

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-

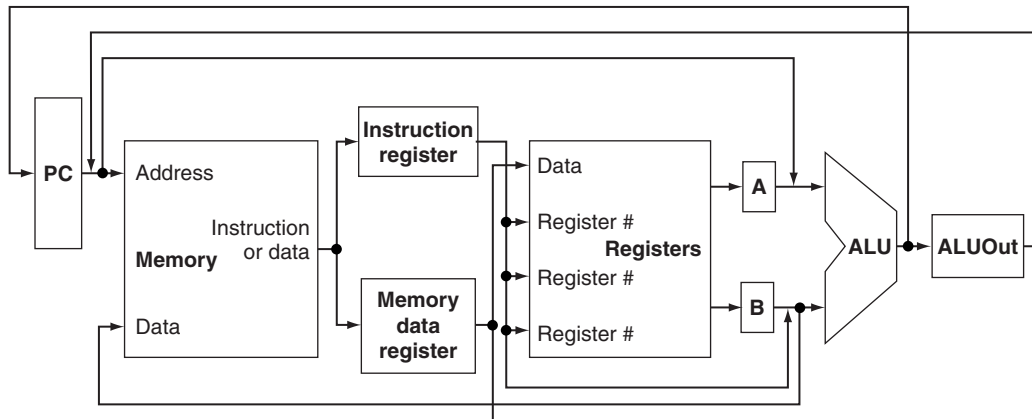


FIGURE 5.25 The high-level view of the multicycle datapath. This picture shows the key elements of the datapath: a shared memory unit, a single ALU shared among instructions, and the connections among these shared units. The use of shared functional units requires the addition or widening of multiplexors 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.

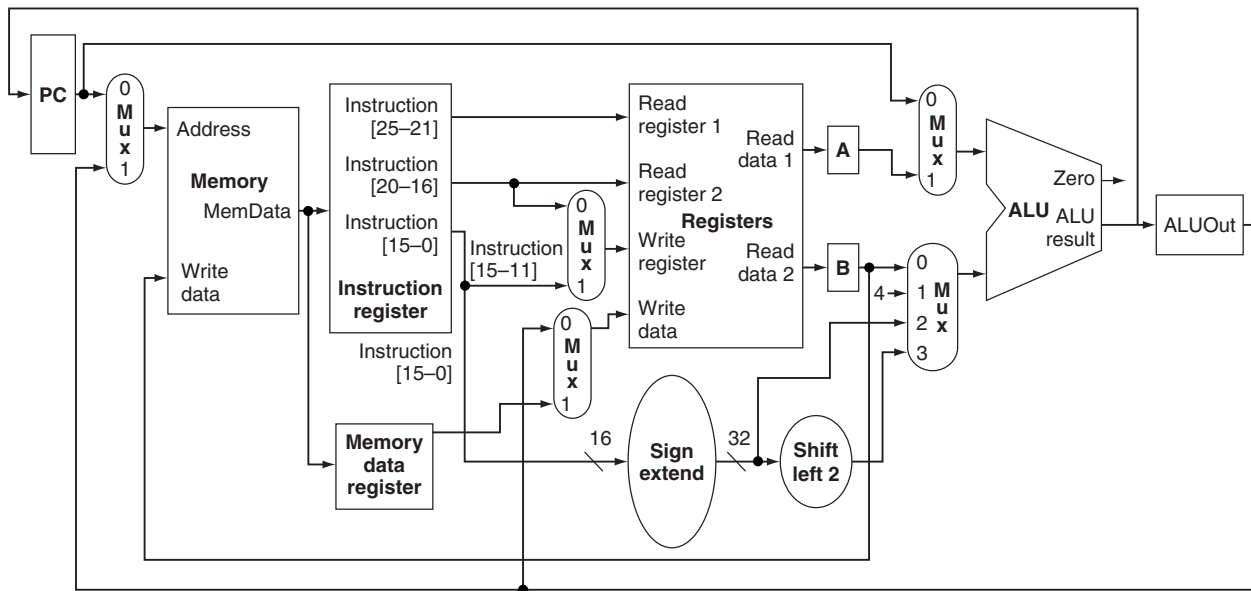


FIGURE 5.26 Multicycle datapath for MIPS handles the basic instructions. Although this datapath supports normal incrementing of the PC, a few more connections and a multiplexor will be needed for branches and jumps; we will add these shortly. The additions versus the single-clock datapath include several 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.

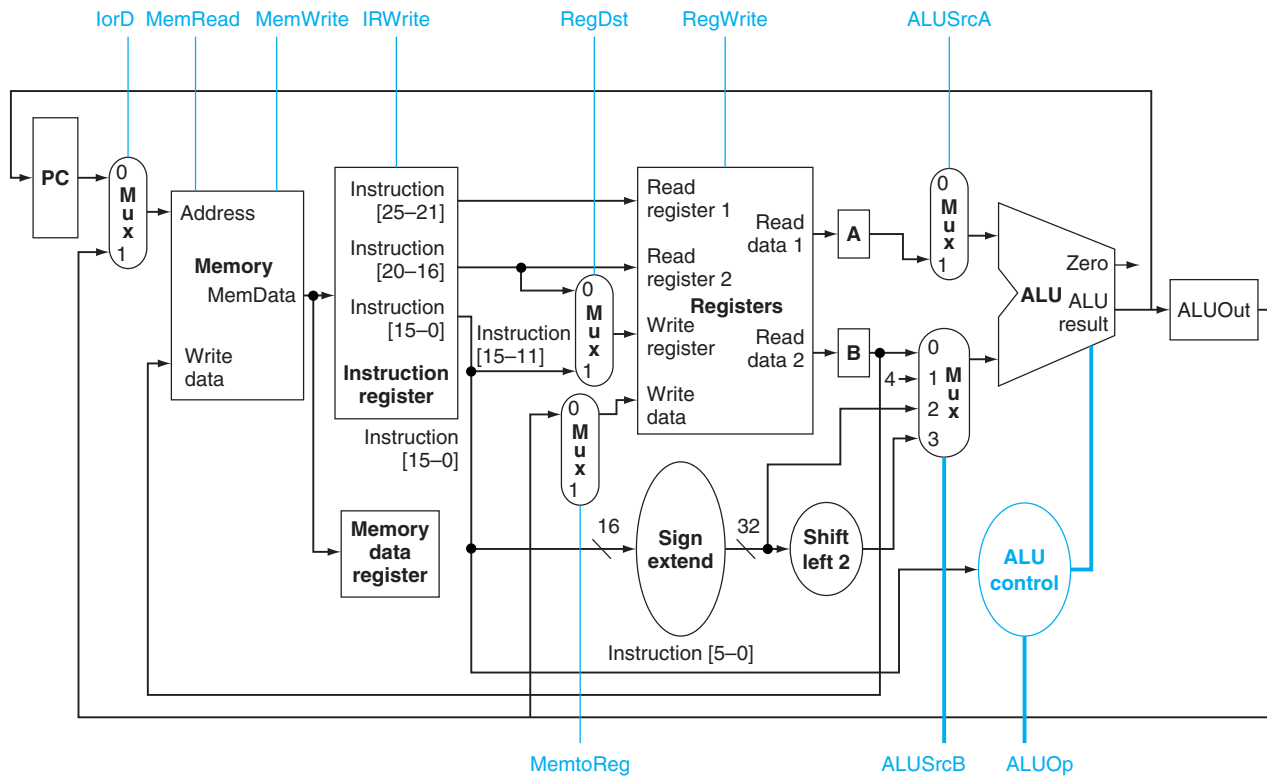


FIGURE 5.27 The multicycle datapath from Figure 5.26 with the control lines shown. The signals **ALUOp** and **ALUSrcB** are 2-bit control signals, while all the other control lines are 1-bit signals. Neither register **A** nor **B** requires a write signal, since their contents are only read on the 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

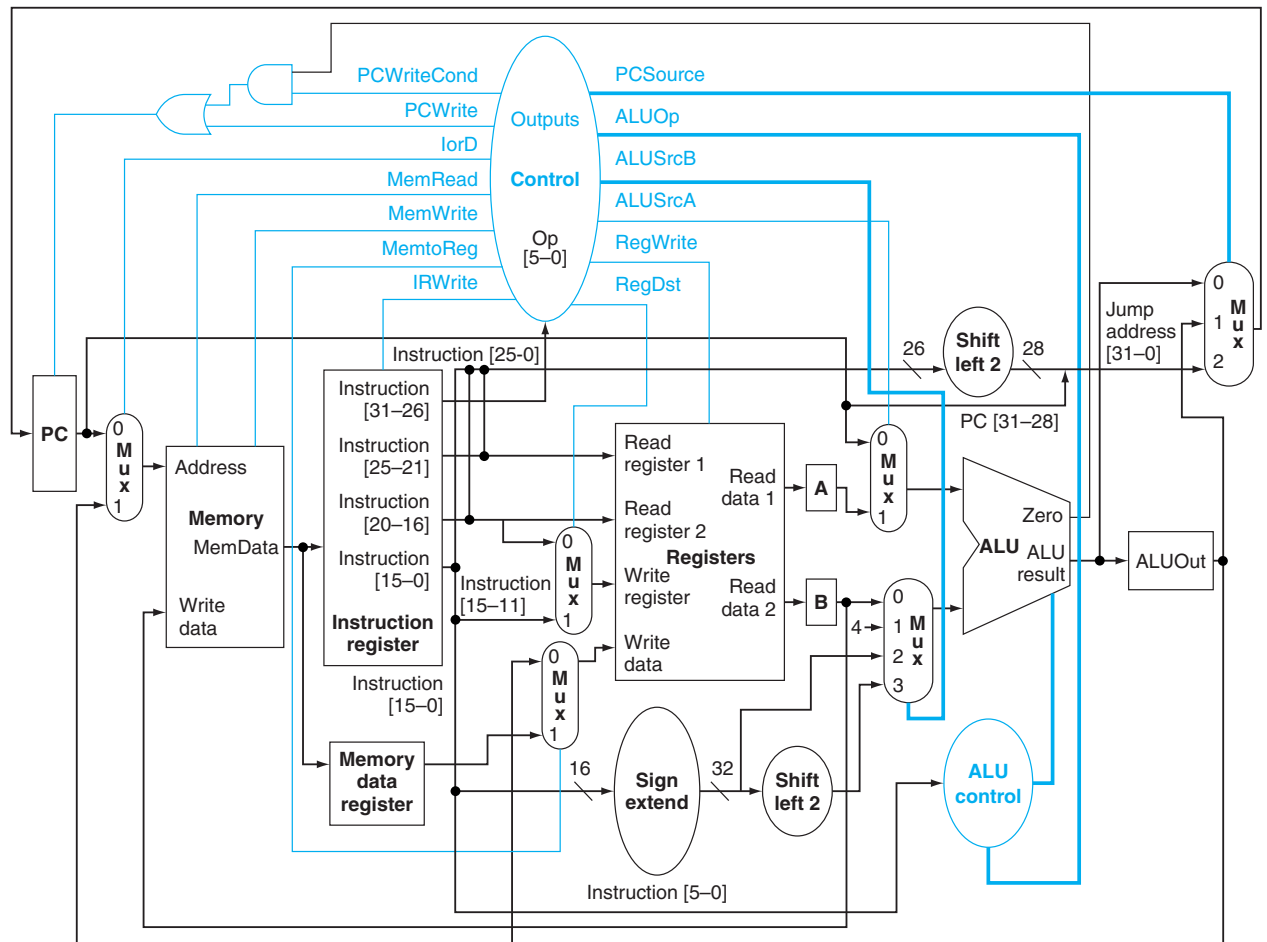


FIGURE 5.28 The complete datapath for the multicycle implementation together with the necessary control lines. The control lines of Figure 5.27 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure 5.27 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.
lorD	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 PCSrc.
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 PCSrc) 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 set-

ting 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):

$\text{Reg}[\text{IR}[15:11]] \leftarrow \text{ALUOut};$

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:

$\text{Reg}[\text{IR}[20:16]] \leftarrow \text{MDR};$

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	$\text{IR} \leftarrow \text{Memory}[\text{PC}]$ $\text{PC} \leftarrow \text{PC} + 4$			
Instruction decode/register fetch	$A \leftarrow \text{Reg}[\text{IR}[25:21]]$ $B \leftarrow \text{Reg}[\text{IR}[20:16]]$ $\text{ALUOut} \leftarrow \text{PC} + (\text{sign-extend}(\text{IR}[15:0]) \ll 2)$			
Execution, address computation, branch/jump completion	$\text{ALUOut} \leftarrow A \text{ op } B$	$\text{ALUOut} \leftarrow A + \text{sign-extend}(\text{IR}[15:0])$	if (A == B) $\text{PC} \leftarrow \text{ALUOut}$	$\text{PC} \leftarrow \{\text{PC}[31:28], (\text{IR}[25:0]), 2'b00\}$
Memory access or R-type completion	$\text{Reg}[\text{IR}[15:11]] \leftarrow \text{ALUOut}$	Load: $\text{MDR} \leftarrow \text{Memory}[\text{ALUOut}]$ or Store: $\text{Memory}[\text{ALUOut}] \leftarrow B$		
Memory read completion		Load: $\text{Reg}[\text{IR}[20:16]] \leftarrow \text{MDR}$		

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.

[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.

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

EXAMPLE

ANSWER

CPI in a Multicycle CPU

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?

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:

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

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

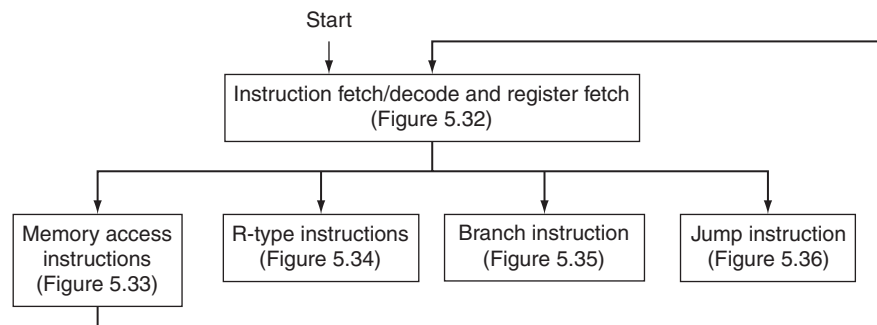


FIGURE 5.31 The high-level view of the finite state machine control. The first steps are independent of the instruction class; then a series of sequences that depend on the instruction opcode are used to complete each instruction class. After completing the actions needed for that instruction class, 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.

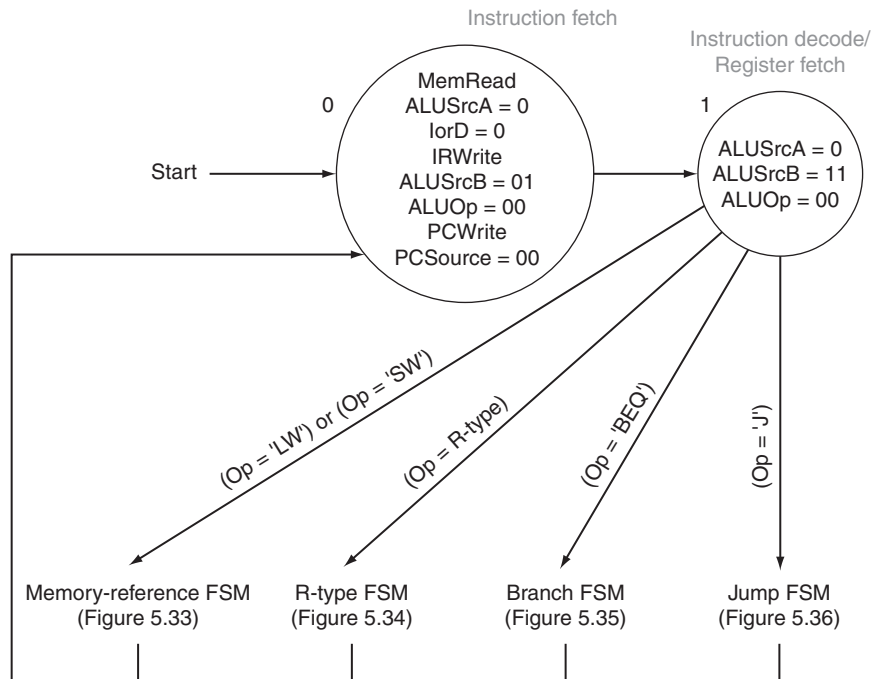


FIGURE 5.32 The instruction fetch and decode portion of every instruction is identical. These states correspond to the top box in the abstract finite state machine in Figure 5.31. In the first state we assert two signals to cause the memory to read an instruction and write it into the Instruction register (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.

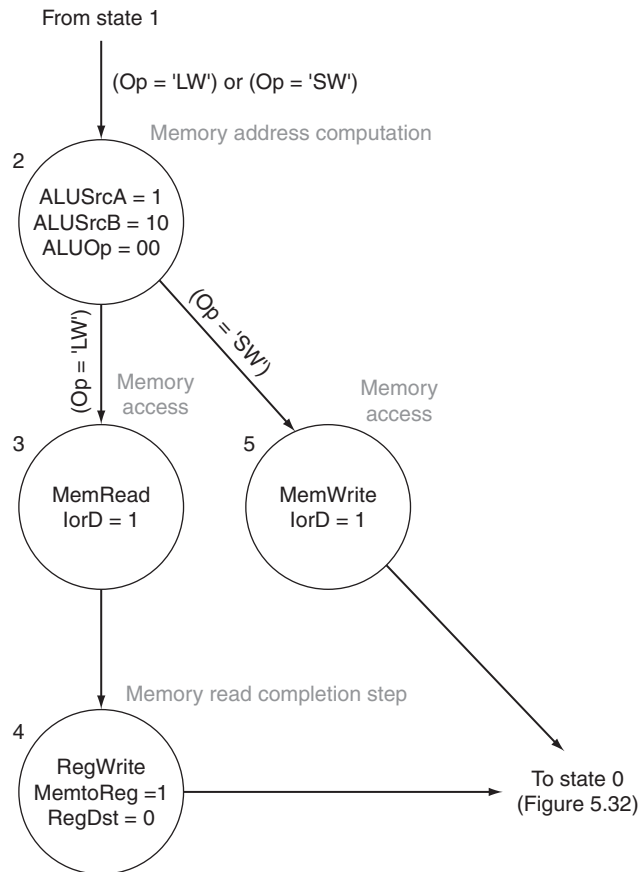


FIGURE 5.33 The finite state machine for controlling memory-reference instructions has four states. These states correspond to the box labeled “Memory access instructions” in Figure 5.31. After performing a memory address calculation, a separate sequence is needed for load and for store. The setting 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 (corre-

sponding 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

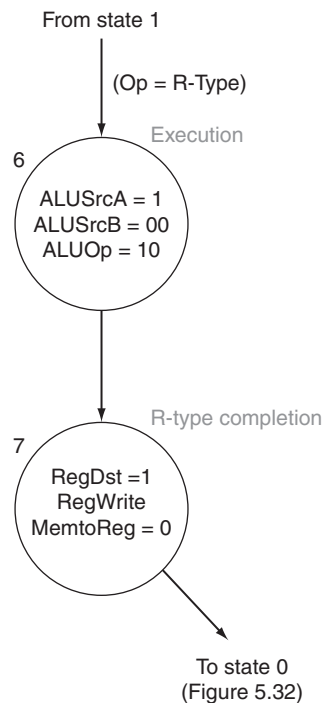


FIGURE 5.34 R-type instructions can be implemented with a simple two-state finite state machine. These states correspond to the box labeled “R-type instructions” in Figure 5.31. The first state causes the ALU operation to occur, while the second state causes the ALU result (which is in ALUOut) to be written in the register. 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 `00two` 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-

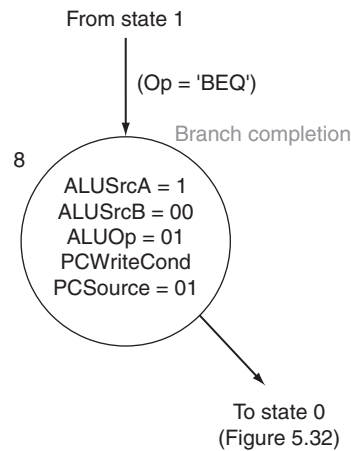


FIGURE 5.35 The branch instruction requires a single state. The first three outputs that are asserted cause the ALU to compare the registers (ALUSrcA, ALUSrcB, and ALUOp), while the signals PCSource and PCWriteCond perform the conditional write if the branch condition is true. Notice that we do not use 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.

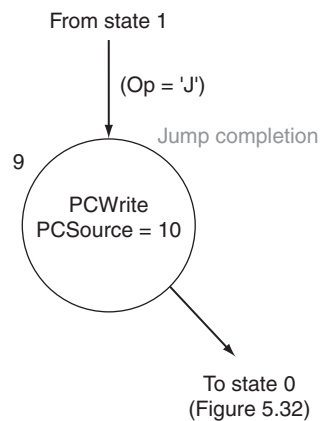


FIGURE 5.36 The jump instruction requires a single state that asserts two control signals to write the PC with the lower 26 bits of the Instruction register shifted left 2 bits and concatenated to the upper 4 bits of the PC of this instruction.

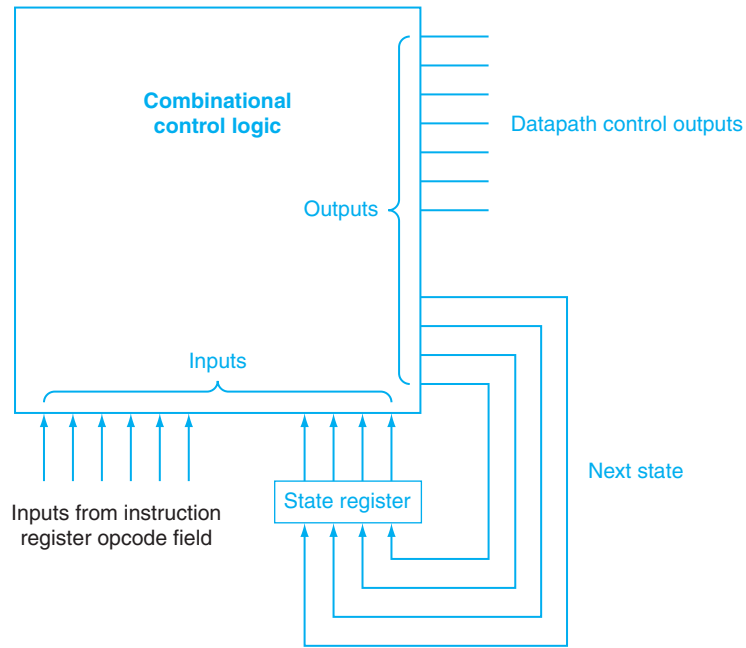


FIGURE 5.37 Finite state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state. The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational 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

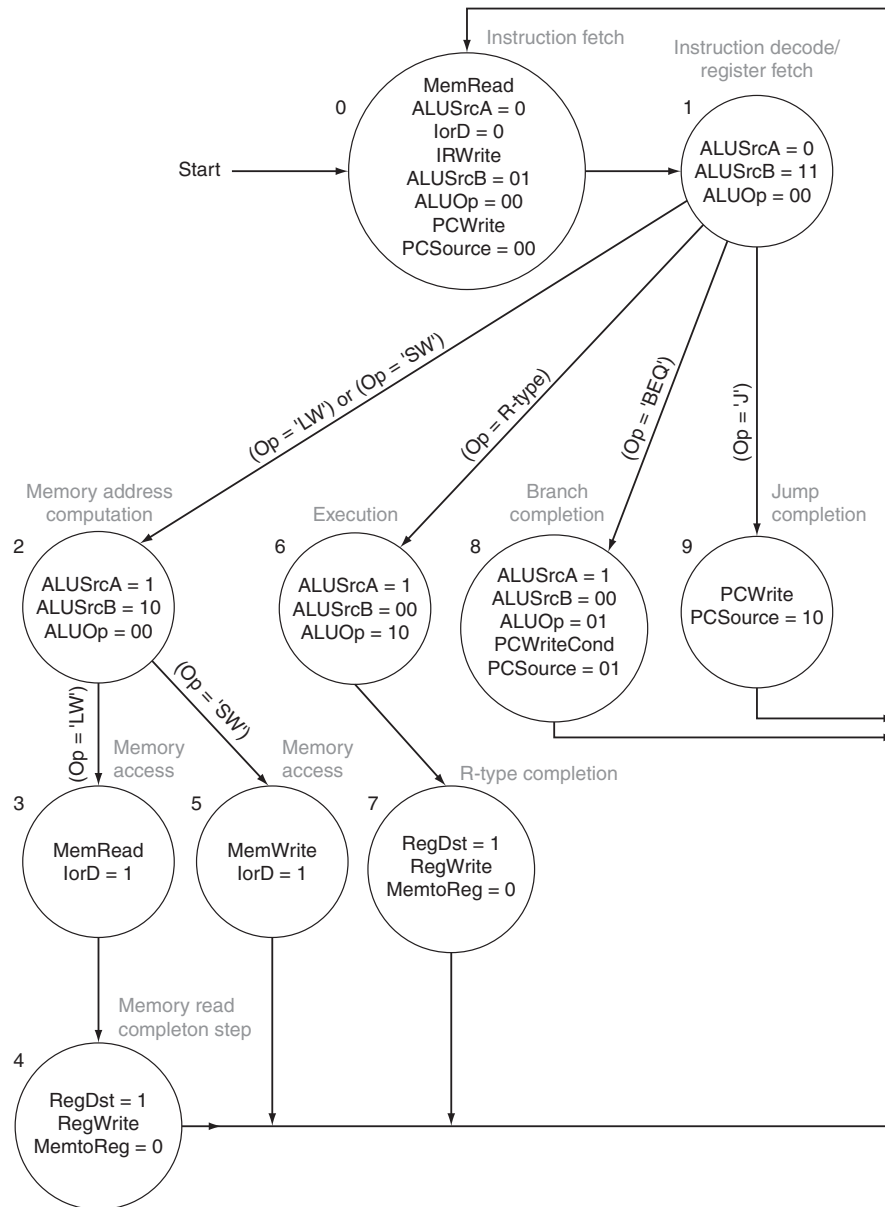


FIGURE 5.38 The complete finite state machine control for the datapath shown in Figure 5.28. The labels on the arcs are conditions that are tested to determine which state is the next state; when the next state is unconditional, no label is given. The labels inside the nodes indicate the output signals 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

or external; we use the term *interrupt* only when the event is externally caused. The Intel IA-32 architecture uses the word *interrupt* for all these events.

Interrupts were initially created to handle unexpected events like arithmetic overflow and to signal requests for service from I/O devices. The same basic mechanism was extended to handle internally generated exceptions as well. Here are some examples showing whether the situation is generated internally by the processor or externally generated:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Many of the requirements to support exceptions come from the specific situation that causes an exception to occur. Accordingly, we will return to this topic in Chapter 7, when we discuss memory hierarchies, and in Chapter 8, when we discuss I/O, and we better understand the motivation for additional capabilities in the exception mechanism. In this section, we deal with the control implementation for detecting two types of exceptions that arise from the portions of the instruction set and implementation that we have already discussed.

Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a machine, which determines the clock cycle time and thus performance. Without proper attention to exceptions during design of the control unit, attempts to add exceptions to a complicated implementation can significantly reduce performance, as well as complicate the task of getting the design correct.

How Exceptions Are Handled

The two types of exceptions that our current implementation can generate are execution of an undefined instruction and an arithmetic overflow. The basic action that the machine must perform when an exception occurs is to save the address of the offending instruction in the exception program counter (EPC) and then transfer control to the operating system at some specified address.

The operating system can then take the appropriate action, which may involve providing some service to the user program, taking some predefined action in response to an overflow, or stopping the execution of the program and reporting an error. After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution

of the program. In Chapter 7, we will look more closely at the issue of restarting the execution.

For the operating system to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it. There are two main methods used to communicate the reason for an exception. The method used in the MIPS architecture is to include a status register (called the *Cause register*), which holds a field that indicates the reason for the exception.

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

A second method is to use **vectored interrupts**. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception. For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:

Exception type	Exception vector address (in hex)
Undefined instruction	C000 0000 _{hex}
Arithmetic overflow	C000 0020 _{hex}

The operating system knows the reason for the exception by the address at which it is initiated. The addresses are separated by 32 bytes or 8 instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence. When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause.

We can perform the processing required for exceptions by adding a few extra registers and control signals to our basic implementation and by slightly extending the finite state machine. Let's assume that we are implementing the exception system used in the MIPS architecture. (Implementing vectored exceptions is no more difficult.) We will need to add two additional registers to the datapath:

- **EPC:** A 32-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)
- **Cause:** A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused. Assume that the low-order bit of this register encodes the two possible exception sources mentioned above: undefined instruction = 0 and arithmetic overflow = 1.

We will need to add two control signals to cause the EPC and Cause registers to be written; call these *EPCWrite* and *CauseWrite*. In addition, we will need a 1-bit control signal to set the low-order bit of the Cause register appropriately; call this signal *IntCause*. Finally, we will need to be able to write the *exception address*, which is the operating system entry point for exception handling, into the PC; in

the MIPS architecture, this address is $8000\ 0180_{\text{hex}}$. (The SPIM simulator for MIPS uses $8000\ 0080_{\text{hex}}$.) Currently, the PC is fed from the output of a three-way multiplexor, which is controlled by the signal PCSource (see Figure 5.28 on page 323). We can change this to a four-way multiplexor, with additional input wired to the constant value $8000\ 0180_{\text{hex}}$. Then PCSource can be set to 11_{two} to select this value to be written into the PC.

Because the PC is incremented during the first cycle of every instruction, we cannot just write the value of the PC into the EPC, since the value in the PC will be the instruction address plus 4. However, we can use the ALU to subtract 4 from the PC and write the output into the EPC. This requires no additional control signals or paths, since we can use the ALU to subtract, and the constant 4 is already a selectable ALU input. The data write port of the EPC, therefore, is connected to the ALU output. Figure 5.39 shows the multicycle datapath with these additions needed for implementing exceptions.

Using the datapath of Figure 5.39, the action to be taken for each different type of exception can be handled in one state apiece. In each case, the state sets the Cause register, computes and saves the original PC into the EPC, and writes the exception address into the PC. Thus, to handle the two exception types we are considering, we will need to add only the two states, but before we add them we must determine how to check for exceptions, since these checks will control the arcs to the new states.

How Control Checks for Exceptions

Now we have to design a method to detect these exceptions and to transfer control to the appropriate state in the exception states. Figure 5.40 shows the two new states (10 and 11) as well as their connection to the rest of the finite state control. Each of the two possible exceptions is detected differently:

- *Undefined instruction:* This is detected when no next state is defined from state 1 for the op value. We handle this exception by defining the next-state value for all op values other than $lw, sw, 0$ (R-type), j , and beq as state 10. We show this by symbolically using *other* to indicate that the op field does not match any of the opcodes that label arcs out of state 1 to the new state 10, which is used for this exception.
- *Arithmetic overflow:* The ALU, designed in [Appendix B](#), included logic to detect overflow, and a signal called *Overflow* is provided as an output from the ALU. This signal is used in the modified finite state machine to specify an additional possible next state (state 11) for state 7, as shown in Figure 5.40.

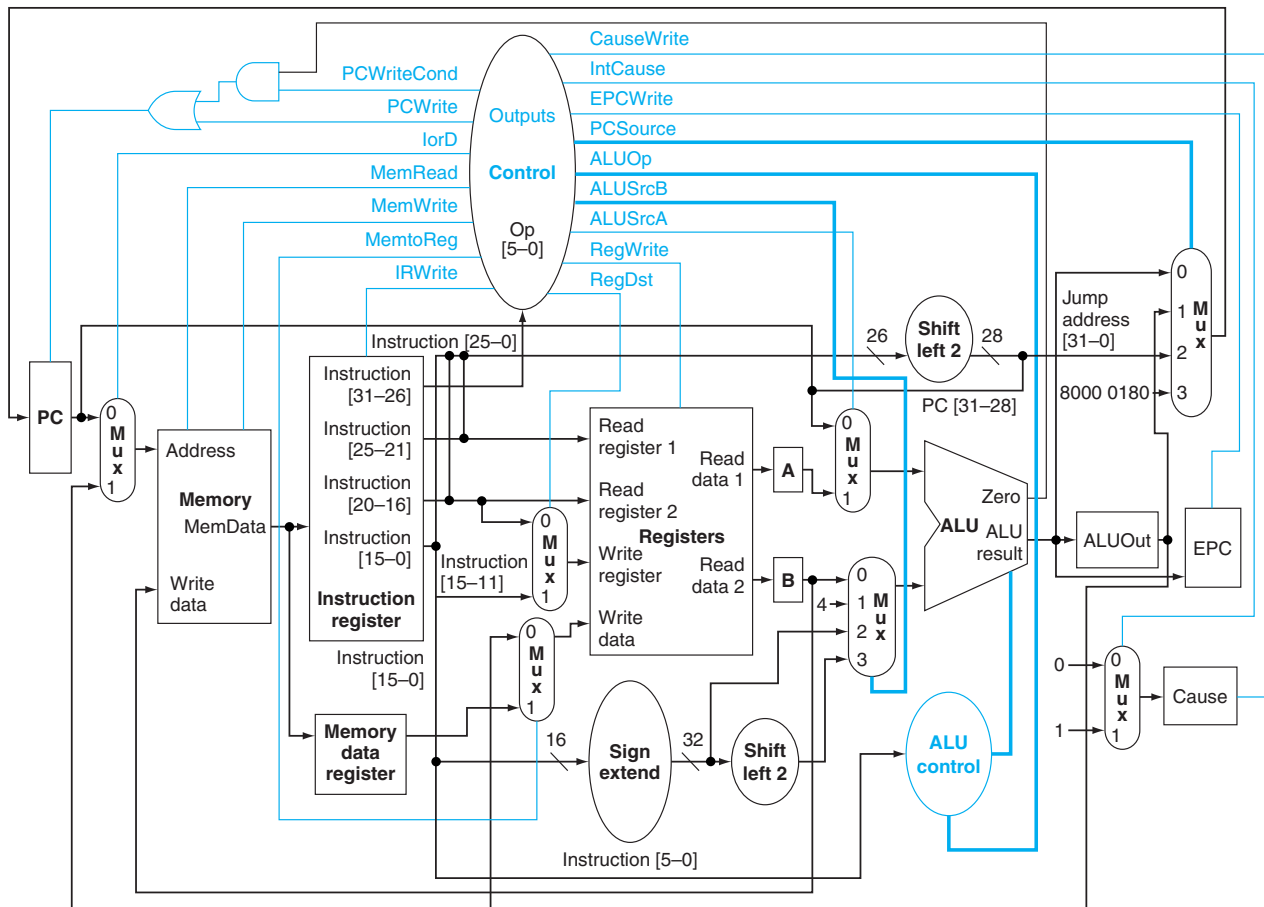


FIGURE 5.39 The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause and EPC registers, a multiplexor to control the value sent to the Cause register, an expansion of the multiplexor controlling the value written into the PC, and control lines for the added multiplexor and registers. For simplicity, this figure does not show the ALU overflow signal, which would need to be stored in a one-bit register and delivered as an additional input to the control unit (see Figure 5.40 to see how it is used).

Figure 5.40 represents a complete specification of the control for this MIPS subset with two types of exceptions. Remember that the challenge in designing the control of a real machine is to handle the variety of different interactions between instructions and other exception-causing events in such a way that the control logic remains both small and fast. The complex interactions that are possible are what make the control unit the most challenging aspect of hardware design.

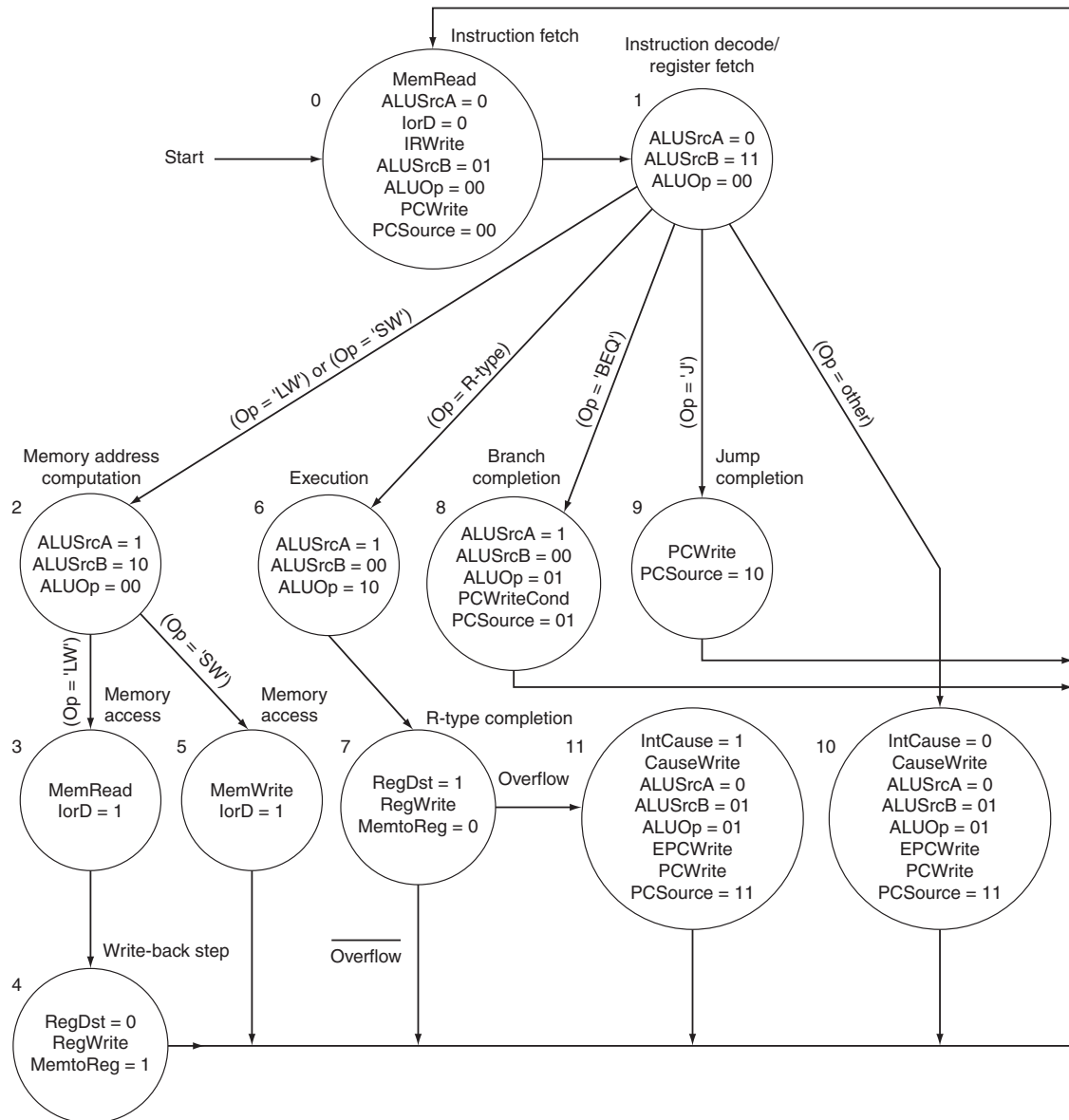


FIGURE 5.40 This shows the finite state machine with the additions to handle exception detection. States 10 and 11 are the new states that generate the appropriate control for exceptions. The branch out of state 1 labeled (*Op = other*) indicates the next state when the input does not match the opcode of any of lw, sw, 0 (R-type), j, or beq. The branch out of state 7 labeled *Overflow* indicates the action to be taken when the ALU signals an overflow.

Elaboration: If you examine the finite state machine in Figure 5.40 closely, you can see that some problems could occur in the way the exceptions are handled. For example, in the case of arithmetic overflow, the instruction causing the overflow completes writing its result because the overflow branch is in the state when the write completes. However, it's possible that the architecture defines the instruction as having no effect if the instruction causes an exception; this is what the MIPS instruction set architecture specifies. In Chapter 7, we will see that certain classes of exceptions require us to prevent the instruction from changing the machine state, and that this aspect of handling exceptions becomes complex and potentially limits performance.

Check Yourself

Is this optimization proposed in the Check Yourself on page 340 concerning PCSource still valid in the extended control for exceptions shown in Figure 5.40 on page 345? Why or why not?



Microprogramming: Simplifying Control Design

Microprogramming is a technique for designing complex control units. It uses a very simple hardware engine that can then be programmed to implement a more complex instruction set. Microprogramming is used today to implement some parts of a complex instruction set, such as a Pentium, as well as in special-purpose processors. This section, which appears on the CD, explains the basic concepts and shows how they can be used to implement the MIPS multicycle control.



An Introduction to Digital Design Using a Hardware Design Language

Modern digital design is done using hardware description languages and modern computer-aided synthesis tools that can create detailed hardware designs from the descriptions using both libraries and logic synthesis. Entire books are written on such languages and their use in digital design. This section, which appears on the CD, gives a brief introduction and shows how a hardware design language, Verilog in this case, can be used to describe the MIPS multicycle control both behaviorally and in a form suitable for hardware synthesis.

5.9**Real Stuff: The Organization of Recent Pentium Implementations**

The techniques described in this chapter for building datapaths and control units are at the heart of every computer. All recent computers, however, go beyond the techniques of this chapter and use pipelining. *Pipelining*, which is the subject of the next chapter, improves performance by overlapping the execution of multiple instructions, achieving throughput close to one instruction per clock cycle (like our single-cycle implementation) with a clock cycle time determined by the delay of individual functional units rather than the entire execution path of an instruction (like our multicycle design). The last Intel IA-32 processor without pipelining was the 80386, introduced in 1985; the very first MIPS processor, the R2000, also introduced in 1985, was pipelined.

Recent Intel IA-32 processors (the Pentium II, III, and 4) employ sophisticated pipelining approaches. These processors, however, are still faced with the challenge of implementing control for the complex IA-32 instruction set, described in Chapter 2. The basic functional units and datapaths in use in modern processors, while significantly more complex than those described in this chapter, have the same basic functionality and similar types of control signals. Thus the task of designing a control unit builds on the same principles used in this chapter.

Challenges Implementing More Complex Architectures

Unlike the MIPS architecture, the IA-32 architecture contains instructions that are very complex and can take tens, if not hundreds, of cycles to execute. For example, the string move instruction (MOVSB) requires calculating and updating two different memory addresses as well as loading and storing a byte of the string. The larger number and greater complexity of addressing modes in the IA-32 architecture complicates implementation of even simple instructions similar to those on MIPS. Fortunately, a multicycle datapath is well structured to adapt to variations in the amount of work required per instruction that are inherent in IA-32 instructions. This adaptability comes from two capabilities:

1. A multicycle datapath allows instructions to take varying numbers of clock cycles. Simple IA-32 instructions that are similar to those in the MIPS architecture can execute in 3 or 4 clock cycles, while more complex instructions can take tens of cycles.
2. A multicycle datapath can use the datapath components more than once per instruction. This is critical to handling more complex addressing modes, as well as implementing more complex operations, both of which are present in the IA-32 architecture. Without this capability, the datapath

microprogrammed control A method of specifying control that uses microcode rather than a finite state representation.

hardwired control An implementation of finite state machine control typically using programmable logic arrays (PLAs) or collections of PLAs and random logic.

microcode The set of microinstructions that control a processor.

superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle.

microinstruction A representation of control using low-level instructions, each of which asserts a set of control signals that are active on a given clock cycle as well as specifies what microinstruction to execute next.

micro-operations The RISC-like instructions directly executed by the hardware in recent Pentium implementations.

would need to be extended to handle the demands of the more complex instructions without reusing components, which would be completely impractical. For example, a single-cycle datapath, which doesn't reuse components, for the IA-32 would require several data memories and a very large number of ALUs.

Using the multicycle datapath and a **microprogrammed controller** provides a framework for implementing the IA-32 instruction set. The challenging task, however, is creating a high-performance implementation, which requires dealing with the diversity of the requirements arising from different instructions. Simply put, a high-performance implementation needs to ensure that the simple instructions execute quickly, and that the burden of the complexities of the instruction set penalize primarily the complex, less frequently used, instructions.

To accomplish this goal, every Intel implementation of the IA-32 architecture since the 486 has used a combination of **hardwired control** to handle simple instructions, and **microcoded** control to handle the more complex instructions. For those instructions that can be executed in a single pass through the datapath—those with complexity similar to a MIPS instruction—the hardwired control generates the control information and executes the instruction in one pass through the datapath that takes a small number of clock cycles. Those instructions that require multiple datapath passes and complex sequencing are handled by the microcoded controller that takes a larger number of cycles and multiple passes through the datapath to complete the execution of the instruction. The benefit of this approach is that it enables the designer to achieve low cycle counts for the simple instructions without having to build the enormously complex datapath that would be required to handle the full generality of the most complex instructions.

The Structure of the Pentium 4 Implementation

Recent Pentium processors are capable of executing more than one instruction per clock, using an advanced pipelining technique, called **superscalar**. We describe how a superscalar processor works in the next chapter. The important thing to understand here is that executing more than one instruction per clock requires duplicating the datapath resources. The simplest way to think about this is that the processor has multiple datapaths, although these are tailored to handle one class of instructions: say, loads and stores, ALU operations, or branches. In this way, the processor is able to execute a load or store in the same clock cycle that it is also executing a branch and an ALU operation. The Pentium III and 4 allow up to three IA-32 instructions to execute in a clock cycle.

The Pentium III and Pentium 4 execute simple **microinstructions** similar to MIPS instructions, called **micro-operations** in Intel terminology. These microinstructions are fully self-contained operations that are initially about 70 bits wide. The control of datapath to implement these microinstructions is completely hard-

wired. This last level of control expands up to three microinstructions into about 120 control lines for the integer datapaths and 275 to over 400 control lines for the floating-point datapath—the latter number for the new SSE2 instructions included in the Pentium 4. This last step of expanding the microinstructions into control lines is very similar to the control generation for the single-cycle datapath or for the ALU control.

How is the translation between IA-32 instructions and microinstructions performed? In earlier Pentium implementations (i.e., the Pentium Pro, Pentium II, and Pentium III), the instruction decode unit would look at up to three IA-32 instructions at a time and use a set of PLAs to generate up to six microinstructions per cycle. With the significantly higher clock rate introduced in the Pentium 4, this solution was no longer adequate and an entirely new method of generating microinstructions was needed.

The solution adopted in the Pentium 4 is to include a **trace cache** of microinstructions, which is accessed by the IA-32 program counter. A trace cache is a sophisticated form of instruction cache, which we explain in detail in Chapter 7. For now, think of it as a buffer that holds the microinstructions that implement a given IA-32 instruction. When the trace cache is accessed with the address of the next IA-32 instruction to be executed, one of several events occurs:

- The translation of the IA-32 instruction is in the trace cache. In this case, up to three microinstructions are produced from the trace cache. These three microinstructions represent from one to three IA-32 instructions. The IA-32 PC is advanced one to three instructions depending on how many fit in the three microinstruction sequence.
- The translation of the IA-32 instruction is in the trace cache, but it requires more than four microinstructions to implement. For such complex IA-32 instructions, there is a microcode ROM; the control unit transfers control to the microprogram residing in the ROM. Microinstructions are produced from the microprogram until the more complex IA-32 instruction has been completed. The microcode ROM provides a total of more than 8000 microinstructions, with a number of sequences being shared among IA-32 instructions. Control then transfers back to fetching instructions from the trace cache.
- The translation of the designated IA-32 instruction is not in the trace cache. In this case, an IA-32 instruction decoder is used to decode the IA-32 instruction. If the number of microinstructions is four or less, the decoded microinstructions are placed in the trace cache, where they may be found on the next execution of this instruction. Otherwise, the microcode ROM is used to complete the sequence.

From one to three microinstructions are sent from the trace cache to the Pentium 4 microinstruction pipeline, which we describe in detail at the end of Chapter 6. The use of simple low-level hardwired control and simple datapaths for

trace cache An instruction cache that holds a sequence of instructions with a given starting address; in recent Pentium implementations the trace cache holds microoperations rather than IA-32 instructions.

handling the microinstructions together with the trace cache of decoded instructions allows the Pentium 4 to achieve impressive clock rates, similar to those for microprocessors implementing simpler instruction set architectures. Furthermore, the translation process, which combines direct hardwired control for simple instructions with microcoded control for complex instructions, allows the Pentium 4 to execute the simple, high-frequency instructions in the IA-32 instruction set at a high rate, yielding a low, and very competitive, CPI.

Understanding Program Performance

dispatch An operation in a microprogrammed control unit in which the next microinstruction is selected on the basis of one or more fields of a macroinstruction, usually by creating a table containing the addresses of the target microinstructions and indexing the table using a field of the macroinstruction. The dispatch tables are typically implemented in ROM or programmable logic array (PLA). The term *dispatch* is also used in dynamically scheduled processors to refer to the process of sending an instruction to a queue.

Although most of the Pentium 4 performance, ignoring the memory system, depends on the efficiency of the pipelined microoperations, the effectiveness of the front end in decoding IA-32 instructions can have a significant effect on performance. In particular, because of the structure of the decoder, using simpler IA-32 instructions that require four or fewer microoperations, and hence, avoiding a microcode **dispatch**, is likely to lead to better performance. Because of this implementation strategy (and a similar one on the Pentium III), compiler writers and assembly language programmers should try to make use of sequences of simple IA-32 instructions rather than more complex alternatives.

5.10 Fallacies and Pitfalls

Pitfall: Adding a complex instruction implemented with microprogramming may not be faster than a sequence using simpler instructions.

Most machines with a large and complex instruction set are implemented, at least in part, using microcode stored in ROM. Surprisingly, on such machines, sequences of individual simpler instructions are sometimes as fast as or even faster than the custom microcode sequence for a particular instruction.

How can this possibly be true? At one time, microcode had the advantage of being fetched from a much faster memory than instructions in the program. Since caches came into use in 1968, microcode no longer has such a consistent edge in fetch time. Microcode does, however, still have the advantage of using internal temporary registers in the computation, which can be helpful on machines with few general-purpose registers. The disadvantage of microcode is that the algorithms must be selected before the machine is announced and can't be

changed until the next model of the architecture. The instructions in a program, on the other hand, can utilize improvements in its algorithms at any time during the life of the machine. Along the same lines, the microcode sequence is probably not optimal for all possible combinations of operands.

One example of such an instruction in the IA-32 implementations is the move string instruction (MOVS) used with a repeat prefix that we discussed in Chapter 2. This instruction is often slower than a loop that moves words at a time, as we saw earlier in the Fallacies and Pitfalls (see page 350).

Another example involves the LOOP instruction, which decrements a register and branches to the specified label if the decremented register is not equal to zero. This instruction is designed to be used as the branch at the bottom of loops that have a fixed number of iterations (e.g., many *for* loops). Such an instruction, in addition to packing in some extra work, has benefits in minimizing the potential losses from the branch in pipelined machines (as we will see when we discuss branches in the next chapter).

Unfortunately, on all recent Intel IA-32 implementations, the LOOP instruction is always slower than the macrocode sequence consisting of simpler individual instructions (assuming that the small code size difference is not a factor). Thus, optimizing compilers focusing on speed never generate the LOOP instruction. This, in turn, makes it hard to motivate making LOOP fast in future implementations, since it is so rarely used!

Fallacy: If there is space in the control store, new instructions are free of cost.

One of the benefits of a microprogrammed approach is that control store implemented in ROM is not very expensive, and as transistor budgets grew, extra ROM was practically free. The analogy here is that of building a house and discovering, near completion, that you have enough land and materials left to add a room. This room wouldn't be free, however, since there would be the costs of labor and maintenance for the life of the home. The temptation to add "free" instructions can occur only when the instruction set is not fixed, as is likely to be the case in the first model of a computer. Because upward compatibility of binary programs is a highly desirable feature, all future models of this machine will be forced to include these so-called free instructions, even if space is later at a premium.

During the design of the 80286, many instructions were added to the instruction set. The availability of more silicon resource and the use of microprogrammed implementation made such additions seem painless. Possibly the largest addition was a sophisticated protection mechanism, which is largely unused today, but still must be implemented in newer implementations. This addition was motivated by a perceived need for such a mechanism and the desire to enhance microprocessor architectures to provide functionality equal to that of

larger computers. Likewise, a number of decimal instructions were added to provide decimal arithmetic on bytes. Such instructions are rarely used today because using binary arithmetic on 32 bits and converting back and forth to decimal representation is considerably faster. Like the protection mechanisms, the decimal instructions must be implemented in newer processors even if only rarely used.

5.11 Concluding Remarks

As we have seen in this chapter, both the datapath and control for a processor can be designed starting with the instruction set architecture and an understanding of the basic characteristics of the technology. In Section 5.3, we saw how the datapath for a MIPS processor could be constructed based on the architecture and the decision to build a single-cycle implementation. Of course, the underlying technology also affects many design decisions by dictating what components can be used in the datapath, as well as whether a single-cycle implementation even makes sense. Along the same lines, in the first portion of Section 5.5, we saw how the decision to break the clock cycle into a series of steps led to the revised multicycle datapath. In both cases, the top-level organization—a single-cycle or multicycle machine—together with the instruction set, prescribed many characteristics of the datapath design.

The BIG Picture

Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique. Figure 5.41 shows the variety of methods for specifying the control and moving from the specification to an implementation using some form of structured logic.

Similarly, the control is largely defined by the instruction set architecture, the organization, and the datapath design. In the single-cycle organization, these three aspects essentially define how the control signals must be set. In the multicycle design, the exact decomposition of the instruction execution into cycles, which is based on the instruction set architecture, together with the datapath, defines the requirements on the control.

Control is one of the most challenging aspects of computer design. A major reason is that designing the control requires an understanding of how all the com-

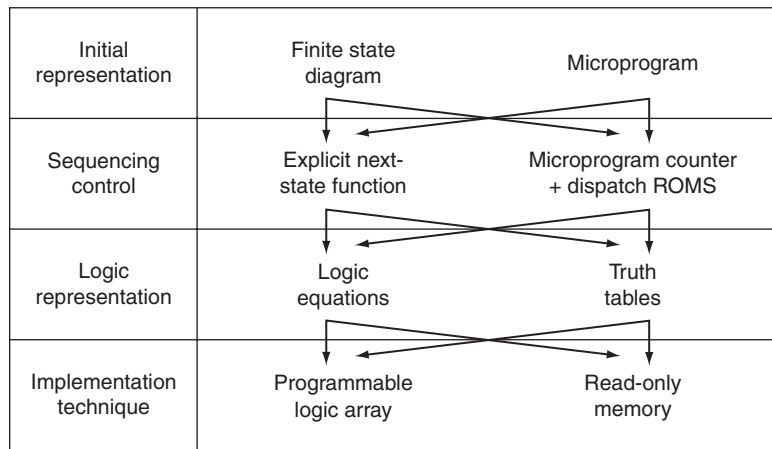


FIGURE 5.41 Alternative methods for specifying and implementing control. The arrows indicate possible design paths: any path from the initial representation to the final implementation technology is viable. Traditionally, “hardwired control” means that the techniques on the left-hand side are used, and “microprogrammed control” means that the techniques on the right-hand side are used.

ponents in the processor operate. To help meet this challenge, we examined two techniques for specifying control: finite state diagrams and microprogramming. These control representations allow us to abstract the specification of the control from the details of how to implement it. Using abstraction in this fashion is the major method we have to cope with the complexity of computer designs.

Once the control has been specified, we can map it to detailed hardware. The exact details of the control implementation will depend on both the structure of the control and on the underlying technology used to implement it. Abstracting the specification of control is also valuable because the decisions of how to implement the control are technology dependent and likely to change over time.



Historical Perspective and Further Reading

The rise of microprogramming and its effect on instruction set design and computer development is one of the more interesting interactions in the first few decades of the electronic computer. This story is the focus of the historical perspectives section on the CD.

5.13 Exercises

5.1 [6] <§5.2> Do we need combinational logic, sequential logic, or a combination of the two to implement each of the following:

- a. multiplexor
- b. comparator
- c. incrementer/decrementer
- d. barrel shifter
- e. multiplier with shifters and adders
- f. register
- g. memory
- h. ALU (the ones in single-cycle and multiple-cycle datapaths)
- i. carry look-ahead adder
- j. latch
- k. general finite state machine (FSM)

5.2 [10] <§5.4> Describe the effect that a single stuck-at-0 fault (i.e., regardless of what it should be, the signal is always 0) would have for the signals shown below, in the single-cycle datapath in Figure 5.17 on page 307. Which instructions, if any, will not work correctly? Explain why.

Consider each of the following faults separately:

- a. RegWrite = 0
- b. ALUOp0 = 0
- c. ALUOp1 = 0
- d. Branch = 0
- e. MemRead = 0
- f. MemWrite = 0

5.3 [5] <§5.4> This exercise is similar to Exercise 5.2, but this time consider stuck-at-1 faults (the signal is always 1).

5.4 [5] <§5.4>  [For More Practice](#): Single Cycle Datapaths with Floating Point.

5.5 [5] <§5.4>  [For More Practice](#): Single Cycle Datapaths with Floating Point.

5.6 [10] <§5.4>  [For More Practice](#): Single Cycle Datapaths with Floating Point.

5.7 [2–3 months] <§5.1–5.4> Using standard parts, build a machine that implements the single-cycle machine in this chapter.

5.8 [15] <§5.4> We wish to add the instruction `jr` (jump register) to the single-cycle datapath described in this chapter. Add any necessary datapaths and control signals to the single-cycle datapath of Figure 5.17 on page 307 and show the necessary additions to Figure 5.18 on page 308. You can photocopy these figures to make it faster to show the additions.

5.9 [10] <§5.4> This question is similar to Exercise 5.8 except that we wish to add the instruction `sll` (shift left logical), which is described in Section 2.5.

5.10 [15] <§5.4> This question is similar to Exercise 5.8 except that we wish to add the instruction `lui` (load upper immediate), which is described in Section 2.9.

5.11 [20] <§5.4> This question is similar to Exercise 5.8 except that we wish to add a variant of the `lw` (load word) instruction, which increments the index register after loading word from memory. This instruction (`lw_inc`) corresponds to the following two instructions:

```
lw    $rs, L($rt)
addi  $rt, $rt, 1
```

5.12 [5] <§5.4> Explain why it is not possible to modify the single-cycle implementation to implement the load with increment instruction described in Exercise 5.12 without modifying the register file.

5.13 [7] <§5.4> Consider the single-cycle datapath in Figure 5.17. A friend is proposing to modify this single-cycle datapath by eliminating the control signal `MemtoReg`. The multiplexor that has `MemtoReg` as an input will instead use either the `ALUSrc` or the `MemRead` control signal. Will your friend's modification work? Can one of the two signals (`MemRead` and `ALUSrc`) substitute for the other? Explain.

5.14 [10] <§5.4> MIPS chooses to simplify the structure of its instructions. The way we implement complex instructions through the use of MIPS instructions is to decompose such complex instructions into multiple simpler MIPS ones. Show how MIPS can implement the instruction `swap $rs, $rt`, which swaps the contents of registers `$rs` and `$rt`. Consider the case in which there is an available register that may be destroyed as well as the case in which no such register exists.

If the implementation of this instruction in hardware will increase the clock period of a single-instruction implementation by 10%, what percentage of swap operations in the instruction mix would recommend implementing it in hardware?

5.15 [5] <§5.4>  **For More Practice:** Effects of Faults in Control Multiplexors

5.16 [5] <\$5.4>  **For More Practice:** Effects of Faults in Control Multiplexors

5.17 [5] <\$5.5>  **For More Practice:** Effects of Faults in Control Multiplexors

5.18 [5] <\$5.5>  **For More Practice:** Effects of Faults in Control Multiplexors

5.19 [15] <\$5.4>  **For More Practice:** Adding Instructions to the Datapath

5.20 [15] <\$5.4>  **For More Practice:** Adding Instructions to the Datapath

5.21 [8] <\$5.4>  **For More Practice:** Adding Instructions to the Datapath

5.22 [8] <\$5.4>  **For More Practice:** Adding Instructions to the Datapath

5.23 [5] <\$5.4>  **For More Practice:** Adding Instructions to the Datapath

5.24 [10] <\$5.4>  **For More Practice:** Datapath Control Signals

5.25 [10] <\$5.4>  **For More Practice:** Datapath Control Signals

5.26 [15] <\$5.4>  **For More Practice:** Modifying the Datapath and Control

5.27 [8] <\$5.4> Repeat Exercise 5.14, but apply your solution to the instruction load with increment: `l_incr $rt, Address($rs)`.

5.28 [5] <\$5.4> The concept of the “critical path,” the longest possible path in the machine, was introduced in 5.4 on page 315. Based on your understanding of the single-cycle implementation, show which units can tolerate more delays (i.e., are not on the critical path), and which units can benefit from hardware optimization. Quantify your answers taking the same numbers presented on page 315 (Section 5.4, “Example: Performance of Single-Cycle Machines”).

5.29 [5] <\$5.5> This exercise is similar to Exercise 5.2, but this time consider the effect that the stuck-at-0 faults would have on the *multiple-cycle* datapath in Figure 5.27. Consider each of the following faults:

- a. `RegWrite = 0`
- b. `MemRead = 0`
- c. `MemWrite = 0`
- d. `IRWrite = 0`
- e. `PCWrite = 0`
- f. `PCWriteCond = 0`.

5.30 [5] <\$5.5> This exercise is similar to Exercise 5.29, but this time consider stuck-at-1 faults (the signal is always 1).

5.31 [[15] <\$5.4, 5.5> This exercise is similar to Exercise 5.13 but more general. Determine whether any of the control signals in the single-cycle implementation

can be eliminated and replaced by another existing control signal, or its inverse. Note that such redundancy is there because we have a very small set of instructions at this point, and it will disappear (or be harder to find) when we implement a larger number of instructions.

5.32 [15] <\$5.5> We wish to add the instruction `lui` (load upper immediate) described in Chapter 3 to the multicycle datapath described in this chapter. Use the same structure of the multicycle datapath of Figure 5.28 on page 323 and show the necessary modifications to the finite state machine of Figure 5.38 on page 339. You may find it helpful to examine the execution steps shown on pages 325 through 329 and consider the steps that will need to be performed to execute the new instruction. How many cycles are required to implement this instruction?

5.33 [15] <\$5.5> You are asked to modify the implementation of `lui` in Exercise 5.32 in order to cut the execution time by 1 cycle. Add any necessary datapaths and control signals to the multicycle datapath of Figure 5.28 on page 323. You can photocopy existing figures to make it easier to show your modifications. You have to maintain the assumption that you don't know what the instruction is before the end of state 1 (end of second cycle). Please explicitly state how many cycles it takes to execute the new instruction on your modified datapath and finite state machine.

5.34 [20] <\$5.5> This question is similar to Exercise 5.32 except that we wish to implement a new instruction `ldi` (load immediate) that loads a 32-bit immediate value from the memory location following the instruction address.

5.35 [15] <\$5.5> Consider a change to the multiple-cycle implementation that alters the register file so that it has only one read port. Describe (via a diagram) any additional changes that will need to be made to the datapath in order to support this modification. Modify the finite state machine to indicate how the instructions will work, given your new datapath.

5.36 [15] <\$5.5> Two important parameters control the performance of a processor: cycle time and cycles per instruction. There is an enduring trade-off between these two parameters in the design process of microprocessors. While some designers prefer to increase the processor frequency at the expense of large CPI, other designers follow a different school of thought in which reducing the CPI comes at the expense of lower processor frequency.

Consider the following machines, and compare their performance using the SPEC CPUint 2000 data from Figure 3.26 on page 228.

M1: The multicycle datapath of Chapter 5 with a 1 GHz clock.

M2: A machine like the multicycle datapath of Chapter 5, except that register updates are done in the same clock cycle as a memory read or ALU operation.

Thus in Figure 5.38 on page 339, states 6 and 7 and states 3 and 4 are combined. This machine has an 3.2 GHz clock, since the register update increases the length of the critical path.

M3: A machine like M2 except that effective address calculations are done in the same clock cycle as a memory access. Thus states 2, 3, and 4 can be combined, as can 2 and 5, as well as 6 and 7. This machine has a 2.8 GHz clock because of the long cycle created by combining address calculation and memory access.

Find out which of the machines is fastest. Are there instruction mixes that would make another machine faster, and if so, what are they?










5.37 [20] <\$5.5> Your friends at C³ (Creative Computer Corporation) have determined that the critical path that sets the clock cycle length of the multicycle datapath is memory access for loads and stores (not for fetching instructions). This has caused their newest implementation of the MIPS 30000 to run at a clock rate of 4.8 GHz rather than the target clock rate of 5.6 GHz. However, Clara at C³ has a solution. If all the cycles that access memory are broken into two clock cycles, then the machine can run at its target clock rate.

Using the SPEC CPUint 2000 mixes shown in Chapter 3 (Figure 3.26 on page 228), determine how much faster the machine with the two-cycle memory accesses is compared with the 4.8 GHz machine with single-cycle memory access. Assume that all jumps and branches take the same number of cycles and that the set instructions and arithmetic immediate instructions are implemented as R-type instructions. Would you consider the further step of splitting instruction fetch into two cycles if it would raise the clock rate up to 6.4 GHz? Why?

5.38 [20] <\$5.5> Suppose there were a MIPS instruction, called `bcmp`, that compares two blocks of words in two memory addresses. Assume that this instruction requires that the starting address of the first block is in register `$t1` and the starting address of the second block is in `$t2`, and that the number of words to compare is in `$t3` (which is `$t3 ≥ 0`). Assume the instruction can leave the result (the address of the first mismatch or zero if a complete match) in `$t1` and/or `$t2`. Furthermore, assume that the values of these registers as well as registers `$t4` and `t5` can be destroyed in executing this instruction (so that the registers can be used as temporaries to execute the instruction).

Write the MIPS assembly language program to implement (emulate the behavior of) block compare. How many instructions will be executed to compare two 100-word blocks? Using the CPI of the instructions in the multicycle implementation, how many cycles are needed for the 100-word block compare?

5.39 [2–3 months] <§§5.1–5.5> Using standard parts, build a machine that implements the multicycle machine in this chapter.

- 5.40** [15] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.41** [15] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.42** [15] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.43** [15] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.44** [15] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.45** [20] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.46** [10] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.47** [15] <§§5.1–5.5>  **For More Practice:** Comparing Processor Performance
- 5.48** [20] <§5.5>  **For More Practice:** Implementing Instructions in MIPS
- 5.49** [30] <§5.6> We wish to add the instruction `eret` (exception return) to the multicycle datapath described in this chapter. A primary task of the `eret` instruction is to reload the PC with the return address at which an exception, or error trap occurred. Suppose that if the processor is serving an error trap, then the PC has to be loaded from a register `ErrorPC`. Otherwise the processor is serving an exception) the PC has to be loaded from `EPC`. Suppose that there is a bit in the cause register called `trap` to encode an error trap when it occurs and to save the PC in the `ErrorPC` register. Add any necessary datapaths and control signals to the multicycle datapath of Figure 5.39 on page 344 to accommodate the trap/exception call and return, and show the necessary modifications to the finite state machine of Figure 5.40 on page 345 to implement the `eret` instruction. You can photocopy the figures to make it easier to show your modifications.
- 5.50** [6] <§5.6> Exceptions occur when a control flow change is required to handle an unexpected event in the processor. How can the cause and the instruction that caused the exception, be represented by the hardware in a MIPS machine? Give two examples for conditions that a processor can handle by restarting execution of instructions after handling the exception, and two others for exceptions that lead to program termination.
- 5.51** [6] <§5.6> Exception detection is an important aspect of exception handling. Try to identify the cycle in which the following exceptions can be detected for the multicycle datapath in Figure 5.28 on page 323.

Consider the following exceptions:

- Divide by zero exception (suppose we use the same ALU for division in one cycle, and that it is recognized by the rest of the control)
- Overflow exception

- c. Invalid instruction
- d. External interrupt
- e. Invalid instruction memory address
- f. Invalid data memory address


5.52 [15] <\$5.6>  **For More Practice:** Adding Instructions to the Datapath

5.53 [30] <\$5.7> Microcode has been used to add more powerful instructions to an instruction set; let's explore the potential benefits of this approach. Devise a strategy for implementing the `bcmp` instruction described in Exercise 5.38 using the multicycle datapath and microcode. You will probably need to make some changes to the datapath in order to efficiently implement the `bcmp` instruction. Provide a description of your proposed changes and describe how the `bcmp` instruction will work. Are there any advantages that can be obtained by adding internal registers to the datapath to help support the `bcmp` instruction? Estimate the improvement in performance that you can achieve by implementing the instruction in hardware (as opposed to the software solution you obtained in Exercise 5.38) and explain where the performance increase comes from.

5.54 [30] <\$5.7>  **For More Practice:** Microcode

5.55 [30] <\$5.7>  **For More Practice:** Microcode

5.56 [5] <\$5.7>  **For More Practice:** Microcode

5.57 [30] <\$5.8> Using the strategy you developed in Exercise 5.53, modify the MIPS microinstruction format described in  Figure 5.7.1 and provide the complete microprogram for the `bcmp` instruction. Describe in detail how you extended the microcode so as to support the creation of more complex control structures (such as a loop) within the microcode. Has support for the `bcmp` instruction changed the size of the microcode? Will other instructions besides `bcmp` be affected by the change in the microinstruction format?

5.58 [5] <\$5.8> A and B are registers defined through the following Verilog initialization code:

```
reg A,B
initial begin
    A = 1;
    B = 2;
end
```


Analyze the following two segments of Verilog description lines, and compare the results of variables A and B, and the operation done in each example.


```
a)      always @(negedge clock) begin
        A = B;
        B = A;
      end
b)      always @(negedge clock) begin
        A <= B;
        B <= A;
      end
```

5.59 [15] <§§5.4, 5.8> Write the ALUControl module in combinational Verilog using the following form as the basis:

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
    input ALUOp[1:0], FuncCode[5:0];
    output ALUCtl[3:0];
    ....
endmodule
```

5.60 [1 week] <§§5.3, 5.4, 5.8> Using a hardware simulation language such as Verilog, implement a functional simulator for the single-cycle version. Build your simulator using an existing library of parts, if such a library is available. If the parts contain timing information, determine what the cycle time of your implementation will be.

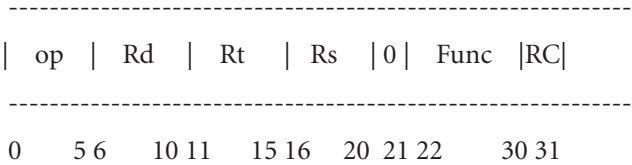
5.61 [2–4 hours] <§§4.7, 5.5, 5.8, 5.8> Extend the multicycle Verilog description in  5.8 by adding an implementation of the unsigned MIPS multiply instruction; assume it is implemented using the MIPS ALU and a shift and add operation.

5.62 [2–4 hours] <§§4.7, 5.5, 5.8, 5.9> Extend the multicycle Verilog description in  5.8 by adding an implementation of the unsigned MIPS divide instruction; assume it is implemented using the MIPS ALU with a one-bit-at-a-time algorithm.

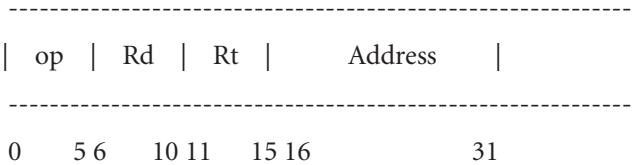
5.63 [1 week] <§§5.5, 5.8> Using a hardware simulation language such as Verilog, implement a functional simulator for a multicycle implementation of the design of a PowerPC processor. Build your simulator using an existing library of parts, if such a library is available. If the parts contain timing information, determine what the cycle time of your implementation will be.

Like MIPS, the PowerPC instructions are 32 bits each. Assume that your instruction set supports the following instruction formats:

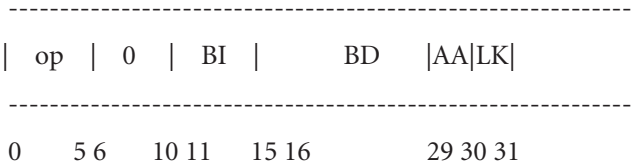
R-type



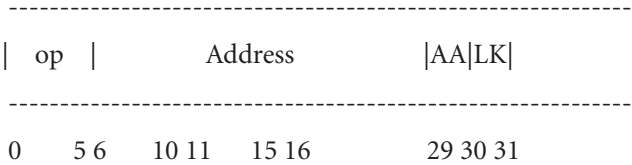
Load/store & immediate



Branch conditional



Jump



RC-reg



 0 1 2 3

Func field (22:30): Similar to MIPS, identifies function code.

RC bit(31): IF set (1), update the RC-reg control bits to reflect the results of the instruction (all R-type).

AA(30): 1 indicates that the given address is an absolute address;
 0 indicates relative address.

LK: IF 1, updates LNK R (the link register), which can be later used for subroutine return implementation.

BI: Encodes the branch condition (e.g., beg -> BI = 2, blt -> BI = 0, etc.)

BD: Branch relative destination.

Your simplified PowerPC implementation should be able to implement the following instructions:

```
add:      add $Rd, $Rt, $Rs      ($Rd <- $Rt + $Rs)
          addi $Rd, $Rt, #n      ($Rd <- $Rt + #n)
subtract: sub $Rd, $Rt, $Rs      ($Rd <- $Rt - $Rs)
          subi $Rd, $Rt, #n      ($Rd <- $Rt - #n)
load:     lw $Rd, Addr($Rt)      ($Rd <- Memory[$Rt + Addr])
store:    sw $Rd, Addr($Rt)      (Memory[$Rt + Addr] <- $Rd)
AND, OR:  and/or $Rd, $Rt, $Rs    ($Rd <- $Rt AND/OR $Rs)
          andi/ori $Rd, $Rt, #n    ($Rd <- $Rt AND/OR #n)
Jump:     jmp Addr (PC <- Addr)
Branch conditional: Beq Addr (CR[2]==1? PC<- PC+BD : PC <- PC+4)
subroutine call: jal Addr (LNKR <- PC+4; PC<- Addr)
subroutine restore: Ret (PC <- LNK R)
```

5.64 [Discussion] <§§5.7, 5.10, 5.11> Hypothesis: If the first implementation of an architecture uses microprogramming, it affects the instruction set architecture. Why might this be true? Can you find an architecture that will probably always use microcode? Why? Which machines will never use microcode? Why? What control

implementation do you think the architect had in mind when designing the instruction set architecture?

5.65 [Discussion] <§§5.7, 5.12> Wilkes invented microprogramming in large part to simplify construction of control. Since 1980, there has been an explosion of computer-aided design software whose goal is also to simplify construction of control. This has made control design much easier. Can you find evidence, based either on the tools or on real designs, that supports or refutes this hypothesis?

5.66 [Discussion] <§5.12> The MIPS instructions and the MIPS microinstructions have many similarities. What would make it difficult for a compiler to produce MIPS microcode rather than macrocode? What changes to the microarchitecture would make the microcode more useful for this application.

Answers to Check Yourself

§5.1, page 289: 3.

§5.2, page 292: false.

§5.3, page 299: A.

§5.4, page 318: Yes, MemtoReg and RegDst are inverses of one another. Yes, simply use the other signal and flip the order of the inputs to the multiplexor!

§5.5, page 340: 1. False. 2. Maybe: If the signal PCSource[0] is always set to zero when it is a don't care (which is most states), then it is identical to PCWriteCond.

§5.6, page 346: No, since the value of 11, which was formerly unused, is now used!

§5.7, page 5.7-13: 4 tables with 55 entries (don't forget the primary dispatch!)

§5.8, page 5.8-7: 1. 0, 1, 1, X, 0. 2. No, since state is not assigned on every path.