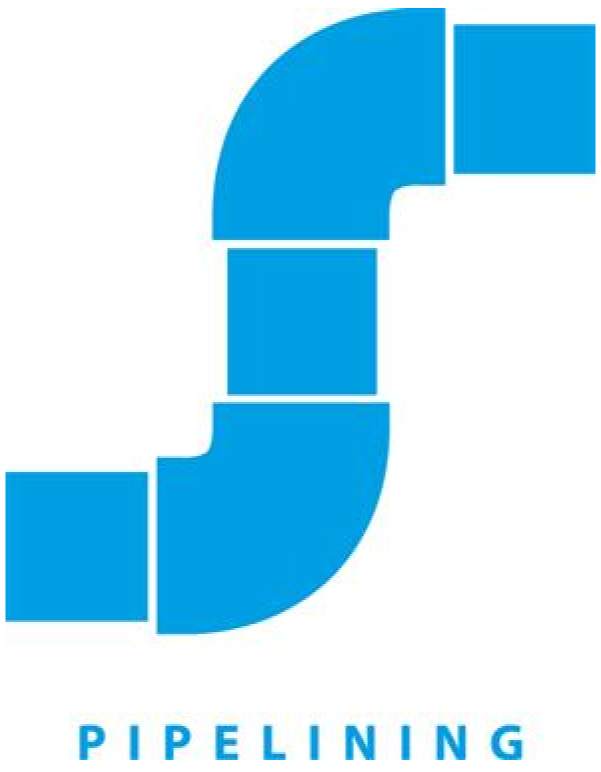
Many superscalars extend the basic framework of dynamic issue decisions to include **dynamic pipeline scheduling**. Dynamic pipeline scheduling chooses which instructions to execute in a given clock cycle while trying to avoid hazards and stalls. Let's start with a simple example of avoiding a data hazard. Consider the following code sequence:

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

dynamic pipeline scheduling

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

Even though the `sub` instruction is ready to execute, it must wait for the `lw` and `addu` to complete first, which might take many clock cycles if memory is slow. (Chapter 5 explains cache misses, the reason that memory accesses are sometimes very slow.) Dynamic **pipeline** scheduling allows such hazards to be avoided either fully or partially.



Dynamic Pipeline Scheduling

Dynamic pipeline scheduling chooses which instructions to execute next, possibly reordering them to avoid stalls. In such processors, the pipeline is divided into three major units: an instruction fetch and issue unit, multiple functional units (a dozen or more in high-end designs in 2020), and a **commit unit**. Figure 4.72 shows the model. The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution. Each functional unit has buffers, called **reservation stations**, which hold the operands and the operation. (In the next section, we will discusses an alternative to reservation stations used by many recent processors.) As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated. When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit, which buffers the result until it is safe to put the result into the register file or, for a store, into memory. The buffer in the commit unit, often called the **reorder buffer**, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline. Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.

commit unit

The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer-visible registers and memory.

reservation station

A buffer within a functional unit that holds the operands and the operation.

reorder buffer

The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register.

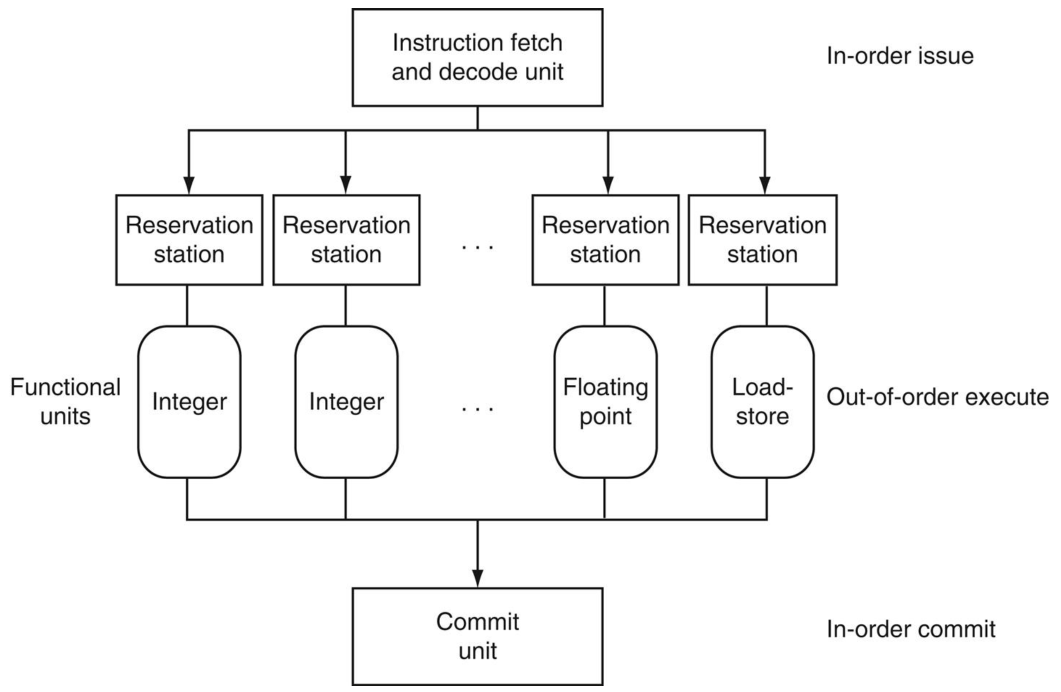


FIGURE 4.72 The three primary units of a dynamically scheduled pipeline.
The final step of updating the state is also called retirement or graduation.

The combination of buffering operands in the reservation stations and results in the reorder buffer provides a form of register renaming, just like that used by the compiler in our earlier loop-unrolling example on page 350. To see how this conceptually works, consider the following steps:

1. When an instruction issues, it is copied to a reservation station for the appropriate functional unit. Any operands that are available in the register file or reorder buffer are also immediately copied into the reservation station. The instruction is buffered in the reservation station until all the operands and the functional unit are available. For the issuing instruction, the register copy of the operand is no longer required, and if a write to that register occurred, the value could be overwritten.
2. If an operand is not in the register file or reorder buffer, it must be waiting to be produced by a functional unit. The name of the functional unit that will produce the result is tracked. When that

unit eventually produces the result, it is copied directly into the waiting reservation station from the functional unit bypassing the registers.

These steps effectively use the reorder buffer and the reservation stations to implement register renaming.

Conceptually, you can think of a dynamically scheduled pipeline as analyzing the data flow structure of a program. The processor then executes the instructions in some order that preserves the data flow order of the program. This style of execution is called an **out-of-order execution**, since the instructions can be executed in a different order than they were fetched.

out-of-order execution

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

To make programs behave as if they were running on a simple in-order pipeline, the instruction fetch and decode unit is required to issue instructions in order, which allows dependences to be tracked, and the commit unit is required to write results to registers and memory in program fetch order. This conservative mode is called **in-order commit**. Hence, if an exception occurs, the computer can point to the last instruction executed, and the only registers updated will be those written by instructions before the instruction causing the exception. Although the front end (fetch and issue) and the back end (commit) of the pipeline run in order, the functional units are free to initiate execution whenever the data they need is available. Today, all dynamically scheduled pipelines use in-order commit.

in-order commit

A commit in which the results of pipelined execution are written to the programmer visible state in the same order that instructions are fetched.

Dynamic scheduling is often extended by including hardware-based speculation, especially for branch outcomes. By predicting the direction of a branch, a dynamically scheduled processor can continue to fetch and execute instructions along the predicted path. Because the instructions are committed in order, we know whether or not the branch was correctly predicted before any instructions from the predicted path are committed. A speculative, dynamically scheduled pipeline can also support speculation on load addresses, allowing load-store reordering, and using the commit unit to avoid incorrect speculation. In the next section, we will look at the use of dynamic scheduling with speculation in the Intel Core i7 design.

Hardware/Software Interface

Out-of-order execution creates new pipeline hazards that we did not see in the earlier pipelines. A *name dependence* occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name. There are two types of name dependences between an instruction i that precedes instruction j in program order:

1. An *antidependence* between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i reads the correct value.
2. An *output dependence* occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

Our original pipeline hazard was the result of what is called a *true data dependence*.

For example, in this code below there is an antidependence between `swc1` and `addiu` on register x1 and a true data dependence between `lwc1` and `add.s` on register f0. While there are no output dependencies between instructions in a single loop, there are between different iterations of the loop, for example, between the `addiu` instructions of the first and second iterations.

Loop: lwc1 $f0,0(x1) # f0=array elementadd.s $f4,$f0,$f2 # add scalar in f2swc1 $f4,0(x1) # store resultaddiu x1,x1,4 # decrement pointer 8 bytesbne x1,x2,Loop # branch if x1 != x2

A pipeline hazard exists whenever there is a name or data dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence. They lead to these more intuitive names of pipeline hazards:

1. An *antidependence* can lead to a Write-After-Read (WAR) hazard.
2. An *output dependence* can lead to a Write-After-Write (WAW) hazard.
3. A *true data dependence* or a Read-After-Write (RAW) hazard.

We do not have hazards for WAR or WAW hazards in our earlier pipelines because all instructions execute in order and the writes occur only in the last pipeline stage for register–register instructions and always in the same pipeline stage for data access in load and store instructions.

Understanding Program Performance

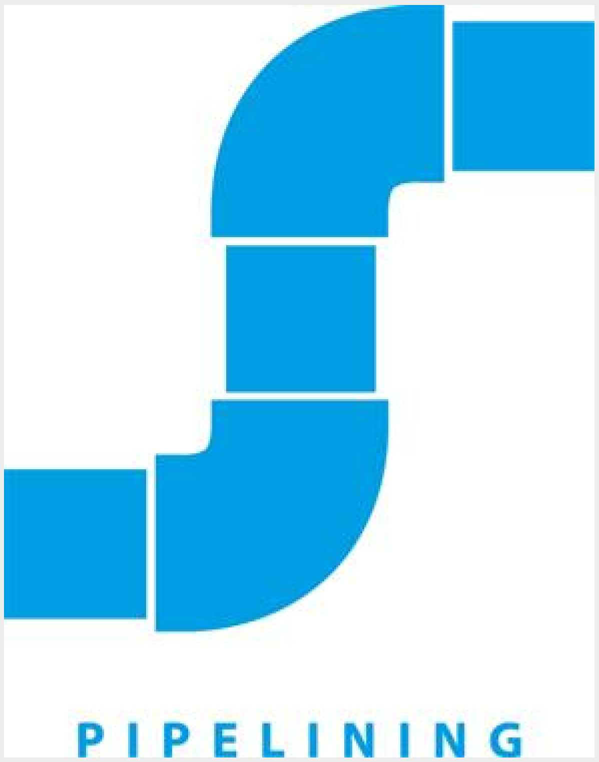
Given that compilers can also schedule code around data dependences, you might ask why a superscalar processor would use dynamic scheduling. There are three major reasons. First, not all stalls are predictable. In particular, cache misses (see Chapter 5) in the **memory hierarchy** cause unpredictable stalls. Dynamic scheduling allows the processor to hide some of those stalls by continuing to execute instructions while waiting for the stall to end.

Second, if the processor speculates on branch outcomes using dynamic branch **prediction**, it cannot know the exact order of instructions at compile time, since it depends on the predicted and actual behavior of branches. Incorporating dynamic speculation to exploit more *instruction-level parallelism* (ILP) without incorporating dynamic scheduling would significantly restrict the benefits of speculation.

Third, as the pipeline latency and issue width change from one implementation to another, the best way to compile a code sequence also changes. For example, how to schedule a sequence of dependent instructions is affected by both issue width and latency. The pipeline structure affects both the number of times a loop must be unrolled to avoid stalls as well as the process of compiler-based register renaming. Dynamic scheduling allows the hardware to hide most of these details. Thus, users and software distributors do not need to worry about having multiple versions of a program for different implementations of the same instruction set. Similarly, old legacy code will get much of the benefit of a new implementation without the need for recompilation.

The BIG Picture

Both **pipelining** and multiple-issue execution increase peak instruction throughput and attempt to exploit instruction-level **parallelism** (ILP). Data and control dependences in programs, however, offer an upper limit on sustained performance because the processor must sometimes wait for a dependence to be resolved. Software-centric approaches to exploiting ILP rely on the ability of the compiler to find and reduce the effects of such dependences, while hardware-centric approaches rely on extensions to the pipeline and issue mechanisms. Speculation, performed by the compiler or the hardware, can increase the amount of ILP that can be exploited via **prediction**, although care must be taken since speculating incorrectly is likely to reduce performance.



PARALLELISM

Hardware/Software Interface

Modern, high-performance microprocessors are capable of issuing several instructions per clock; unfortunately, sustaining that issue rate is very difficult. For example, despite the existence of processors with four to six issues per clock, very few applications can sustain more than two instructions per clock. There are two primary reasons for this.

First, within the pipeline, the major performance bottlenecks arise from dependences that cannot be alleviated, thus reducing the parallelism among instructions and the sustained issue rate. Although little can be done about true data dependences, often the compiler or hardware does not know precisely whether a dependence exists or not, and so must conservatively assume the dependence exists. For example, code that makes use of pointers, particularly in ways that may lead to aliasing, will lead to more implied potential dependences. In contrast, the greater regularity of array accesses often allows a compiler to deduce that no dependences exist. Similarly, branches that cannot be accurately predicted whether at runtime or compile time will limit the ability to exploit ILP. Often, additional ILP is available, but the ability of the compiler or the hardware to find ILP that may be widely separated (sometimes by the execution of thousands of instructions) is limited.

Second, losses in the **memory hierarchy** (the topic of Chapter 5) also limit the ability to keep the pipeline full. Some memory system stalls can be hidden, but limited amounts of ILP also limit the extent to which such stalls can be hidden.

Energy Efficiency and Advanced Pipelining

The downside to the increasing exploitation of instruction-level parallelism via dynamic multiple issue and speculation is potential energy inefficiency. Each innovation was able to turn more transistors into performance, but they often did so very ineffectively. Now that we have hit the power wall, we are seeing designs with multiple processors per chip where the processors are not as deeply pipelined or as aggressively speculative as the predecessors.

The belief is that while the simpler processors are not as fast as their sophisticated brethren, they deliver better performance per joule, so that they can deliver more performance per chip when designs are constrained more by energy than they are by number of transistors.

Figure 4.73 shows the number of pipeline stages, the issue width, speculation level, clock rate, cores per chip, and power of several past and recent microprocessors. Note the drop in pipeline stages and power as companies switch to multicore designs.

Elaboration

A commit unit controls updates to the register file *and* memory. Some dynamically scheduled processors update the register file immediately during execution, using extra registers to implement the renaming function and preserving the older copy of a register until the instruction updating the register is no longer speculative. Other processors buffer the result, typically in a structure called a reorder buffer, and the actual update to the register file occurs later as part of the commit. Stores to memory must be buffered until commit time either in a *store buffer* (see Chapter 5) or in the reorder buffer. The commit unit allows the store to write to memory from the buffer when the buffer has a valid address and valid data, and when the store is no longer dependent on predicted branches.

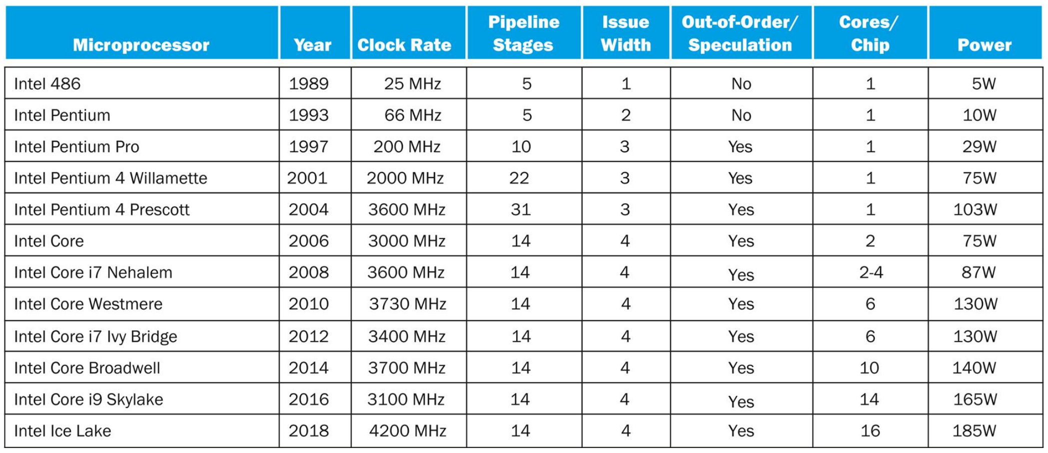
Elaboration

Memory accesses benefit from *nonblocking caches*, which continue servicing cache accesses during a cache miss (see Chapter 5). Out-of-order execution processors need the cache design to allow instructions to execute during a miss.

Check Yourself

State whether the following techniques or components are associated primarily with a software- or hardware-based approach to exploiting ILP. In some cases, the answer may be both.

1. Branch prediction
2. Multiple issue
3. VLIW
4. Superscalar
5. Dynamic scheduling
6. Out-of-order execution
7. Speculation
8. Reorder buffer
9. Register renaming



Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5W
Intel Pentium	1993	66 MHz	5	2	No	1	10W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103W
Intel Core	2006	3000 MHz	14	4	Yes	2	75W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185W

FIGURE 4.73 **Record of Intel Microprocessors in terms of pipeline complexity, number of cores, and power.**
The Pentium 4 pipeline stages do not include the commit stages. If we included them, the Pentium 4 pipelines would be even deeper.

4.12 Putting It All Together: The Intel Core i7 6700 and ARM Cortex-A53

In this section, we explore the design of two multiple issue processors: the ARM Cortex-A53 core, which is used as the basis for several tablets and cell phones, and the Intel Core i7 6700, a high-end, dynamically scheduled, speculative processor intended for high-end desktops and server applications. We begin with the simpler processor. This section is based on Section 3.12 of *Computer Architecture: A Quantitative Approach*, sixth edition.

The ARM Cortex-A53

The A53 is a dual-issue, statically scheduled superscalar with dynamic issue detection, which allows the processor to issue two instructions per clock. Figure 4.74 shows the basic pipeline structure of the pipeline. For nonbranch, integer instructions, there are eight stages: F1, F2, D1, D2, D3/ISS, EX1, EX2, and WB, as described in the caption. The pipeline is in order, so an instruction can initiate execution only when its results are available and when proceeding instructions have initiated. Thus, if the next two instructions are dependent, both can proceed to the appropriate execution pipeline, but they will be serialized when they get to the beginning of that pipeline. When the pipeline issue logic indicates that the result from the first instruction is available, the second instruction can issue.

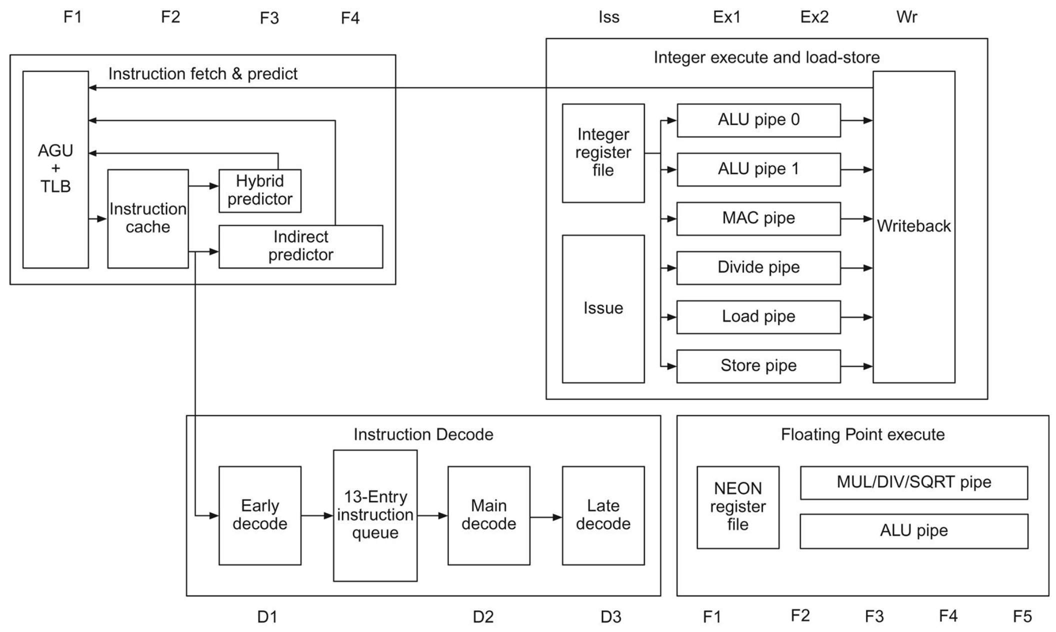


FIGURE 4.74 **The basic structure of the A53 integer pipeline is eight stages: F1 and F2 fetch the instruction, D1 and D2 do the basic decoding, and D3 decodes some more complex instructions and is overlapped with the first stage of the execution pipeline (ISS).** After ISS, the Ex1, EX2, and WB stages complete the integer pipeline. Branches use four different predictors, depending on the type. The floating-point execution pipeline is five cycles deep, in addition to the five cycles needed for fetch and decode, yielding 10 stages in total. AGU stands for Address Generation Unit and TLB for Transaction Lookaside Buffer (See Chapter 5). The NEON unit performs the ARM SIMD instructions of the same name. (From Hennessy JL, Patterson DA: Computer architecture: A quantitative approach, ed 6, Cambridge MA, 2018, Morgan Kaufmann.)

The four cycles of instruction fetch include an address generation unit that produces the next PC either by incrementing the last PC or from one of four predictors:

1. A single-entry branch target cache containing two instruction cache fetches (the next two instructions following the branch, assuming the prediction is correct). This target cache is checked during the first fetch cycle, if it hits; then the next two instructions are supplied from the target cache. In case of a hit and a correct prediction, the branch is executed with no delay cycles.
2. A 3072-entry hybrid predictor, used for all instructions that do not hit in the branch target cache, and operating during F3. Branches handled by this predictor incur a two-cycle delay.
3. A 256-entry indirect branch predictor that operates during F4; branches predicted by this predictor incur a three-cycle delay when predicted correctly.
4. An 8-deep return stack, operating during F4 and incurring a three-cycle delay.

Branch decisions are made in ALU pipe 0, resulting in a branch misprediction penalty of eight cycles. Figure 4.75 shows the misprediction rate for SPECint2006. The amount of work that is wasted depends on both the misprediction rate and the issue rate sustained during the time that the mispredicted branch was followed. As Figure 4.76 shows, wasted work generally follows the misprediction rate, though it may be larger or occasionally shorter.

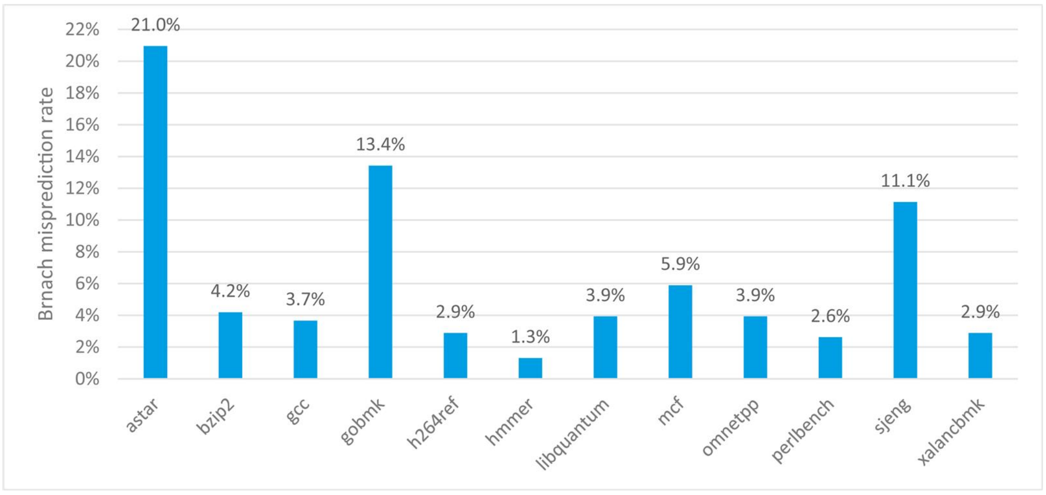


FIGURE 4.75 Misprediction rate of the A53 branch predictor for SPECint2006. (Adapted from Hennessy JL, Patterson DA: *Computer architecture: A quantitative approach*, ed 6, Cambridge MA, 2018, Morgan Kaufmann.)

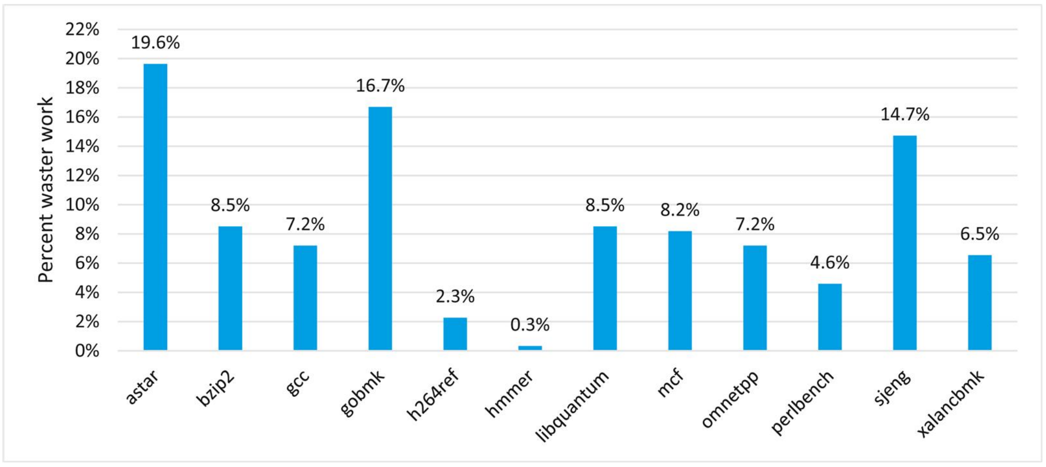


FIGURE 4.76 **Wasted work due to branch misprediction on the A53. Because the A53 is an in-order machine, the amount of wasted work depends on a variety of factors, including data dependences and cache misses, both of which will cause a stall.** (Adapted from Hennessy JL, Patterson DA: *Computer architecture: A quantitative approach*, ed 6, Cambridge MA, 2018, Morgan Kaufmann.)

Performance of the A53 Pipeline

The A53 has an ideal CPI of 0.5 because of its dual-issue structure. Pipeline stalls can arise from three sources:

1. Functional hazards, which occur because two adjacent instructions selected for issue simultaneously, use the same functional pipeline. Because the A53 is statically scheduled, the compiler should try to avoid such conflicts. When such instructions appear sequentially, they will be serialized at the beginning of the execution pipeline, when only the first instruction will begin execution.
2. Data hazards, which are detected early in the pipeline and may stall either both instructions (if the first cannot issue, the second is always stalled) or the second of a pair. Again, the compiler should try to prevent such stalls when possible.

3. Control hazards, which arise only when branches are mispredicted.

Both TLB misses (Chapter 5) and cache misses also cause stalls. Figure 4.77 shows the CPI and the estimated contributions from various sources.

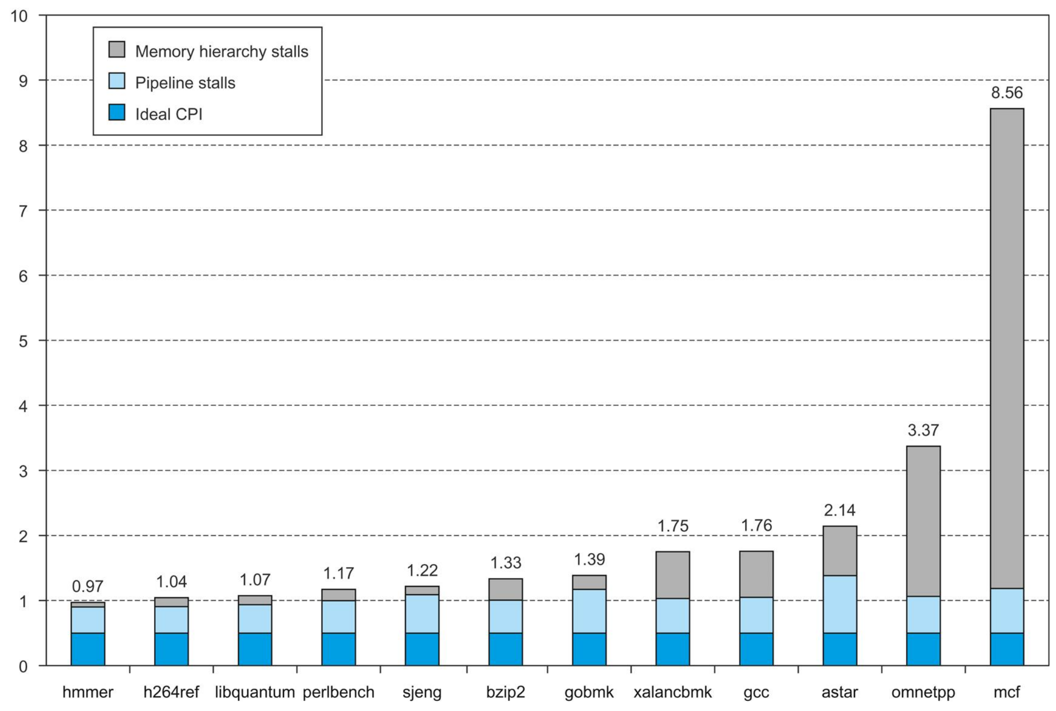


FIGURE 4.77 **The estimated composition of the CPI on the ARM A53 shows that pipeline stalls are significant but are outweighed by cache misses in the poorest performing programs (Chapter 5).**
These are subtracted from the CPI measured by a detailed simulator to obtain the pipeline stalls. Pipeline stalls include all three hazards. (From Hennessy JL, Patterson DA: *Computer architecture: A quantitative approach*, ed 6, Cambridge MA, 2018, Morgan Kaufmann.)

The A53 uses a shallow pipeline and a reasonably aggressive branch predictor, leading to modest pipeline losses, while allowing the processor to achieve high clock rates at modest power consumption. In comparison with the i7, the A53 consumes approximately 1/200 the power for a quad core processor!

Elaboration

The Cortex-A53 is a configurable core that supports the ARMv8 instruction set architecture. It is delivered as an IP (*Intellectual Property*) *core*. IP cores are the dominant form of technology delivery in the embedded, personal mobile device, and related markets; billions of ARM and MIPS processors have been created from these IP cores.

Note that IP cores are different than the cores in the Intel i7 multicore computers. An IP core (which may itself be a multicore) is designed to be incorporated with other logic (hence it is the “core” of a chip), including application-specific processors (such as an encoder or decoder for video), I/O interfaces, and memory interfaces, and then fabricated to yield a processor optimized for a particular application. Although the processor core is almost identical, the resultant chips have many differences. One parameter is the size of the L2 cache, which can vary by a factor of 16.

The Intel Core i7 6700

x86 microprocessors employ sophisticated pipelining approaches, using both dynamic multiple issue and dynamic pipeline scheduling with out-of-order execution and speculation for its 14-stage pipeline. These processors, however, are still faced with the challenge of implementing the complex x86 instruction set, described in Chapter 2. Intel fetches x86 instructions and translates them into internal MIPS-like instructions, which Intel calls *micro-operations*. The micro-operations are then executed by a sophisticated, dynamically scheduled, speculative pipeline capable of sustaining an execution rate of up to six micro-operations per clock cycle. This section focuses on that micro-operation pipeline.

When we consider the design of sophisticated, dynamically scheduled processors, the design of the functional units, the cache and register file, instruction issue, and overall pipeline control becomes intermingled, making it difficult to separate the datapath from the pipeline. Because of this interdependence, many engineers and researchers use the term **microarchitecture** to refer to the detailed internal architecture of a processor.

The Intel Core i7 uses a scheme for resolving antidependences and incorrect speculation that uses a reorder buffer together with register renaming. Register renaming explicitly renames the **architectural registers** in a processor (16 in the case of the 64-bit version of the x86 architecture) to a larger set of physical registers. The Core i7 uses register renaming to remove antidependences. Register renaming requires the processor to maintain a map between the architectural registers and the physical registers, indicating which physical register is the most current copy of an architectural register. By keeping track of the renamings that have occurred, register renaming offers another approach to recovery in the event of incorrect speculation: simply undo the mappings that have occurred since the first incorrectly speculated instruction. This adjustment will cause the state of the processor to return to the last correctly executed instruction, keeping the correct mapping between the architectural and physical registers.

Figure 4.78 shows the overall structure of the i7 pipeline. We will examine the pipeline by starting with instruction fetch and continuing on to instruction commit, following eight steps labeled in the figure.

1. Instruction fetch—The processor uses a sophisticated multilevel branch predictor to achieve a balance between speed and prediction accuracy. There is also a return address stack to speed up function return. Mispredictions cause a penalty of about 17 cycles. Using the predicted address, the instruction fetch unit fetches 16 bytes from the instruction cache.
2. The 16 bytes are placed in the predecode instruction buffer—The predecode stage also breaks the 16 bytes into individual x86 instructions. This predecode is nontrivial because the length of an x86 instruction can be from 1 to 17 bytes and the predecoder must look through a number of bytes before it knows the instruction length. Individualx86 instructions are placed into the instruction queue.

3. Micro-op decode—Three of the decoders handle x86 instructions that translate directly into one micro-operation. For x86 instructions that have more complex semantics, there is a microcode engine that is used to produce the micro-operation sequence; it can produce up to four micro-operations every cycle and continues until the necessary micro-operation sequence has been generated. The micro-operations are placed according to the order of the x86 instructions in the 64-entry micro-operation buffer.
4. The micro-operation buffer preforms *loop stream detection*—If there is a small sequence of instructions (less than 64 instructions) that comprises a loop, the loop stream detector will find the loop and directly issue the micro-operations from the buffer, eliminating the need for the instruction fetch and instruction decode stages to be activated.
5. Perform the basic instruction issue—Looking up the register location in the register tables, renaming the registers, allocating a reorder buffer entry, and fetching any results from the registers or reorder buffer before sending the micro-operations to the reservation stations. Up to four micro-operations can be processed every clock cycle; they are assigned the next available reorder buffer entries.
6. The i7 uses a centralized reservation station shared by six functional units. Up to six micro-operations may be dispatched to the functional units every clock cycle.
7. The individual function units execute the micro-operations, and then results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state once it is known that the instruction is no longer speculative. The entry corresponding to the instruction in the reorder buffer is marked as complete.
8. When one or more instructions at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.

Elaboration

Hardware in the second and fourth steps can combine or *fuse* operations together to reduce the number of operations that must be performed. *Macro-op fusion* in the second step takes x86 instruction combinations, such as compare followed by a branch, and fuses them into a single operation. *Microfusion* in the fourth step combines micro-operation pairs such as load/ALU operation and ALU operation/store and issues them to a single reservation station (where they can still issue independently), thus increasing the usage of the buffer. In a study of the Intel Core architecture, which also incorporated microfusion and macrofusion, Bird *et al.* [2007] discovered that microfusion had little impact on performance, while macrofusion appears to have a modest positive impact on integer performance and little impact on floating-point performance.

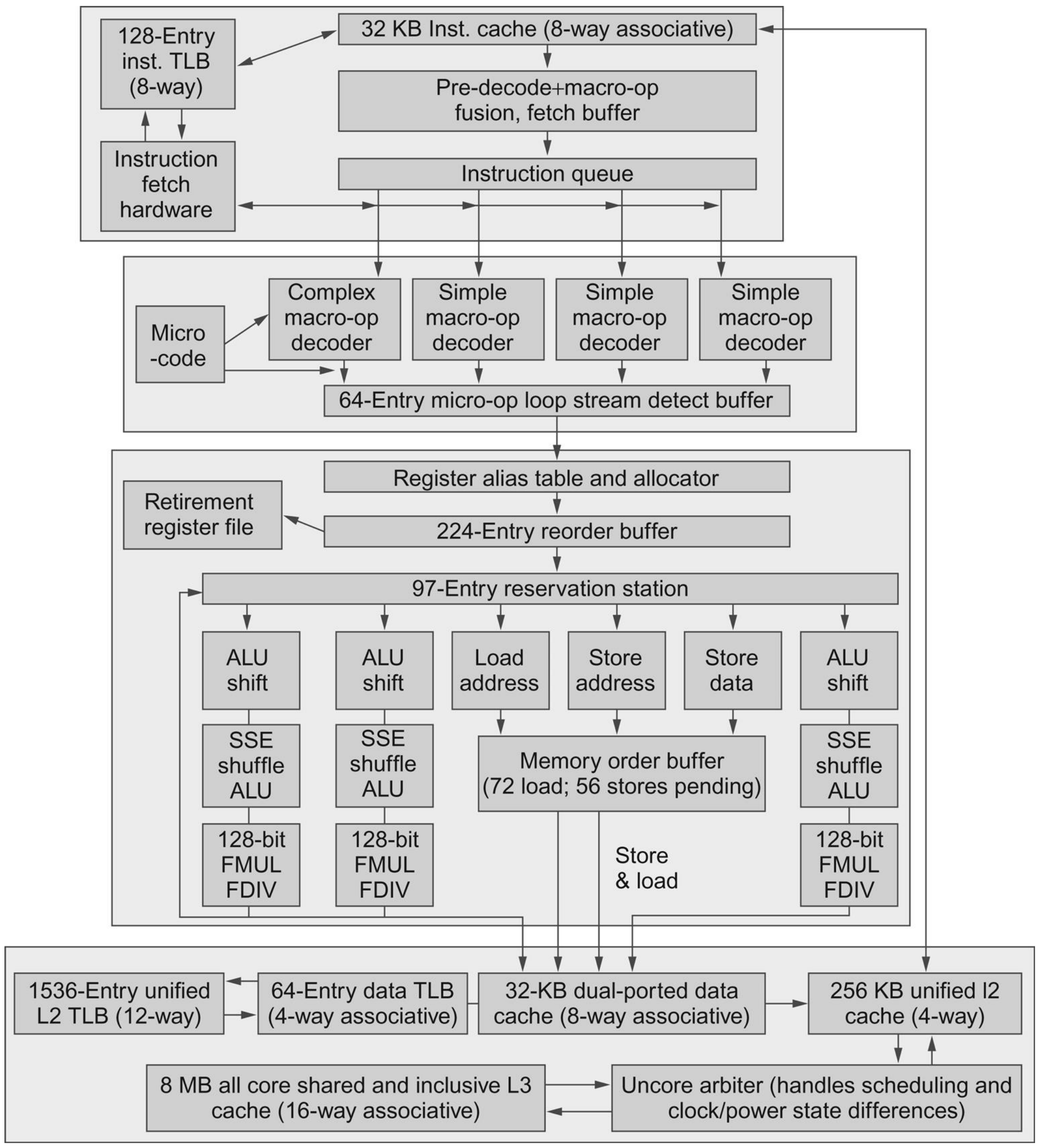

128-Entry inst. TLB (8-way)
32 KB Inst. cache (8-way associative)
Pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
Instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
64-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
224-Entry reorder buffer
97-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer (72 load; 56 stores pending)
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
1536-Entry unified L2 TLB (12-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (4-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

FIGURE 4.78 **The Intel Core i7 pipeline structure shown with the memory system components. The total pipeline depth is 14 stages, with branch mispredictions typically costing 17 cycles, with the extra few cycles likely due to the time to reset the branch predictor.** This design can buffer 72 loads and 56 stores. The six independent functional units can each begin execution of a ready microoperation in the same cycle. Up to four micro-operations can be processed in the register renaming table. The first

i7 processor was introduced in 2008; the i7 6700 is the sixth generation. The basic structure of the i7 is similar, but successive generations have enhanced performance by changing cache strategies (Chapter 5), increasing memory bandwidth, expanding the number of instructions in flight, enhancing branch prediction, and improving graphics support. (From Hennessy JL, Patterson DA: Computer architecture: A quantitative approach , ed 6, Cambridge MA, 2018, Morgan Kaufmann.)

Performance of the i7

Because of the presence of aggressive speculation, it is difficult to accurately attribute the gap between idealized performance and actual performance. The extensive queues and buffers on the 6700 reduce the probability of stalls because of a lack of reservation stations, renaming registers, or reorder buffers significantly.

Thus, most losses come either from branch mispredicts or cache misses. The cost of a branch mispredict is 17 cycles, whereas the cost of an L1 miss is about 10 cycles (Chapter 5). An L2 miss is slightly more than three times as costly as an L1 miss, and an L3 miss costs about 13 times what an L1 miss costs (130–135 cycles). Although the processor will attempt to find alternative instructions to execute during L2 and L3 misses, it is likely that some of the buffers will fill before a miss completes, causing the processor to stop issuing instructions.

Figure 4.79 shows the overall CPI for the 19 SPECCPUint2006 benchmarks. The average CPI on the i7 6700 is 0.71. Figure 4.80 shows the misprediction rate of the branch predictors of the Intel i7 6700. The misprediction rates are roughly half of those for the A53 in Figure 4.76—the median is 2.3% versus 3.9% for SPEC2006—and the CPI is less than half: the median is 0.64 versus 1.36 for the much more aggressive architecture. The clock rate is 3.4 GHz on the i7 versus up to 1.3 GHz for the A53, so the average instruction time is 0.64 × 1/3.4 GHz = 0.18 ns versus 1.36 × 1/1.3 GHz = 1.05 ns, or more than five times as fast. On the other hand, the i7 uses 200×as much power!

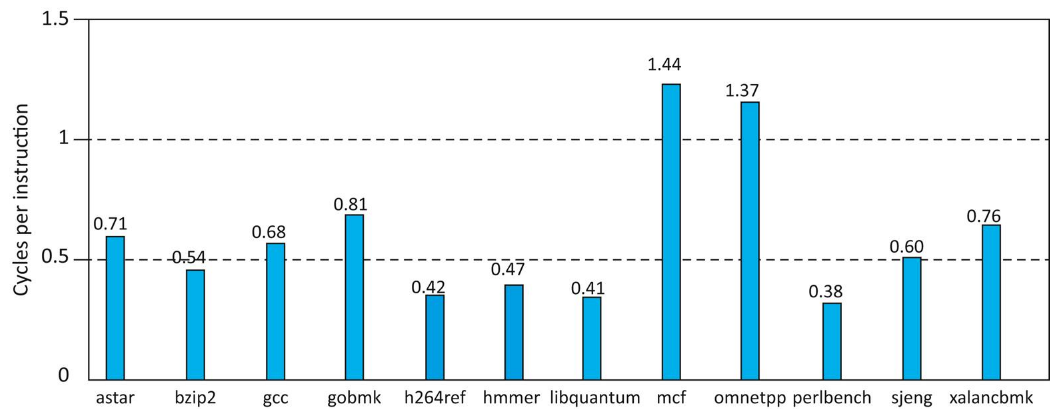


FIGURE 4.79 The CPI for the SPECCPUint2006 benchmarks on the i7 6700. The data in this section were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University.

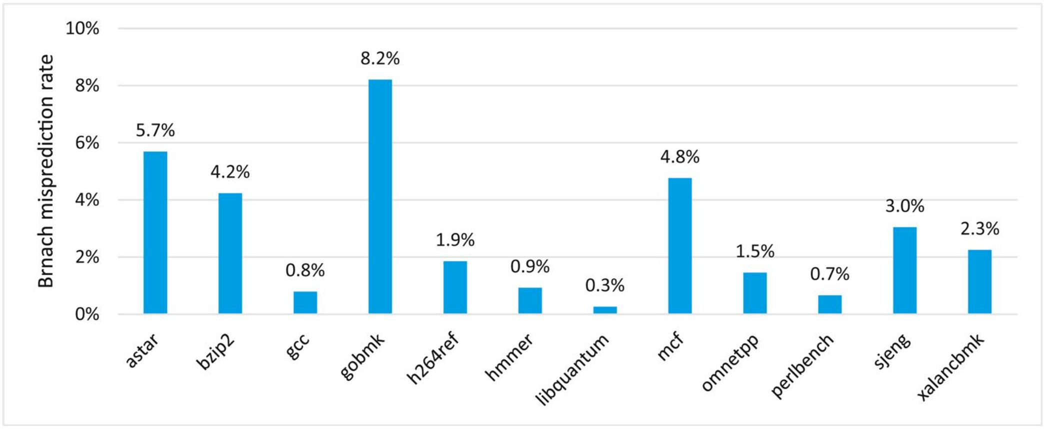


FIGURE 4.80 The misprediction rate for the integer SPECCPU2006 benchmarks on the Intel Core i7 6700. The misprediction rate is computed as the ratio of completed branches that are mispredicted versus all completed branches. (Adapted from Hennessy JL, Patterson DA: *Computer architecture: A quantitative approach*, ed 6, Cambridge MA, 2018, Morgan Kaufmann.)

The Intel Core i7 combines a 14-stage pipeline and aggressive multiple issue to achieve high performance. By keeping the latencies for back-to-back operations low, the impact of data dependences is reduced. What are the most serious potential performance bottlenecks for programs running on this processor? The following list includes some potential performance problems, the last three of which can apply in some form to any high-performance pipelined processor.

- The use of x86 instructions that do not map to a few simple micro-operations
- Branches that are difficult to predict, causing misprediction stalls and restarts when speculation fails
- Long dependences—typically caused by long-running instructions or the **memory hierarchy**—that lead to stalls
- Performance delays arising in accessing memory (see Chapter 5) that cause the processor to stall

HIERARCHY

4.13 Going Faster: Instruction-Level Parallelism and Matrix Multiply

Returning to the DGEMM example from Chapter 3, we can see the impact of instruction level parallelism by unrolling the loop so that the multiple issue, out-of-order execution processor has more instructions to work with. Figure 4.81 shows the unrolled version of Figure 3.21, which contains the C intrinsics to produce the AVX instructions.

```
1   #include <x86intrin.h>
2   #define UNROLL (4)
3
4   void dgemm (int n, double* A, double* B, double* C)
5   {
6     for (int i = 0; i < n; i+=UNROLL*8)
7       for (int j = 0; j < n; ++j){
8         __m512d c[UNROLL];
9         for (int r=0;r<UNROLL;r++)
10          c[r] =  _mm512_load_pd(C+i+r*8+j*n); //[ UNROLL];
11
12        for( int k = 0; k < n; k++ )
13        {
14          __m512d bb = _mm512_broadcastsd_pd(_mm_load_sd(B+j*n+k));
15          for (int r=0;r<UNROLL;r++)
16            c[r] = _mm512_fmadd_pd(_mm512_load_pd(A+n*k+r*8+i), bb, c[r]);
17       }
18
19       for (int r=0;r<UNROLL;r++)
20         _mm512_store_pd(C+i+r*8+j*n, c[r]);
21       }
22  }
```

FIGURE 4.81 **Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86 (Figure 3.21) and loop unrolling to create more opportunities for instruction-level parallelism.**
Figure 4.82 shows the assembly language produced by the compiler for the inner loop, which unrolls the three for-loop bodies to expose instruction level parallelism.

Like the unrolling example in Figure 4.71 above, we are going to unroll the loop 4 times. (We use the constant `UNROLL` in the C code to control the amount of unrolling in case we want to try other values.) Rather than manually unrolling the loop in C by making 4 copies of each of the intrinsics in Figure 3.21, we can rely on the gcc compiler to do the unrolling at − O3 optimization. We surround each intrinsic with a simple *for* loop with 4 iterations (lines 9, 15, and 19) and replace the scalar `c0` in Figure 3.22 with a 4-element array `c[]` (lines 8, 10, 16, and 20).

Figure 4.82 shows the assembly language output of the unrolled code. As expected, in Figure 4.82 there are 4 versions of each of the AVX instructions in Figure 3.22, with one exception. We only need 1 copy of the `vbroadcastsd` instruction, since we can use the eight copies of the B element in register `%zmm0` repeatedly throughout the loop. Thus the 4 AVX instructions in Figure 3.22 become 13 in Figure 4.82, and the 7 integer instructions appear in both, although the constants and addressing changes to account for the unrolling. Hence, despite unrolling 4 times, the number of instructions in the body of the loop only doubles: from 11 to 20.

```
vmovapd      (%r11),%zmm4                # Load 8 elements of C into %zmm4
mov          %rbx,%rcx                   # register %rcx = %rbx
xor          %eax,%eax                   # register %eax = 0
vmovapd      0x20(%r11),%zmm3            # Load 8 elements of C into %zmm3
vmovapd      0x40(%r11),%zmm2            # Load 8 elements of C into %zmm2
vmovapd      0x60(%r11),%zmm1            # Load 8 elements of C into %zmm1
vbroadcastsd (%rax,%r8,8),%zmm0          # Make 8 copies of B element in %zmm0
                                         # register %rax = %rax + 8
add          $0x8,%rax
vfmadd231pd  (%rcx),%zmm0,%zmm4          # Parallel mul & add %zmm0, %zmm4
vfmadd231pd  0x20(%rcx),%zmm0,%zmm3      # Parallel mul & add %zmm0, %zmm3
vfmadd231pd  0x40(%rcx),%zmm0,%zmm2      # Parallel mul & add %zmm0, %zmm2
vfmadd231pd  0x60(%rcx),%zmm0,%zmm1      # Parallel mul & add %zmm0, %zmm1
add          %r9,%rcx                    # register %rcx = %rcx
cmp          %r10,%rax                   # compare %r10 to %rax
jne          50 <dgemm+0x50>             # jump if not %r10 != %rax
add          $0x1, %esi                  # register % esi = % esi + 1
vmovapd      %zmm4, (%r11)               # Store %zmm4 into 8 C elements
vmovapd      %zmm3, 0x20(%r11)           # Store %zmm3 into 8 C elements
vmovapd      %zmm2, 0x40(%r11)           # Store %zmm2 into 8 C elements
vmovapd      %zmm1, 0x60(%r11)           # Store %zmm1 into 8 C elements
```

FIGURE 4.82 The x86 assembly language for the body of the nested loops generated by compiling the unrolled C code in Figure 4.81.

Unrolling nearly doubles performance. Optimizations for **subword parallelism** and **instruction level parallelism** result in an overall speedup of 4.4 versus the DGEMM in Figure 3.21. Compared to the Python version in Chapter 1, it is 4600 times as fast.

PARALLELISM

Elaboration

There are no pipeline stalls despite the reuse of register `%zmm5` in lines 9 to 12 of Figure 4.82 because the Intel Core i7 pipeline renames the registers.

Check Yourself

Are the following statements true or false?

1. The Intel Core i7 uses a multiple-issue pipeline to directly execute x86 instructions.
2. Both the A53 and the Core i7 use dynamic multiple issue.
3. The Core i7 microarchitecture has many more registers than x86 requires.
4. The Intel Core i7 uses less than half the pipeline stages of the earlier Intel Pentium 4 Prescott (see Figure 4.73).

4.14 Advanced Topic: an Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations

Modern digital design is done using hardware description languages and modern computer-aided synthesis tools that can create detailed hardware designs from the descriptions using both libraries and logic synthesis. Entire books are written on such languages and their use in digital design. This section, which appears online, gives a brief introduction and shows how a hardware design language, Verilog in this case, can be used to describe the MIPS control both behaviorally and in a form suitable for hardware synthesis. It then provides a series of behavioral models in Verilog of the MIPS five-stage pipeline. The initial model ignores hazards, and additions to the model highlight the changes for forwarding, data hazards, and branch hazards.

We then provide about a dozen illustrations using the single-cycle graphical pipeline representation for readers who want to see more detail on how pipelines work for a few sequences of MIPS instructions.

4.14 An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations

This CD section covers hardware decription languages and then gives a dozen examples of pipeline diagrams, starting on page 4.13-18.

As mentioned in Appendix C, Verilog can describe processors for simulation or with the intention that the Verilog specification be synthesized. To achieve acceptable synthesis results in size and speed, a behavioral specification intended for synthesis must carefully delineate the highly combinational portions of the design, such as a datapath, from the control. The datapath can then be synthesized using available libraries. A Verilog specification intended for synthesis is usually longer and more complex.

We start with a behavioral model of the 5-stage pipeline. To illustrate the dichotomy between behavioral and synthesizeable designs, we then give two Verilog descriptions of a multiple-cycle-per-instruction MIPS processor: one intended solely for simulations and one suitable for synthesis.

Using Verilog for Behavioral Specification with Simulation for the 5-Stage Pipeline

Figure e4.14.1 shows a Verilog behavioral description of the pipeline that handles ALU instructions as well as loads and stores. It does not accommodate branches (even incorrectly!), which we postpone including until later in the chapter.

```
module CPU (clock);
  // Instruction opcodes
   parameter LW = 6'b100011, SW = 6'b101011, BEQ = 6'b000100, no-op = 32'b00000_100000, ALUop = 6'b0;
   input clock;
   reg[31:0] PC, Regs[0:31], IMemory[0:1023], DMemory[0:1023], // separate memories
            IFIDIR, IDEXA, IDEXB, IDEXIR, EXMEMIR, EXMEMB, // pipeline registers
            EXMEMALUOut, MEMWBValue, MEMWBIR; // pipeline registers
   wire [4:0] IDEXrs, IDEXrt, EXMEMrd, MEMWBrd, MEMWBrt; // Access register fields
   wire [5:0] EXMEMop, MEMWBop, IDEXop; // Access opcodes
wire [31:0] Ain, Bin; // the ALU inputs
// These assignments define fields from the pipeline registers
   assign IDEXrs = IDEXIR[25:21];   // rs field
   assign IDEXrt = IDEXIR[20:16];   // rt field
   assign EXMEMrd = EXMEMIR[15:11]; // rd field
   assign MEMWBrd = MEMWBIR[15:11]; //rd field
   assign MEMWBrt = MEMWBIR[20:16]; //rt field--used for loads
   assign EXMEMop = EXMEMIR[31:26]; // the opcode
   assign MEMWBop = MEMWBIR[31:26]; // the opcode
   assign IDEXop = IDEXIR[31:26];   // the opcode
   // Inputs to the ALU come directly from the ID/EX pipeline registers
   assign Ain = IDEXA;
   assign Bin = IDEXB;
   reg [5:0] i; //used to initialize registers
   initial begin
       PC = 0;
       IFIDIR = no-op; IDEXIR = no-op; EXMEMIR = no-op; MEMWBIR = no-op; // put no-ops in pipeline registers
       for (i=0;i<=31;i=i+1) Regs[i] = i; //initialize registers--just so they aren't cares
   end
   always @ (posedge clock) begin
    // Remember that ALL these actions happen every pipe stage and with the use of <= they happen in parallel!
   // first instruction  in the pipeline is being fetched
          IFIDIR <= IMemory[PC>>2];
          PC <= PC + 4;
      end // Fetch & increment PC
      // second instruction in pipeline is fetching registers
         IDEXA <= Regs[IFIDIR[25:21]]; IDEXB <= Regs[IFIDIR[20:16]]; // get two registers
         IDEXIR <= IFIDIR;  //pass along IR--can happen anywhere, since this affects next stage only!
      // third instruction is doing address calculation or ALU operation
      if ((IDEXop==LW) |(IDEXop==SW))  // address calculation
           EXMEMALUOut <= IDEXA +{{16{IDEXIR[15]}}, IDEXIR[15:0]};
         else if (IDEXop==ALUop) case (IDEXIR[5:0]) //case for the various R-type instructions
              32: EXMEMALUOut <= Ain + Bin;  //add operation
              default: ; //other R-type operations: subtract, SLT, etc.
         endcase
      EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; //pass along the IR & B register
     //Mem stage of pipeline
     if (EXMEMop==ALUop) MEMWBValue <= EXMEMALUOut; //pass along ALU result
         else if (EXMEMop == LW) MEMWBValue <= DMemory[EXMEMALUOut>>2];
         else if (EXMEMop == SW) DMemory[EXMEMALUOut>>2] <=EXMEMB; //store
        MEMWBIR <= EXMEMIR; //pass along IR
     // the WB stage
     if ((MEMWBop==ALUop) & (MEMWBrd != 0)) // update registers if ALU operation and destination not 0
         Regs[MEMWBrd] <= MEMWBValue; // ALU operation
         else if ((EXMEMop == LW)& (MEMWBrt != 0)) // Update registers if load and destination not 0
            Regs[MEMWBrt] <= MEMWBValue;
   end
endmodule
```

FIGURE E4.14.1 **A Verilog behavioral model for the MIPS five-stage pipeline, ignoring branch and data hazards.** As in the design earlier in Chapter 4, we use separate instruction and data memories, which would be implemented using separate caches as we describe in Chapter 5.

Because Verilog lacks the ability to define registers with named fields such as structures in C, we use several independent registers for each pipeline register. We name these registers with a prefix using the same convention; hence, IFIDIR is the IR portion of the IFID pipeline register.

This version is a behavioral description not intended for synthesis. Instructions take the same number of clock cycles as our hardware design, but the control is done in a simpler fashion by repeatedly decoding fields of the instruction in each pipe stage. Because of this difference, the instruction register (IR) is needed throughout the pipeline, and the entire IR is passed from pipe stage to pipe stage. As you read the Verilog descriptions in this chapter, remember that the actions in the always block all occur in parallel on every clock cycle. Since there are no blocking assignments, the order of the events within the always block is arbitrary.

Implementing Forwarding in Verilog

To further extend the Verilog model, Figure e4.14.2 shows the addition of forwarding logic for the case when the source instruction is an ALU instruction and the source. Neither load stalls nor branches are handled; we will add these shortly. The changes from the earlier Verilog description are highlighted.

Check Yourself

Someone has proposed moving the write for a result from an ALU instruction from the WB to the MEM stage, pointing out that this would reduce the maximum length of forwards from an ALU instruction by one cycle. Which of the following are accurate reasons *not to* consider such a change?

1. It would not actually change the forwarding logic, so it has no advantage.
2. It is impossible to implement this change under any circum stance since the write for the ALU result must stay in the same pipe stage as the write for a load result.
3. Moving the write for ALU instructions would create the possibility of writes occurring from two different instructions during the same clock cycle. Either an extra write port would be required on the register file or a structural hazard would be created.
4. The result of an ALU instruction is not available in time to do the write during MEM.

```
module CPU (clock);
parameter LW = 6'b100011, SW = 6'b101011, BEQ = 6'b000100, no-op = 32'b00000_100000, ALUop = 6'b0;
input clock;
   reg[31:0] PC, Regs[0:31], IMemory[0:1023], DMemory[0:1023], // separate memories
            IFIDIR, IDEXA, IDEXB, IDEXIR, EXMEMIR, EXMEMB, // pipeline registers
            EXMEMALUOut, MEMWBValue, MEMWBIR; // pipeline registers
   wire [4:0] IDEXrs, IDEXrt, EXMEMrd, MEMWBrd, MEMWBrt; //hold register fields
   wire [5:0] EXMEMop, MEMWBop, IDEXop; Hold opcodes
   wire [31:0] Ain, Bin;

// declare the bypass signals
   wire bypassAfromMEM, bypassAfromALUinWB,bypassBfromMEM, bypassBfromALUinWB,
        bypassAfromLWinWB, bypassBfromLWinWB;

   assign IDEXrs = IDEXIR[25:21];    assign IDEXrt = IDEXIR[15:11];    assign EXMEMrd = EXMEMIR[15:11];
   assign MEMWBrd = MEMWBIR[20:16]; assign EXMEMop = EXMEMIR[31:26];
   assign MEMWBrt = MEMWBIR[25:20];
   assign MEMWBop = MEMWBIR[31:26];  assign IDEXop = IDEXIR[31:26];

   // The bypass to input A from the MEM stage for an ALU operation
   assign bypassAfromMEM = (IDEXrs == EXMEMrd) & (IDEXrs!=0) & (EXMEMop==ALUop); // yes, bypass

   // The bypass to input B from the MEM stage for an ALU operation
   assign bypassBfromMEM = (IDEXrt == EXMEMrd)&(IDEXrt!=0) & (EXMEMop==ALUop); // yes, bypass

   // The bypass to input A from the WB stage for an ALU operation
   assign bypassAfromALUinWB =( IDEXrs == MEMWBrd) & (IDEXrs!=0) & (MEMWBop==ALUop);

   // The bypass to input B from the WB stage for an ALU operation
   assign bypassBfromALUinWB = (IDEXrt == MEMWBrd) & (IDEXrt!=0) & (MEMWBop==ALUop); /

   // The bypass to input A from the WB stage for an LW operation
   assign bypassAfromLWinWB =( IDEXrs == MEMWBIR[20:16]) & (IDEXrs!=0) & (MEMWBop==LW);

   // The bypass to input B from the WB stage for an LW operation
   assign bypassBfromLWinWB = (IDEXrt == MEMWBIR[20:16]) & (IDEXrt!=0) & (MEMWBop==LW);

   // The A input to the ALU is bypassed from MEM if there is a bypass there,
   // Otherwise from WB if there is a bypass there, and otherwise comes from the IDEX register
   assign Ain = bypassAfromMEM? EXMEMALUOut :
               (bypassAfromALUinWB | bypassAfromLWinWB)? MEMWBValue : IDEXA;

   // The B input to the ALU is bypassed from MEM if there is a bypass there,
   // Otherwise from WB if there is a bypass there, and otherwise comes from the IDEX register
   assign Bin = bypassBfromMEM? EXMEMALUOut :
               (bypassBfromALUinWB | bypassBfromLWinWB)? MEMWBValue: IDEXB;

   reg [5:0] i; //used to initialize registers
   initial begin
      PC = 0;
     IFIDIR = no-op; IDEXIR = no-op; EXMEMIR = no-op; MEMWBIR = no-op; // put no-ops in pipeline registers
      for (i = 0;i<=31;i = i+1) Regs[i] = i; //initialize registers--just so they aren't cares
   end
   always @ (posedge clock) begin
      // first instruction in the pipeline is being fetched
          IFIDIR <= IMemory[PC>>2];
          PC <= PC + 4;
      end // Fetch & increment PC
      // second instruction is in register fetch
         IDEXA <= Regs[IFIDIR[25:21]]; IDEXB <= Regs[IFIDIR[20:16]]; // get two registers
         IDEXIR <= IFIDIR;  //pass along IR--can happen anywhere, since this affects next stage only!
      // third instruction is doing address calculation or ALU operation
      if ((IDEXop==LW) |(IDEXop==SW))  // address calculation & copy B
EXMEMALUOut <= IDEXA +{{16{IDEXIR[15]}}, IDEXIR[15:0]};
else if (IDEXop==ALUop) case (IDEXIR[5:0]) //case for the various R-type instructions
           32: EXMEMALUOut <= Ain + Bin;  //add operation
           default: ; //other R-type operations: subtract, SLT, etc.
          endcase
      EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; //pass along the IR & B register
      //Mem stage of pipeline
       if (EXMEMop==ALUop) MEMWBValue <= EXMEMALUOut; //pass along ALU result
          else if (EXMEMop == LW) MEMWBValue <= DMemory[EXMEMALUOut>>2];
          else if (EXMEMop == SW) DMemory[EXMEMALUOut>>2] <=EXMEMB; //store
       MEMWBIR <= EXMEMIR; //pass along IR
      // the WB stage
      if ((MEMWBop==ALUop) & (MEMWBrd != 0)) Regs[MEMWBrd] <= MEMWBValue; // ALU operation
      else if ((EXMEMop == LW)& (MEMWBrt != 0)) Regs[MEMWBrt] <= MEMWBValue;
   end
endmodule
```

FIGURE E4.14.2 **A behavioral definition of the five-stage MIPS pipeline with bypassing to ALU operations and address calculations.** The code added to Figure e4.14.1 to handle bypassing is highlighted. Because these bypasses only require changing where the ALU inputs come from, the only changes required are in the combinational logic responsible for selecting the ALU inputs.

The Behavioral Verilog with Stall Detection

If we ignore branches, stalls for data hazards in the MIPS pipe line are confined to one simple case: loads whose results are currently in the WB clock stage. Thus, extending the Verilog to handle a load with a destination that is either an ALU instruction or an effective address calculation is reasonably straightforward, and Figure 4.13.3 shows the few additions needed.

Check Yourself

Someone has asked about the possibility of data hazards occur ring through memory, as opposed to through a register. Which of the following statements about such hazards are true?

1. Since memory accesses only occur in the MEM stage, all memory operations are done in the same order as instruction execution, making such hazards impossible in this pipe line.
2. Such hazards *are* possible in this pipeline; we just have not discussed them yet.
3. No pipeline can ever have a hazard involving memory, since it is the programmer's job to keep the order of memory references accurate.
4. Memory hazards may be possible in some pipelines, but they cannot occur in this particular pipeline.
5. Although the pipeline control would be obligated to maintain ordering among memory references to avoid hazards, it is impossible to design a pipeline where the references could be out of order.

```
module CPU (clock);
parameter LW = 6'b100011, SW = 6'b101011, BEQ = 6'b000100, no-op = 32'b00000_100000, ALUop = 6'b0;
input clock;
   reg[31:0] PC, Regs[0:31], IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXA, IDEXB, IDEXIR, EXMEMIR, EXMEMB, // pipeline registers
             EXMEMALUOut, MEMWBValue, MEMWBIR; // pipeline registers
   wire [4:0] IDEXrs, IDEXrt, EXMEMrd, MEMWBrd, MEMWBrt; //hold register fields
   wire [5:0] EXMEMop, MEMWBop, IDEXop; Hold opcodes
   wire [31:0] Ain, Bin;

// declare the bypass signals
   wire   stall , bypassAfromMEM, bypassAfromALUinWB,bypassBfromMEM, bypassBfromALUinWB,
        bypassAfromLWinWB, bypassBfromLWinWB;

   assign IDEXrs = IDEXIR[25:21];    assign IDEXrt = IDEXIR[15:11];    assign EXMEMrd = EXMEMIR[15:11];
   assign MEMWBrd = MEMWBIR[20:16]; assign EXMEMop = EXMEMIR[31:26];
      assign MEMWBrt = MEMWBIR[25:20];
   assign MEMWBop = MEMWBIR[31:26];  assign IDEXop = IDEXIR[31:26];
   // The bypass to input A from the MEM stage for an ALU operation
   assign bypassAfromMEM = (IDEXrs == EXMEMrd) & (IDEXrs!=0) & (EXMEMop==ALUop); // yes, bypass
assign bypassAfromMEM = (IDEXrs == EXMEMrd) & (IDEXrs!=0) & (EXMEMop==ALUop); // yes, bypass
// The bypass to input B from the MEM stage for an ALU operation
assign bypassBfromMEM = (IDEXrt== EXMEMrd)&(IDEXrt!=0) & (EXMEMop==ALUop); // yes, bypass
// The bypass to input A from the WB stage for an ALU operation
assign bypassAfromALUinWB =( IDEXrs == MEMWBrd) & (IDEXrs!=0) & (MEMWBop==ALUop);
// The bypass to input B from the WB stage for an ALU operation
assign bypassBfromALUinWB = (IDEXrt==MEMWBrd) & (IDEXrt!=0) & (MEMWBop==ALUop); /
// The bypass to input A from the WB stage for an LW operation
assign bypassAfromLWinWB =( IDEXrs ==MEMWBIR[20:16]) & (IDEXrs!=0) & (MEMWBop==LW);
// The bypass to input B from the WB stage for an LW operation
assign bypassBfromLWinWB = (IDEXrt==MEMWBIR[20:16]) & (IDEXrt!=0) & (MEMWBop==LW);
// The A input to the ALU is bypassed from MEM if there is a bypass there,
// Otherwise from WB if there is a bypass there, and otherwise comes from the IDEX register
assign Ain = bypassAfromMEM? EXMEMALUOut :
             (bypassAfromALUinWB | bypassAfromLWinWB)? MEMWBValue : IDEXA;
// The B input to the ALU is bypassed from MEM if there is a bypass there,
// Otherwise from WB if there is a bypass there, and otherwise comes from the IDEX register
assign Bin = bypassBfromMEM? EXMEMALUOut :
             (bypassBfromALUinWB | bypassBfromLWinWB)? MEMWBValue: IDEXB;

// The signal for detecting a stall based on the use of a result from LW
assign stall = (MEMWBIR[31:26]==LW) && // source instruction is a load
     ((((IDEXop==LW)|(IDEXop==SW)) && (IDEXrs==MEMWBrd)) | // stall for address calc
((IDEXop==ALUop) && ((IDEXrs==MEMWBrd)|(IDEXrt==MEMWBrd)))); // ALU use

reg [5:0] i; //used to initialize registers

initial begin
  PC = 0;
 IFIDIR = no-op; IDEXIR = no-op; EXMEMIR = no-op; MEMWBIR = no-op; // put no-ops in pipeline registers
  for (i = 0;i<=31;i = i+1) Regs[i] = i; //initialize registers--just so they aren't cares
end

always @ (posedge clock) begin

  if (~stall) begin // the first three pipeline stages stall if there is a load hazard

   // first instruction  in the pipeline is being fetched
       IFIDIR <= IMemory[PC>>2];
       PC <= PC + 4;

      IDEXIR <= IFIDIR;  //pass along IR--can happen anywhere, since this affects next stage only!

   // second instruction is in register fetch
    IDEXA <= Regs[IFIDIR[25:21]]; IDEXB <= Regs[IFIDIR[20:16]]; // get two registers

   // third instruction is doing address calculation or ALU operation
       if ((IDEXop==LW) |(IDEXop==SW))  // address calculation & copy B
              EXMEMALUOut <= IDEXA +{{16{IDEXIR[15]}}, IDEXIR[15:0]};
      else if (IDEXop==ALUop) case (IDEXIR[5:0]) //case for the various R-type instructions
           32: EXMEMALUOut <= Ain + Bin;  //add operation
           default: ; //other R-type operations: subtract, SLT, etc.
         endcase
    EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; //pass along the IR & B register
  end

else EXMEMIR <= no-op; /Freeze first three stages of pipeline; inject a nop into the EX output

   //Mem stage of pipeline
    if (EXMEMop==ALUop) MEMWBValue <= EXMEMALUOut; //pass along ALU result
       else if (EXMEMop == LW) MEMWBValue <= DMemory[EXMEMALUOut>>2];
         else if (EXMEMop == SW) DMemory[EXMEMALUOut>>2] <=EXMEMB; //store

    MEMWBIR <= EXMEMIR; //pass along IR

   // the WB stage

   if ((MEMWBop==ALUop) & (MEMWBrd != 0)) Regs[MEMWBrd] <= MEMWBValue; // ALU operation

   else if ((EXMEMop == LW)& (MEMWBrt != 0)) Regs[MEMWBrt] <= MEMWBValue;

end
endmodule
```

FIGURE E4.14.3 **A behavioral definition of the five-stage MIPS pipeline with stalls for loads**

when the destination is an ALU instruction or effective address calculation.
The changes from Figure e4.14.2 are highlighted.

Implementing the Branch Hazard Logic in Verilog

We can extend our Verilog behavioral model to implement the control for branches. We add the code to model branch equal using a "predict not taken" strategy. The Verilog code is shown in Figure e4.14.4. It implements the branch hazard by detecting a taken branch in ID and using that signal to squash the instruction in IF (by setting the IR to 0, which is an effective `no-op` in MIPS-32); in addition, the PC is assigned to the branch target. Note that to prevent an unexpected latch, it is important that the PC is clearly assigned on every path through the always block; hence, we assign the PC in a single *if* statement. Lastly, note that although Figure e4.14.4 incorporates the basic logic for branches and control hazards, the incorporation of branches requires additional bypassing and data hazard detection, which we have not included.

```
module CPU (clock);
parameter LW = 6'b100011, SW = 6'b101011, BEQ = 6'b000100, no-op = 32'b0000000_0000000_0000000_0000000, ALUop = 6'b0 ;
input clock;
   reg[31:0] PC, Regs[0:31], IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXA, IDEXB, IDEXIR, EXMEMIR, EXMEMB, // pipeline registers
             EXMEMALUOut, MEMWBValue, MEMWBIR; // pipeline registers
   wire [4:0] IDEXrs, IDEXrt, EXMEMrd, MEMWBrd; //hold register fields
   wire [5:0] EXMEMop, MEMWBop, IDEXop; Hold opcodes
   wire [31:0] Ain, Bin;
   // declare the bypass signals
   wire  takebranch , stall, bypassAfromMEM, bypassAfromALUinWB,bypassBfromMEM, bypassBfromALUinWB,
      bypassAfromLWinWB, bypassBfromLWinWB;
   assign IDEXrs = IDEXIR[25:21];  assign IDEXrt = IDEXIR[15:11];  assign EXMEMrd = EXMEMIR[15:11];
   assign MEMWBrd = MEMWBIR[20:16]; assign EXMEMop = EXMEMIR[31:26];
   assign MEMWBop = MEMWBIR[31:26];  assign IDEXop = IDEXIR[31:26];
   // The bypass to input A from the MEM stage for an ALU operation
   assign bypassAfromMEM = (IDEXrs == EXMEMrd) & (IDEXrs!=0) & (EXMEMop==ALUop); // yes, bypass
   // The bypass to input B from the MEM stage for an ALU operation
   assign bypassBfromMEM = (IDEXrt == EXMEMrd)&(IDEXrt!=0) & (EXMEMop==ALUop); // yes, bypass
   // The bypass to input A from the WB stage for an ALU operation
   assign bypassAfromALUinWB =( IDEXrs == MEMWBrd) & (IDEXrs!=0) & (MEMWBop==ALUop);
   // The bypass to input B from the WB stage for an ALU operation
   assign bypassBfromALUinWB = (IDEXrt == MEMWBrd) & (IDEXrt!=0) & (MEMWBop==ALUop); /
   // The bypass to input A from the WB stage for an LW operation
   assign bypassAfromLWinWB =( IDEXrs == MEMWBIR[20:16]) & (IDEXrs!=0) & (MEMWBop==LW);
   // The bypass to input B from the WB stage for an LW operation
   assign bypassBfromLWinWB = (IDEXrt == MEMWBIR[20:16]) & (IDEXrt!=0) & (MEMWBop==LW);
   // The A input to the ALU is bypassed from MEM if there is a bypass there,
   // Otherwise from WB if there is a bypass there, and otherwise comes from the IDEX register
   assign Ain = bypassAfromMEM? EXMEMALUOut :
               (bypassAfromALUinWB | bypassAfromLWinWB)? MEMWBValue : IDEXA;
   // The B input to the ALU is bypassed from MEM if there is a bypass there,
   // Otherwise from WB if there is a bypass there, and otherwise comes from the IDEX register
   assign Bin = bypassBfromMEM? EXMEMALUOut :
               (bypassBfromALUinWB | bypassBfromLWinWB)? MEMWBValue: IDEXB;
   // The signal for detecting a stall based on the use of a result from LW
   assign stall = (MEMWBIR[31:26]==LW) && // source instruction is a load
         ((((IDEXop==LW)|(IDEXop==SW)) && (IDEXrs==MEMWBrd)) | // stall for address calc
((IDEXop==ALUop) && ((IDEXrs==MEMWBrd)|(IDEXrt==MEMWBrd)))); // ALU use
// Signal for a taken branch: instruction is BEQ and registers are equal
assign takebranch = (IFIDIR[31:26]==BEQ) && (Regs[IFIDIR[25:21]]== Regs[IFIDIR[20:16]]);
   reg [5:0] i; //used to initialize registers
   initial begin
      PC = 0;
      IFIDIR = no-op; IDEXIR = no-op; EXMEMIR = no-op; MEMWBIR = no-op; // put no-ops in pipeline registers
      for (i = 0;i<=31;i = i+1) Regs[i] = i; //initialize registers--just so they aren't don't cares
   end
   always @ (posedge clock) begin
   if (~stall) begin // the first three pipeline stages stall if there is a load hazard
      if (~takebranch) begin           // first instruction in the pipeline is being fetched normally
          IFIDIR <= IMemory[PC>>2];
          PC <= PC + 4;
      end else begin // a taken branch is in ID; instruction in IF is wrong; insert a no-op and reset the PC
         IFDIR <= no-op;
         PC <= PC + 4 + ({{16{IFIDIR[15]}}, IFIDIR[15:0]}<<2);
         end
      // second instruction is in register fetch
       IDEXA <= Regs[IFIDIR[25:21]]; IDEXB <= Regs[IFIDIR[20:16]]; // get two registers
      // third instruction is doing address calculation or ALU operation
         IDEXIR <= IFIDIR;  //pass along IR
if ((IDEXop==LW) |(IDEXop==SW))  // address calculation & copy B
           EXMEMALUOut <= IDEXA +{{16{IDEXIR[15]}}, IDEXIR[15:0]};
       else if (IDEXop==ALUop) case (IDEXIR[5:0]) //case for the various R-type instructions
            32: EXMEMALUOut <= Ain + Bin;  //add operation
            default: ; //other R-type operations: subtract, SLT, etc.
           endcase
       EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; //pass along the IR & B register
     end
   else EXMEMIR <= no-op; /Freeze first three stages of pipeline; inject a nop into the EX output
      //Mem stage of pipeline
       if (EXMEMop==ALUop) MEMWBValue <= EXMEMALUOut; //pass along ALU result
          else if (EXMEMop == LW) MEMWBValue <= DMemory[EXMEMALUOut>>2];
            else if (EXMEMop == SW) DMemory[EXMEMALUOut>>2] <=EXMEMB; //store
      // the WB stage
MEMWBIR <= EXMEMIR; //pass along IR
      if ((MEMWBop==ALUop) & (MEMWBrd != 0)) Regs[MEMWBrd] <= MEMWBValue; // ALU operation
      else if ((EXMEMop == LW)& (MEMWBIR[20:16] != 0)) Regs[MEMWBIR[20:16]] <= MEMWBValue;
   end
endmodule
```

FIGURE E4.14.4 **A behavioral definition of the five-stage MIPS pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation.**
The changes from Figure e4.14.3 are highlighted.

Using Verilog for Behavioral Specification with Synthesis

To demonstate the contrasting types of Verilog, we show two descriptions of a different, nonpipelined implementation style of MIPS that uses multiple clock cycles per instruction. (Since some instructors make a synthesizeable description of the MIPS pipe line project for a class, we chose not to include it here. It would also be long.)

Figure e4.14.5 gives a behavioral specification of a multicycle implementation of the MIPS processor. Because of the use of behavioral operations, it would be difficult to synthesize a separate datapath and control unit with any reasonable efficiency. This version demonstrates another approach to the control by using a Mealy finite-state machine (see discussion in Section C.10 of Appendix B). The use of a Mealy machine, which allows the output to depend both on inputs and the current state, allows us to decrease the total number of states.

```
module CPU (clock);

parameter LW = 6'b100011, SW = 6'b101011, BEQ=6'b000100, J=6'd2;

input clock; //the clock is an external input

// The architecturally visible registers and scratch registers for implementation
reg [31:0] PC, Regs[0:31], Memory [0:1023], IR, ALUOut, MDR, A, B;

reg [2:0] state; // processor state

wire [5:0] opcode; //use to get opcode easily

wire [31:0] SignExtend,PCOffset; //used to get sign-extended offset field

assign opcode = IR[31:26]; //opcode is upper 6 bits

assign SignExtend = {{16{IR[15]}},IR[15:0]}; //sign extension of lower 16 bits of instruction

assign PCOffset = SignExtend << 2; //PC offset is shifted

// set the PC to 0 and start the control in state 0
initial begin PC = 0; state = 1; end


//The state machine--triggered on a rising clock
always @(posedge clock) begin

    Regs[0] = 0; //make R0 0 //shortcut way to make sure R0 is always 0

    case (state) //action depends on the state

      1: begin // first step: fetch the instruction, increment PC, go to next state

         IR <= Memory[PC>>2];
         PC <= PC + 4;
         state = 2; //next state

      end


      2: begin // second step: Instruction decode, register fetch, also compute branch address
    2: begin // second step: instruction decode, register fetch, also compute branch address
       A <= Regs[IR[25:21]];
       B <= Regs[IR[20:16]];
       state = 3;
       ALUOut <= PC + PCOffset; // compute PC-relative branch target
    end

    3: begin // third step: Load-store execution, ALU execution, Branch completion
         state = 4; // default next state
         if ((opcode==LW) |(opcode==SW)) ALUOut <= A + SignExtend; //compute effective address
         else if (opcode==6'b0) case (IR[5:0]) //case for the various R-type instructions
           32: ALUOut = A + B; //add operation
           default: ALUOut = A; //other R-type operations: subtract, SLT, etc.
         endcase
        else if (opcode == BEQ) begin
                   if (A==B) PC <= ALUOut; // branch taken--update PC
                   state = 1;
         end
         else if (opocde=J) begin
              PC = {PC[31:28], IR[25:0],2'b00}; // the jump target PC
              state = 1;
         end  //Jumps
                 else ; // other opcodes or exception for undefined instruction would go here
 end

 4: begin
     if (opcode==6'b0) begin //ALU Operation
           Regs[IR[15:11]] <= ALUOut; // write the result
           state = 1;
     end //R-type finishes
        else if (opcode == LW) begin // load instruction
            MDR <= Memory[ALUOut>>2]; // read the memory
            state = 5; // next state
         end
                else if (opcode == LW) begin
                    Memory[ALUOut>>2] <= B; // write the memory
                    state = 1; // return to state 1
                end //store finishes
                     else ; // other instructions go here
       end

 5: begin // LW is the only instruction still in execution
       Regs[IR[20:16]] = MDR; // write the MDR to the register
       state = 1;
    end //complete an LW instruction
 endcase
end
endmodule
```

FIGURE E4.14.5 **A behavioral specification of the multicycle MIPS design.**
This has the same cycle behavior as the multicycle design, but is purely for simulation and specification. It cannot be used for synthesis.

Since a version of the MIPS design intended for synthesis is considerably more complex, we have relied on a number of Verilog modules that were specified in Appendix B, including the following:

- The 4-to-1 multiplexor shown in Figure B.4.2, and the 3-to-1 multiplexor that can be trivially derived based on the 4-to-1 multiplexor.
- The MIPS ALU shown in Figure B.5.15.
- The MIPS ALU control defined in Figure B.5.16.
- The MIPS register file defined in Figure B.8.11.

Now, let's look at a Verilog version of the MIPS processor intended for synthesis. Figure e4.14.6 shows the structural version of the MIPS datapath. Figure e4.14.7 uses the datapath module to specify the MIPS CPU. This version also demonstrates another approach to implementing the control unit, as well as some optimizations that rely on relationships between various control signals. Observe that the state machine specification only provides the sequencing actions.

```
module Datapath (ALUOp, RegDst, MemtoReg, MemRead, MemWrite, IorD, RegWrite, IRWrite,
PCWrite,   PCWriteCond, ALUSrcA, ALUSrcB, PCSource, opcode, clock); // the control inputs + clock
input [1:0] ALUOp, ALUSrcB, PCSource; // 2-bit control signals
input RegDst, MemtoReg, MemRead, MemWrite, IorD, RegWrite, IRWrite, PCWrite, PCWriteCond,
ALUSrcA,    clock; // 1-bit control signals
output [5:0] opcode ;// opcode is needed as an output by control
reg [31:0] PC, Memory [0:1023], MDR,IR, ALUOut; // CPU state + some temporaries
wire [31:0] A,B,SignExtendOffset, PCOffset, ALUResultOut, PCValue, JumpAddr, Writedata, ALUAin,
    ALUBin,MemOut; / these are signals derived from registers
wire [3:0] ALUCtl; //. the ALU control lines
wire Zero; the Zero out signal from the ALU
wire[4:0] Writereg;// the signal used to communicate the destination register
 initial PC = 0; //start the PC at 0

//Combinational signals used in the datapath

// Read using word address with either ALUOut or PC as the address source
assign MemOut = MemRead ? Memory[(IorD ? ALUOut : PC)>>2]:0;
assign opcode = IR[31:26];// opcode shortcut

// Get the write register address from one of two fields depending on RegDst
assign Writereg = RegDst ? IR[15:11]: IR[20:16];

// Get the write register data either from the ALUOut or from the MDR
assign Writedata = MemtoReg ? MDR : ALUOut;

// Sign-extend the lower half of the IR from load/store/branch offsets
assign SignExtendOffset = {{16{IR[15]}},IR[15:0]}; //sign-extend lower 16 bits;

// The branch offset is also shifted to make it a word offset
assign PCOffset = SignExtendOffset << 2;

// The A input to the ALU is either the rs register or the PC
assign ALUAin = ALUSrcA ? A : PC; //ALU input is PC or A

// Compose the Jump address
assign JumpAddr = {PC[31:28], IR[25:0],2'b00}; //The jump address

// Creates an instance of the ALU control unit (see the module de             fined in Figure C.5.16 on page C-3       8

   // Input ALUOp is control-unit set and used to describe the instruction class as in Chapter 4
   // Input IR[5:0] is the function code field for an ALU instruction
   // Output ALUCtl are the actual ALU control bits as in Chapter 4

ALUControl alucontroller (ALUOp,IR[5:0],ALUCtl); //ALU control unit

// Creates a 3-to-1 multiplexor used to select the source of the next PC

    // Inputs are ALUResultOut (the incremented PC) , ALUOut (the branch address), the jump target address
   // PCSource is the selector input and PCValue is the multiplexor output

Mult3to1 PCdatasrc (ALUResultOut,ALUOut,JumpAddr, PCSource , PCValue);

 // Creates a 4-to-1 multiplexor used to select the B input of the ALU

   //  Inputs are register B,constant 4, sign-extended lower half of IR, sign-extended lower half of IR <<                     2
   // ALUSrcB is the selector input
   // ALUBin is the multiplexor output

Mult4to1 ALUBinput (B,32'd4,SignExtendOffset,PCOffset,ALUSrcB,ALUBin);

 // Creates a MIPS ALU

   // Inputs are ALUCtl (the ALU control), ALU value inputs (ALUAin, ALUBin)
   // Outputs are ALUResultOut (the 32-bit output) and Zero (zero detection output)

MIPSALU ALU (ALUCtl, ALUAin, ALUBin, ALUResultOut,Zero); //the ALU

// Creates a MIPS register file

   // Inputs are
   // the rs and rt fields of the IR used to specify which registers to read,
   // Writereg (the write register number), Writedata (the data to be written), RegWrite (indicates a
    write), the clock
    // Outputs are A and B, the registers read
    registerfile regs (IR[25:21],IR[20:16],Writereg,Writedata,RegWrite,A,B,clock); //Register file

// The clock-triggered actions of the datapath

always @(posedge clock) begin   if (MemWrite) Memory[ALUOut>>2] <= B; // Write memory--must be a store

   ALUOut <= ALUResultOut; //Save the ALU result for use on a later clock cycle

   if (IRWrite) IR <= MemOut; // Write the IR if an instruction fetch

   MDR <= MemOut; // Always save the memory read value

   // The PC is written both conditionally (controlled by PCWrite) and unconditionally
      if (PCWrite || (PCWriteCond & Zero)) PC <=PCValue;

end
endmodule
```

FIGURE E4.14.6 **A Verilog version of the multicycle MIPS datapath that is appropriate for synthesis.**
This datapath relies on several units from Appendix B. Initial statements do not synthesize, and a version used for synthesis would have to incorporate a reset signal that had this effect. Also note that resetting `R0` to 0 on every clock is not the best way to ensure that `R0` stays 0; instead, modifying the register file module to produce 0 whenever `R0` is read and to ignore writes to `R0` would be a more efficient solution.

```
module CPU (clock);
   parameter LW = 6'b100011, SW = 6'b101011, BEQ = 6'b000100, J = 6'd2; //constants
   input clock; reg [2:0] state;
   wire [1:0] ALUOp, ALUSrcB, PCSource; wire [5:0] opcode;
   wire RegDst, MemRead, MemWrite, IorD, RegWrite, IRWrite, PCWrite, PCWriteCond,
        ALUSrcA, MemoryOp, IRWwrite, Mem2Reg;
// Create an instance of the MIPS datapath, the inputs are the control signals; opcode is only output
Datapath MIPSDP (ALUOp,RegDst,Mem2Reg, MemRead, MemWrite, IorD, RegWrite,
   IRWrite, PCWrite, PCWriteCond, ALUSrcA, ALUSrcB, PCSource, opcode, clock);
 initial begin state = 1; end // start the state machine in state 1
// These are the definitions of the control signals
assign IRWrite = (state==1);
assign Mem2Reg = ~ RegDst;
assign MemoryOp = (opcode==LW)|(opcode==SW); // a memory operation
assign ALUOp = ((state==1)|(state==2)|((state==3)&MemoryOp)) ? 2'b00 : // add
       ((state==3)&(opcode==BEQ)) ? 2'b01 : 2'b10; // subtract or use function code
   assign RegDst = ((state==4)&(opcode==0)) ? 1 : 0;
   assign MemRead = (state==1) | ((state==4)&(opcode==LW));
   assign MemWrite = (state==4)&(opcode==SW);
   assign IorD = (state==1) ? 0 : (state==4) ? 1 : X;
   assign RegWrite = (state==5) | ((state==4) &(opcode==0));
   assign PCWrite = (state==1) | ((state==3)&(opcode==J));
   assign PCWriteCond = (state==3)&(opcode==BEQ);
   assign ALUSrcA = ((state==1)|(state==2)) ? 0 :1;
   assign ALUSrcB = ((state==1) | ((state==3)&(opcode==BEQ))) ? 2'b01 : (state==2) ? 2'b11 :
          ((state==3)&MemoryOp) ? 2'b10 : 2'b00; // memory operation or other
   assign PCSource = (state==1) ? 2'b00 : ((opcode==BEQ) ? 2'b01 : 2'b10);
// Here is the state machine, which only has to sequence states

   always @(posedge clock) begin // all state updates on a positive clock edge
      case (state)
      1: state = 2;  //unconditional next state
      2: state = 3;  //unconditional next state
      3: // third step: jumps and branches complete
         state = ((opcode==BEQ) | (opcode==J)) ? 1 : 4;// branch or jump go back else next state
      4: state = (opcode==LW) ? 5 : 1; //R-type and SW finish
      5: state = 1; // go back
       endcase
end
  endmodule
```

FIGURE E4.14.7 The MIPS CPU using the datapath from Figure e4.14.6.

The setting of the control lines is done with a series of `assign` statements that depend on the state as well as the opcode field of the instruction register. If one were to fold the setting of the control into the state specification, this would look like a Mealy-style finite-state control unit. Because the setting of the control lines is specified using `assign` statements outside of the always block, most logic synthesis systems will generate a small implementation of a finite-state machine that determines the setting of the state register and then uses external logic to derive the control inputs to the datapath.

In writing this version of the control, we have also taken advantage of a number of insights about the relationship between various control signals as well as situations where we don't care about the control signal value; some examples of these are given in the following elaboration.

Elaboration

When specifying control, designers often take advantage of knowledge of the control so as to simplify or shorten the control specification. Here are a few examples from the specification in Figures e4.14.6 and e4.14.7.

1. `MemtoReg` is set only in two cases, and then it is always the inverse of `RegDst`, so we just use the inverse of `RegDst`.
2. `IRWrite` is set only in state 1.
3. The ALU does not operate in every state and, when unused, can safely do anything.
4. `RegDst` is 1 in only one case and can otherwise be set to 0. In practice it might be better to set it explicitly when needed and otherwise set it to X, as we do for `IorD`. First, it allows additional logic optimization possibilities through the exploitation of don't-care terms (see Appendix B for further discussion and examples). Second, it is a more precise specification, and this allows the simulation to more closely model the hardware, possibly uncovering additional errors in the specification.

More Illustrations of Instruction Execution on the Hardware

To reduce the cost of this book, in the third edition we moved sections and figures that were used by a minority of instructors online. This subsection recaptures those figures for readers who would like more supplemental material to better understand pipelining. These are all single-clock-cycle pipeline diagrams, which take many figures to illustrate the execution of a sequence of instructions.

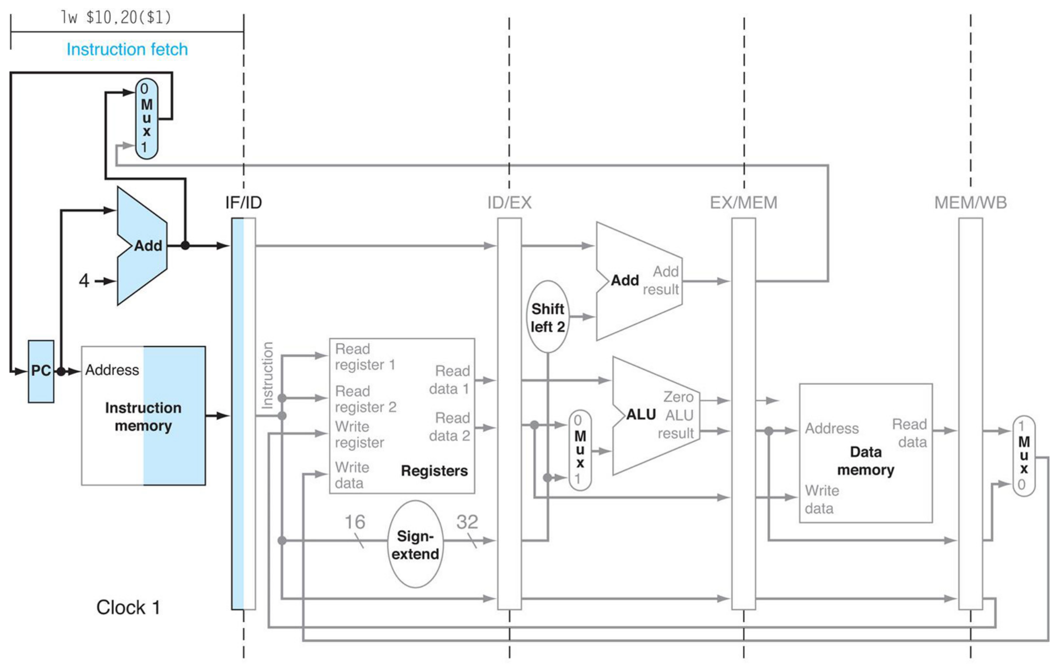
The three examples are for code with no hazards, an example of forwarding on the pipelined implementation, and an example of bypassing on the pipelined implementation.

No Hazard Illustrations

On page 297, we gave the example code sequence

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

Figures 4.43 and 4.44 showed the multiple-clock-cycle pipeline diagrams for this two-instruction sequence executing across six clock cycles. Figures e4.14.8 through e4.14.10 show the corresponding single-clock-cycle pipeline diagrams for these two instructions. Note that the order of the instructions differs between these two types of diagrams: the newest instruction is at the *bottom and to the right* of the multiple-clock-cycle pipeline diagram, and it is on the *left* in the single-clock-cycle pipeline diagram.



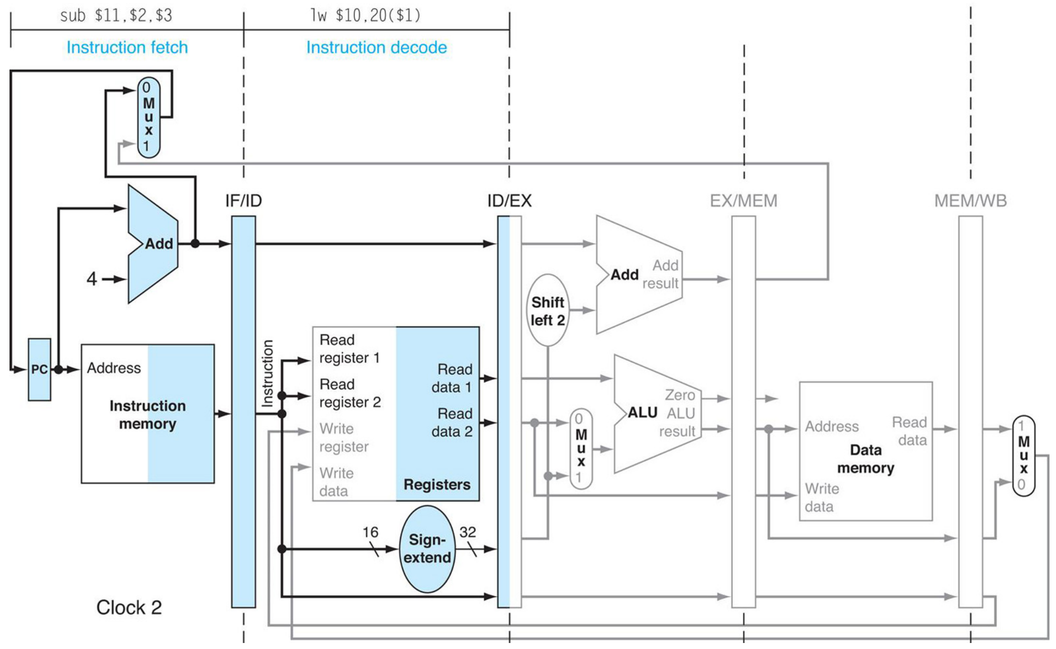
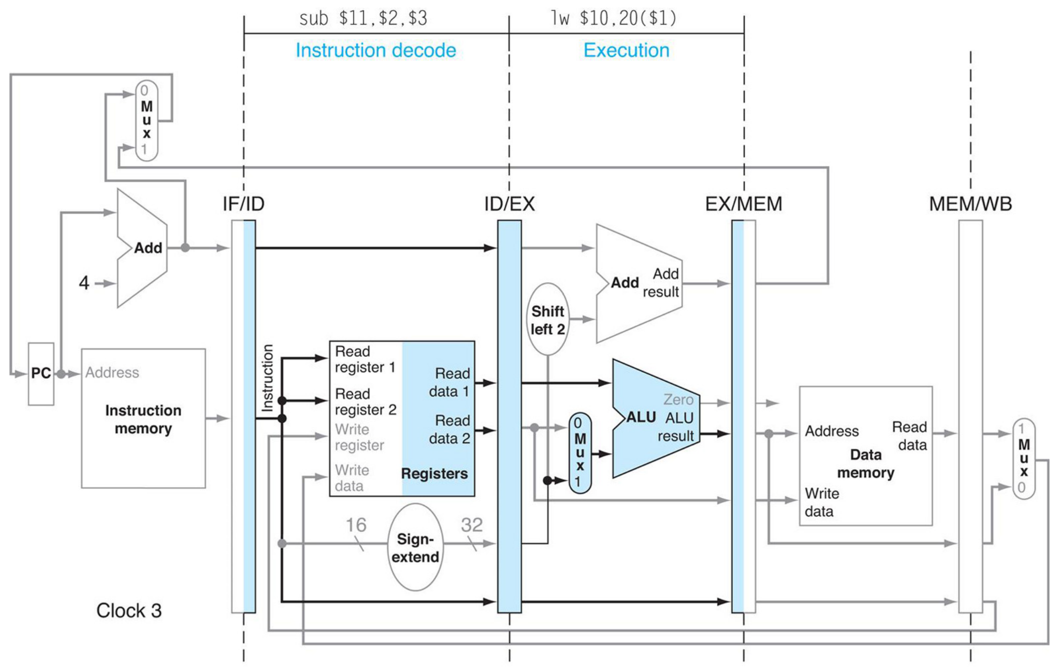


FIGURE E4.14.8 **Single-cycle pipeline diagrams for clock cycles 1 (top diagram) and 2 (bottom diagram).**
This style of pipeline representation is a snap shot of every instruction executing during one clock cycle. Our example has but two instructions, so at most two stages are identified

in each clock cycle; normally, all five stages are occupied. The highlighted portions of the datapath are active in that clock cycle. The load is fetched in clock cycle 1 and decoded in clock cycle 2, with the subtract fetched in the second clock cycle. To make the figures easier to understand, the other pipeline stages are empty, but normally there is an instruction in every pipeline stage.



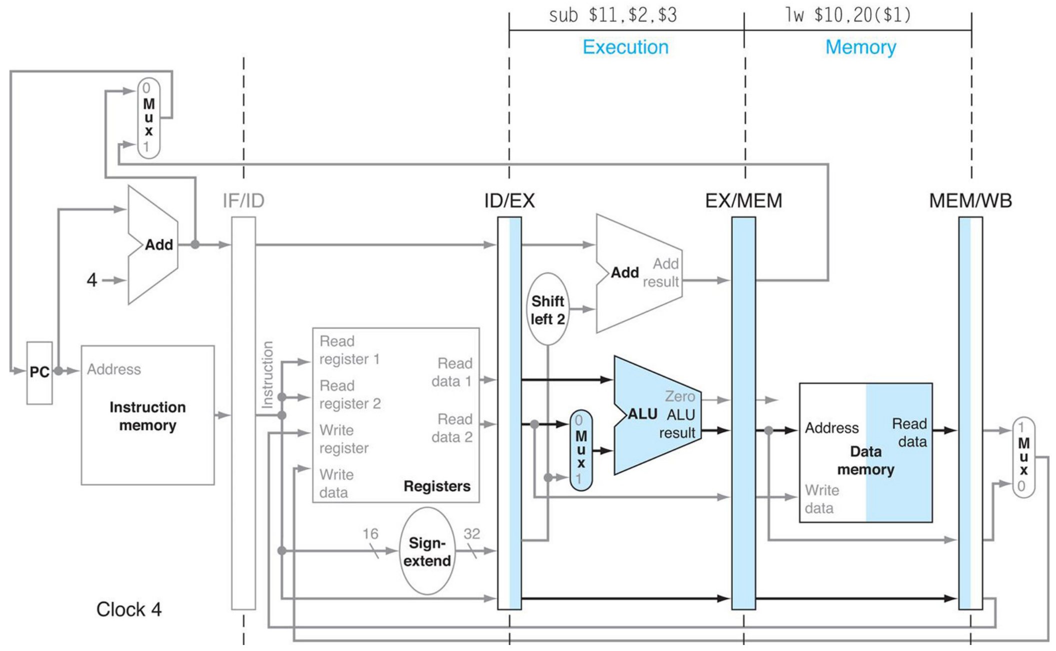


FIGURE E4.14.9 **Single-cycle pipeline diagrams for clock cycles 3 (top diagram) and 4 (bottom diagram).**

In the third clock cycle in the top diagram, `lw` enters the EX stage. At the same time, sub enters ID. In the fourth clock cycle (bottom datapath), `lw` moves into MEM stage, reading memory using the address found in EX/MEM at the beginning of clock cycle 4. At the same time, the ALU subtracts and then places the difference into EX/MEM at the end of the clock cycle.

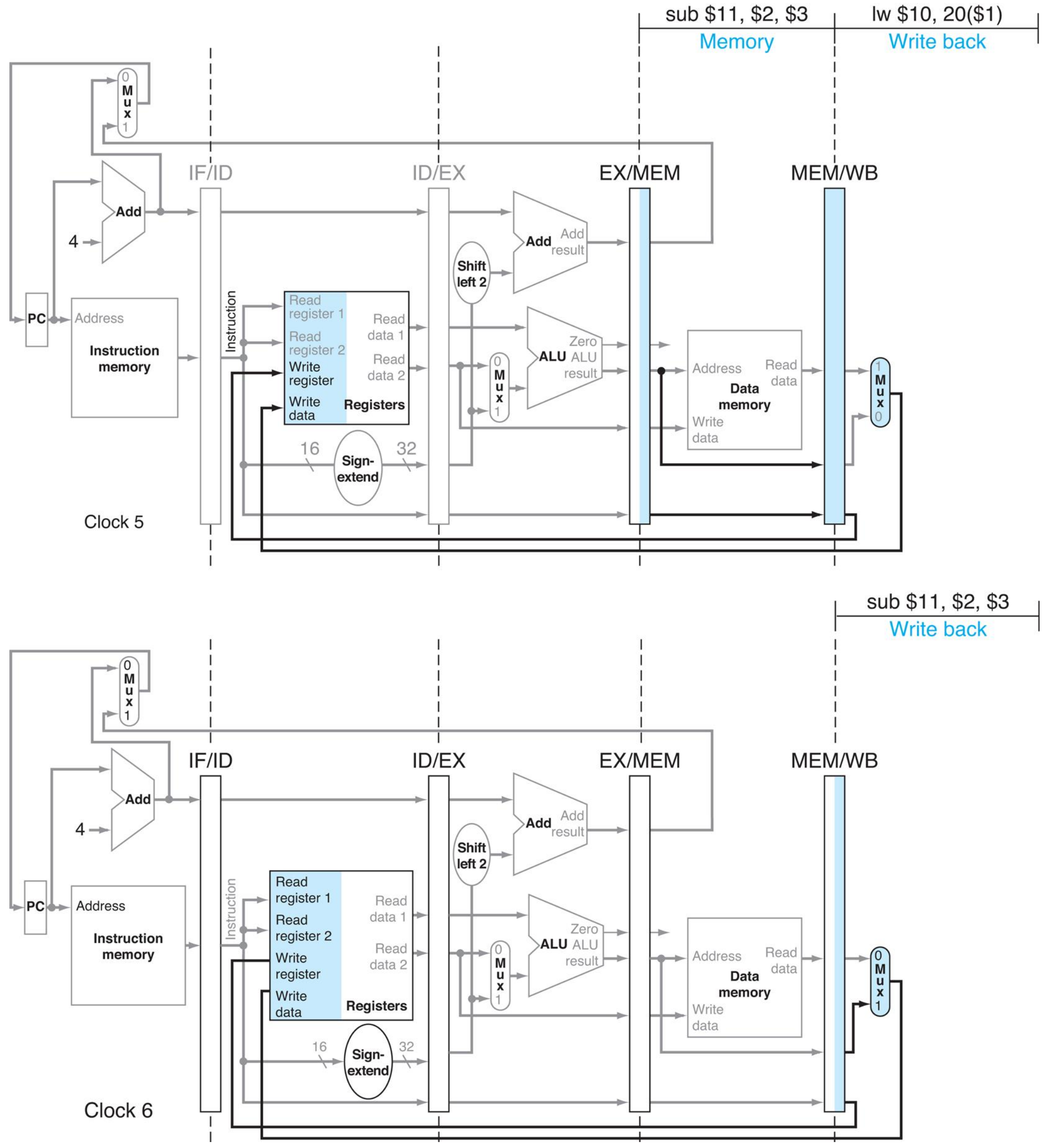


FIGURE E4.14.10 **Single-cycle pipeline diagrams for clock cycles 5 (top diagram) and 6 (bottom diagram).**
In clock cycle 5, `lw` completes by writing the data in MEM/WB into register 10, and sub sends the difference in EX/MEM to `MEM/WB`. In the next clock cycle, sub writes the value in MEM/WB to register 11.

More Examples

To understand how pipeline control works, let’s consider these five instructions going through the pipeline:

`lw`	`$10, 20($1)`
`sub`	`$11, $2, $3`
`and`	`$12, $4, $5`
`or`	`$13, $6, $7`
`add`	`$14, $8, $9`

Figures e4.14.11 through e4.14.15 show these instructions proceeding through the nine clock cycles it takes them to complete execution, highlighting what is active in a stage and identifying the instruction associated with each stage during a clock cycle. If you examine them carefully, you may notice:

- In Figure e4.14.13 you can see the sequence of the destination register numbers from left to right at the bottom of the pipeline registers. The numbers advance to the right during each clock cycle, with the MEM/WB pipeline register sup plying the number of the register written during the WB stage.
- When a stage is inactive, the values of control lines that are deasserted are shown as 0 or X (for don’t care).
- Sequencing of control is embedded in the pipeline structure itself. First, all instructions take the same number of clock cycles, so there is no special control for instruction duration. Second, all control information is computed during instruction decode and then passed along by the pipeline registers.

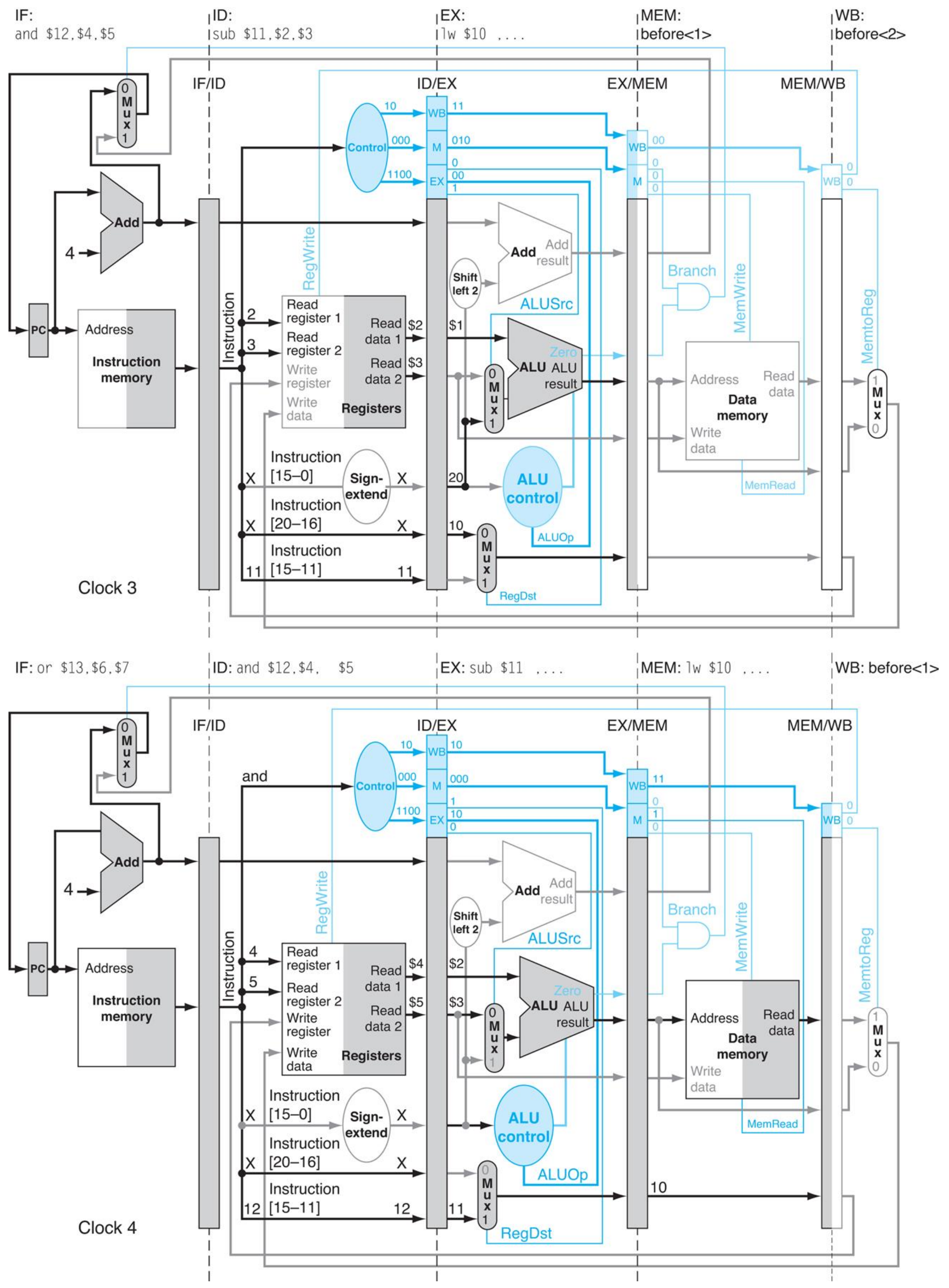


FIGURE E4.14.11 **Clock cycles 1 and 2.** The phrase "*before <i>*" means the *i*th instruction before `lw`. The `lw` instruction in the top datapath is in the IF stage. At the end of the clock cycle, the `lw` instruction is in the IF/ID pipeline registers. In

the second clock cycle, seen in the bottom datapath, the `lw` moves to the ID stage, and `sub` enters in the IF stage. Note that the values of the instruction fields and the selected source registers are shown in the ID stage. Hence register `$1` and the constant 20, the operands of `lw`, are written into the ID/EX pipeline register. The number 10, representing the destination register number of `lw`, is also placed in ID/EX. Bits 15–11 are 0, but we use *X* to show that a field plays no role in a given instruction. The top of the ID/EX pipeline register shows the control values for `lw` to be used in the remaining stages. These control values can be read from the `lw` row of the table in Figure 4.18.

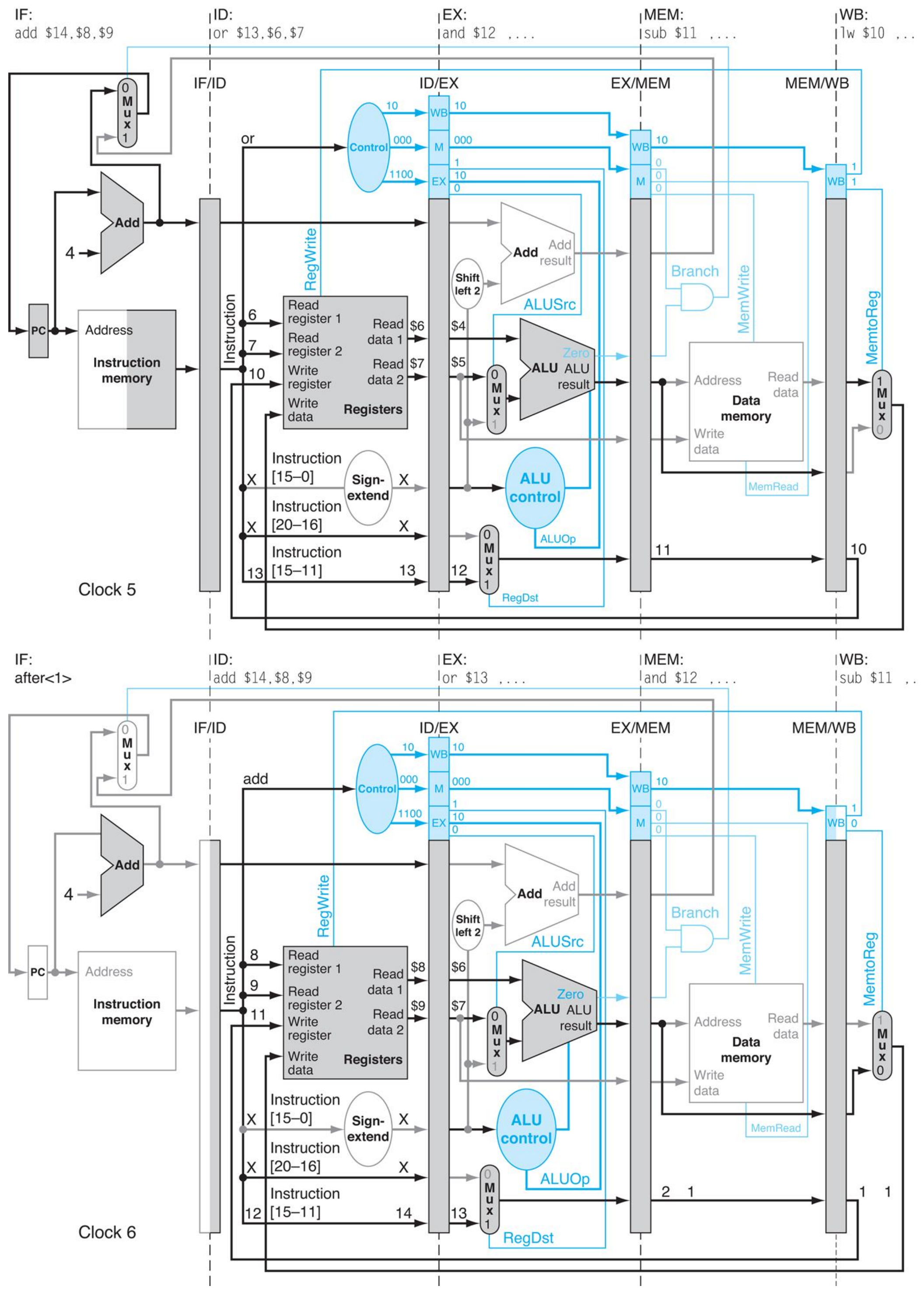


FIGURE E4.14.12 **Clock cycles 3 and 4.**
In the top diagram, `lw` enters the EX stage in the third clock cycle, adding `$1` and 20 to form the address in the EX/MEM pipeline register. (The `lw`

instruction is written `lw $10`,… upon reaching EX, because the identity of instruction operands is not needed by EX or the subsequent stages. In this version of the pipeline, the actions of EX, MEM, and WB depend only on the instruction and its destination register or its target address.) At the same time, `sub` enters ID, reading registers `$2` and `$3`, and the `and` instruction starts IF. In the fourth clock cycle (bottom datapath), `lw` moves into MEM stage, reading memory using the value in EX/MEM as the address. In the same clock cycle, the ALU subtracts `$3` from `$2` and places the difference into EX/MEM registers `$4` and `$5` are read during ID and the `or` instruction enters IF. The two diagrams show the control signals being created in the ID stage and peeled off as they are used in subsequent pipe stages.

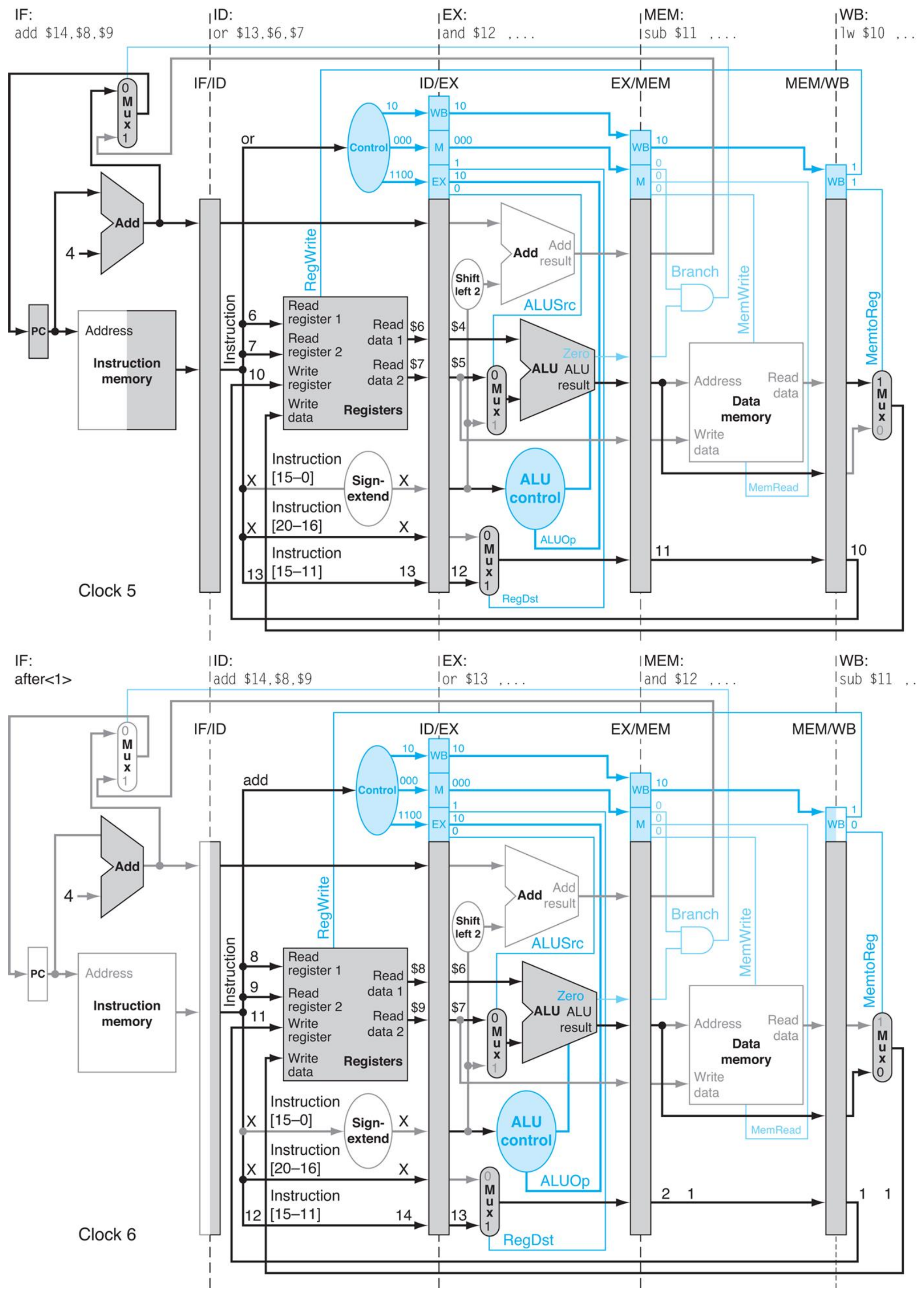


FIGURE E4.14.13 **Clock cycles 5 and 6.** With `add`, the final instruction in this example, entering IF in the top datapath, all instructions are engaged. By writing the data in MEM/WB into

register 10, `lw` completes; both the data and the register number are in MEM/WB. In the same clock cycle, `sub` sends the difference in EX/MEM to MEM/WB, and the rest of the instructions move forward. In the next clock cycle, `sub` selects the value in MEM/WB to write to register number 11, again found in MEM/WB. The remaining instructions play follow-the-leader: the ALU calculates the OR of `$6` and `$7` for the `or` instruction in the EX stage, and registers `$8` and `$9` are read in the ID stage for the `add` instruction. The instructions after `add` are shown as inactive just to emphasize what occurs for the five instructions in the example. The phrase “after<i>” means the *i*th instruction after `add`.

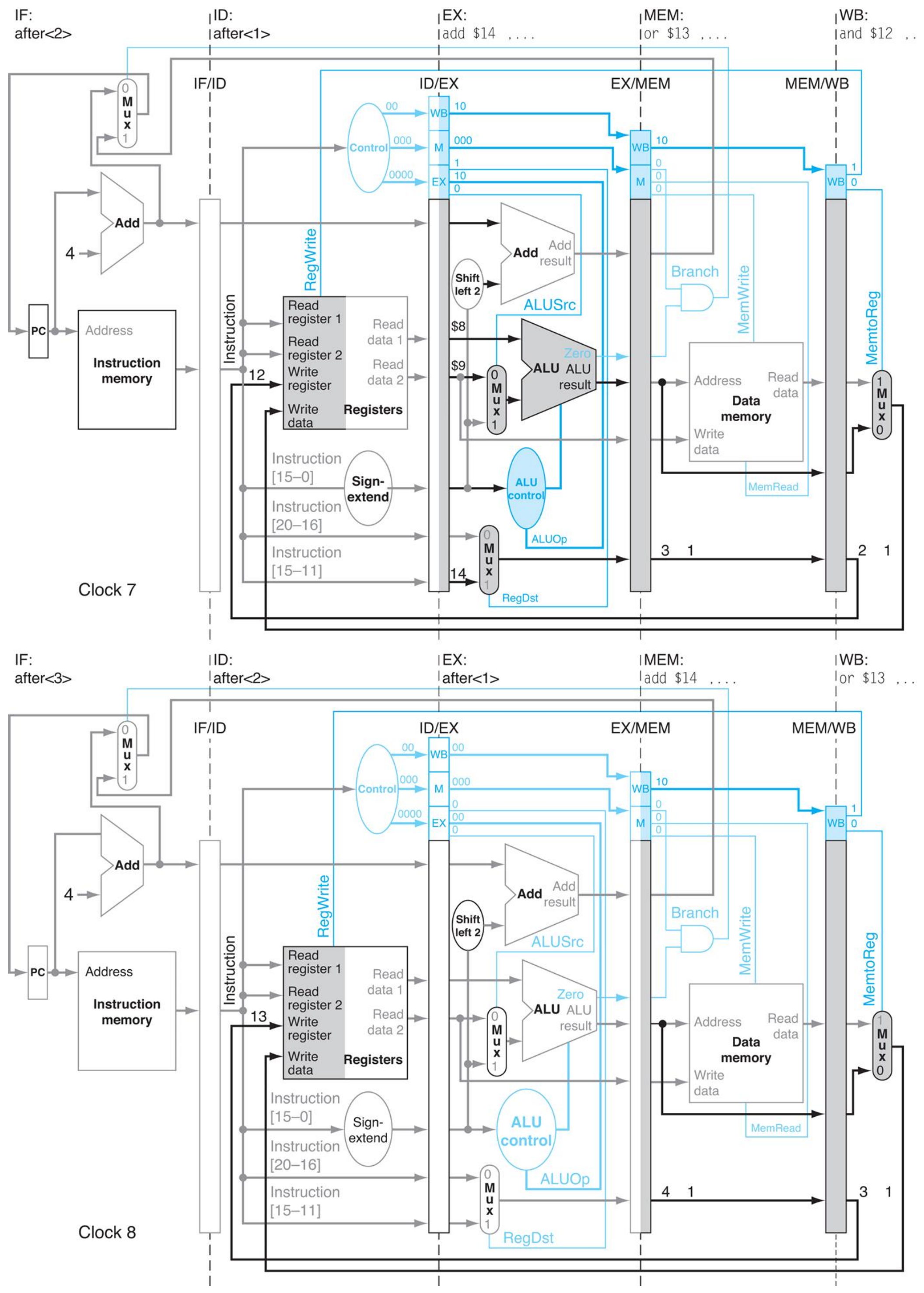


FIGURE E4.14.14 **Clock cycles 7 and 8.**
In the top datapath, the `add` instruction brings up the rear, adding the values corresponding to registers `$8` and `$9` during the EX stage. The

result of the `or` instruction is passed from EX/MEM to MEM/WB in the MEM stage, and the WB stage writes the result of the and instruction in MEM/WB to register `$12`. Note that the control signals are deasserted (set to 0) in the ID stage, since no instruction is being executed. In the following clock cycle (lower drawing), the WB stage writes the result to register `$13`, thereby completing `or`, and the MEM stage passes the sum from the `add` in EX/MEM to MEM/WB. The instructions after `add` are shown as inactive for pedagogical reasons.

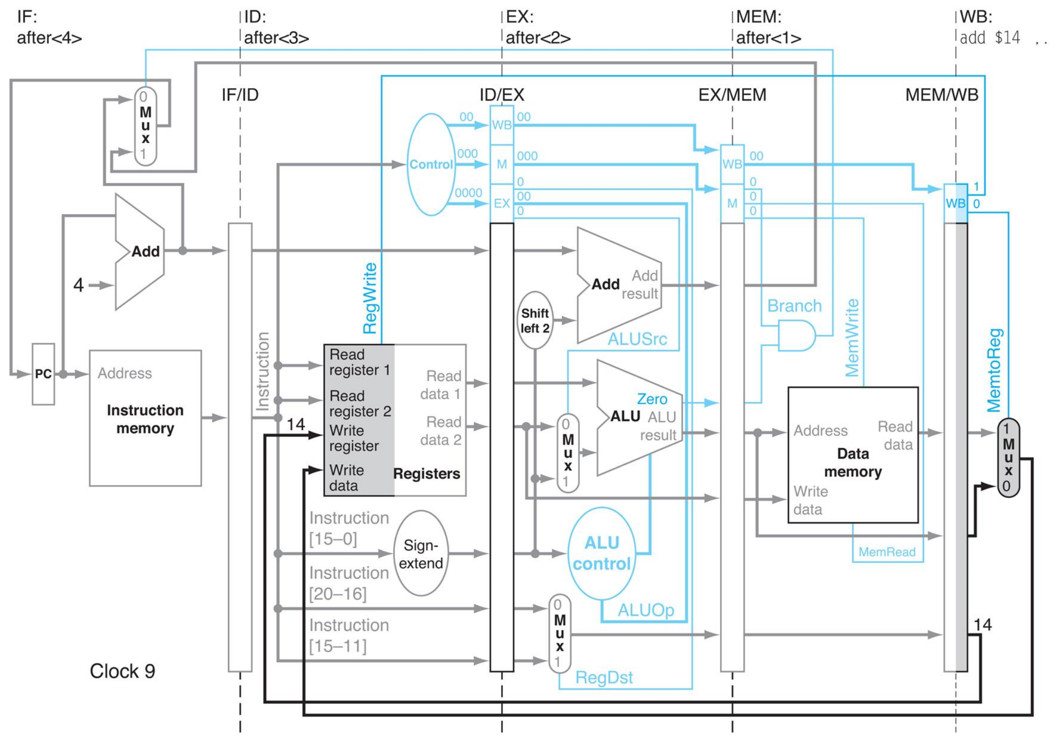


FIGURE E4.14.15 **Clock cycle 9.** The WB stage writes the `sum` in MEM/WB into register `$14`, completing `add` and the five-instruction sequence. The instructions after `add` are shown as inactive for pedagogical reasons.

Forwarding Illustrations

We can use the single-clock-cycle pipeline diagrams to show how forwarding operates, as well as how the control activates the forwarding paths. Consider the following code sequence in which the dependences have been highlighted:

`sub`	`$2, $1, $3`
`and`	`$4, $2, $5`
`or`	`$4, $4, $2`
`add`	`$9, $4, $2`

Figures e4.14.16 and e4.14.17 show the events in clock cycles 3–6 in the execution of these instructions.

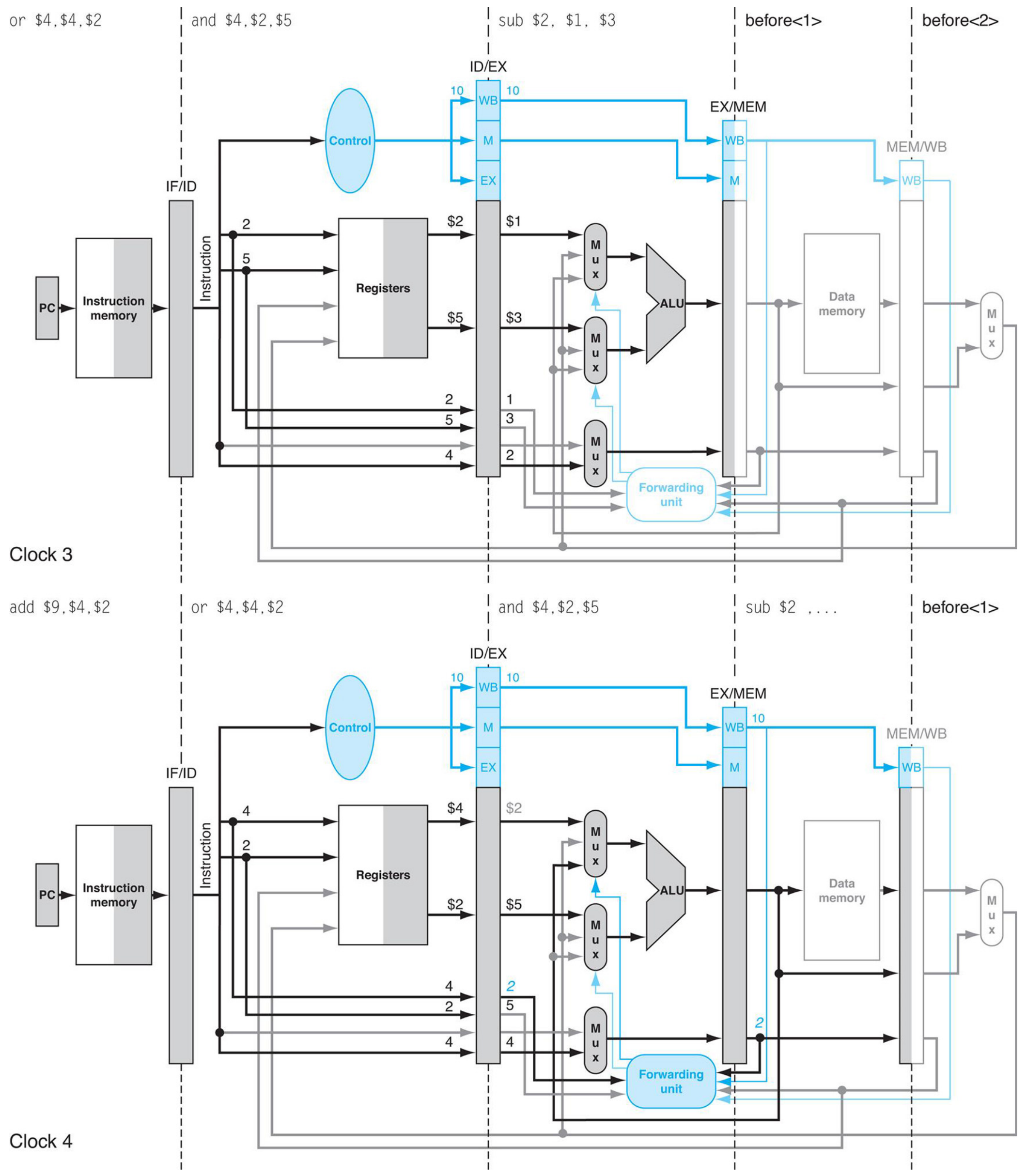


FIGURE E4.14.16 **Clock cycles 3 and 4 of the instruction sequence on page 4.13-26.** The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard. The forwarding unit is highlighted by shading it when it is forwarding data to the ALU. The instructions before `sub` are shown as inactive just to emphasize what occurs for the four instructions in the example. Operand names are used in EX for control of forwarding;

thus they are included in the instruction label for EX. Operand names are not needed in MEM or WB, so … is used. Compare this with Figures e4.14.12 through e4.14.15, which show the datapath without forwarding where ID is the last stage to need operand information.

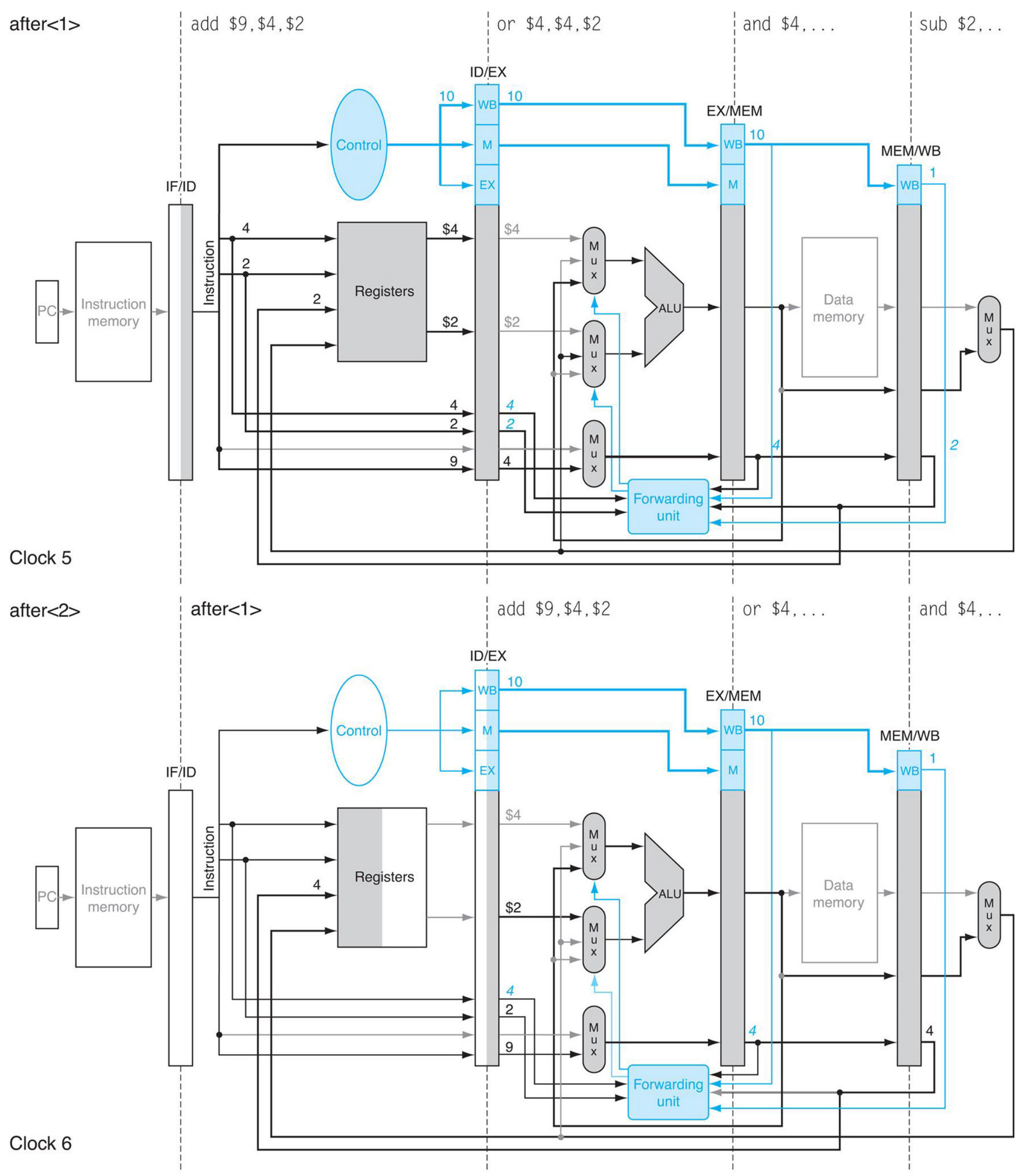


FIGURE E4.14.17 **Clock cycles 5 and 6 of the instruction sequence on page 4.13-26.** The forwarding unit is highlighted when it is forwarding data to the ALU. The two instructions after `add` are shown as inactive just to emphasize what occurs for the four instructions in the example. The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard.

In clock cycle 4, the forwarding unit sees the writing by the `sub` instruction of register `$2` in the MEM stage, while the and instruction in the EX stage is reading register `$2`. The forwarding unit selects the EX/MEM pipeline register instead of the ID/EX pipeline register as the upper input to the ALU to get the proper value for register `$2`. The following or instruction reads register $4, which is written by the and instruction, and register `$2`, which is written by the `sub` instruction.

Thus, in clock cycle 5, the forwarding unit selects the EX/MEM pipeline register for the upper input to the ALU and the MEM/WB pipeline register for the lower input to the ALU. The following `add` instruction reads both register `$4`, the target of the and instruction, and register `$2`, which the sub instruction has already written. Notice that the prior two instructions both write register `$4`, so the forwarding unit must pick the immediately preceding one (MEM stage).

In clock cycle 6, the forwarding unit thus selects the EX/MEM pipeline register, containing the result of the `or` instruction, for the upper ALU input but uses the nonforwarding register value for the lower input to the ALU.

Illustrating Pipelines with Stalls and Forwarding

We can use the single-clock-cycle pipeline diagrams to show how the control for stalls works. Figures e4.14.18 through e4.14.20 show the single-cycle diagram for clocks 2 through 7 for the following code sequence (dependences highlighted):

`1w`	`$2, 20($1)`
`and`	`$4, $2,$5`
`or`	`$4, $4,$2`
`add`	`$9, $4,$2`

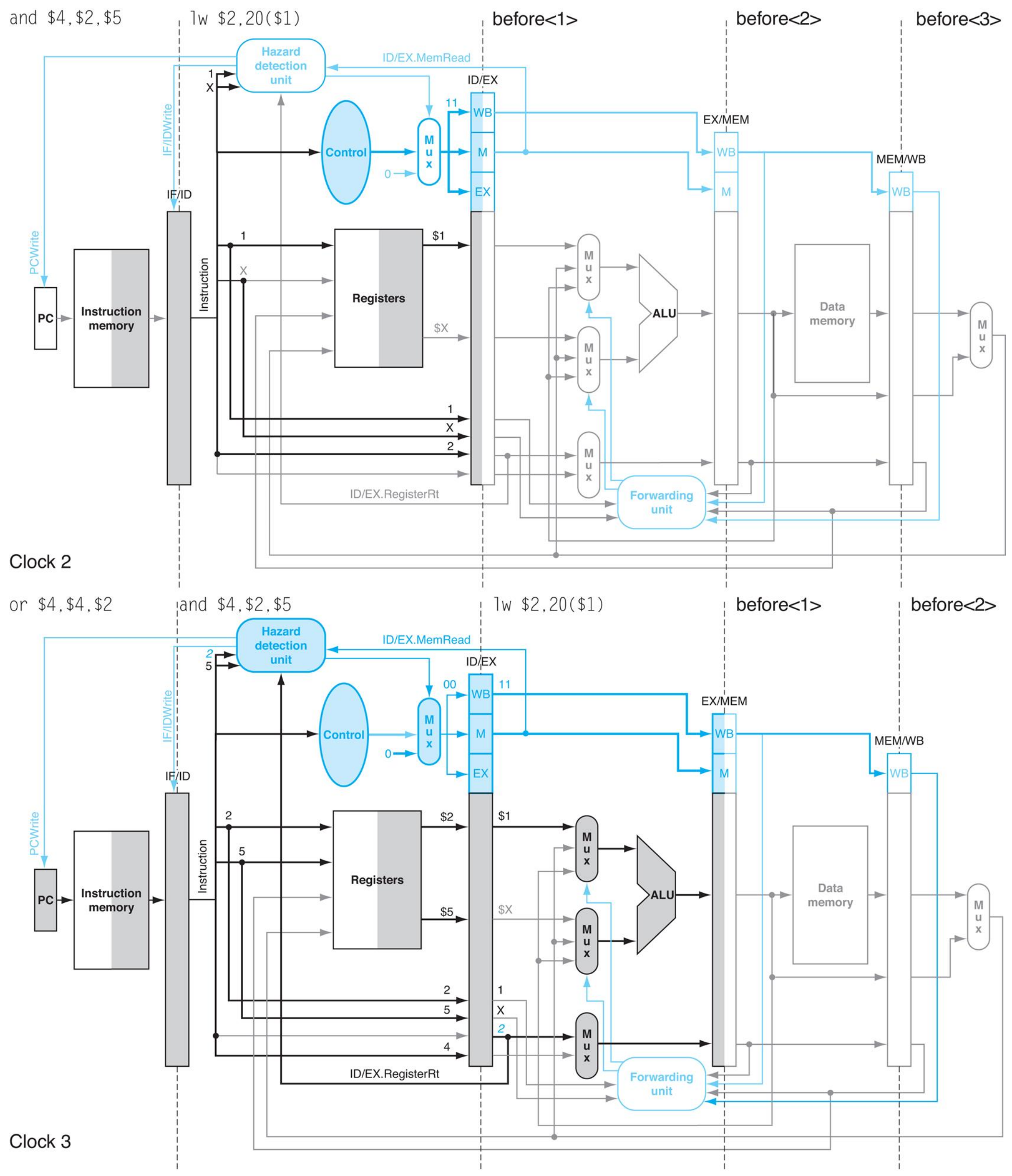


FIGURE E4.14.18 **Clock cycles 2 and 3 of the instruction sequence on page 4.13-26 with a load replacing `sub`.**
The bold lines are those active in a clock cycle, the italicized register numbers in color indicate a hazard, and the … in the place of operands means that their identity is information not needed by that stage. The values of the significant control lines, registers, and register numbers are labeled in the figures. The `and`

instruction wants to read the value created by the `lw` instruction in clock cycle 3, so the hazard detection unit stalls the `and` and `or` instructions. Hence, the hazard detection unit is highlighted.

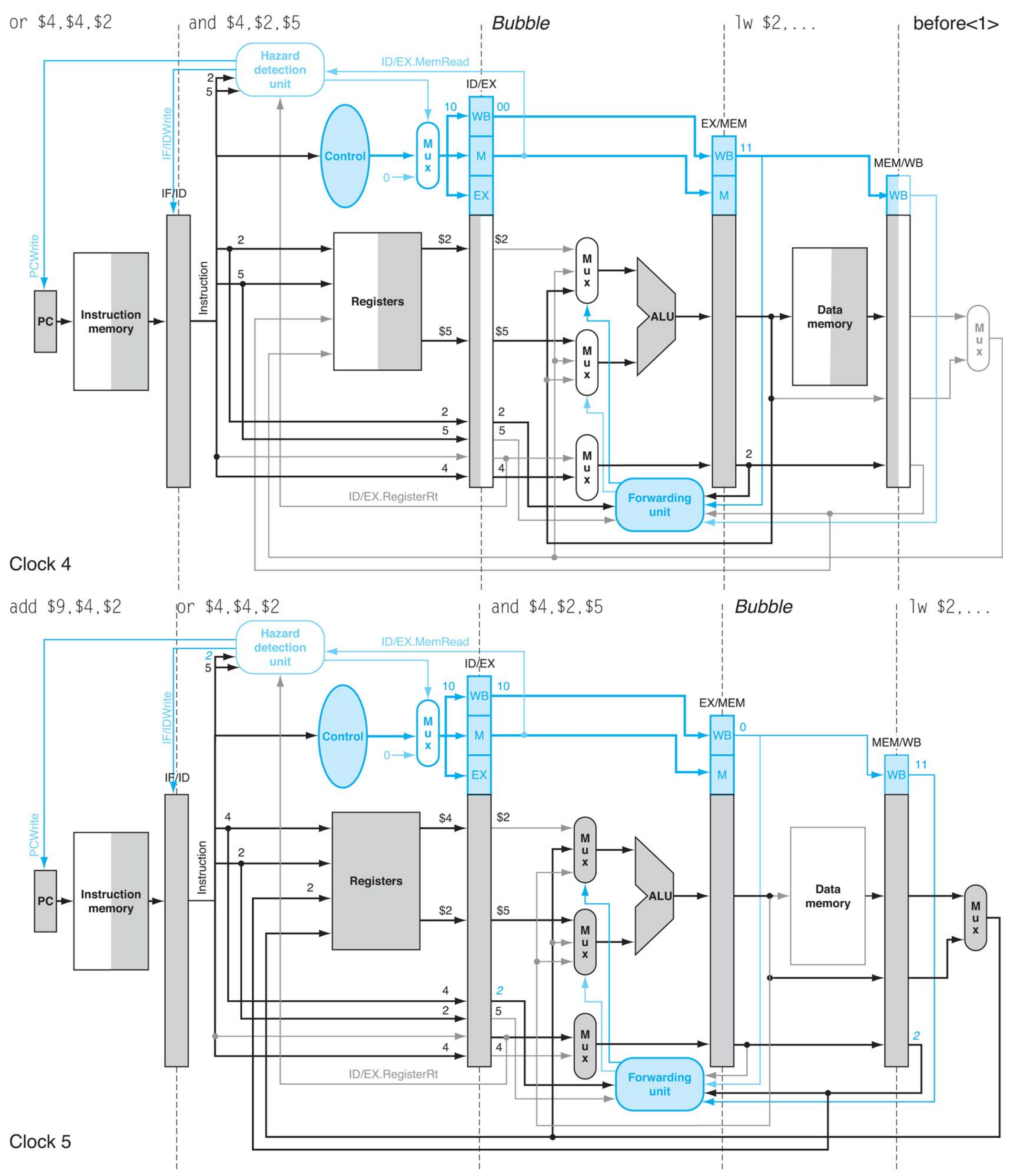


FIGURE E4.14.19 **Clock cycles 4 and 5 of the instruction sequence on page 4.13-26 with a load replacing `sub`.**
The bubble is inserted in the pipeline in clock cycle 4, and then the `and` instruction is allowed to proceed in clock cycle 5. The forwarding unit is highlighted in clock cycle 5 because it is forwarding data from `lw` to the ALU. Note that in clock cycle 4, the forwarding unit forwards the address of the `lw` as if it were the contents of

register `$2`; this is rendered harmless by the insertion of the bubble. The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard.

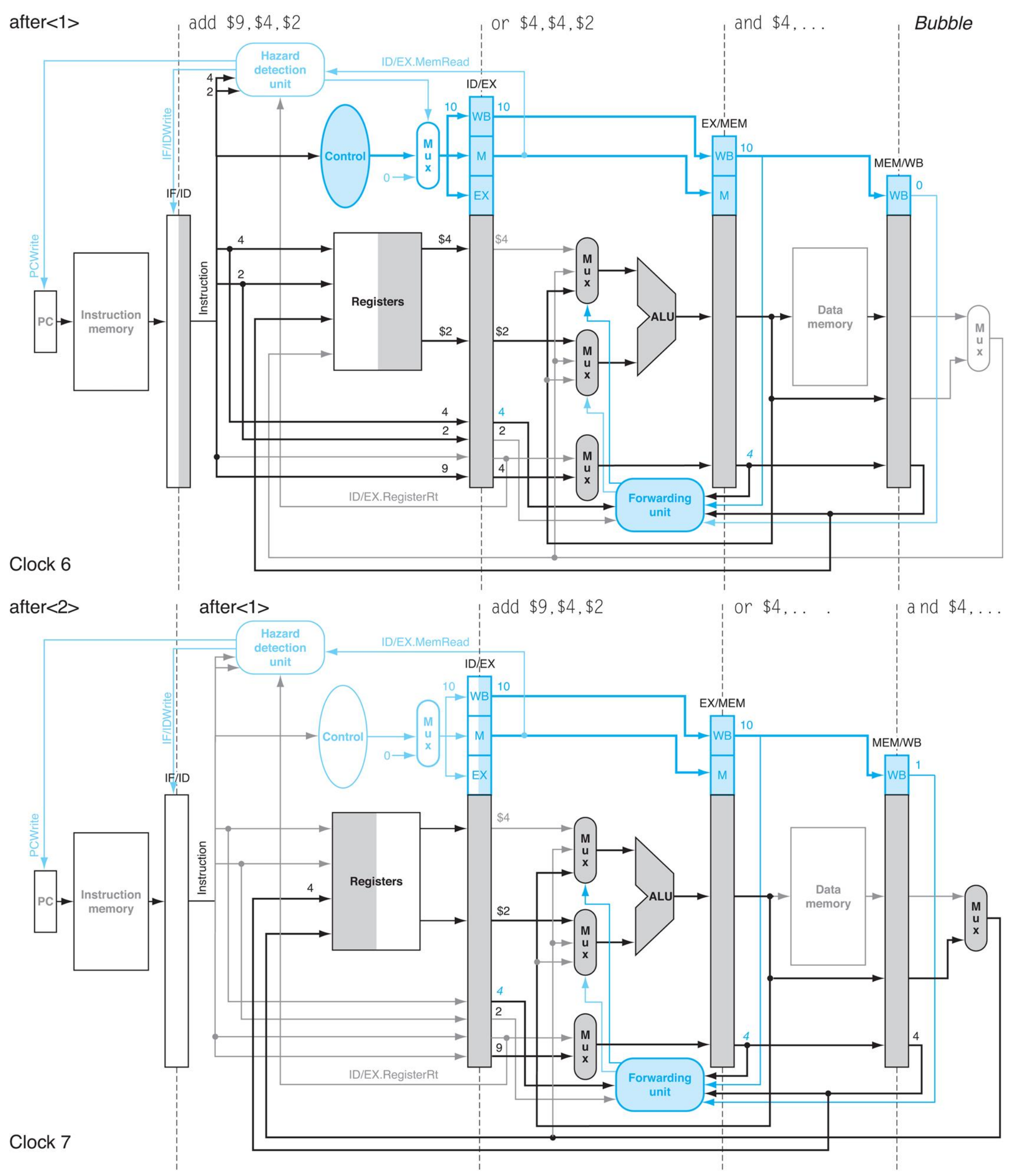


FIGURE E4.14.20 **Clock cycles 6 and 7 of the instruction sequence on page 4.13-26 with a load replacing `sub`.**
Note that unlike in Figure e4.14.17, the stall allows the `lw` to complete, and so there is no forwarding from MEM/WB in clock cycle 6. Register `$4` for the `add` in the EX stage still depends on the result from or in EX/MEM, so the forwarding unit passes the result to the ALU. The bold lines show ALU input lines active in a clock

cycle, and the italicized register numbers indicate a hazard. The instructions after `add` are shown as inactive for pedagogical reasons.

4.15 Fallacies and Pitfalls

Fallacy: Pipelining is easy.

Our books testify to the subtlety of correct pipeline execution. Our advanced book had a pipeline bug in its first edition, despite its being reviewed by more than 100 people and being class-tested at 18 universities. The bug was uncovered only when someone tried to build the computer in that book. The fact that the Verilog to describe a pipeline like that in the Intel Core i7 will be many thousands of lines is an indication of the complexity. Beware!

Fallacy: Pipelining ideas can be implemented independent of technology.

When the number of transistors on-chip and the speed of transistors made a five-stage pipeline the best solution, then the delayed branch (see the first *Elaboration* on page 267) was a simple solution to control hazards. With longer pipelines, superscalar execution, and dynamic branch prediction, it is now redundant. In the early 1990s, dynamic pipeline scheduling took too many resources and was not required for high performance, but as transistor budgets continued to double due to Moore's Law logic became much faster than memory, then multiple functional units and dynamic pipelining made more sense. Today, concerns about power are leading to less aggressive designs.

Pitfall: Failure to consider instruction set design can adversely impact pipelining.

Many of the difficulties of pipelining arise because of instruction set complications. Here are some examples:

- Widely variable instruction lengths and running times can lead to imbalance among pipeline stages and severely complicate hazard detection in a design pipelined at the instruction set level. This problem was overcome, initially in the DEC VAX 8500 in the late 1980s, using the micro-operations and micropipelined scheme that the Intel Core i7 employs today. Of course, the overhead of translation and maintaining correspondence between the micro-operations and the actual instructions remains.
- Sophisticated addressing modes can lead to different sorts of problems. Addressing modes that update registers complicate hazard detection. Other addressing modes that require multiple memory accesses substantially complicate pipeline control and make it difficult to keep the pipeline flowing smoothly.
- Perhaps the best example is the DEC Alpha and the DEC NVAX. In comparable technology, the newer instruction set architecture of the Alpha allowed an implementation whose performance is more than twice as fast as NVAX. In another example, Bhandarkar and Clark [1991] compared the MIPS M/2000 and the DEC VAX 8700 by counting clock cycles of the SPEC benchmarks; they concluded that although the MIPS M/2000 executes more instructions, the VAX on average executes 2.7 times as many clock cycles, so the MIPS is faster.

4.16 Concluding Remarks

Nine-tenths of wisdom consists of being wise in time.

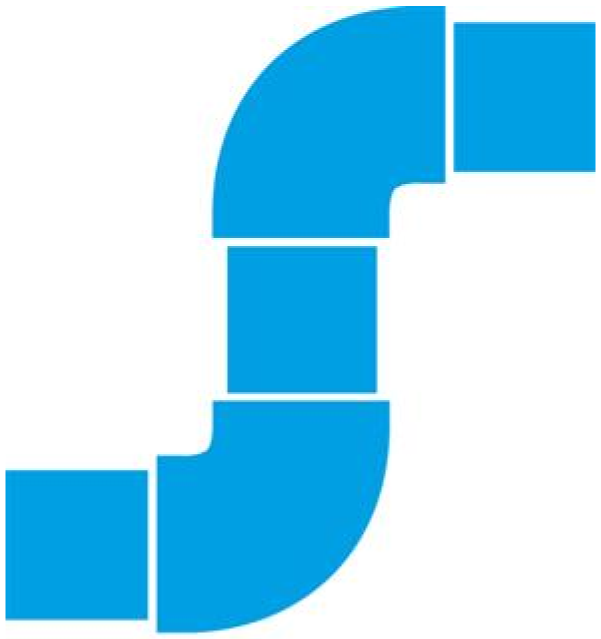
American proverb

As we have seen in this chapter, both the datapath and control for a processor can be designed starting with the instruction set architecture and an understanding of the basic characteristics of the technology. In Section 4.3, we saw how the datapath for a MIPS processor could be constructed based on the architecture and the decision to build a single-cycle implementation. Of course, the underlying technology also affects many design decisions by dictating what components can be used in the datapath, as well as whether a single-cycle implementation even makes sense.

Pipelining improves throughput but not the inherent execution time, or **instruction latency**, of instructions; for some instructions, the latency is similar in length to the single-cycle approach. Multiple instruction issue adds additional datapath hardware to allow multiple instructions to begin every clock cycle, but at an increase in effective latency. Pipelining was presented as reducing the clock cycle time of the simple single-cycle datapath. Multiple instruction issue, in comparison, clearly focuses on reducing *clock cycles per instruction* (CPI).

instruction latency

The inherent execution time for an instruction.



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

PREDICTION

We showed that unrolling the DGEMM loop four times exposed more instructions that could take advantage of the out-of-order execution engine of the Core i7 to more than double performance.

The switch to longer pipelines, multiple instruction issue, and dynamic scheduling in the mid-1990s helped sustain the 60% per year processor performance increase that started in the early 1980s. As mentioned in Chapter 1, these microprocessors preserved the sequential programming model, but they eventually ran into the power wall. Thus, the industry was forced to switch to multiprocessors, which exploit parallelism at much coarser levels (the subject of Chapter 6). This trend has also caused designers to reassess the energy-performance implications of some of the inventions since the mid-1990s, resulting in a simplification of pipelines in the more recent versions of microarchitectures.

To sustain the advances in processing performance via parallel processors, Amdahl’s law suggests that another part of the system will become the bottleneck. That bottleneck is the topic of the next chapter: the **memory hierarchy**.

4.17 Historical Perspective and Further Reading

This section, which appears online, discusses the history of the first pipelined processors, the earliest superscalars, and the development of out-of-order and speculative techniques, as well as important developments in the accompanying compiler technology.

4.17 Historical Perspective and Further Reading

supercomputer: *Any machine still on the drawing board.*

Stan Kelly-Bootle, *The Devil's DP Dictionary*, 1981

This section discusses the history of the first pipelined processors, the earliest superscalars, and the development of out-of-order and speculative techniques, as well as important developments in the accompanying compiler technology.

It is generally agreed that one of the first general-purpose pipelined computers was Stretch, the IBM 7030 (Figure e4.17.1). Stretch followed the IBM 704 and had a goal of being 100 times faster than the 704. The goals were a "stretch" of the state of the art at that time—hence the nickname. The plan was to obtain a factor of 1.6 from overlap ping fetch, decode, and exe cute by using a four-stage pipeline. Apparently, the rest was to come from much more hardware and faster logic. Stretch was also a training ground for both the architects of the IBM 360, Gerrit Blaauw and Fred Brooks, Jr., and the architect of the IBM RS/6000, John Cocke. Both Brooks and Cocke later won ACM A.M. Turing Awards, the highest honor in computer science.

FIGURE E4.17.1 **The Stretch computer, one of the first pipelined computers.**

Control Data Corporation (CDC) delivered what is considered to be the first supercomputer, the CDC 6600, in 1964 (Figure e4.17.2). The core instructions of Cray's subsequent computers have many similarities to those of the original CDC 6600. The CDC 6600 was unique in many ways. The interaction between pipelining and instruction set design was understood, and the instruction set was kept simple to promote pipelining. The CDC 6600 also used an advanced pack aging technology. James Thornton's book [1970] provides an excellent description of the entire computer, from technology to architecture, and includes a foreword by Seymour Cray. (Unfortunately, this book is currently out of print.) Jim Smith, then working at CDC, developed the original 2-bit branch prediction scheme and explored several techniques for enhancing instruction issue. Cray, Thornton, and Smith have each won the ACM Eckert-Mauchly Award (in 1989, 1994, and 1999, respectively).

FIGURE E4.17.2 The CDC 6600, the first supercomputer.

The IBM 360/91 introduced many new concepts, including dynamic detection of memory hazards, generalized forwarding, and reservation stations (Figure e4.17.3). The approach is normally named *Tomasulo's algorithm*, after an engineer who worked on the project. The team that created the 360/91 was led by Michael Flynn, who was given the 1992 ACM Eckert-Mauchly Award, in part for his contributions to the IBM 360/91; in 1997, the same award went to Robert Tomasulo for his pioneering work on out-of-order processing.

FIGURE E4.17.3 **The IBM 360/91 pushed the state of the art in pipe lined execution when it was unveiled in 1966.**

The internal organization of the 360/91 shares many features with the Pentium III and Pentium 4, as well as with several other microprocessors. One major difference was that there was no branch prediction in the 360/91 and hence no speculation. Another major difference was that there was no commit unit, so once the instructions finished execution, they updated the registers. Out-of-order instruction commit led to *imprecise interrupts*, which proved to be unpopular and led to the commit units in dynamically scheduled pipelined processors since that time. Although the 360/91 was not a success, its key ideas were resurrected later and exist in some form in the majority of desktop and server microprocessors since 2000.

Improving Pipelining Effectiveness and Adding Multiple Issue

The RISC processors refined the notion of compiler-scheduled pipelines in the early 1980s. The concepts of delayed branches and delayed loads—common in microprogramming—were extended into the high-level architecture. In fact, the Stanford processor that led to the commercial

MIPS architecture was called “Microprocessor without Interlocked Pipelined Stages” because it was up to the assembler or compiler to avoid data hazards.

In addition to its contribution to the development of the RISC concepts, IBM did pioneering work on multiple issue. In the 1960s, a project called ACS was under way. It included multiple-instruction issue concepts and the notion of integrated compiler and architecture design, but it never reached product stage. The earliest proposal for a superscalar processor that dynamically makes issue decisions was by John Cocke; he described the key ideas in several talks in the mid-1980s and, with Tilak Agarwala, coined the name *superscalar*. This original design was a two-issue machine named Cheetah, which was followed by a more widely discussed four-issue machine named America. The IBM Power-1 architecture, used in the RS/6000 line, is based on these ideas, and the PowerPC is a variation of the Power-1 architecture. Cocke won the Turing Award, the highest award in computer science and engineering, for his architecture work.

Static multiple issue, as exemplified by the *long instruction word* (LIW) or sometimes *very long instruction word* (VLIW) approaches, appeared in real designs before the superscalar approach. In fact, the earliest multiple-issue machines were special-purpose attached processors designed for scientific applications. Culler Scientific and Floating Point Systems were two of the most prominent manufacturers of such computers. Another inspiration for the use of multiple operations per instruction came from those working on microcode compilers. Such inspiration led to a research project at Yale led by Josh Fisher, who coined the term VLIW. Cydrome and Multiflow were two early companies involved in building mini-supercomputers using processors with multiple-issue capability. These processors, built with bit-slice and multiple-chip gate array implementations, arrived on the market at the same time as the first RISC microprocessors. Despite some promising performance on high-end scientific codes, the much better cost/performance of the microprocessor-based computers doomed the first generation of VLIW computers. Bob Rau and Josh Fisher won the Eckert-Mauchly Award in 2002 and 2003, respectively, for their contributions to the development of multiple processors and software techniques to exploit ILP.

The very beginning of the 1990s saw the first superscalar processors using static scheduling and no speculation, including versions of the MIPS and PowerPC architectures. The early 1990s also saw important research at a number of universities, including Wisconsin, Stanford, Illinois, and Michigan, focused on techniques for exploiting additional ILP through multiple issue with and without speculation. These research insights were used to build dynamically scheduled, speculative processors, including the Motorola 88110, MIPS R10000, DEC Alpha 21264, PowerPC 603, and the Intel Pentium Pro, Pentium III, and Pentium 4.

In 2001, Intel introduced the IA-64 architecture and its first implementation, Itanium. Itanium represented a return to a more compiler-intensive approach that they called EPIC. EPIC represented a considerable enhancement over the early VLIW architectures, removing many of their drawbacks. It has had modest sales. In 2019 Intel announced that it would be discontinuing its Itanium 9700-series processors, the last EPIC chips on the market.

Compiler Technology for Exploiting ILP

Successful development of processors to exploit ILP has depended on progress in compiler technology. The concept of loop-unrolling was understood early, and a number of companies and researchers—including Floating Point Systems, Cray, and the Stanford MIPS project—developed compilers that made use of loop-unrolling and pipeline scheduling to improve instruction through put. A special purpose processor called WARP, designed at Carnegie Mellon University, inspired the development of software pipelining, an approach that symbolically unrolls loops.

To exploit higher levels of ILP, more aggressive compiler technology was needed. The VLIW project at Yale developed the concept of trace scheduling that Multi-flow implemented in their compilers. Trace scheduling relies on aggressive loop unrolling and path prediction to compile favored execution traces efficiently. The Cydrome designers created early versions of predication and support for software pipelining. Hwu at Illinois worked on extended versions of loop-unrolling, called *superblocks*, and techniques for compiling with predication. The concepts from Multiflow, Cydrome, and the research group at Illinois served as the architectural and compiler basis for the IA-64 architecture.

Further Reading

Bhandarkar, D. and D. W. Clark [1991]. “Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations,” *Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Palo Alto, CA, 310–19.

A quantitative comparison of RISC and CISC written by scholars who argued for CISCs as well as built them; they conclude that MIPS is between 2 and 4 times faster than a VAX built with similar technology, with a mean of 2.7.

Fisher, J. A. and B. R. Rau [1993]. *Journal of Supercomputing* (January), Kluwer.

This entire issue is devoted to the topic of exploiting ILP. It contains papers on both the architecture and software and is a wonderful source for further references.

Hennessy, J. L. and D. A. Patterson [2001]. *Computer Architecture: A Quantitative Approach*, fourth edition, Morgan Kaufmann, San Francisco.

Chapter 2 and Appendix A *go into considerably more detail about pipelined processors (almost 200 pages), including superscalar processors and VLIW processors. Appendix G describes Itanium.*

Jouppi, N. P. and D. W.Wall [1989]. “Available instruction-level parallelism for superscalar and superpipelined processors,” *Proc. Third Conf. on Architectural Sup port for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, 272–82.

A comparison of deeply pipelined (also called superpipelined) and superscalar systems.

Kogge, P. M. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York.

A formal text on pipelined control, with emphasis on underlying principles.

Russell, R. M. [1978]. “The CRAY-1 computer system,” *Comm. of the ACM* 21:1 (January), 63–72.

A short summary of a classic computer that uses vectors of operations to remove pipeline stalls.

Smith, A. and J. Lee [1984]. "Branch prediction strategies and branch target buffer design," *Computer* 17:1 (January), 6–22.

An early survey on branch prediction.

Smith, J. E. and A. R. Plezkun [1988]. "Implementing precise interrupts in pipelined processors," *IEEE Trans. on Computers* 37:5 (May), 562–73.

Covers the difficulties in interrupting pipelined computers.

Thornton, J. E. [1970]. *Design of a Computer. The Control Data 6600*, Glenview, IL: Scott, Foresman.

A classic book describing a classic computer, considered the first supercomputer.

4.18 Self Study

While higher performance processors have much longer pipelines than the five stages, some very low cost or low energy processors have shorter pipelines. Assume the same timing of the data path components as in Figures 4.26 and 4.27.

Three-Stage Pipe. How would you split the data path in stages if it was a three-stage pipeline instead of five?

Clock Rate. Ignoring an impact of pipeline registers or forwarding logic on the clock cycle time, what are the clock rates for the five-stage versus the three-stage pipelines? Assume the same timing of the data path components as in Figures 4.26 and 4.27.

Register Write/Read Data Hazards? Do you still have them in with three stages? If so, will forwarding fix them?

Load-Use Data Hazards? Do you still have them in with three stages? Do you need to stall the pipeline or can forwarding fix them?

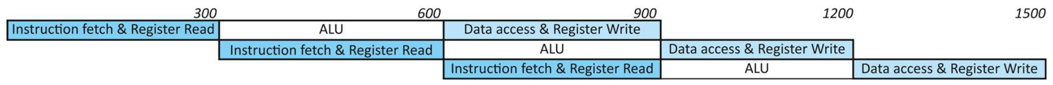
Control Hazards? Do you still have them in with three stages? If so, how can you reduce their impact?

CPI. Will the clocks per instruction of the three-stage pipeline be higher or lower than the five-stage pipeline?

Self-Study Answers

Three-Stage Pipe. While there are multiple possible solutions, this one is a sensible split:

1. Instruction fetch, register read (300 ps)
2. ALU (200 ps)
3. Data access, register write (300 ps)


300
600
900
1200
1500
Instruction fetch & Register Read
ALU
Data access & Register Write
Instruction fetch & Register Read
ALU
Data access & Register Write
Instruction fetch & Register Read
ALU
Data access & Register Write

Clock Rate. Figure 4.27 shows the clock cycle time of the five-stage pipeline is 200 ps, so the clock rate is 1/200 ps or 5 GHz. The worst case stage for this three-stage pipeline is 300 ps, so the clock rate is 1/300 ps or 3.33 GHz.

Register Write/Read Data Hazards. As the pipeline drawing previously shows, there is still a write/read hazard. The first instruction does not write the data to register the third stage, but the next instruction needs the new value at the beginning of its second stage. The forwarding solution in Section 4.8 works fine for the three-stage pipeline, as the ALU result of the prior instruction is ready before the beginning of its second stage.

Load-Use Data Hazards. Even with three stages, we have to stall one clock cycle for a load use hazard as in Section 4.8. The data is not available until the third stage of the load instruction, but the following instruction needs the new data at the beginning of its second stage.

Control Hazards. This hazard is where the three stage pipeline shines. We can use the same optimization as in Section 4.9 to calculate the branch address and compare registers for equality before the ALU stage as we did in Figure 4.62. That calculation is performed before the instruction fetch of the following instruction, so early branch logic resolves the control hazard with no pipeline penalty.

CPI. The average clocks per instruction will shrink (get better) with a three-stage pipeline for a couple of reasons:

- Given the clock cycle is longer, it will take fewer clock cycles to access DRAM memory, which will affect the CPI when we have a miss in the cache (see Chapter 5).

■ Branches will always execute in one clock cycle, whereas any software and hardware scheme to accelerate branches with five stages will fail some of the time, increasing the effective CPI.

■ Our clock cycle time is longer for the ALU, which may allow some complex operations that might take more than one clock cycle in the five-stage pipeline. For example, integers that multiply or divide might need fewer of these longer clock cycles than does the five-stage pipeline.

4.19 Exercises

4.1 Consider the following instruction:

Instruction: `and rd, rsl, rs2`

Interpretation: `Reg[rd] = Reg[rs1] AND Reg[rs2]`

4.1.1 [5] <§4.3> What are the values of control signals generated by the control in Figure 4.10 for this instruction?

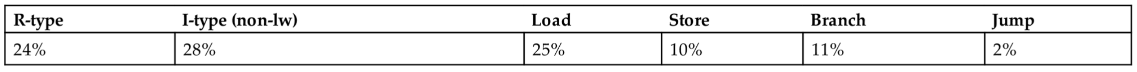
4.1.2 [5] <§4.3> Which resources (blocks) perform a useful function for this instruction?

4.1.3 [10] <§4.3> Which resources (blocks) produce no output for this instruction? Which resources produce output that is not used?

4.2 [10] <§4.4> Explain each of the “don’t cares” in Figure 4.18.

4.3 Consider the following instruction mix:

R-type	I-type (non-lw)	Load	Store	Branch	Jump
24%	28%	25%	10%	11%	2%



R-type	I-type (non-lw)	Load	Store	Branch	Jump
24%	28%	25%	10%	11%	2%

4.3.1 [5] <§4.4> What fraction of all instructions use data memory?

4.3.2 [5] <§4.4> What fraction of all instructions use instruction memory?

4.3.3 [5] <§4.4> What fraction of all instructions use the sign extend?

4.3.4 [5] <§4.4> What is the sign extend doing during cycles in which its output is not needed?

4.4 When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits. A very common defect is for one signal wire to get “broken” and always register a logical 0. This is often called a “stuck-at-0” fault.

4.4.1 [5] <§4.4> Which instructions fail to operate correctly if the `MemToReg` wire is stuck at 0?

4.4.2 [5] <§4.4> Which instructions fail to operate correctly if the `ALUSrc` wire is stuck at 0?

4.5 In this exercise, we examine in detail how an instruction is executed in a single-cycle datapath. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word: `0x00c6ba23`.

4.5.1 [10] <§4.4> What are the values of the ALU control unit’s inputs for this instruction?

4.5.2 [5] <§4.4> What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

4.5.3 [10] <§4.4> For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at `Reg [xn]`.

4.5.4 [10] <§4.4> What are the input values for the ALU and the two add units?

4.5.5 [10] <§4.4> What are the values of all inputs for the registers unit?

4.6 Section 4.4 does not discuss I-type instructions like `addi` or `andi`.

4.6.1 [5] <§4.4> What additional logic blocks, if any, are needed to add I-type instructions to the CPU shown in Figure 4.21? Add any necessary logic blocks to Figure 4.21 and explain their purpose.

4.6.2 [10] <§4.4> List the values of the signals generated by the control unit for `addi`. Explain the reasoning for any “don’t care” control signals.

4.7 Problems in this exercise assume that the logic blocks used to implement a processor's datapath have the following latencies:

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Singlegate	Register Read	Register Setup	Sign extend	Control
250ps	150ps	25ps	200ps	150ps	5ps	30ps	20ps	50ps	50ps

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Singlegate	Register Read	Register Setup	Sign extend	Control
250ps	150ps	25ps	200ps	150ps	5ps	30ps	20ps	50ps	50ps

"Register read" is the time needed after the rising clock edge for the new register value to appear on the output. This value applies to the PC only. "Register setup" is the amount of time a register's data input must be stable before the rising edge of the clock. This value applies to both the PC and Register File.

4.7.1 [5] <§4.4> What is the latency of an R-type instruction (i.e., how long must the clock period be to ensure that this instruction works correctly)?

4.7.2 [10] <§4.4> What is the latency of `lw`? (Check your answer carefully. Many students place extra muxes on the critical path.)

4.7.3 [10] <§4.4> What is the latency of `sw`? (Check your answer carefully. Many students place extra muxes on the critical path.)

4.7.4 [5] <§4.4> What is the latency of `beq`?

4.7.5 [5] <§4.4> What is the latency of an arithmetic, logical, or shift I-type (non-load) instruction?

4.7.6 [5] <§4.4> What is the minimum clock period for this CPU?

4.8 [10] <§4.4> Suppose you could build a CPU where the clock cycle time was different for each instruction. What would the speedup of this new CPU be over the CPU presented in Figure 4.21 given the instruction mix below?

R-type/I-type (non-lw)	lw	sw	beq
52%	25%	11%	12%

R-type/I-type (non-lw)	lw	sw	beq
52%	25%	11%	12%

4.9 Consider the addition of a multiplier to the CPU shown in Figure 4.21. This addition will add 300 ps to the latency of the ALU, but will reduce the number of instructions by 5% (because there will no longer be a need to emulate the multiply instruction).

4.9.1 [5] <§4.4> What is the clock cycle time with and without this improvement?

4.9.2 [10] <§4.4> What is the speedup achieved by adding this improvement?

4.9.3 [10] <§4.4> What is the slowest the new ALU can be and still result in improved performance?

4.10 When processor designers consider a possible improvement to the processor datapath, the decision usually depends on the cost/performance trade-off. In the following three problems, assume that we are beginning with the datapath from Figure 4.21, the latencies from Exercise 4.7, and the following costs:

I-Mem	Register File	Mux	ALU	Adder	D-Mem	Single Register	Sign extend	Sign gate	Control
1000	200	10	100	30	2000	5	100	1	500

I-Mem	Register File	Mux	ALU	Adder	D-Mem	Single Register	Sign extend	Sign gate	Control
1000	200	10	100	30	2000	5	100	1	500

Suppose doubling the number of general purpose registers from 32 to 64 would reduce the number of `lw` and `sw` instruction executed by 12%, but increase the latency of the register file from 150 ps to 160 ps and double the cost from 200 to 400. (Use the instruction mix from Exercise 4.8 and ignore the other effects on the ISA discussed in Exercise 2.18.)

4.10.1 [5] <§4.4> What is the speedup achieved by adding this improvement?

4.10.2 [10] <§4.4> Compare the change in performance to the change in cost.

4.10.3 [10] <§4.4> Given the cost/performance ratios you just calculated, describe a situation where it makes sense to add more registers and describe a situation where it doesn't make sense to add more registers.

4.11 Examine the difficulty of adding a proposed `lwi.drd, rsl, rs2` ("Load With Increment") instruction to MIPS.

Interpretation: `Reg[rd] = Mem[Reg[rs1] + Reg[rs2]]`

4.11.1 [5] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.11.2 [5] <§4.4> Which existing functional blocks (if any) require modification?

4.11.3 [5] <§4.4> Which new data paths (if any) do we need for this instruction?

4.11.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.12 Examine the difficulty of adding a proposed swap `rs, rt` instruction to MIPS.

Interpretation: `Reg[rt] = Reg[rs]; Reg[rs] = Reg[rt]`

4.12.1 [5] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.12.2 [10] <§4.4> Which existing functional blocks (if any) require modification?

4.12.3 [5] <§4.4> What new data paths do we need (if any) to support this instruction?

4.12.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.12.5 [5] <§4.4> Modify Figure 4.21 to demonstrate an implementation of this new instruction.

4.13 Examine the difficulty of adding a proposed `ss rt, rs, imm` (Store Sum) instruction to MIPS.

Interpretation: `Mem[Reg[rt]=Reg[rs]+immediate`

4.13.1 [10] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.13.2 [10] <§4.4> Which existing functional blocks (ifany) require modification?

4.13.3 [5] <§4.4> What new data paths do we need (if any) to support this instruction?

4.13.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.13.5 [5] <§4.4> Modify Figure 4.21 to demonstrate an implementation of this new instruction.

4.14 [5] <§4.4> For which instructions (if any) is the Imm Gen block on the critical path?

4.15 `lw` is the instruction with the longest latency on the CPU from Section 4.4. If we modified `lw` and `sw` so that there was no offset (i.e., the address to be loaded from/stored to must be calculated and placed in `rs` before calling `lw/sw`), then no instruction would use both the ALU and Data memory. This would allow us to reduce the clock cycle time. However, it would also increase the number of instructions, because many `ld` and `sd` instructions would need to be replaced with `lw/add` or `sw/add` combinations.

4.15.1 [5] <§4.4> What would the new clock cycle time be?

4.15.2 [10] <§4.4> Would a program with the instruction mix presented in Exercise 4.7 run faster or slower on this new CPU? By how much? (For simplicity, assume every `lw` and `sw` instruction is replaced with a sequence of two instructions.)

4.15.3 [5] <§4.4> What is the primary factor that influences whether a program will run faster or slower on the new CPU?

4.15.4 [5] <§4.4> Do you consider the original CPU (as shown in Figure 4.21) a better overall design; or do you consider the new CPU a better overall design? Why?

4.16 In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

Also, assume that instructions executed by the processor are broken down as follows:

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

4.16.1 [5] <§4.6> What is the clock cycle time in a pipelined and non-pipelined processor?

4.16.2 [10] <§4.6> What is the total latency of an `lw` instruction in a pipelined and non-pipelined processor?

4.16.3 [10] <§4.6> If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

4.16.4 [10] <§4.6> Assuming there are no stalls or hazards, what is the utilization of the data memory?

4.16.5 [10] <§4.6> Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit?

4.17 [10] <§4.6> What is the minimum number of cycles needed to completely execute n instructions on a CPU with a k stage pipeline? Justify your formula.

4.18 [5] <§4.6> Assume that `$s0` is initialized to 11 and `$s1` is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.6 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting `NOP` instructions where necessary). What would the final values of registers `$s2` and `$s3` be?

```
addi $s0, $s1, 5
add $s2, $s0, $s1
addi $s3, $s0, 15
```

4.19 [10] <§4.6> Assume that `$s0` is initialized to 11 and `$s1` is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.6 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). What would the final values of register `$s4` be? Assume the register file is written at the beginning of the cycle and read at the end of a cycle. Therefore, an ID stage will return the results of a WB state occurring during the same cycle. See Section 4.8 and Figure 4.51 for details.

```
addi $s0, $s1, 5
add $s2, $s0, $s1
addi $s3, $s0, 15
add $s4, $s0, $s0
```

4.20 [5] <§4.6> Add `NOP` instructions to the code below so that it will run correctly on a pipeline that does not handle data hazards.

```
addi $s0, $s1, 5
add $s2, $s0, $s1
addi $s3, $s0, 15
add $s4, $s2, $s1
```

4.21 Consider a version of the pipeline from Section 4.6 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting `NOP` instructions where necessary). Suppose that (after optimization) a typical `n`-instruction program requires an additional `4*n NOP` instructions to correctly handle data hazards.

4.21.1 [5] <§4.6> Suppose that the cycle time of this pipeline without forwarding is 250 ps. Suppose also that adding forwarding hardware will reduce the number of `NOP`s from `.4*n to .05*n,` but increase the cycle time to 300 ps. What is the speedup of this new pipeline compared to the one without forwarding?

4.21.2 [10] <§4.6> Different programs will require different amounts of `NOP`s. How many `NOP`s (as a percentage of code instructions) can remain in the typical program before that program runs slower on the pipeline with forwarding?

4.21.3 [10] <§4.6> Repeat 4.21.2; however, this time let `x` represent the number of `NOP` instructions relative to `n`. (In 4.21.2, `x` was equal to `.4`.) Your answer will be with respect to `x`.

4.21.4 [10] <§4.6> Can a program with only `.07 5*n NOPs` possibly run faster on the pipeline with forwarding? Explain why or why not.

4.21.5 [10] <§4.6> At minimum, how many `NOP`s (as a percentage of code instructions) must a program have before it can possibly run faster on the pipeline with forwarding?

4.22 [5] <§4.6> Consider the fragment of MIPS assembly below:

```
sd $s5, 12($s3)
Id $s5, 8($s3)
sub $s4, $s2, $s1
beqz $s4, label
add $s2, $s0, $s1
sub $s2, $s6, $s1
```

Suppose we modify the pipeline so that it has only one memory (that handles both instructions and data). In this case, there will be a structural hazard every time a program needs to fetch an instruction during the same cycle in which another instruction accesses data.

4.22.1 [5] <§4.6> Draw a pipeline diagram to show were the code above will stall.

4.22.2 [5] <§4.6> In general, is it possible to reduce the number of stalls/`NOP`s resulting from this structural hazard by reordering code?

4.22.3 [5] <§4.6> Must this structural hazard be handled in hardware? We have seen that data hazards can be eliminated by adding `NOP`s to the code. Can you do the same with this structural hazard? If so, explain how. If not, explain why not.

4.22.4 [5] <§4.6> Approximately how many stalls would you expect this structural hazard to generate in a typical program? (Use the instruction mix from Exercise 4.8.)

4.23 If we change load/store instructions to use a register (without an offset) as the address, these instructions no longer need to use the ALU. (See Exercise 4.15.) As a result, the MEM and EX stages can be overlapped and the pipeline has only four stages.

4.23.1 [10] <§4.6> How will the reduction in pipeline depth affect the cycle time?

4.23.2 [5] <§4.6> How might this change improve the performance of the pipeline?

4.23.3 [5] <§4.6> How might this change degrade the performance of the pipeline?

4.24 [10] <§4.8> Which of the two pipeline diagrams below better describes the operation of the pipeline's hazard detection unit? Why?

Choice 1:

ld x11, 0(x12): IF ID EX ME WB
add x13, x11, x14: IF ID EX..ME WB
or x15, x16, x17: IF ID..EX ME WB

Choice 2:

ld x11, 0(x12): IF ID EX ME WB
add x13, x11, x14: IF ID..EX ME WB
or x15, x16, x17: IF..ID EX ME WB

4.25 Consider the following loop.

```
LOOP: ld $s0, 0($s3)
  ld $s1, 8($s3)
  add $s2, $s0, $s1
  addi $s3, $s3, -16
  bnez $s2, LOOP
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, that the pipeline has full forwarding support, and that branches are resolved in the EX (as opposed to the ID) stage.

4.25.1 [10] <§4.8> Show a pipeline execution diagram for the first two iterations of this loop.

4.25.2 [10] <§4.8> Mark pipeline stages that do not perform useful work. How often while the pipeline is full do we have a cycle in which all five pipeline stages are doing useful work? (Begin with the cycle during which the `addi` is in the IF stage. End with the cycle during which the `bnez` is in the IF stage.)

4.26 This exercise is intended to help you understand the cost/complexity/performance trade-offs of forwarding in a pipelined processor. Problems in this exercise refer to pipelined datapaths from Figure 4.53. These problems assume that, of all the instructions executed in a processor, the following fraction of these instructions has a particular type of RAW data dependence. The type of RAW data dependence is identified by the stage that produces the result (EX or MEM) and the next instruction that consumes the result (1st instruction that follows the one that produces the result, 2nd instruction that follows, or both). We assume that the register write is done in the first half of the clock cycle and that register reads are done in the second half of the cycle, so "EX to 3rd" and "MEM to 3rd" dependences are not counted because they cannot result in data hazards. We also assume that branches are resolved in the EX stage (as opposed to the ID stage), and that the CPI of the processor is 1 if there are no data hazards.

EX to 1st Only	**MEM to 1st Only**	**EX to 2nd Only**	**MEM to 2nd Only**	**EX to 1st and EX to 2nd**
5%	20%	5%	10%	10%

EX to 1st Only	**MEM to 1st Only**	**EX to 2nd Only**	**MEM to 2nd Only**	**EX to 1st and EX to 2nd**
5%	20%	5%	10%	10%

Assume the following latencies for individual pipeline stages. For the

EX stage, latencies are given separately for a processor without forwarding and for a processor with different kinds of forwarding.

IF	ID	EX (no FW)	EX (full FW)	EX (FW from EX/ MEM only)	EX (FW from MEM/ WB only)	MEM	WB
120ps	100ps	110ps	130ps	120ps	120ps	120ps	100ps

IF	ID	EX (no FW)	EX (full FW)	EX (FW from EX/ MEM only)	EX (FW from MEM/ WB only)	MEM	WB
120ps	100ps	110ps	130ps	120ps	120ps	120ps	100ps

4.26.1 [5] <§4.8> For each RAW dependency listed above, give a sequence of at least three assembly statements that exhibits that dependency.

4.26.2 [5] <§4.8> For each RAW dependency above, how many `NOPs` would need to be inserted to allow your code from 4.26.1 to run correctly on a pipeline with no forwarding or hazard detection? Show where the `NOPs` could be inserted.

4.26.3 [10] <§4.8> Analyzing each instruction independently will over-count the number of NOPsneeded to run a program on a pipeline with no forwarding or hazard detection. Write a sequence of three assembly instructions so that, when you consider each instruction in the sequence independently, the sum of the stalls is larger than the number of stalls the sequence actually needs to avoid data hazards.

4.26.4 [5] <§4.8> Assuming no other hazards, what is the CPI for the program described by the table above when run on a pipeline with no forwarding? What percent of cycles are stalls? (For simplicity, assume that all necessary cases are listed above and can be treated independently.)

4.26.5 [5] <§4.8> What is the CPI if we use full forwarding (forward all results that can be forwarded)? What percent of cycles are stalls?

4.26.6 [10] <§4.8> Let us assume that we cannot afford to have three-input multiplexors that are needed for full forwarding. We have to decide if it is better to forward only from the EX/MEM pipeline register (next-cycle forwarding) or only from the

MEM/WB pipeline register (two-cycle forwarding). What is the CPI for each option?

4.26.7 [5] <§4.8> For the given hazard probabilities and pipeline stage latencies, what is the speedup achieved by each type of forwarding (EX/MEM, MEM/WB, for full) as compared to a pipeline that has no forwarding?

4.26.8 [5] <§4.8> What would be the additional speedup (relative to the fastest processor from 4.26.7) be if we added "time-travel" forwarding that eliminates all data hazards? Assume that the yet-to-be-invented time-travel circuitry adds 100 ps to the latency of the full-forwarding EX stage.

4.26.9 [5] <§4.8> The table of hazard types has separate entries for "EX to 1st" and "EX to 1st and EX to 2nd". Why is there no entry for "MEM to 1st and MEM to 2nd"?

4.27 Problems in this exercise refer to the following sequence of instructions, and assume that it is executed on a five-stage pipelined datapath:

```
add $s3, $s1, $s0
lw $s2, 4($s3)
lw $s1, 0($s4)
or $s2, $s3, $s2
sw $s2, 0($s3)
```

4.27.1 [5] <§4.8> If there is no forwarding or hazard detection, insert `NOPs` to ensure correct execution.

4.27.2 [10] <§4.8> Now, change and/or rearrange the code to minimize the number of `NOPs` needed. You can assume register `$t0` can be used to hold temporary values in your modified code.

4.27.3 [10] <§4> If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when the original code executes?

4.27.4 [20] <§4.8> If there is forwarding, for the first seven cycles during the execution of this code, specify which signals are asserted in each cycle by hazard detection and forwarding units in Figure 4.59.

4.27.5 [10] <§4.8> If there is no forwarding, what new input and output signals do we need for the hazard detection unit in Figure 4.59? Using this instruction sequence as an example, explain why each signal is needed.

4.27.6 [20] <§4.8> For the new hazard detection unit from 4.26.5, specify which output signals it asserts in each of the first five cycles during the execution of this code.

4.28 The importance of having a good branch predictor depends on how often conditional branches are executed. Together with branch predictor accuracy, this will determine how much time is spent stalling due to mispredicted branches. In this exercise, assume that the breakdown of dynamic instructions into various instruction categories is as follows:

R-type	**beqz/bnez**	**jal**	**lw**	**sw**
40%	25%	5%	25%	5%

R-type	**beqz/bnez**	**jal**	**lw**	**sw**
40%	25%	5%	25%	5%

Also, assume the following branch predictor accuracies:

Always-Taken	**Always-Not-Taken**	**2-Bit**
45%	55%	85%

4.28.1 [10] <§4.9> Stall cycles due to mispredicted branches increase the CPI. What is the extra CPI due to mispredicted branches with the always-taken predictor? Assume that branch outcomes are determined in the ID stage and applied in the EX stage that there are no data hazards, and that no delay slots are used.

4.28.2 [10] <§4.9> Repeat 4.28.1 for the “always-not-taken” predictor.

4.28.3 [10] <§4.9> Repeat 4.28.1 for the 2-bit predictor.

4.28.4 [10] <§4.9> With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions to some ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

4.28.5 [10] <§4.9> With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaced each branch instruction with two ALU instructions? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

4.28.6 [10] <§4.9> Some branch instructions are much more predictable than others. If we knowthat 80% of all executed branch instructions are easy-to-predict loop-back branches that are always predicted correctly, what is the accuracy of the 2-bit predictor on the remaining 20% of the branch instructions?

4.29 This exercise examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes: `T, NT, T, T, NT`.

4.29.1 [5] <§4.9> What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

4.29.2 [5] <§4.9> What is the accuracy of the 2-bit predictor for the first four branches in this pattern, assuming that the predictor starts off in the bottom left state from Figure 4.61 (predict not taken)?

4.29.3 [10] <§4.9> What is the accuracy of the 2-bit predictor if this pattern is repeated forever?

4.29.4 [30] <§4.9> Design a predictor that would achieve a perfect accuracy if this pattern is repeated forever. You predictor should be a sequential circuit with one output that provides a prediction (1 for taken, 0 for not taken) and no inputs other than the clock and the control signal that indicates that the instruction is a conditional branch.

4.29.5 [10] <§4.9> What is the accuracy of your predictor from 4.29.4 if it is given a repeating pattern that is the exact opposite of this one?

4.29.6 [20] <§4.9> Repeat 4.29.4, but now your predictor should be able to eventually (after a warm-up period during which it can make wrong predictions) start perfectly predicting both this pattern and its opposite. Your predictor should have an input that tells it what the real outcome was. Hint: this input lets your predictor determine which of the two repeating patterns it is given.

4.30 This exercise explores how exception handling affects pipeline design. The first three problems in this exercise refer to the following two instructions:

Instruction 1	**Instruction 2**
beqz $s0, LABEL	ld $s0, 0($s1)

4.30.1 [5] <§4.10> Which exceptions can each of these instructions trigger? For each of these exceptions, specify the pipeline stage in which it is detected.

4.30.2 [10] <§4.10> If there is a separate handler address for each exception, show how the pipeline organization must be changed to be able to handle this exception. You can assume that the addresses of these handlers are known when the processor is designed.

4.30.3 [10] <§4.10> If the second instruction is fetched immediately after the first instruction, describe what happens in the pipeline when the first instruction causes the first exception you listed in Exercise 4.30.1. Show the pipeline execution diagram from the time the first instruction is fetched until the time the first instruction of the exception handler is completed.

4.30.4 [20] <§4.10> In vectored exception handling, the table of exception handler addresses is in data memory at a known (fixed) address. Change the pipeline to implement this exception handling mechanism. Repeat Exercise 4.30.3 using this modified pipeline and vectored exception handling.

4.30.5 [15] <§4.10> We want to emulate vectored exception handling (described in Exercise 4.30.4) on a machine that has only one fixed handler address. Write the code that should be at that fixed address. Hint: this code should identify the exception, get the right address from the exception vector table, and transfer execution to that handler.

4.31 In this exercise we compare the performance of 1-issue and 2-issue processors, taking into account program transformations that can be made to optimize for 2-issue execution. Problems in this exercise refer to the following loop (written in C):

```
for(i=0;i!=j;i+=2)
 b[i]=a[i]-a[i+1];
```

A compiler doing little or no optimization might produce the following MIPS assembly code:

```
li $s0, 0
jal ENT
TOP: sll $t0, $s0, 3
  add $t1, $s2, $t0
  lw $t2, 0($t1)
  lw $t3, 8($t1)
  sub $t4, $t2, $t3
  add $t5, $s3, $t0
  sw $t4, 0($t5)
  addi $s0, $s0, 2
ENT: bne $s0, $s1, TOP
```

The code above uses the following registers:

i	**j**	**a**	**b**	**Temporary values**
$s0	$s1	$s2	$s3	$t0-$t5

i	**j**	**a**	**b**	**Temporary values**
$s0	$s1	$s2	$s3	$t0-$t5

Assume the two-issue, statically scheduled processor for this exercise has the following properties:

1 One instruction must be a memory operation; the other must be an arithmetic/logic instruction or a branch.
2 The processor has all possible forwarding paths between stages (including paths to the ID stage for branch resolution).
3 The processor has perfect branch prediction.
4 Two instruction may not issue together in a packet if one depends on the other. (See page 336.)
5 If a stall is necessary, both instructions in the issue packet must stall. (See page 336.)

As you complete these exercises, notice how much effort goes into generating code that will produce a near-optimal speedup.

4.31.1 [30] <§4.11> Draw a pipeline diagram showing how MIPS code given above executes on the two-issue processor. Assume that the loop exits after two iterations.

4.31.2 [10] <§4.11> What is the speedup of going from a one-issue to a two- issue processor? (Assume the loop runs thousands of iterations.)

4.31.3 [10] <§4.11> Rearrange/rewrite the MIPS code given above to achieve better performance on the one-issue processor. Hint: Use the instruction “`beqz $s1,DONE`” to skip the loop entirely if `j = 0`.

4.31.4 [20] <§4.11> Rearrange/rewrite the MIPS code given above to achieve better performance on the two-issue processor. (Do not unroll the loop, however.)

4.31.5 [30] <§4.11> Repeat Exercise 4.31.1, but this time use your optimized code from Exercise 4.31.4.

4.31.6 [10] <§4.11> What is the speedup of going from a one-issue processor to a two-issue processor when running the optimized code from Exercises 4.31.3 and 4.31.4.

4.31.7 [10] <§4.11> Unroll the MIPS code from Exercise 4.31.3 so that each iteration of the unrolled loop handles two iterations of the original loop. Then, rearrange/rewrite your unrolled code to achieve better performance on the one- issue processor. You may assume that `j` is a multiple of 4.

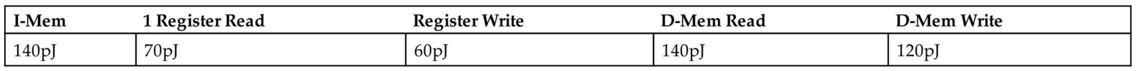
4.31.8 [20] <§4.11> Unroll the MIPS code from Exercise 4.31.4 so that each iteration of the unrolled loop handles two iterations of the original loop. Then, rearrange/rewrite your unrolled code to achieve better performance on the two- issue processor. You may assume that j is a multiple of 4. (Hint: Re-organize the loop so that some calculations appear both outside the loop and at the end of the loop. You may assume that the values in temporary registers are not needed after the loop.)

4.31.9 [10] <§4.11> What is the speedup of going from a one-issue processor to a two-issue processor when running the unrolled, optimized code from Exercises 4.31.7 and 4.31.8?

4.31.10 [30] <§4.11> Repeat Exercises 4.31.8 and 4.31.9, but this time assume the two-issue processor can run two arithmetic/logic instructions together. (In other words, the first instruction in a packet can be any type of instruction, but the second must be an arithmetic or logic instruction. Two memory operations cannot be scheduled at the same time.)

4.32 This exercise explores energy efficiency and its relationship with performance. Problems in this exercise assume the following energy consumption for activity in Instruction memory, Registers, and Data memory. You can assume that the other components of the datapath consume a negligible amount of energy. ("Register Read" and "Register Write" refer to the register file only.)

I-Mem	**1 Register Read**	**Register Write**	**D-Mem Read**	**D-Mem Write**
140pJ	70pJ	60pJ	140pJ	120pJ



I-Mem	1 Register Read	Register Write	D-Mem Read	D-Mem Write
140pJ	70pJ	60pJ	140pJ	120pJ

Assume that components in the datapath have the following latencies. You can assume that the other components of the datapath have negligible latencies.

I-Mem	Control	Register Read or Write	ALU	D-Mem Read or Write
200ps	150ps	90ps	90ps	250ps

I-Mem	Control	Register Read or Write	ALU	D-Mem Read or Write
200ps	150ps	90ps	90ps	250ps

4.32.1 [5] <§§4.3, 4.7, 4.15> How much energy is spent to execute an add instruction in a single-cycle design and in the five-stage pipelined design?

4.32.2 [10] <§§4.7, 4.15> What is the worst-case MIPS instruction in terms of energy consumption? What is the energy spent to execute it?

4.32.3 [10] <§§4.7, 4.15> If energy reduction is paramount, how would you change the pipelined design? What is the percentage reduction in the energy spent by an `ld` instruction after this change?

4.32.4 [10] <§§4.7, 4.15> What other instructions can potentially benefit from the change discussed in Exercise 4.32.3?

4.32.5 [10] <§§4.7, 4.15> How do your changes from Exercise 4.32.3 affect the performance of a pipelined CPU?

4.32.6 [10] <§§4.7, 4.15> We can eliminate the `MemRead` control signal and have the data memory be readm every cycle, i.e., we can permanently have `MemRead=1`. Explain why the processor still functions correctly after this change. If 25% of instructions are loads, what is the effect of this change on clock frequency and energy consumption?

4.33 When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits. A very common defect is for one wire to affect the signal in another. This is called a “cross-talk fault”. A special class of cross-talk faults is when a signal is connected to a wire that has a constant logical value (e.g., a power supply wire). These faults, where the affected signal always has a logical value of either 0 or 1 are called “stuck-at-0” or “stuck- at-1” faults. The following problems refer to bit 0 of the Write Register input on the register file in Figure 4.21.

4.33.1 [10] <§§4.3, 4.4> Let us assume that processor testing is done by (1) filling the PC, registers, and data and instruction memories with some values (you can choose which values), (2) letting a single instruction execute, then (3) reading the PC, memories, and registers. These values are then examined to determine if a particular fault is present. Can you design a test (values for PC, memories, and registers) that would determine if there is a stuck-at-0 fault on this signal?

4.33.2 [10] <§§4.3, 4.4> Repeat Exercise 4.33.1 for a stuck-at-1 fault. Can you use a single test for both stuck-at-0 and stuck-at-1? If yes, explain how; if no, explain why not.

4.33.3 [10] <§§4.3, 4.4> If we know that the processor has a stuck-at-1 fault on this signal, is the processor still usable? To be usable, we must be able to convert any program that executes on a normal MIPS processor into a program that works on this processor. You can assume that there is enough free instruction memory and data memory to let you make the program longer and store additional data.

4.33.4 [10] <§§4.3, 4.4> Repeat Exercise 4.33.1; but now the fault to test for is whether the `MemRead` control signal becomes 0 if the `branch` control signal is 0, no fault otherwise.

4.33.5 [10] <§§4.3, 4.4> Repeat Exercise 4.33.1; but now the fault to test for is whether the `MemRead` control signal becomes 1 if `RegRd` control signal is 1, no fault otherwise. Hint: This problem requires knowledge of operating systems. Consider what causes segmentation faults.

Answers to Check Yourself

§4.1, page 260: 3 of 5: Control, Datapath, Memory. Input and Output are missing.

§4.2, page 263: false. Edge-triggered state elements make simultaneous reading and writing both possible and unambiguous.

§4.3, page 269: I. a. II. c.

§4.4, page 284: Yes, Branch and ALUOp0 are identical. In addition, MemtoReg and RegDst are inverses of one another. You don’t need an inverter; simply use the other signal and flip the order of the inputs to the multiplexor!

§4.5, page 4.5-22: 1. False. 2. Maybe; If the signal PCSource[0] is always set to zero when it is a don't care (which is most states), then it is identical to the PCWriteCond.

§4.6, page 297: 1. Stall on the `lw` result. 2. Bypass the first `add` result written into `$t1`. 3. No stall or bypass required.

§4.7, page 310: Statements 2 and 4 are correct; the rest are incorrect.

§4.9, page 336: 1. Predict not taken. 2. Predict taken. 3. Dynamic prediction.

§4.10, page 344: The first instruction, since it is logically executed before the others.

§4.11, page 357: 1. Both. 2. Both. 3. Software. 4. Hardware. 5. Hardware. 6. Hardware. 7. Both. 8. Hardware. 9. Both.

§4.13, page 368: First two are false and the last two are true.