ity of a pipelined processor. For those, who want to understand how the hardware really implements the control, forge ahead!

**Check Yourself**

Look at the control signal in Figure 5.22 on page 312. Can any control signal 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?

## 5.5 A Multicycle Implementation

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

In an earlier example, we broke each instruction into a series of steps corresponding to the functional unit operations that were needed. We can use these steps to create a **multicycle implementation**. In a multicycle implementation, each *step* in the execution will take 1 clock cycle. The multicycle implementation allows a functional unit to be used more than once per instruction, as long as it is used on different clock cycles. This sharing can help reduce the amount of hardware required. The ability to allow instructions to take different numbers of clock cycles and the ability to share functional units within the execution of a single instruction are the major advantages of a multicycle design. Figure 5.25 shows the abstract version of the mul-

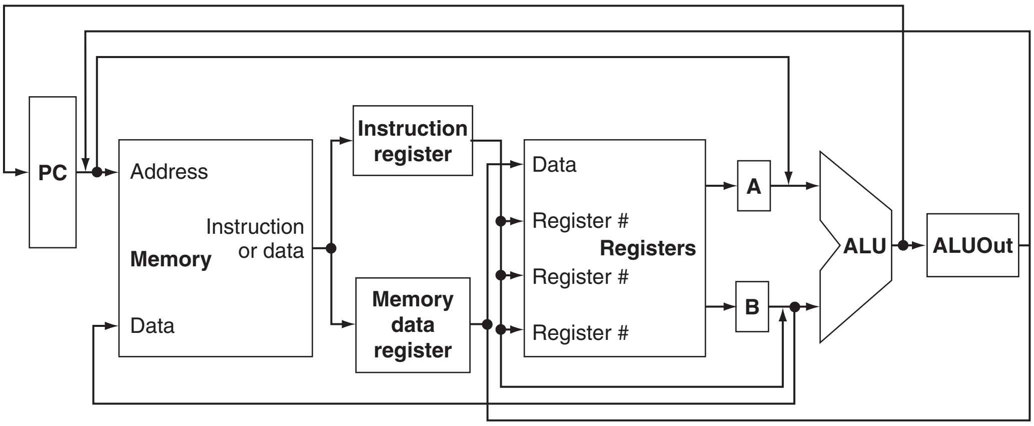

**FIGURE 5.25 The high-level view of the multicycle datapath.** This picture shows the key elements of the datapath: a shared memory unit, a single ALU shared among instructions, and the connections among these shared units. The use of shared functional units requires the addition or widening of multiplexors as well as new temporary registers that hold data between clock cycles of the same instruction. The additional registers are the Instruction register (IR), the Memory data register (MDR), A, B, and ALUOut.

ticycle datapath. If we compare Figure 5.25 to the datapath for the single-cycle version in Figure 5.11 on page 300, we can see the following differences:

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

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

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

The following temporary registers are added to meet these requirements:

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

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

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

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

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

Figure 5.26 shows the details of the datapath with these additional multiplexors. By introducing a few registers and multiplexors, we are able to reduce the number of memory units from two to one and eliminate two adders. Since registers and multiplexors are fairly small compared to a memory unit or ALU, this could yield a substantial reduction in the hardware cost.

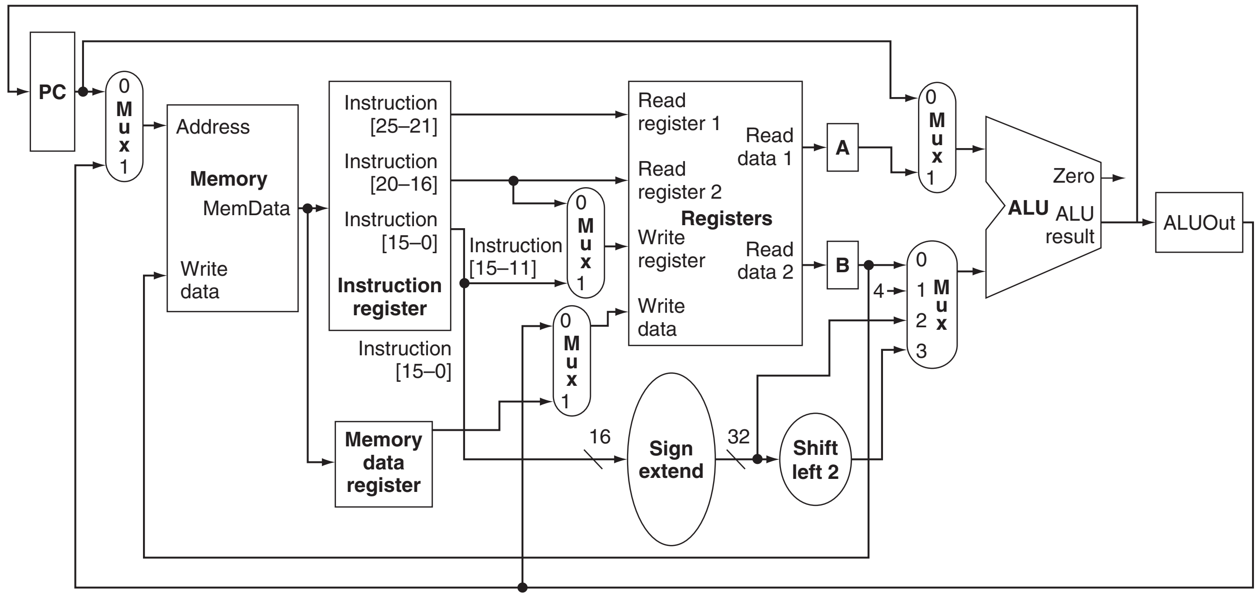

**FIGURE 5.26 Multicycle datapath for MIPS handles the basic instructions.** Although this datapath supports normal incrementing of the PC, a few more connections and a multiplexor will be needed for branches and jumps; we will add these shortly. The additions versus the single-clock datapath include several registers (IR, MDR, A, B, ALUOut), a multiplexor for the memory address, a multiplexor for the top ALU input, and expanding the multiplexor on the bottom ALU input into a four-way selector. These small additions allow us to remove two adders and a memory unit.

Because the datapath shown in Figure 5.26 takes multiple clock cycles per instruction, it will require a different set of control signals. The programmer-visible state units (the PC, the memory, and the registers) as well as the IR will need write control signals. The memory will also need a read signal. We can use the ALU control unit from the single-cycle datapath (see Figure 5.13 and Appendix C) to control the ALU here as well. Finally, each of the two-input multiplexors requires a single control line, while the four-input multiplexor requires two control lines. Figure 5.27 shows the datapath of Figure 5.26 with these control lines added.

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

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

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

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

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

Figure 5.28 shows the complete multicycle datapath and control unit, including the additional control signals and multiplexor for implementing the PC updating.

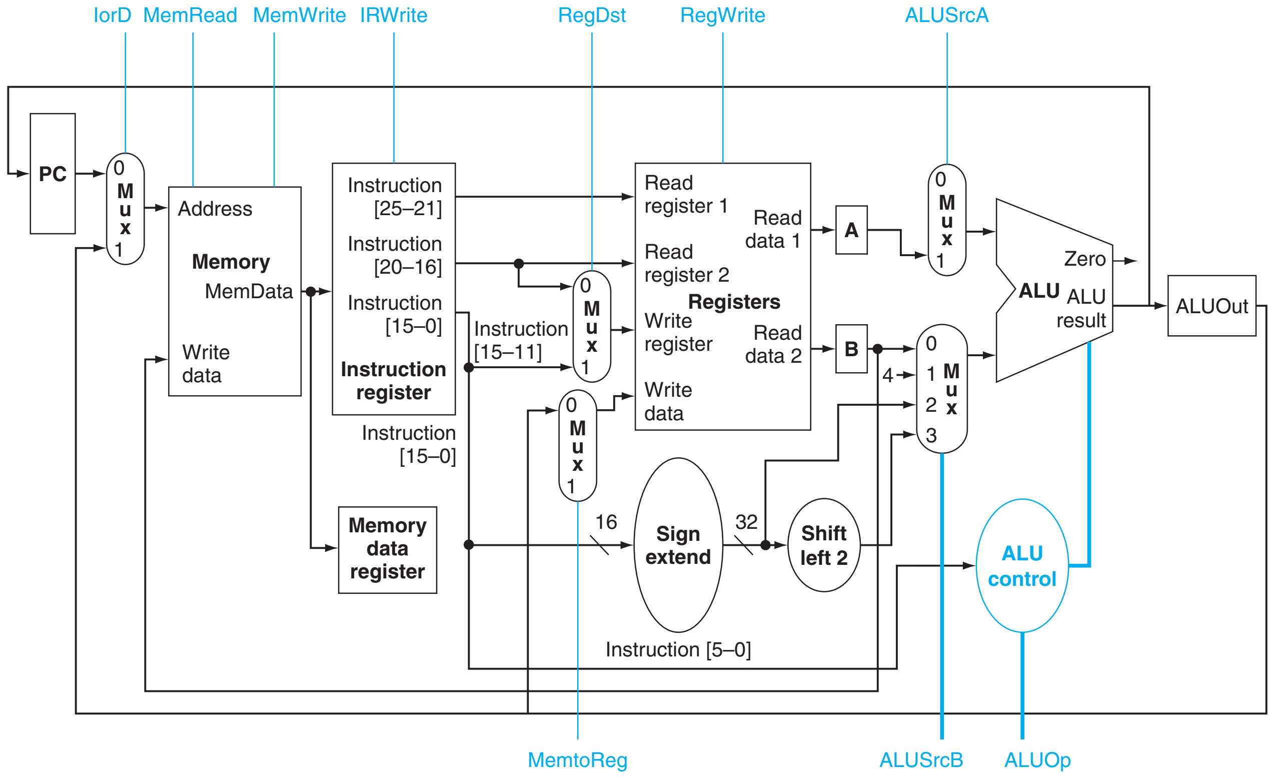

**FIGURE 5.27 The multicycle datapath from Figure 5.26 with the control lines shown.** The signals ALUOp and ALUSrcB are 2-bit control signals, while all the other control lines are 1-bit signals. Neither register A nor B requires a write signal, since their contents are only read on the cycle immediately after it is written. The memory data register has been added to hold the data from a load when the data returns from memory. Data from a load returning from memory cannot be written directly into the register file since the clock cycle cannot accommodate the time required for both the memory access and the register file write. The MemRead signal has been moved to the top of the memory unit to simplify the figures. The full set of datapaths and control lines for branches will be added shortly.

Before examining the steps to execute each instruction, let us informally examine the effect of all the control signals (just as we did for the single-cycle design in Figure 5.16 on page 306). Figure 5.29 shows what each control signal does when asserted and deasserted.

**Elaboration:** To reduce the number of signal lines interconnecting the functional units, designers can use *shared buses*. A shared bus is a set of lines that connect multiple units; in most cases, they include multiple sources that can place data on the bus

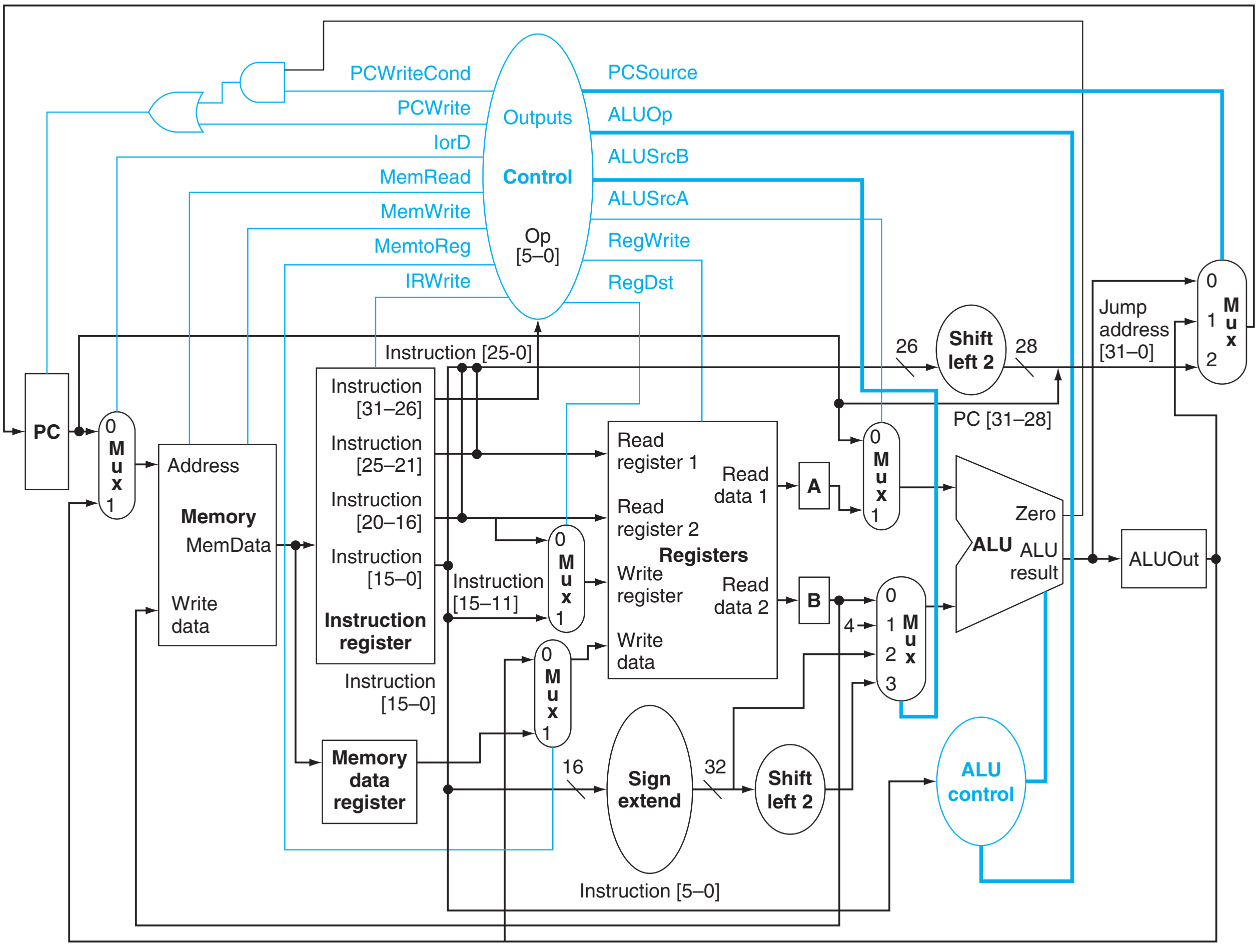

**FIGURE 5.28 The complete datapath for the multicycle implementation together with the necessary control lines.** The control lines of Figure 5.27 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure 5.27 include the multiplexor used to select the source of a new PC value; gates used to combine the PC write signals; and the control signals PCSource, PCWrite, and PCWriteCond. The PCWriteCond signal is used to decide whether a conditional branch should be taken. Support for jumps is included.

and multiple readers of the value. Just as we reduced the number of functional units for the datapath, we can reduce the number of buses interconnecting these units by sharing the buses. For example, there are six sources coming to the ALU; however, only two of them are needed at any one time. Thus, a pair of buses can be used to hold values that are being sent to the ALU. Rather than placing a large multiplexor in front of the

**Actions of the 1-bit control signals**

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

**Actions of the 2-bit control signals**

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

**FIGURE 5.29 The action caused by the setting of each control signal in Figure 5.28 on page 323.** The top table describes the 1-bit control signals, while the bottom table describes the 2-bit signals. Only those control lines that affect multiplexors have an action when they are deasserted. This information is similar to that in Figure 5.16 on page 306 for the single-cycle datapath, but adds several new control lines (IRWrite, PCWrite, PCWriteCond, ALUSrcB, and PCSource) and removes control lines that are no longer used or have been replaced (PCSrc, Branch, and Jump).

ALU, a designer can use a shared bus and then ensure that only one of the sources is driving the bus at any point. Although this saves signal lines, the same number of control lines will be needed to control what goes on the bus. The major drawback to using such bus structures is a potential performance penalty, since a bus is unlikely to be as fast as a point-to-point connection.

## Breaking the Instruction Execution into Clock Cycles

Given the datapath in Figure 5.28, we now need to look at what should happen in each clock cycle of the multicycle execution, since this will determine what additional control signals may be needed, as well as the setting of the control signals. Our goal in breaking the execution into clock cycles should be to maximize performance. We can begin by breaking the execution of any instruction into a series of steps, each taking one clock cycle, attempting to keep the amount of work per cycle roughly equal. For example, we will restrict each step to contain at most one ALU operation, or one register file access, or one memory access. With this restriction, the clock cycle could be as short as the longest of these operations.

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

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

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

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

### 1. Instruction fetch step

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

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

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

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

### 2. Instruction decode and register fetch step

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

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

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

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

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

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

### 3. Execution, memory address computation, or branch completion

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

*Memory reference:*

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

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

*Arithmetic-logical instruction (R-type):*

```
ALUOut <= A op B;
```

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

*Branch:*

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

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

*Jump:*

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

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

### 4. Memory access or R-type instruction completion step

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

*Memory reference:*

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

or

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

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

*Arithmetic-logical instruction (R-type):*

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

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

## 5. Memory read completion step

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

*Load:*

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

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

This five-step sequence is summarized in Figure 5.30. From this sequence we can determine what the control must do on each clock cycle.

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

**FIGURE 5.30 Summary of the steps taken to execute any instruction class.** Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

## Defining the Control

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

In this subsection and in Section 5.7, we will look at two different techniques to specify the control. The first technique is based on finite state machines that are usually represented graphically. The second technique, called **microprogramming**, uses a programming representation for control. Both of these techniques represent the control in a form that allows the detailed implementation—using gates, ROMs, or PLAs—to be synthesized by a CAD system. In this chapter, we will focus on the design of the control and its representation in these two forms.

**microprogram** A symbolic representation of control in the form of instructions, called microinstructions, that are executed on a simple micromachine.

Section 5.8 shows how hardware design languages are used to design modern processors with examples of both the multicycle datapath and the finite state control. In modern digital systems design, the final step of taking a hardware description to actual gates is handled by logic and datapath synthesis tools. Appendix C shows how this process operates by translating the multicycle control unit to a detailed hardware implementation. The key ideas of control can be grasped from this chapter without examining the material in either Section 5.8 or Appendix C. However, if you want to actually do some hardware design, Section 5.9 is useful, and Appendix C can show you what the implementations are likely to look like at the gate level.

Given this implementation, and the knowledge that each state requires 1 clock cycle, we can find the CPI for a typical instruction mix.

### CPI in a Multicycle CPU

**EXAMPLE**

Using the SPECINT2000 instruction mix shown in Figure 3.26, what is the CPI, assuming that each state in the multicycle CPU requires 1 clock cycle?

**ANSWER**

The mix is 25% loads (1% load byte + 24% load word), 10% stores (1% store byte + 9% store word), 11% branches (6% `beq`, 5% `bne`), 2% jumps (1% `jal` + 1% `jr`), and 52% ALU (all the rest of the mix, which we assume to be ALU instructions). From Figure 5.30 on page 329, the number of clock cycles for each instruction class is the following:

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

The CPI is given by the following:

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

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

The ratio

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

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

$$\text{CPI} = 0.25 \times 5 + 0.10 \times 4 + 0.52 \times 4 + 0.11 \times 3 + 0.02 \times 3 = 4.12$$

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

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

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

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

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

The finite state control essentially corresponds to the five steps of execution shown on pages 325 through 329; each state in the finite state machine will take 1 clock cycle. The finite state machine will consist of several parts. Since the first two steps of execution are identical for every instruction, the initial two states of the finite state machine will be common for all instructions. Steps 3 through 5 differ, depending on the opcode. After the execution of the last step for a particular instruction class, the finite state machine will return to the initial state to begin fetching the next instruction.

Figure 5.31 shows this abstracted representation of the finite state machine. To fill in the details of the finite state machine, we will first expand the instruction fetch and decode portion, and then we will show the states (and actions) for the different instruction classes.

We show the first two states of the finite state machine in Figure 5.32 using a traditional graphic representation. We number the states to simplify the explanation, though the numbers are arbitrary. State 0, corresponding to step 1, is the starting state of the machine.

The signals that are asserted in each state are shown within the circle representing the state. The arcs between states define the next state and are labeled with

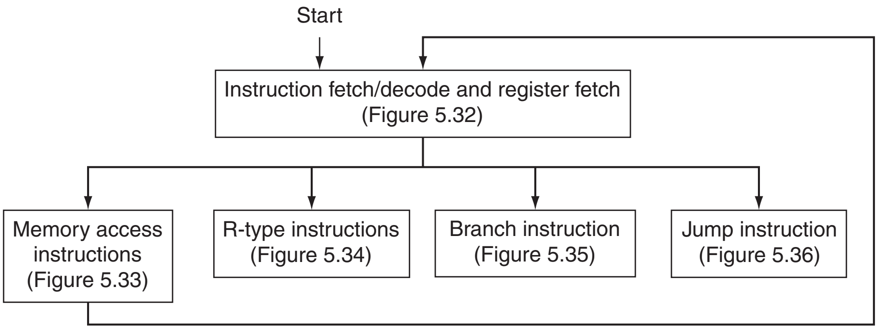
Start
Instruction fetch/decode and register fetch (Figure 5.32)
Memory access instructions (Figure 5.33)
R-type instructions (Figure 5.34)
Branch instruction (Figure 5.35)
Jump instruction (Figure 5.36)

**FIGURE 5.31 The high-level view of the finite state machine control.** The first steps are independent of the instruction class; then a series of sequences that depend on the instruction opcode are used to complete each instruction class. After completing the actions needed for that instruction class, the control returns to fetch a new instruction. Each box in this figure may represent one to several states. The arc labeled *Start* marks the state in which to begin when the first instruction is to be fetched.

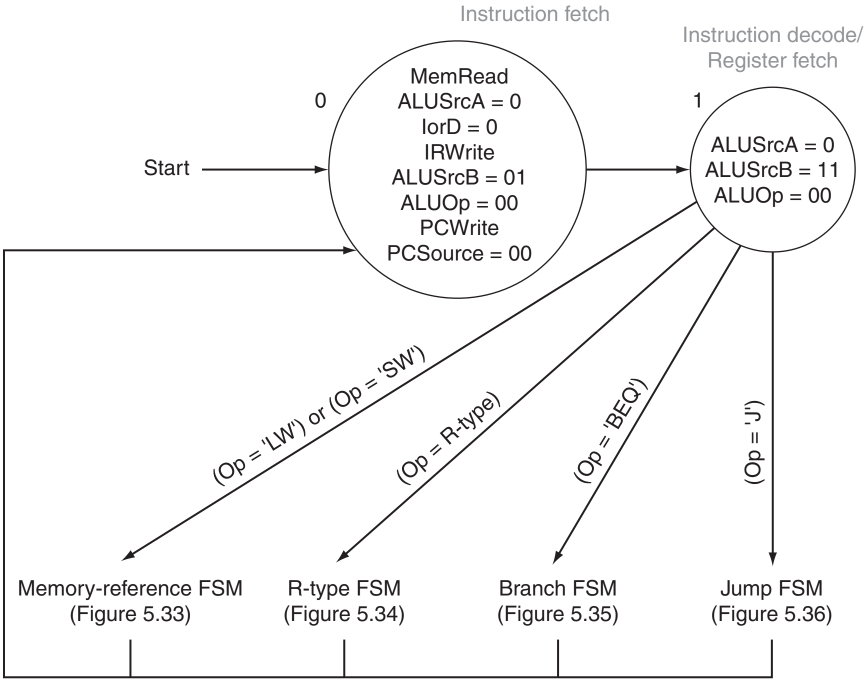

**FIGURE 5.32 The instruction fetch and decode portion of every instruction is identical.** These states correspond to the top box in the abstract finite state machine in Figure 5.31. In the first state we assert two signals to cause the memory to read an instruction and write it into the Instruction register (MemRead and IRWrite), and we set IorD to 0 to choose the PC as the address source. The signals ALUSrcA, ALUSrcB, ALUOp, PCWrite, and PCSource are set to compute PC + 4 and store it into the PC. (It will also be stored into ALUOut, but never used from there.) In the next state, we compute the branch target address by setting ALUSrcB to 11 (causing the shifted and sign-extended lower 16 bits of the IR to be sent to the ALU), setting ALUSrcA to 0 and ALUOp to 00; we store the result in the ALUOut register, which is written on every cycle. There are four next states that depend on the class of the instruction, which is known during this state. The control unit input, called Op, is used to determine which of these arcs to follow. Remember that all signals not explicitly asserted are deasserted; this is particularly important for signals that control writes. For multiplexors controls, lack of a specific setting indicates that we do not care about the setting of the multiplexor.

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

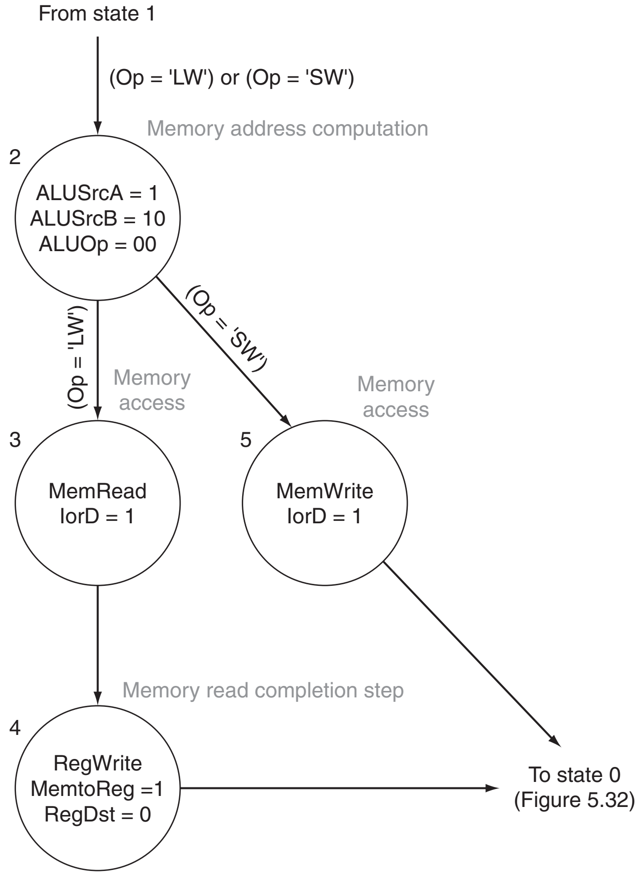

**FIGURE 5.33 The finite state machine for controlling memory-reference instructions has four states.** These states correspond to the box labeled "Memory access instructions" in Figure 5.31. After performing a memory address calculation, a separate sequence is needed for load and for store. The setting of the control signals ALUSrcA, ALUSrcB, and ALUOp is used to cause the memory address computation in state 2. Loads require an extra state to write the result from the MDR (where the result is written in state 3) into the register file.

Figure 5.33 shows the portion of the finite state machine needed to implement the memory-reference instructions. For the memory-reference instructions, the first state after fetching the instruction and registers computes the memory address (state 2). To compute the memory address, the ALU input multiplexors must be set so that the first input is the A register, while the second input is the sign-extended displacement field; the result is written into the ALUOut register. After the memory address calculation, the memory should be read or written; this requires two different states. If the instruction opcode is `lw`, then state 3 (corresponding to the step Memory access) does the memory read (MemRead is asserted). The output of the memory is always written into MDR. If it is `sw`, state 5 does a memory write (MemWrite is asserted). In states 3 and 5, the signal IorD is set to 1 to force the memory address to come from the ALU. After performing a write, the instruction `sw` has completed execution, and the next state is state 0. If the instruction is a load, however, another state (state 4) is needed to write the result from the memory into the register file. Setting the multiplexor controls MemtoReg = 1 and RegDst = 0 will send the loaded value in the MDR to be written into the register file, using rt as the register number. After this state, corresponding to the Memory read completion step, the next state is state 0.

To implement the R-type instructions requires two states corresponding to steps 3 (Execute) and 4 (R-type completion). Figure 5.34 shows this two-state portion of the finite state machine. State 6 asserts ALUSrcA and sets the ALUSrcB

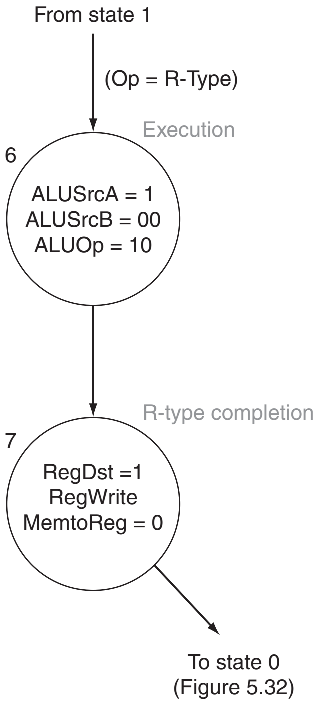

**FIGURE 5.34 R-type instructions can be implemented with a simple two-state finite state machine.** These states correspond to the box labeled "R-type instructions" in Figure 5.31. The first state causes the ALU operation to occur, while the second state causes the ALU result (which is in ALUOut) to be written in the register file. The three signals asserted during state 7 cause the contents of ALUOut to be written into the register file in the entry specified by the rd field of the Instruction register.

signals to 00; this forces the two registers that were read from the register file to be used as inputs to the ALU. Setting ALUOp to 10 causes the ALU control unit to use the function field to set the ALU control signals. In state 7, RegWrite is asserted to cause the register file to write, RegDst is asserted to cause the rd field to be used as the register number of the destination, and MemtoReg is deasserted to select ALUOut as the source of the value to write into the register file.

For branches, only a single additional state is necessary because they complete execution during the third step of instruction execution. During this state, the control signals that cause the ALU to compare the contents of registers A and B must be set, and the signals that cause the PC to be written conditionally with the address in the ALUOut register are also set. To perform the comparison requires that we assert ALUSrcA and set ALUSrcB to 00, and set the ALUOp value to 01 (forcing a subtract). (We use only the Zero output of the ALU, not the result of the subtraction.) To control the writing of the PC, we assert PCWriteCond and set PCSource = 01, which will cause the value in the ALUOut register (containing the branch address calculated in state 1, Figure 5.32 on page 333) to be written into the PC if the Zero bit out of the ALU is asserted. Figure 5.35 shows this single state.

The last instruction class is jump; like branch, it requires only a single state (shown in Figure 5.36) to complete its execution. In this state, the signal PCWrite is asserted to cause the PC to be written. By setting PCSource to 10, the value supplied for writing will be the lower 26 bits of the Instruction register with $00_{two}$ added as the low-order bits concatenated with the upper 4 bits of the PC.

We can now put these pieces of the finite state machine together to form a specification for the control unit, as shown in Figure 5.38. In each state, the signals that are asserted are shown. The next state depends on the opcode bits of the instruction, so we label the arcs with a comparison for the corresponding instruction opcodes.

A finite state machine can be implemented with a temporary register that holds the current state and a block of combinational logic that determines both the datapath signals to be asserted as well as the next state. Figure 5.37 shows how such an implementation might look. Appendix C describes in detail how the finite state machine is implemented using this structure. In Section C.3, the combinational control logic for the finite state machine of Figure 5.38 is implemented both with a ROM (read-only memory) and a PLA (programmable logic array). (Also see Appendix B for a description of these logic elements.) In the next section of this chapter, we consider another way to represent control. Both of these techniques are simply different representations of the same control information.

Pipelining, which is the subject of Chapter 6, is almost always used to accelerate the execution of instructions. For simple instructions, pipelining is capable of achieving the higher clock rate of a multicycle design and a single-cycle CPI of a single-clock design. In most pipelined processors, however, some instructions take longer than a single cycle and require multicycle control. Floating point-

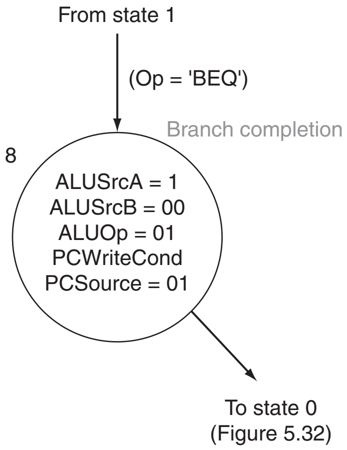

**FIGURE 5.35 The branch instruction requires a single state.** The first three outputs that are asserted cause the ALU to compare the registers (ALUSrcA, ALUSrcB, and ALUOp), while the signals PCSource and PCWriteCond perform the conditional write if the branch condition is true. Notice that we do not use the value written into ALUOut; instead, we use only the Zero output of the ALU. The branch target address is read from ALUOut, where it was saved at the end of state 1.

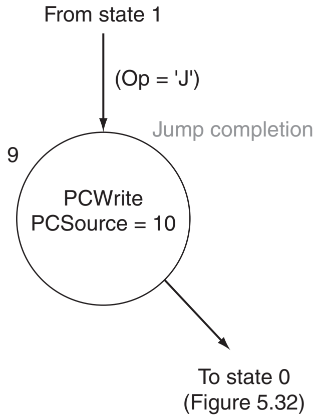

**FIGURE 5.36 The jump instruction requires a single state that asserts two control signals to write the PC with the lower 26 bits of the Instruction register shifted left 2 bits and concatenated to the upper 4 bits of the PC of this instruction.**

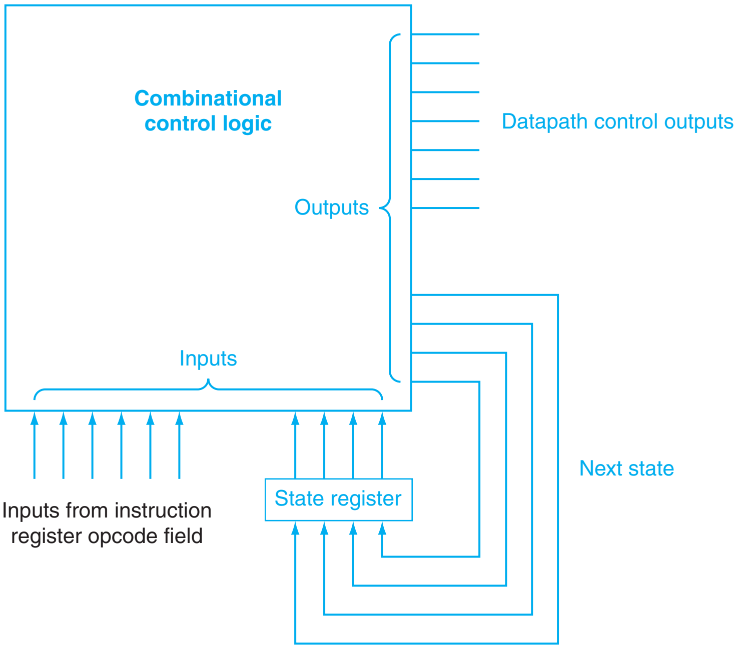
Combinational control logic
Datapath control outputs
Outputs
Inputs
Next state
State register
Inputs from instruction register opcode field

**FIGURE 5.37 Finite state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state.** The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational logic are the current state and any inputs used to determine the next state. In this case, the inputs are the instruction register opcode bits. Notice that in the finite state machine used in this chapter, the outputs depend only on the current state, not on the inputs. The Elaboration above explains this in more detail.

instructions are one universal example. There are many examples in the IA-32 architecture that require the use of multicycle control.

**Elaboration:** The style of finite state machine in Figure 5.37 is called a Moore machine, after Edward Moore. Its identifying characteristic is that the output depends only on the current state. For a Moore machine, the box labeled combinational control logic can be split into two pieces. One piece has the control output and only the state input, while the other has only the next-state output.

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

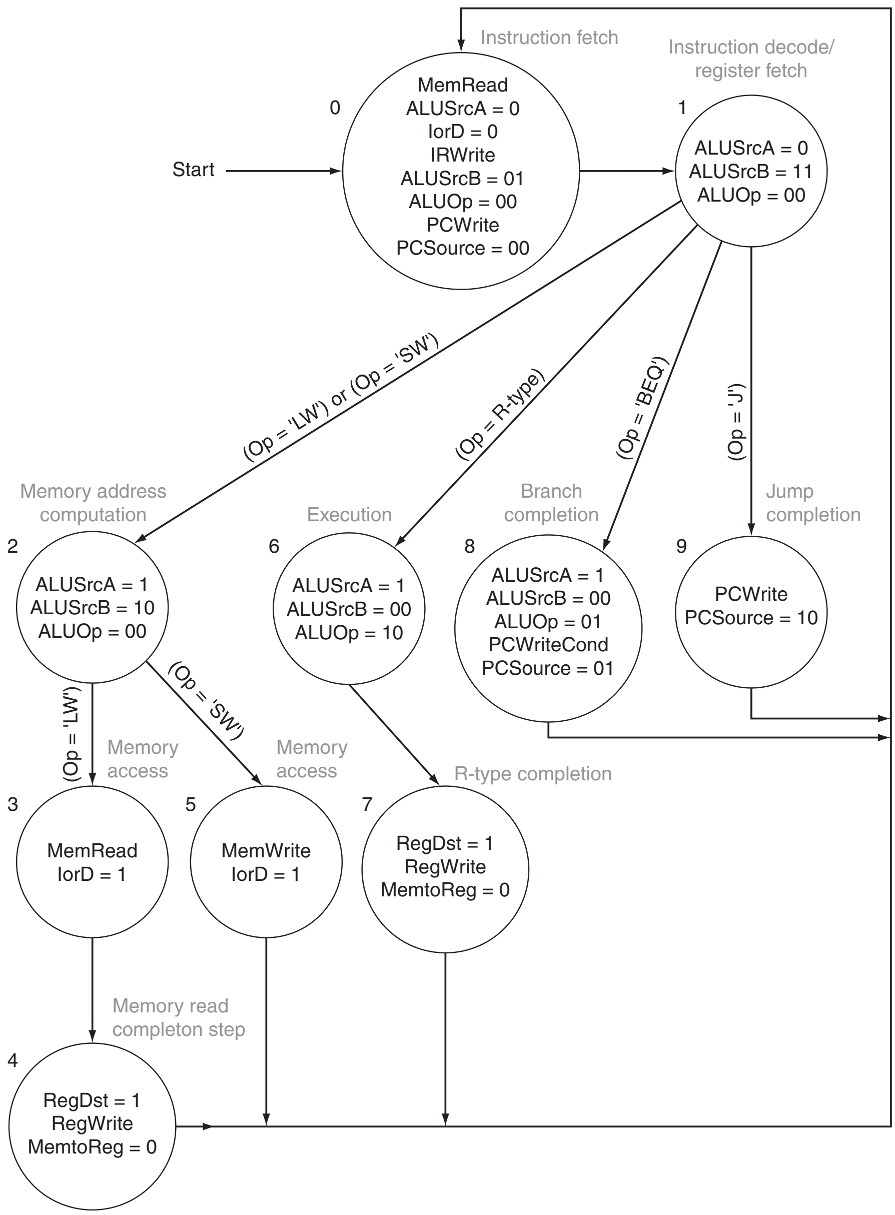

**FIGURE 5.38 The complete finite state machine control for the datapath shown in Figure 5.28.** The labels on the arcs are conditions that are tested to determine which state is the next state; when the next state is unconditional, no label is given. The labels inside the nodes indicate the output signals asserted during that state; we always specify the setting of a multiplexor control signal if the correct operation requires it. Hence, in some states a multiplexor control will be set to 0.

where there is a one-state difference between two sequences of states, the Mealy machine may unify the states by making the outputs depend on the inputs.

### Understanding Program Performance

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

### Check Yourself

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

## 5.6 Exceptions

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

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

Control is the most challenging aspect of processor design: it is both the hardest part to get right and the hardest part to make fast. One of the hardest parts of control is implementing **exceptions** and **interrupts**—events other than branches or jumps that change the normal flow of instruction execution. An exception is an unexpected event from within the processor; arithmetic overflow is an example of an exception. An interrupt is an event that also causes an unexpected change in control flow but comes from outside of the processor. Interrupts are used by I/O devices to communicate with the processor, as we will see in Chapter 8.

Many architectures and authors do not distinguish between interrupts and exceptions, often using the older name *interrupt* to refer to both types of events. We follow the MIPS convention, using the term *exception* to refer to *any* unexpected change in control flow without distinguishing whether the cause is internal