

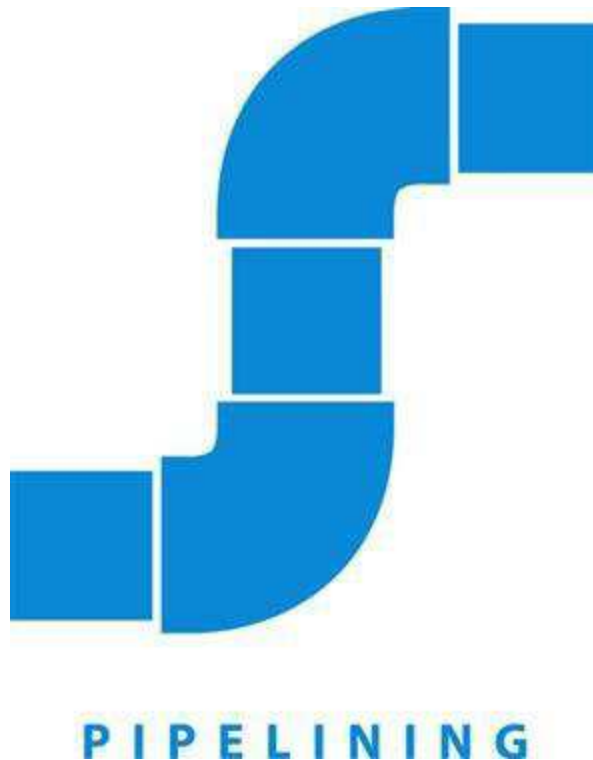
The Five Classic Components of a Computer

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 RISC-V 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 RISC-V. The bulk of the chapter covers a more realistic **pipelined** RISC-V 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 4.5](#) present the basic concepts of pipelining. Current trends are covered in [Section 4.10](#), and [Section 4.11](#) describes the recent Intel Core i7 and ARM Cortex-A53 architectures. [Section 4.12](#) shows how to use instruction-level parallelism to more than double the performance of the matrix multiply from [Section 3.9](#). 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 [4.6](#) will be useful. Those interested in learning how to build a processor should also cover [Sections 4.2](#), [4.7–4.9](#). For readers with an interest in modern

hardware design,  [Section 4.13](#) 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 RISC-V Implementation

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

- The memory-reference instructions *load doubleword* (`ld`) and *store doubleword* (`sd`)
- The arithmetic-logical instructions `add`, `sub`, `and`, and `or`
- The conditional branch instruction *branch if equal* (`beq`)

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 RISC-V 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 RISC-V 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 `ld` 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 RISC-V instruction set simplify the implementation by making the execution of many of the instruction classes similar.

For example, all instruction classes 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 conditional branches for the equality test. 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 conditional branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by four to get the address of the subsequent instruction.

Figure 4.1 shows the high-level view of a RISC-V 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.

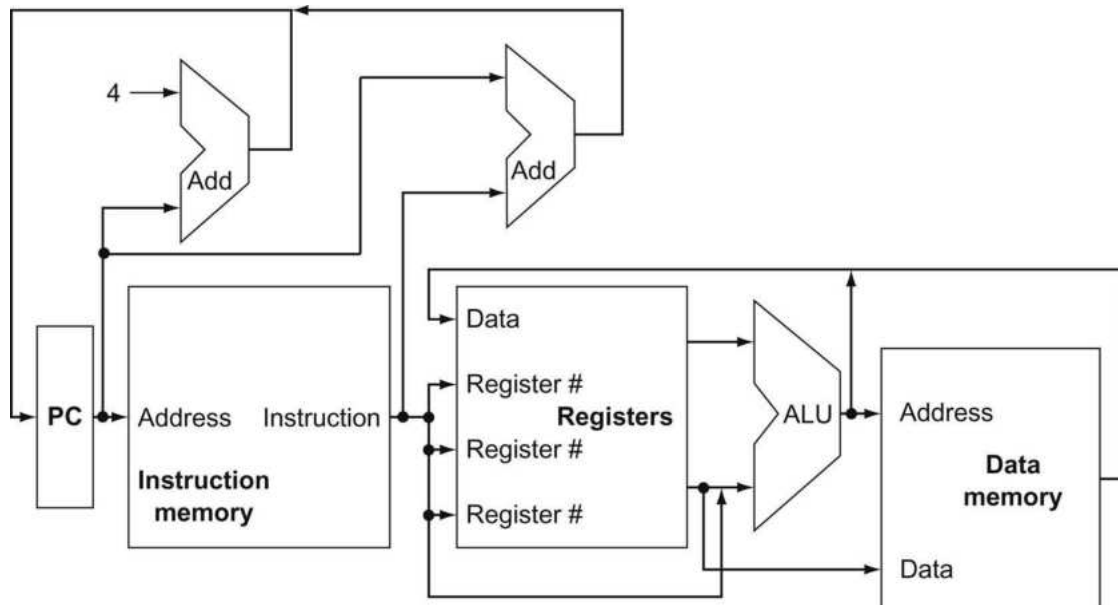


FIGURE 4.1 An abstract view of the implementation of the RISC-V subset showing the major functional units and the major connections between them.

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 an equality check (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the adder (where the PC and branch offset are summed) or from an adder that increments the current PC by four. 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 A](#) 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 an arithmetic-logical instruction. And, of course, the ALU must perform one of several operations. ([Appendix A](#) describes the detailed design of the ALU.) Like the multiplexors, control lines that are set based on various fields in the instruction 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 top 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 RISC-V instruction set mean that a simple decoding process can be used to determine how to set the control lines.

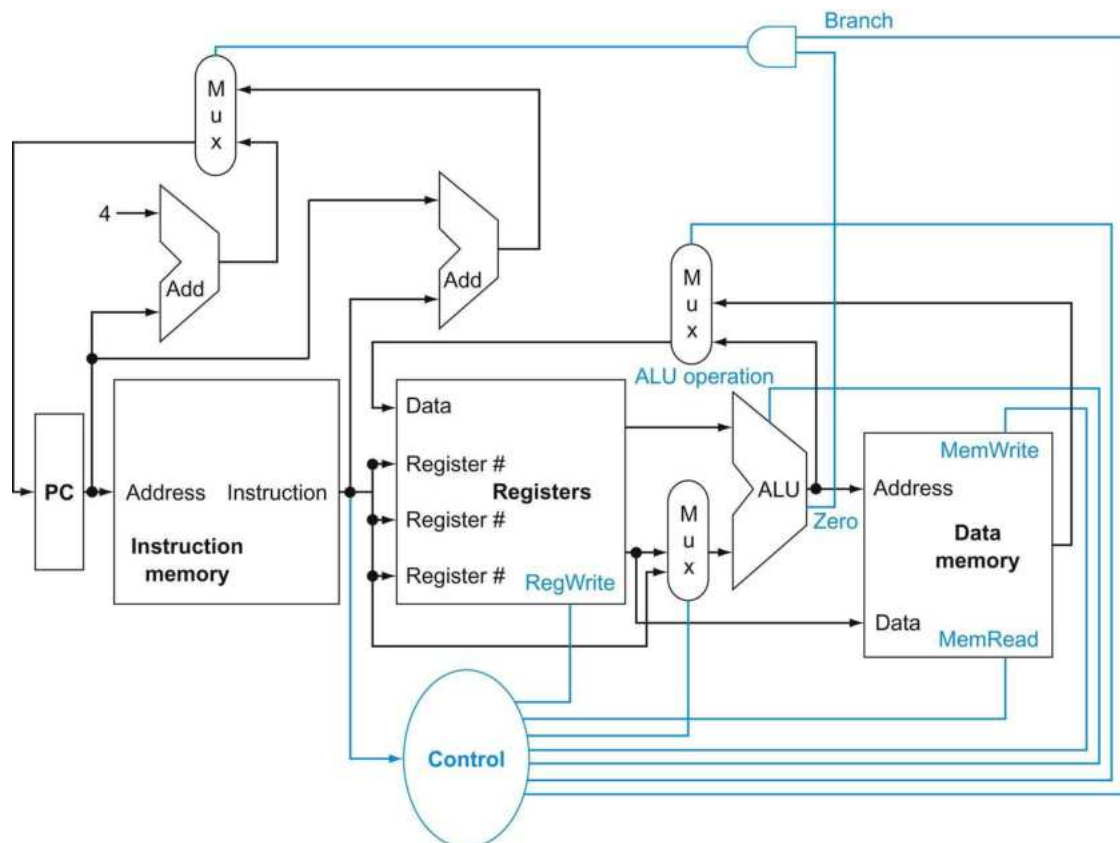


FIGURE 4.2 The basic implementation of the RISC-V subset, including the necessary multiplexors and control lines.

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 bottom-most 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 pipelined implementation with all its complexities, including exceptions.

Check Yourself

How many of the five classic components of a computer—shown on page 235—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 A](#) before continuing.

The datapath elements in the RISC-V implementation consist of two different types of logic elements: elements that operate on data values and elements that contain state. The elements that operate on data values are all **combinational**, which means that their outputs depend only on the current inputs. Given the same input, a combinational element always produces the same output. The ALU shown in [Figure 4.1](#) and discussed in [Appendix A](#) 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 A](#)), which has exactly these two inputs (a value and a clock) and one output. In addition to flip-flops, our RISC-V 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. [Appendix A](#) discusses the operation of both the combinational and sequential elements and their construction in more detail.

Clocking Methodology

clocking methodology

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

A **clocking methodology** defines when signals can be read and when they can be written. It is important to specify the timing of reads and writes, because if a signal is written at the same time that 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.

edge-triggered clocking

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

For simplicity, we will assume an **edge-triggered clocking** methodology. An edge-triggered clocking methodology means that any values stored in a sequential logic element are updated only on a clock edge, 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.

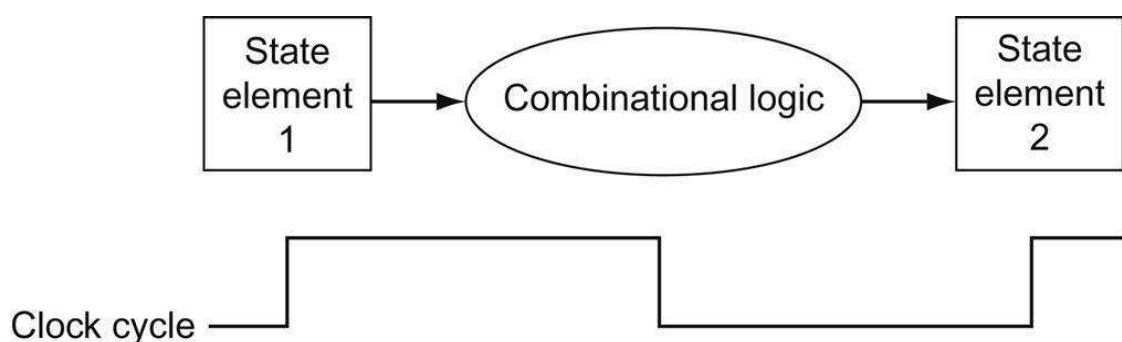


FIGURE 4.3 Combinational logic, state elements, and the clock are closely related.

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 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.

control signal

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

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

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 A](#), we briefly discuss additional timing constraints (such as setup and hold times) as well as other timing methodologies.

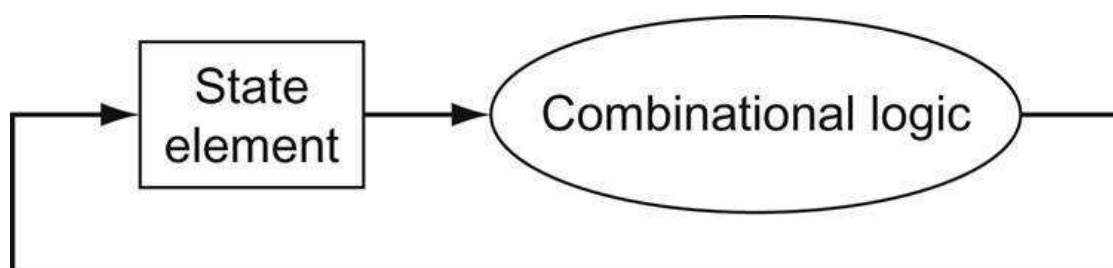


FIGURE 4.4 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values.

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 64-bit RISC-V architecture, nearly all of these state and logic elements will have inputs and outputs that are 64 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 64 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 64-bit bus by combining two 32-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 contrary to a signal that carries data; this distinction will become clearer as we proceed through this chapter.

Check Yourself

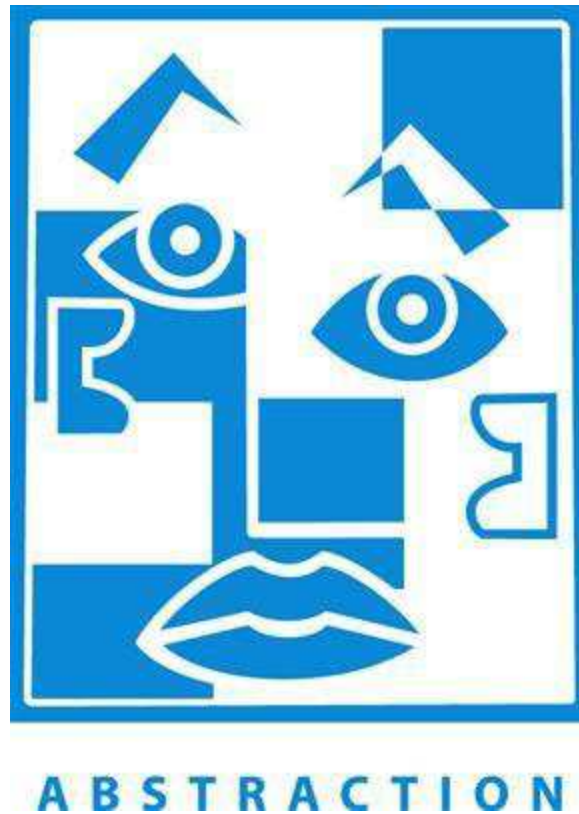
True or false: Because the register file is both read and written on the same clock cycle, any RISC-V datapath using edge-triggered writes must have more than one copy of the register file.

Elaboration

There is also a 32-bit version of the RISC-V architecture, and, naturally enough, most paths in its implementation would be 32 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 RISC-V 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 RISC-V implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

program counter (PC)

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

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 A 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 4.5c](#), to indicate that it has been permanently made an adder and cannot perform the other ALU functions.

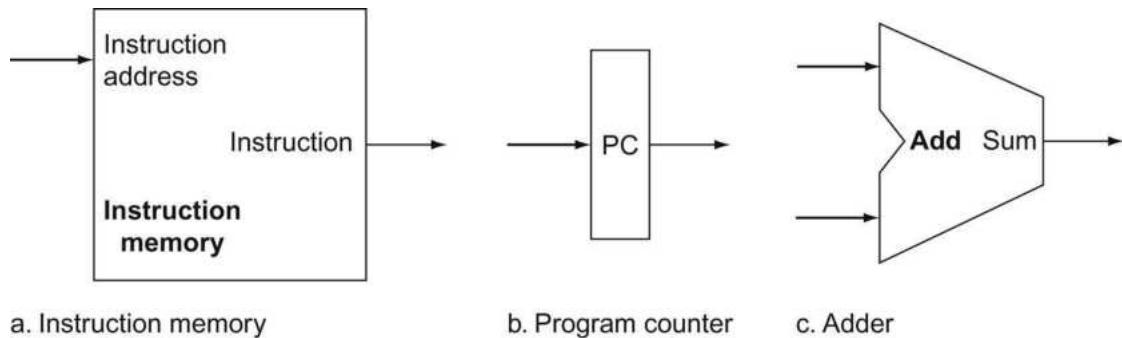


FIGURE 4.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.

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 64-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 64-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 a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

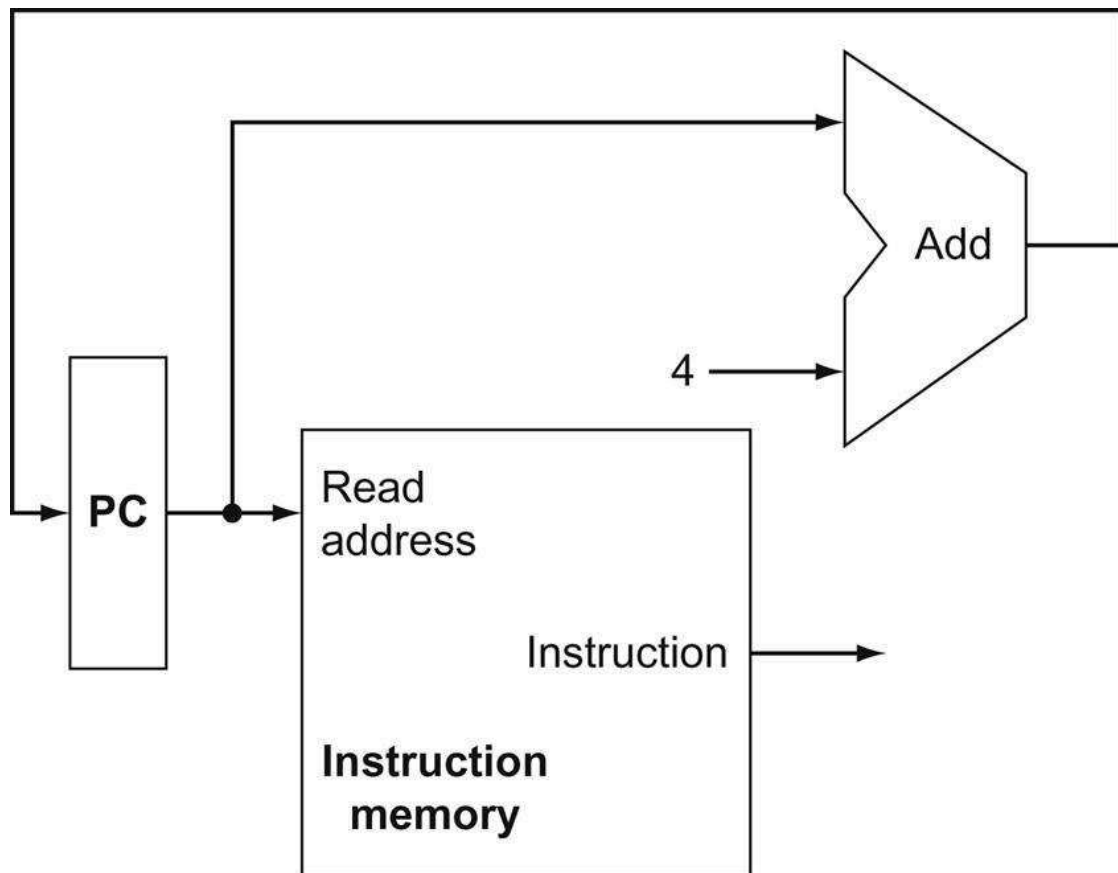


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter.

The fetched instruction is used by other parts of the datapath.

Now let's consider the R-format instructions (see [Figure 2.19](#) on page 120). 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`, and `or`, which were introduced in [Chapter 2](#). Recall that a typical instance of such an instruction is `add x1, x2, x3`, which reads `x2` and `x3` and writes the sum into `x1`.

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.

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.

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 three inputs (two 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 64 bits wide.

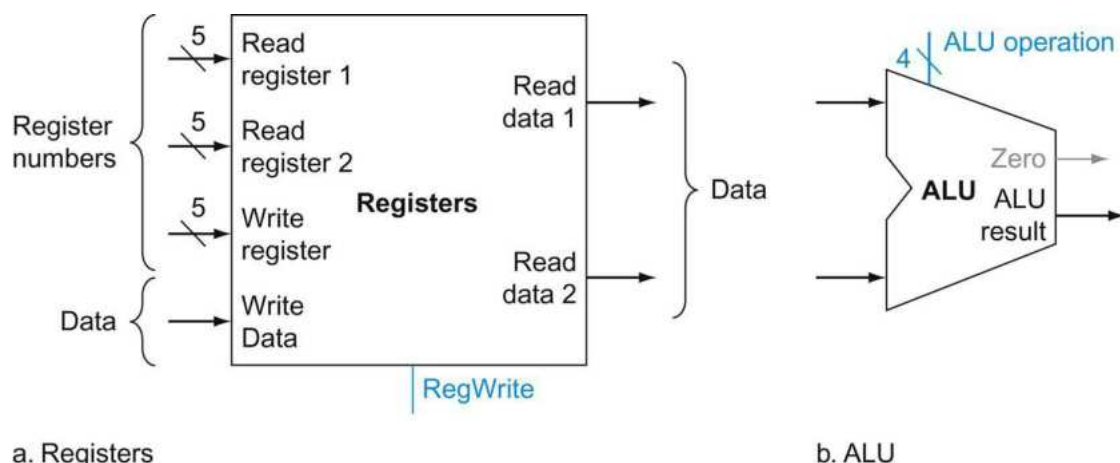


FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU.

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 A.8](#) of

[Appendix A](#). 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 64 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 A](#). We will use the Zero detection output of the ALU shortly to implement conditional branches.

[Figure 4.7b](#) shows the ALU, which takes two 64-bit inputs and produces a 64-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 A](#); we will review the ALU control shortly when we need to know how to set it.

Next, consider the RISC-V load register and store register instructions, which have the general form `ld x1, offset(x2)` or `sd x1, offset(x2)`. These instructions compute a memory address by adding the base register, which is `x2`, to the 12-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 `x1`. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is `x1`. Thus, we will need both the register file and the ALU from [Figure 4.7](#).

sign-extend

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

destination data item.

branch target address

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

In addition, we will need a unit to **sign-extend** the 12-bit offset field in the instruction to a 64-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.

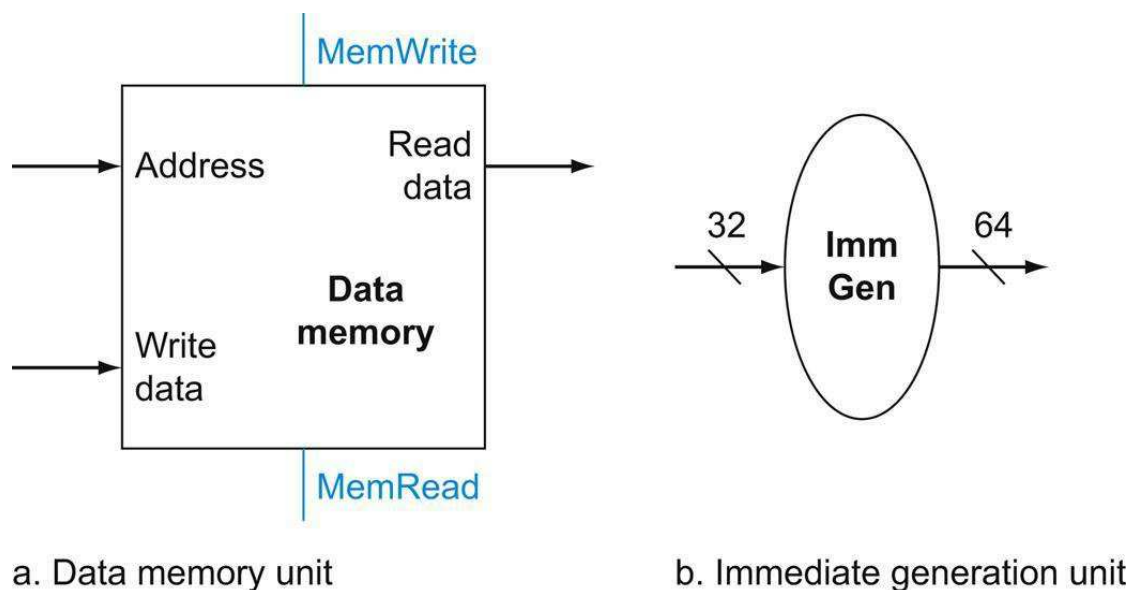


FIGURE 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of [Figure 4.7](#), are the data memory unit and the immediate generation unit.

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 immediate generation unit

(ImmGen) has a 32-bit instruction as input that selects a 12-bit field for load, store, and branch if equal that is sign-extended into a 64-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 A.8 of Appendix A](#) for further discussion of how real memory chips work.

The `beq` instruction has three operands, two registers that are compared for equality, and a 12-bit offset used to compute the **branch target address** relative to the branch instruction address. Its form is `beq x1, x2, 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 branch instruction.
- The architecture also states that the offset field is shifted left 1 bit so that it is a half word offset; this shift increases the effective range of the offset field by a factor of 2.

To deal with the latter complication, we will need to shift the offset field by 1.

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., two operands are equal), the branch target address becomes the new PC, and we say that the **branch is taken**. If the operand is not zero, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the **branch is not taken**.

branch taken

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

branches are taken branches.

branch not taken 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 test 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 an immediate generation 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 two register operands (although we will not need to write into the register file). In addition, the equality comparison can be done using the ALU we designed in [Appendix A](#). Since that ALU provides an output signal that indicates whether the result was 0, we can send both register operands to the ALU with the control set to subtract two values. If the Zero signal out of the ALU unit is asserted, we know that the register values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equality test of conditional branches. Later, we will show exactly how to connect the control signals of the ALU for use in the datapath.

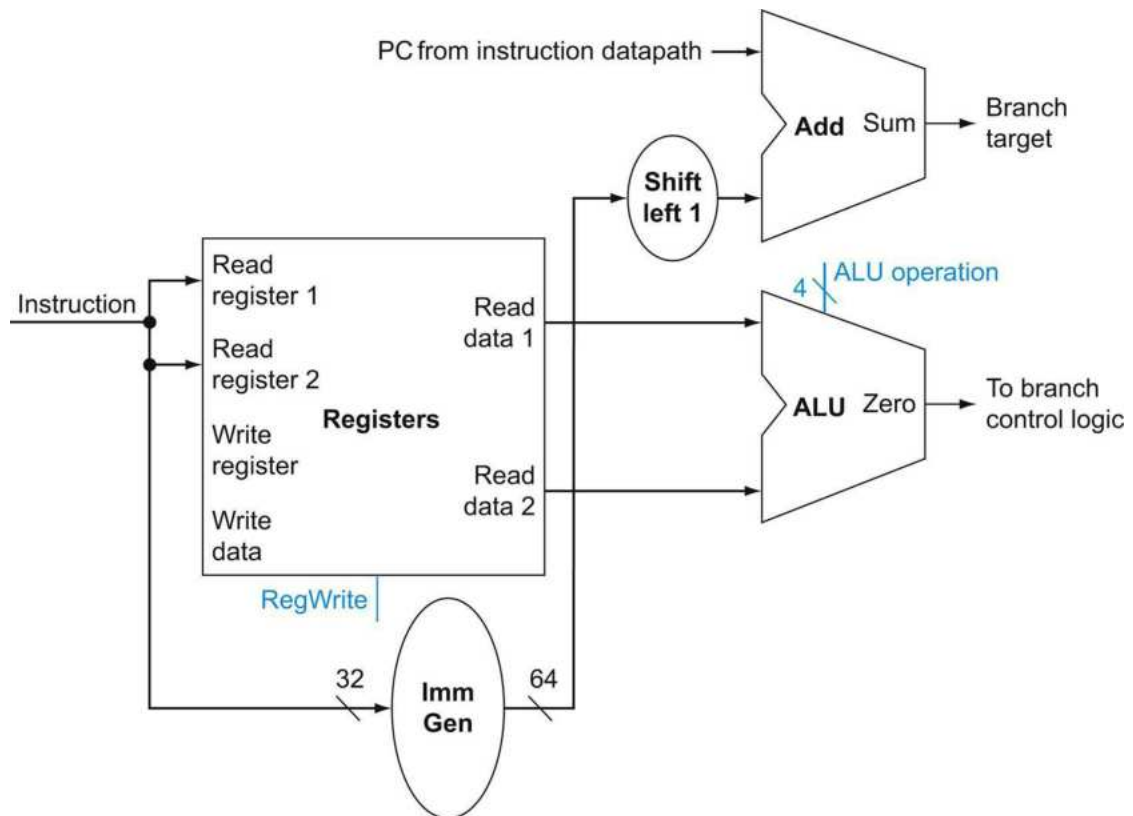


FIGURE 4.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the PC and the sign-extended 12 bits of the instruction (the branch displacement), shifted left 1 bit.

The unit labeled *Shift left 1* is simply a routing of the signals between input and output that adds 0_{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 12 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 branch instruction operates by adding the PC with the 12 bits of the instruction shifted left by 1 bit. Simply concatenating 0 to the branch offset accomplishes this shift, as described in [Chapter 2](#).

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. This design means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated. We therefore need a memory for instructions separate from one for data.

Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows.

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

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 12-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.

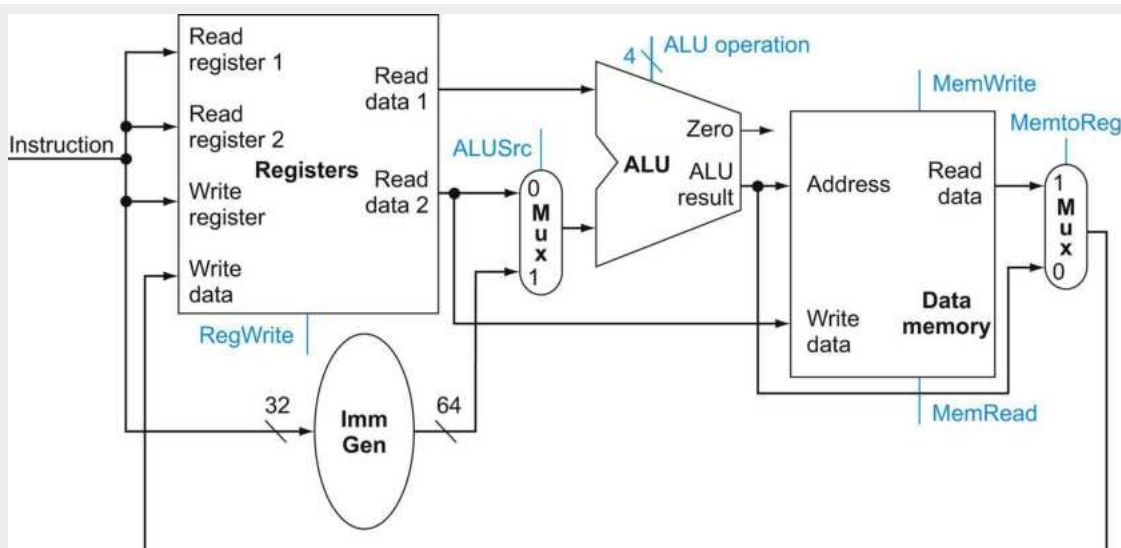


FIGURE 4.10 The datapath for the memory instructions and the R-type instructions.

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 RISC-V 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 to compare two register operands for equality, 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.

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 RISC-V, and hence different memories are needed;
 - b. having separate memories is less expensive;
 - c. the processor operates in one cycle and cannot use a (single-ported) memory for two different accesses within that cycle.

data transfer instructions and 1 for conditional branches, and RISC-V opcode bit 5 happens to be 0 for load instructions and 1 for store instructions. Thus, bits 5 and 6 can control a 3:1 multiplexor inside the immediate generation logic that selects the appropriate 12-bit field for load, store, and conditional branch instructions.

4.4 A Simple Implementation Scheme

In this section, we look at what might be thought of as a simple implementation of our RISC-V subset. We build this simple implementation using the datapath of the last section and adding a simple control function. This simple implementation covers *load doubleword* (`ld`), *store doubleword* (`sd`), *branch if equal* (`beq`), and the arithmetic-logical instructions `add`, `sub`, `and`, and `or`.

The ALU Control

The RISC-V ALU in [Appendix A](#) defines the four following combinations of four control inputs:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Depending on the instruction class, the ALU will need to perform one of these four functions. For load and store 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 four actions (`AND`, `OR`, `add`, or `subtract`), depending on the value of the 7-bit `funct7` field (bits 31:25) and 3-bit `funct3` field (bits 14:12) in the instruction (see [Chapter 2](#)). For the conditional branch if equal instruction, the ALU subtracts two operands and tests to see if the result is 0.

We can generate the 4-bit ALU control input using a small control unit that has as inputs the `funct7` and `funct3` fields 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` and test if zero (01) for `beq`, or be determined by the operation encoded in the `funct7` and `funct3` fields (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, funct7, and funct3 fields. Later in this chapter, we will see how the ALUOp bits are generated from the main control unit.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

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

FIGURE 4.12 How the ALU control bits are set depends on the ALUOp control bits and the different opcodes for the R-type instruction.

The instruction, 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 funct7 or funct3 fields; in this case, we say that we “don’t care” about the value of the opcode, and the bits are shown as Xs. When the ALUOp value is 10, then the funct7 and funct3 fields are used to set the ALU control input. See [Appendix A](#).

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 reduce the latency of the control unit. Such optimizations are

important, since the latency of the control unit is often a critical factor in determining the clock cycle time.

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

As a step in designing this logic, it is useful to create a *truth table* for the interesting combinations of funct fields and the ALUOp signals, as we've done in [Figure 4.13](#); this **truth table** shows how the 4-bit ALU control is set depending on these input fields. Since the full truth table is very large, 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 C.2](#)

of  [Appendix C](#).)

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		Funct7 field							Funct3 field			Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation).

The inputs are the ALUOp and funct fields. 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. While we show all 10 bits of funct fields, note that the only bits with different values for the four R-format instructions are bits 30, 14, 13, and 12. Thus, we only need these four funct field bits as input for ALU control instead of all 10.

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 funct fields. In this case, then, the funct 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 A](#) for more information.

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

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.

Designing the Main Control Unit

Now that we have described how to design an ALU that uses the opcode 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 four instruction classes: arithmetic, load, store, and conditional branch instructions. Figure 4.14 shows these formats.

Name (Bit position)	Fields					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

FIGURE 4.14 The four instruction classes (arithmetic, load, store, and conditional branch) use four different instruction formats.

(a) Instruction format for R-type arithmetic instructions (opcode = 51_{ten}), which have three register operands: rs1, rs2, and rd. Fields rs1 and rd are sources, and rd is the destination. The ALU function is in the funct3 and funct7 fields and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are `add`, `sub`, `and`, and `or`. (b) Instruction format for I-type load instructions (opcode = 3_{ten}). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. Field rd is the destination register for the loaded value. (c) Instruction format for S-type store instructions (opcode = 35_{ten}). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. (The immediate field is split into a 7-bit piece and a 5-bit piece.) Field rs2 is the source register whose value should be stored into memory. (d) Instruction format for SB-type conditional branch instructions (opcode = 99_{ten}). The registers rs1 and rs2 compared. The 12-bit immediate address field is sign-extended, shifted left 1 bit, and added to the PC to compute the branch target address.

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

opcode

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

- The **opcode** field, which as we saw in [Chapter 2](#), is always in bits 6:0. Depending on the opcode, the funct3 field (bits 14:12) and funct7 field (bits 31:25) serve as an extended opcode field.
- The first register operand is always in bit positions 19:15 (rs1) for R-type instructions and branch instructions. This field also specifies the base register for load and store instructions.
- The second register operand is always in bit positions 24:20 (rs2) for R-type instructions and branch instructions. This field also specifies the register operand that gets copied to memory for store instructions.
- Another operand can also be a 12-bit offset for branch or load-store instructions.
- The destination register is always in bit positions 11:7 (rd) for R-type instructions and load instructions.

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 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.

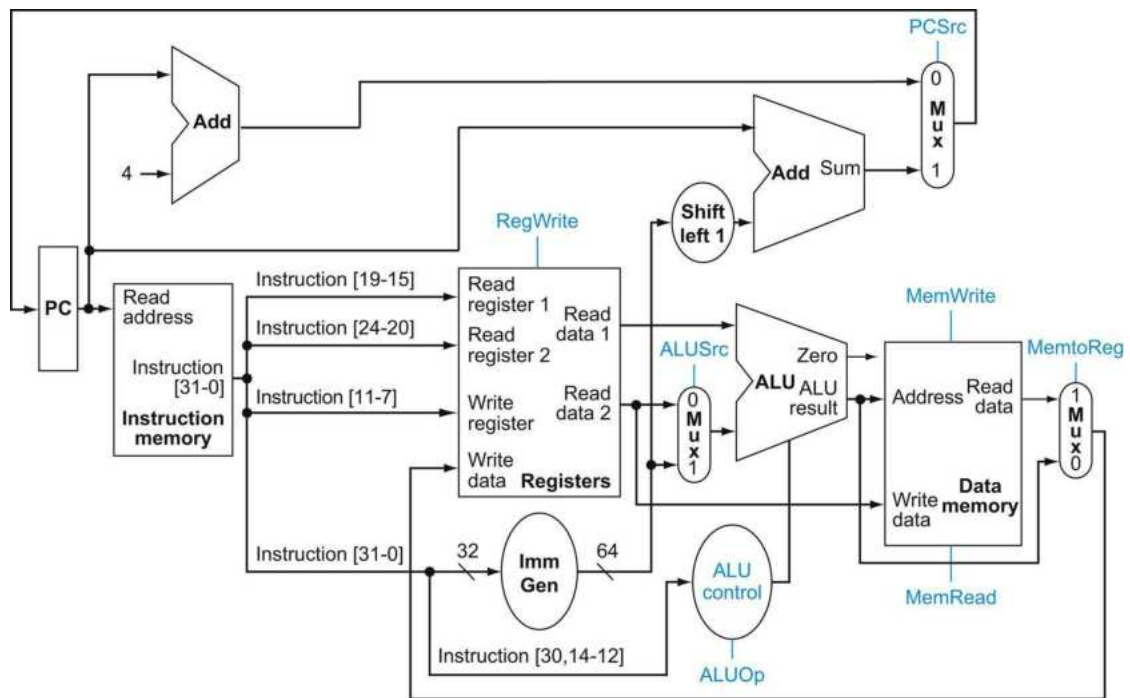


FIGURE 4.15 The datapath of [Figure 4.11](#) with all necessary multiplexors and all control lines identified.

The control lines are shown in color. The ALU control block has also been added, which depends on the funct3 field and part of the funct7 field. 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 six 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 six 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 six control lines.

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of $PC + 4$.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 4.16 The effect of each of the six 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 A](#) 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 and funct fields of the instruction. The PCSrc control line is the exception. That control line should be asserted if the instruction is branch if equal (a decision that the control unit can make) *and* the Zero output of the ALU, which is used for the equality test, 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 eight control signals (six from [Figure 4.16](#) and two for ALUOp) can now be set based on the input signals to the control unit, which are the opcode bits 6:0. [Figure 4.17](#) shows the datapath with the control unit and the control signals.

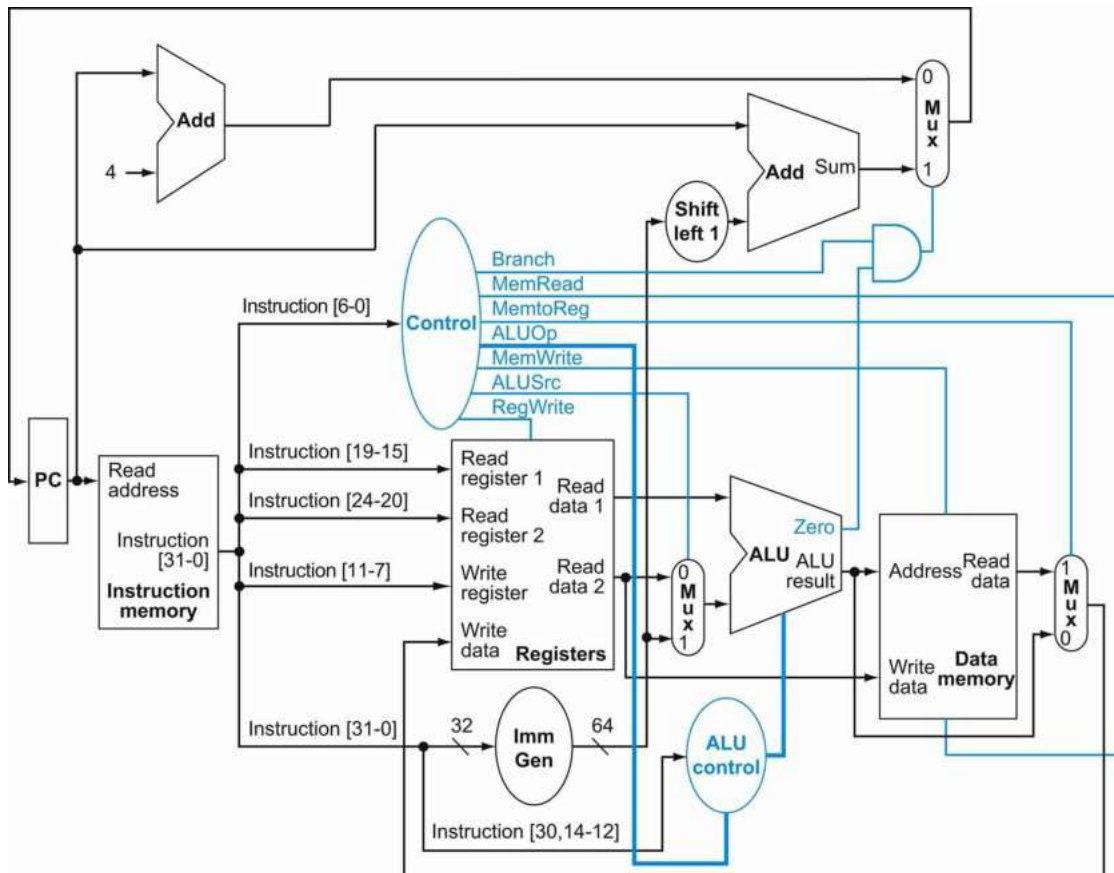


FIGURE 4.17 The simple datapath with the control unit.

The input to the control unit is the 7-bit opcode field from the instruction. The outputs of the control unit consist of two 1-bit signals that are used to control multiplexors (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	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	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`, and `or`). For all these instructions, the source register fields are `rs1` and `rs2`, and the destination register field is `rd`; this defines how the signals `ALUSrc` is set. Furthermore, an R-type instruction writes a register (`RegWrite = 1`), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with `PC + 4`; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high.

The `ALUOp` field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the `funct` fields. The second and third rows of this table give the control signal settings for `ld` and `sd`. These

`ALUSrc` and `ALUOp` fields are set to perform the address calculation. The `MemRead` and `MemWrite` are set to perform the memory access. Finally, `RegWrite` is set for a load to cause the result to be stored in the `rd` register. The `ALUOp` field for branch is set for 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. 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 x1, x2, x3`. 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, `x2` and `x3`, 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 portions of the opcode to generate the ALU function.
4. The result from the ALU is written into the destination register (`x1`) in the register file.

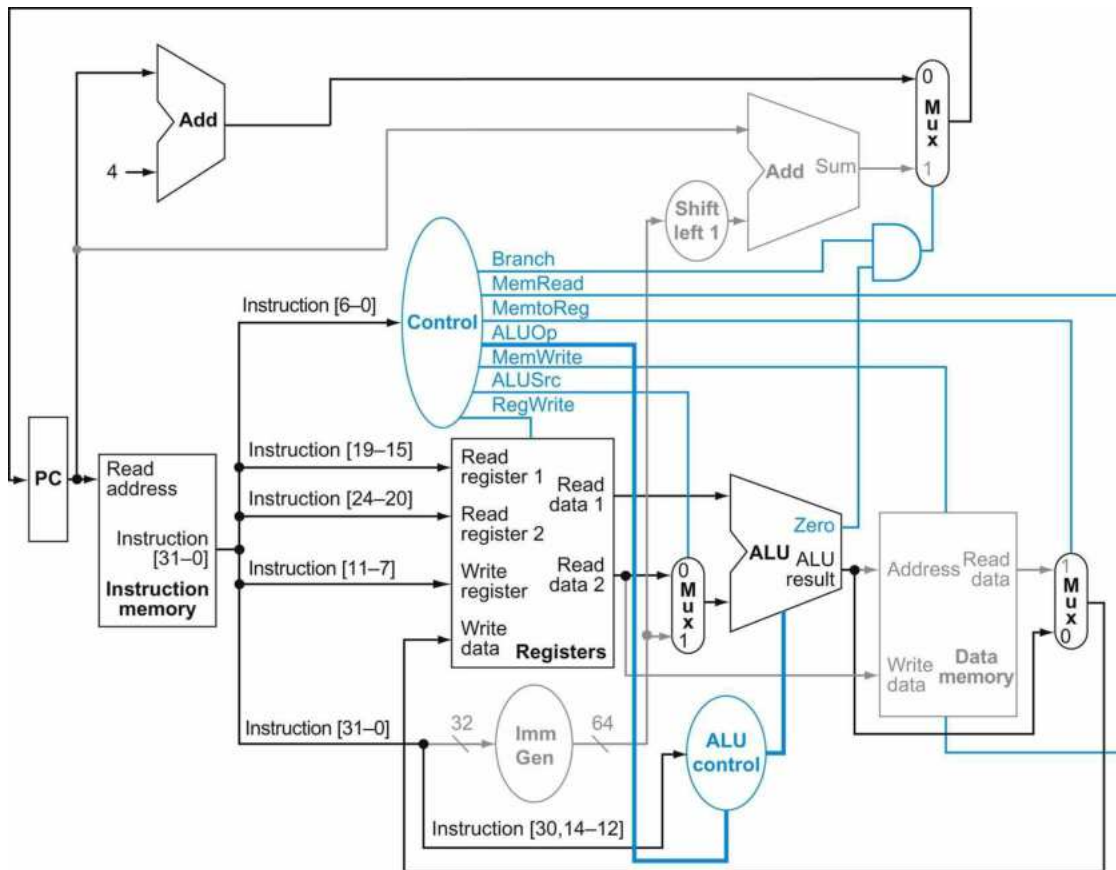


FIGURE 4.19 The datapath in operation for an R-type instruction, such as `add x1, x2, x3`.

The control lines, datapath units, and connections that are active are highlighted.

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

```
ld x1, offset(x2)
```

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 (`x2`) 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 12 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

(x1).

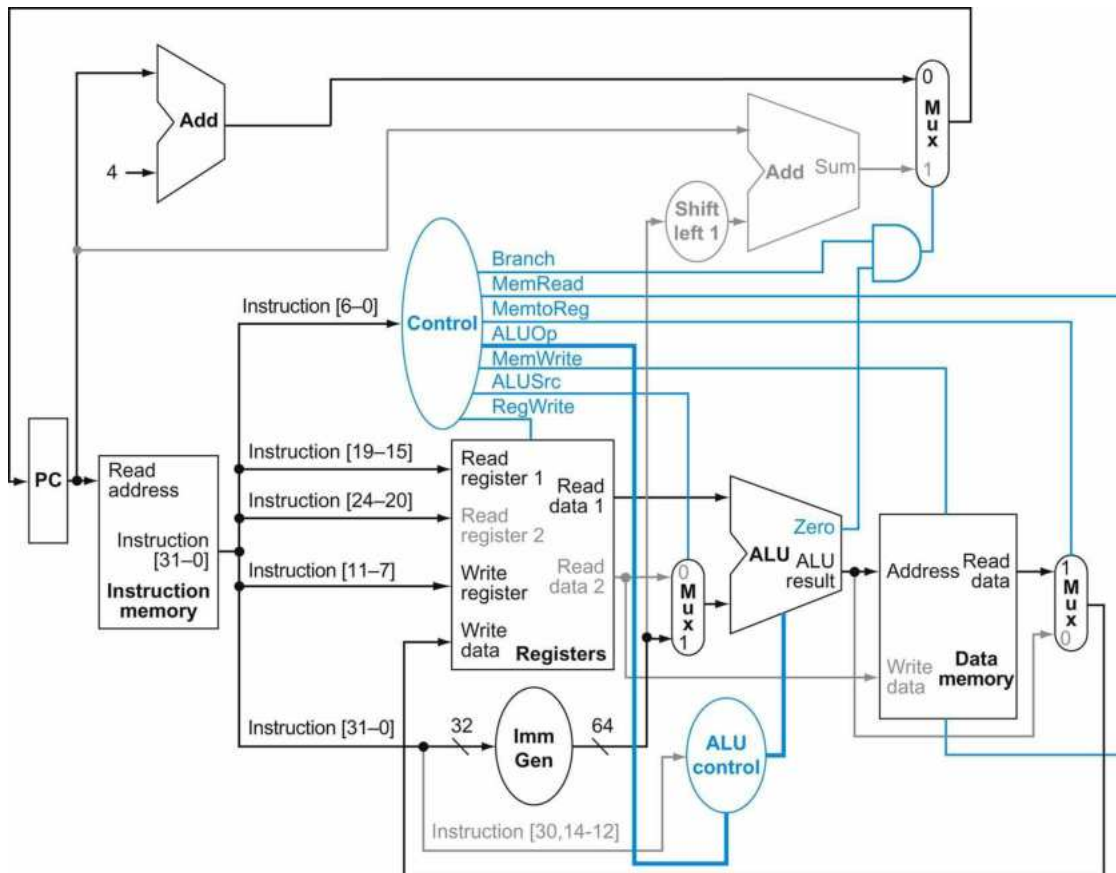


FIGURE 4.20 The datapath in operation for a load instruction.

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-if-equal instruction, such as `beq x1, x2, 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, `x1` and `x2`, are read from the register file.
3. The ALU subtracts one data value from the other data value, both

read from the register file. The value of PC is added to the sign-extended, 12 bits of the instruction (`offset`) left shifted by one; the result is the branch target address.

4. The Zero status information from the ALU is used to decide which adder result to store in the PC.

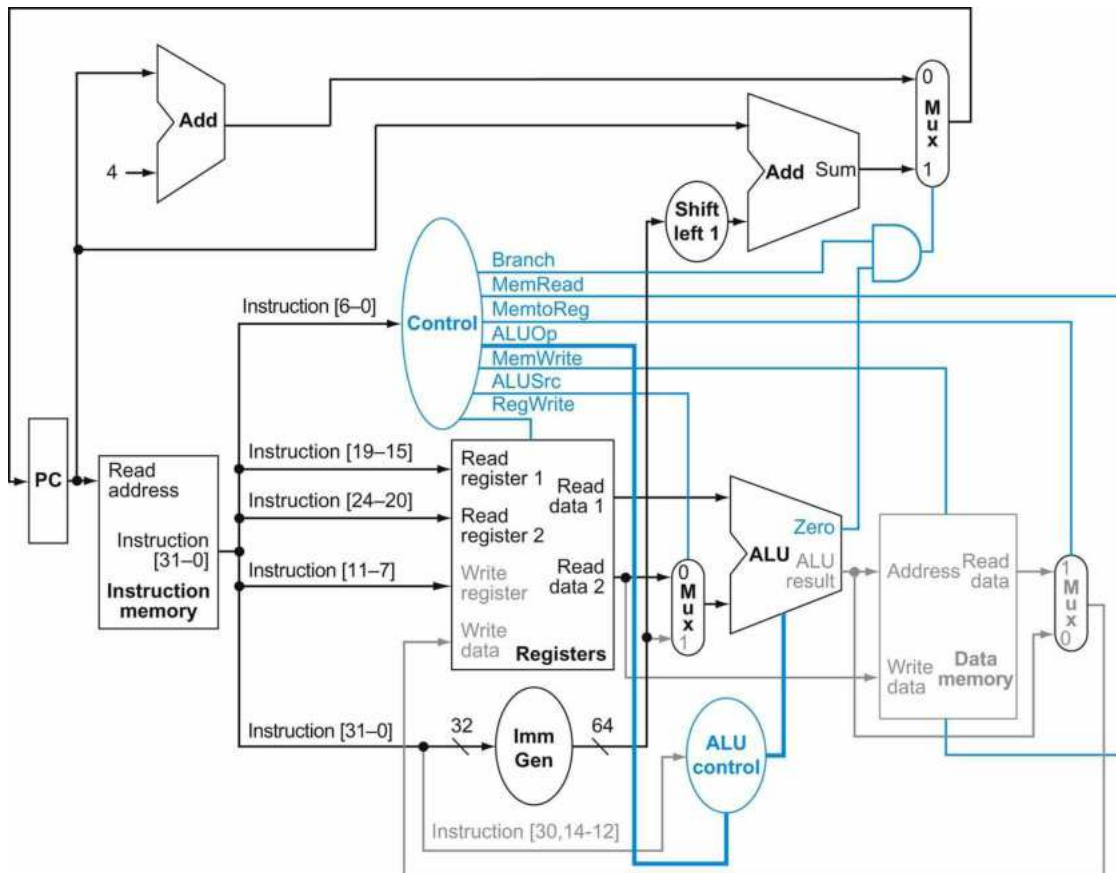


FIGURE 4.21 The datapath in operation for a branch-if-equal instruction.

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 inputs are the opcode bits. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.

[Figure 4.22](#) defines the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and we can implement it directly in gates in an automated fashion. We show

this final step in [Section C.2](#) in  [Appendix C](#).

Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	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 half of the table gives the combinations of input signals that correspond to the four instruction classes, one per column, that determine the control output settings. 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 $Op4 \cdot Op5$, since this is sufficient to distinguish the R-format instructions from *ld*, *sd*, and *beq*. We do not take advantage of this simplification, since the rest of the RISC-V opcodes are used in a full implementation.

Why a Single-Cycle Implementation is not Used Today

Although the single-cycle design will work correctly, it is too

inefficient to be used in modern designs. 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 most likely 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**.



COMMON CASE FAST

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

instructions simultaneously.

Check Yourself

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

4.5 An Overview of Pipelining

pipelining

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

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

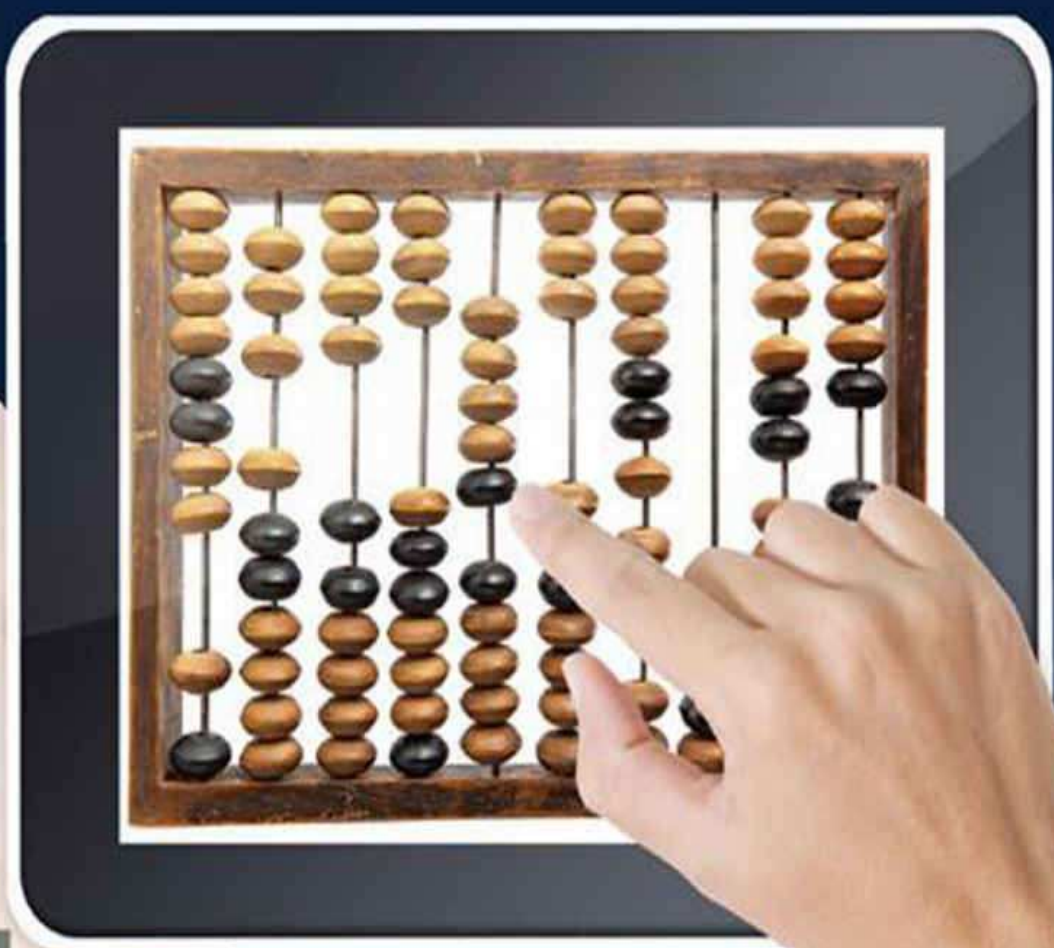
Never waste time.

American proverb

COMPUTER ORGANIZATION AND DESIGN

THE HARDWARE/SOFTWARE INTERFACE

 **RISC-V** EDITION



MK
MORGAN KAUFMANN

DAVID A. PATTERSON
JOHN L. HENNESSY