Microarchitecture

7

7.1 INTRODUCTION

In this chapter, you will learn how to piece together a MIPS microprocessor. Indeed, you will puzzle out three different versions, each with different trade-offs between performance, cost, and complexity.

To the uninitiated, building a microprocessor may seem like black magic. But it is actually relatively straightforward, and by this point you have learned everything you need to know. Specifically, you have learned to design combinational and sequential logic given functional and timing specifications. You are familiar with circuits for arithmetic and memory. And you have learned about the MIPS architecture, which specifies the programmer's view of the MIPS processor in terms of registers, instructions, and memory.

This chapter covers *microarchitecture*, which is the connection between logic and architecture. Microarchitecture is the specific arrangement of registers, ALUs, finite state machines (FSMs), memories, and other logic building blocks needed to implement an architecture. A particular architecture, such as MIPS, may have many different microarchitectures, each with different trade-offs of performance, cost, and complexity. They all run the same programs, but their internal designs vary widely. We will design three different microarchitectures in this chapter to illustrate the trade-offs.

This chapter draws heavily on David Patterson and John Hennessy's classic MIPS designs in their text *Computer Organization and Design*. They have generously shared their elegant designs, which have the virtue of illustrating a real commercial architecture while being relatively simple and easy to understand.

7.1.1 Architectural State and Instruction Set

Recall that a computer architecture is defined by its instruction set and *architectural state*. The architectural state for the MIPS processor consists

David Patterson was the first in his family to graduate from college (UCLA, 1969). He has been a professor of computer science at UC Berkeley since 1977, where he coinvented RISC, the Reduced Instruction Set Computer. In 1984, he developed the SPARC architecture used by Sun Microsystems. He is also the father of *RAID* (*Redundant Array of Inexpensive Disks*) and *NOW* (*Network of Workstations*).

John Hennessy is president of Stanford University and has been a professor of electrical engineering and computer science there since 1977. He coinvented RISC. He developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems. As of 2004, more than 300 million MIPS microprocessors have been sold.

In their copious free time, these two modern paragons write textbooks for recreation and relaxation.

of the program counter and the 32 registers. Any MIPS microarchitecture must contain all of this state. Based on the current architectural state, the processor executes a particular instruction with a particular set of data to produce a new architectural state. Some microarchitectures contain additional *nonarchitectural state* to either simplify the logic or improve performance; we will point this out as it arises.

To keep the microarchitectures easy to understand, we consider only a subset of the MIPS instruction set. Specifically, we handle the following instructions:

- R-type arithmetic/logic instructions: `add`, `sub`, `and`, `or`, `slt`
- Memory instructions: `lw`, `sw`
- Branches: `beq`

After building the microarchitectures with these instructions, we extend them to handle `addi` and `j`. These particular instructions were chosen because they are sufficient to write many interesting programs. Once you understand how to implement these instructions, you can expand the hardware to handle others.

7.1.2 Design Process

We will divide our microarchitectures into two interacting parts: the *datapath* and the *control*. The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers. MIPS is a 32-bit architecture, so we will use a 32-bit datapath. The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

A good way to design a complex system is to start with hardware containing the state elements. These elements include the memories and the architectural state (the program counter and registers). Then, add blocks of combinational logic between the state elements to compute the new state based on the current state. The instruction is read from part of memory; load and store instructions then read or write data from another part of memory. Hence, it is often convenient to partition the overall memory into two smaller memories, one containing instructions and the other containing data. Figure 7.1 shows a block diagram with the four state elements: the program counter, register file, and instruction and data memories.

In Figure 7.1, heavy lines are used to indicate 32-bit data busses. Medium lines are used to indicate narrower busses, such as the 5-bit address busses on the register file. Narrow blue lines are used to indicate

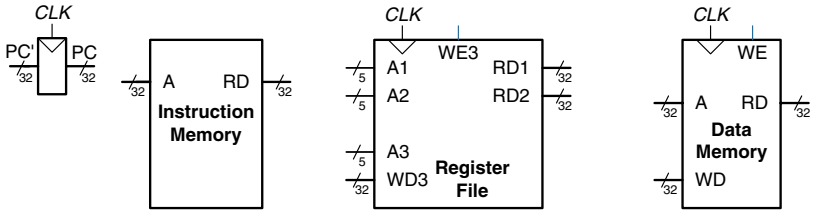

Figure 7.1 State elements of MIPS processor

control signals, such as the register file write enable. We will use this convention throughout the chapter to avoid cluttering diagrams with bus widths. Also, state elements usually have a reset input to put them into a known state at start-up. Again, to save clutter, this reset is not shown.

The *program counter* is an ordinary 32-bit register. Its output, *PC*, points to the current instruction. Its input, *PC'*, indicates the address of the next instruction.

The *instruction memory* has a single read port.[1] It takes a 32-bit instruction address input, *A*, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, *RD*.

The 32-element × 32-bit *register file* has two read ports and one write port. The read ports take 5-bit address inputs, *A1* and *A2*, each specifying one of $2^5 = 32$ registers as source operands. They read the 32-bit register values onto read data outputs *RD1* and *RD2*, respectively. The write port takes a 5-bit address input, *A3*; a 32-bit write data input, *WD*; a write enable input, *WE3*; and a clock. If the write enable is 1, the register file writes the data into the specified register on the rising edge of the clock.

The *data memory* has a single read/write port. If the write enable, *WE*, is 1, it writes data *WD* into address *A* on the rising edge of the clock. If the write enable is 0, it reads address *A* onto *RD*.

The instruction memory, register file, and data memory are all read *combinationally*. In other words, if the address changes, the new data appears at *RD* after some propagation delay; no clock is involved. They are written only on the rising edge of the clock. In this fashion, the state of the system is changed only at the clock edge. The address, data, and write enable must setup sometime before the clock edge and must remain stable until a hold time after the clock edge.

Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits. The microprocessor is

Resetting the PC

At the very least, the program counter must have a reset signal to initialize its value when the processor turns on. MIPS processors initialize the PC to `0xBFC00000` on reset and begin executing code to start up the operating system (OS). The OS then loads an application program at `0x00400000` and begins executing it. For simplicity in this chapter, we will reset the PC to `0x00000000` and place our programs there instead.

[1] This is an oversimplification used to treat the instruction memory as a ROM; in most real processors, the instruction memory must be writable so that the OS can load a new program into memory. The multicycle microarchitecture described in Section 7.4 is more realistic in that it uses a combined memory for instructions and data that can be both read and written.

built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit. Indeed, the processor can be viewed as a giant finite state machine, or as a collection of simpler interacting state machines.

7.1.3 MIPS Microarchitectures

In this chapter, we develop three microarchitectures for the MIPS processor architecture: single-cycle, multicycle, and pipelined. They differ in the way that the state elements are connected together and in the amount of nonarchitectural state.

The *single-cycle microarchitecture* executes an entire instruction in one cycle. It is easy to explain and has a simple control unit. Because it completes the operation in one cycle, it does not require any nonarchitectural state. However, the cycle time is limited by the slowest instruction.

The *multicycle microarchitecture* executes instructions in a series of shorter cycles. Simpler instructions execute in fewer cycles than complicated ones. Moreover, the multicycle microarchitecture reduces the hardware cost by reusing expensive hardware blocks such as adders and memories. For example, the adder may be used on several different cycles for several purposes while carrying out a single instruction. The multicycle microprocessor accomplishes this by adding several nonarchitectural registers to hold intermediate results. The multicycle processor executes only one instruction at a time, but each instruction takes multiple clock cycles.

The *pipelined microarchitecture* applies pipelining to the single-cycle microarchitecture. It therefore can execute several instructions simultaneously, improving the throughput significantly. Pipelining must add logic to handle dependencies between simultaneously executing instructions. It also requires nonarchitectural pipeline registers. The added logic and registers are worthwhile; all commercial high-performance processors use pipelining today.

We explore the details and trade-offs of these three microarchitectures in the subsequent sections. At the end of the chapter, we briefly mention additional techniques that are used to get even more speed in modern high-performance microprocessors.

7.2 PERFORMANCE ANALYSIS

As we mentioned, a particular processor architecture can have many microarchitectures with different cost and performance trade-offs. The cost depends on the amount of hardware required and the implementation technology. Each year, CMOS processes can pack more transistors on a chip for the same amount of money, and processors take advantage

of these additional transistors to deliver more performance. Precise cost calculations require detailed knowledge of the implementation technology, but in general, more gates and more memory mean more dollars. This section lays the foundation for analyzing performance.

There are many ways to measure the performance of a computer system, and marketing departments are infamous for choosing the method that makes their computer look fastest, regardless of whether the measurement has any correlation to real world performance. For example, Intel and Advanced Micro Devices (AMD) both sell compatible microprocessors conforming to the IA-32 architecture. Intel Pentium III and Pentium 4 microprocessors were largely advertised according to clock frequency in the late 1990s and early 2000s, because Intel offered higher clock frequencies than its competitors. However, Intel's main competitor, AMD, sold Athlon microprocessors that executed programs faster than Intel's chips at the same clock frequency. What is a consumer to do?

The only gimmick-free way to measure performance is by measuring the execution time of a program of interest to you. The computer that executes your program fastest has the highest performance. The next best choice is to measure the total execution time of a collection of programs that are similar to those you plan to run; this may be necessary if you haven't written your program yet or if somebody else who doesn't have your program is making the measurements. Such collections of programs are called *benchmarks*, and the execution times of these programs are commonly published to give some indication of how a processor performs.

The execution time of a program, measured in seconds, is given by Equation 7.1.

$$Execution\ Time = \left(\#\ instructions\right)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right) \quad (7.1)$$

The number of instructions in a program depends on the processor architecture. Some architectures have complicated instructions that do more work per instruction, thus reducing the number of instructions in a program. However, these complicated instructions are often slower to execute in hardware. The number of instructions also depends enormously on the cleverness of the programmer. For the purposes of this chapter, we will assume that we are executing known programs on a MIPS processor, so the number of instructions for each program is constant, independent of the microarchitecture.

The number of cycles per instruction, often called *CPI*, is the number of clock cycles required to execute an average instruction. It is the reciprocal of the throughput (instructions per cycle, or *IPC*). Different microarchitectures have different CPIs. In this chapter, we will assume

we have an ideal memory system that does not affect the CPI. In Chapter 8, we examine how the processor sometimes has to wait for the memory, which increases the CPI.

The number of seconds per cycle is the clock period, T_c. The clock period is determined by the critical path through the logic on the processor. Different microarchitectures have different clock periods. Logic and circuit designs also significantly affect the clock period. For example, a carry-lookahead adder is faster than a ripple-carry adder. Manufacturing advances have historically doubled transistor speeds every 4–6 years, so a microprocessor built today will be much faster than one from last decade, even if the microarchitecture and logic are unchanged.

The challenge of the microarchitect is to choose the design that minimizes the execution time while satisfying constraints on cost and/or power consumption. Because microarchitectural decisions affect both CPI and T_c and are influenced by logic and circuit designs, determining the best choice requires careful analysis.

There are many other factors that affect overall computer performance. For example, the hard disk, the memory, the graphics system, and the network connection may be limiting factors that make processor performance irrelevant. The fastest microprocessor in the world doesn't help surfing the Internet on a dial-up connection. But these other factors are beyond the scope of this book.

7.3 SINGLE-CYCLE PROCESSOR

We first design a MIPS microarchitecture that executes instructions in a single cycle. We begin constructing the datapath by connecting the state elements from Figure 7.1 with combinational logic that can execute the various instructions. Control signals determine which specific instruction is carried out by the datapath at any given time. The controller contains combinational logic that generates the appropriate control signals based on the current instruction. We conclude by analyzing the performance of the single-cycle processor.

7.3.1 Single-Cycle Datapath

This section gradually develops the single-cycle datapath, adding one piece at a time to the state elements from Figure 7.1. The new connections are emphasized in black (or blue, for new control signals), while the hardware that has already been studied is shown in gray.

The program counter (PC) register contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. Figure 7.2 shows that the PC is simply connected to the address input of the instruction memory. The instruction memory reads out, or *fetches*, the 32-bit instruction, labeled *Instr*.

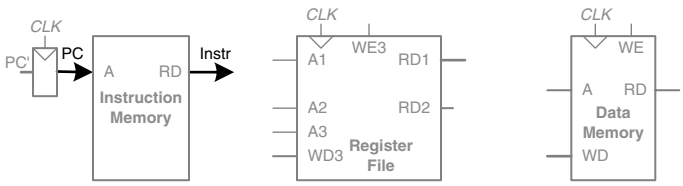

Figure 7.2 Fetch instruction from memory

The processor's actions depend on the specific instruction that was fetched. First we will work out the datapath connections for the `lw` instruction. Then we will consider how to generalize the datapath to handle the other instructions.

For a `lw` instruction, the next step is to read the source register containing the base address. This register is specified in the `rs` field of the instruction, $Instr_{25:21}$. These bits of the instruction are connected to the address input of one of the register file read ports, *A1*, as shown in Figure 7.3. The register file reads the register value onto *RD1*.

The `lw` instruction also requires an offset. The offset is stored in the immediate field of the instruction, $Instr_{15:0}$. Because the 16-bit immediate might be either positive or negative, it must be sign-extended to 32 bits, as shown in Figure 7.4. The 32-bit sign-extended value is called *SignImm*. Recall from Section 1.4.6 that sign extension simply copies the sign bit (most significant bit) of a short input into all of the upper bits of the longer output. Specifically, $SignImm_{15:0} = Instr_{15:0}$ and $SignImm_{31:16} = Instr_{15}$.

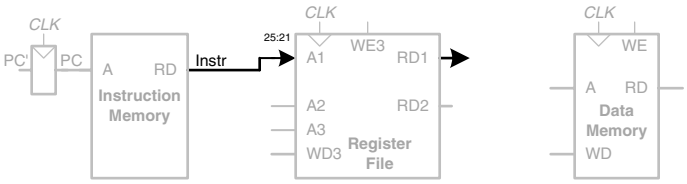

Figure 7.3 Read source operand from register file

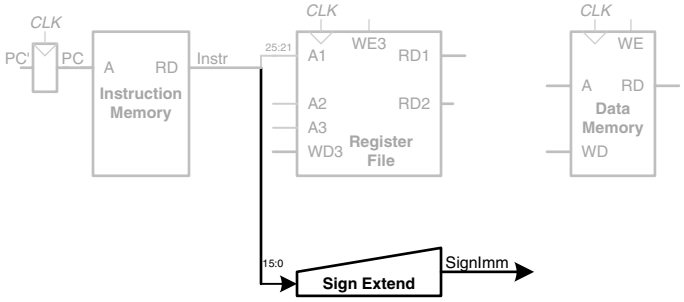

Figure 7.4 Sign-extend the immediate

The processor must add the base address to the offset to find the address to read from memory. Figure 7.5 introduces an ALU to perform this addition. The ALU receives two operands, *SrcA* and *SrcB*. *SrcA* comes from the register file, and *SrcB* comes from the sign-extended immediate. The ALU can perform many operations, as was described in Section 5.2.4. The 3-bit *ALUControl* signal specifies the operation. The ALU generates a 32-bit *ALUResult* and a *Zero* flag, that indicates whether *ALUResult* == 0. For a `lw` instruction, the *ALUControl* signal should be set to 010 to add the base address and offset. *ALUResult* is sent to the data memory as the address for the load instruction, as shown in Figure 7.5.

The data is read from the data memory onto the *ReadData* bus, then written back to the destination register in the register file at the end of the cycle, as shown in Figure 7.6. Port 3 of the register file is the

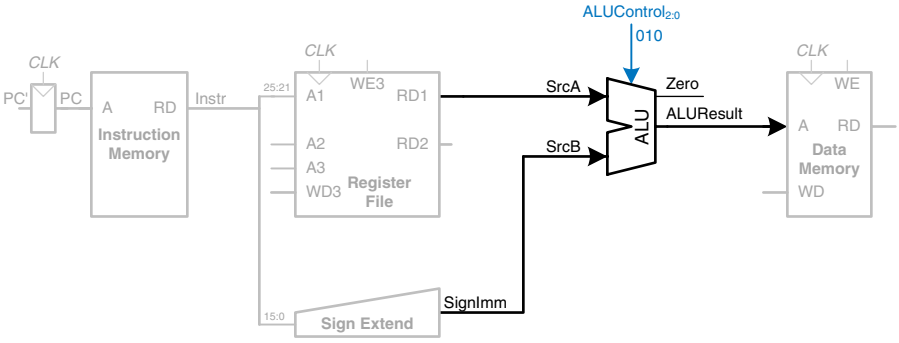

Figure 7.5 Compute memory address

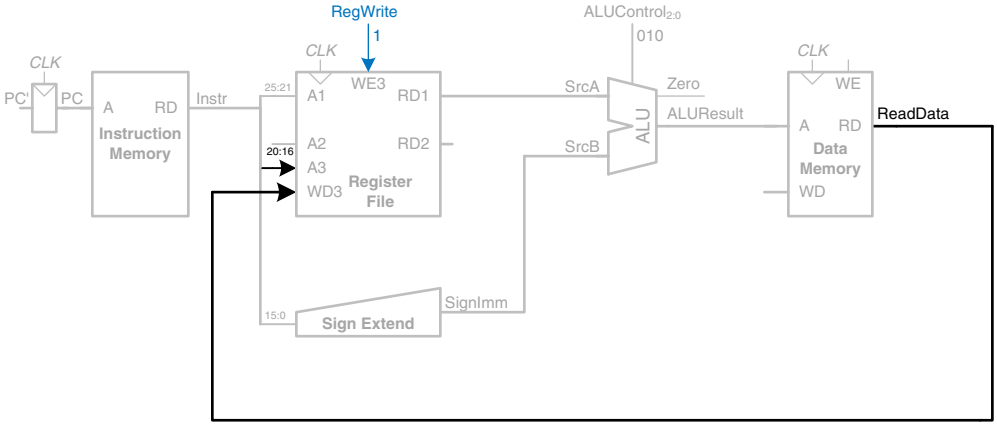

Figure 7.6 Write data back to register file

write port. The destination register for the `lw` instruction is specified in the `rt` field, $Instr_{20:16}$, which is connected to the port 3 address input, *A3*, of the register file. The *ReadData* bus is connected to the port 3 write data input, *WD3*, of the register file. A control signal called *RegWrite* is connected to the port 3 write enable input, *WE3*, and is asserted during a `lw` instruction so that the data value is written into the register file. The write takes place on the rising edge of the clock at the end of the cycle.

While the instruction is being executed, the processor must compute the address of the next instruction, *PC′*. Because instructions are 32 bits = 4 bytes, the next instruction is at *PC* + 4. Figure 7.7 uses another adder to increment the *PC* by 4. The new address is written into the program counter on the next rising edge of the clock. This completes the datapath for the `lw` instruction.

Next, let us extend the datapath to also handle the `sw` instruction. Like the `lw` instruction, the `sw` instruction reads a base address from port 1 of the register and sign-extends an immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported by the datapath.

The `sw` instruction also reads a second register from the register file and writes it to the data memory. Figure 7.8 shows the new connections for this function. The register is specified in the `rt` field, $Instr_{20:16}$. These bits of the instruction are connected to the second register file read port, *A2*. The register value is read onto the *RD2* port. It is connected to the write data port of the data memory. The write enable port of the data memory, *WE*, is controlled by *MemWrite*. For a `sw` instruction, *MemWrite* = 1, to write the data to memory; *ALUControl* = 010, to add the base address

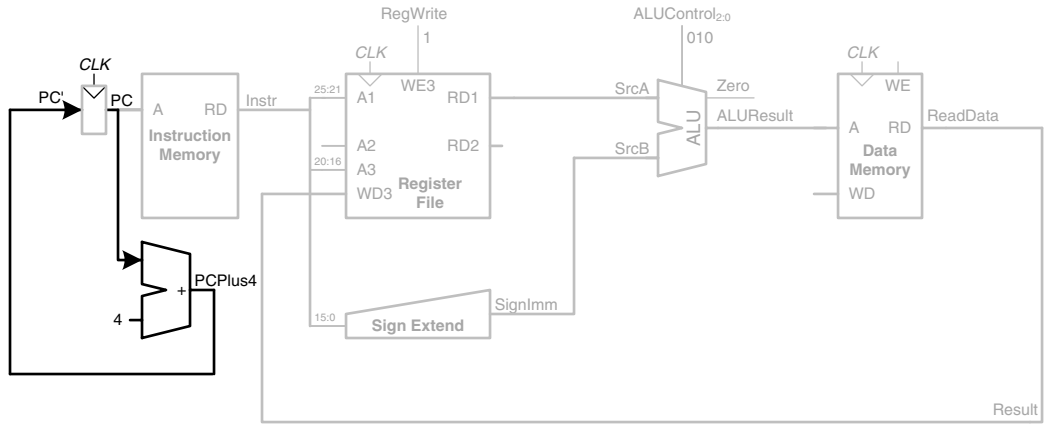

Figure 7.7 Determine address of next instruction for PC

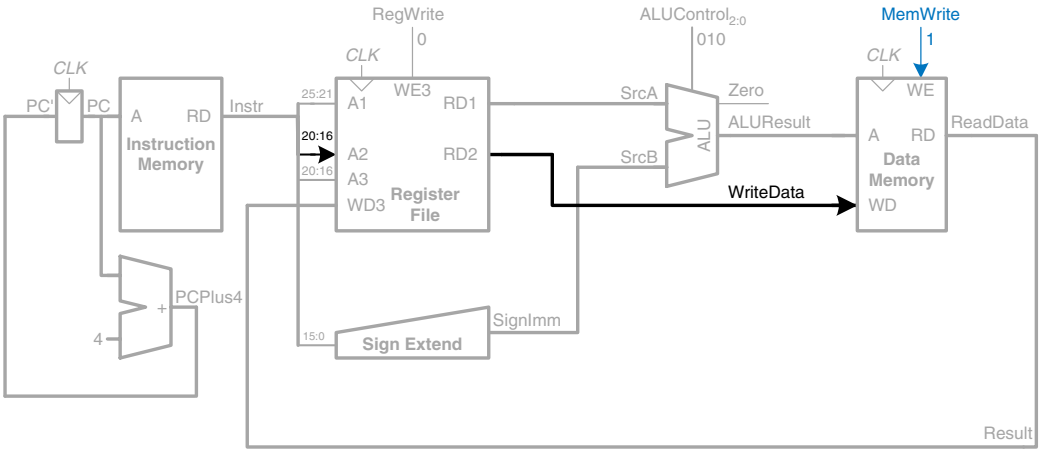

Figure 7.8 Write data to memory for `sw` instruction

and offset; and *RegWrite* = 0, because nothing should be written to the register file. Note that data is still read from the address given to the data memory, but that this *ReadData* is ignored because *RegWrite* = 0.

Next, consider extending the datapath to handle the R-type instructions `add`, `sub`, `and`, `or`, and `slt`. All of these instructions read two registers from the register file, perform some ALU operation on them, and write the result back to a third register file. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware, using different *ALUControl* signals.

Figure 7.9 shows the enhanced datapath handling R-type instructions. The register file reads two registers. The ALU performs an operation on these two registers. In Figure 7.8, the ALU always received its *SrcB* operand from the sign-extended immediate (*SignImm*). Now, we add a multiplexer to choose *SrcB* from either the register file *RD2* port or *SignImm*.

The multiplexer is controlled by a new signal, *ALUSrc*. *ALUSrc* is 0 for R-type instructions to choose *SrcB* from the register file; it is 1 for `lw` and `sw` to choose *SignImm*. This principle of enhancing the datapath's capabilities by adding a multiplexer to choose inputs from several possibilities is extremely useful. Indeed, we will apply it twice more to complete the handling of R-type instructions.

In Figure 7.8, the register file always got its write data from the data memory. However, R-type instructions write the *ALUResult* to the register file. Therefore, we add another multiplexer to choose between *ReadData* and *ALUResult*. We call its output *Result*. This multiplexer is controlled by another new signal, *MemtoReg*. *MemtoReg* is 0

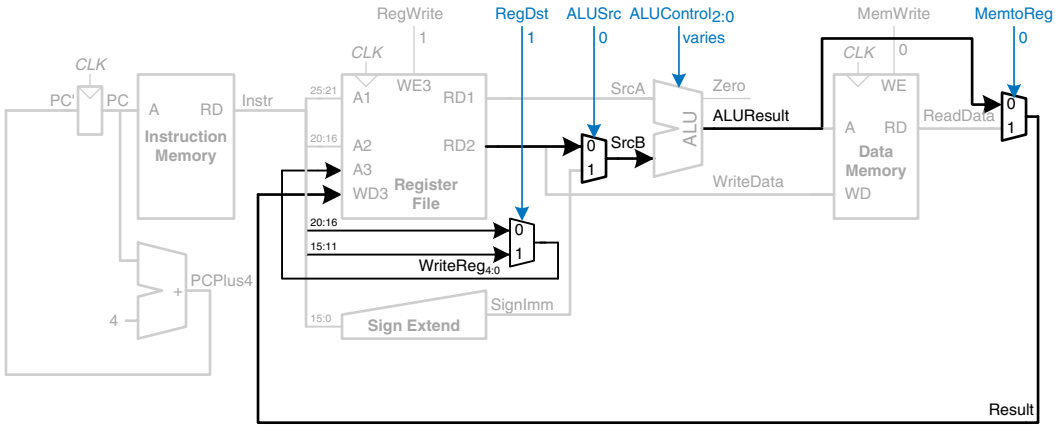

Figure 7.9 Datapath enhancements for R-type instruction

for R-type instructions to choose *Result* from the *ALUResult*; it is 1 for `lw` to choose *ReadData*. We don't care about the value of *MemtoReg* for `sw`, because `sw` does not write to the register file.

Similarly, in Figure 7.8, the register to write was specified by the `rt` field of the instruction, $Instr_{20:16}$. However, for R-type instructions, the register is specified by the `rd` field, $Instr_{15:11}$. Thus, we add a third multiplexer to choose *WriteReg* from the appropriate field of the instruction. The multiplexer is controlled by *RegDst*. *RegDst* is 1 for R-type instructions to choose *WriteReg* from the `rd` field, $Instr_{15:11}$; it is 0 for `lw` to choose the `rt` field, $Instr_{20:16}$. We don't care about the value of *RegDst* for `sw`, because `sw` does not write to the register file.

Finally, let us extend the datapath to handle `beq`. `beq` compares two registers. If they are equal, it takes the branch by adding the branch offset to the program counter. Recall that the offset is a positive or negative number, stored in the `imm` field of the instruction, $Instr_{31:26}$. The offset indicates the number of instructions to branch past. Hence, the immediate must be sign-extended and multiplied by 4 to get the new program counter value: $PC' = PC + 4 + SignImm \times 4$.

Figure 7.10 shows the datapath modifications. The next *PC* value for a taken branch, *PCBranch*, is computed by shifting *SignImm* left by 2 bits, then adding it to *PCPlus4*. The left shift by 2 is an easy way to multiply by 4, because a shift by a constant amount involves just wires. The two registers are compared by computing $SrcA - SrcB$ using the ALU. If *ALUResult* is 0, as indicated by the *Zero* flag from the ALU, the registers are equal. We add a multiplexer to choose *PC'* from either *PCPlus4* or *PCBranch*. *PCBranch* is selected if the instruction is

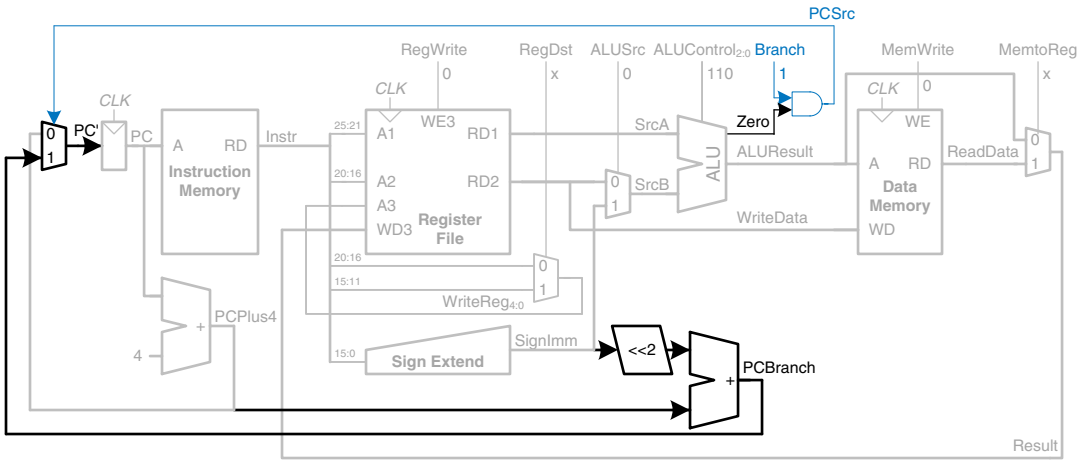

Figure 7.10 Datapath enhancements for `beq` instruction

a branch and the *Zero* flag is asserted. Hence, *Branch* is 1 for `beq` and 0 for other instructions. For `beq`, *ALUControl* = 110, so the ALU performs a subtraction. *ALUSrc* = 0 to choose *SrcB* from the register file. *RegWrite* and *MemWrite* are 0, because a branch does not write to the register file or memory. We don't care about the values of *RegDst* and *MemtoReg*, because the register file is not written.

This completes the design of the single-cycle MIPS processor datapath. We have illustrated not only the design itself, but also the design process in which the state elements are identified and the combinational logic connecting the state elements is systematically added. In the next section, we consider how to compute the control signals that direct the operation of our datapath.

7.3.2 Single-Cycle Control

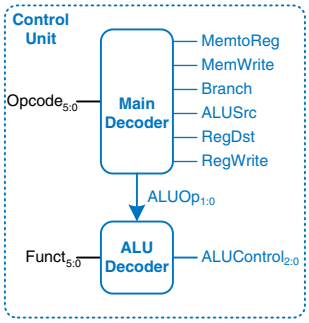

Figure 7.12 Control unit internal structure

The control unit computes the control signals based on the `opcode` and `funct` fields of the instruction, $Instr_{31:26}$ and $Instr_{5:0}$. Figure 7.11 shows the entire single-cycle MIPS processor with the control unit attached to the datapath.

Most of the control information comes from the `opcode`, but R-type instructions also use the `funct` field to determine the ALU operation. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic, as shown in Figure 7.12. The *main decoder* computes most of the outputs from the `opcode`. It also determines a 2-bit *ALUOp* signal. The ALU decoder uses this *ALUOp* signal in conjunction with the `funct` field to compute *ALUControl*. The meaning of the *ALUOp* signal is given in Table 7.1.

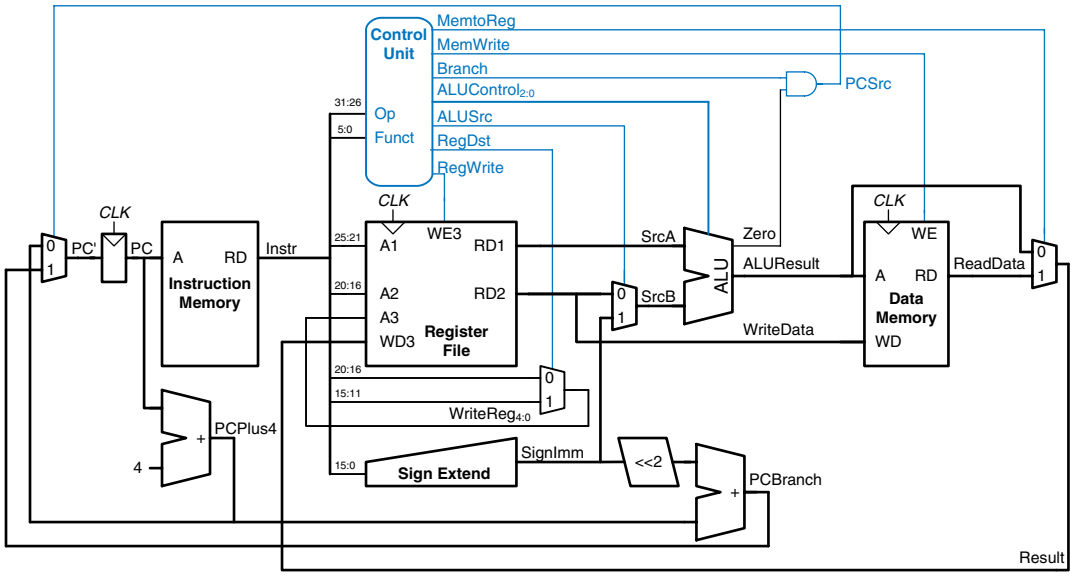

Figure 7.11 Complete single-cycle MIPS processor

Table 7.1 ***ALUOp*** **encoding**

ALUOp	Meaning
00	add
01	subtract
10	look at `funct` field
11	n/a

Table 7.2 is a truth table for the ALU decoder. Recall that the meanings of the three *ALUControl* signals were given in Table 5.1. Because *ALUOp* is never 11, the truth table can use don't care's X1 and 1X instead of 01 and 10 to simplify the logic. When *ALUOp* is 00 or 01, the ALU should add or subtract, respectively. When *ALUOp* is 10, the decoder examines the `funct` field to determine the *ALUControl*. Note that, for the R-type instructions we implement, the first two bits of the `funct` field are always 10, so we may ignore them to simplify the decoder.

The control signals for each instruction were described as we built the datapath. Table 7.3 is a truth table for the main decoder that summarizes the control signals as a function of the opcode. All R-type instructions use the same main decoder values; they differ only in the

Table 7.2 ALU decoder truth table

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

Table 7.3 Main decoder truth table

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

ALU decoder output. Recall that, for instructions that do not write to the register file (e.g., `sw` and `beq`), the *RegDst* and *MemtoReg* control signals are don't cares (X); the address and data to the register write port do not matter because *RegWrite* is not asserted. The logic for the decoder can be designed using your favorite techniques for combinational logic design.

Example 7.1 SINGLE-CYCLE PROCESSOR OPERATION

Determine the values of the control signals and the portions of the datapath that are used when executing an `or` instruction.

Solution: Figure 7.13 illustrates the control signals and flow of data during execution of the `or` instruction. The PC points to the memory location holding the instruction, and the instruction memory fetches this instruction.

The main flow of data through the register file and ALU is represented with a dashed blue line. The register file reads the two source operands specified by $Instr_{25:21}$ and $Instr_{20:16}$. *SrcB* should come from the second port of the register

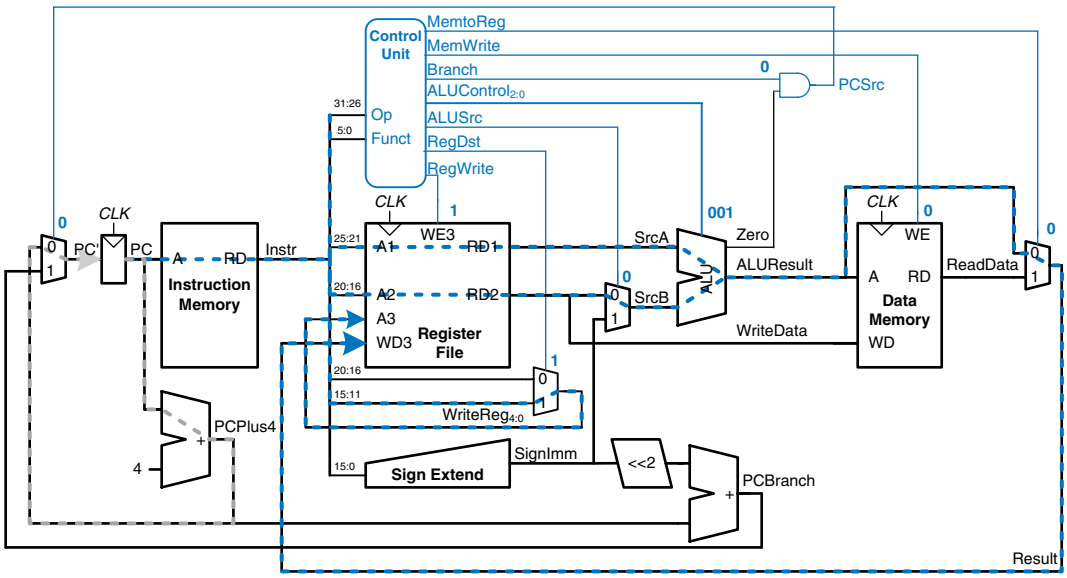

Figure 7.13 Control signals and data flow while executing `or` instruction

file (not *SignImm*), so *ALUSrc* must be 0. `or` is an R-type instruction, so *ALUOp* is 10, indicating that *ALUControl* should be determined from the `funct` field to be 001. *Result* is taken from the ALU, so *MemtoReg* is 0. The result is written to the register file, so *RegWrite* is 1. The instruction does not write memory, so *MemWrite* = 0.

The selection of the destination register is also shown with a dashed blue line. The destination register is specified in the `rd` field, $Instr_{15:11}$, so *RegDst* = 1.

The updating of the PC is shown with the dashed gray line. The instruction is not a branch, so *Branch* = 0 and, hence, *PCSrc* is also 0. The PC gets its next value from *PCPlus4*.

Note that data certainly does flow through the nonhighlighted paths, but that the value of that data is unimportant for this instruction. For example, the immediate is sign-extended and data is read from memory, but these values do not influence the next state of the system.

7.3.3 More Instructions

We have considered a limited subset of the full MIPS instruction set. Adding support for the `addi` and `j` instructions illustrates the principle of how to handle new instructions and also gives us a sufficiently rich instruction set to write many interesting programs. We will see that

supporting some instructions simply requires enhancing the main decoder, whereas supporting others also requires more hardware in the datapath.

Example 7.2 `addi` INSTRUCTION

The add immediate instruction, `addi`, adds the value in a register to the immediate and writes the result to another register. The datapath already is capable of this task. Determine the necessary changes to the controller to support `addi`.

Solution: All we need to do is add a new row to the main decoder truth table showing the control signal values for `addi`, as given in Table 7.4. The result should be written to the register file, so *RegWrite* = 1. The destination register is specified in the `rt` field of the instruction, so *RegDst* = 0. *SrcB* comes from the immediate, so *ALUSrc* = 1. The instruction is not a branch, nor does it write memory, so *Branch* = *MemWrite* = 0. The result comes from the ALU, not memory, so *MemtoReg* = 0. Finally, the ALU should add, so *ALUOp* = 00.

Table 7.4 Main decoder truth table enhanced to support `addi`

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01
`addi`	001000	1	0	1	0	0	0	00

Example 7.3 `j` INSTRUCTION

The jump instruction, `j`, writes a new value into the PC. The two least significant bits of the PC are always 0, because the PC is word aligned (i.e., always a multiple of 4). The next 26 bits are taken from the jump address field in $Instr_{25:0}$. The upper four bits are taken from the old value of the PC.

The existing datapath lacks hardware to compute *PC′* in this fashion. Determine the necessary changes to both the datapath and controller to handle `j`.

Solution: First, we must add hardware to compute the next PC value, *PC′*, in the case of a `j` instruction and a multiplexer to select this next PC, as shown in Figure 7.14. The new multiplexer uses the new *Jump* control signal.

Now we must add a row to the main decoder truth table for the `j` instruction and a column for the *Jump* signal, as shown in Table 7.5. The *Jump* control signal is 1 for the `j` instruction and 0 for all others. `j` does not write the register file or memory, so *RegWrite* = *MemWrite* = 0. Hence, we don't care about the computation done in the datapath, and *RegDst* = *ALUSrc* = *Branch* = *MemtoReg* = *ALUOp* = X.

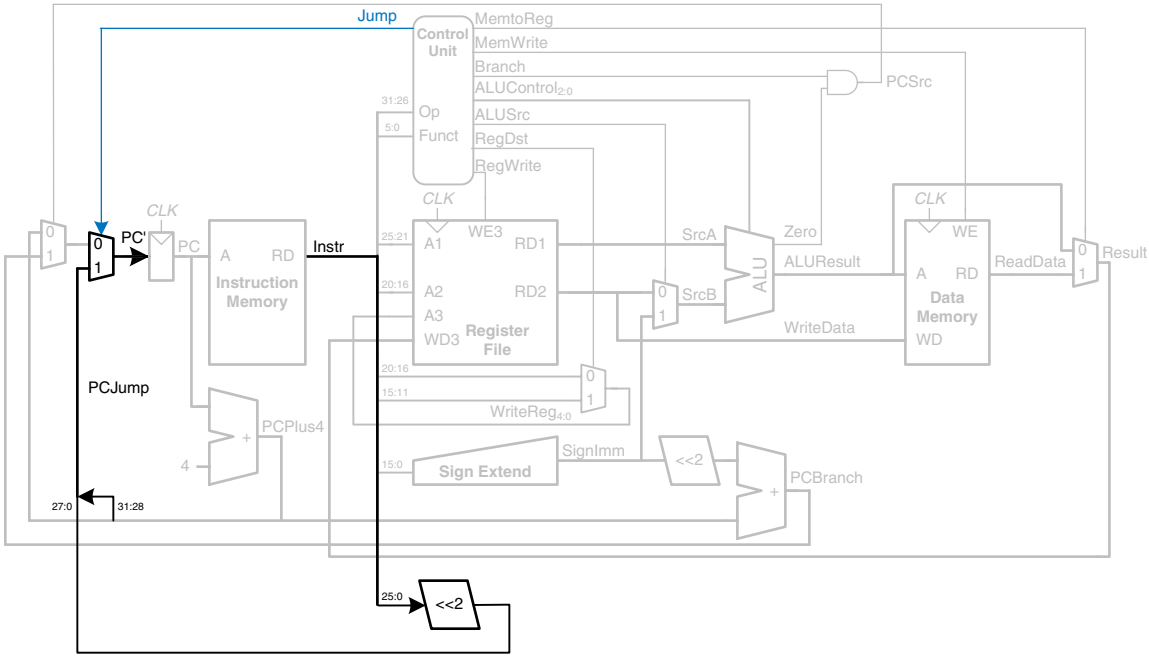

Figure 7.14 Single-cycle MIPS datapath enhanced to support the `j` instruction

Table 7.5 Main decoder truth table enhanced to support `j`

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	00	0
`sw`	101011	0	X	1	0	1	X	00	0
`beq`	000100	0	X	0	1	0	X	01	0
`addi`	001000	1	0	1	0	0	0	00	0
`j`	000010	0	X	X	X	0	X	XX	1

7.3.4 Performance Analysis

Each instruction in the single-cycle processor takes one clock cycle, so the CPI is 1. The critical path for the `lw` instruction is shown in Figure 7.15 with a heavy dashed blue line. It starts with the PC loading a new address on the rising edge of the clock. The instruction memory reads the next instruction. The register file reads *SrcA*. While the register file is reading, the immediate field is sign-extended and selected at the *ALUSrc* multiplexer to determine *SrcB*. The ALU adds *SrcA* and *SrcB* to find the effective address. The data memory reads from this address. The *MemtoReg* multiplexer selects *ReadData*. Finally, *Result* must setup at the register file before the next rising clock edge, so that it can be properly written. Hence, the cycle time is

$$T_c = t_{pcq\_\mathrm{PC}} + t_{\mathrm{mem}} + \max[t_{RF\mathrm{read}}, t_{sext}] + t_{\mathrm{mux}} + t_{\mathrm{ALU}} + t_{\mathrm{mem}} + t_{\mathrm{mux}} + t_{RF\mathrm{setup}} \quad (7.2)$$

In most implementation technologies, the ALU, memory, and register file accesses are substantially slower than other operations. Therefore, the cycle time simplifies to

$$T_c = t_{pcq\_\mathrm{PC}} + 2t_{\mathrm{mem}} + t_{RF\mathrm{read}} + 2t_{\mathrm{mux}} + t_{\mathrm{ALU}} + t_{RF\mathrm{setup}} \quad (7.3)$$

The numerical values of these times will depend on the specific implementation technology.

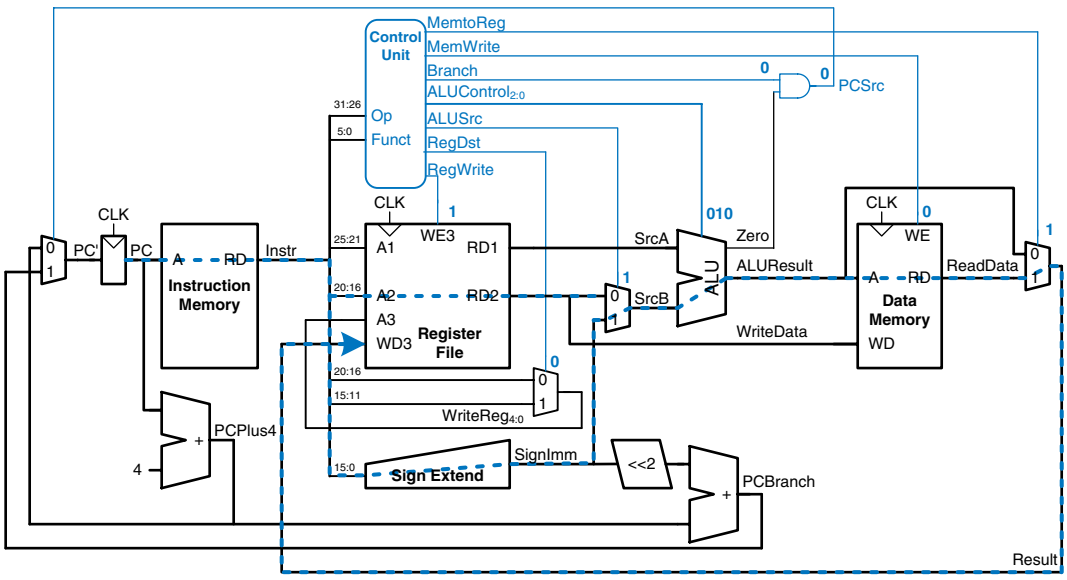

Figure 7.15 Critical path for `lw` instruction

Other instructions have shorter critical paths. For example, R-type instructions do not need to access data memory. However, we are disciplining ourselves to synchronous sequential design, so the clock period is constant and must be long enough to accommodate the slowest instruction.

Example 7.4 SINGLE-CYCLE PROCESSOR PERFORMANCE

Ben Bitdiddle is contemplating building the single-cycle MIPS processor in a 65 nm CMOS manufacturing process. He has determined that the logic elements have the delays given in Table 7.6. Help him compare the execution time for a program with 100 billion instructions.

Solution: According to Equation 7.3, the cycle time of the single-cycle processor is $T_{c1} = 30 + 2(250) + 150 + 2(25) + 200 + 20 = 950$ ps. We use the subscript "1" to distinguish it from subsequent processor designs. According to Equation 7.1, the total execution time is $T_1 = (100 \times 10^9 \text{ instructions})(1 \text{ cycle/instruction})(950 \times 10^{-12} \text{ s/cycle}) = 95$ seconds.

Table 7.6 Delays of circuit elements

Element	Parameter	Delay (ps)
register clk-to-Q	t_{pcq}	30
register setup	t_{setup}	20
multiplexer	t_{mux}	25
ALU	t_{ALU}	200
memory read	t_{mem}	250
register file read	t_{RFread}	150
register file setup	$t_{RFsetup}$	20

7.4 MULTICYCLE PROCESSOR

The single-cycle processor has three primary weaknesses. First, it requires a clock cycle long enough to support the slowest instruction (`lw`), even though most instructions are faster. Second, it requires three adders (one in the ALU and two for the PC logic); adders are relatively expensive circuits, especially if they must be fast. And third, it has separate instruction and data memories, which may not be realistic. Most computers have a single large memory that holds both instructions and data and that can be read and written.