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

Example 7.2 `addi` INSTRUCTION

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

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

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

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

Example 7.3 `j` INSTRUCTION

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

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

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

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

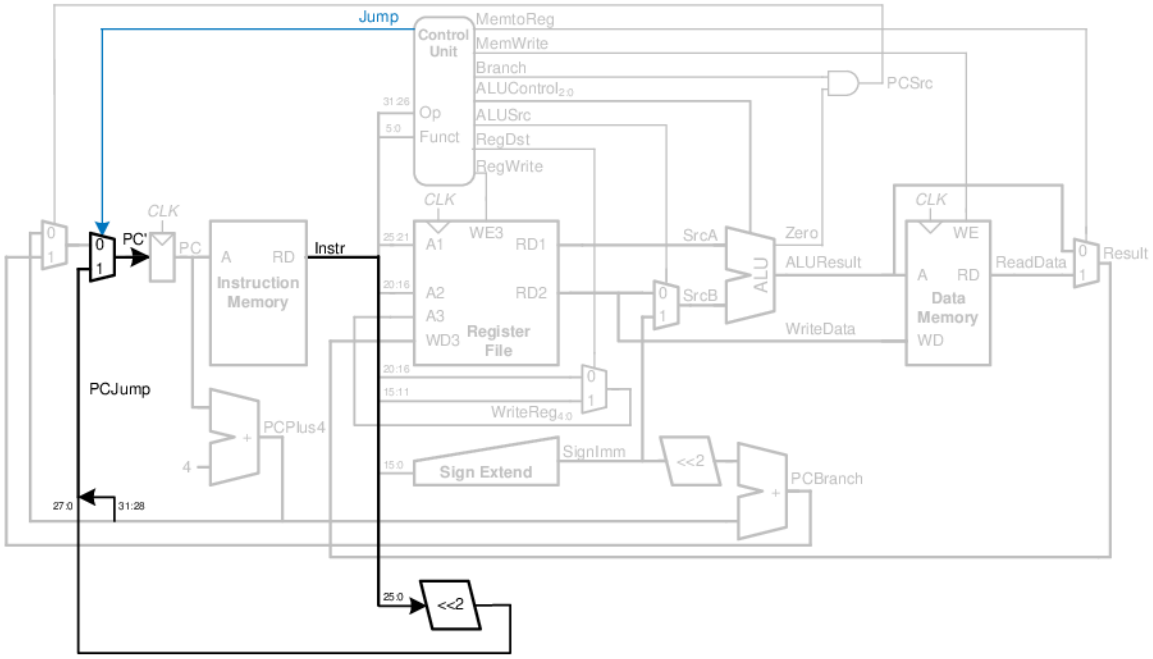

Figure 7.14 Single-cycle MIPS datapath enhanced to support the `j` instruction

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

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

7.3.4 Performance Analysis

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

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

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

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

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

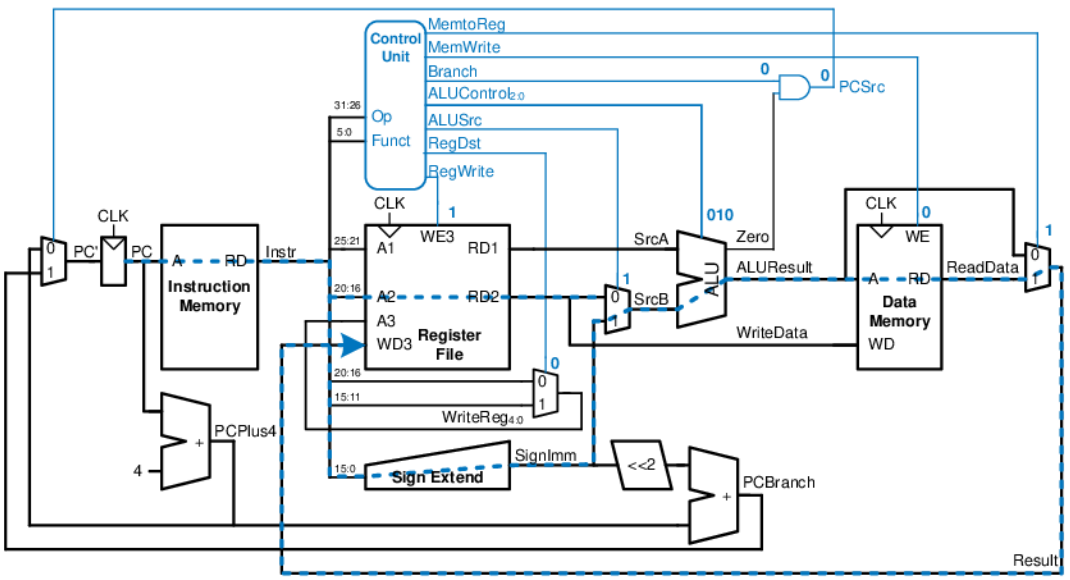

Figure 7.15 Critical path for `lw` instruction

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

Example 7.4 SINGLE-CYCLE PROCESSOR PERFORMANCE

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

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

Table 7.6 Delays of circuit elements

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

7.4 MULTICYCLE PROCESSOR

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

The multicycle processor addresses these weaknesses by breaking an instruction into multiple shorter steps. In each short step, the processor can read or write the memory or register file or use the ALU. Different instructions use different numbers of steps, so simpler instructions can complete faster than more complex ones. The processor needs only one adder; this adder is reused for different purposes on various steps. And the processor uses a combined memory for instructions and data. The instruction is fetched from memory on the first step, and data may be read or written on later steps.

We design a multicycle processor following the same procedure we used for the single-cycle processor. First, we construct a datapath by connecting the architectural state elements and memories with combinational logic. But, this time, we also add nonarchitectural state elements to hold intermediate results between the steps. Then we design the controller. The controller produces different signals on different steps during execution of a single instruction, so it is now a finite state machine rather than combinational logic. We again examine how to add new instructions to the processor. Finally, we analyze the performance of the multicycle processor and compare it to the single-cycle processor.

7.4.1 Multicycle Datapath

Again, we begin our design with the memory and architectural state of the MIPS processor, shown in Figure 7.16. In the single-cycle design, we used separate instruction and data memories because we needed to read the instruction memory and read or write the data memory all in one cycle. Now, we choose to use a combined memory for both instructions and data. This is more realistic, and it is feasible because we can read the instruction in one cycle, then read or write the data in a separate cycle. The PC and register file remain unchanged. We gradually build the datapath by adding components to handle each step of each instruction. The new connections are emphasized in black (or blue, for new control signals), whereas the hardware that has already been studied is shown in gray.

The PC contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. Figure 7.17 shows that the PC is simply connected to the address input of the instruction memory. The instruction is read and stored in a new nonarchitectural

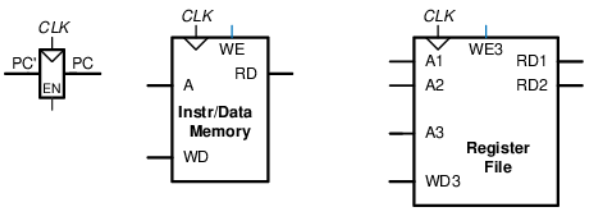

Figure 7.16 State elements with unified instruction/data memory

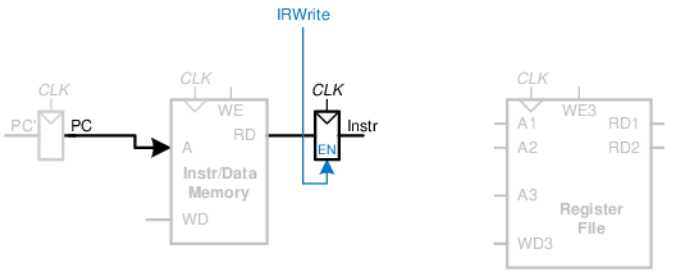

Figure 7.17 Fetch instruction from memory

Instruction Register so that it is available for future cycles. The Instruction Register receives an enable signal, called *IRWrite*, that is asserted when it should be updated with a new instruction.

As we did with the single-cycle processor, we will work out the datapath connections for the `lw` instruction. Then we will enhance the datapath to handle the other instructions. For a `lw` instruction, the next step is to read the source register containing the base address. This register is specified in the `rs` field of the instruction, $Instr_{25:21}$. These bits of the instruction are connected to one of the address inputs, *A1*, of the register file, as shown in Figure 7.18. The register file reads the register onto *RD1*. This value is stored in another nonarchitectural register, *A*.

The `lw` instruction also requires an offset. The offset is stored in the immediate field of the instruction, $Instr_{15:0}$ and must be sign-extended to 32 bits, as shown in Figure 7.19. The 32-bit sign-extended value is called *SignImm*. To be consistent, we might store *SignImm* in another nonarchitectural register. However, *SignImm* is a combinational function of *Instr* and will not change while the current instruction is being processed, so there is no need to dedicate a register to hold the constant value.

The address of the load is the sum of the base address and offset. We use an ALU to compute this sum, as shown in Figure 7.20. *ALUControl* should be set to 010 to perform an addition. *ALUResult* is stored in a nonarchitectural register called *ALUOut*.

The next step is to load the data from the calculated address in the memory. We add a multiplexer in front of the memory to choose the

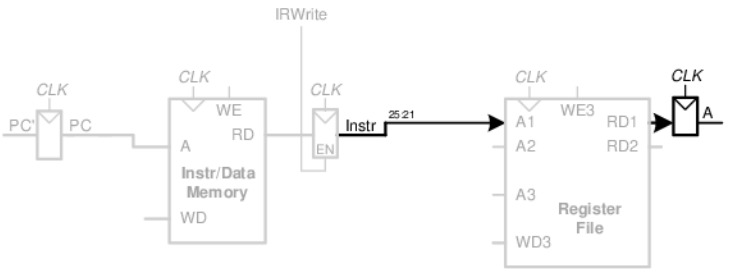

Figure 7.18 Read source operand from register file

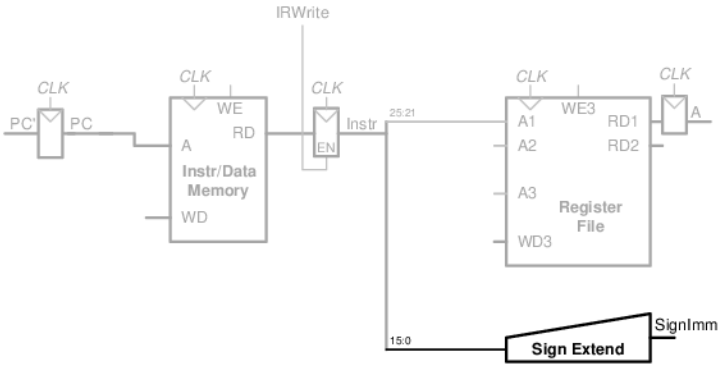

Figure 7.19 Sign-extend the immediate

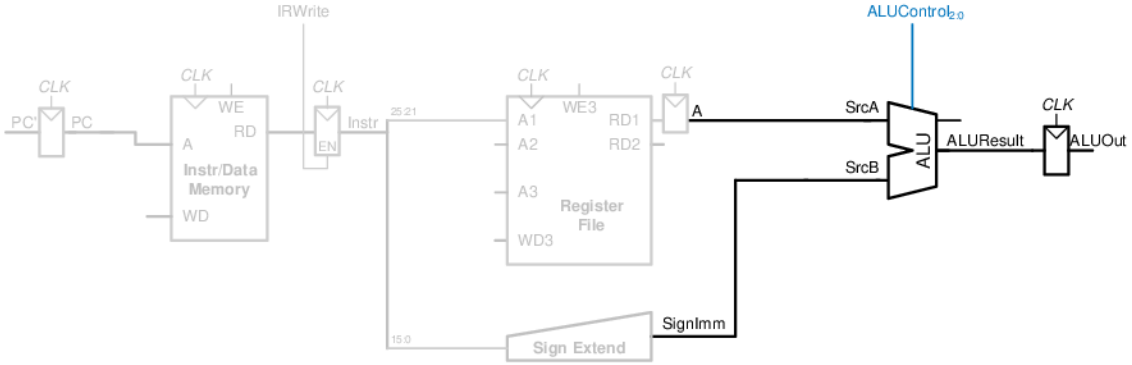

Figure 7.20 Add base address to offset

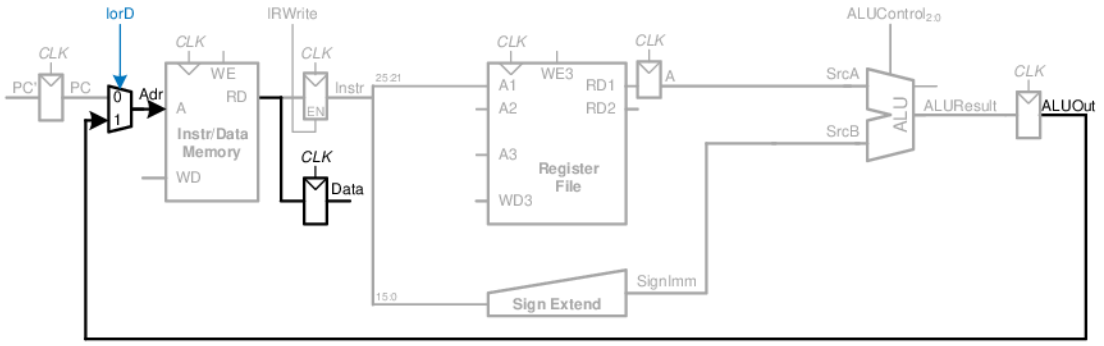

Figure 7.21 Load data from memory

memory address, *Adr*, from either the PC or *ALUOut*, as shown in Figure 7.21. The multiplexer select signal is called *IorD*, to indicate either an instruction or data address. The data read from the memory is stored in another nonarchitectural register, called *Data*. Notice that the address multiplexer permits us to reuse the memory during the `lw` instruction. On the first step, the address is taken from the PC to fetch the instruction. On a later step, the address is taken from *ALUOut* to load the data. Hence, *IorD* must have different values on different steps. In Section 7.4.2, we develop the FSM controller that generates these sequences of control signals.

Finally, the data is written back to the register file, as shown in Figure 7.22. The destination register is specified by the `rt` field of the instruction, $Instr_{20:16}$.

While all this is happening, the processor must update the program counter by adding 4 to the old PC. In the single-cycle processor, a separate adder was needed. In the multicycle processor, we can use the existing ALU on one of the steps when it is not busy. To do so, we must insert source multiplexers to choose the PC and the constant 4 as ALU inputs, as shown in Figure 7.23. A two-input multiplexer controlled by *ALUSrcA* chooses either the PC or register *A* as *SrcA*. A four-input multiplexer controlled by *ALUSrcB* chooses either 4 or *SignImm* as *SrcB*. We use the other two multiplexer inputs later when we extend the datapath to handle other instructions. (The numbering of inputs to the multiplexer is arbitrary.) To update the PC, the ALU adds *SrcA* (*PC*) to *SrcB* (4), and the result is written into the program counter register. The *PCWrite* control signal enables the PC register to be written only on certain cycles.

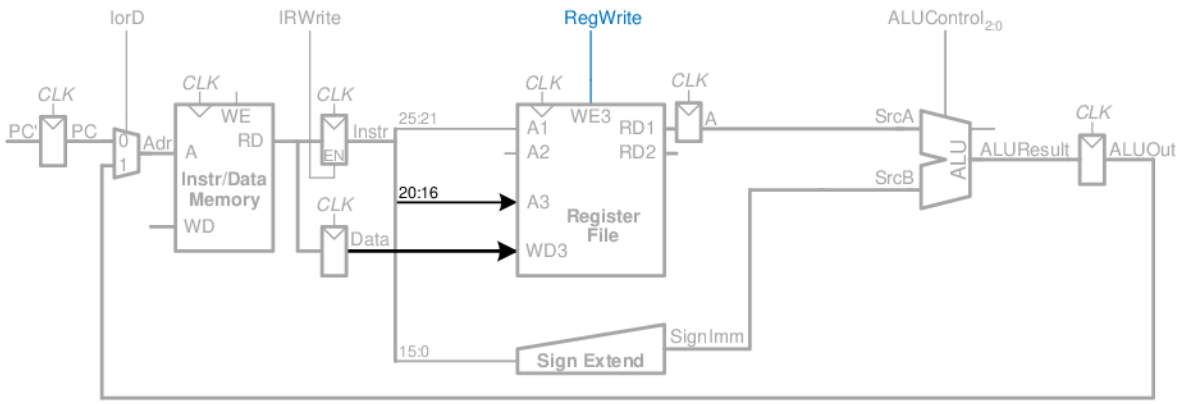

Figure 7.22 Write data back to register file

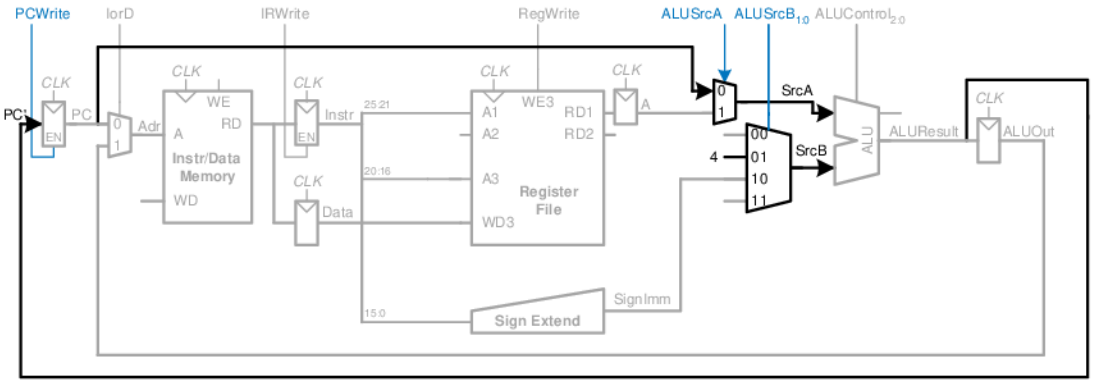

Figure 7.23 Increment PC by 4

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

The only new feature of `sw` is that we must read a second register from the register file and write it into the memory, as shown in Figure 7.24. The register is specified in the `rt` field of the instruction, $Instr_{20:16}$, which is connected to the second port of the register file. When the register is read, it is stored in a nonarchitectural register, *B*. On the next step, it is sent to the write data port (*WD*) of the data memory to be written. The memory receives an additional *MemWrite* control signal to indicate that the write should occur.

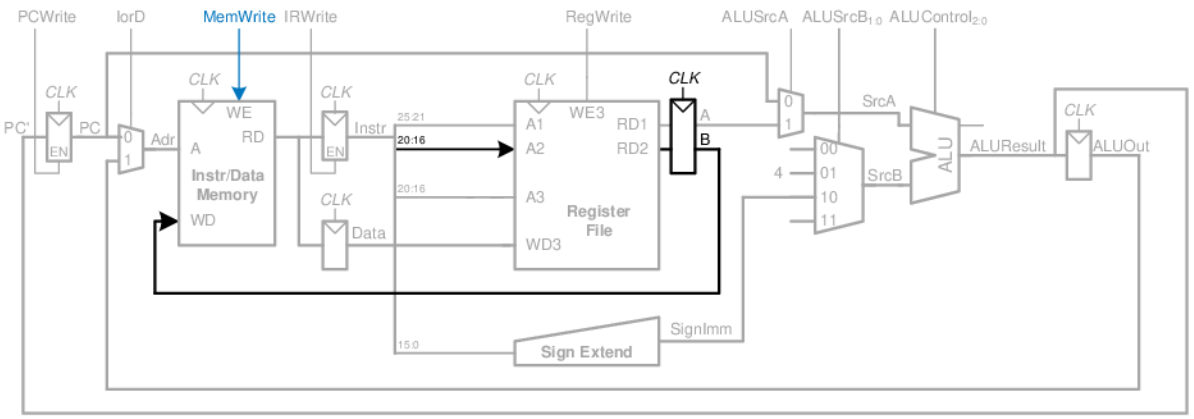

Figure 7.24 Enhanced datapath for `sw` instruction

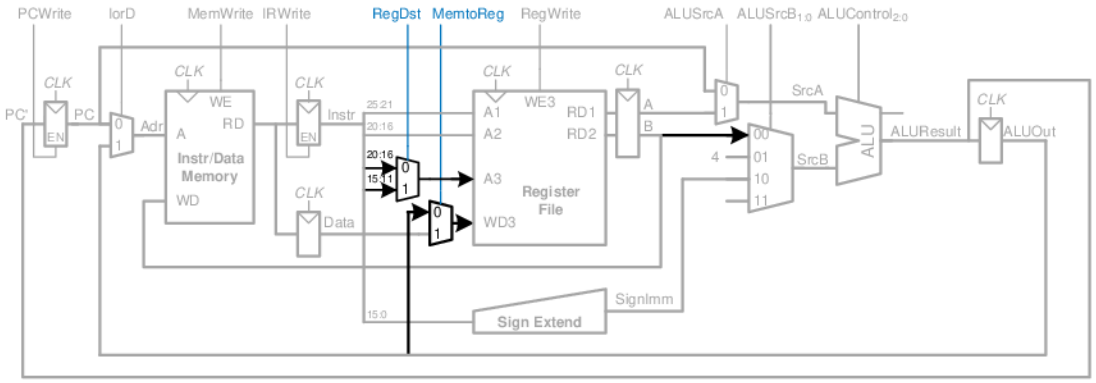

Figure 7.25 Enhanced datapath for R-type instructions

For R-type instructions, the instruction is again fetched, and the two source registers are read from the register file. Another input of the *SrcB* multiplexer is used to choose register *B* as the second source register for the ALU, as shown in Figure 7.25. The ALU performs the appropriate operation and stores the result in *ALUOut*. On the next step, *ALUOut* is written back to the register specified by the `rd` field of the instruction, $Instr_{15:11}$. This requires two new multiplexers. The *MemtoReg* multiplexer selects whether *WD3* comes from *ALUOut* (for R-type instructions) or from *Data* (for `lw`). The *RegDst* instruction selects whether the destination register is specified in the `rt` or `rd` field of the instruction.

For the `beq` instruction, the instruction is again fetched, and the two source registers are read from the register file. To determine whether the registers are equal, the ALU subtracts the registers and examines the *Zero* flag. Meanwhile, the datapath must compute the next value of the PC if the branch is taken: $PC' = PC + 4 + SignImm \times 4$. In the single-cycle processor, yet another adder was needed to compute the branch address. In the multicycle processor, the ALU can be reused again to save hardware. On one step, the ALU computes $PC + 4$ and writes it back to the program counter, as was done for other instructions. On another step, the ALU uses this updated PC value to compute $PC + SignImm \times 4$. *SignImm* is left-shifted by 2 to multiply it by 4, as shown in Figure 7.26. The *SrcB* multiplexer chooses this value and adds it to the PC. This sum represents the destination of the branch and is stored in *ALUOut*. A new multiplexer, controlled by *PCSrc*, chooses what signal should be sent to *PC'*. The program counter should be written either when *PCWrite* is asserted or when a branch is taken. A new control signal, *Branch*, indicates that the `beq` instruction is being executed. The branch is taken if *Zero* is also asserted. Hence, the datapath computes a new PC write

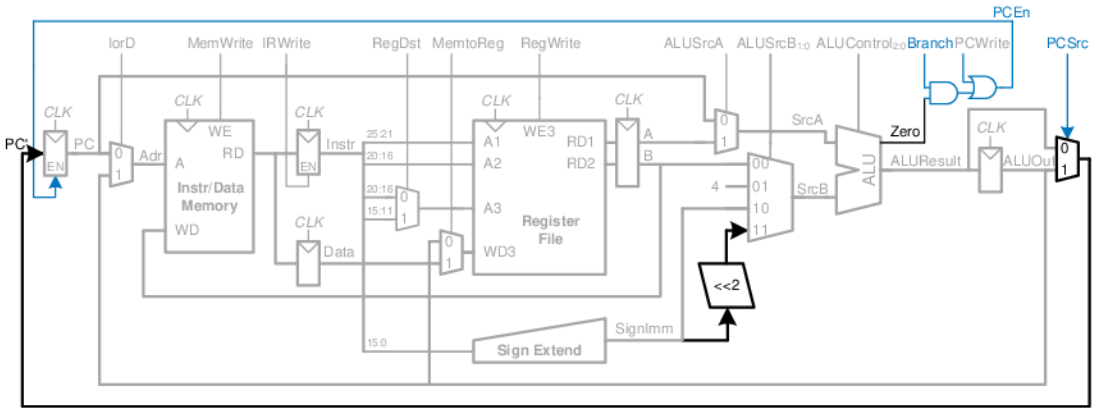

Figure 7.26 Enhanced datapath for `beq` instruction

enable, called *PCEn*, which is TRUE either when *PCWrite* is asserted or when both *Branch* and *Zero* are asserted.

This completes the design of the multicycle MIPS processor datapath. The design process is much like that of the single-cycle processor in that hardware is systematically connected between the state elements to handle each instruction. The main difference is that the instruction is executed in several steps. Nonarchitectural registers are inserted to hold the results of each step. In this way, the ALU can be reused several times, saving the cost of extra adders. Similarly, the instructions and data can be stored in one shared memory. In the next section, we develop an FSM controller to deliver the appropriate sequence of control signals to the datapath on each step of each instruction.

7.4.2 Multicycle Control

As in the single-cycle processor, the control unit computes the control signals based on the `opcode` and `funct` fields of the instruction, $Instr_{31:26}$ and $Instr_{5:0}$. Figure 7.27 shows the entire multicycle MIPS processor with the control unit attached to the datapath. The datapath is shown in black, and the control unit is shown in blue.

As in the single-cycle processor, the control unit is partitioned into a main controller and an ALU decoder, as shown in Figure 7.28. The ALU decoder is unchanged and follows the truth table of Table 7.2. Now, however, the main controller is an FSM that applies the proper control signals on the proper cycles or steps. The sequence of control signals depends on the instruction being executed. In the remainder of this section, we will develop the FSM state transition diagram for the main controller.

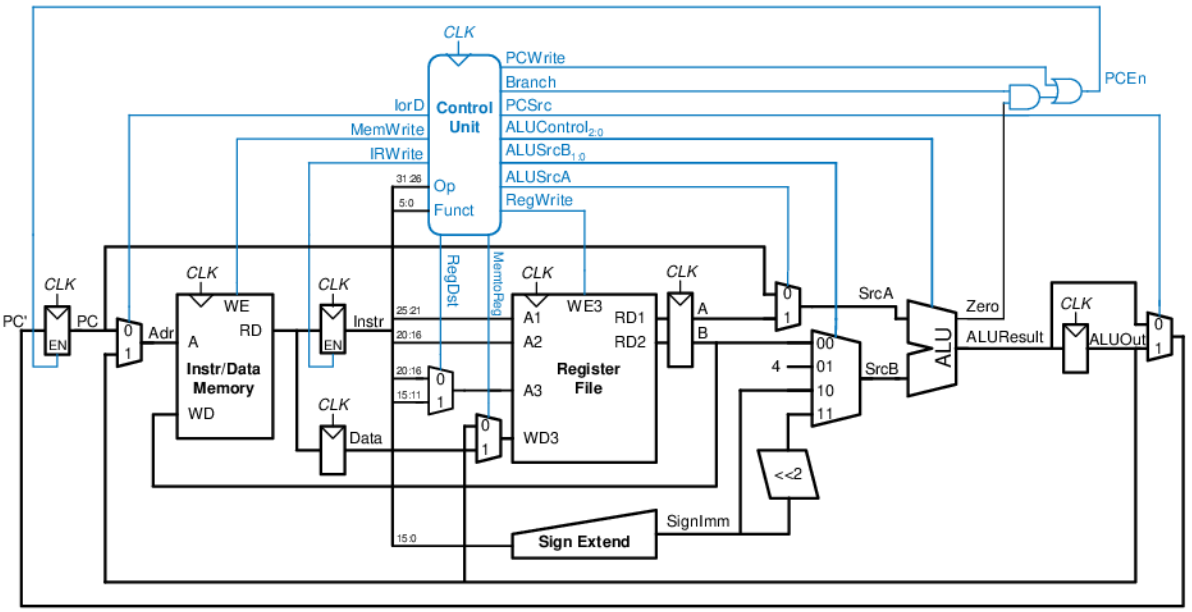

Figure 7.27 Complete multicycle MIPS processor

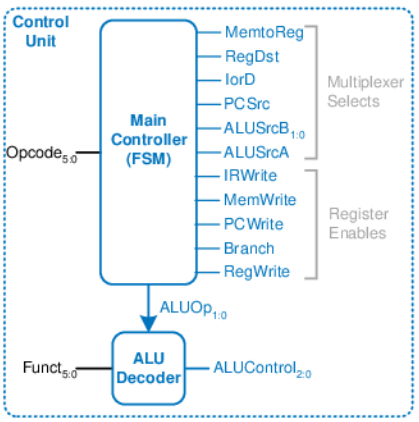

Figure 7.28 Control unit internal structure

The main controller produces multiplexer select and register enable signals for the datapath. The select signals are *MemtoReg*, *RegDst*, *IorD*, *PCSrc*, *ALUSrcB*, and *ALUSrcA*. The enable signals are *IRWrite*, *MemWrite*, *PCWrite*, *Branch*, and *RegWrite*.

To keep the following state transition diagrams readable, only the relevant control signals are listed. Select signals are listed only when their value matters; otherwise, they are don't cares. Enable signals are listed only when they are asserted; otherwise, they are 0.

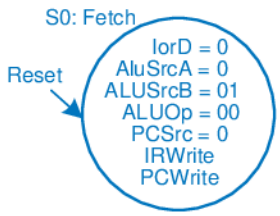

Figure 7.29 Fetch

The first step for any instruction is to fetch the instruction from memory at the address held in the PC. The FSM enters this state on reset. To read memory, *IorD* = 0, so the address is taken from the PC. *IRWrite* is asserted to write the instruction into the instruction register, IR. Meanwhile, the PC should be incremented by 4 to point to the next instruction. Because the ALU is not being used for anything else, the processor can use it to compute *PC* + 4 at the same time that it fetches the instruction. *ALUSrcA* = 0, so *SrcA* comes from the PC. *ALUSrcB* = 01, so *SrcB* is the constant 4. *ALUOp* = 00, so the ALU decoder produces *ALUControl* = 010 to make the ALU add. To update the PC with this new value, *PCSrc* = 0, and *PCWrite* is asserted. These control signals are shown in Figure 7.29. The data flow on this step is shown in Figure 7.30, with the instruction fetch shown using the dashed blue line and the PC increment shown using the dashed gray line.

The next step is to read the register file and decode the instruction. The register file always reads the two sources specified by the `rs` and `rt` fields of the instruction. Meanwhile, the immediate is sign-extended. Decoding involves examining the `opcode` of the instruction to determine what to do next. No control signals are necessary to decode the instruction, but the FSM must wait 1 cycle for the reading and decoding to complete, as shown in Figure 7.31. The new state is highlighted in blue. The data flow is shown in Figure 7.32.

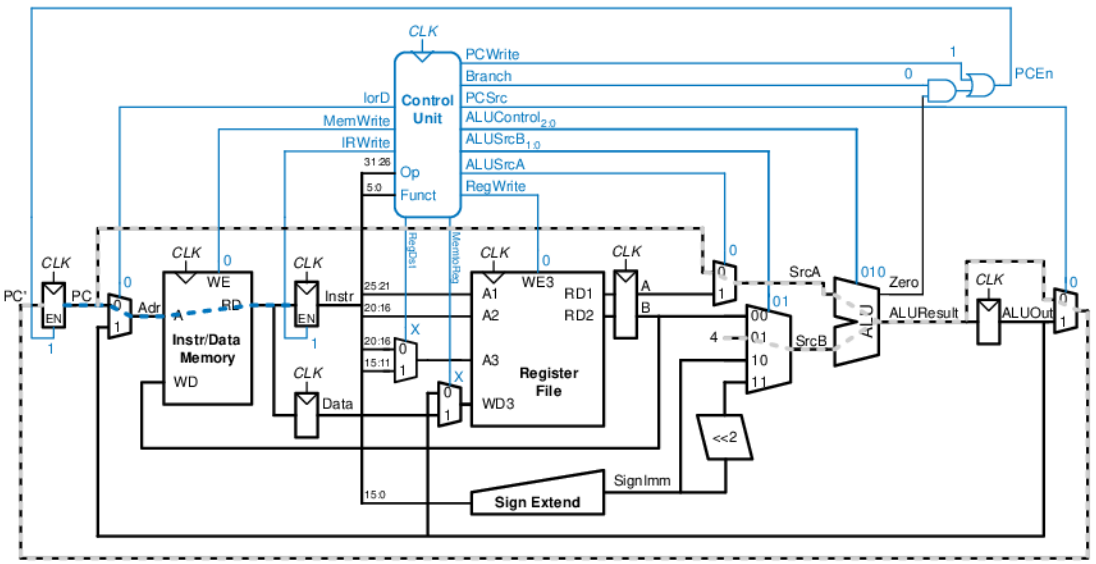

Figure 7.30 Data flow during the fetch step

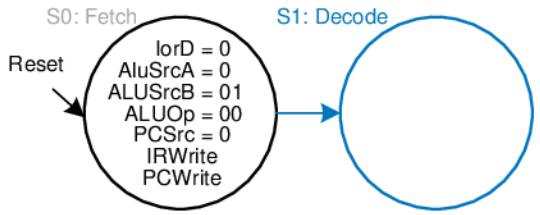

Figure 7.31 Decode

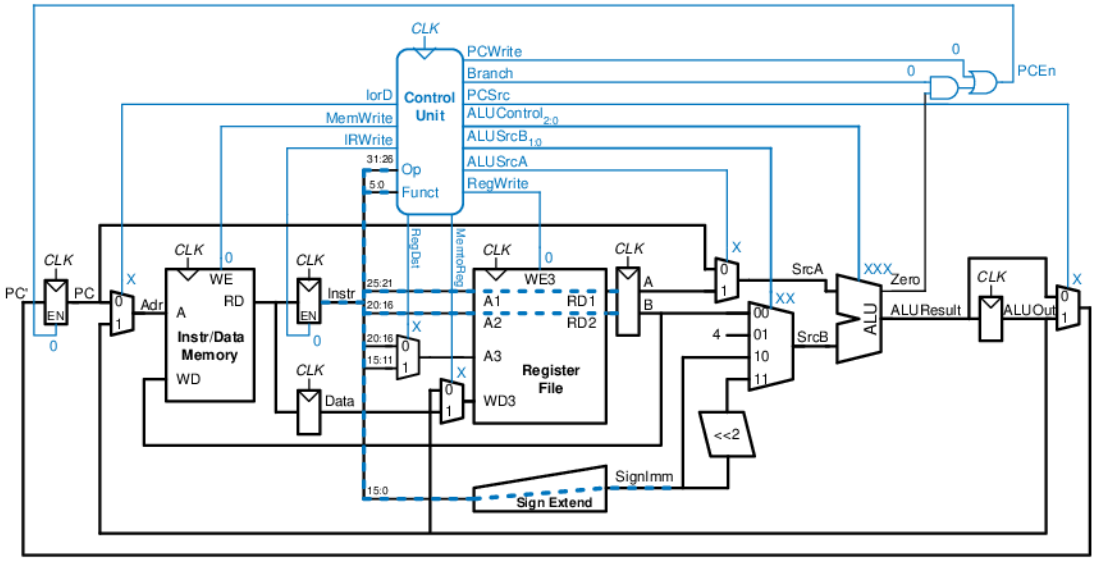

Figure 7.32 Data flow during the decode step

Now the FSM proceeds to one of several possible states, depending on the `opcode`. If the instruction is a memory load or store (`lw` or `sw`), the multicycle processor computes the address by adding the base address to the sign-extended immediate. This requires *ALUSrcA* = 1 to select register *A* and *ALUSrcB* = 10 to select *SignImm*. *ALUOp* = 00, so the ALU adds. The effective address is stored in the *ALUOut* register for use on the next step. This FSM step is shown in Figure 7.33, and the data flow is shown in Figure 7.34.

If the instruction is `lw`, the multicycle processor must next read data from memory and write it to the register file. These two steps are shown in Figure 7.35. To read from memory, *IorD* = 1 to select the memory address that was just computed and saved in *ALUOut*. This address in memory is read and saved in the Data register during step S3. On the next step, S4, *Data* is written to the register file. *MemtoReg* = 1 to select

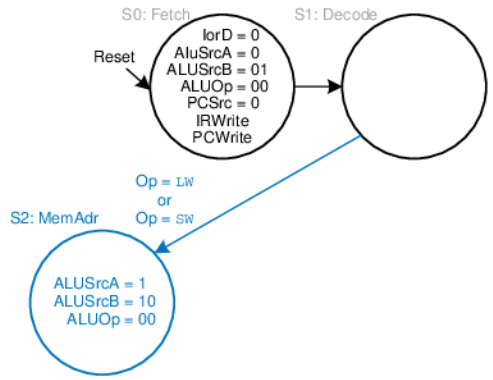

Figure 7.33 Memory address computation

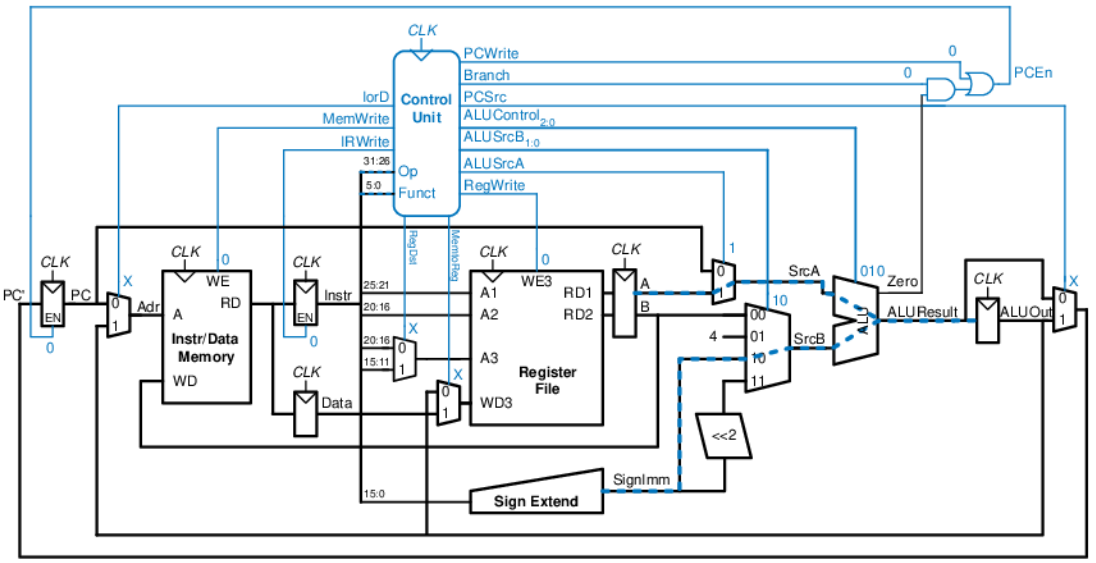

Figure 7.34 Data flow during memory address computation

Data, and *RegDst* = 0 to pull the destination register from the `rt` field of the instruction. *RegWrite* is asserted to perform the write, completing the `lw` instruction. Finally, the FSM returns to the initial state, S0, to fetch the next instruction. For these and subsequent steps, try to visualize the data flow on your own.

From state S2, if the instruction is `sw`, the data read from the second port of the register file is simply written to memory. *IorD* = 1 to select the address computed in S2 and saved in *ALUOut*. *MemWrite* is asserted to write the memory. Again, the FSM returns to S0 to fetch the next instruction. The added step is shown in Figure 7.36.

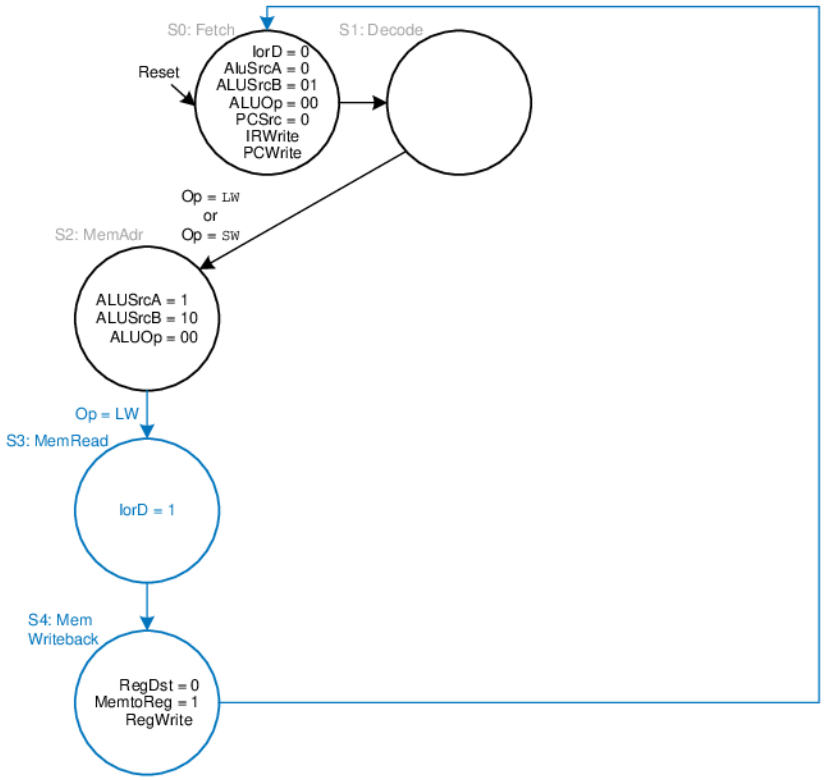

Figure 7.35 Memory read

If the `opcode` indicates an R-type instruction, the multicycle processor must calculate the result using the ALU and write that result to the register file. Figure 7.37 shows these two steps. In S6, the instruction is executed by selecting the *A* and *B* registers (*ALUSrcA* = 1, *ALUSrcB* = 00) and performing the ALU operation indicated by the `funct` field of the instruction. *ALUOp* = 10 for all R-type instructions. The *ALUResult* is stored in *ALUOut*. In S7, *ALUOut* is written to the register file, *RegDst* = 1, because the destination register is specified in the `rd` field of the instruction. *MemtoReg* = 0 because the write data, *WD3*, comes from *ALUOut*. *RegWrite* is asserted to write the register file.

For a `beq` instruction, the processor must calculate the destination address and compare the two source registers to determine whether the branch should be taken. This requires two uses of the ALU and hence might seem to demand two new states. Notice, however, that the ALU was not used during S1 when the registers were being read. The processor might as well use the ALU at that time to compute the destination address by adding the incremented PC, $PC + 4$, to $SignImm \times 4$, as shown in

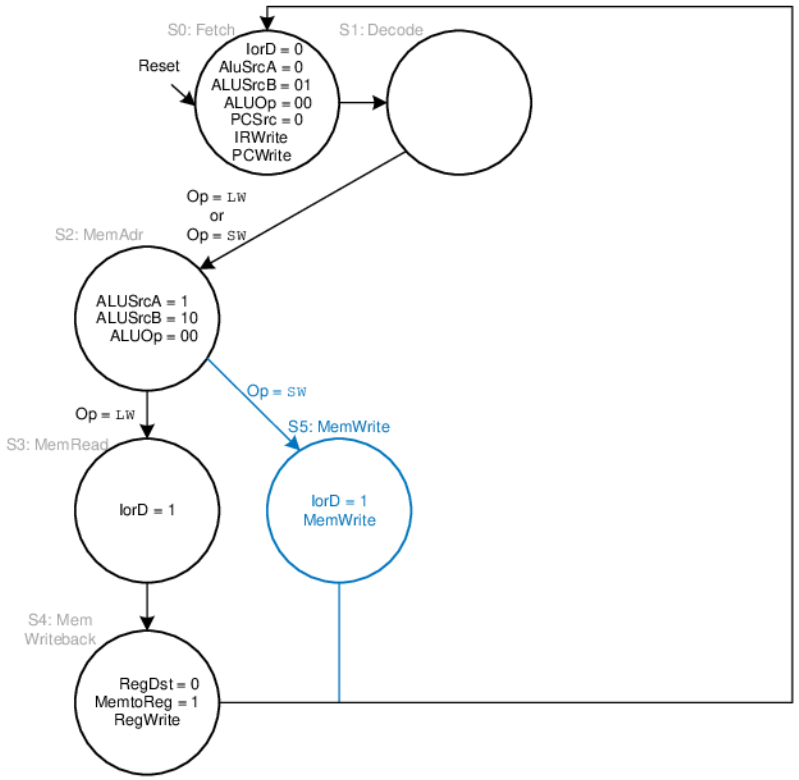

Figure 7.36 Memory write

Figure 7.38 (see page 396). *ALUSrcA* = 0 to select the incremented PC, *ALUSrcB* = 11 to select *SignImm* × 4, and *ALUOp* = 00 to add. The destination address is stored in *ALUOut*. If the instruction is not `beq`, the computed address will not be used in subsequent cycles, but its computation was harmless. In S8, the processor compares the two registers by subtracting them and checking to determine whether the result is 0. If it is, the processor branches to the address that was just computed. *ALUSrcA* = 1 to select register *A*; *ALUSrcB* = 00 to select register *B*; *ALUOp* = 01 to subtract; *PCSrc* = 1 to take the destination address from *ALUOut*, and *Branch* = 1 to update the PC with this address if the ALU result is 0.[2]

Putting these steps together, Figure 7.39 shows the complete main controller state transition diagram for the multicycle processor (see page 397). Converting it to hardware is a straightforward but tedious task using the techniques of Chapter 3. Better yet, the FSM can be coded in an HDL and synthesized using the techniques of Chapter 4.

[2] Now we see why the *PCSrc* multiplexer is necessary to choose *PC'* from either *ALUResult* (in S0) or *ALUOut* (in S8).

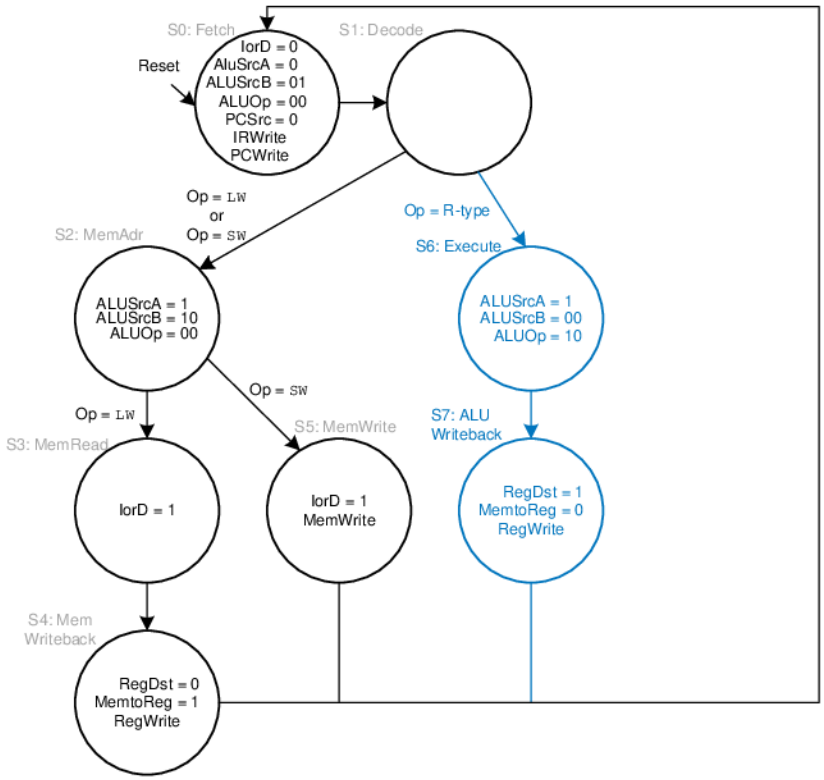

Figure 7.37 Execute R-type operation

7.4.3 More Instructions

As we did in Section 7.3.3 for the single-cycle processor, let us now extend the multicycle processor to support the `addi` and `j` instructions. The next two examples illustrate the general design process to support new instructions.

Example 7.5 `addi` INSTRUCTION

Modify the multicycle processor to support `addi`.

Solution: The datapath is already capable of adding registers to immediates, so all we need to do is add new states to the main controller FSM for `addi`, as shown in Figure 7.40 (see page 398). The states are similar to those for R-type instructions. In S9, register *A* is added to *SignImm* (*ALUSrcA* = 1, *ALUSrcB* = 10, *ALUOp* = 00) and the result, *ALUResult*, is stored in *ALUOut*. In S10, *ALUOut* is written

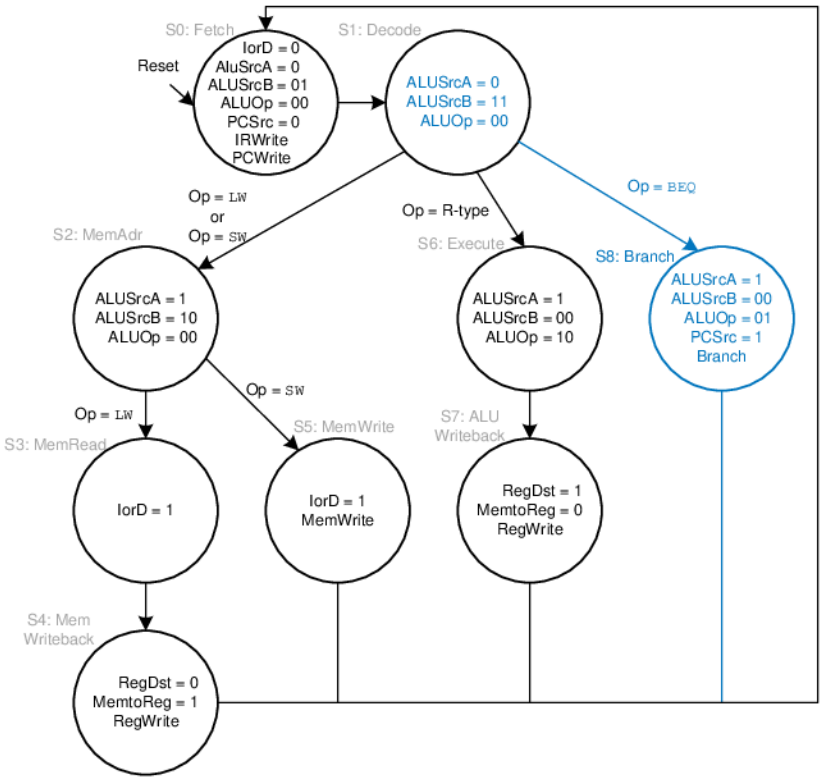

Figure 7.38 Branch

to the register specified by the `rt` field of the instruction (*RegDst* = 0, *MemtoReg* = 0, *RegWrite* asserted). The astute reader may notice that S2 and S9 are identical and could be merged into a single state.

Example 7.6 `j` INSTRUCTION

Modify the multicycle processor to support `j`.

Solution: First, we must modify the datapath to compute the next PC value in the case of a `j` instruction. Then we add a state to the main controller to handle the instruction.

Figure 7.41 shows the enhanced datapath (see page 399). The jump destination address is formed by left-shifting the 26-bit `addr` field of the instruction by two bits, then prepending the four most significant bits of the already incremented PC. The *PCSrc* multiplexer is extended to take this address as a third input.

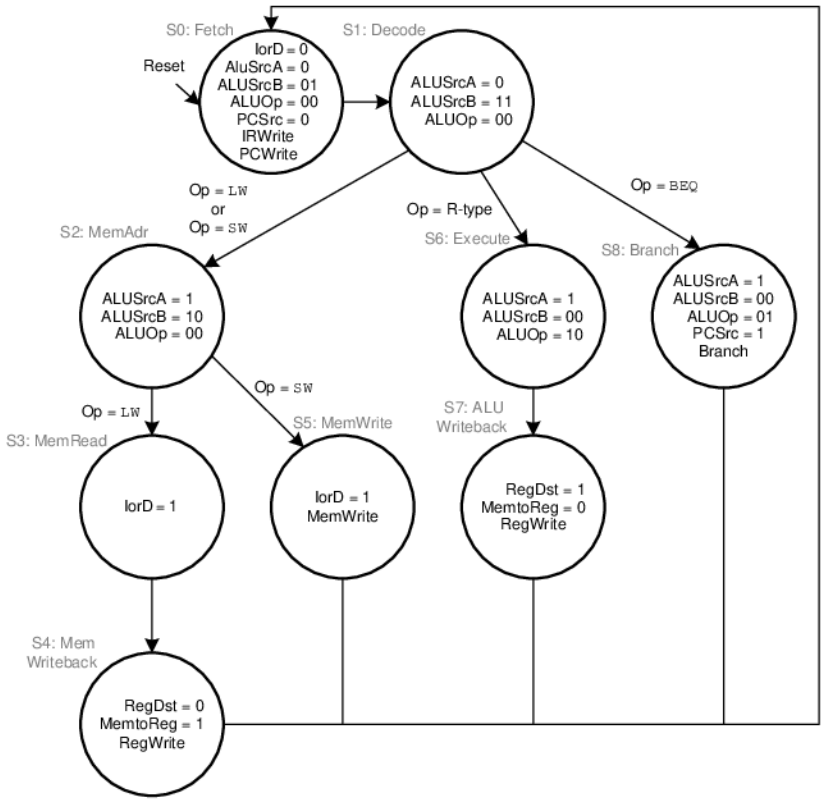

Figure 7.39 Complete multicycle control FSM

Figure 7.42 shows the enhanced main controller (see page 400). The new state, S11, simply selects PC' as the *PCJump* value (*PCSrc* = 10) and writes the PC. Note that the *PCSrc* select signal is extended to two bits in S0 and S8 as well.

7.4.4 Performance Analysis

The execution time of an instruction depends on both the number of cycles it uses and the cycle time. Whereas the single-cycle processor performed all instructions in one cycle, the multicycle processor uses varying numbers of cycles for the various instructions. However, the multicycle processor does less work in a single cycle and, thus, has a shorter cycle time.

The multicycle processor requires three cycles for `beq` and `j` instructions, four cycles for `sw`, `addi`, and R-type instructions, and five cycles for `lw` instructions. The CPI depends on the relative likelihood that each instruction is used.

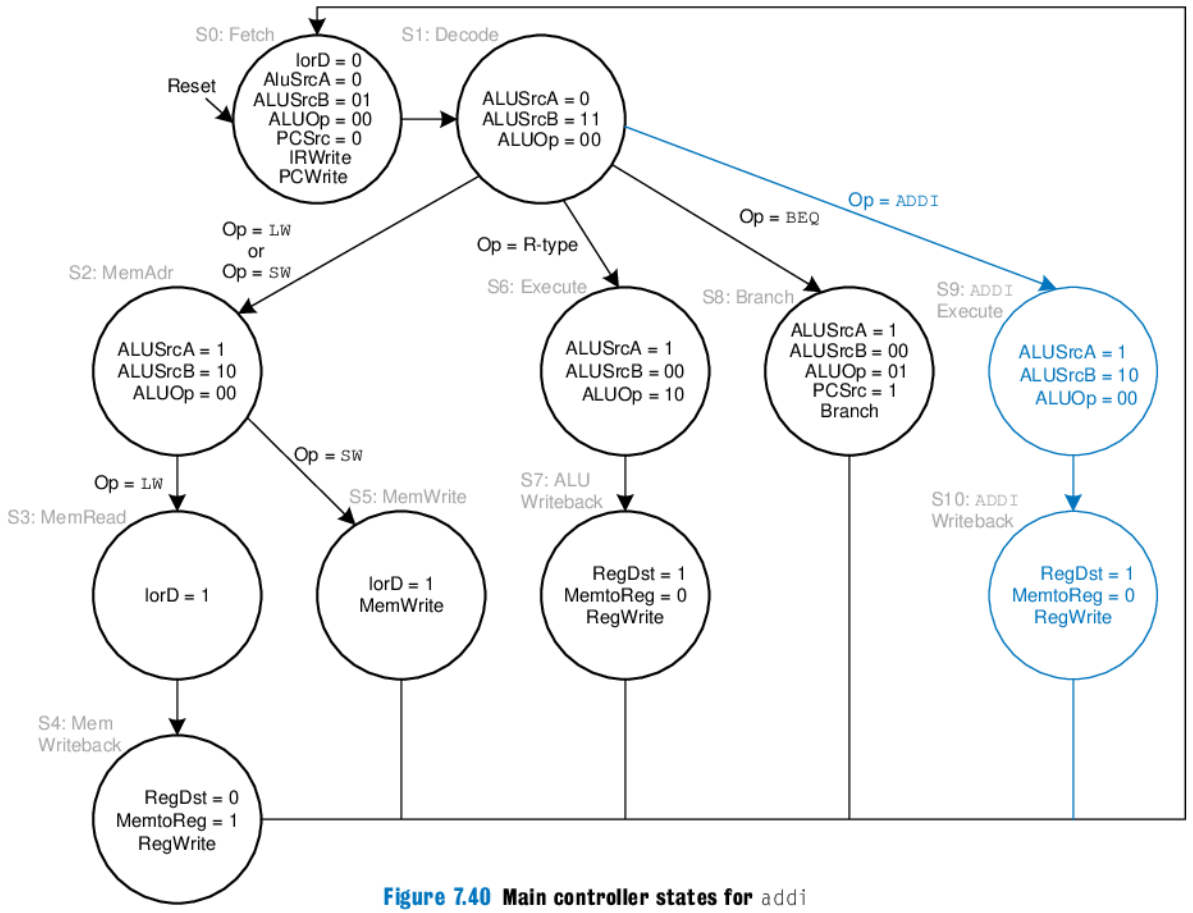

Figure 7.40 Main controller states for `addi`

Example 7.7 MULTICYCLE PROCESSOR CPI

The SPECINT2000 benchmark consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions.[3] Determine the average CPI for this benchmark.

Solution: The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of the time that instruction is used. For this benchmark, Average CPI = (0.11 + 0.02)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12. This is better than the worst-case CPI of 5, which would be required if all instructions took the same time.

---

[3] Data from Patterson and Hennessy, *Computer Organization and Design*, 3rd Edition, Morgan Kaufmann, 2005.

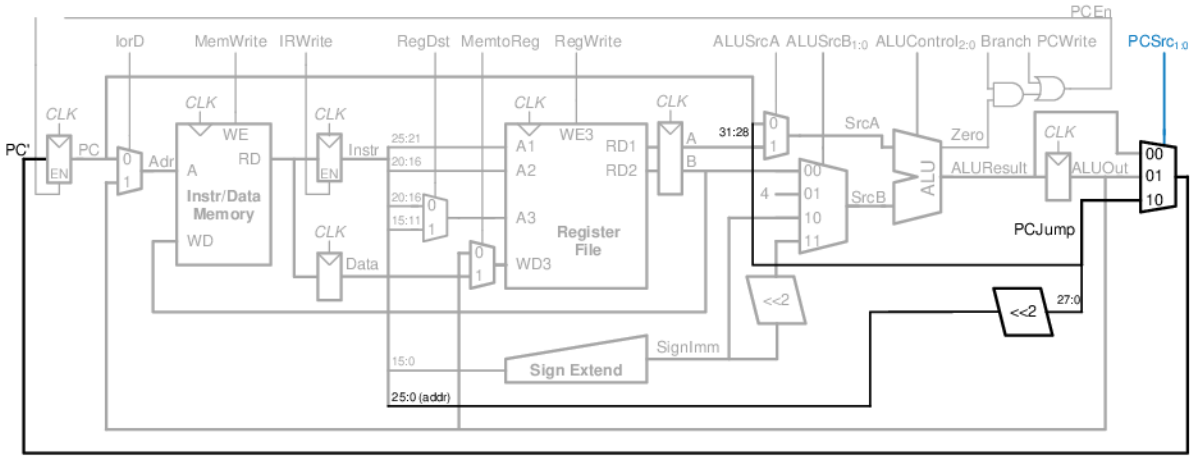

Figure 7.41 **Multicycle MIPS datapath enhanced to support the `j` instruction**

Recall that we designed the multicycle processor so that each cycle involved one ALU operation, memory access, or register file access. Let us assume that the register file is faster than the memory and that writing memory is faster than reading memory. Examining the datapath reveals two possible critical paths that would limit the cycle time:

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{\text{setup}} \qquad (7.4)$$

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

Example 7.8 PROCESSOR PERFORMANCE COMPARISON

Ben Bitdiddle is wondering whether he would be better off building the multicycle processor instead of the single-cycle processor. For both designs, he plans on using a 65 nm CMOS manufacturing process with the delays given in Table 7.6. Help him compare each processor's execution time for 100 billion instructions from the SPECINT2000 benchmark (see Example 7.7).

Solution: According to Equation 7.4, the cycle time of the multicycle processor is $T_{c2} = 30 + 25 + 250 + 20 = 325$ ps. Using the CPI of 4.12 from Example 7.7, the total execution time is $T_2 = (100 \times 10^9 \text{ instructions})(4.12 \text{ cycles/instruction})(325 \times 10^{-12} \text{ s/cycle}) = 133.9$ seconds. According to Example 7.4, the single-cycle processor had a cycle time of $T_{c1} = 950$ ps, a CPI of 1, and a total execution time of 95 seconds.

One of the original motivations for building a multicycle processor was to avoid making all instructions take as long as the slowest one. Unfortunately, this example shows that the multicycle processor is slower than the single-cycle

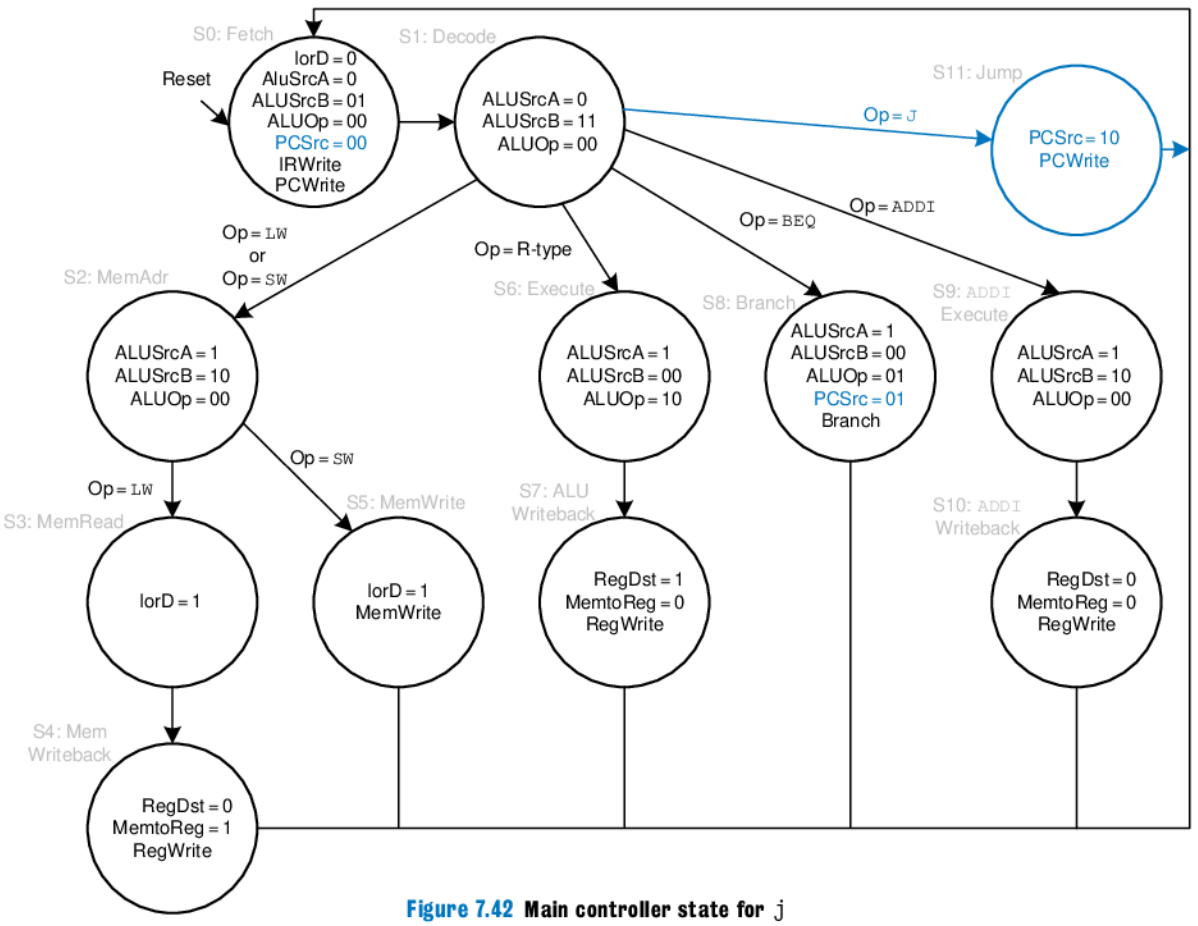

Figure 7.42 **Main controller state for `j`**

processor given the assumptions of CPI and circuit element delays. The fundamental problem is that even though the slowest instruction, `lw`, was broken into five steps, the multicycle processor cycle time was not nearly improved fivefold. This is partly because not all of the steps are exactly the same length, and partly because the 50-ps sequencing overhead of the register clk-to-Q and setup time must now be paid on every step, not just once for the entire instruction. In general, engineers have learned that it is difficult to exploit the fact that some computations are faster than others unless the differences are large.

Compared with the single-cycle processor, the multicycle processor is likely to be less expensive because it eliminates two adders and combines the instruction and data memories into a single unit. It does, however, require five nonarchitectural registers and additional multiplexers.

7.5 PIPELINED PROCESSOR

Pipelining, introduced in Section 3.6, is a powerful way to improve the throughput of a digital system. We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can execute simultaneously, one in each stage. Because each stage has only one-fifth of the entire logic, the clock frequency is almost five times faster. Hence, the latency of each instruction is ideally unchanged, but the throughput is ideally five times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. Pipelining introduces some overhead, so the throughput will not be quite as high as we might ideally desire, but pipelining nevertheless gives such great advantage for so little cost that all modern high-performance microprocessors are pipelined.

Reading and writing the memory and register file and using the ALU typically constitute the biggest delays in the processor. We choose five pipeline stages so that each stage involves exactly one of these slow steps. Specifically, we call the five stages *Fetch, Decode, Execute, Memory*, and *Writeback*. They are similar to the five steps that the multicycle processor used to perform `lw`. In the *Fetch* stage, the processor reads the instruction from instruction memory. In the *Decode* stage, the processor reads the source operands from the register file and decodes the instruction to produce the control signals. In the *Execute* stage, the processor performs a computation with the ALU. In the *Memory* stage, the processor reads or writes data memory. Finally, in the *Writeback* stage, the processor writes the result to the register file, when applicable.

Figure 7.43 shows a timing diagram comparing the single-cycle and pipelined processors. Time is on the horizontal axis, and instructions are on the vertical axis. The diagram assumes the logic element delays from Table 7.6 but ignores the delays of multiplexers and registers. In the single-cycle processor, Figure 7.43(a), the first instruction is read from memory at time 0; next the operands are read from the register file; and then the ALU executes the necessary computation. Finally, the data memory may be accessed, and the result is written back to the register file by 950 ps. The second instruction begins when the first completes. Hence, in this diagram, the single-cycle processor has an instruction latency of 250 + 150 + 200 + 250 + 100 = 950 ps and a throughput of 1 instruction per 950 ps (1.05 billion instructions per second).

In the pipelined processor, Figure 7.43(b), the length of a pipeline stage is set at 250 ps by the slowest stage, the memory access (in the Fetch or Memory stage). At time 0, the first instruction is fetched from memory. At 250 ps, the first instruction enters the Decode stage, and

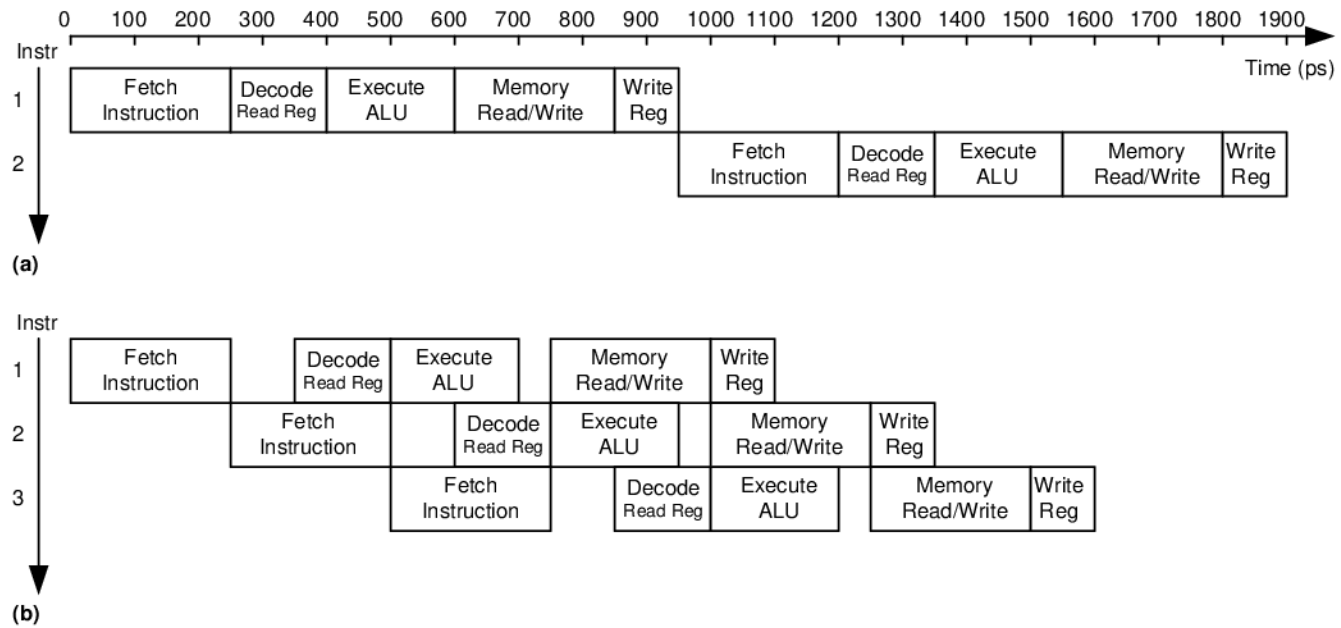

Figure 7.43 Timing diagrams: (a) single-cycle processor, (b) pipelined processor

a second instruction is fetched. At 500 ps, the first instruction executes, the second instruction enters the Decode stage, and a third instruction is fetched. And so forth, until all the instructions complete. The instruction latency is $5 \times 250 = 1250$ ps. The throughput is 1 instruction per 250 ps (4 billion instructions per second). Because the stages are not perfectly balanced with equal amounts of logic, the latency is slightly longer for the pipelined than for the single-cycle processor. Similarly, the throughput is not quite five times as great for a five-stage pipeline as for the single-cycle processor. Nevertheless, the throughput advantage is substantial.

Figure 7.44 shows an abstracted view of the pipeline in operation in which each stage is represented pictorially. Each pipeline stage is represented with its major component—instruction memory (IM), register file (RF) read, ALU execution, data memory (DM), and register file write-back—to illustrate the flow of instructions through the pipeline. Reading across a row shows the clock cycles in which a particular instruction is in each stage. For example, the `sub` instruction is fetched in cycle 3 and executed in cycle 5. Reading down a column shows what the various pipeline stages are doing on a particular cycle. For example, in cycle 6, the `or` instruction is being fetched from instruction memory, while `$s1` is being read from the register file, the ALU is computing `$t5` AND `$t6`, the data memory is idle, and the register file is writing a sum to `$s3`. Stages are shaded to indicate when they are used. For example, the data memory is used by `lw` in cycle 4 and by `sw` in cycle 8. The instruction memory and ALU are used in every cycle. The register file is written by

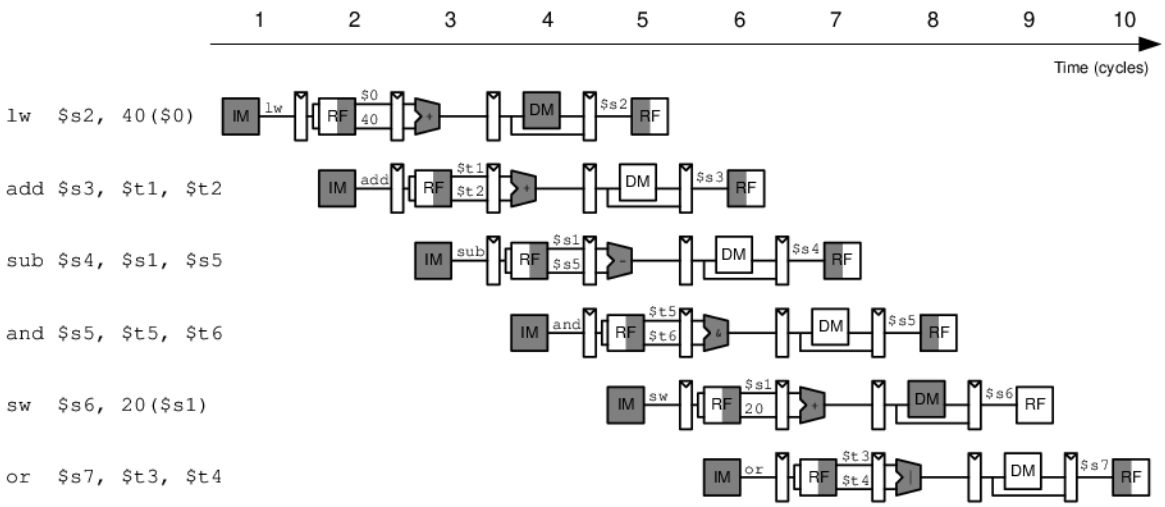

Figure 7.44 Abstract view of pipeline in operation

every instruction except `sw`. We assume that in the pipelined processor, the register file is written in the first part of a cycle and read in the second part, as suggested by the shading. This way, data can be written and read back within a single cycle.

A central challenge in pipelined systems is handling *hazards* that occur when the results of one instruction are needed by a subsequent instruction before the former instruction has completed. For example, if the `add` in Figure 7.44 used `$s2` rather than `$t2`, a hazard would occur because the `$s2` register has not been written by the `lw` by the time it is read by the `add`. This section explores *forwarding*, *stalls*, and *flushes* as methods to resolve hazards. Finally, this section revisits performance analysis considering sequencing overhead and the impact of hazards.

7.5.1 Pipelined Datapath

The pipelined datapath is formed by chopping the single-cycle datapath into five stages separated by pipeline registers. Figure 7.45(a) shows the single-cycle datapath stretched out to leave room for the pipeline registers. Figure 7.45(b) shows the pipelined datapath formed by inserting four pipeline registers to separate the datapath into five stages. The stages and their boundaries are indicated in blue. Signals are given a suffix (F, D, E, M, or W) to indicate the stage in which they reside.

The register file is peculiar because it is read in the Decode stage and written in the Writeback stage. It is drawn in the Decode stage, but the write address and data come from the Writeback stage. This feedback will lead to pipeline hazards, which are discussed in Section 7.5.3.

One of the subtle but critical issues in pipelining is that all signals associated with a particular instruction must advance through the pipeline in unison. Figure 7.45(b) has an error related to this issue. Can you find it?

The error is in the register file write logic, which should operate in the Writeback stage. The data value comes from *ResultW*, a Writeback stage signal. But the address comes from *WriteRegE*, an Execute stage signal. In the pipeline diagram of Figure 7.44, during cycle 5, the result of the `lw` instruction would be incorrectly written to register `$s4` rather than `$s2`.

Figure 7.46 shows a corrected datapath. The *WriteReg* signal is now pipelined along through the Memory and Writeback stages, so it remains in sync with the rest of the instruction. *WriteRegW* and *ResultW* are fed back together to the register file in the Writeback stage.

The astute reader may notice that the *PC′* logic is also problematic, because it might be updated with a Fetch or a Memory stage signal (*PCPlus4F* or *PCBranchM*). This control hazard will be fixed in Section 7.5.3.

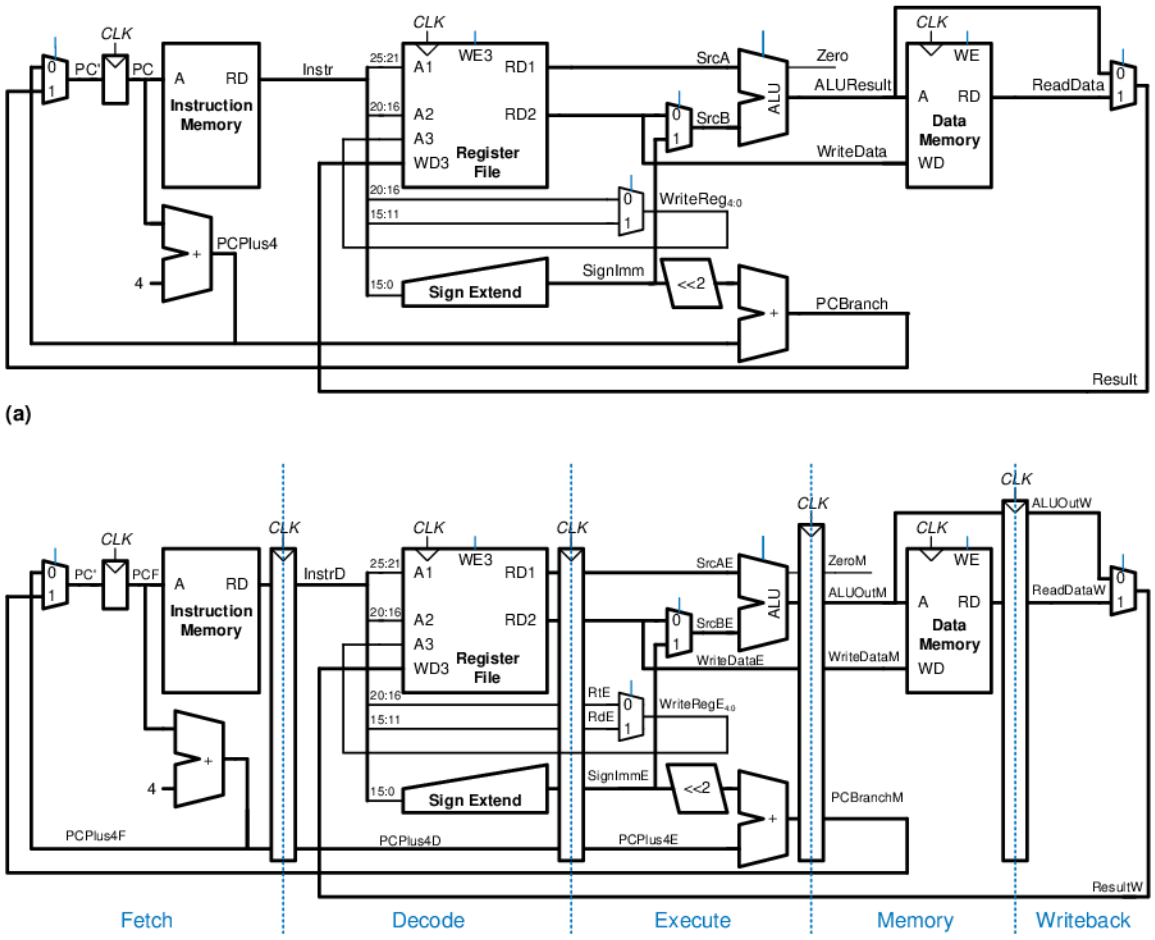

Figure 7.45 Single-cycle and pipelined datapaths

7.5.2 Pipelined Control

The pipelined processor takes the same control signals as the single-cycle processor and therefore uses the same control unit. The control unit examines the `opcode` and `funct` fields of the instruction in the Decode stage to produce the control signals, as was described in Section 7.3.2. These control signals must be pipelined along with the data so that they remain synchronized with the instruction.

The entire pipelined processor with control is shown in Figure 7.47. *RegWrite* must be pipelined into the Writeback stage before it feeds back to the register file, just as *WriteReg* was pipelined in Figure 7.46.

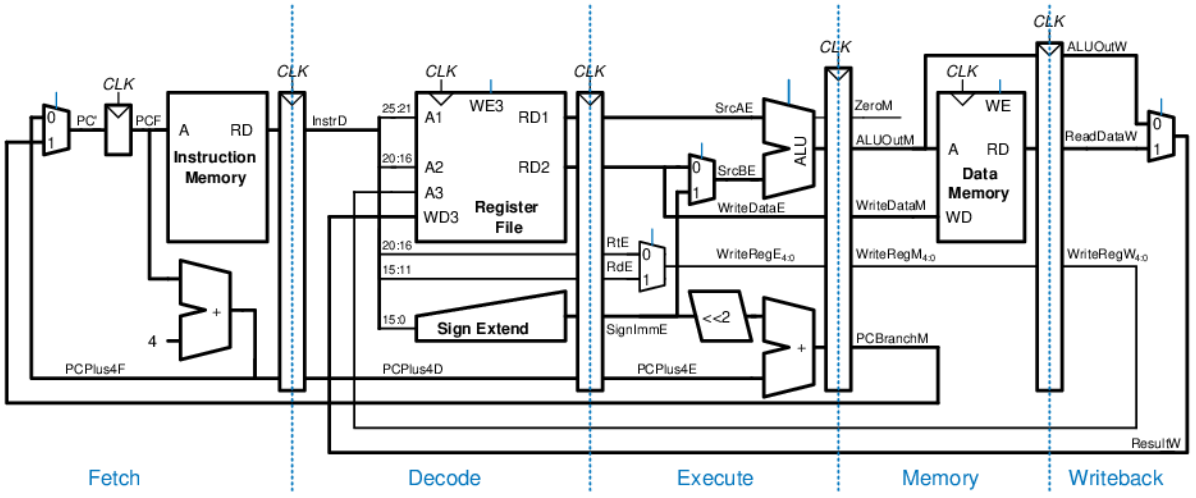

Figure 7.46 Corrected pipelined datapath

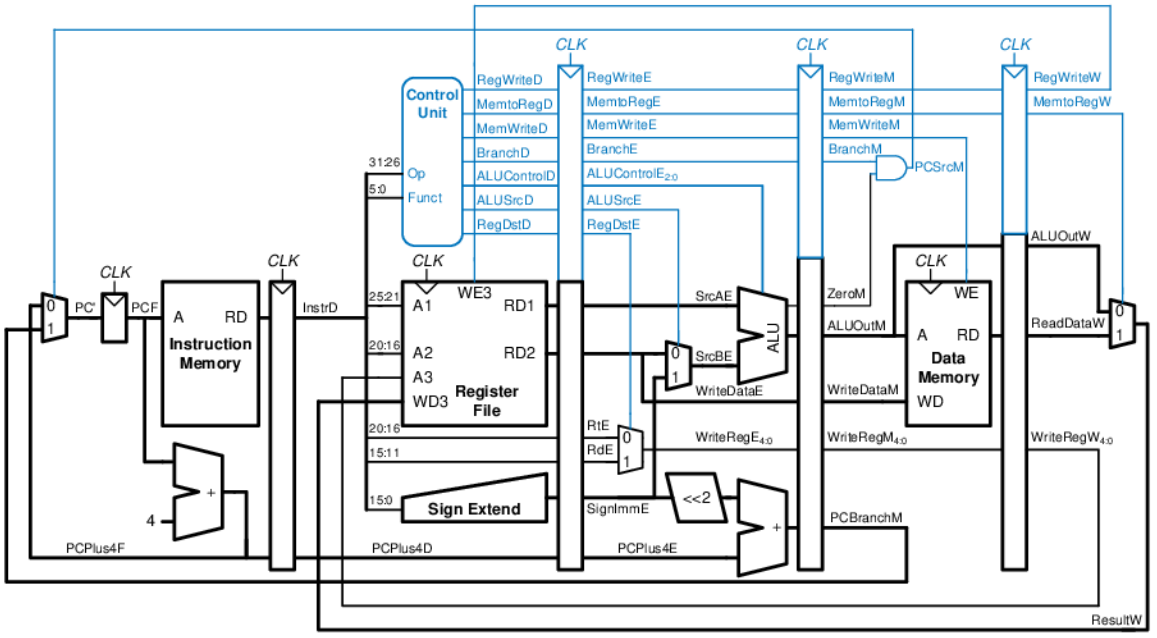

Figure 7.47 Pipelined processor with control

7.5.3 Hazards

In a pipelined system, multiple instructions are handled concurrently. When one instruction is *dependent* on the results of another that has not yet completed, a *hazard* occurs.

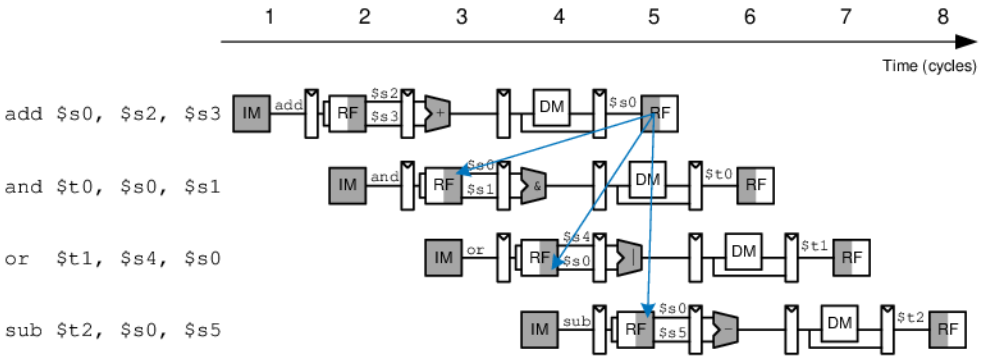

Figure 7.48 Abstract pipeline diagram illustrating hazards

The register file can be read and written in the same cycle. Let us assume that the write takes place during the first half of the cycle and the read takes place during the second half of the cycle, so that a register can be written and read back in the same cycle without introducing a hazard.

Figure 7.48 illustrates hazards that occur when one instruction writes a register (`$s0`) and subsequent instructions read this register. This is called a *read after write* (*RAW*) hazard. The `add` instruction writes a result into `$s0` in the first half of cycle 5. However, the `and` instruction reads `$s0` on cycle 3, obtaining the wrong value. The `or` instruction reads `$s0` on cycle 4, again obtaining the wrong value. The `sub` instruction reads `$s0` in the second half of cycle 5, obtaining the correct value, which was written in the first half of cycle 5. Subsequent instructions also read the correct value of `$s0`. The diagram shows that hazards may occur in this pipeline when an instruction writes a register and either of the two subsequent instructions read that register. Without special treatment, the pipeline will compute the wrong result.

On closer inspection, however, observe that the sum from the `add` instruction is computed by the ALU in cycle 3 and is not strictly needed by the `and` instruction until the ALU uses it in cycle 4. In principle, we should be able to forward the result from one instruction to the next to resolve the RAW hazard without slowing down the pipeline. In other situations explored later in this section, we may have to stall the pipeline to give time for a result to be computed before the subsequent instruction uses the result. In any event, something must be done to solve hazards so that the program executes correctly despite the pipelining.

Hazards are classified as data hazards or control hazards. A *data hazard* occurs when an instruction tries to read a register that has not yet been written back by a previous instruction. A *control hazard* occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place. In the remainder of this section, we will

enhance the pipelined processor with a hazard unit that detects hazards and handles them appropriately, so that the processor executes the program correctly.

Solving Data Hazards with Forwarding

Some data hazards can be solved by *forwarding* (also called *bypassing*) a result from the Memory or Writeback stage to a dependent instruction in the Execute stage. This requires adding multiplexers in front of the ALU to select the operand from either the register file or the Memory or Writeback stage. Figure 7.49 illustrates this principle. In cycle 4, `$s0` is forwarded from the Memory stage of the `add` instruction to the Execute stage of the dependent `and` instruction. In cycle 5, `$s0` is forwarded from the Writeback stage of the `add` instruction to the Execute stage of the dependent `or` instruction.

Forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or Writeback stage. Figure 7.50 modifies the pipelined processor to support forwarding. It adds a *hazard detection unit* and two forwarding multiplexers. The hazard detection unit receives the two source registers from the instruction in the Execute stage and the destination registers from the instructions in the Memory and Writeback stages. It also receives the *RegWrite* signals from the Memory and Writeback stages to know whether the destination register will actually be written (for example, the `sw` and `beq` instructions do not write results to the register file and hence do not need to have their results forwarded). Note that the *RegWrite* signals are *connected by name*. In other words, rather than cluttering up the diagram with long wires running from the control signals at the top to the hazard unit at the bottom, the connections are indicated by a short stub of wire labeled with the control signal name to which it is connected.

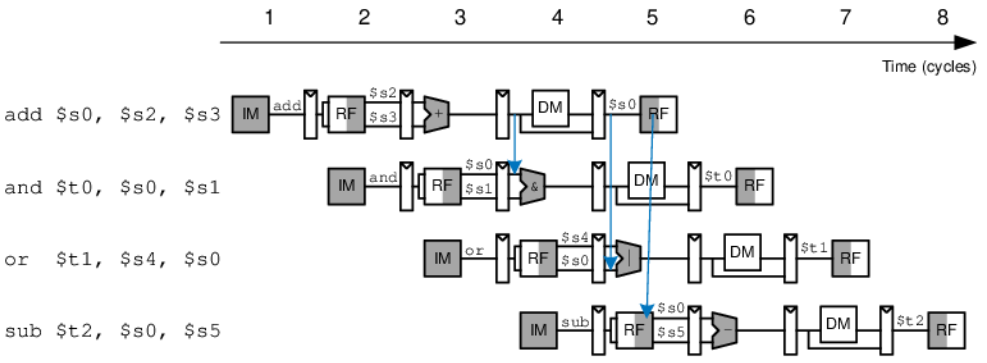

Figure 7.49 Abstract pipeline diagram illustrating forwarding

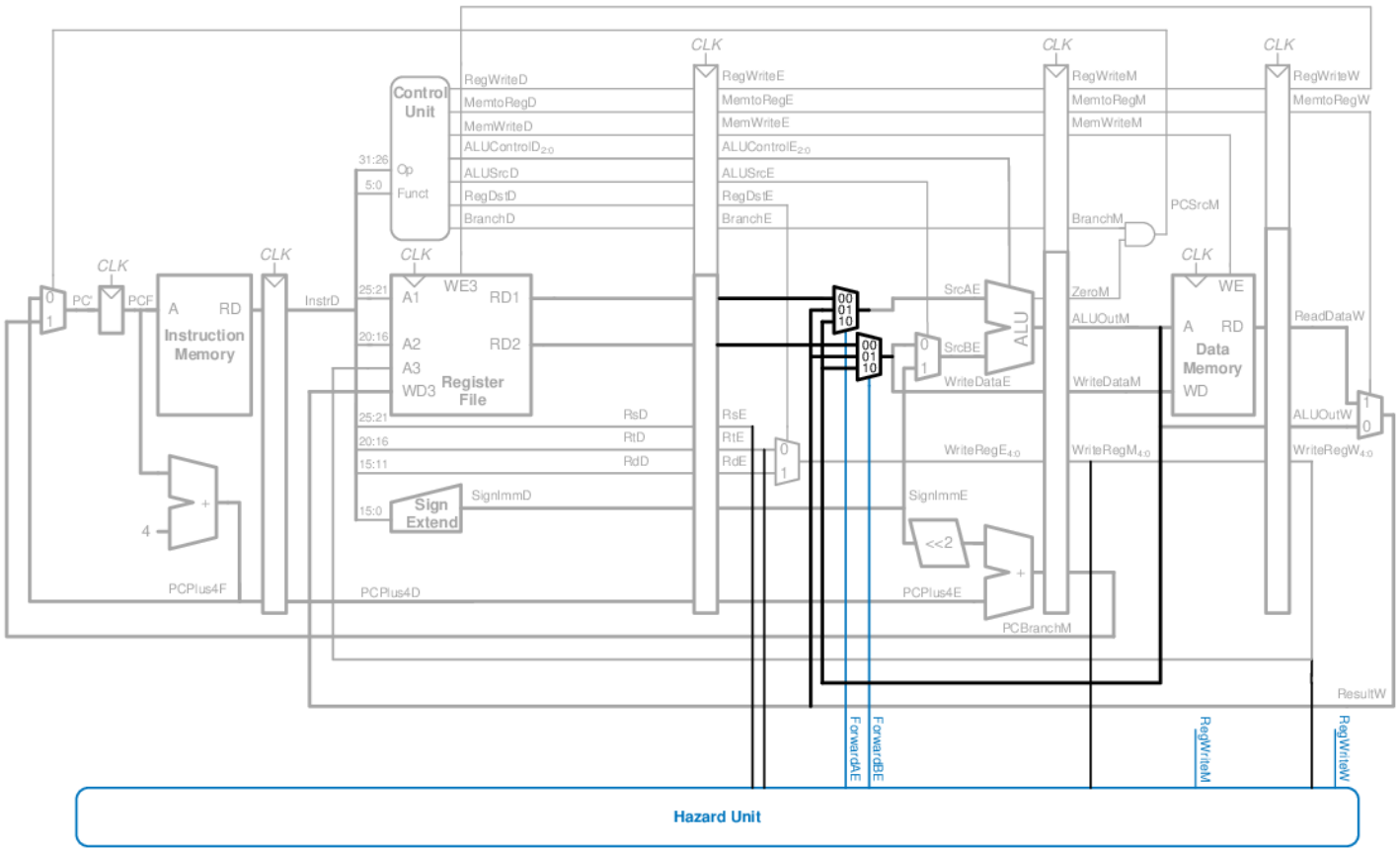

Figure 7.50 Pipelined processor with forwarding to solve hazards

The hazard detection unit computes control signals for the forwarding multiplexers to choose operands from the register file or from the results in the Memory or Writeback stage. It should forward from a stage if that stage will write a destination register and the destination register matches the source register. However, `$0` is hardwired to 0 and should never be forwarded. If both the Memory and Writeback stages contain matching destination registers, the Memory stage should have priority, because it contains the more recently executed instruction. In summary, the function of the forwarding logic for *SrcA* is given below. The forwarding logic for *SrcB* (*ForwardBE*) is identical except that it checks `rt` rather than `rs`.

```
if       ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
                                  ForwardAE = 10
else if  ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then
                                  ForwardAE = 01
else                              ForwardAE = 00
```

Solving Data Hazards with Stalls

Forwarding is sufficient to solve RAW data hazards when the result is computed in the Execute stage of an instruction, because its result can then be forwarded to the Execute stage of the next instruction. Unfortunately, the `lw` instruction does not finish reading data until the end of the Memory stage, so its result cannot be forwarded to the Execute stage of the next instruction. We say that the `lw` instruction has a *two-cycle latency*, because a dependent instruction cannot use its result until two cycles later. Figure 7.51 shows this problem. The `lw` instruction receives data from memory at the end of cycle 4. But the `and` instruction needs that data as a source operand at the beginning of cycle 4. There is no way to solve this hazard with forwarding.

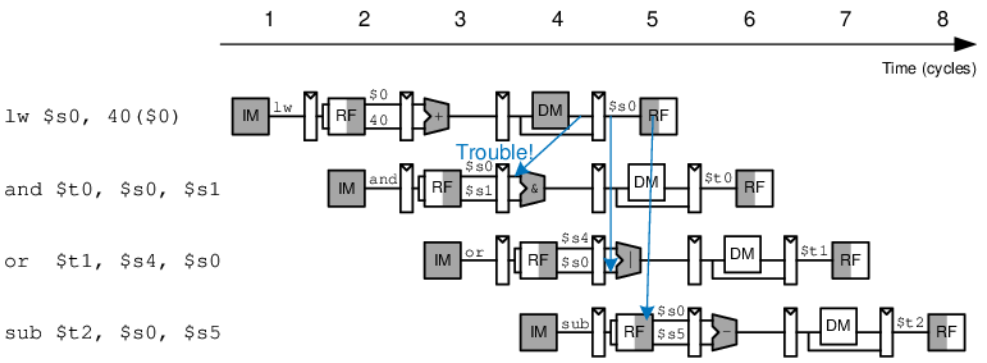

Figure 7.51 Abstract pipeline diagram illustrating trouble forwarding from `lw`

The alternative solution is to *stall* the pipeline, holding up operation until the data is available. Figure 7.52 shows stalling the dependent instruction (`and`) in the Decode stage. `and` enters the Decode stage in cycle 3 and stalls there through cycle 4. The subsequent instruction (`or`) must remain in the Fetch stage during both cycles as well, because the Decode stage is full.

In cycle 5, the result can be forwarded from the Writeback stage of `lw` to the Execute stage of `and`. In cycle 6, source `$s0` of the `or` instruction is read directly from the register file, with no need for forwarding.

Notice that the Execute stage is unused in cycle 4. Likewise, Memory is unused in Cycle 5 and Writeback is unused in cycle 6. This unused stage propagating through the pipeline is called a *bubble*, and it behaves like a `nop` instruction. The bubble is introduced by zeroing out the Execute stage control signals during a Decode stall so that the bubble performs no action and changes no architectural state.

In summary, stalling a stage is performed by disabling the pipeline register, so that the contents do not change. When a stage is stalled, all previous stages must also be stalled, so that no subsequent instructions are lost. The pipeline register directly after the stalled stage must be cleared to prevent bogus information from propagating forward. Stalls degrade performance, so they should only be used when necessary.

Figure 7.53 modifies the pipelined processor to add stalls for `lw` data dependencies. The hazard unit examines the instruction in the Execute stage. If it is `lw` and its destination register (`rtE`) matches either source operand of the instruction in the Decode stage (`rsD` or `rtD`), that instruction must be stalled in the Decode stage until the source operand is ready.

Stalls are supported by adding enable inputs (*EN*) to the Fetch and Decode pipeline registers and a synchronous reset/clear (*CLR*) input to the Execute pipeline register. When a `lw` stall occurs, *StallD* and *StallF*

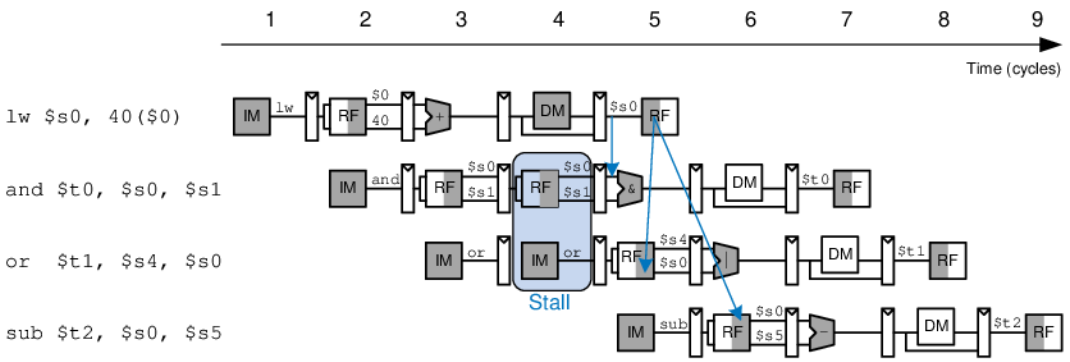

Figure 7.52 Abstract pipeline diagram illustrating stall to solve hazards

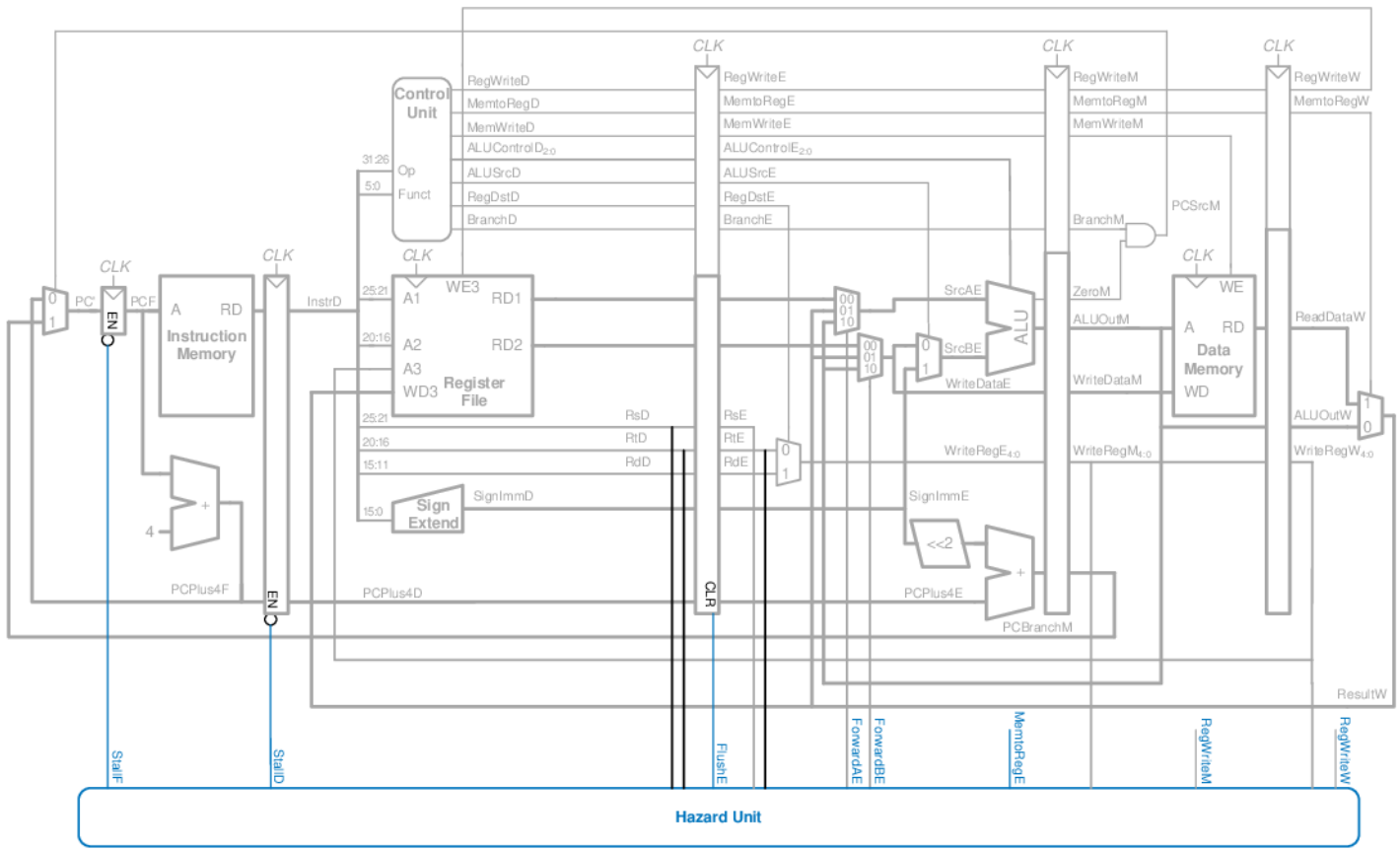

Figure 7.53 **Pipelined processor with stalls to solve `lw` data hazard**

are asserted to force the Decode and Fetch stage pipeline registers to hold their old values. *FlushE* is also asserted to clear the contents of the Execute stage pipeline register, introducing a bubble.[4]

The *MemtoReg* signal is asserted for the `lw` instruction. Hence, the logic to compute the stalls and flushes is

```
lwstall = ((rsD  == rtE) OR (rtD == rtE)) AND MemtoRegE
StallF  = StallD = FlushE = lwstall
```

Solving Control Hazards

The `beq` instruction presents a control hazard: the pipelined processor does not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction is fetched.

One mechanism for dealing with the control hazard is to stall the pipeline until the branch decision is made (i.e., *PCSrc* is computed). Because the decision is made in the Memory stage, the pipeline would have to be stalled for three cycles at every branch. This would severely degrade the system performance.

An alternative is to predict whether the branch will be taken and begin executing instructions based on the prediction. Once the branch decision is available, the processor can throw out the instructions if the prediction was wrong. In particular, suppose that we predict that branches are not taken and simply continue executing the program in order. If the branch should have been taken, the three instructions following the branch must be *flushed* (discarded) by clearing the pipeline registers for those instructions. These wasted instruction cycles are called the *branch misprediction penalty*.

Figure 7.54 shows such a scheme, in which a branch from address 20 to address 64 is taken. The branch decision is not made until cycle 4, by which point the `and`, `or`, and `sub` instructions at addresses 24, 28, and 2C have already been fetched. These instructions must be flushed, and the `slt` instruction is fetched from address 64 in cycle 5. This is somewhat of an improvement, but flushing so many instructions when the branch is taken still degrades performance.

We could reduce the branch misprediction penalty if the branch decision could be made earlier. Making the decision simply requires comparing the values of two registers. Using a dedicated equality comparator is much faster than performing a subtraction and zero detection. If the comparator is fast enough, it could be moved back into the Decode stage, so that the operands are read from the register file and compared to determine the next PC by the end of the Decode stage.

---

[4] Strictly speaking, only the register designations (*RsE*, *RtE*, and *RdE*) and the control signals that might update memory or architectural state (*RegWrite*, *MemWrite*, and *Branch*) need to be cleared; as long as these signals are cleared, the bubble can contain random data that has no effect.

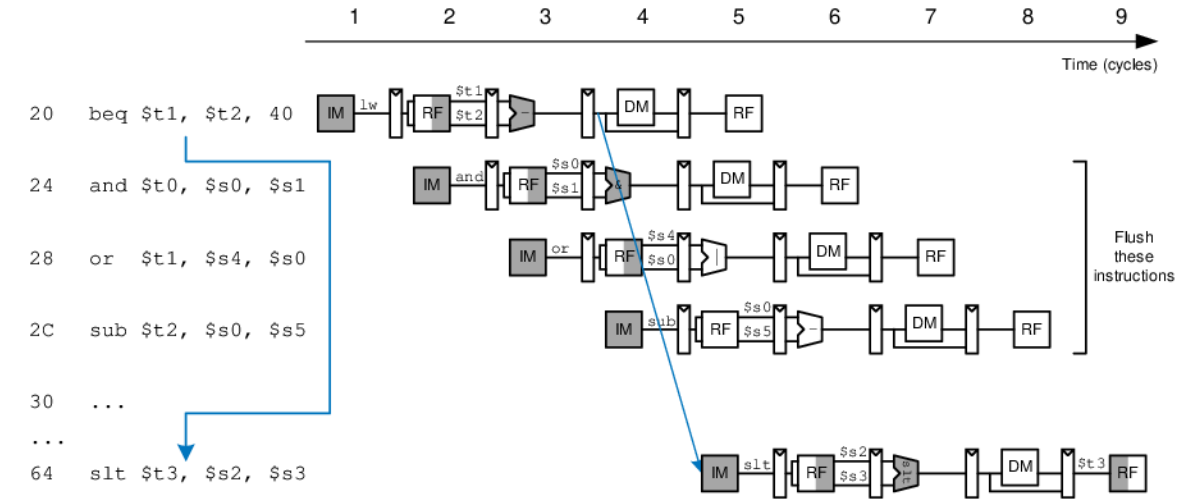

Figure 7.54 Abstract pipeline diagram illustrating flushing when a branch is taken

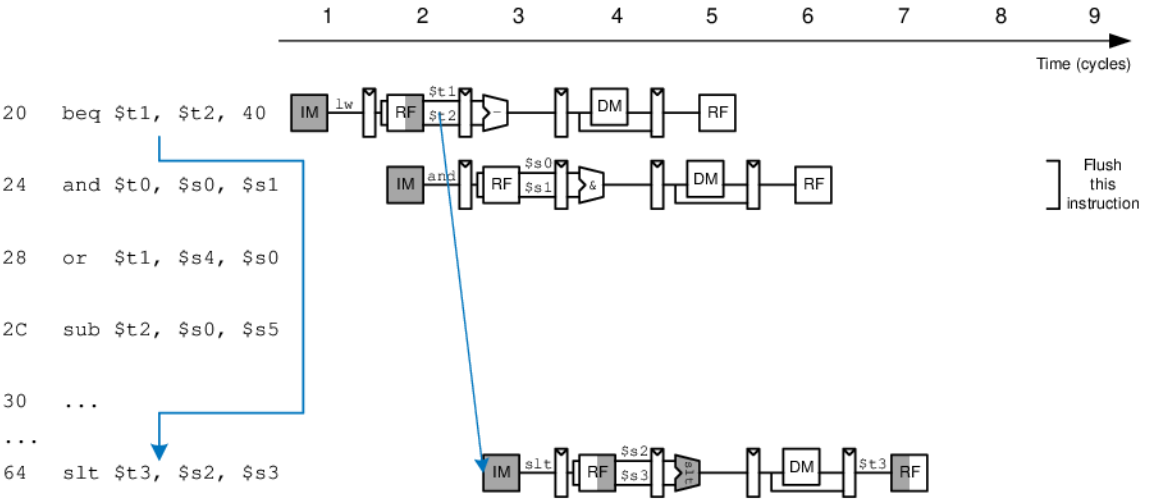

Figure 7.55 Abstract pipeline diagram illustrating earlier branch decision

Figure 7.55 shows the pipeline operation with the early branch decision being made in cycle 2. In cycle 3, the `and` instruction is flushed and the `slt` instruction is fetched. Now the branch misprediction penalty is reduced to only one instruction rather than three.

Figure 7.56 modifies the pipelined processor to move the branch decision earlier and handle control hazards. An equality comparator is added to the Decode stage and the *PCSrc* AND gate is moved earlier, so

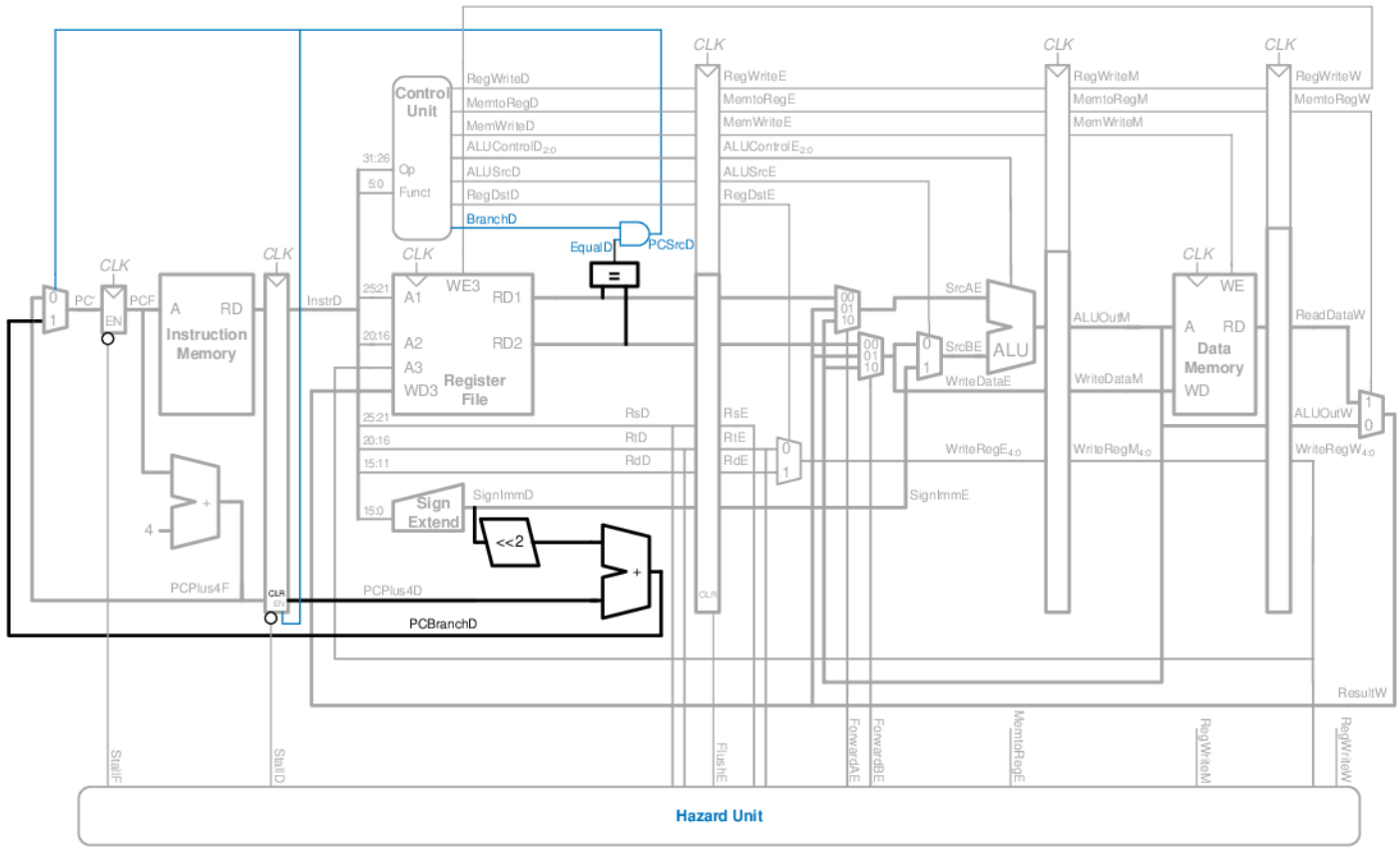

Figure 7.56 Pipelined processor handling branch control hazard

that *PCSrc* can be determined in the Decoder stage rather than the Memory stage. The *PCBranch* adder must also be moved into the Decode stage so that the destination address can be computed in time. The synchronous clear input (*CLR*) connected to *PCSrcD* is added to the Decode stage pipeline register so that the incorrectly fetched instruction can be flushed when a branch is taken.

Unfortunately, the early branch decision hardware introduces a new RAW data hazard. Specifically, if one of the source operands for the branch was computed by a previous instruction and has not yet been written into the register file, the branch will read the wrong operand value from the register file. As before, we can solve the data hazard by forwarding the correct value if it is available or by stalling the pipeline until the data is ready.

Figure 7.57 shows the modifications to the pipelined processor needed to handle the Decode stage data dependency. If a result is in the Writeback stage, it will be written in the first half of the cycle and read during the second half, so no hazard exists. If the result of an ALU instruction is in the Memory stage, it can be forwarded to the equality comparator through two new multiplexers. If the result of an ALU instruction is in the Execute stage or the result of a `lw` instruction is in the Memory stage, the pipeline must be stalled at the Decode stage until the result is ready.

The function of the Decode stage forwarding logic is given below.

```
ForwardAD = (rsD != 0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD != 0) AND (rtD == WriteRegM) AND RegWriteM
```

The function of the stall detection logic for a branch is given below. The processor must make a branch decision in the Decode stage. If either of the sources of the branch depends on an ALU instruction in the Execute stage or on a `lw` instruction in the Memory stage, the processor must stall until the sources are ready.

```
branchstall =
  BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)
                            OR
  BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)
```

Now the processor might stall due to either a load or a branch hazard:

```
StallF = StallD = FlushE = lwstall OR branchstall
```

Hazard Summary

In summary, RAW data hazards occur when an instruction depends on the result of another instruction that has not yet been written into the register file. The data hazards can be resolved by forwarding if the result is computed soon enough; otherwise, they require stalling the pipeline until the result is available. Control hazards occur when the decision of what instruction to fetch has not been made by the time the next instruction must be fetched. Control hazards are solved by predicting which

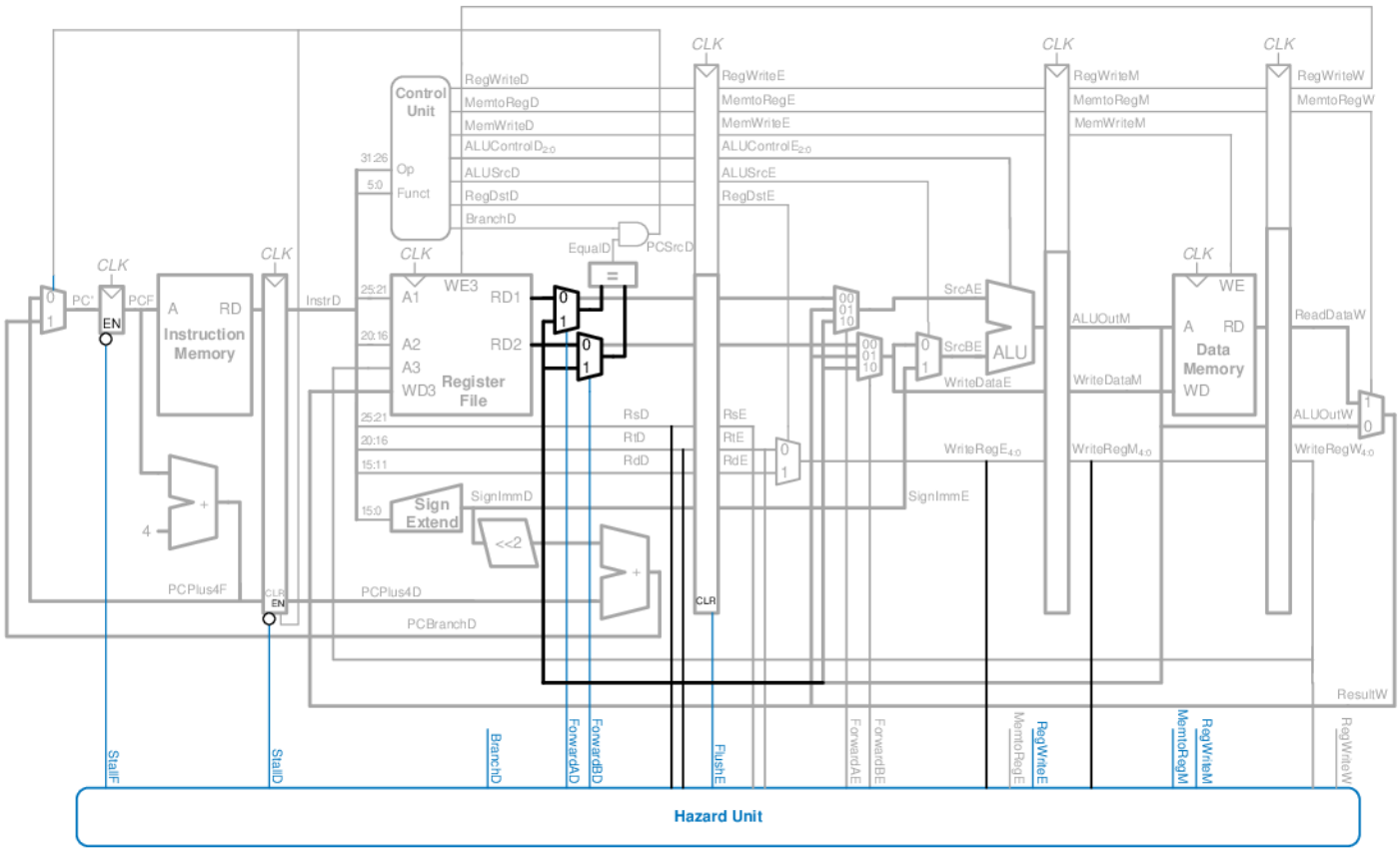

Figure 7.57 Pipelined processor handling data dependencies for branch instructions

instruction should be fetched and flushing the pipeline if the prediction is later determined to be wrong. Moving the decision as early as possible minimizes the number of instructions that are flushed on a misprediction. You may have observed by now that one of the challenges of designing a pipelined processor is to understand all the possible interactions between instructions and to discover all the hazards that may exist. Figure 7.58 shows the complete pipelined processor handling all of the hazards.

7.5.4 More Instructions

Supporting new instructions in the pipelined processor is much like supporting them in the single-cycle processor. However, new instructions may introduce hazards that must be detected and solved.

In particular, supporting `addi` and `j` instructions on the pipelined processor requires enhancing the controller, exactly as was described in Section 7.3.3, and adding a jump multiplexer to the datapath after the branch multiplexer. Like a branch, the jump takes place in the Decode stage, so the subsequent instruction in the Fetch stage must be flushed. Designing this flush logic is left as Exercise 7.29.

7.5.5 Performance Analysis

The pipelined processor ideally would have a CPI of 1, because a new instruction is issued every cycle. However, a stall or a flush wastes a cycle, so the CPI is slightly higher and depends on the specific program being executed.

Example 7.9 PIPELINED PROCESSOR CPI

The SPECINT2000 benchmark considered in Example 7.7 consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions. Assume that 40% of the loads are immediately followed by an instruction that uses the result, requiring a stall, and that one quarter of the branches are mispredicted, requiring a flush. Assume that jumps always flush the subsequent instruction. Ignore other hazards. Compute the average CPI of the pipelined processor.

Solution: The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of time that instruction is used. Loads take one clock cycle when there is no dependency and two cycles when the processor must stall for a dependency, so they have a CPI of $(0.6)(1) + (0.4)(2) = 1.4$. Branches take one clock cycle when they are predicted properly and two when they are not, so they have a CPI of $(0.75)(1) + (0.25)(2) = 1.25$. Jumps always have a CPI of 2. All other instructions have a CPI of 1. Hence, for this benchmark, Average CPI $= (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1) = 1.15$.

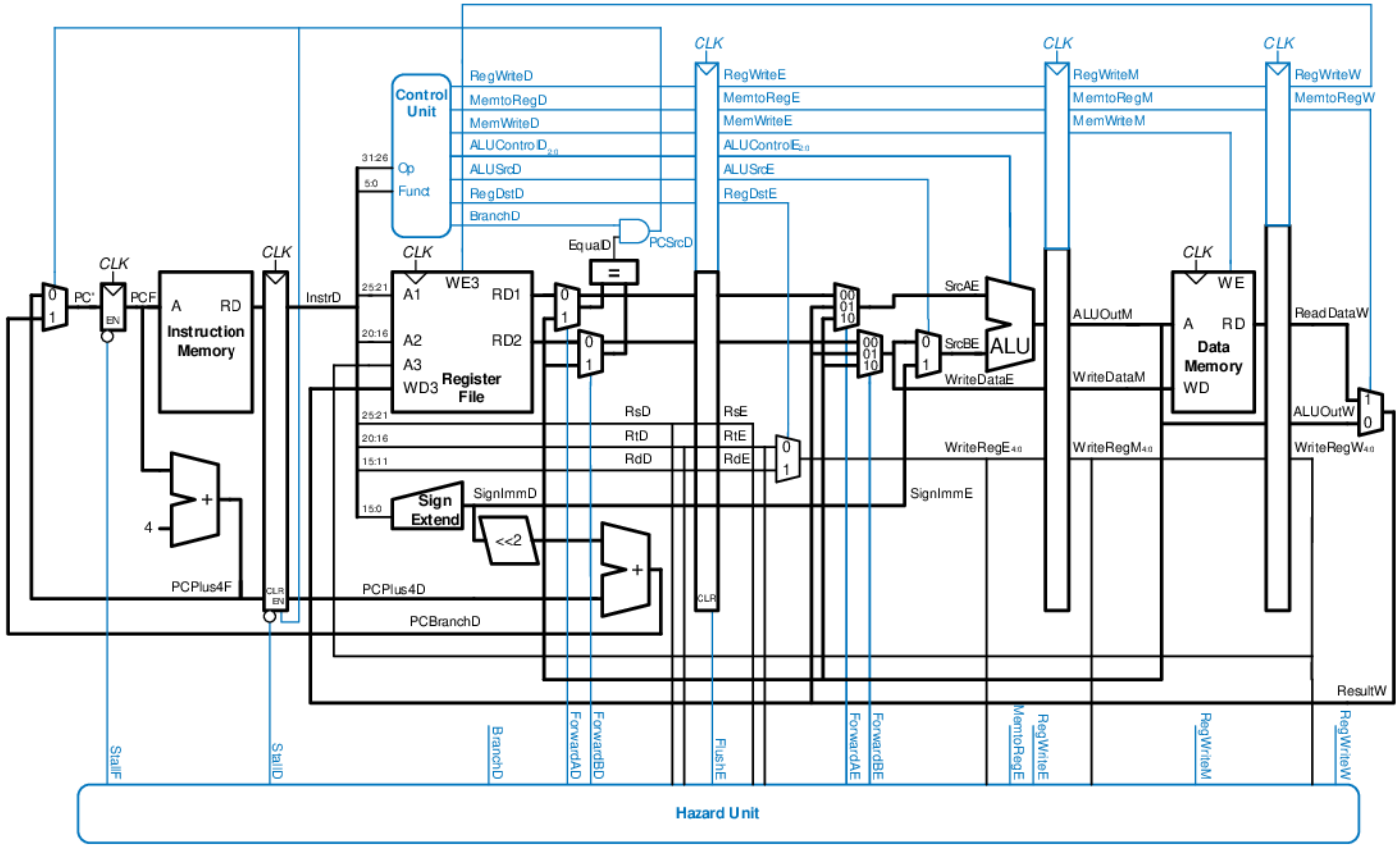

Figure 7.58 Pipelined processor with full hazard handling

We can determine the cycle time by considering the critical path in each of the five pipeline stages shown in Figure 7.58. Recall that the register file is written in the first half of the Writeback cycle and read in the second half of the Decode cycle. Therefore, the cycle time of the Decode and Writeback stages is twice the time necessary to do the half-cycle of work.

$$T_c = max\left\{\begin{array}{cc} t_{pcq} + t_{\text{mem}} + t_{\text{setup}} & \text{Fetch} \\ 2(t_{RF\text{read}} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + t_{\text{mux}} + t_{\text{setup}}) & \text{Decode} \\ t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}} & \text{Execute} \\ t_{pcq} + t_{\text{memwrite}} + t_{\text{setup}} & \text{Memory} \\ 2(t_{pcq} + t_{\text{mux}} + t_{RF\text{write}}) & \text{Writeback} \end{array}\right\} \quad (7.5)$$

Example 7.10 PROCESSOR PERFORMANCE COMPARISON

Ben Bitdiddle needs to compare the pipelined processor performance to that of the single-cycle and multicycle processors considered in Example 7.8. Most of the logic delays were given in Table 7.6. The other element delays are 40 ps for an equality comparator, 15 ps for an AND gate, 100 ps for a register file write, and 220 ps for a memory write. Help Ben compare the execution time of 100 billion instructions from the SPECINT2000 benchmark for each processor.

Solution: According to Equation 7.5, the cycle time of the pipelined processor is T_{c3} = max[30 + 250 + 20, 2(150 + 25 + 40 + 15 + 25 + 20), 30 + 25 + 25 + 200 + 20, 30 + 220 + 20, 2(30 + 25 + 100)] = 550 ps. According to Equation 7.1, the total execution time is T_3 = (100 × 10^9 instructions)(1.15 cycles/ instruction)(550 × 10^{-12} s/cycle) = 63.3 seconds. This compares to 95 seconds for the single-cycle processor and 133.9 seconds for the multicycle processor.

The pipelined processor is substantially faster than the others. However, its advantage over the single-cycle processor is nowhere near the five-fold speedup one might hope to get from a five-stage pipeline. The pipeline hazards introduce a small CPI penalty. More significantly, the sequencing overhead (clk-to-Q and setup times) of the registers applies to every pipeline stage, not just once to the overall datapath. Sequencing overhead limits the benefits one can hope to achieve from pipelining.

The careful reader might observe that the Decode stage is substantially slower than the others, because the register file write, read, and branch comparison must all happen in half a cycle. Perhaps moving the branch comparison to the Decode stage was not such a good idea. If branches were resolved in the Execute stage instead, the CPI would increase slightly, because a mispredict would flush two instructions, but the cycle time would decrease substantially, giving an overall speedup.

The pipelined processor is similar in hardware requirements to the single-cycle processor, but it adds a substantial number of pipeline registers, along with multiplexers and control logic to resolve hazards.

7.6 HDL REPRESENTATION*

This section presents HDL code for the single-cycle MIPS processor supporting all of the instructions discussed in this chapter, including `addi` and `j`. The code illustrates good coding practices for a moderately complex system. HDL code for the multicycle processor and pipelined processor are left to Exercises 7.22 and 7.33.

In this section, the instruction and data memories are separated from the main processor and connected by address and data busses. This is more realistic, because most real processors have external memory. It also illustrates how the processor can communicate with the outside world.

The processor is composed of a datapath and a controller. The controller, in turn, is composed of the main decoder and the ALU decoder. Figure 7.59 shows a block diagram of the single-cycle MIPS processor interfaced to external memories.

The HDL code is partitioned into several sections. Section 7.6.1 provides HDL for the single-cycle processor datapath and controller. Section 7.6.2 presents the generic building blocks, such as registers and multiplexers, that are used by any microarchitecture. Section 7.6.3

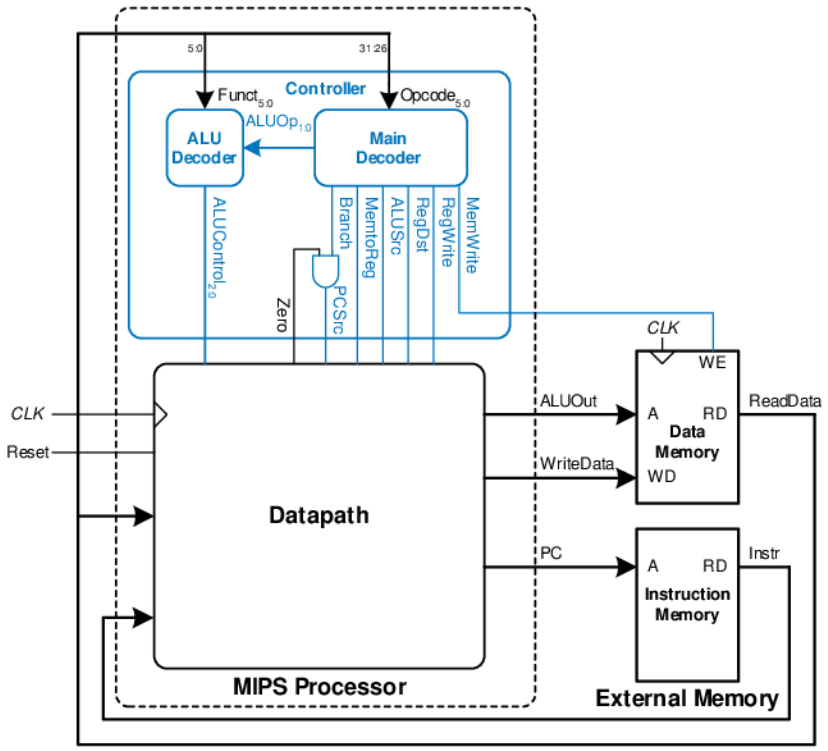

Figure 7.59 MIPS single-cycle processor interfaced to external memory

introduces the testbench and external memories. The HDL is available in electronic form on the this book's Web site (see the preface).

7.6.1 Single-Cycle Processor

The main modules of the single-cycle MIPS processor module are given in the following HDL examples.

HDL Example 7.1 SINGLE-CYCLE MIPS PROCESSOR

Verilog

```
module mips(input          clk, reset,
            output [31:0]  pc,
            input  [31:0]  instr,
            output         memwrite,
            output [31:0]  aluout, writedata,
            input  [31:0]  readdata);

  wire       memtoreg, branch,
             alusrc, regdst, regwrite, jump;
  wire [2:0] alucontrol;

  controller c(instr[31:26], instr[5:0], zero,
               memtoreg, memwrite, pcsrc,
               alusrc, regdst, regwrite, jump,
               alucontrol);
  datapath dp(clk, reset, memtoreg, pcsrc,
              alusrc, regdst, regwrite, jump,
              alucontrol,
              zero, pc, instr,
              aluout, writedata, readdata);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mips is -- single cycle MIPS processor
  port(clk, reset:        in  STD_LOGIC;
       pc:                out STD_LOGIC_VECTOR(31 downto 0);
       instr:             in  STD_LOGIC_VECTOR(31 downto 0);
       memwrite:          out STD_LOGIC;
       aluout, writedata: out STD_LOGIC_VECTOR(31 downto 0);
       readdata:          in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of mips is
  component controller
    port(op, funct:         in  STD_LOGIC_VECTOR(5 downto 0);
         zero:              in  STD_LOGIC;
         memtoreg, memwrite: out STD_LOGIC;
         pcsrc, alusrc:     out STD_LOGIC;
         regdst, regwrite:  out STD_LOGIC;
         jump:              out STD_LOGIC;
         alucontrol:        out STD_LOGIC_VECTOR(2 downto 0));
 end component;
 component datapath
   port(clk, reset:        in  STD_LOGIC;
        memtoreg, pcsrc:   in  STD_LOGIC;
        alusrc, regdst:    in  STD_LOGIC;
        regwrite, jump:    in  STD_LOGIC;
        alucontrol:        in  STD_LOGIC_VECTOR(2 downto 0);
        zero:              out STD_LOGIC;
        pc:                buffer STD_LOGIC_VECTOR(31 downto 0);
        instr:             in  STD_LOGIC_VECTOR(31 downto 0);
        aluout, writedata: buffer STD_LOGIC_VECTOR(31 downto 0);
        readdata:          in  STD_LOGIC_VECTOR(31 downto 0));
 end component;
 signal memtoreg, alusrc, regdst, regwrite, jump, pcsrc:
  STD_LOGIC;
 signal zero: STD_LOGIC;
 signal alucontrol: STD_LOGIC_VECTOR(2 downto 0);
begin
  cont: controller port map(instr(31 downto 26), instr
                            (5 downto 0), zero, memtoreg,
                            memwrite, pcsrc, alusrc, regdst,
                            regwrite, jump, alucontrol);
  dp: datapath port map(clk, reset, memtoreg, pcsrc, alusrc,
                        regdst, regwrite, jump, alucontrol,
                        zero, pc, instr, aluout, writedata,
                        readdata);
end;
```

HDL Example 7.2 CONTROLLER

Verilog

```
module controller(input  [5:0] op, funct,
                  input        zero,
                  output       memtoreg, memwrite,
                  output       pcsrc, alusrc,
                  output       regdst, regwrite,
                  output       jump,
                  output [2:0] alucontrol);

  wire [1:0] aluop;
  wire       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump,
             aluop);
  aludec ad(funct, aluop, alucontrol);

  assign pcsrc = branch & zero;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- single cycle control decoder
 port(op, funct:           in  STD_LOGIC_VECTOR(5 downto 0);
      zero:                in  STD_LOGIC;
      memtoreg, memwrite: out STD_LOGIC;
      pcsrc, alusrc:      out STD_LOGIC;
      regdst, regwrite:   out STD_LOGIC;
      jump:               out STD_LOGIC;
      alucontrol:         out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture struct of controller is
 component maindec
   port(op:                 in  STD_LOGIC_VECTOR(5 downto 0);
        memtoreg, memwrite: out STD_LOGIC;
        branch, alusrc:     out STD_LOGIC;
        regdst, regwrite:   out STD_LOGIC;
        jump:               out STD_LOGIC;
        aluop:              out STD_LOGIC_VECTOR(1 downto 0));
 end component;
 component aludec
   port(funct:      in  STD_LOGIC_VECTOR(5 downto 0);
        aluop:      in  STD_LOGIC_VECTOR(1 downto 0);
        alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
 end component;
 signal aluop:  STD_LOGIC_VECTOR(1 downto 0);
 signal branch: STD_LOGIC;
begin
 md: maindec port map(op, memtoreg, memwrite, branch,
                      alusrc, regdst, regwrite, jump, aluop);
 ad: aludec port map(funct, aluop, alucontrol);

 pcsrc <= branch and zero;
end;
```

HDL Example 7.3 MAIN DECODER

Verilog

```
module maindec(input  [5:0] op,
               output       memtoreg, memwrite,
               output       branch, alusrc,
               output       regdst, regwrite,
               output       jump,
               output [1:0] aluop);

 reg [8:0] controls;

 assign {regwrite, regdst, alusrc,
         branch, memwrite,
         memtoreg, jump, aluop} = controls;

 always@(*)
  case(op)
    6'b000000: controls <= 9'b110000010; //Rtyp
    6'b100011: controls <= 9'b101001000; //LW
    6'b101011: controls <= 9'b001010000; //SW
    6'b000100: controls <= 9'b000100001; //BEQ
    6'b001000: controls <= 9'b101000000; //ADDI
    6'b000010: controls <= 9'b000000100; //J
    default:   controls <= 9'bxxxxxxxxx; //???
  endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity maindec is -- main control decoder
 port(op:                in  STD_LOGIC_VECTOR(5 downto 0);
      memtoreg, memwrite: out STD_LOGIC;
      branch, alusrc:    out STD_LOGIC;
      regdst, regwrite:  out STD_LOGIC;
      jump:              out STD_LOGIC;
      aluop:             out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture behave of maindec is
 signal controls: STD_LOGIC_VECTOR(8 downto 0);
begin
 process(op) begin
  case op is
     when "000000" => controls <= "110000010"; -- Rtyp
     when "100011" => controls <= "101001000"; -- LW
     when "101011" => controls <= "001010000"; -- SW
     when "000100" => controls <= "000100001"; -- BEQ
     when "001000" => controls <= "101000000"; -- ADDI
     when "000010" => controls <= "000000100"; -- J
     when others   => controls <= "---------"; -- illegal op
  end case;
 end process;

 regwrite <= controls(8);
 regdst   <= controls(7);
 alusrc   <= controls(6);
 branch   <= controls(5);
 memwrite <= controls(4);
 memtoreg <= controls(3);
 jump     <= controls(2);
 aluop    <= controls(1 downto 0);
end;
```

HDL Example 7.4 ALU DECODER

Verilog

```
module aludec(input      [5:0] funct,
              input      [1:0] aluop,
              output reg [2:0] alucontrol);

always@(*)
 case(aluop)
  2'b00: alucontrol <= 3'b010; // add
  2'b01: alucontrol <= 3'b110; // sub
  default: case(funct)         // RTYPE
      6'b100000: alucontrol <= 3'b010; // ADD
      6'b100010: alucontrol <= 3'b110; // SUB
      6'b100100: alucontrol <= 3'b000; // AND
      6'b100101: alucontrol <= 3'b001; // OR
      6'b101010: alucontrol <= 3'b111; // SLT
      default:   alucontrol <= 3'bxxx; // ???
    endcase
  endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity aludec is -- ALU control decoder
  port(funct:      in  STD_LOGIC_VECTOR(5 downto 0);
       aluop:      in  STD_LOGIC_VECTOR(1 downto 0);
       alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture behave of aludec is
begin
 process(aluop, funct) begin
  case aluop is
    when "00" => alucontrol <= "010"; -- add (for lb/sb/addi)
    when "01" => alucontrol <= "110"; -- sub (for beq)
    when others => case funct is      -- R-type instructions
                     when "100000" => alucontrol <=
                       "010"; -- add
                     when "100010" => alucontrol <=
                       "110"; -- sub
                     when "100100" => alucontrol <=
                       "000"; -- and
                     when "100101" => alucontrol <=
                       "001"; -- or
                     when "101010" => alucontrol <=
                       "111"; -- slt
                     when others   => alucontrol <=
                       "---"; -- ???
                   end case;
  end case;
 end process;
end;
```

HDL Example 7.5 DATAPATH

Verilog

```
module datapath(input          clk, reset,
                input          memtoreg, pcsrc,
                input          alusrc, regdst,
                input          regwrite, jump,
                input   [2:0]  alucontrol,
                output         zero,
                output  [31:0] pc,
                input   [31:0] instr,
                output  [31:0] aluout, writedata,
                input   [31:0] readdata);

  wire [4:0]  writereg;
  wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  wire [31:0] signimm, signimmsh;
  wire [31:0] srca, srcb;
  wire [31:0] result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder       pcadd1(pc, 32'b100, pcplus4);
  sl2         immsh(signimm, signimmsh);
  adder       pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc,
                      pcnextbr);
  mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00},
                    jump, pcnext);

  // register file logic
  regfile     rf(clk, regwrite, instr[25:21],
                 instr[20:16], writereg,
                 result, srca, writedata);
  mux2 #(5)   wrmux(instr[20:16], instr[15:11],
                    regdst, writereg);
  mux2 #(32)  resmux(aluout, readdata,
                     memtoreg, result);
  signext     se(instr[15:0], signimm);

  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc,
                      srcb);
  alu         alu(srca, srcb, alucontrol,
                  aluout, zero);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all; use
IEEE.STD_LOGIC_ARITH.all;
entity datapath is -- MIPS datapath
  port(clk, reset:        in     STD_LOGIC;
       memtoreg, pcsrc:   in     STD_LOGIC;
       alusrc, regdst:    in     STD_LOGIC;
       regwrite, jump:    in     STD_LOGIC;
       alucontrol:        in     STD_LOGIC_VECTOR(2 downto 0);
       zero:              out    STD_LOGIC;
       pc:                buffer STD_LOGIC_VECTOR(31 downto 0);
       instr:             in     STD_LOGIC_VECTOR(31 downto 0);
       aluout, writedata: buffer STD_LOGIC_VECTOR(31 downto 0);
       readdata:          in     STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of datapath is
  component alu
    port(a, b:       in  STD_LOGIC_VECTOR(31 downto 0);
         alucontrol: in  STD_LOGIC_VECTOR(2 downto 0);
         result:     buffer STD_LOGIC_VECTOR(31 downto 0);
         zero:       out STD_LOGIC);
  end component;
  component regfile
    port(clk:            in  STD_LOGIC;
         we3:            in  STD_LOGIC;
         ra1, ra2, wa3: in  STD_LOGIC_VECTOR(4 downto 0);
         wd3:            in  STD_LOGIC_VECTOR(31 downto 0);
         rd1, rd2:       out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:    out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component sl2
    port(a: in  STD_LOGIC_VECTOR(31 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component signext
    port(a: in  STD_LOGIC_VECTOR(15 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flopr generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux2 generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
 end component;
 signal writereg: STD_LOGIC_VECTOR(4 downto 0);
 signal pcjump, pcnext, pcnextbr,
        pcplus4, pcbranch: STD_LOGIC_VECTOR(31 downto 0);
 signal signimm, signimmsh: STD_LOGIC_VECTOR(31 downto 0);
 signal srca, srcb, result: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- next PC logic
  pcjump <= pcplus4(31 downto 28) & instr(25 downto 0) & "00";
  pcreg: flopr generic map(32) port map(clk, reset, pcnext, pc);
  pcadd1: adder port map(pc, X"00000004", pcplus4);
  immsh: sl2 port map(signimm, signimmsh);
  pcadd2: adder port map(pcplus4, signimmsh, pcbranch);
  pcbrmux: mux2 generic map(32) port map(pcplus4, pcbranch,
                                         pcsrc, pcnextbr);
  pcmux: mux2 generic map(32) port map(pcnextbr, pcjump, jump,
                                       pcnext);

  -- register file logic
  rf: regfile port map(clk, regwrite, instr(25 downto 21),
                       instr(20 downto 16), writereg, result, srca,
                       writedata);
  wrmux: mux2 generic map(5) port map(instr(20 downto 16),
  instr(15 downto 11), regdst, writereg);
  resmux: mux2 generic map(32) port map(aluout, readdata,
                                        memtoreg, result);
  se: signext port map(instr(15 downto 0), signimm);

  -- ALU logic
  srcbmux: mux2 generic map (32) port map(writedata, signimm,
                                          alusrc, srcb);
  mainalu: alu port map(srca, srcb, alucontrol, aluout, zero);
end;
```

7.6.2 Generic Building Blocks

This section contains generic building blocks that may be useful in any MIPS microarchitecture, including a register file, adder, left shift unit, sign-extension unit, resettable flip-flop, and multiplexer. The HDL for the ALU is left to Exercise 5.9.

HDL Example 7.6 REGISTER FILE

Verilog

```
module regfile(input         clk,
               input         we3,
               input  [4:0]  ra1, ra2, wa3,
               input  [31:0] wd3,
               output [31:0] rd1, rd2);

  reg [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clock
  // register 0 hardwired to 0

  always @(posedge clk)
    if(we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity regfile is -- three-port register file
  port(clk:          in  STD_LOGIC;
       we3:          in  STD_LOGIC;
       ra1, ra2, wa3: in  STD_LOGIC_VECTOR(4 downto 0);
       wd3:          in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:     out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array(31 downto 0) of STD_LOGIC_VECTOR(31
                        downto 0);
  signal mem: ramtype;
begin
  -- three-ported register file
  -- read two ports combinationally
  -- write third port on rising edge of clock
  process(clk) begin
   if clk'event and clk = '1' then
    if we3 = '1' then mem(CONV_INTEGER(wa3)) <= wd3;
    end if;
   end if;
  end process;
  process(ra1, ra2) begin
    if(conv_integer(ra1) = 0) then rd1 <= X"00000000";
       -- register 0 holds 0
    else rd1 <= mem(CONV_INTEGER(ra1));
    end if;
    if(conv_integer(ra2) = 0) then rd2 <= X"00000000";
    else rd2 <= mem(CONV_INTEGER(ra2));
    end if;
  end process;
end;
```

HDL Example 7.7 ADDER

Verilog

```
module adder(input  [31:0] a, b,
             output [31:0] y);

  assign y = a + b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity adder is -- adder
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin
  y <= a + b;
end;
```

HDL Example 7.8 LEFT SHIFT (MULTIPLY BY 4)

Verilog

```
module sl2 (input  [31:0] a,
            output [31:0] y);

  // shift left by 2
  assign y = {a[29:01], 2'b00};
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity sl2 is -- shift left by 2
  port (a: in  STD_LOGIC_VECTOR (31 downto 0);
        y: out STD_LOGIC_VECTOR (31 downto 0));
end;

architecture behave of sl2 is
begin
  y <= a(29 downto 0) & "00";
end;
```

HDL Example 7.9 SIGN EXTENSION

Verilog

```
module signext (input  [15:0] a,
                output [31:0] y);

  assign y = {{16{a[15]}}, a};
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity signext is -- sign extender
port(a: in  STD_LOGIC_VECTOR (15 downto 0);
     y: out STD_LOGIC_VECTOR (31 downto 0));
end;

architecture behave of signext is
begin
  y <= X"0000" & a when a(15) = '0' else X"ffff" & a;
end;
```

HDL Example 7.10 RESETTABLE FLIP-FLOP

Verilog

```
module flopr #(parameter WIDTH = 8)
              (input                  clk, reset,
               input      [WIDTH-1:0] d,
               output reg [WIDTH-1:0] q);

  always @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all; use
IEEE.STD_LOGIC_ARITH.all;
entity flopr is -- flip-flop with synchronous reset
  generic (width: integer);
  port (clk, reset: in   STD_LOGIC;
        d:          in   STD_LOGIC_VECTOR(width-1 downto 0);
        q:          out  STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopr is
begin
  process (clk, reset) begin
    if reset = '1' then q <= CONV_STD_LOGIC_VECTOR(0, width);
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end;
```