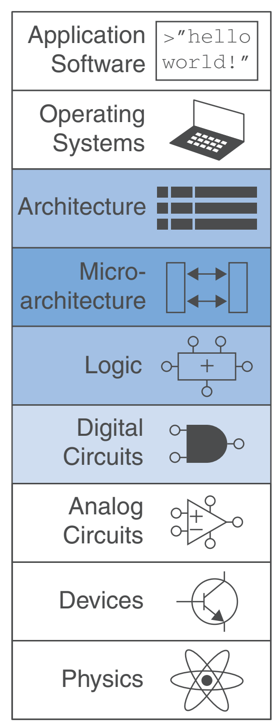
Microarchitecture

7

7.1 Introduction
7.2 Performance Analysis
7.3 Single-Cycle Processor
7.4 Multicycle Processor
7.5 Pipelined Processor
7.6 HDL Representation*
7.7 Advanced Microarchitecture*
7.8 Real-World Perspective: Evolution of RISC-V Microarchitecture*
7.9 Summary
Exercises
Interview Questions



7.1 INTRODUCTION

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

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

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

7.1.1 Architectural State and Instruction Set

Recall that a computer architecture is defined by its instruction set and architectural state. The *architectural state* for the RISC-V processor consists of the program counter and the 32 32-bit registers. Any RISC-V microarchitecture must contain all of this state. Based on the current architectural state, the processor executes a particular instruction with a particular set of data to produce a new architectural state. Some

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00007-6

The *architectural state* is the information necessary to define what a computer is doing. If one were to save a copy of the architectural state and contents of memory, then turn off a computer, then turn it back on and restore the architectural state and memory, the computer would resume the program it was running, unaware that it had been powered off and back on. Think of a science fiction novel in which the protagonist's brain is frozen, then thawed years later to wake up in a new world.

microarchitectures contain additional *nonarchitectural state* to either simplify the logic or improve performance; we point this out as it arises.

To keep the microarchitectures easy to understand, we focus on a subset of the RISC-V instruction set. Specifically, we handle the following instructions:

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

These particular instructions were chosen because they are sufficient to write useful programs. Once you understand how to implement these instructions, you can expand the hardware to handle others.

7.1.2 Design Process

We divide our microarchitectures into two interacting parts: the *datapath* and the *control unit*. The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers. We are implementing the 32-bit RISC-V (RV32I) architecture, so we use a 32-bit datapath. The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

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

At the very least, the program counter must have a reset signal to initialize its value when the processor turns on. Upon reset, RISC-V processors normally initialize the PC to a low address in memory, such as 0x00001000, and we start our programs there.

In this chapter, heavy lines indicate 32-bit data busses. Medium lines indicate narrower busses, such as the 5-bit address busses on the register file. Narrow lines indicate 1-bit wires, and blue lines are used for control signals, such as the register file write enable. Registers usually have a reset input to put them into a known state at start-up, but reset is not shown to reduce clutter.

The *program counter* (PC) points to the current instruction. Its input, *PCNext*, indicates the address of the next instruction.

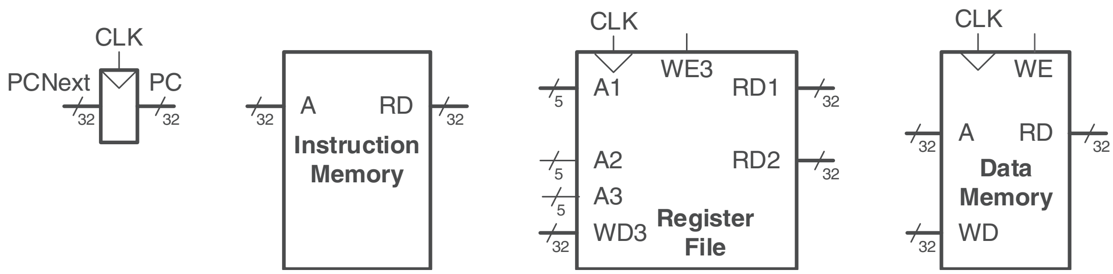


Figure 7.1 State elements of a RISC-V processor

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

The 32-element × 32-bit register file holds registers `x0-x31`. Recall that `x0` is hardwired to 0. The register file has two read ports and one write port. The read ports take 5-bit address inputs, *A1* and *A2*, each specifying one of the $2^5 = 32$ registers as source operands. The register file places the 32-bit register values onto read data outputs *RD1* and *RD2*. The write port, port 3, takes a 5-bit address input, *A3*; a 32-bit write data input, *WD3*; a write enable input, *WE3*; and a clock. If its write enable (*WE3*) is asserted, then the register file writes the data (*WD3*) into the specified register (*A3*) on the rising edge of the clock.

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

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

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

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

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

7.1.3 Microarchitectures

In this chapter, we develop three microarchitectures for the RISC-V architecture: single-cycle, multicycle, and pipelined. They differ in how the state elements are connected and in the amount of nonarchitectural state needed.

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

The *multicycle microarchitecture* executes instructions in a series of shorter cycles. Simpler instructions execute in fewer cycles than complicated ones. Moreover, the multicycle microarchitecture reduces the hardware cost by reusing expensive hardware blocks, such as adders and memories. For example, the adder may be used on different cycles for several purposes while carrying out a single instruction. The multicycle microprocessor accomplishes this by introducing several nonarchitectural registers to hold intermediate results. The multicycle processor executes only one instruction at a time, but each instruction takes multiple clock cycles. This processor requires only a single memory, accessing it on one cycle to fetch the instruction and on another to read or write data. Because they use less hardware than single-cycle processors, multicycle processors were the historical choice for inexpensive systems.

The *pipelined microarchitecture* applies pipelining to the single-cycle microarchitecture. It therefore can execute several instructions simultaneously, improving the throughput significantly. Pipelining must add logic to handle dependencies between simultaneously executing instructions. It also requires nonarchitectural pipeline registers. Pipelined processors must access instructions and data in the same cycle; they generally use separate instruction and data caches for this purpose, as discussed in Chapter 8. The added logic and registers are worthwhile; all commercial high-performance processors use pipelining today.

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

Examples of classic multicycle processors include the 1947 MIT Whirlwind, the IBM System/360, the Digital Equipment Corporation VAX, the 6502 used in the Apple II, and the 8088 used in the IBM PC. Multicycle microarchitectures are still used in inexpensive microcontrollers such as the 8051, the 68HC11, and the PIC16-series found in appliances, toys, and gadgets.

Intel processors have been pipelined since the 80486 was introduced in 1989. Nearly all RISC microprocessors are also pipelined, and all commercial RISC-V processors have been pipelined. Because of the decreasing cost of transistors, pipelined processors now cost fractions of a penny, and the entire system, with memory and peripherals, costs 10's of cents. Thus, pipelined processors are replacing their slower multicycle siblings in even the most cost-sensitive applications.

7.2 PERFORMANCE ANALYSIS

As we mentioned, a particular processor architecture can have many microarchitectures with different cost and performance trade-offs. The cost depends on the amount of hardware required and the implementation technology. Precise cost calculations require detailed knowledge of the implementation technology but, in general, more gates and more memory mean more dollars.

This section lays the foundation for analyzing performance. There are many ways to measure the performance of a computer system, and marketing departments are infamous for choosing the method that makes their computer look fastest, regardless of whether the measurement has any correlation to real-world performance. For example, microprocessor makers often market their products based on the clock frequency and the number of cores. However, they gloss over the complications that some processors accomplish more work than others in a clock cycle and that this varies from program to program. What is a buyer to do?

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

Dhrystone, CoreMark, and SPEC are three popular benchmarks. The first two are *synthetic benchmarks* composed of important common pieces of programs. Dhrystone was developed in 1984 and remains commonly used for embedded processors, although the code is somewhat unrepresentative of real-life programs. CoreMark is an improvement over Dhrystone and involves matrix multiplications that exercise the multiplier and adder, linked lists to exercise the memory system, state machines to exercise the branch logic, and cyclical redundancy checks that involve many parts of the processor. Both benchmarks are less than 16 KB in size and do not stress the instruction cache.

The SPECspeed 2017 Integer benchmark from the Standard Performance Evaluation Corporation (SPEC) is composed of real programs, including x264 (video compression), deepsjeng (an artificial intelligence chess player), omnetpp (simulation), and GCC (a C compiler). The benchmark is widely used for high-performance processors because it stresses the entire system in a representative way.

When customers buy computers based on benchmarks, they must be careful because computer makers have strong incentive to bias the benchmark. For example, Dhrystone involves extensive string copying, but the strings are of known constant length and word alignment. Thus, a smart compiler may replace the usual code involving loops and byte accesses with a series of word loads and stores, improving Dhrystone scores by more than 30% but not speeding up real-world applications. The SPEC89 benchmark contained a Matrix 300 program in which 99% of the execution time was in one line. IBM sped up the program by a factor of 9 using a compiler technique called *blocking*. Benchmarking multicore computing is even harder because there are many ways to write programs, some of which speed up in proportion to the number of cores available but are inefficient on a single core. Others are fast on a single core but scarcely benefit from extra cores.

Equation 7.1 gives the execution time of a program, measured in seconds.

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

The number of instructions in a program depends on the processor architecture. Some architectures have complicated instructions that do more work per instruction, thus reducing the number of instructions in a program. However, these complicated instructions are often slower to execute in hardware. The number of instructions also depends enormously on the cleverness of the programmer. For the purposes of this chapter, we assume that we are executing known programs on a RISC-V processor, so the number of instructions for each program is constant, independent of the microarchitecture. The *cycles per instruction* (*CPI*) is the number of clock cycles required to execute an average instruction. It is the reciprocal of the throughput (*instructions per cycle*, or *IPC*). Different microarchitectures have different CPIs. In this chapter, we assume we have an ideal memory system that does not affect the CPI. In Chapter 8, we examine how the processor sometimes has to wait for the memory, which increases the CPI.

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

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

Many other factors affect overall computer performance. For example, the hard disk, the memory, the graphics system, and the network connection may be limiting factors that make processor performance irrelevant. The fastest microprocessor in the world does not help surfing the Internet on a poor connection. But these other factors are beyond the scope of this book.

7.3 SINGLE-CYCLE PROCESSOR

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

7.3.1 Sample Program

For the sake of concreteness, we will have the single-cycle processor run the short program from Figure 7.2 that exercises loads, stores, an R-type instruction (`or`), and a branch (`beq`). Suppose that the program is stored in memory starting at address 0x1000. The figure indicates the address of each instruction, the instruction type, the instruction fields, and the hexadecimal machine language code for the instruction.

Assume that register `x5` initially contains the value 6 and `x9` contains 0x2004. Memory location 0x2000 contains the value 10. The program counter begins at 0x1000. The `lw` reads 10 from address (0x2004 – 4) = 0x2000 and puts it in `x6`. The `sw` writes 10 to address (0x2004 + 8) = 0x200C. The `or` computes `x4` $= 6 \mid 10 = 0110_2 \mid 1010_2 = 1110_2 = 14$. Then, `beq` goes back to label `L7`, so the program repeats forever.

7.3.2 Single-Cycle Datapath

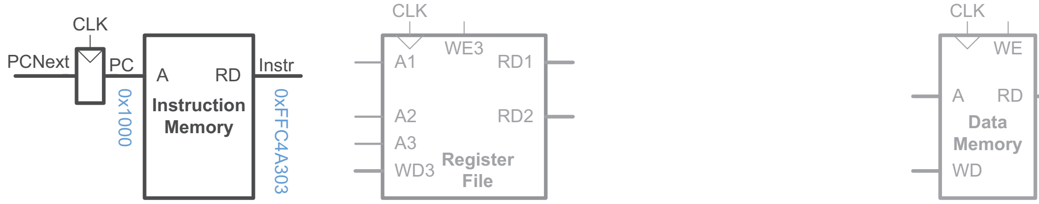
This section gradually develops the single-cycle datapath, adding one piece at a time to the state elements from Figure 7.1. The new connections are emphasized in black (or blue, for new control signals), whereas the hardware that has already been studied is shown in gray. The example instruction being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

The program counter contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. Figure 7.3 shows that the PC is simply connected to the address input of the instruction memory. The instruction memory reads out, or *fetches*, the 32-bit instruction, labeled *Instr*. In our sample program from Figure 7.2, PC is 0x1000. (Note that this is a 32-bit processor, so PC is really 0x00001000, but we omit leading zeros to avoid cluttering the figure.)

We italicize signal names in the text but not the names of hardware modules. For example, *PC* is the signal coming out of the PC register, or simply, the PC.

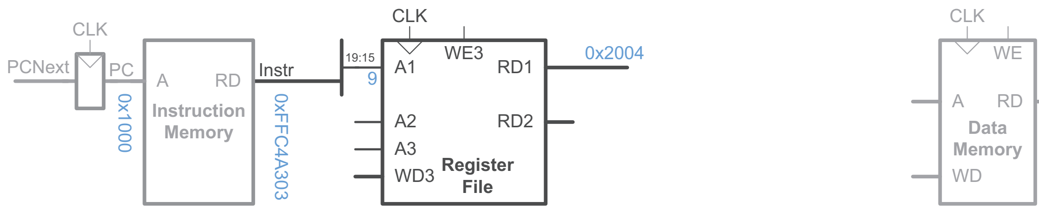
Address	Instruction	Type	Fields						Machine Language
			imm$_{11:0}$		rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100		01001	010	00110	0000011	FFC4A303
			imm$_{11:5}$	rs2	rs1	f3	imm$_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233
			imm$_{12,10:5}$	rs2	rs1	f3	imm$_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Figure 7.2 Sample program exercising different types of instructions



Address	Instruction	Type	Fields				Machine Language	
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.3 Fetch instruction from memory



Address	Instruction	Type	Fields				Machine Language	
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.4 Read source operand from register file

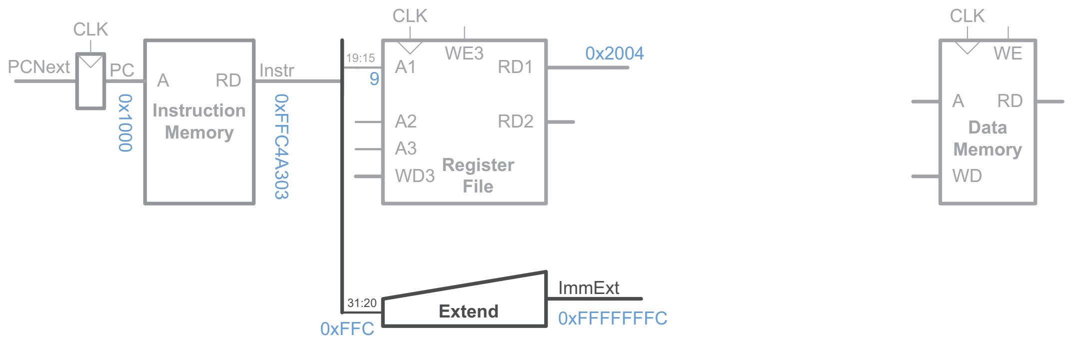
Instr is `lw`, 0xFFC4A303, as also shown at the bottom of Figure 7.3. These sample values are annotated in light blue on the diagram.

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

`lw`

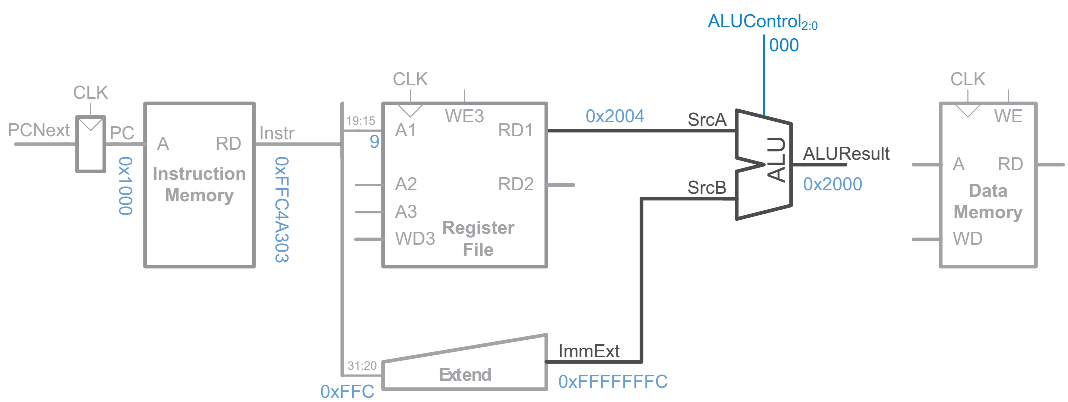
For the `lw` instruction, the next step is to read the source register containing the base address. Recall that `lw` is an I-type instruction, and the base register is specified in the **rs1** field of the instruction, $Instr_{19:15}$. These bits of the instruction connect to the *A1* address input of the register file, as shown in Figure 7.4. The register file reads the register value onto *RD1*. In our example, the register file reads 0x2004 from `x9`.

The `lw` instruction also requires an offset. The offset is stored in the 12-bit immediate field of the instruction, $Instr_{31:20}$. It is a signed value, so it must be sign-extended to 32 bits. Sign extension simply means copying the sign bit into the most significant bits: $ImmExt_{31:12} = Instr_{31}$, and $ImmExt_{11:0} = Instr_{31:20}$. Sign-extension is performed by



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.5 Sign-extend the immediate

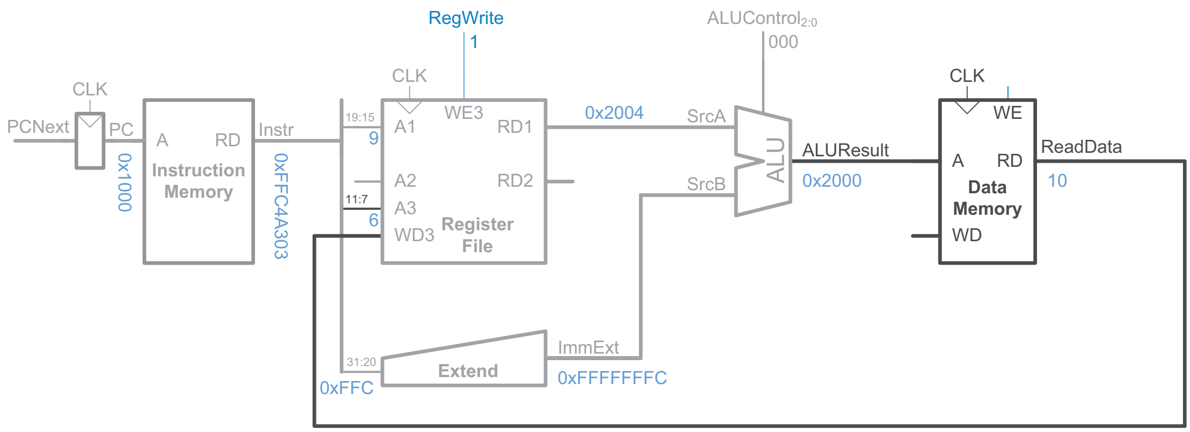


Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.6 Compute memory address

an Extend unit, as shown in Figure 7.5, which receives the 12-bit signed immediate in $Instr_{31:20}$ and produces the 32-bit sign-extended immediate, *ImmExt*. In our example, the two's complement immediate −4 is extended from its 12-bit representation 0xFFC to a 32-bit representation 0xFFFFFFFC.

The processor adds the base address to the offset to find the address to read from memory. Figure 7.6 introduces an ALU to perform this addition. The ALU receives two operands, *SrcA* and *SrcB*. *SrcA* is the base address from the register file, and *SrcB* is the offset from the sign-extended immediate, *ImmExt*. The ALU can perform many operations, as was described in Section 5.2.4. The 3-bit *ALUControl* signal



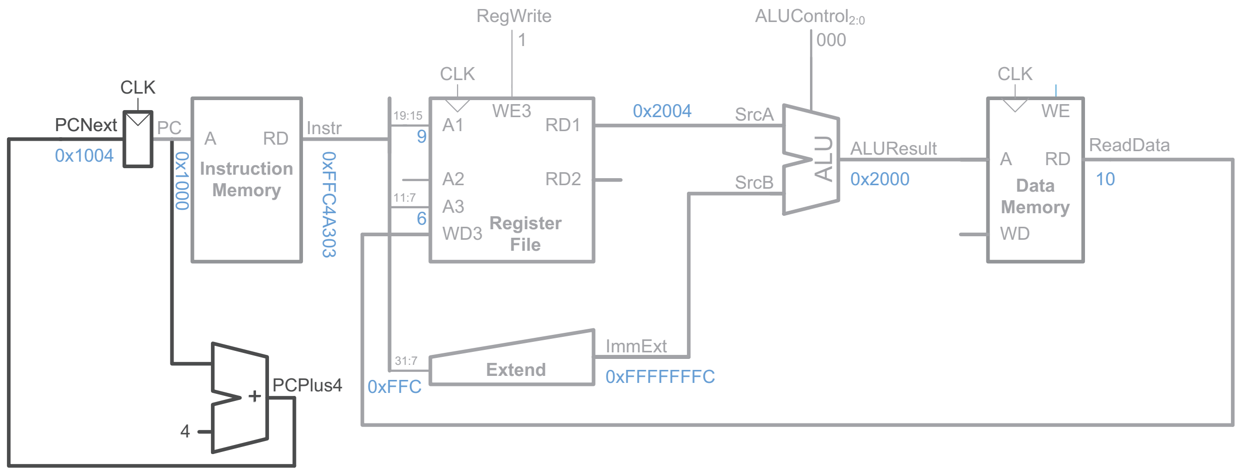
Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.7 Read memory and write result back to register file

specifies the operation (see Table 5.3 on page 250). The ALU receives 32-bit operands and generates a 32-bit *ALUResult*. For the `lw` instruction, *ALUControl* should be set to 000 to perform addition. *ALUResult* is sent to the data memory as the address to read, as shown in Figure 7.6. In our example, the ALU computes 0x2004 + 0xFFFFFFFC = 0x2000. Again, this is a 32-bit value, but we omit the leading zeros to avoid cluttering the figure.

This memory address from the ALU is provided to the address (*A*) port of the data memory. The data is read from the data memory onto the *ReadData* bus and then written back to the destination register at the end of the cycle, as shown in Figure 7.7. Port 3 of the register file is the write port. `lw`'s destination register, indicated by the **rd** field ($Instr_{11:7}$), is connected to *A3*, port 3's address input. The *ReadData* bus is connected to *WD3*, port 3's write data input. A control signal called *RegWrite* (register write) is connected to *WE3*, port 3's write enable input, and is asserted during the `lw` instruction so that the data value is written into the register file. The write takes place on the rising edge of the clock at the end of the cycle. In our example, the processor reads 10 from address 0x2000 in the data memory and puts that value (10) into `x6` in the register file.

While the instruction is being executed, the processor must also compute the address of the next instruction, *PCNext*. Because instructions are 32 bits (4 bytes), the next instruction is at PC+4. Figure 7.8 uses an adder to increment the PC by 4. In our example, *PCNext* = 0x1000 + 4 = 0x1004. The new address is written into the program counter on the next rising edge of the clock. This completes the datapath for the `lw` instruction.



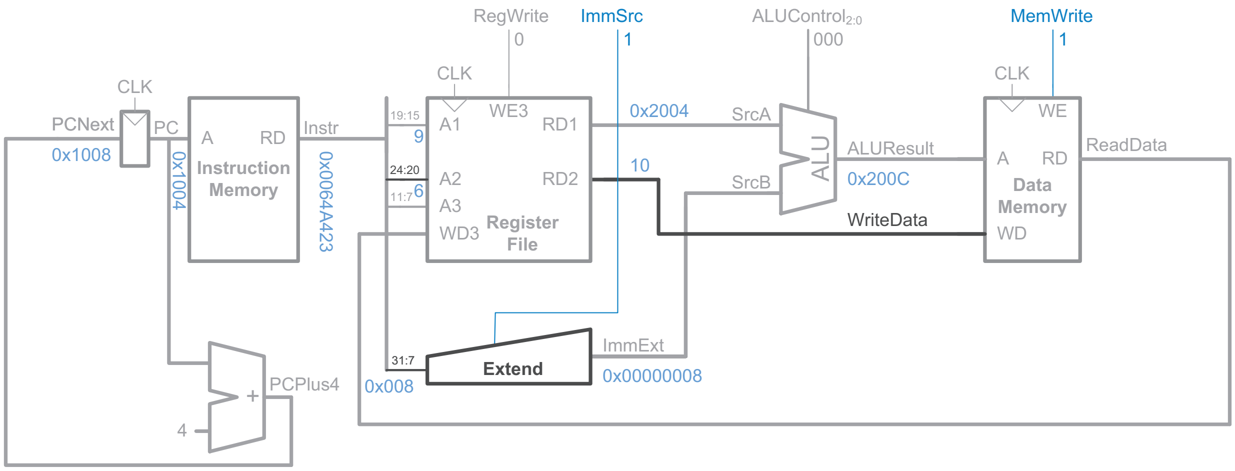
Address	Instruction	Type	Fields				Machine Language
			imm$_{11:0}$	rs1	f3	rd	op
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011 FFC4A303

Figure 7.8 Increment program counter

`sw`

Next, let us extend the datapath to handle `sw`, which is an S-type instruction. Like `lw`, `sw` 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 in the datapath, but the 12-bit signed immediate is stored in $Instr_{31:25,11:7}$ (instead of $Instr_{31:20}$, as it was for `lw`). Thus, the Extend unit must be modified to also receive these additional bits, $Instr_{11:7}$. For simplicity (and for future instructions such as `jal`), the Extend unit receives all the bits of $Instr_{31:7}$. A control signal, *ImmSrc*, decides which instruction bits to use as the immediate. When *ImmSrc* = 0 (for `lw`), the Extend unit chooses $Instr_{31:20}$ as the 12-bit signed immediate; when *ImmSrc* = 1 (for `sw`), it chooses $Instr_{31:25,11:7}$.

The `sw` instruction also reads a second register from the register file and writes its contents to the data memory. Figure 7.9 shows the new connections for this added functionality. The register is specified in the **rs2** field, $Instr_{24:20}$, which is connected to the address 2 (*A2*) input of the register file. The register's contents are read onto the read data 2 (*RD2*) output, which, in turn, is connected to the write data (*WD*) input of the data memory. The write enable port of the data memory, *WE*, is controlled by *MemWrite*. For an `sw` instruction: *MemWrite* = 1 to write the data to memory; *ALUControl* = 000 to add the base address and offset; and *RegWrite* = 0, because nothing should be written to the register file. Note that data is still read from the address given to the data memory, but this *ReadData* is ignored because *RegWrite* = 0.



Address	Instruction	Type	imm$_{11:5}$	rs2	rs1	f3	imm$_{4:0}$	op	Machine Language
0x1004	`sw x6, 8(x9)`	S	0000000	00110	01001	010	01000	0100011	0064A423

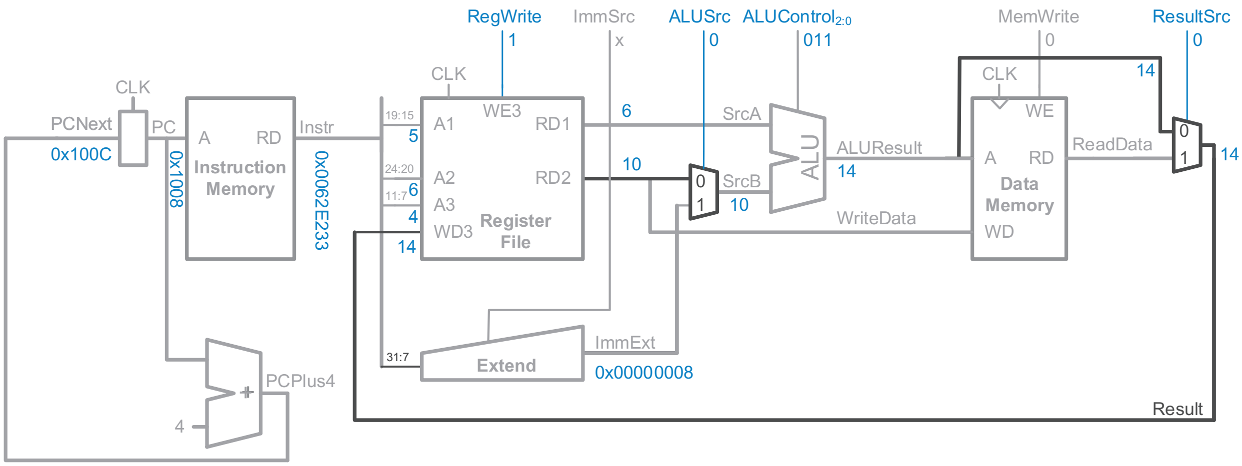
Figure 7.9 Write data to memory for `sw` instruction

In our example, the PC is 0x1004. Thus, the instruction memory reads out the `sw` instruction, 0x0064A423. The register file reads 0x2004 (the base address) from `x9` and 10 from `x6` while the Extend unit extends the immediate offset 8 from 12 to 32 bits. The ALU computes 0x2004 + 8 = 0x200C. The data memory writes 10 to address 0x200C. Meanwhile, the PC is incremented to 0x1008.

R-Type Instructions

Next, consider extending the datapath to handle the R-type instructions, `add`, `sub`, `and`, `or`, and `slt`. All of these instructions read two source registers from the register file, perform some ALU operation on them, and write the result back to the destination register. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware but with different *ALUControl* signals. Recall from Section 5.2.4 that *ALUControl* is 000 for addition, 001 for subtraction, 010 for AND, 011 for OR, and 101 for set less than.

Figure 7.10 shows the enhanced datapath handling these R-type instructions. The datapath reads **rs1** and **rs2** from ports 1 and 2 of the register file and performs an ALU operation on them. We introduce a multiplexer and a new select signal, *ALUSrc*, to select between *ImmExt* and *RD2* as the second ALU source, *SrcB*. For `lw` and `sw`, *ALUSrc* is 1 to select *ImmExt*; for R-type instructions, *ALUSrc* is 0 to select the register file output *RD2* as *SrcB*.



Address	Instruction	Type	Fields					Machine Language	
			funct7	rs2	rs1	f3	rd	op	
0x1008	`or x4, x5, x6`	R	0000000	00110	00101	110	00100	0110011	0062E233

Figure 7.10 Datapath enhancements for R-type instructions

Let us name the value to be written back to the register file *Result*. For `lw`, *Result* comes from the *ReadData* output of the memory. However, for R-type instructions, *Result* comes from the *ALUResult* output of the ALU. We add the Result multiplexer to choose the proper *Result* based on the type of instruction. The multiplexer select signal *ResultSrc* is 0 for R-type instructions to choose *ALUResult* as *Result*; *ResultSrc* is 1 for `lw` to choose *ReadData*. We do not care about the value of *ResultSrc* for `sw` because it does not write the register file.

In our example, the PC is 0x1008. Thus, the instruction memory reads out the `or` instruction 0x0062E233. The register file reads source operands 6 from `x5` and 10 from `x6`. *ALUControl* is 011, so the ALU computes $6 \mid 10 = 0110_2 \mid 1010_2 = 1110_2 = 14$. The result is written back to `x4`. Meanwhile, the PC is incremented to 0x100C.

`beq`

Finally, we extend the datapath to handle the branch if equal (`beq`) instruction. `beq` compares two registers. If they are equal, it takes the branch by adding the branch offset to the program counter (PC).

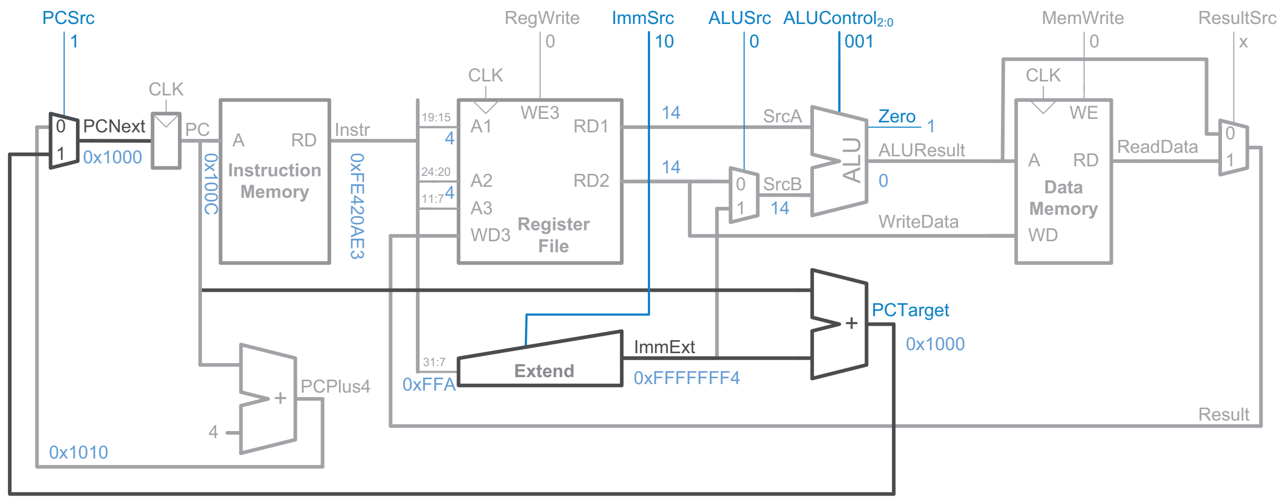
The branch offset is a 13-bit signed immediate stored in the 12-bit immediate field of the B-type instruction. Thus, the Extend logic needs yet another mode to choose the proper immediate. *ImmSrc* is increased to 2 bits, using the encoding from Table 7.1. *ImmExt* is now either the

Observe that our hardware computes all the possible answers needed by different instructions (e.g., *ALUResult* and *ReadData*) and then uses a multiplexer to choose the appropriate one based on the instruction. This is an important design strategy. Throughout the rest of this chapter, we will add multiplexers to choose the desired answer.

One of the major differences between software and hardware is that software operates sequentially, so we can compute just the answer we need. Hardware operates in parallel; therefore, we often compute all the possible answers and then pick the one we need. For example, while executing an R-type instruction with the ALU, the memory still receives an address and reads data from this address even though we don't care what that data might be.

Table 7.1 ***ImmSrc* encoding**

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate



Address	Instruction	Type	Fields						Machine Language
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Figure 7.11 **Datapath enhancements for** `beq`

Logically, we can build the Extend unit from a 32-bit 3:1 multiplexer choosing one of three possible inputs based on *ImmSrc* and the various bitfields of the instruction. In practice, the upper bits of the sign-extended immediate always come from bit 31 of the instruction, $Instr_{31}$, so we can optimize the design and only use a multiplexer to select the lower bits.

sign-extended immediate (when *ImmSrc* = 00 or 01) or the branch offset (when *ImmSrc* = 10).

Figure 7.11 shows the modifications to the datapath. We need another adder to compute the branch target address, *PCTarget* = *PC* + *ImmExt*. The two source registers are compared by computing (*SrcA* – *SrcB*) using the ALU. If *ALUResult* is 0, as indicated by the ALU's *Zero* flag, the registers are equal. We add a multiplexer to choose *PCNext* from either *PCPlus4* or *PCTarget*. *PCTarget* is selected if the instruction is a branch and the *Zero* flag is asserted. For `beq`, *ALUControl* = 001, so that the ALU performs a subtraction. *ALUSrc* = 0 to choose *SrcB* from the register file. *RegWrite* and *MemWrite* are 0, because a branch does not write to the register file or memory. We don't care about the value of *ResultSrc*, because the register file is not written.

In our example, the PC is 0x100C, so the instruction memory reads out the `beq` instruction 0xFE420AE3. Both source registers are `x4`, so the

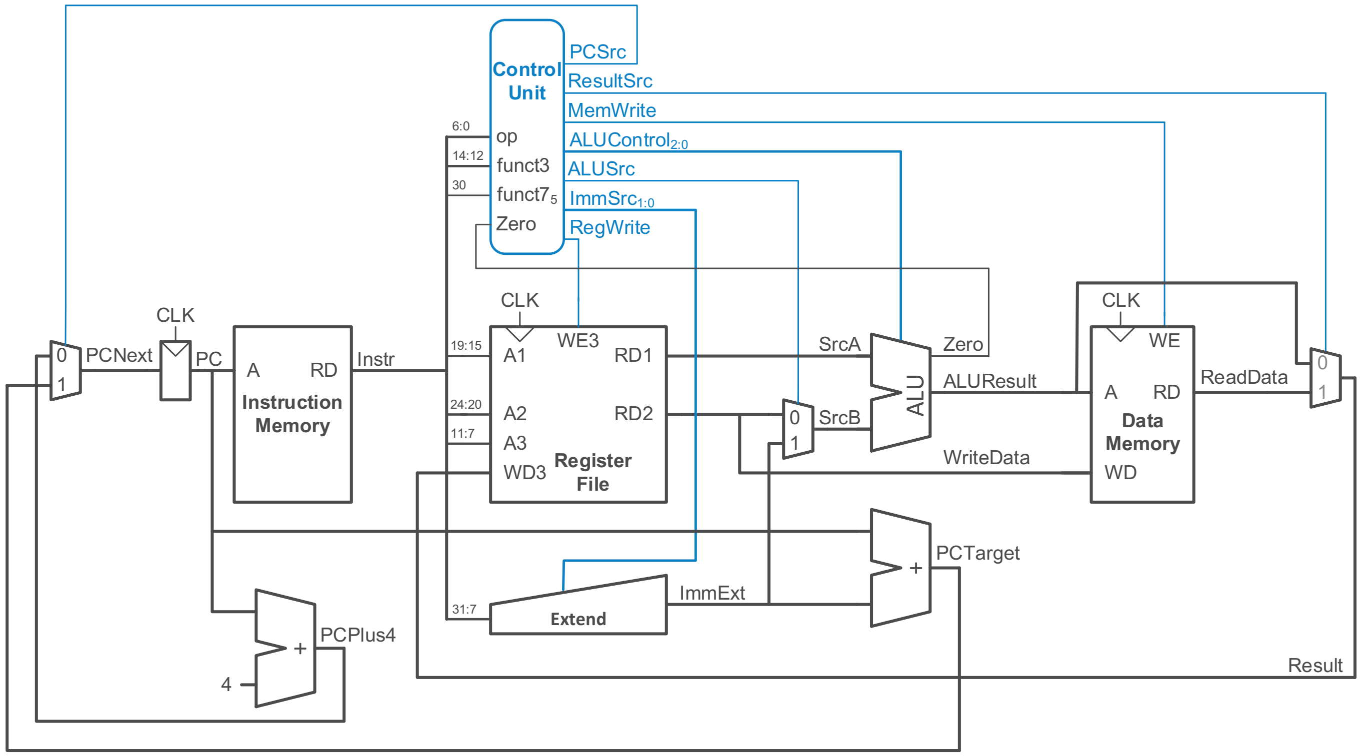


Figure 7.12 Complete single-cycle processor

register file reads 14 on both ports. The ALU computes 14 – 14 = 0, and the *Zero* flag is asserted. Meanwhile, the Extend unit produces 0xFFFFFFF4 (i.e., –12), which is added to *PC* to obtain *PCTarget* = 0x1000. Note that we show the unswizzled upper 12 bits of the 13-bit immediate on the input of the Extend unit (0xFFA). The PCNext mux chooses *PCTarget* as the next PC and branches back to the start of the code at the next clock edge.

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

We name the multiplexers (muxes) by the signals they produce. For example, the PCNext mux produces the *PCNext* signal, and the Result mux produces the *Result* signal.

7.3.3 Single-Cycle Control

The single-cycle processor's control unit computes the control signals based on **op**, **funct3**, and **funct7**. For the RV32I instruction set, only bit 5 of **funct7** is used, so we just need to consider **op** ($Instr_{6:0}$), **funct3** ($Instr_{14:12}$), and **funct7**$_5$ ($Instr_{30}$). Figure 7.12 shows the entire single-cycle processor with the control unit attached to the datapath.

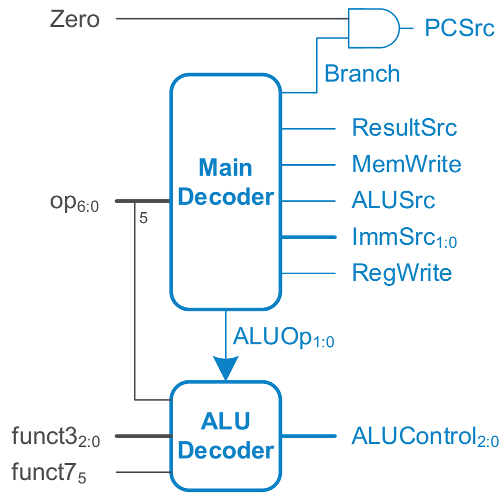


Figure 7.13 Single-cycle processor control unit

Table 7.2 Main Decoder truth table

Instruction	Op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
`lw`	0000011	1	00	1	0	1	0	00
`sw`	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
`beq`	1100011	0	10	0	0	x	1	01

Figure 7.13 hierarchically decomposes the control unit, which is also referred to as the *controller* or the *decoder*, because it decodes what the instruction should do. We partition it into two major parts: the Main Decoder, which produces most of the control signals, and the ALU Decoder, which determines what operation the ALU performs.

Table 7.2 shows the control signals that the Main Decoder produces, as we determined while designing the datapath. The Main Decoder determines the instruction type from the opcode and then produces the appropriate control signals for the datapath. The Main Decoder generates most of the control signals for the datapath. It also produces internal signals *Branch* and *ALUOp*, signals used within the controller. The logic for the Main Decoder can be developed from the truth table using your favorite techniques for combinational logic design.

The ALU Decoder produces *ALUControl* based on *ALUOp* and **funct3**. In the case of the `sub` and `add` instructions, the ALU Decoder also uses **funct7**$_5$ and **op**$_5$ to determine *ALUControl*, as given in in Table 7.3.

Table 7.3 ALU Decoder truth table

ALUOp	funct3	{op_5, $funct7_5$}	ALUControl	Instruction
00	x	x	000 (add)	`lw`, `sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt`
	110	x	011 (or)	`or`
	111	x	010 (and)	`and`

ALUOp of 00 indicates add (e.g., to find the address for loads or stores). *ALUOp* of 01 indicates subtract (e.g., to compare two numbers for branches). *ALUOp* of 10 indicates an R-type ALU instruction where the ALU Decoder must look at the **funct3** field (and sometimes also the **op_5** and **$funct7_5$** bits) to determine which ALU operation to perform (e.g., `add`, `sub`, `and`, `or`, `slt`).

According to Table B.1 in the inside covers of the book, `add`, `sub`, and `addi` all have **funct3** = 000. `add` has **funct7** = 0000000 while `sub` has **funct7** = 0100000, so **$funct7_5$** is sufficient to distinguish these two. But we will soon consider supporting `addi`, which doesn't have a **funct7** field but has an **op** of 0010011. With a bit of thought, we can see that an ALU instruction with **funct3** = 000 is `sub` if **op_5** and **$funct7_5$** are both 1, or `add` or `addi` otherwise.

Example 7.1 SINGLE-CYCLE PROCESSOR OPERATION

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

Solution Figure 7.14 illustrates the control signals and flow of data during execution of an `and` instruction. The PC points to the memory location holding the instruction; the instruction memory outputs this instruction. The main flow of data through the register file and ALU is represented with a heavy blue line. The register file reads the two source operands specified by *Instr*. *SrcB* should come from the second port of the register file (not *ImmExt*), so *ALUSrc* must be 0. The ALU performs a bitwise AND operation, so *ALUControl* must be 010. The result comes from the ALU, so *ResultSrc* is 0, and the result is written to the register file, so *RegWrite* is 1. The instruction does not write memory, so *MemWrite* is 0.

The updating of *PC* with *PCPlus4* is shown by a heavy gray line. *PCSrc* is 0 to select the incremented PC. Note that data does flow through the nonhighlighted paths, but the value of that data is disregarded. For example, the immediate is extended and a value is read from memory, but these values do not influence the next state of the system.

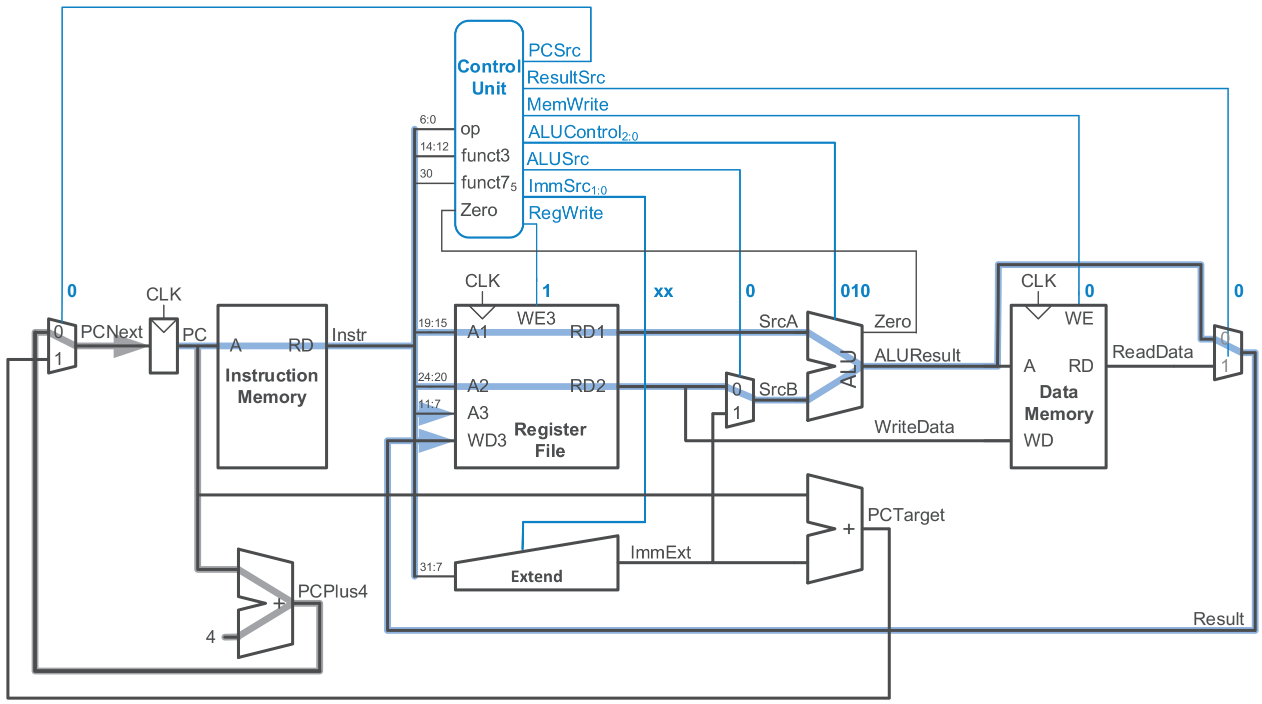


Figure 7.14 **Control signals and data flow while executing an** `and` **instruction**

7.3.4 More Instructions

So far, we have considered only a small subset of the RISC-V instruction set. In this section, we enhance the datapath and controller to support the `addi` (add immediate) and `jal` (jump and link) instructions. These examples illustrate the principle of how to handle new instructions, and they give us a sufficiently rich instruction set to write many interesting programs. With enough effort, you could extend the single-cycle processor to handle every RISC-V instruction. Moreover, we will see that supporting some instructions simply requires enhancing the decoders, whereas supporting others also requires new hardware in the datapath.

Example 7.2 `addi` INSTRUCTION

Recall that `addi rd,rs1,imm` is an I-type instruction that adds the value in **rs1** to a sign-extended immediate and writes the result to **rd**. 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 12-bit immediate in $Instr_{31:20}$ is sign-extended as it was with `lw`, another I-type instruction, so

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

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
`lw`	0000011	1	00	1	0	1	0	00
`sw`	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
`beq`	1100011	0	10	0	0	x	1	01
`addi`	**0010011**	**1**	**00**	**1**	**0**	**0**	**0**	**10**

ImmSrc is 00 (see Table 7.1). *SrcB* comes from the immediate, so *ALUSrc* = 1. The instruction does not write memory nor is it a branch, so *MemWrite* = *Branch* = 0. The result comes from the ALU, not memory, so *ResultSrc* = 0. Finally, the ALU should add, so *ALUOp* = 10; the ALU Decoder makes *ALUControl* = 000 because **funct3** = 000 and $\mathbf{op}_5 = 0$.

The astute reader may note that this change also provides the other I-type ALU instructions: `andi`, `ori`, and `slti`. These other instructions share the same **op** value of 0010011, need the same control signals, and only differ in the **funct3** field, which the ALU Decoder already uses to determine *ALUControl* and, thus, the ALU operation.

Example 7.3 `jal` INSTRUCTION

Show how to change the RISC-V single-cycle processor to support the jump and link (`jal`) instruction. `jal` writes PC+4 to **rd** and changes PC to the jump target address, PC + **imm**.

Solution The processor calculates the jump target address, the value of *PCNext*, by adding PC to the 21-bit signed immediate encoded in the instruction. The least significant bit of the immediate is always 0 and the next 20 most significant bits come from $Instr_{31:12}$. This 21-bit immediate is then sign-extended. The datapath already has hardware for adding *PC* to a sign-extended immediate, selecting this as the next PC, computing PC+4, and writing a value to the register file. Hence, in the datapath, we must only modify the Extend unit to sign-extend the 21-bit immediate and expand the Result multiplexer to choose PC+4 (i.e., *PCPlus4*) as shown in Figure 7.15. Table 7.5 shows the new encoding for *ImmSrc* to support the long immediate for `jal`.

The control unit needs to set *PCSrc* = 1 for the jump. To do this, we add an OR gate and another control signal, *Jump*, as shown in Figure 7.16. When *Jump* asserts, *PCSrc* = 1 and *PCTarget* (the jump target address) is selected as the next PC.

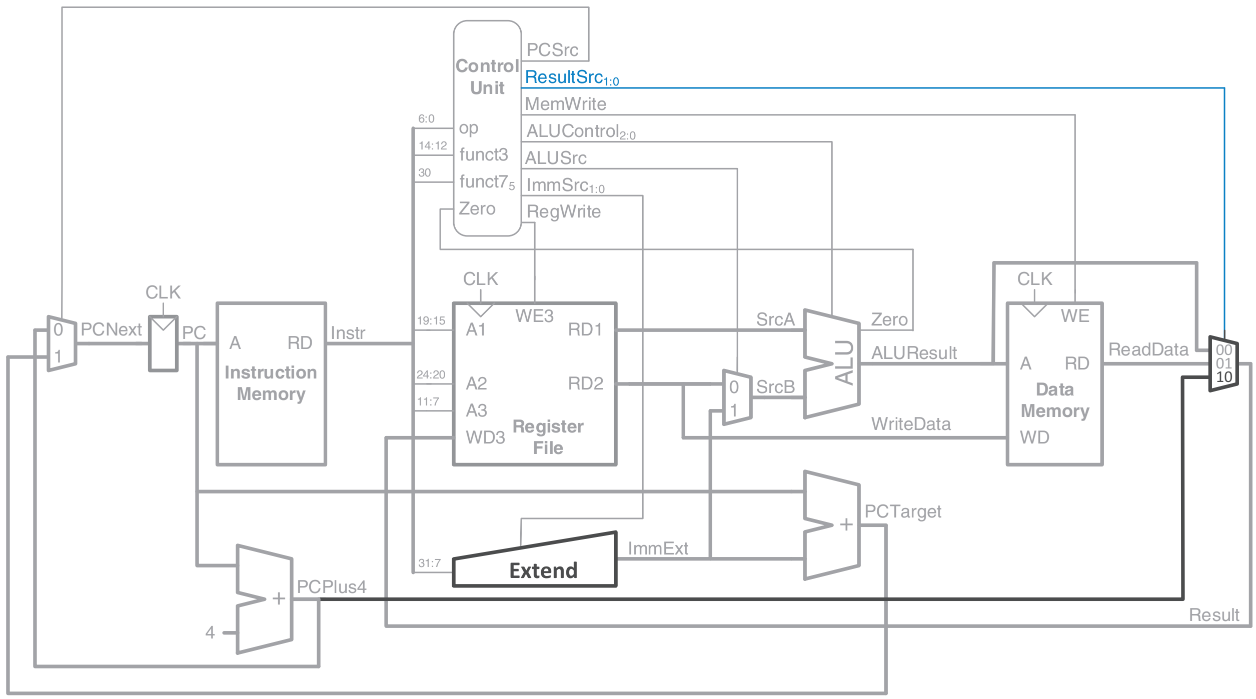


Figure 7.15 Enhanced datapath for `jal`

Table 7.5 ***ImmSrc*** **encoding.**

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1’b0}	B	13-bit signed immediate
11	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1’b0}	J	21-bit signed immediate

Table 7.6 shows the updated Main Decoder table with a new row for `jal`. *RegWrite* = 1 and *ResultSrc* = 10 to write PC+4 into **rd**. *ImmSrc* = 11 to select the 21-bit jump offset. *ALUSrc* and *ALUOp* don’t matter because the ALU is not used. *MemWrite* = 0 because the instruction isn’t a store, and *Branch* = 0 because the instruction isn’t a branch. The new *Jump* signal is 1 to pick the jump target address as the next PC.

7.3.5 Performance Analysis

Recall from Equation 7.1 that the execution time of a program is the product of the number of instructions, the cycles per instruction, and

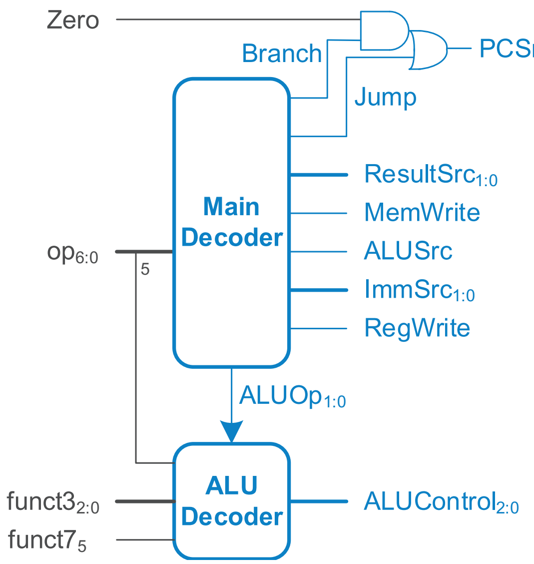


Figure 7.16 Enhanced control unit for `jal`

Table 7.6 Main Decoder truth table enhanced to support `jal`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	00	1	0	**01**	0	00	**0**
`sw`	0100011	0	01	1	1	**xx**	0	00	**0**
R-type	0110011	1	xx	0	0	**00**	0	10	**0**
`beq`	1100011	0	10	0	0	**xx**	1	01	**0**
I-type ALU	0010011	1	00	1	0	**00**	0	10	**0**
`jal`	**1101111**	**1**	**11**	**x**	**0**	**10**	**0**	**xx**	**1**

the cycle time. Each instruction in the single-cycle processor takes one clock cycle, so the clock cycles per instruction (CPI) is 1. The cycle time is set by the critical path. In our processor, the `lw` instruction is the most time-consuming and involves the critical path shown in Figure 7.17. As indicated by heavy blue lines, the critical path starts with the PC loading a new address on the rising edge of the clock. The instruction memory then reads the new instruction, and the register file reads **rs1** as *SrcA*. While the register file is reading, the immediate field is sign-extended based on *ImmSrc* and selected at the SrcB multiplexer (path highlighted in gray). The ALU adds *SrcA* and *SrcB* to find the memory address. The data memory reads from this address, and the Result multiplexer selects *ReadData* as *Result*. Finally, *Result* must set up at the register file before

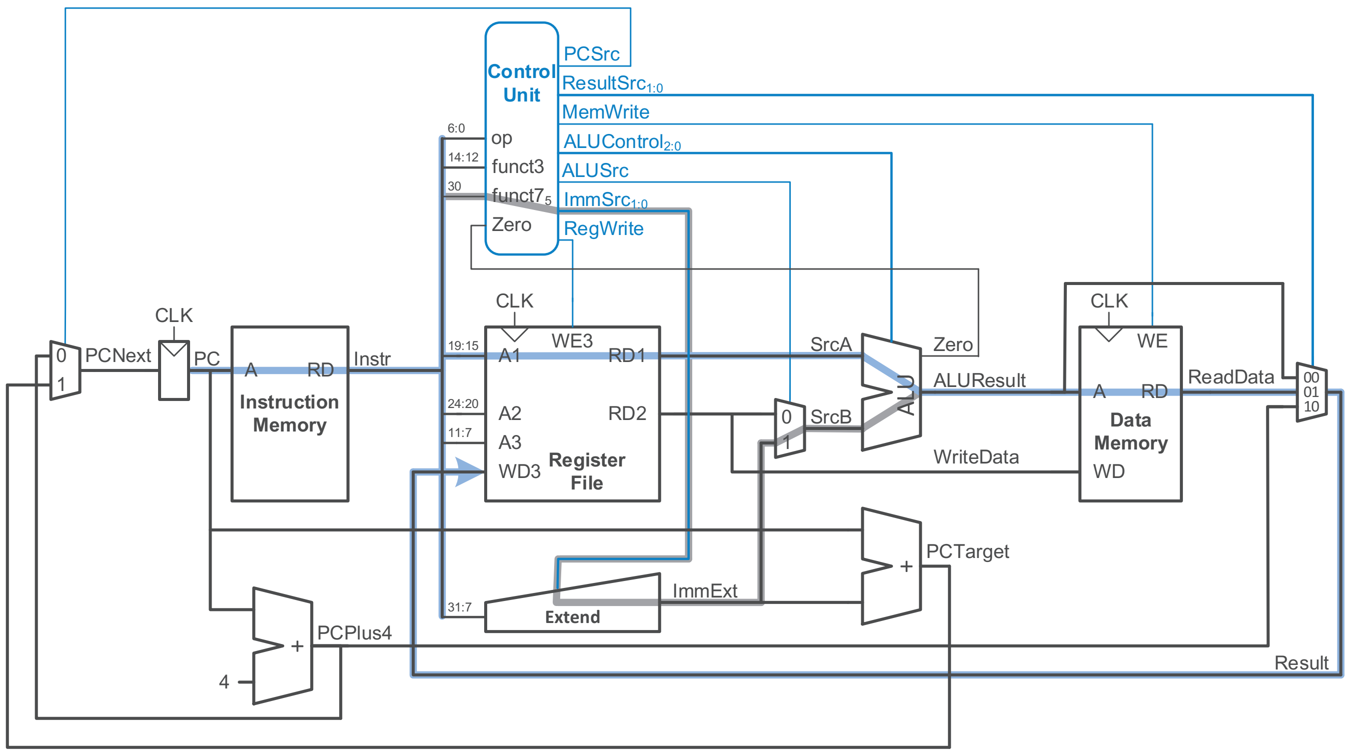


Figure 7.17 Critical path for `lw`

the next rising clock edge so that it can be properly written. Hence, the cycle time of the single-cycle processor is:

$$\begin{aligned} T_{c\_single} = \; & t_{pcq\_PC} + t_{mem} + \max[t_{RFread}, \quad t_{dec} + t_{ext} + t_{mux}] \\ & + \; t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup} \end{aligned} \tag{7.2}$$

Remember that `lw` does not use the second read port (A2/RD2) of the register file.

In most implementation technologies, the ALU, memory, and register file are substantially slower than other combinational blocks. Therefore, the critical path is through the register file—not through the decoder (controller), Extend unit, and multiplexer—and is the path highlighted in blue in Figure 7.17. The cycle time simplifies to:

$$T_{c\_single} = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup} \tag{7.3}$$

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

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.

Table 7.7 Delay of circuit elements

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Example 7.4 SINGLE-CYCLE PROCESSOR PERFORMANCE

Ben Bitdiddle is contemplating building the single-cycle processor in a 7-nm CMOS manufacturing process. He has determined that the logic elements have the delays given in Table 7.7. Help him compute 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 $T_{c\_single} = 40 + 2(200) + 100 + 120 + 30 + 60 = 750$ ps. According to Equation 7.1, the total execution time is $T_{single} = (100 \times 10^9$ instruction) (1 cycle/instruction) $(750 \times 10^{-12}$ s/cycle) = 75 seconds.

7.4 MULTICYCLE PROCESSOR

The single-cycle processor has three notable weaknesses. First, it requires separate memories for instructions and data, whereas most processors have only a single external memory holding both instructions and data. Second, it requires a clock cycle long enough to support the slowest instruction (`lw`) even though most instructions could be faster. Finally, 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.

The multicycle processor addresses these weaknesses by breaking an instruction into multiple shorter steps. The memory, ALU, and register file have the longest delays, so to keep the delay for each short step approximately equal, the processor can use *only one* of those units in each step. The processor uses a single memory because the instruction is read in one step and data is read or written in a later step. And the processor needs only one adder, which is reused for different purposes on different steps. Various instructions use different numbers of steps, so simpler instructions can complete faster than more complex ones.

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. During the execution of a single instruction, the controller produces different signals on each step, so now the controller uses a finite state machine rather than combinational logic. Finally, we analyze the performance of the multicycle processor and compare it with the single-cycle processor.

7.4.1 Multicycle Datapath

Again, we begin our design with the memory and architectural state of the processor, as shown in Figure 7.18. 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 is feasible because we can read the instruction in one cycle, then read or write the data in another cycle. The PC and register file remain unchanged.

As with the single-cycle processor, we gradually build the datapath by adding components to handle each step of each instruction. The PC contains the address of the instruction to execute. The first step is to read this instruction from memory. Figure 7.19 shows that the PC is

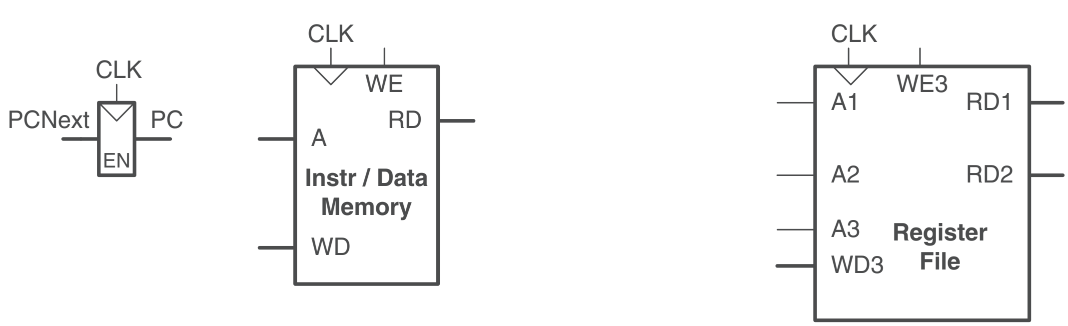


Figure 7.18 State elements with unified instruction/data memory

simply connected to the address input of the memory. The instruction is read and stored in a new nonarchitectural instruction register (IR) so that it is available for future cycles. The IR receives an enable signal, called *IRWrite*, which is asserted when the IR should be loaded with a new instruction.

Like in the single-cycle processor, we name the multiplexers and nonarchitectural registers by the signals they produce. For example, the instruction register produces the instruction signal (*Instr*), and the Result multiplexer produces the *Result* signal.

`lw`

As we did with the single-cycle processor, we first work out the datapath connections for the `lw` instruction. After fetching `lw`, the second step is to read the source register containing the base address. This register is specified in the **rs1** field, $Instr_{19:15}$. These bits of the instruction are connected to address input *A1* of the register file, as shown in Figure 7.20. The register file reads the register onto *RD1*, and this value is stored in another nonarchitectural register, *A*.

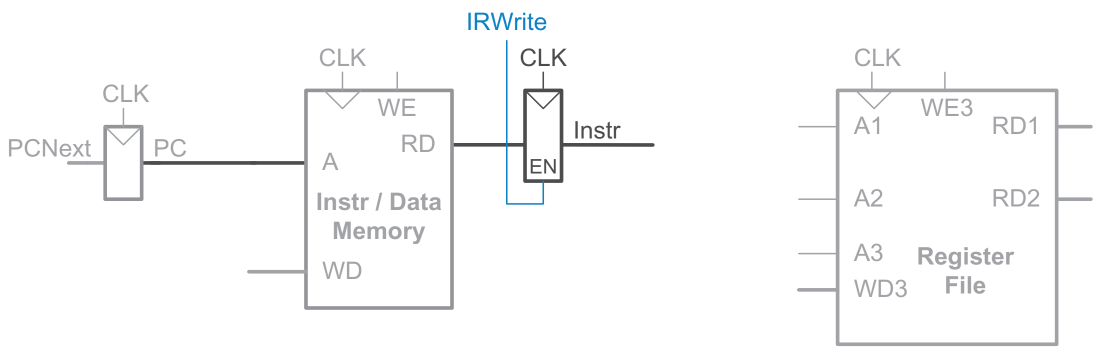


Figure 7.19 Fetch instruction from memory

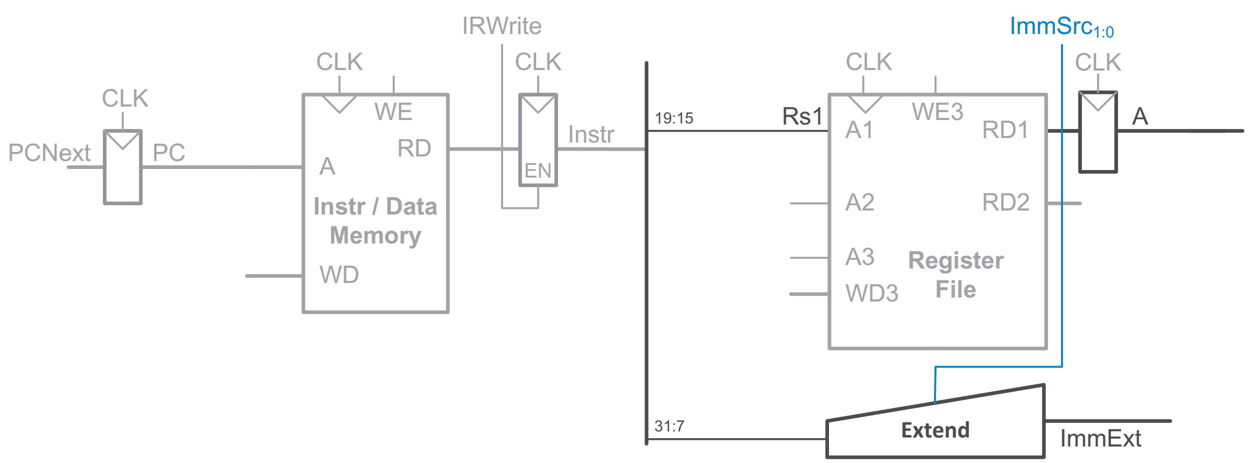


Figure 7.20 Read one source from register file and extend second source from immediate field

The `lw` instruction also requires a 12-bit offset found in the immediate field of the instruction, $Instr_{31:20}$, which must be sign-extended to 32 bits, as shown in Figure 7.20. As in the single-cycle processor, the Extend unit takes a 2-bit *ImmSrc* control signal to specify a 12-, 13-, or 21-bit immediate to extend for various types of instructions. The 32-bit extended immediate is called *ImmExt*. To be consistent, we might store *ImmExt* in another nonarchitectural register. However, *ImmExt* 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. In the third step, we use an ALU to compute this sum, as shown in Figure 7.21. *ALUControl* should be set to 000 to perform the addition. *ALUResult* is stored in a nonarchitectural register called ALUOut.

The fourth 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 memory address, *Adr*, from either the PC or *ALUOut* based on the *AdrSrc* select signal, as shown in Figure 7.22. The data read from memory is stored in another nonarchitectural register, called *Data*. Note that the address (Adr) multiplexer permits us to reuse the memory unit during the `lw` instruction. On the first step, the address is taken from the PC to fetch the instruction. On the fourth step, the address is taken from *ALUOut* to load the data. Hence, *AdrSrc* must have different values during different steps of a single instruction. 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 7.23. The destination register is specified by the **rd** field of the instruction, $Instr_{11:7}$. The result comes from the Data register. Instead of connecting the Data register directly to the register file's *WD3* write port, let us add a multiplexer on the *Result* bus to choose either *ALUOut* or *Data* before feeding *Result* back to the register file's

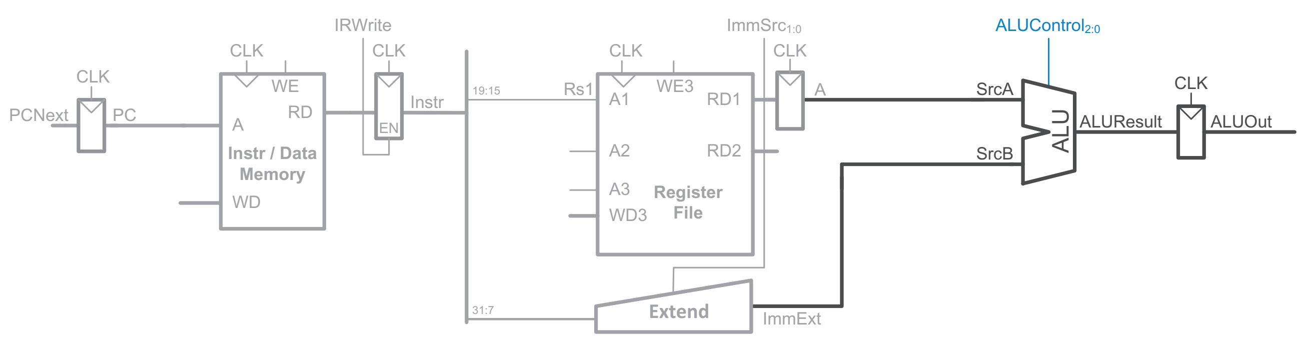


Figure 7.21 Add base address to offset

writedata port (*WD3*). This will be helpful because other instructions will need to write a result from the ALU to the register file. The *RegWrite* signal is 1 to indicate that the register file should be updated.

While all this is happening, the processor must update the program counter by adding 4 to the PC. In the single-cycle processor, a separate adder was needed. In the multicycle processor, we can use the existing ALU during the instruction fetch step because it is not busy. To do so, we must insert source multiplexers to choose *PC* and the constant 4 as ALU inputs, as shown in Figure 7.24. A multiplexer controlled by *ALUSrcA* chooses either *PC* or *A* as *SrcA*. Another multiplexer chooses either 4 or *ImmExt* as *SrcB*. We also show additional multiplexer inputs that will be used when we implement more instructions. To update the PC, the ALU adds *SrcA* (*PC*) to *SrcB* (4), and the result is written into the program counter. The Result multiplexer chooses this sum from *ALUResult* rather than *ALUOut*; this requires a third multiplexer input. The *PCWrite* control signal enables the PC to be written only on certain cycles. This completes the datapath for the `lw` instruction.

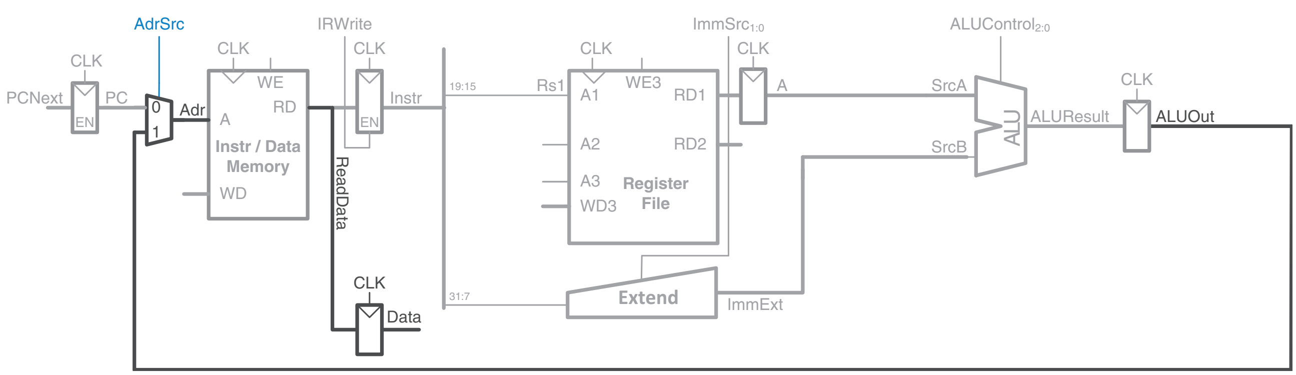


Figure 7.22 Load data from memory

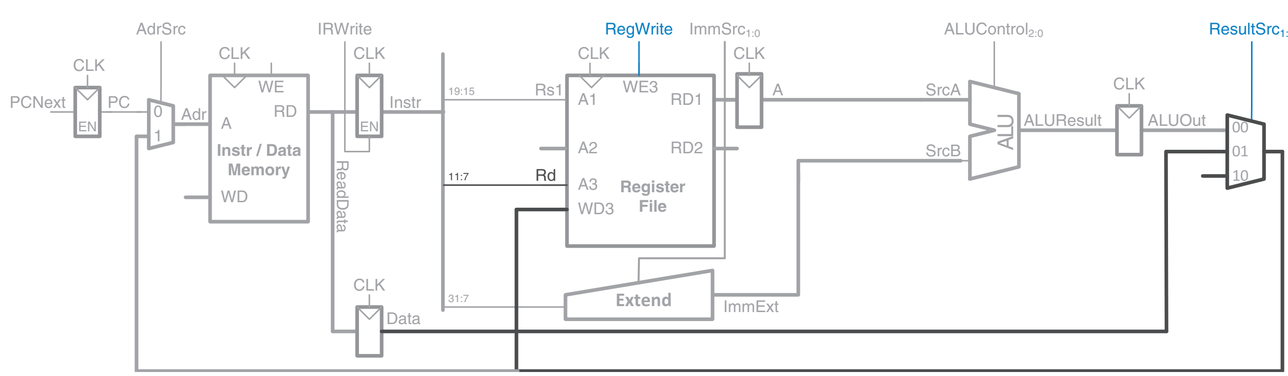


Figure 7.23 Write data back to register file

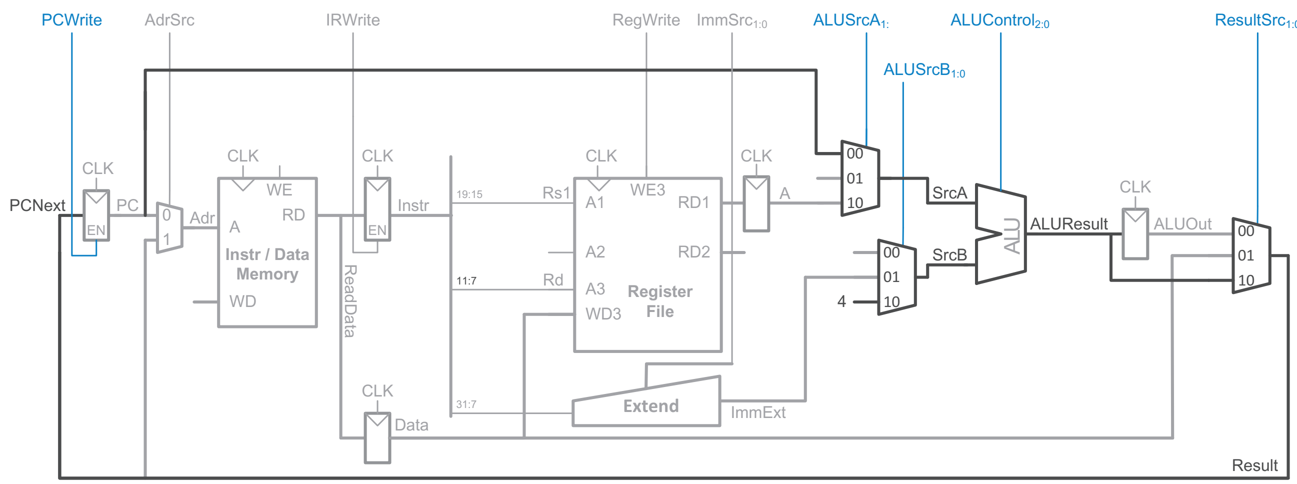


Figure 7.24 Increment PC by 4

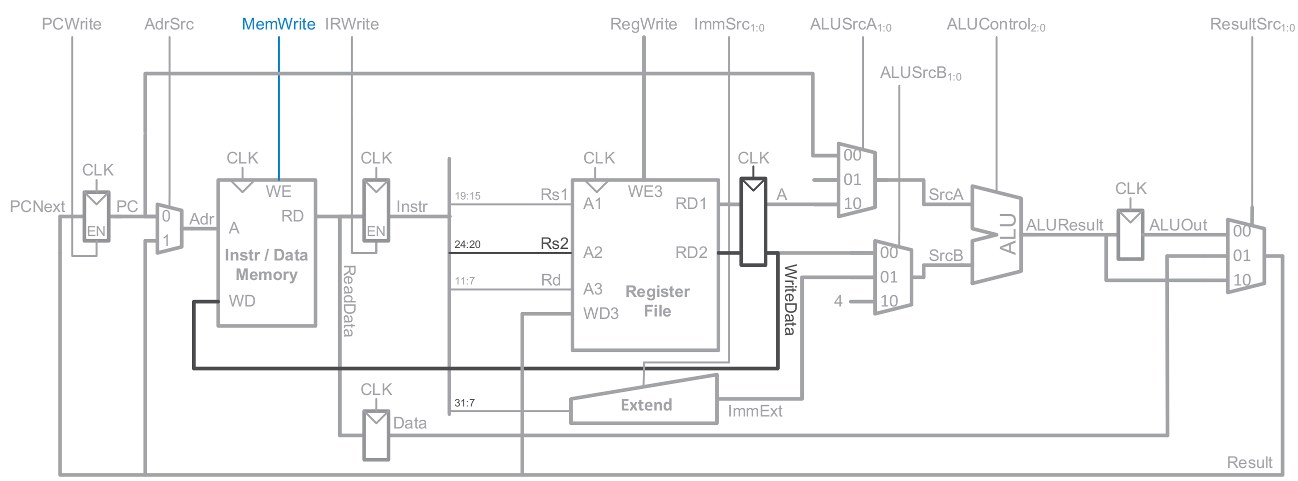


Figure 7.25 Enhanced datapath for `sw` instruction

`sw`

Next, let us extend the datapath to handle the `sw` instruction. Like `lw`, `sw` reads a base address from port 1 of the register file and extends the immediate on the second step. Then, the ALU adds the base address to the immediate to find the memory address on the third step. The only new feature of `sw` is that we must read a second register from the register file and write its contents into memory, as shown in Figure 7.25. The register is specified in the **rs2** field of the instruction, $Instr_{24:20}$, which is connected to the second port of the register file ($A2$). After it is read

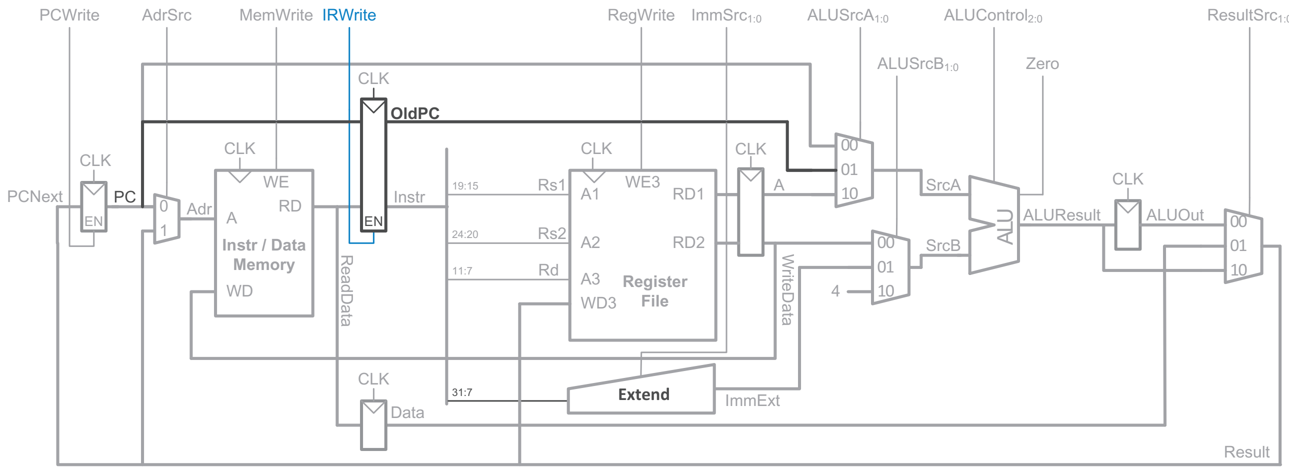


Figure 7.26 Enhanced datapath for `beq` target address calculation

on the second step, the register's contents are then stored in a nonarchitectural register, the WriteData register, just below the A register. It is then sent to the write data port (*WD*) of the data memory to be written on the fourth step. The memory receives the *MemWrite* control signal, which is asserted when memory should be written.

R-Type Instructions

R-type instructions operate on two source registers and write the result back to the register file. The datapath already contains all the connections necessary for these steps.

`beq`

`beq` checks whether two register contents are equal and computes the branch target address by adding the current PC to a 13-bit signed branch offset. The hardware to compare the registers using subtraction is already present in the datapath.

The ALU is not being used during the second step of instruction execution, so we use it then to calculate the branch target address *PCTarget* = *PC* + *ImmExt*. In this step, the instruction has been fetched from memory and PC has already been updated to PC+4. Thus, in the first step, the PC of the current instruction, *OldPC*, must be stored in a nonarchitectural register. In the second step, as the registers are also fetched, the ALU calculates *PC* + *ImmExt* by selecting *OldPC* for *SrcA* and *ImmExt* for *SrcB* and making *ALUControl* = 000 so that it performs addition. The processor stores this sum in the *ALUOut* register. Figure 7.26 shows the updated datapath for `beq`.

In the third step, the ALU subtracts the source registers and asserts the *Zero* output if they are equal. If they are, the control unit asserts

PCWrite and the Result multiplexer selects *ALUOut* (that contains the target address) to feed to the PC. No new hardware is needed.

This completes the design of the multicycle 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 memory can be shared for instructions and data and the ALU can be reused several times, thus reducing hardware costs. 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 **op**, **funct3**, and **funct7$_5$** fields of the instruction ($Instr_{6:0}$, $Instr_{14:12}$, and $Instr_{30}$). Figure 7.27 shows the entire multicycle processor with the control unit attached to the datapath. The datapath is shown in black and the control unit in blue.

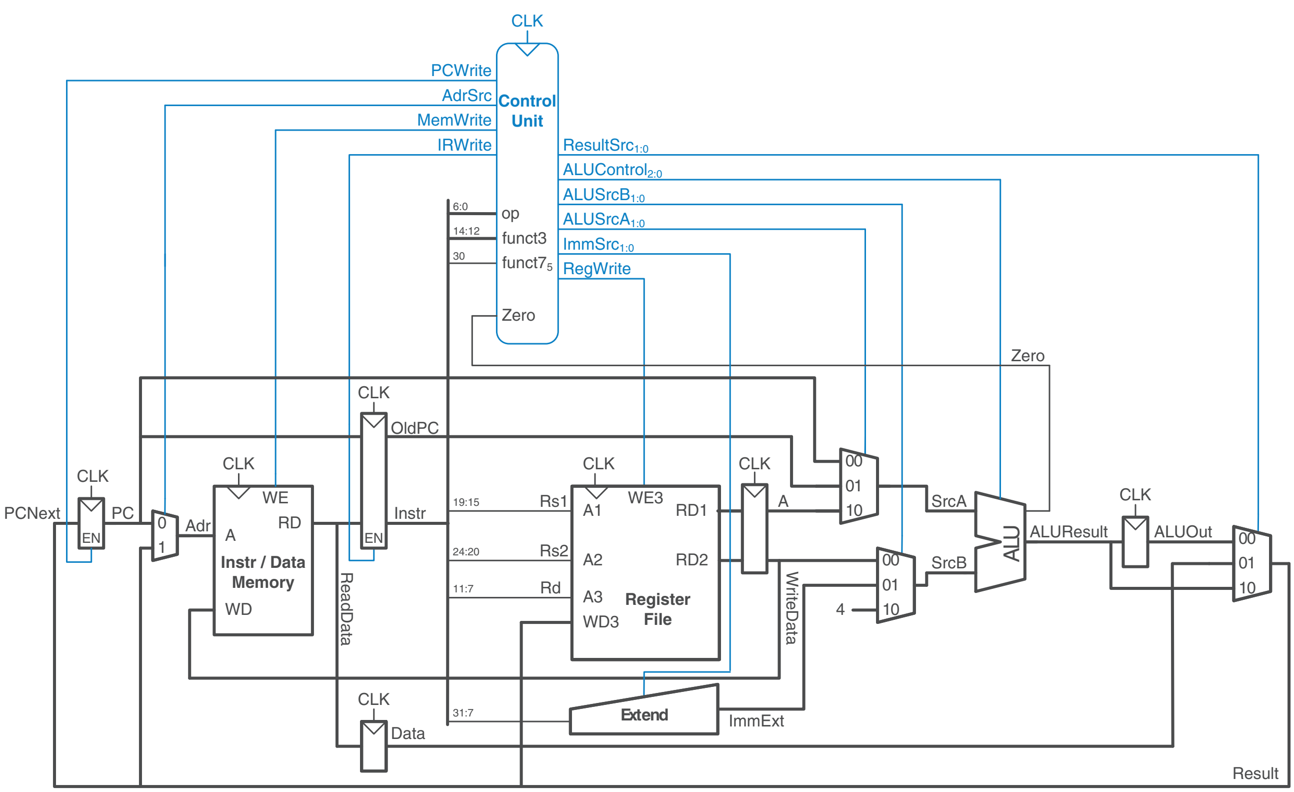


Figure 7.27 Complete multicycle processor

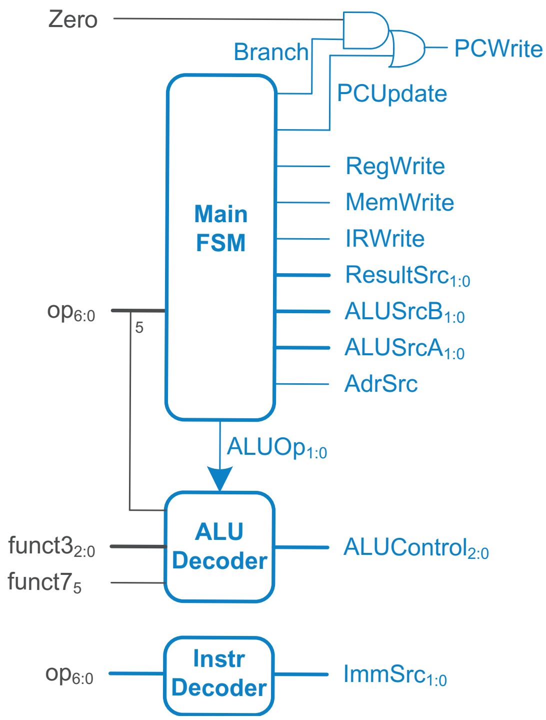


Figure 7.28 Multicycle control unit

The control unit consists of a Main FSM, ALU Decoder, and Instruction Decoder (Instr Decoder) as shown in Figure 7.28. The ALU Decoder is the same as in the single-cycle processor (see Table 7.3), but the combinational Main Decoder of the single-cycle processor is replaced with the Main FSM in the multicycle processor to produce a sequence of control signals on the appropriate cycles. A small Instruction Decoder combinationally produces the *ImmSrc* select signal based on the opcode using the *ImmSrc* column of Table 7.6. We design the Main FSM as a Moore machine so that the outputs are only a function of the current state. The remainder of this section develops the state transition diagram for the Main FSM.

The Main FSM produces multiplexer select, register enable, and memory write enable signals for the datapath. To keep the following state transition diagrams readable, only the relevant control signals are listed. Multiplexer select signals are listed only when their value matters; otherwise, they are don't care. Enable signals (*RegWrite*, *MemWrite*, *IRWrite*, *PCUpdate*, and *Branch*) are listed only when they are asserted; otherwise, they are 0.

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 Fetch state on reset. The control signals are shown in Figure 7.29. To read the instruction

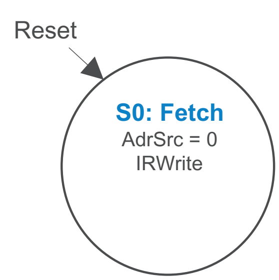


Figure 7.29 Fetch

from memory, *AdrSrc* = 0, so the address is taken from the PC. *IRWrite* is asserted to write the instruction into the instruction register, IR. At the same time, the current PC is written into the *OldPC* register. The data flow through the datapath for this and the next two steps of the `lw` instruction is shown in Figure 7.32, with the flow during the Fetch stage highlighted in gray.

Decode

The second step is to read the register file and decode the instructions. The control unit *decodes* the instruction, that is, figures out what operation should be performed based on **op**, **funct3**, and **funct7**$_5$. In this state, the processor also reads the source registers, **rs1** and **rs2**, and puts the values read into the A and WriteData nonarchitectural registers. No control signals are necessary for these tasks. Figure 7.30 shows the Decode state in the Main FSM and Figure 7.32 shows the flow through the datapath during this state in medium blue lines. After this step, the processor can differentiate its actions based on the instruction because the instruction has been fetched and decoded. We will first show the remaining steps for `lw`, then continue with the steps for the other RISC-V instructions.

MemAdr

The third step for `lw` is to calculate the memory address. The ALU adds the base address and the offset, so *ALUSrcA* = 10 to select *A* (the value read from **rs1**) as *SrcA*, and *ALUSrcB* = 01 to select *ImmExt* as *SrcB*. *ImmSrc*, as determined by the Instruction Decoder, is 00 to sign-extend the I-type immediate, and *ALUOp* is 00 to add *SrcA* and *SrcB*. At the end of this state, the ALU result (i.e., the address calculation) is stored in the ALUOut register. Figure 7.31 shows this MemAdr state added to the Main FSM, and Figure 7.32 shows the datapath flow during this state highlighted in dark-blue lines.

MemRead

The memory read (MemRead) step sends the calculated address to memory by sending *ALUOut* to the address port of the memory, *Adr*.

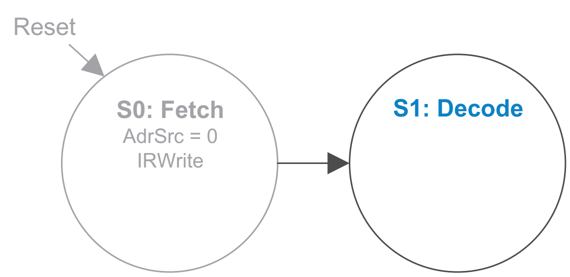


Figure 7.30 Decode

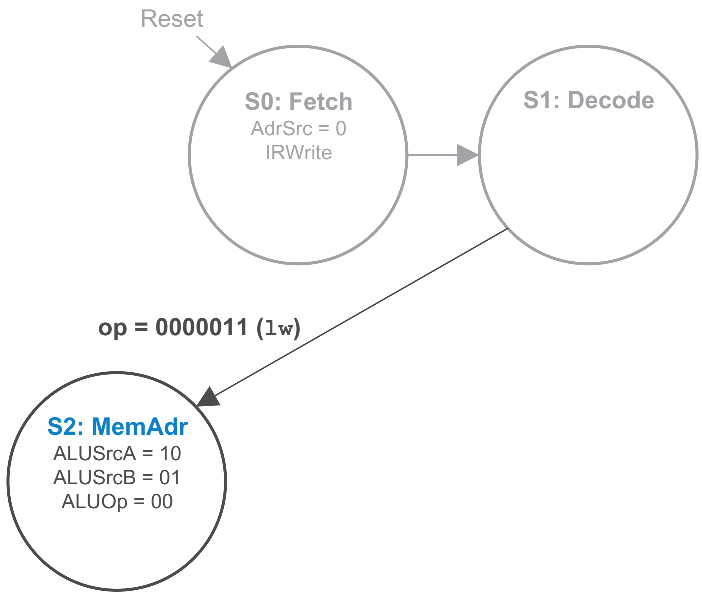


Figure 7.31 Memory address computation

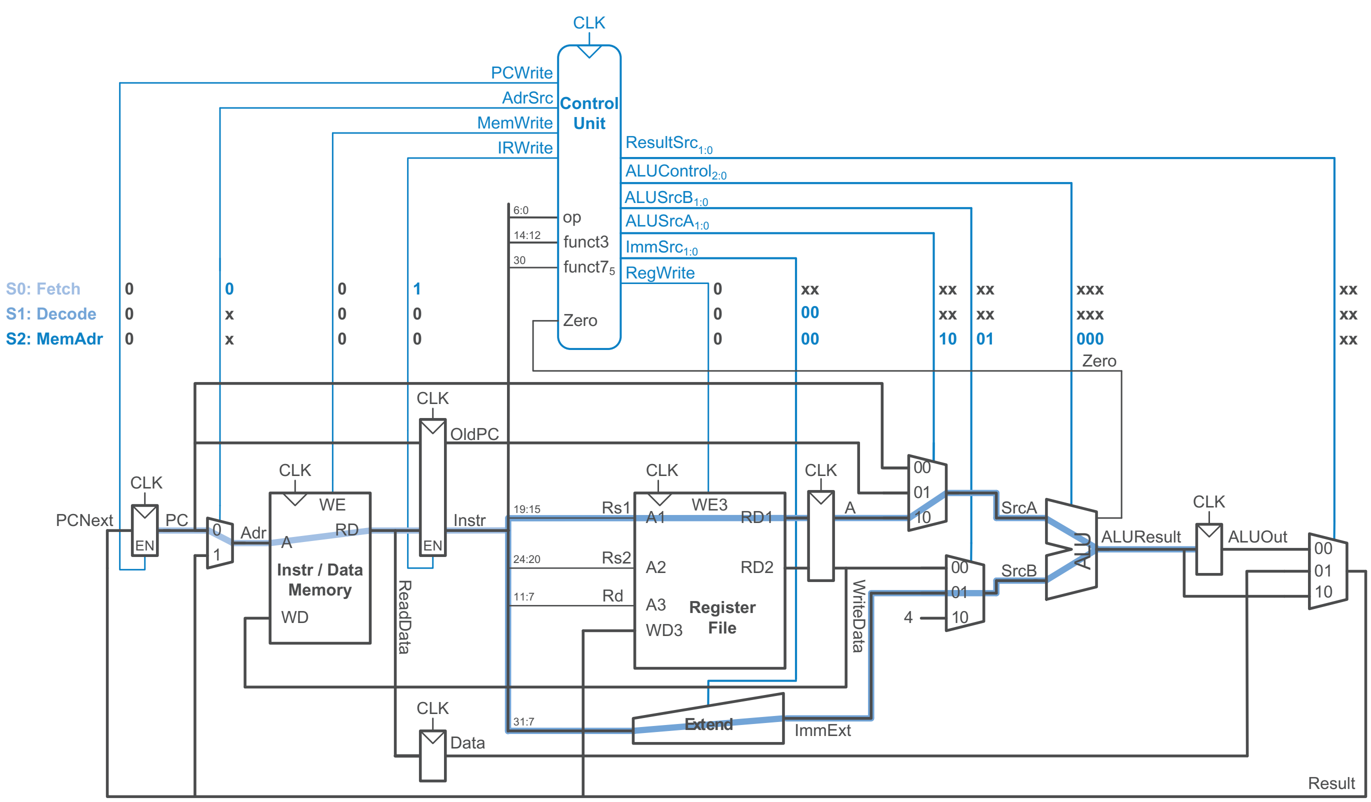


Figure 7.32 Data flow during Fetch, Decode, and MemAdr states

ResultSrc = 00 and *AdrSrc* = 1 to route *ALUOut* through the Result and Adr multiplexers to the memory address input. *ReadData* gets the value read from the desired address in memory. At the end of this state, *ReadData* is written into the Data register.

MemWB

In the memory writeback (MemWB) step, the data read from memory, *Data*, is written to the destination register. *ResultSrc* is 01 to select *Data* as the *Result*, and *RegWrite* is asserted to write the data to the register file. The register file's address and write data inputs for port 3 (*A3* and *WD3*) are already connected to **rd** ($Instr_{11:7}$) and *Result*, respectively. Figures 7.33 and 7.34 show the MemRead and MemWB states and the flow through the datapath for both steps. MemWB is the final step in the `lw` instruction. Figure 7.33 also shows the transition from MemWB back

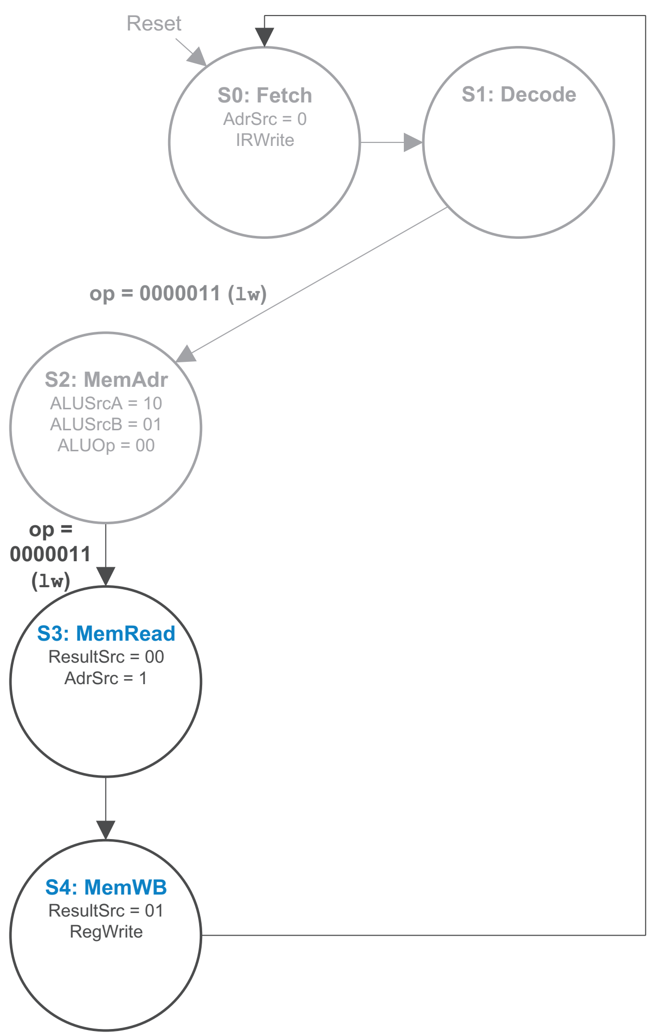


Figure 7.33 **Memory read (MemRead) and memory write back (MemWB) states**

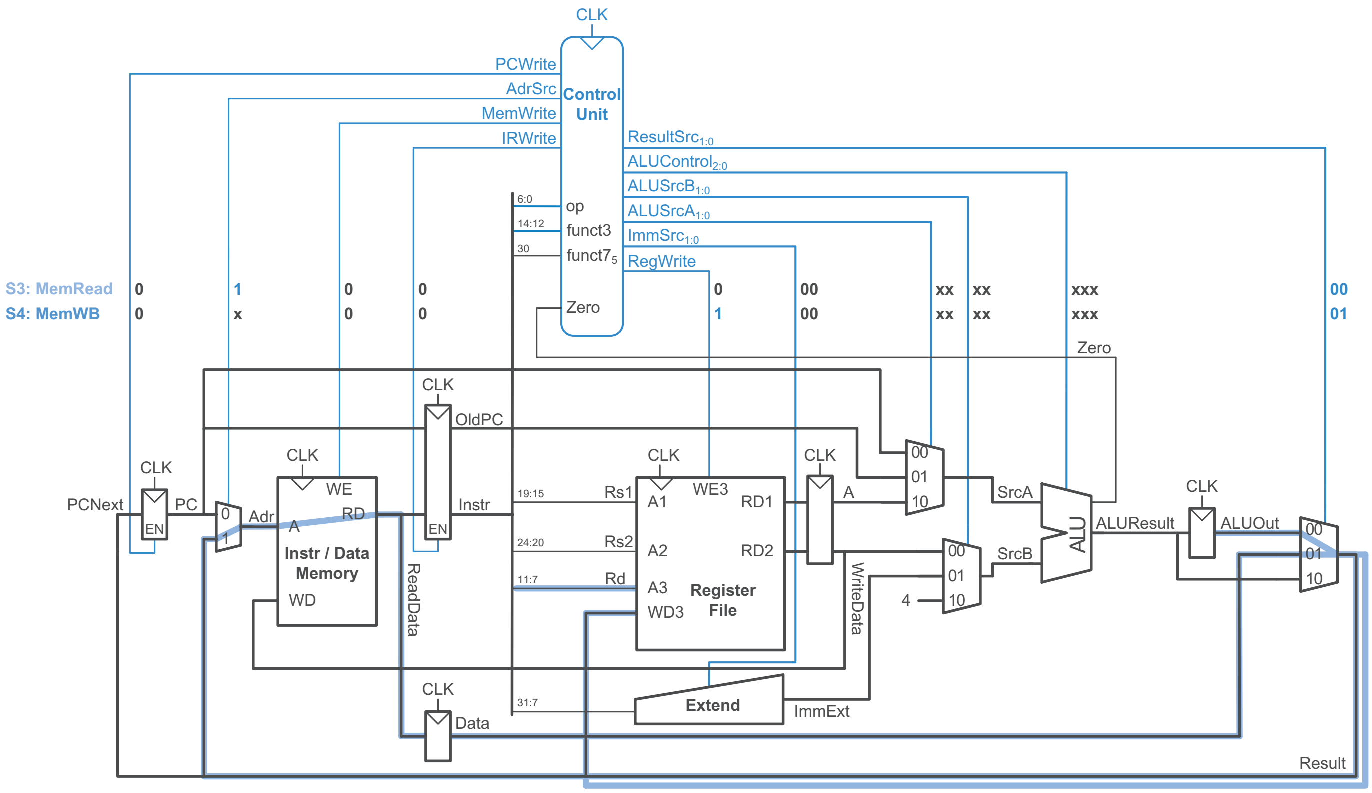


Figure 7.34 Data flow during MemRead and MemWB

to the Fetch state so that the next instruction can be fetched. However, the PC has not yet been incremented. We address this issue next.

Before finishing the `lw` instruction, the processor must increment the PC so that it can fetch the next instruction. We could add another state for doing this but, instead, we save a cycle by noticing that the ALU is not being used in the Fetch step, so the processor can use that state to calculate PC+4 at the same time that it fetches the instruction. *ALUSrcA* = 00 to get *SrcA* from the PC (i.e., from signal *OldPC*). *ALUSrcB* = 10 to get the constant 4 for *SrcB*. *ALUOp* = 00, so that the ALU adds PC to 4. To update the PC with PC+4, *ResultSrc* = 10 to choose *ALUResult* as the *Result*, and *PCUpdate* = 1 to force *PCWrite* high (see Figure 7.28). Figure 7.35 shows the modified Fetch state. The rest of the diagram remains the same as in Figure 7.33. Figure 7.36 highlights in blue the data flow for computing PC+4. The instruction fetch, which is occurring simultaneously, is highlighted in gray.

We started this section stating that only one of the time-consuming units (the memory, ALU, or register file) could be used in each step. However, here, we use *both* the register file and ALU. As long as the units are used at the same time—that is, in parallel—more than one unit can be used in a single step.

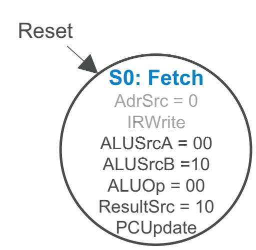


Figure 7.35 Incrementing PC in the Fetch state

`sw`

Now, we expand the Main FSM to handle more RISC-V instructions. All instructions pass through the first two states, Fetch and Decode.

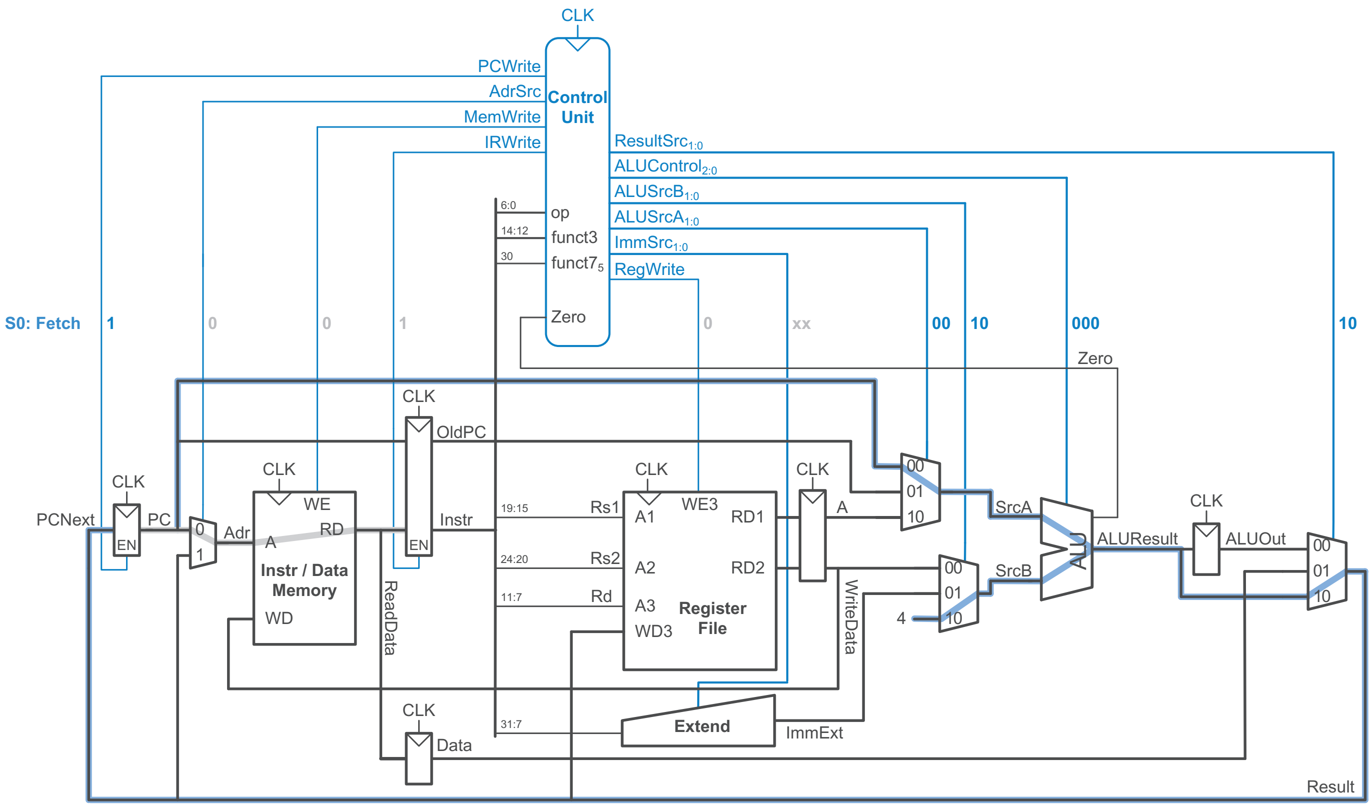


Figure 7.36 Data flow while incrementing PC in the Fetch state

The `sw` instruction uses the same MemAdr state as `lw` to calculate the memory address but then proceeds to the memory write (MemWrite) state, where *WriteData*, the value from **rs2**, is written to memory. *WriteData* is hardwired to the memory's write data port (*WD*). The memory's address port, *Adr*, is set to the calculated address in *ALUOut* by making *ResultSrc* = 00 and *AdrSrc* = 1. *MemWrite* is asserted to write the memory. This completes the `sw` instruction, so the Main FSM returns to the Fetch state to begin the next instruction. Figures 7.37 and 7.38 show the expanded Main FSM and the datapath flow for the MemWrite state. Note that the first two states of the FSM (Fetch and Decode), which are not shown in Figure 7.37, are the same as in Figure 7.33.

The *ImmSrc* signal differs for `lw` and `sw` in the MemAdr state. But recall that *ImmSrc* is produced combinationally by the Instruction Decoder (see Figure 7.28).

R-Type Instructions

After the Decode state, R-type ALU instructions proceed to the execute (ExecuteR) state, which performs the desired ALU computation. Namely, *ALUSrcA* = 10 and *ALUSrcB* = 00 to choose the contents of **rs1** as *SrcA* and **rs2** as *SrcB*. *ALUOp* = 10 so that the ALU Decoder uses the instruction's control fields to determine what operation to perform.

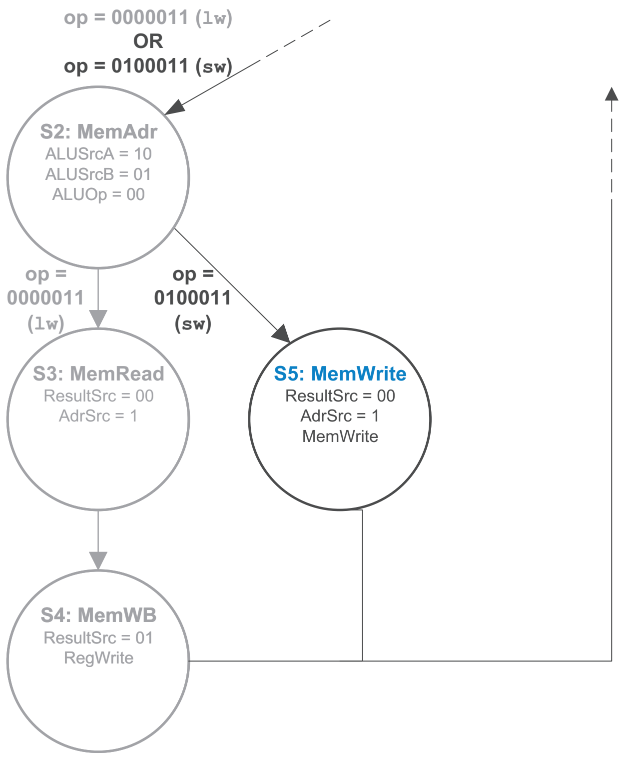


Figure 7.37 Memory write

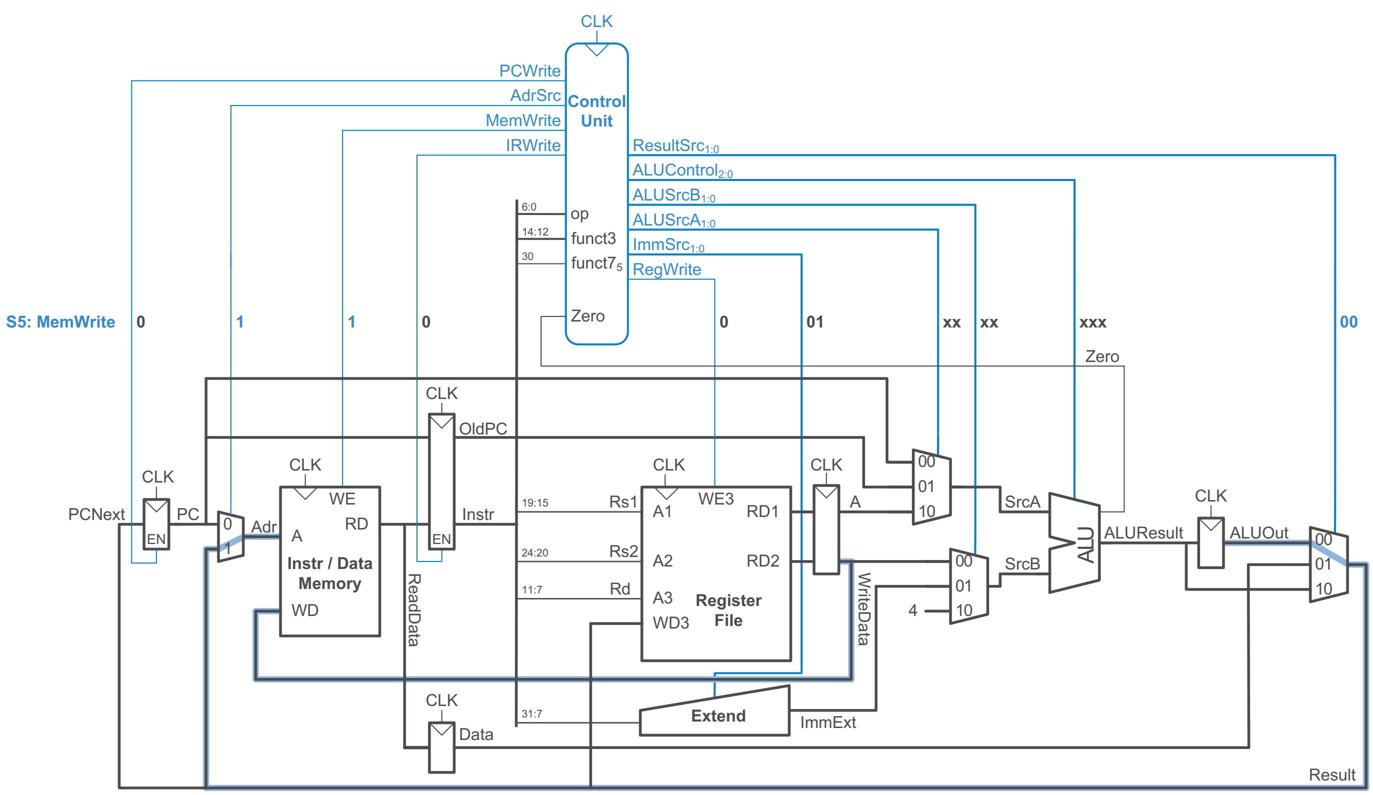


Figure 7.38 Data flow during the memory write (MemWrite) state

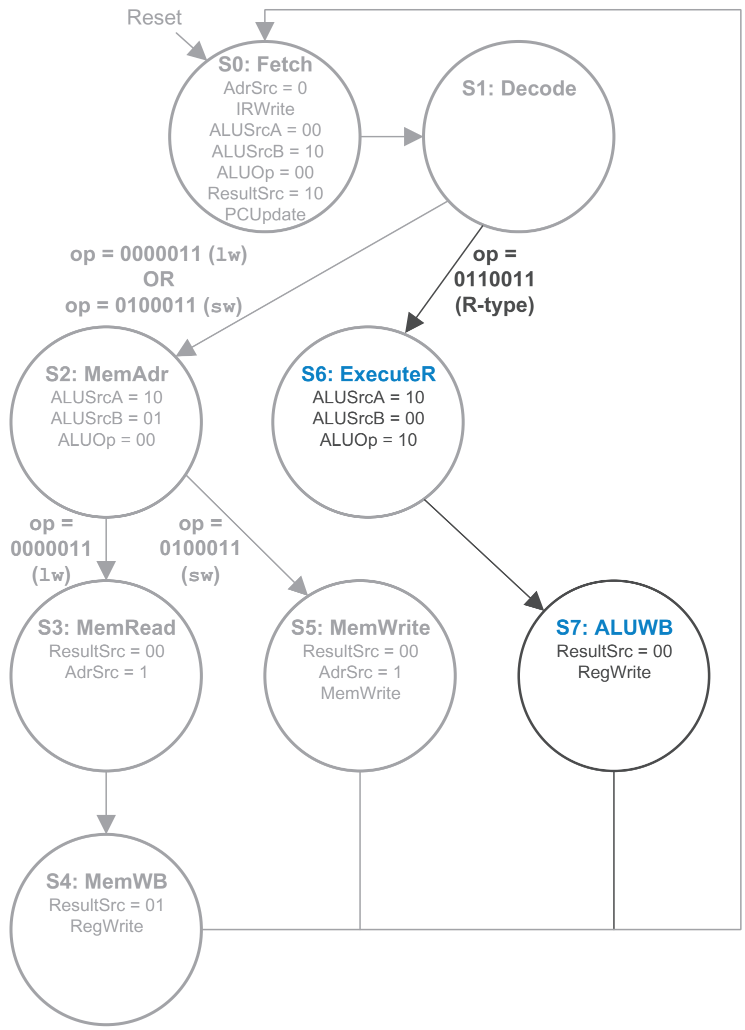


Figure 7.39 **Execute R-type (ExecuteR) and ALU writeback (ALUWB) states**

ALUResult is written to the ALUOut register at the end of the cycle. R-type instructions then go to the ALU writeback (ALUWB) state where the computation result, *ALUOut*, is written back to the register file. In the ALUWB state, *ResultSrc* = 00 to select *ALUOut* as *Result*, and *RegWrite* = 1 so that **rd** is written with the result. Figure 7.39 shows the ExecuteR and ALUWB states added to the Main FSM. Figure 7.40 shows the data flow during both states, with ExecuteR data flow shown in thick light-blue lines and ALUWB data flow in thick dark-blue lines.

`beq`

The final instruction, `beq`, compares two registers and computes the branch target address. Thus far, the ALU is idle during the Decode state, so we can use the ALU during that state to calculate the branch target

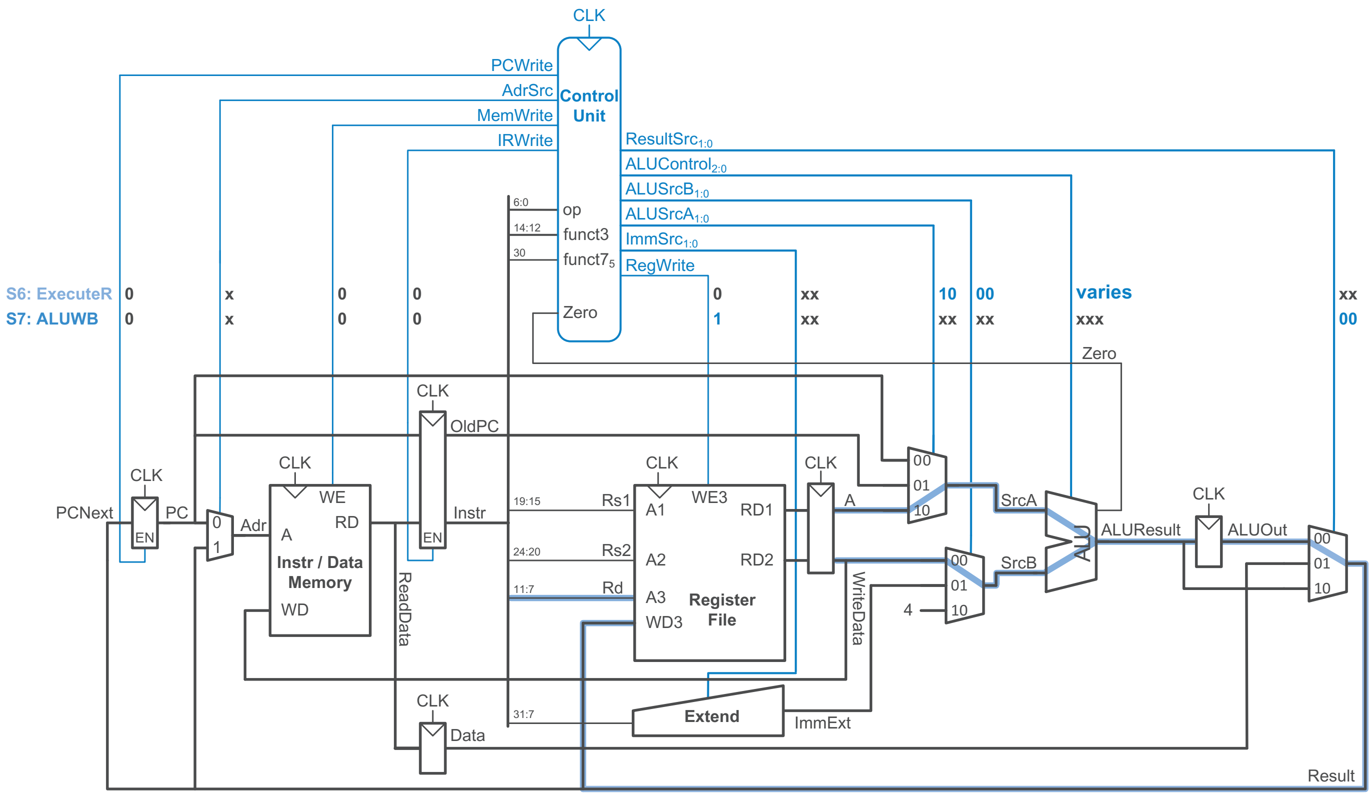


Figure 7.40 Data flow during the ExecuteR and ALUWB states

address, *OldPC* + *ImmExt*. *ALUSrcA* and *ALUSrcB* are both 01 so that *OldPC* is *SrcA* and the branch offset (*ImmExt*) is *SrcB*. *ALUOp* = 00 to make the ALU add. The target address is stored in the ALUOut register at the end of the Decode state. Figure 7.41 shows the enhanced Decode state as well as the subsequent BEQ state, which is discussed next. In Figure 7.42, the data flow during the Decode state is shown in light-blue and gray lines. The branch target address calculation is highlighted in light blue, and the register read and immediate extension is highlighted with thick gray lines.

After the Decode state, `beq` proceeds to the BEQ state, where it compares the source registers. *ALUSrcA* = 10 and *ALUSrcB* = 00 to select the values read from the register file as *SrcA* and *SrcB*. *ALUOp* = 01 so that the ALU performs subtraction. If the source registers are equal, the ALU's *Zero* output asserts (because **rs1** − **rs2** = 0). *Branch* = 1 in this state so that if *Zero* is also set, *PCWrite* is asserted (as shown in the *PCWrite* logic of Figure 7.28) and the branch target address (in *ALUOut*) becomes the next PC. *ALUOut* is routed to the PC register by *ResultSrc* being 00. Figure 7.41 shows the BEQ state, and Figure 7.42

Even though the instruction is not yet decoded at the beginning of the Decode state—and it may not even be a `beq` instruction—the branch target address is calculated as if it were a branch. If it turns out that the instruction is not a branch or if the branch is not taken, the resulting calculation is simply not used.

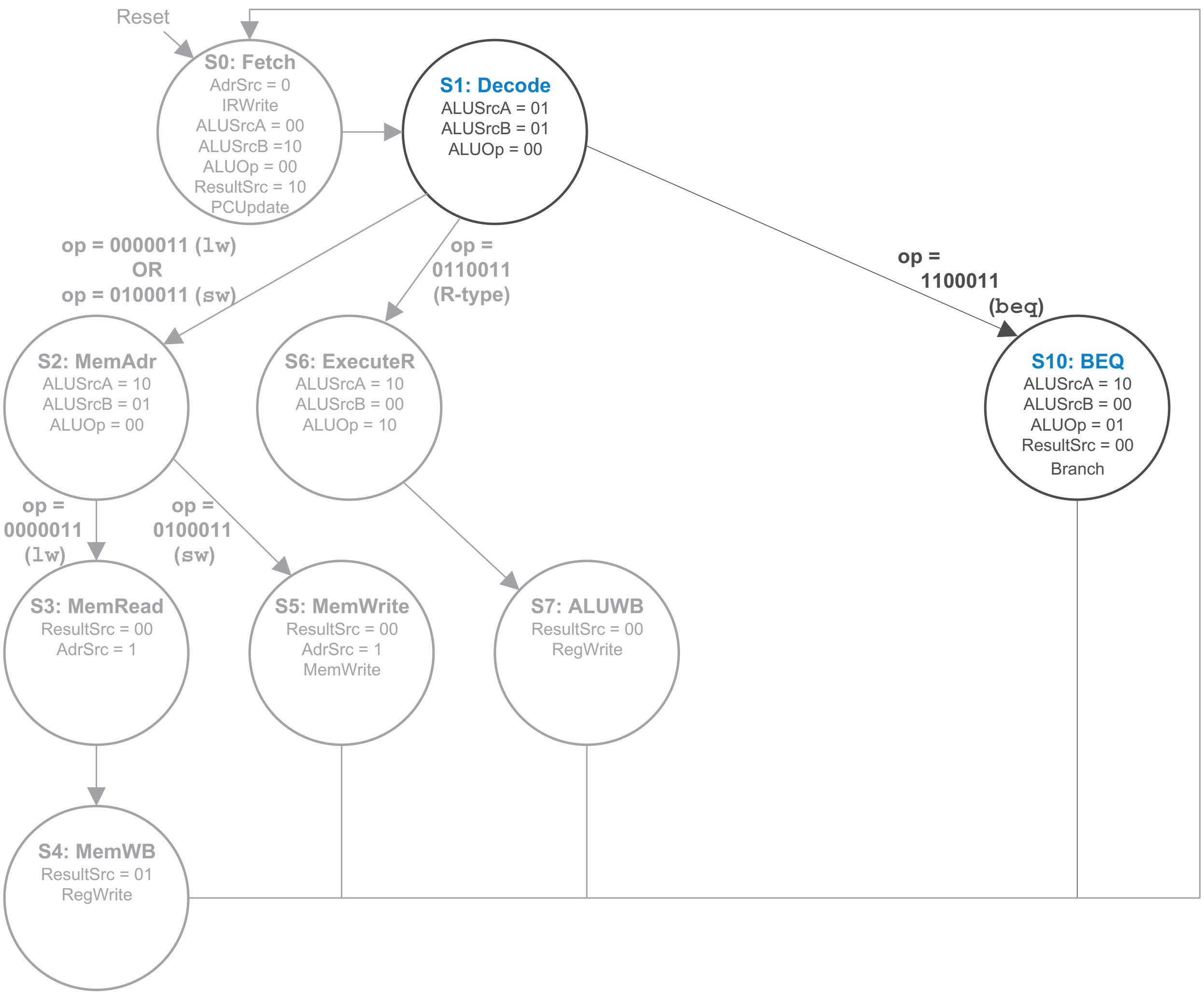


Figure 7.41 Enhanced Decode state, with branch target address calculation, and BEQ state

shows the data flow during the BEQ state. The path for comparing **rs1** and **rs2** is shown in dark blue and the path to (conditionally) set PC to the target address is in gray through the Result register. This concludes the design of the controller for these instructions.

7.4.3 More Instructions

As we did with the single-cycle processor, we next consider examples of how to modify the multicycle processor datapath and controller to handle new instructions: I-type ALU instructions (`addi`, `andi`, `ori`, `slti`) and `jal`.

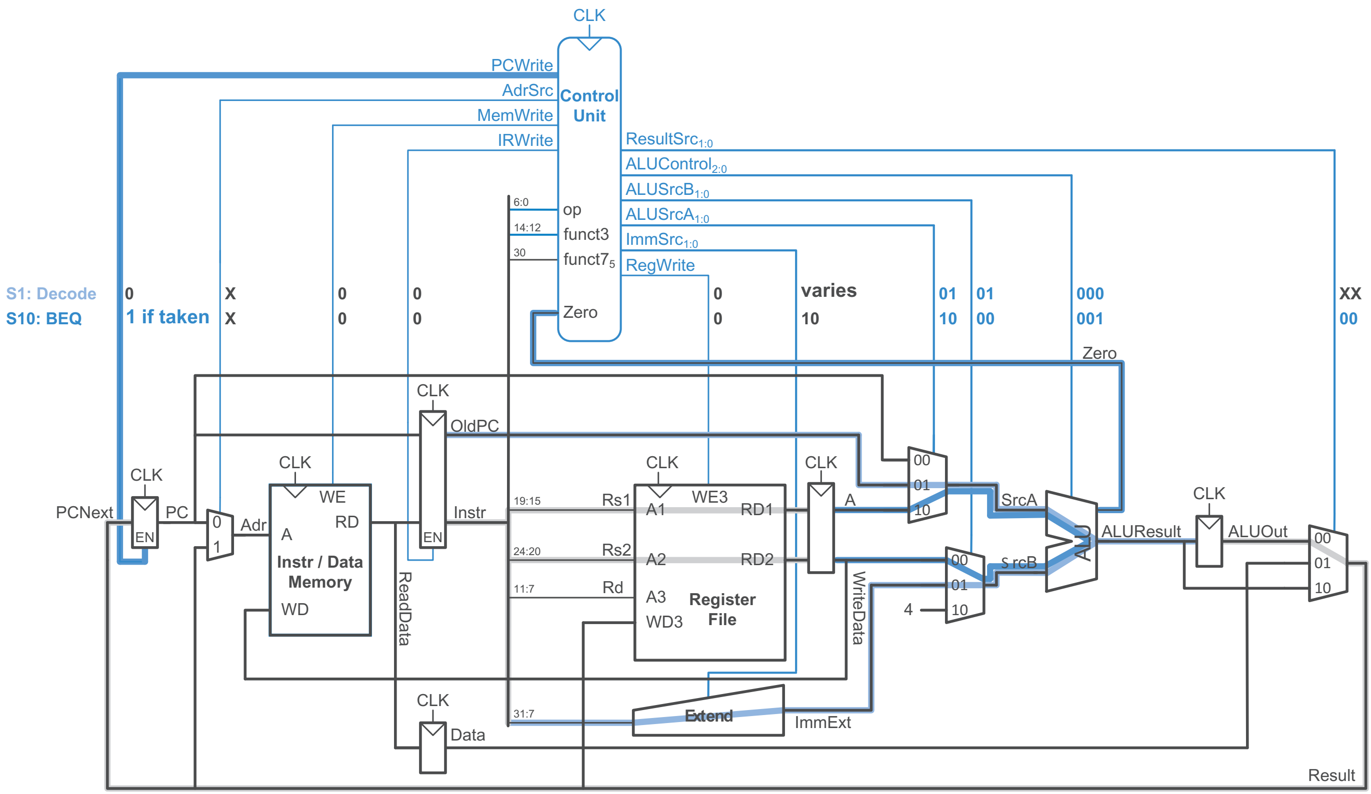


Figure 7.42 Data flow during Decode and BEQ states

Example 7.5 EXPANDING THE MULTICYCLE PROCESSOR TO INCLUDE I-TYPE ALU INSTRUCTIONS

Expand the multicycle processor to include I-type ALU instructions `addi`, `andi`, `ori`, and `slti`.

Solution These I-type ALU instructions are nearly the same as their R-type equivalents (`add`, `and`, `or`, and `slt`) except that the second source comes from *ImmExt* rather than the register file. We introduce the ExecuteI state to perform the desired computation for all I-type ALU instructions. This state is like ExecuteR except that *ALUSrcB* = 01 to choose *ImmExt* as *SrcB*. After the ExecuteI state, I-type ALU instructions proceed to the ALU writeback (ALUWB) state to write the result to the register file. Figure 7.43 shows the enhanced Main FSM, which also includes the JAL state for Example 7.6.

Example 7.6 EXPANDING THE MULTICYCLE PROCESSOR TO INCLUDE `jal`

Expand the multicycle processor to include the jump and link instruction (`jal`).

Solution Like the I-type ALU instructions from Example 7.5, no additional hardware is needed to implement the `jal` instruction. Only the Main FSM needs to be

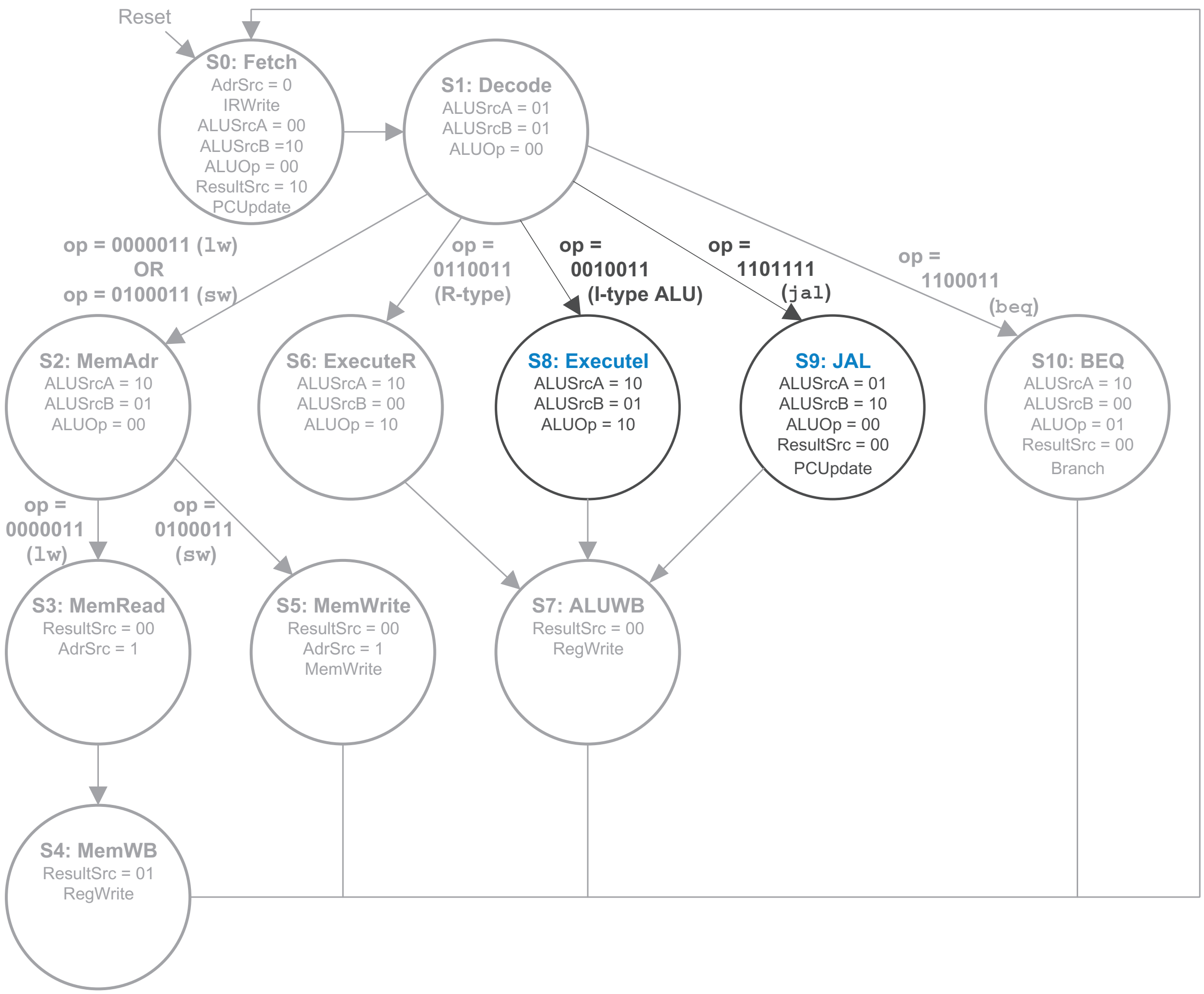


Figure 7.43 **Enhanced Main FSM: ExecuteI and JAL states**

You may notice that by the time the processor reaches the JAL state, the PC register has already been updated to PC+4. So we could have just used the PC register output to write to **rd**. But that would have required us to extend the Result multiplexer to receive *PC* as an input. The solution above requires less hardware because it uses the existing datapath.

updated. The first two steps are the same as the other instructions. During the Decode state, the jump target address is calculated using the same flow as the branch target address calculation but with *ImmSrc* = 11, as set by the Instruction Decoder. Thus, during the Decode state, the jump offset is sign-extended and added to the current PC (contained in signal *OldPC*) to form the jump target address, which is written to the ALUOut register at the end of that state. `jal` then proceeds to the JAL state, where the processor writes the target address to PC and calculates the return address (PC+4) so that it can write it to **rd** in the next state. The ALU calculates PC+4 (i.e., *OldPC*+4) using *ALUSrcA* = 01 (*SrcA* = *OldPC*), *ALUSrcB* = 10 (*SrcB* = 4), and *ALUOp* = 00 for addition. To write the target address to the PC, *ResultSrc* = 00 to select the target address (held in *ALUOut*) as *Result*, and *PCUpdate* = 1 so that *PCWrite* is asserted. Figure 7.43

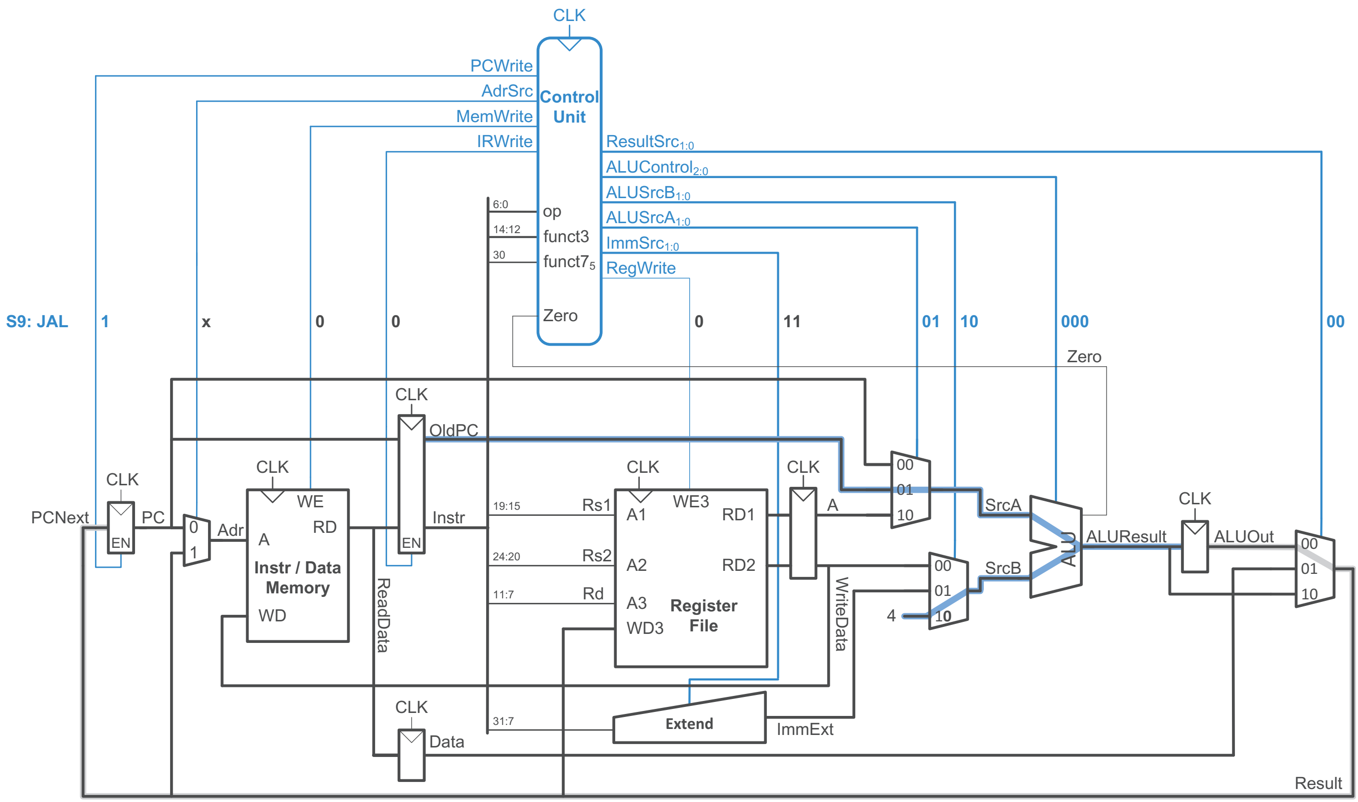


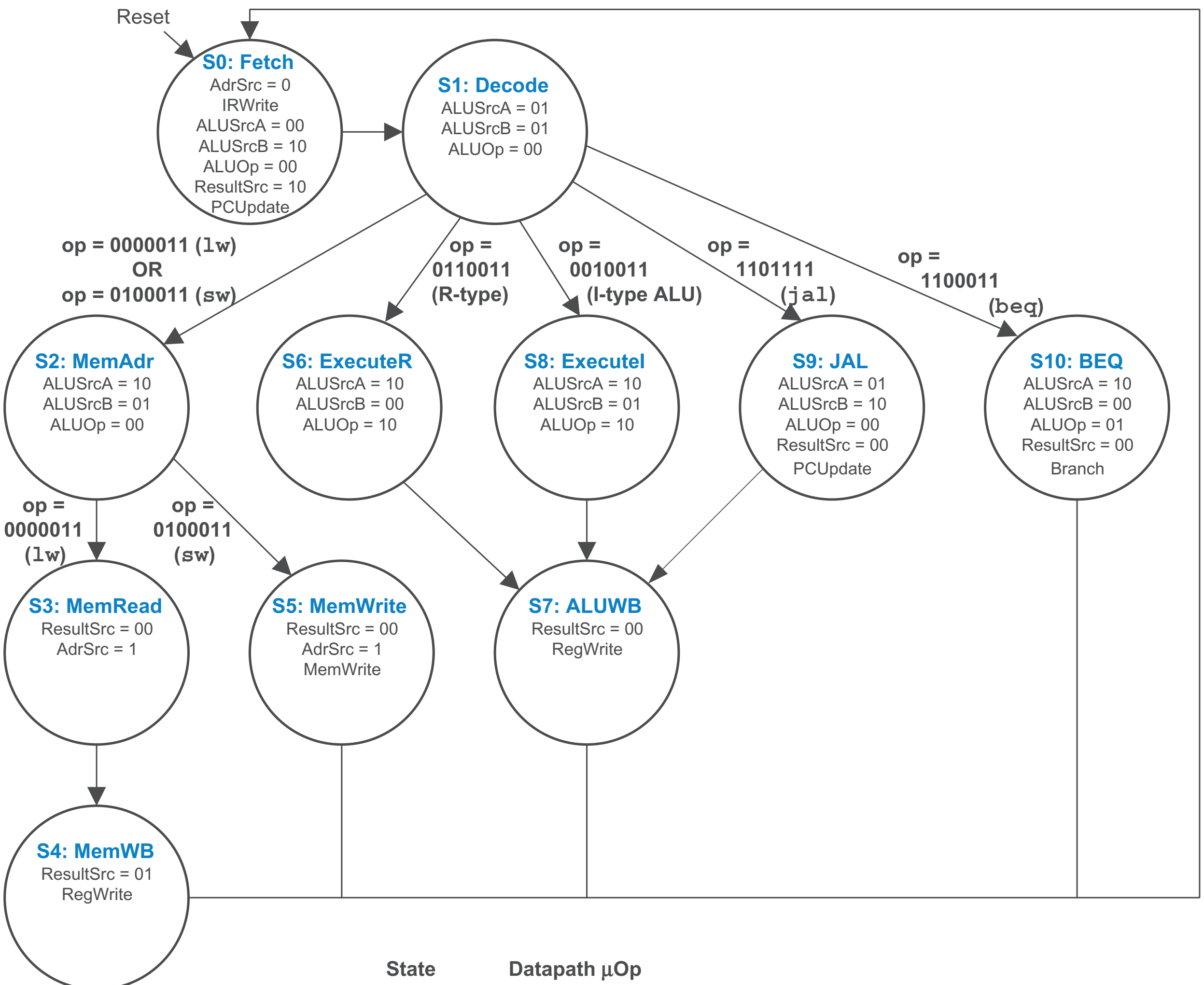
Figure 7.44 Data flow during the JAL state

shows the new JAL state, and Figure 7.44 shows the data flow during the JAL state. The flow for updating the PC to the target address is in gray and the PC+4 calculation is in blue. After the JAL state, `jal` proceeds to the ALUWB state, where the return address (ALUOut = PC+4) is written to rd. This concludes the `jal` instruction, so the Main FSM then goes back to the Fetch state.

Putting these steps together, Figure 7.45 shows the complete Main FSM state transition diagram for the multicycle processor. The function of each state is summarized below the figure. Converting the diagram 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.

7.4.4 Performance Analysis

The execution time of an instruction depends on both the number of cycles it uses and the cycle time. While the single-cycle processor performed all instructions in one cycle, the multicycle processor uses



State	Datapath µOp
Fetch	Instr ←Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1oprs2
ExecuteI	ALUOut ← rs1opimm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4

Figure 7.45 Complete multicycle control FSM

varying numbers of cycles for different 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 branches, four for R-type, I-type ALU, jump, and store instructions, and five for loads. The number of clock cycles per instruction (CPI) depends on the relative likelihood that each instruction is used.

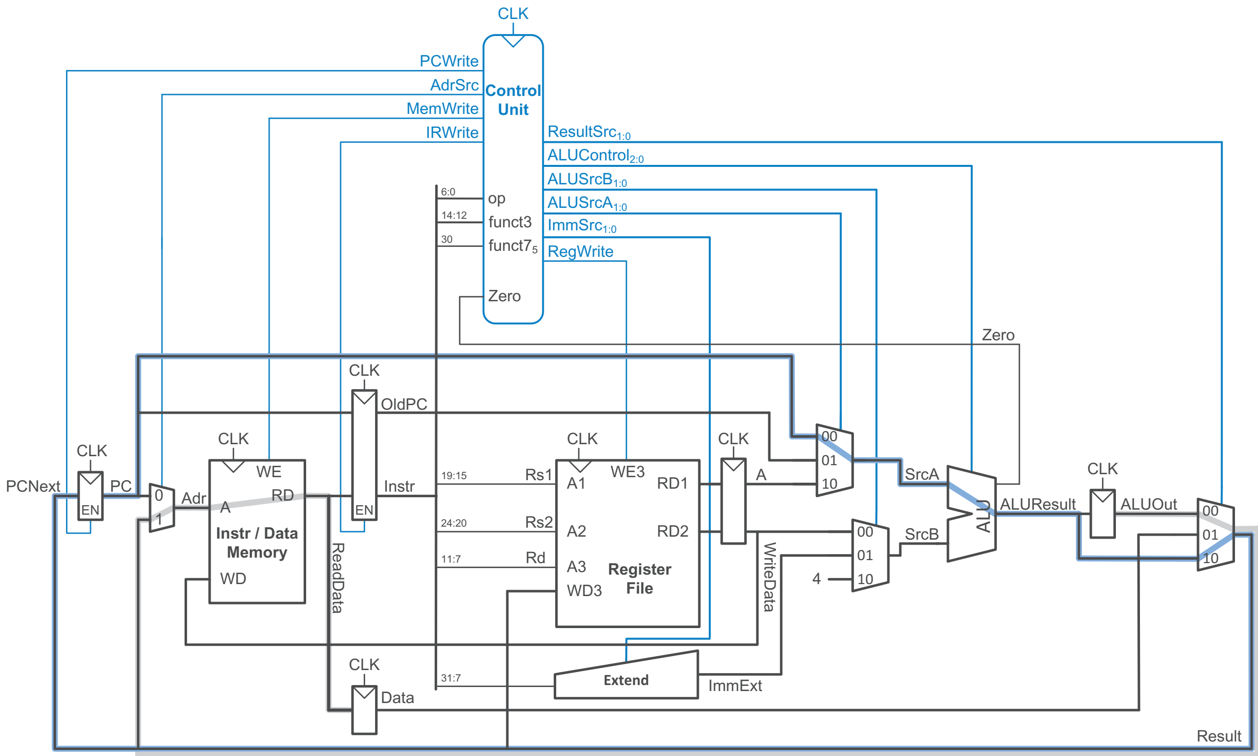


Figure 7.46 Multicycle processor potential critical paths

Example 7.7 MULTICYCLE PROCESSOR CPI

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

Solution The average CPI is the weighted sum over each instruction of the CPI for that instruction multiplied by the fraction of time that instruction is used. For this benchmark, average CPI = (0.11)(3) + (0.10 + 0.02 + 0.52)(4) + (0.25)(5) = 4.14. This is better than the worst-case CPI of 5, which would be required if all instructions took the same number of cycles.

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, as shown in Figure 7.46:

1. **The path to calculate PC+4:** From the PC register through the SrcA multiplexer, ALU, and Result multiplexer back to the PC register (highlighted in thick blue lines); or

---

[2] Instruction frequencies from Patterson and Hennessy, *Computer Organization and Design*, 4th Edition, Morgan Kaufmann, 2011.

2. **The path to read data from memory:** From the ALUOut register through the Result and Adr muxes to read memory into the Data register (highlighted in thick gray lines)

Both of these paths also require a delay through the decoder after the state updates (i.e., after a t_{pcq} delay) to produce the control (multiplexer select and *ALUControl*) signals. Thus, the clock period is given in Equation 7.4.

$$T_{c\_multi} = t_{pcq} + t_{dec} + 2t_{mux} + \max[t_{ALU}, t_{mem}] + t_{setup} \quad (7.4)$$

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

Example 7.8 MULTICYCLE PROCESSOR PERFORMANCE COMPARISON.

Ben Bitdiddle is wondering whether the multicycle processor would be faster than the single-cycle processor. For both designs, he plans on using the 7-nm CMOS manufacturing process with the delays given in Table 7.7 on page 415. Help him compare each processor's execution time for 100 billion instructions from the SPECINT2000 benchmark (see Example 7.4).

Solution According to Equation 7.4, the cycle time of the multicycle processor is $T_{c\_multi} = t_{pcq} + t_{dec} + 2t_{mux} + t_{mem} + t_{setup} = 40 + 25 + 2(30) + 200 + 50 = 375$ ps. Using the CPI of 4.14 from Example 7.7, the total execution time is $T_{multi} = (100 \times 10^9$ instructions)(4.14 cycles/instruction)(375×10^{-12} s/cycle) = 155 seconds.

According to Example 7.4, the single-cycle processor had a total execution time of 75 seconds, so the multicycle processor is slower.

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 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 90-ps sequencing overhead of the register clock-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 shares a single memory for instructions and data and because it eliminates two adders. 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 approximately five times faster. So, ideally, the latency of each instruction is unchanged, but the throughput is 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 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.

Recall that *throughput* is the number of tasks (in this case, instructions) that complete per second. *Latency* is the time it takes for a given instruction to complete, from start to finish. (See Section 3.6)

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, if applicable. Finally, in the *Writeback* stage, the processor writes the result to the register file, if applicable.

Figure 7.47 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 component delays from Table 7.7 (see page 415) but ignores multiplexers and registers for simplicity. In the single-cycle processor in Figure 7.47(a), the first instruction is read from memory at time 0. Next, the operands are read from the register file. Then, the ALU executes the necessary computation. Finally, the data memory may be accessed, and the result is written back to the register file at 680 ps. The second instruction begins when the first completes. Hence, in this diagram, the single-cycle processor has an instruction latency of 200 + 100 + 120 + 200 + 60 = 680 ps (see Table 7.7 on page 415) and a throughput of 1 instruction per 680 ps (1.47 billion instructions per second).

Remember that for this abstract comparison of single-cycle and pipelined processor performance, we are ignoring the overhead of decoder, multiplexer, and register delays.

In the pipelined processor in Figure 7.47(b), the length of a pipeline stage is set at 200 ps by the slowest stage, the memory access in the Fetch or Memory stage. Each pipeline stage is indicated by solid or dashed vertical blue lines. At time 0, the first instruction is fetched from memory. At 200 ps, the first instruction enters the Decode stage, and a second instruction is fetched. At 400 ps, the first instruction executes, the second instruction enters the Decode stage, and a third instruction is fetched.

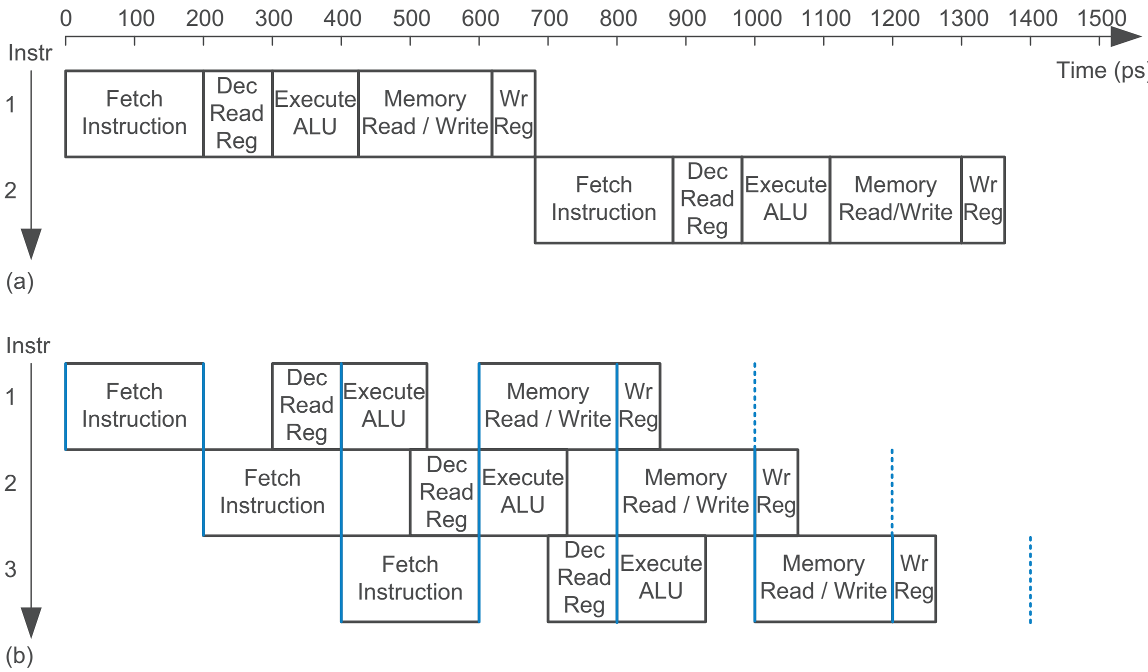


Figure 7.47 **Timing diagrams: (a) single-cycle processor and (b) pipelined processor**

And so forth, until all the instructions complete. The instruction latency is $5 \times 200 = 1000$ ps. Because the stages are not perfectly balanced with equal amounts of logic, the latency is longer for the pipelined processor than for the single-cycle processor. The throughput is 1 instruction per 200 ps (5 billion instructions per second)—that is, one instruction completes every clock cycle. This throughput is 3.4 times as much as the single-cycle processor—not quite 5 times but, nonetheless, a substantial speedup.

Figure 7.48 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 writeback—to illustrate the flow of instructions through the pipeline. Reading across a row shows the clock cycle 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 register file is writing a sum to `s3`, the data memory is idle, the ALU is computing `(s11 & t0)`, `t4` is being read from the register file, and the `or` instruction is being fetched from instruction memory. 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 every instruction except `sw`. In the pipelined processor, the register file is

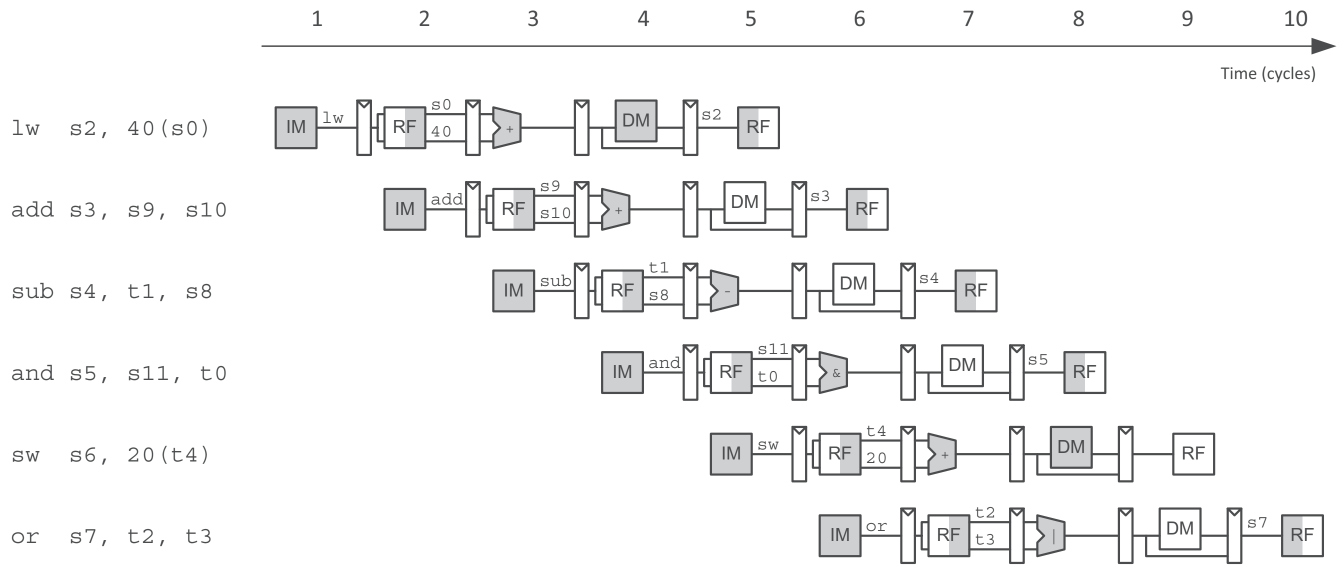


Figure 7.48 **Abstract view of pipeline in operation**

used twice in every cycle: it 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 by one instruction and read by another within a single cycle.

A central challenge in pipelined systems is handling hazards that occur when one instruction's result is needed by a subsequent instruction before the former instruction has completed. For example, if the `add` in Figure 7.48 used `s2` as a source instead of `s10`, a hazard would occur because the `s2` register has not yet been written by the `lw` instruction when it is read by `add` in cycle 3. After designing the pipelined datapath and control, 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 7.49(a) shows the single-cycle datapath stretched out to leave room for the pipeline registers. Figure 7.49(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. So, although the register file is drawn in the Decode stage, its write address and write data come from the Writeback stage. This feedback will lead to pipeline hazards, which are discussed in

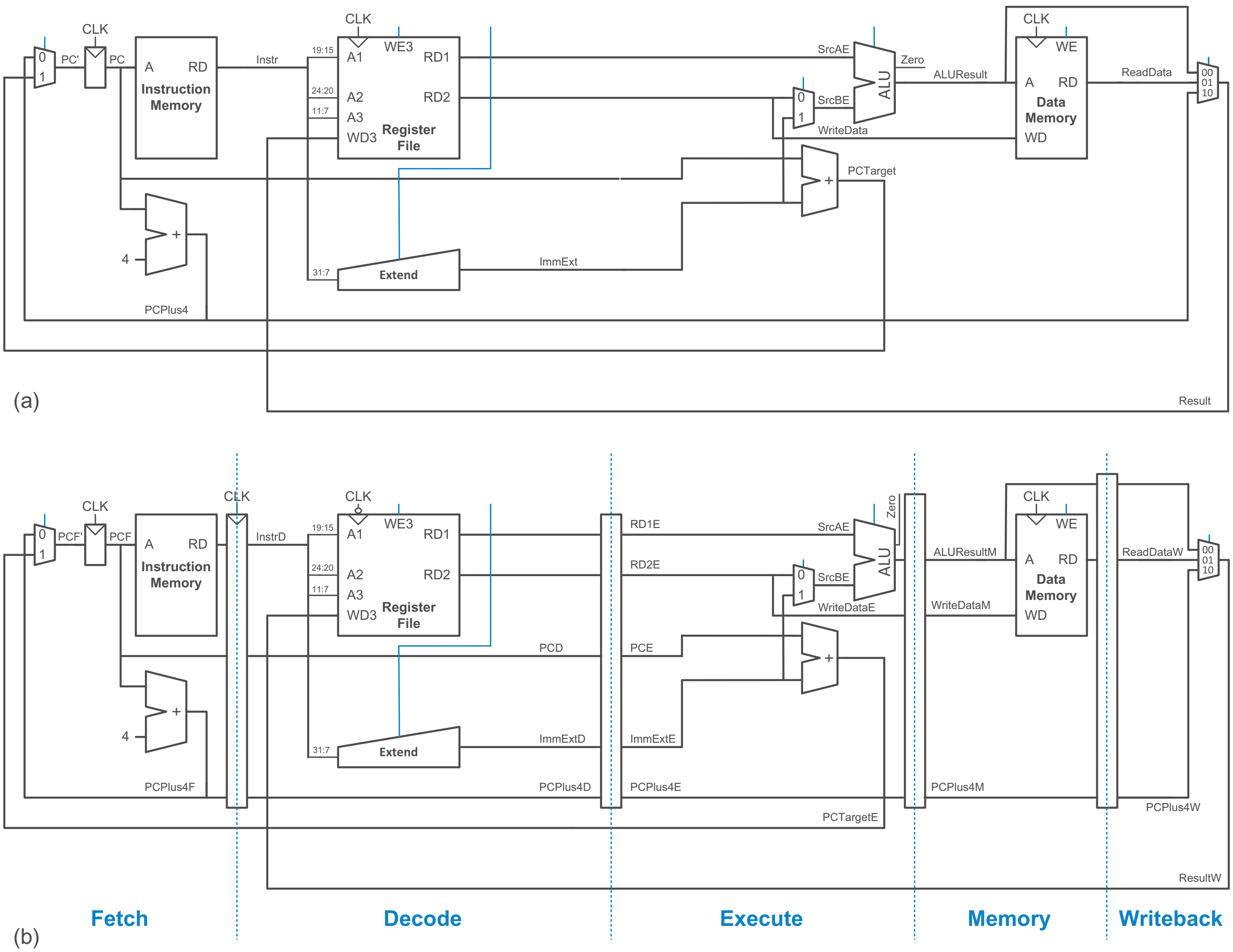


Figure 7.49 **Datapaths: (a) single-cycle and (b) pipelined**

Section 7.5.3. The register file in the pipelined processor writes on the falling edge of *CLK* so that it can write a result in the first half of a cycle and read that result in the second half of the cycle for use in a subsequent instruction.

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 7.49(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 destination register comes from *RdD* ($InstrD_{11:7}$), which is a Decode stage signal. In the pipeline diagram of Figure 7.48, during cycle 5, the result of the `lw` instruction would be incorrectly written to `s5` rather than `s2`.

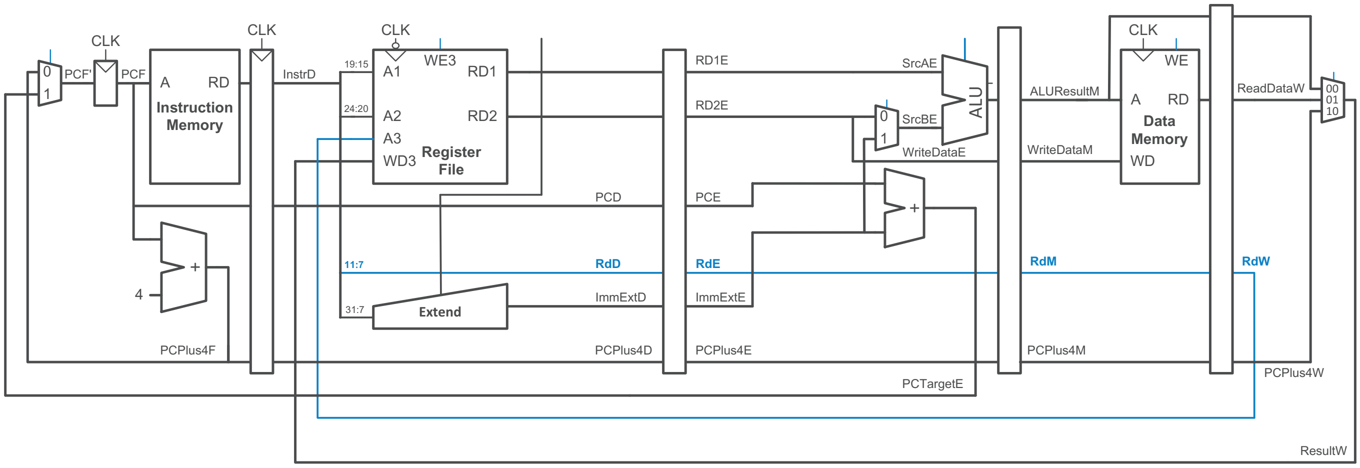


Figure 7.50 Corrected pipelined datapath

Figure 7.50 shows a corrected datapath, with the modification in blue. The *Rd* signal is now pipelined along through the Execution, Memory, and Writeback stages, so it remains in sync with the rest of the instruction. *RdW* and *ResultW* are fed back together to the register file in the Writeback stage.

The astute reader may note that the logic to produce *PCF'* (the next PC) is also problematic because it could be updated with either a Fetch or an Execute stage signal (*PCPlus4F* or *PCTargetE*). This control hazard will be fixed in Section 7.5.3.

7.5.2 Pipelined Control

The pipelined processor uses the same control signals as the single-cycle processor and, therefore, has the same control unit. The control unit examines the **op**, **funct3**, and **funct7**$_5$ fields of the instruction in the Decode stage to produce the control signals, as was described in Section 7.3.3 for the single-cycle processor. 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 7.51. *RegWrite* must be pipelined into the Writeback stage before it feeds back to the register file, just as *Rd* was pipelined in Figure 7.50. In addition to R-type ALU instructions, `lw`, `sw`, and `beq`, this pipelined processor also supports `jal` and I-type ALU instructions.

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

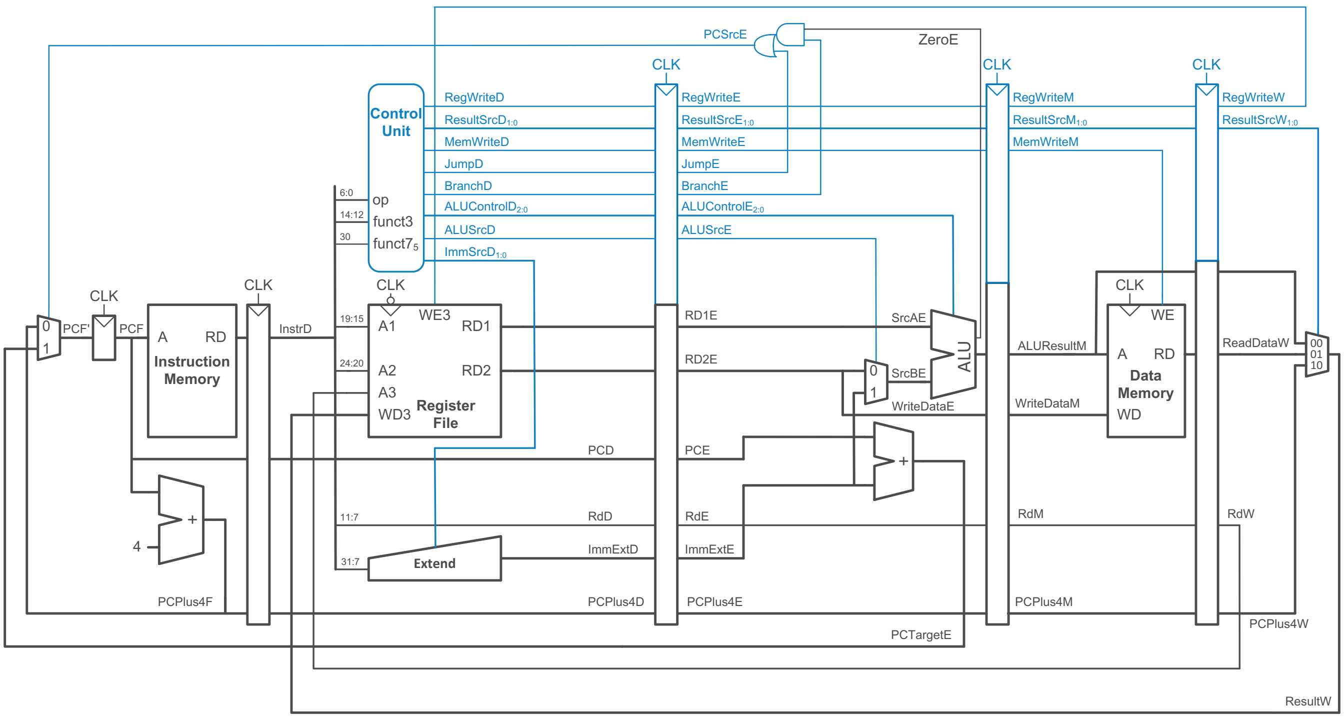


Figure 7.51 **Pipelined processor with control**

not yet completed, a *hazard* occurs. The register file is written during the first half of the cycle and read during the second half of the cycle, so a register can be written and read back in the same cycle without introducing a hazard.

Figure 7.52 illustrates hazards that occur when one instruction writes a register (`s8`) and subsequent instructions read this register. The blue arrows highlight when `s8` is written to the register file (in cycle 5) as compared to when it is needed by subsequent instructions. This is called a *read after write* (*RAW*) *hazard*. The `add` instruction writes a result into `s8` in the first half of cycle 5. However, the `sub` instruction reads `s8` on cycle 3, obtaining the wrong value. The `or` instruction reads `s8` on cycle 4, again obtaining the wrong value. The `and` instruction reads `s8` 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 `s8`. The diagram shows that hazards may occur in this pipeline when an instruction writes a register and either of the two subsequent instructions reads that register. Without special treatment, the pipeline will compute the wrong result.

A software solution would be to require the programmer or compiler to insert nop instructions between the `add` and `sub` instructions so that the dependent instruction does not read the result (`s8`) until it is available

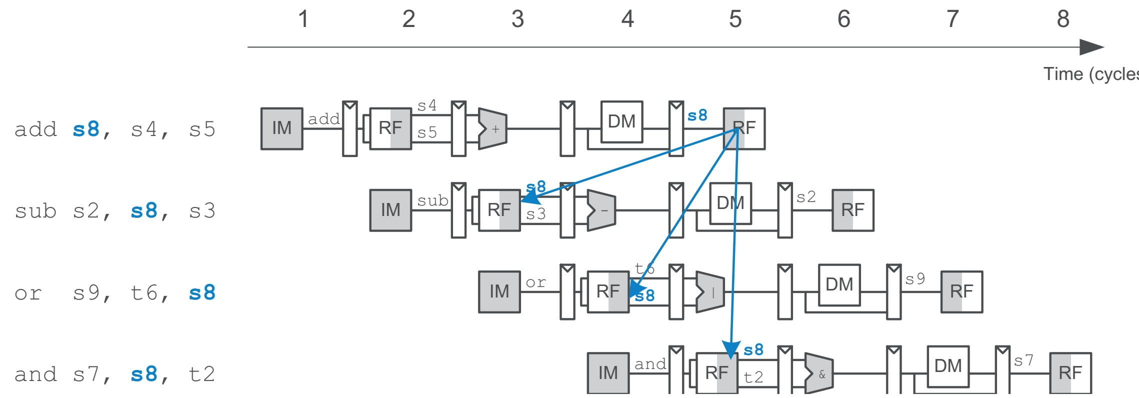


Figure 7.52 **Abstract pipeline diagram illustrating hazards**

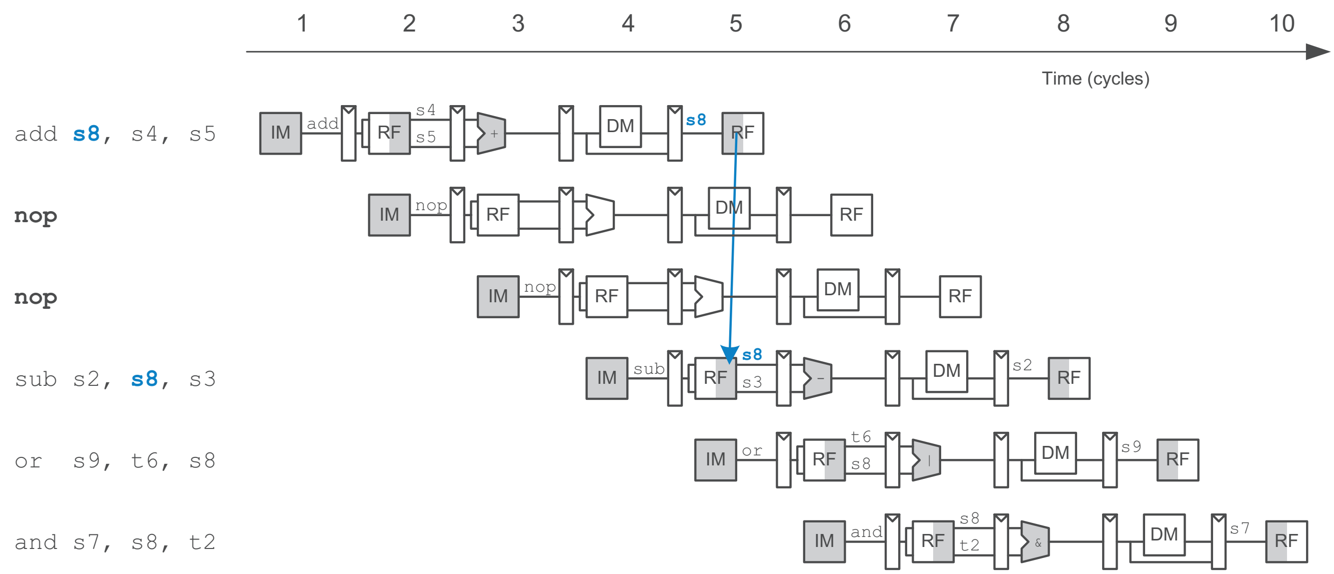


Figure 7.53 **Solving data hazard with nops**

in the register file, as shown in Figure 7.53. Such a *software interlock* complicates programming and degrades performance, so it is not ideal.

On closer inspection, observe from Figure 7.52 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 waiting for the result to appear in the register file and 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 produced 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.

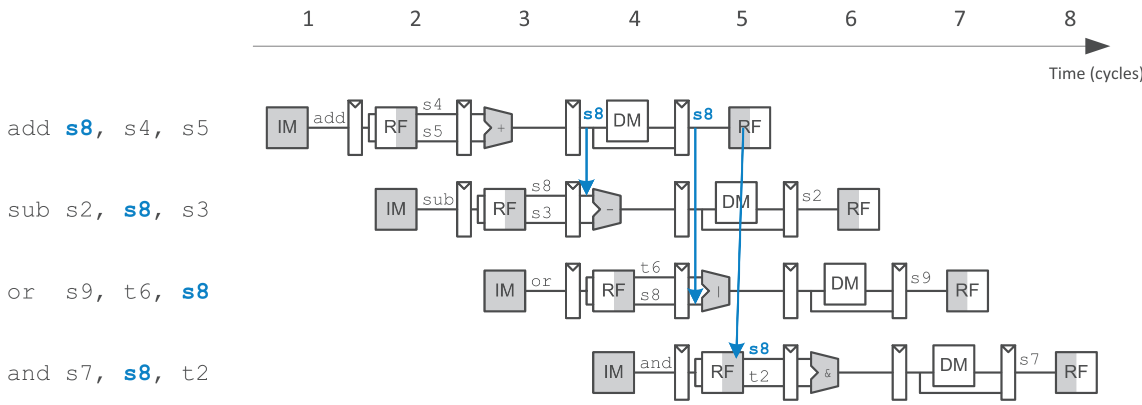


Figure 7.54 **Abstract pipeline diagram illustrating forwarding**

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 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 its operands from the register file or the Memory or Writeback stage. Figure 7.54 illustrates this principle. This program computes `s8` with the `add` instruction and then uses `s8` in the three subsequent instructions. In cycle 4, `s8` is forwarded from the Memory stage of the `add` instruction to the Execute stage of the dependent `sub` instruction. In cycle 5, `s8` is forwarded from the Writeback stage of the `add` instruction to the Execute stage of the dependent `or` instruction. Again, no forwarding is needed for the `and` instruction because `s8` is written to the register file in the first half of cycle 5 and read in the second half.

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 7.55 modifies the pipelined processor to support forwarding. It adds a *Hazard Unit* and two *forwarding multiplexers*. The hazard detection unit receives the two source registers from the instruction in the Execute stage, *Rs1E* and *Rs2E*, and the destination registers from the instructions in the Memory and Writeback

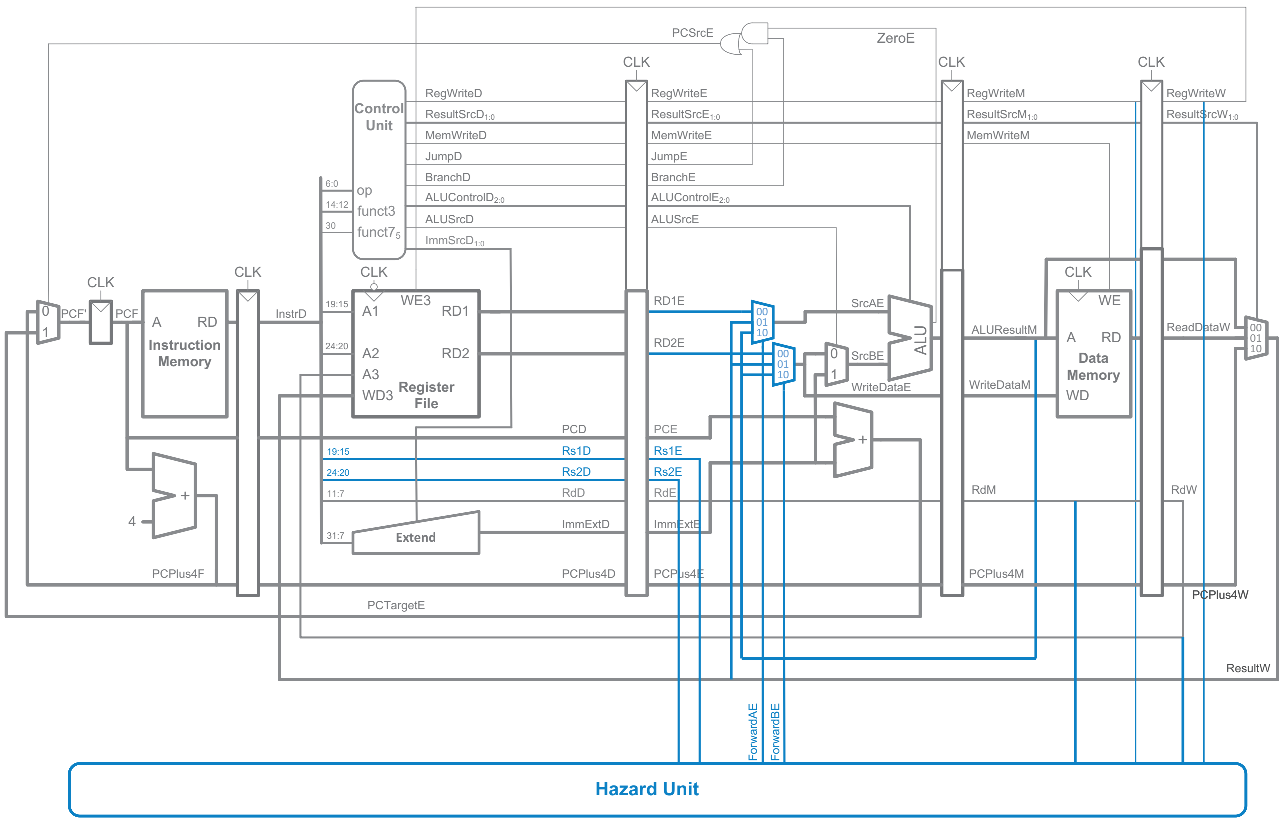


Figure 7.55 Pipelined processor with forwarding to solve some data hazards

stages, *RdM* and *RdW*. It also receives the *RegWrite* signals from the Memory and Writeback stages (*RegWriteM* and *RegWriteW*) to know whether the destination register will actually be written (e.g., the `sw` and `beq` instructions do not write results to the register file and, hence, do not have their results forwarded).

The Hazard 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 (*ALUResultM* or *ResultW*). The Hazard Unit should forward from a stage if that stage will write a destination register *and* the destination register matches the source register. However, `x0` is hardwired to 0 and should never be forwarded. If both the Memory and Writeback stages contain matching destination registers, then the Memory stage should have priority because it contains the more recently executed instruction. In summary, the function of the forwarding logic for *SrcAE* (*ForwardAE*) is given on the next page. The forwarding logic for *SrcBE* (*ForwardBE*) is identical except that it checks *Rs2E* instead of *Rs1E*.

if $((Rs1E == RdM)\ \&\ RegWriteM)\ \&\ (Rs1E\ != 0)$ then **// Forward from Memory stage**
$ForwardAE = 10$
else if $((Rs1E == RdW)\ \&\ RegWriteW)\ \&\ (Rs1E\ != 0)$ then **// Forward from Writeback stage**
$ForwardAE = 01$
else $ForwardAE = 00$ **// No forwarding (use RF output)**

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 7.56 shows this problem. The `lw` instruction receives data from memory at the end of cycle 4, but the `and` instruction needs that data (the value in `s7`) as a source operand at the beginning of cycle 4. There is no way to solve this hazard with forwarding.

A solution is to *stall* the pipeline, holding up operation until the data is available. Figure 7.57 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`. Also, in cycle 5, source `s7` of the `or` instruction is read directly from the register file, with no need for forwarding.

Note 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*, which behaves like a nop instruction. The bubble is introduced by zeroing out the Execute stage control signals during a Decode stage stall so that the bubble performs no action and changes no architectural state.

In summary, stalling a stage is performed by disabling its pipeline register (i.e., the register to the left of a stage) so that the stage's inputs 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 (flushed) to prevent bogus information from propagating forward. Stalls degrade performance, so they should be used only when necessary.

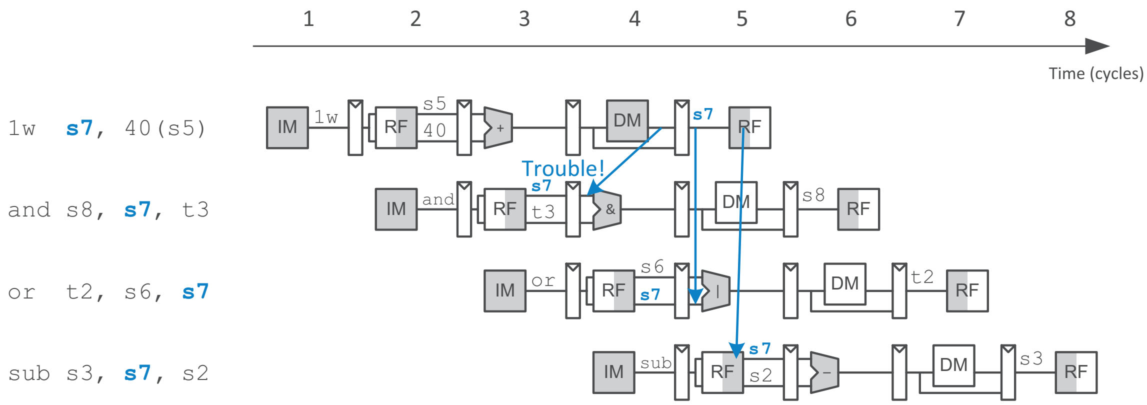


Figure 7.56 Abstract pipeline diagram illustrating trouble forwarding from `lw`

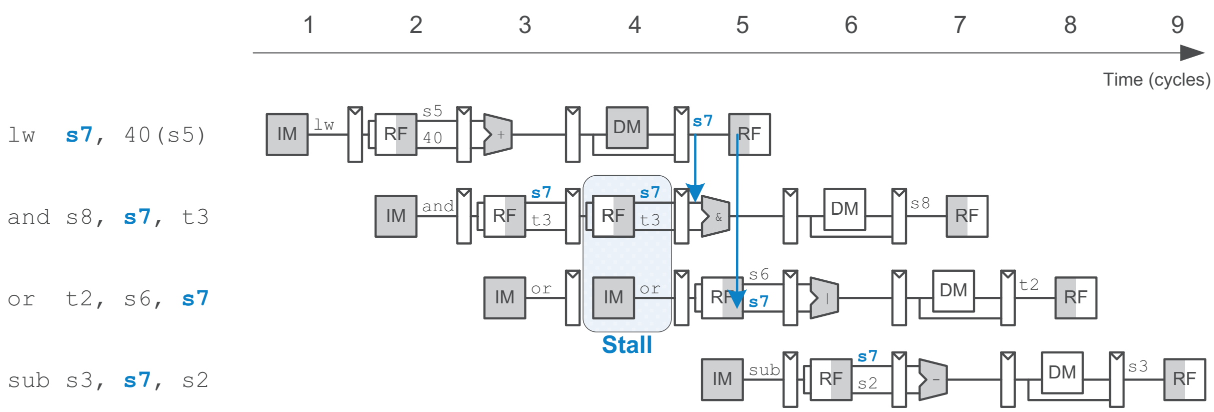


Figure 7.57 Abstract pipeline diagram illustrating stall to solve hazards

Figure 7.58 modifies the pipelined processor to add stalls for `lw` data dependencies. In order for the Hazard Unit to stall the pipeline, the following conditions must be met:

1. A load word is in the Execute stage (indicated by $ResultSrcE_0 = 1$) and
2. The load's destination register (*RdE*) matches *Rs1D* or *Rs2D*, the source operands of the instruction in the Decode stage

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 load word (`lw`) stall occurs, *StallD* and *StallF* are asserted to force the Decode and Fetch stage pipeline registers to retain their existing values. *FlushE* is also asserted to clear the contents of the Execute stage pipeline register, introducing a bubble. The Hazard Unit *lwStall* (load word stall) signal indicates when the pipeline

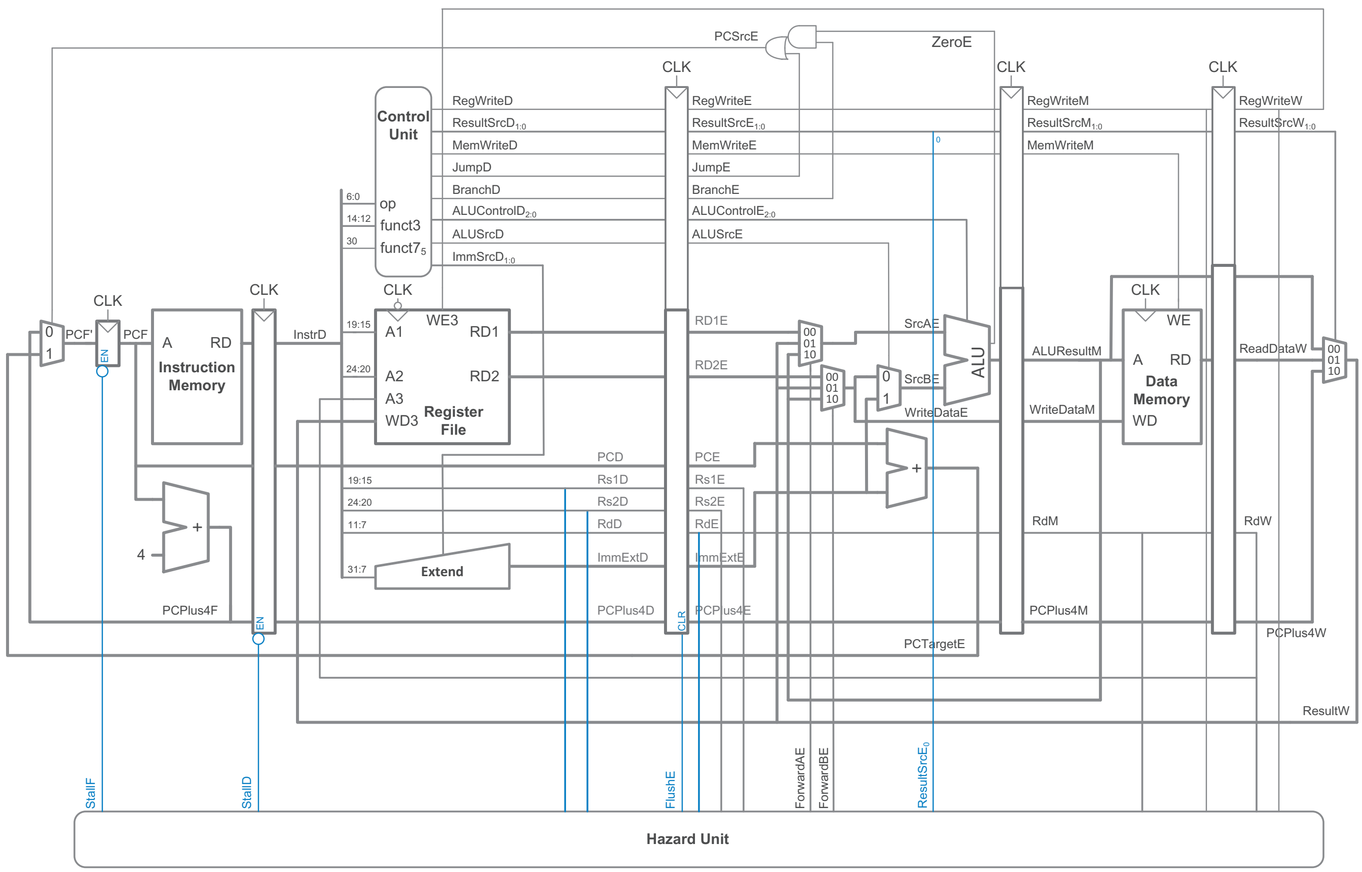


Figure 7.58 Pipelined processor with stalls to solve `lw` data hazard

> The *lwStall* logic described here could cause the processor to stall unnecessarily when the destination of the load is `x0` or when a false dependency exists—that is, when the instruction in the Decode stage is a J- or I-type instruction that randomly causes a false match between bits in their immediate fields and *RdE*. However, these cases are rare (and poor coding practice, in the case of `x0` being the load destination) and they cause only a small performance loss.

should be stalled due to a load word dependency. Whenever *lwStall* is TRUE, all of the stall and flush signals are asserted. Hence, the logic to compute the stalls and flushes is:

$$lwStall = ResultSrcE_0 \ \& \ ((Rs1D == RdE) \mid (Rs2D == RdE))$$
$$StallF = StallD = FlushE = lwStall$$

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 this control hazard is to stall the pipeline until the branch decision is made (i.e., *PCSrcE* is computed). Because the decision is made in the Execute stage, the pipeline would have to be stalled for two cycles at every branch. This would severely degrade the system performance if branches occur often, which is typically the case.

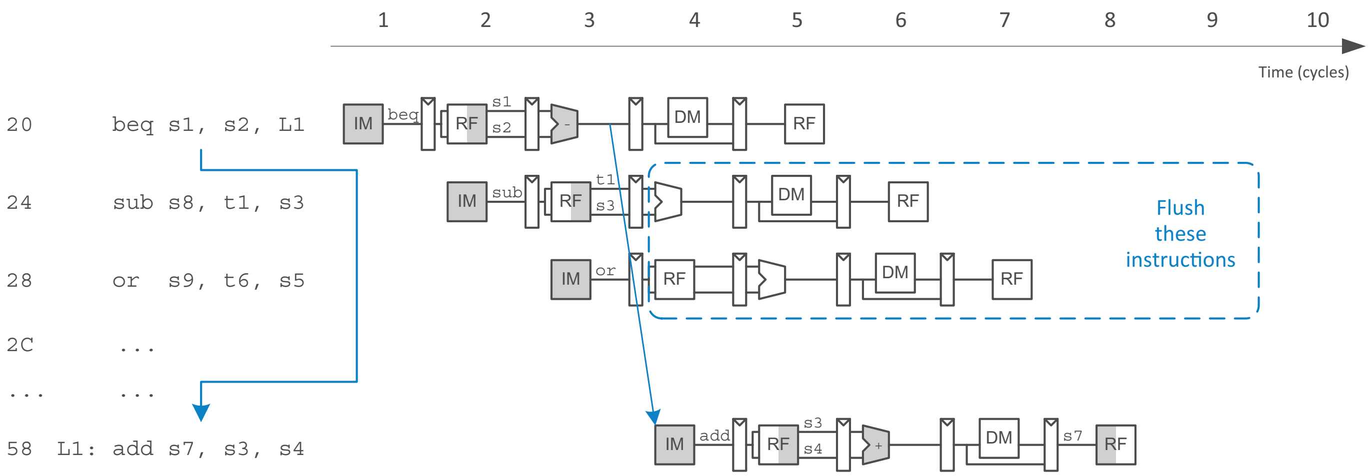


Figure 7.59 **Abstract pipeline diagram illustrating flushing when a branch is taken**

An alternative to stalling the pipeline 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 the pipeline presented so far (Figure 7.58), the processor predicts that branches are not taken and simply continues executing the program in order until *PCSrcE* is asserted to select the next PC from *PCTargetE* instead. If the branch should have been taken, then the two 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 7.59 shows such a scheme in which a branch from address 0x20 to address 0x58 is taken. The PC is not written until cycle 3, by which point the `sub` and `or` instructions at addresses 0x24 and 0x28 have already been fetched. These instructions must be flushed, and the `add` instruction is fetched from address 0x58 in cycle 4.

Finally, we must work out the stall and flush signals to handle branches and PC writes. When a branch is taken, the subsequent two instructions must be flushed from the pipeline registers of the Decode and Execute stages. Thus, we add a synchronous clear input (CLR) to the Decode pipeline register and add the *FlushD* output to the Hazard Unit. (When CLR = 1, the register contents are cleared, that is, become 0.) When a branch is taken (indicated by *PCSrcE* being 1), *FlushD* and *FlushE* must be asserted to flush the Decode and Execute pipeline registers. Figure 7.60 shows the enhanced pipelined processor for handling control hazards. The flushes are now calculated as:

$$FlushD = PCSrcE$$
$$FlushE = lwStall \mid PCSrcE$$

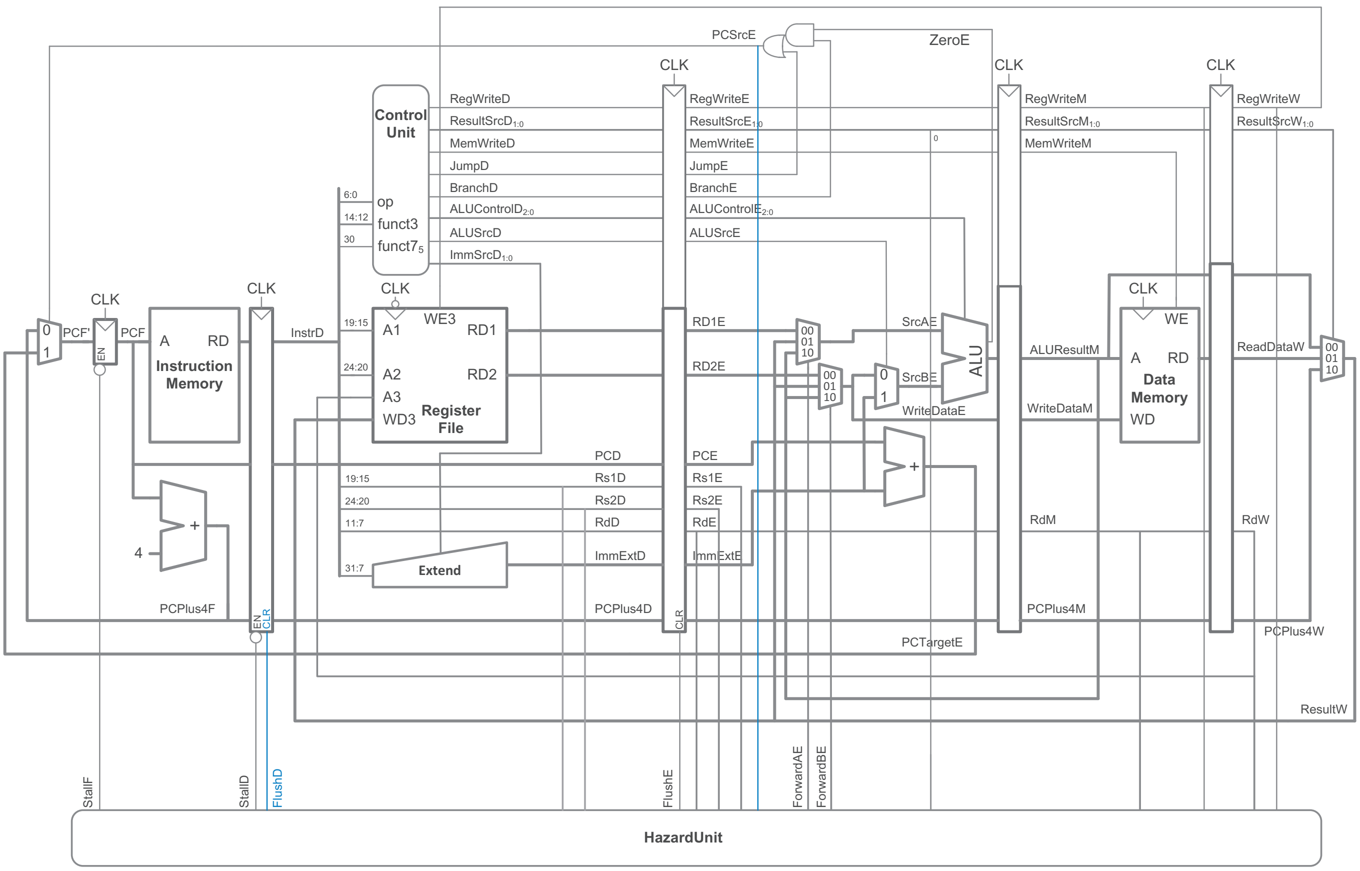


Figure 7.60 Expanded Hazard Unit for handling branch control hazard

Hazard Summary

In summary, RAW data hazards occur when an instruction depends on a result (from another instruction) that has not yet been written into the register file. 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 stalling the pipeline until the decision is made or by predicting which 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 possible interactions between instructions and to discover all of the hazards that may exist. Figure 7.61 shows the complete pipelined processor handling all of the hazards. The hazard logic is summarized on the next page.

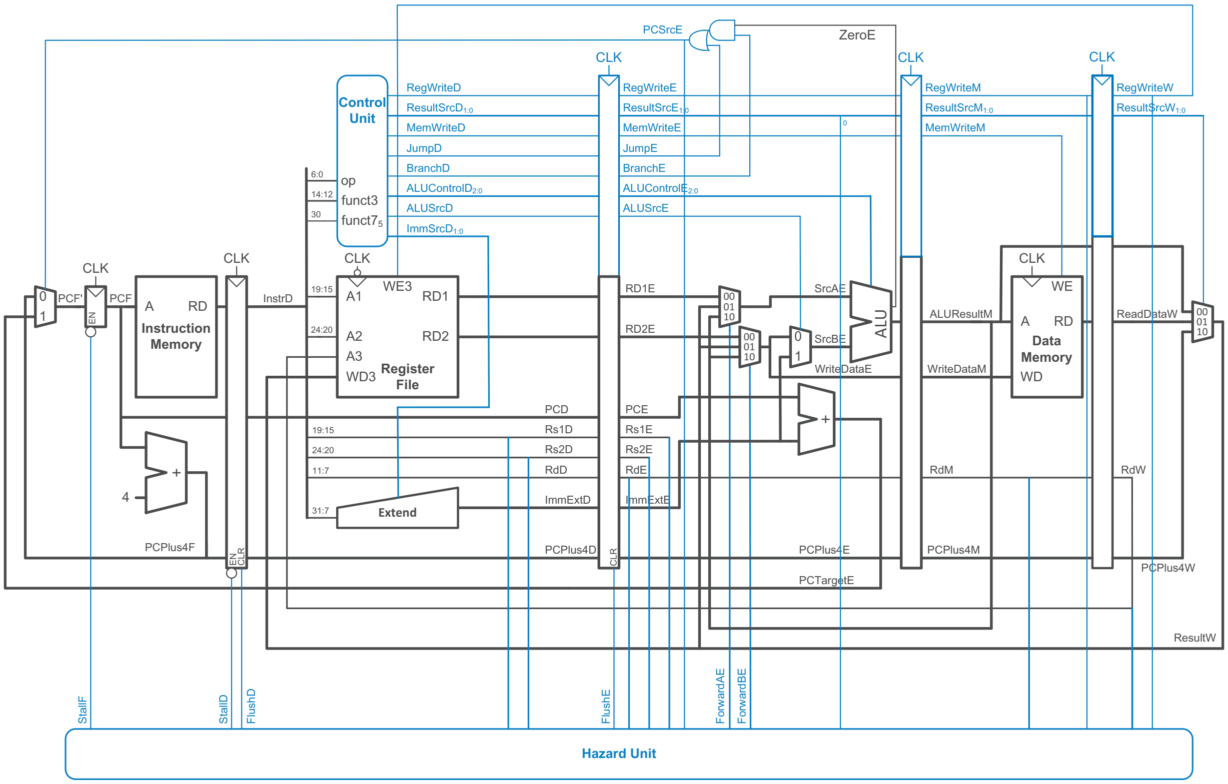


Figure 7.61 Pipelined processor with full hazard handling

Forward to solve data hazards when possible[3]:

if $((Rs1E == RdM)\ \&\ RegWriteM)\ \&\ (Rs1E\ != 0)$ then

$ForwardAE = 10$

else if $((Rs1E == RdW)\ \&\ RegWriteW)\ \&\ (Rs1E\ != 0)$ then

$ForwardAE = 01$

else $ForwardAE = 00$

Stall when a load hazard occurs:

$lwStall = ResultSrcE_0\ \&\ ((Rs1D == RdE)\ |\ (Rs2D == RdE))$

$StallF = lwStall$

$StallD = lwStall$

Flush when a branch is taken or a load introduces a bubble:

$FlushD = PCSrcE$

$FlushE = lwStall\ |\ PCSrcE$

[3] Recall that the forwarding logic for *SrcBE* (*ForwardBE*) is identical except that it checks *Rs2E* instead of *Rs1E*.

7.5.4 Performance Analysis

The pipelined processor ideally would have a CPI of 1 because a new instruction is *issued*—that is, fetched—every cycle. However, a stall or a flush wastes 1 to 2 cycles, 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 7.4 consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R- or I-type ALU instructions. Assume that 40% of the loads are immediately followed by an instruction that uses the result, requiring a stall, and that 50% of the branches are taken (mispredicted), requiring two instructions to be flushed. Ignore other hazards. Compute the average CPI of the pipelined processor.

Solution The average CPI is the weighted 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 three when they are not, so they have a CPI of (0.5)(1) + (0.5)(3) = 2. Jumps take three clock cycles (CPI = 3). All other instructions have a CPI of 1. Hence, for this benchmark, the average CPI = (0.25)(1.4) + (0.1)(1) + (0.11)(2) + (0.02)(3) + (0.52)(1) = 1.25.

We can determine the cycle time by considering the critical path in each of the five pipeline stages shown in Figure 7.61. Recall that the register file is used twice in a single cycle: it is written in the first half of the Writeback cycle and read in the second half of the Decode cycle; so these stages can use only half of the cycle time for their critical path. Another way of saying it is this: twice the critical path for each of those stages must fit in a cycle. Figure 7.62 shows the critical path for the Execute stage. It occurs when a branch is in the Execute stage that requires forwarding from the Writeback stage: the path goes from the Writeback pipeline register, through the Result, ForwardBE, and SrcB multiplexers, through the ALU and AND-OR logic to the PC multiplexer and, finally, to the PC register.

The critical path analysis for the Execute stage assumes that the Hazard Unit delay for calculating *ForwardAE* and *ForwardBE* is less than or equal to the delay of the Result multiplexer. If the Hazard Unit delay is longer, it must be included in the critical path instead of the Result multiplexer delay.

$$T_{c\_pipelined} = max\begin{bmatrix} t_{pcq} + t_{mem} + t_{setup} & \textit{Fetch} \\ 2(t_{RFread} + t_{setup}) & \textit{Decode} \\ t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND-OR} + t_{setup} & \textit{Execute} \\ t_{pcq} + t_{mem} + t_{setup} & \textit{Memory} \\ 2(t_{pcq} + t_{mux} + t_{RFsetup}) & \textit{Writeback} \end{bmatrix} \tag{7.5}$$

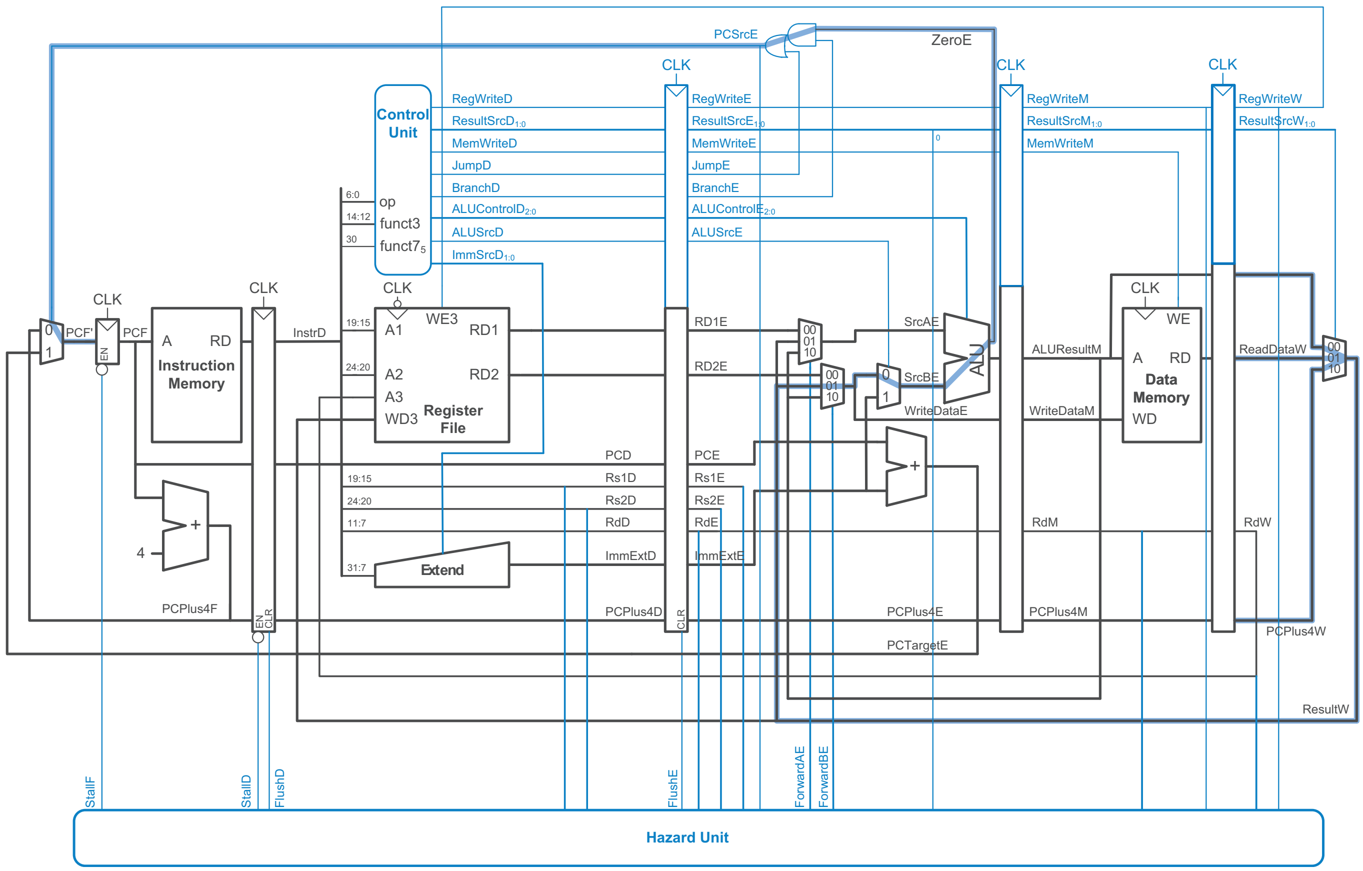


Figure 7.62 Pipelined processor critical path

Example 7.10 PIPELINED PROCESSOR PERFORMANCE COMPARISON

Ben Bitdiddle needs to compare the pipelined processor performance with that of the single-cycle and multicycle processors considered in Examples 7.4 and 7.8. The logic delays were given in Table 7.7 (on page 415). 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 $T_{c\_pipelined} = \max[40 + 200 + 50, 2(100 + 50), 40 + 4(30) + 120 + 20 + 50, 40 + 200 + 50, 2(40 + 30 + 60)] = 350\,\text{ps}$. The Execute stage takes the longest. According to Equation 7.1, the total execution time is $T_{pipelined} = (100 \times 10^9$ instructions) (1.25 cycles/instruction)(350×10^{-12} s/cycle) = 44 seconds. This compares with 75 seconds for the single-cycle processor and 155 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 fivefold speedup one might hope to get from a five-stage pipeline.

Our pipelined processor is unbalanced, with branch resolution in the Execute stage taking much longer than any other stage. The pipeline could be balanced better by pushing the Result multiplexer back into the Memory stage, reducing the cycle time to 320 ps.

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. Imbalanced delay in pipeline stages also decreases the benefits of pipelining. The pipelined processor is similar in hardware requirements to the single-cycle processor, but it adds many 32-bit pipeline registers, along with multiplexers, smaller pipeline registers, and control logic to resolve hazards.

7.6 HDL REPRESENTATION*

This section presents HDL code for the single-cycle RISC-V processor that supports the instructions discussed in this chapter. The code illustrates good coding practices for a moderately complex system. HDL code for the multicycle processor and pipelined processor are left to Exercises 7.25 to 7.27 and 7.42 to 7.44.

In this section, the instruction and data memories are separated from the datapath and connected by address and data busses. In practice, most processors pull instructions and data from separate caches. However, to handle smaller memory maps where data may be intermixed with instructions, a more complete processor must also be able to read data (in addition to instructions) from the instruction memory. Chapter 8 will revisit memory systems, including the interaction of caches with main memory.

Figure 7.63 shows a block diagram of the single-cycle RISC-V processor interfaced to external memories. The processor is composed of

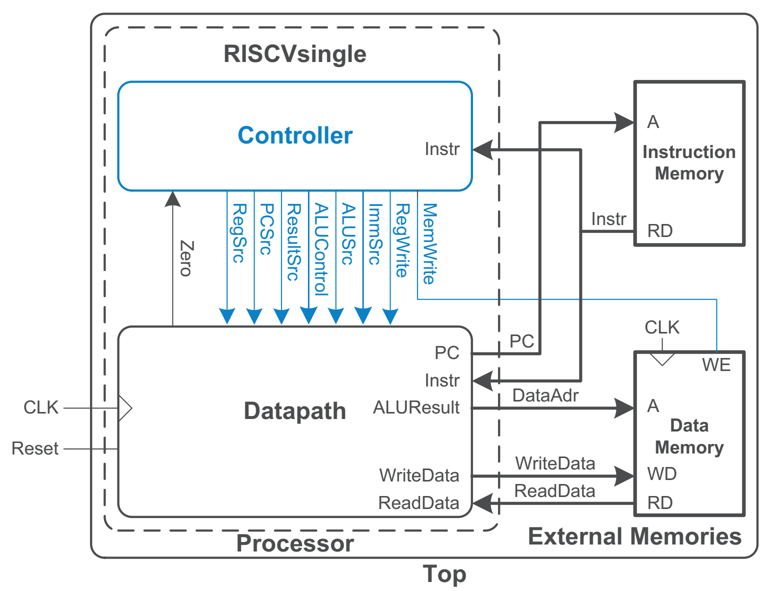


Figure 7.63 Single-cycle processor interfaced to external memories

the datapath from Figure 7.15 and the controller from Figure 7.16. The controller, in turn, is composed of the Main Decoder and the ALU Decoder.

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, which are used by any microarchitecture. Section 7.6.3 introduces the test program, testbench, and external memories. The HDL and test program are available in electronic form on this book's website (see the Preface).

7.6.1 Single-Cycle Processor

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

HDL Example 7.1 SINGLE-CYCLE PROCESSOR

SystemVerilog

```
module riscvsingle(input  logic        clk, reset,
                   output logic [31:0] PC,
                   input  logic [31:0] Instr,
                   output logic        MemWrite,
                   output logic [31:0] ALUResult, WriteData,
                   input  logic [31:0] ReadData);

  logic       ALUSrc, RegWrite, Jump, Zero;
  logic [1:0] ResultSrc, ImmSrc;
  logic [2:0] ALUControl;

  controller c(Instr[6:0], Instr[14:12], Instr[30], Zero,
               ResultSrc, MemWrite, PCSrc,
               ALUSrc, RegWrite, Jump,
               ImmSrc, ALUControl);
  datapath dp(clk, reset, ResultSrc, PCSrc,
              ALUSrc, RegWrite,
              ImmSrc, ALUControl,
              Zero, PC, Instr,
              ALUResult, WriteData, ReadData);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity riscvsingle is
  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;
       ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
       ReadData:             in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of riscvsingle is
  component controller
    port(op:             in  STD_LOGIC_VECTOR(6 downto 0);
         funct3:         in  STD_LOGIC_VECTOR(2 downto 0);
         funct7b5, Zero: in  STD_LOGIC;
         ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
         MemWrite:       out STD_LOGIC;
         PCSrc, ALUSrc:  out STD_LOGIC;
         RegWrite, Jump: out STD_LOGIC;
         ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
         ALUControl:     out STD_LOGIC_VECTOR(2 downto 0));
  end component;
  component datapath
    port(clk, reset: in STD_LOGIC;
         ResultSrc:            in  STD_LOGIC_VECTOR(1  downto 0);
         PCSrc, ALUSrc:        in  STD_LOGIC;
         RegWrite:             in  STD_LOGIC;
         ImmSrc:               in  STD_LOGIC_VECTOR(1  downto 0);
         ALUControl:           in  STD_LOGIC_VECTOR(2  downto 0);
         Zero:                 out STD_LOGIC;
         PC:                   out STD_LOGIC_VECTOR(31 downto 0);
         Instr:                in  STD_LOGIC_VECTOR(31 downto 0);
         ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
         ReadData:             in  STD_LOGIC_VECTOR(31 downto 0));
  end component;
```

```
  signal ALUSrc, RegWrite, Jump, Zero, PCSrc: STD_LOGIC;
  signal ResultSrc, ImmSrc: STD_LOGIC_VECTOR(1 downto 0);
  signal ALUControl: STD_LOGIC_VECTOR(2 downto 0);
begin
  c: controller port map(Instr(6 downto 0), Instr(14 downto 12),
                         Instr(30), Zero, ResultSrc, MemWrite,
                         PCSrc, ALUSrc, RegWrite, Jump,
                         ImmSrc, ALUControl);
  dp: datapath port map(clk, reset, ResultSrc, PCSrc, ALUSrc,
                         RegWrite, ImmSrc, ALUControl, Zero,
                         PC, Instr, ALUResult,WriteData,
                         ReadData);

end;
```

HDL Example 7.2 CONTROLLER

SystemVerilog

```
module controller(input  logic [6:0] op,
                  input  logic [2:0] funct3,
                  input  logic       funct7b5,
                  input  logic       Zero,
                  output logic [1:0] ResultSrc,
                  output logic       MemWrite,
                  output logic       PCSrc, ALUSrc,
                  output logic       RegWrite, Jump,
                  output logic [1:0] ImmSrc,
                  output logic [2:0] ALUControl);
  logic [1:0] ALUOp;
  logic       Branch;

  maindec md(op, ResultSrc, MemWrite, Branch,
             ALUSrc, RegWrite, Jump, ImmSrc, ALUOp);
  aludec  ad(op[5], funct3, funct7b5, ALUOp, ALUControl);

  assign PCSrc = Branch & Zero | Jump;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity controller is
  port(op:             in     STD_LOGIC_VECTOR(6 downto 0);
       funct3:         in     STD_LOGIC_VECTOR(2 downto 0);
       funct7b5, Zero: in     STD_LOGIC;
       ResultSrc:      out    STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:       out    STD_LOGIC;
       PCSrc, ALUSrc:  out    STD_LOGIC;
       RegWrite:       out    STD_LOGIC;
       Jump:           buffer STD_LOGIC;
       ImmSrc:         out    STD_LOGIC_VECTOR(1 downto 0);
       ALUControl:     out    STD_LOGIC_VECTOR(2 downto 0));
end;

architecture struct of controller is
  component maindec
    port(op:               in  STD_LOGIC_VECTOR(6 downto 0);
         ResultSrc:        out STD_LOGIC_VECTOR(1 downto 0);
         MemWrite:         out STD_LOGIC;
         Branch, ALUSrc:   out STD_LOGIC;
         RegWrite, Jump:   out STD_LOGIC;
         ImmSrc:           out STD_LOGIC_VECTOR(1 downto 0);
         ALUOp:            out STD_LOGIC_VECTOR(1 downto 0));
  end component;
  component aludec
    port(opb5:       in  STD_LOGIC;
         funct3:     in  STD_LOGIC_VECTOR(2 downto 0);
         funct7b5:   in  STD_LOGIC;
         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, ResultSrc, MemWrite, Branch,
                       ALUSrc, RegWrite, Jump, ImmSrc, ALUOp);
  ad: aludec port map(op(5), funct3, funct7b5, ALUOp, ALUControl);
  PCSrc <= (Branch and Zero) or Jump;
end;
```

HDL Example 7.3 MAIN DECODER

SystemVerilog

```
module maindec(input  logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic       MemWrite,
               output logic       Branch, ALUSrc,
               output logic       RegWrite, Jump,
               output logic [1:0] ImmSrc,
               output logic [1:0] ALUOp);
  logic [10:0] controls;

  assign {RegWrite, ImmSrc, ALUSrc, MemWrite,
          ResultSrc, Branch, ALUOp, Jump} = controls;

  always_comb
    case(op)
    // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump
      7'b0000011: controls = 11'b1_00_1_0_01_0_00_0; // lw
      7'b0100011: controls = 11'b0_01_1_1_00_0_00_0; // sw
      7'b0110011: controls = 11'b1_xx_0_0_00_0_10_0; // R-type
      7'b1100011: controls = 11'b0_10_0_0_00_1_01_0; // beq
      7'b0010011: controls = 11'b1_00_1_0_00_0_10_0; // I-type ALU
      7'b1101111: controls = 11'b1_11_0_0_10_0_00_1; // jal
      default:    controls = 11'bx_xx_x_x_xx_x_xx_x; // ???
    endcase
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity maindec is
  port(op:             in  STD_LOGIC_VECTOR(6 downto 0);
       ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:       out STD_LOGIC;
       Branch, ALUSrc: out STD_LOGIC;
       RegWrite, Jump: out STD_LOGIC;
       ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
       ALUOp:          out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture behave of maindec is
  signal controls: STD_LOGIC_VECTOR(10 downto 0);
begin
  process(op) begin
    case op is
      when "0000011" => controls <= "10010010000"; -- lw
      when "0100011" => controls <= "00111000000"; -- sw
      when "0110011" => controls <= "1--00000100"; -- R-type
      when "1100011" => controls <= "01000001010"; -- beq
      when "0010011" => controls <= "10010000100"; -- I-type ALU
      when "1101111" => controls <= "11100100001"; -- jal
      when others    => controls <= "-----------"; -- not valid
    end case;
  end process;

  (RegWrite, ImmSrc(1), ImmSrc(0), ALUSrc, MemWrite,
  ResultSrc(1), ResultSrc(0), Branch, ALUOp(1), ALUOp(0),
  Jump) <= controls;
end;
```

HDL Example 7.4 ALU DECODER

SystemVerilog

```
module aludec(input  logic       opb5,
              input  logic [2:0] funct3,
              input  logic       funct7b5,
              input  logic [1:0] ALUOp,
              output logic [2:0] ALUControl);

  logic  RtypeSub;
  assign RtypeSub = funct7b5 & opb5; // TRUE for R-type subtract

  always_comb
    case(ALUOp)
      2'b00:              ALUControl = 3'b000; // addition
      2'b01:              ALUControl = 3'b001; // subtraction
      default: case(funct3) // R-type or I-type ALU
                 3'b000:  if (RtypeSub)
                            ALUControl = 3'b001; // sub
                          else
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity aludec is
  port(opb5:       in  STD_LOGIC;
       funct3:     in  STD_LOGIC_VECTOR(2 downto 0);
       funct7b5:   in  STD_LOGIC;
       ALUOp:      in  STD_LOGIC_VECTOR(1 downto 0);
       ALUControl: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture behave of aludec is
  signal RtypeSub: STD_LOGIC;
begin
  RtypeSub <= funct7b5 and opb5; -- TRUE for R-type subtract
  process(opb5, funct3, funct7b5, ALUOp, RtypeSub) begin
    case ALUOp is
```

```
                          ALUControl = 3'b000; // add, addi
            3'b010:       ALUControl = 3'b101; // slt, slti
            3'b110:       ALUControl = 3'b011; // or, ori
            3'b111:       ALUControl = 3'b010; // and, andi
            default:      ALUControl = 3'bxxx; // ???
          endcase
      endcase
endmodule
```

```
      when "00" =>          ALUControl <= "000"; -- addition
      when "01" =>          ALUControl <= "001"; -- subtraction
      when others => case funct3 is     -- R-type or I-type ALU
          when "000" = if RtypeSub = '1' then
                            ALUControl <= "001"; -- sub
                          else
                            ALUControl <= "000"; -- add, addi
                          end if;
          when "010"  =>  ALUControl <= "101"; -- slt, slti
          when "110"  =>  ALUControl <= "011"; -- or, ori
          when "111"  =>  ALUControl <= "010"; -- and, andi
          when others =>  ALUControl <= "---"; -- unknown
        end case;
    end case;
  end process;
end;
```

HDL Example 7.5 DATAPATH

SystemVerilog

```
module datapath(input  logic        clk, reset,
                input  logic [1:0]  ResultSrc,
                input  logic        PCSrc, ALUSrc,
                input  logic        RegWrite,
                input  logic [1:0]  ImmSrc,
                input  logic [2:0]  ALUControl,
                output logic        Zero,
                output logic [31:0] PC,
                input  logic [31:0] Instr,
                output logic [31:0] ALUResult, WriteData,
                input  logic [31:0] ReadData);

  logic [31:0] PCNext, PCPlus4, PCTarget;
  logic [31:0] ImmExt;
  logic [31:0] SrcA, SrcB;
  logic [31:0] Result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, PCNext, PC);
  adder       pcadd4(PC, 32'd4, PCPlus4);
  adder       pcaddbranch(PC, ImmExt, PCTarget);
  mux2 #(32)  pcmux(PCPlus4, PCTarget, PCSrc, PCNext);

  // register file logic
  regfile     rf(clk, RegWrite, Instr[19:15], Instr[24:20],
                 Instr[11:7], Result, SrcA, WriteData);
  extend      ext(Instr[31:7], ImmSrc, ImmExt);

  // ALU logic
  mux2 #(32)  srcbmux(WriteData, ImmExt, ALUSrc, SrcB);
  alu         alu(SrcA, SrcB, ALUControl, ALUResult, Zero);
  mux3 #(32)  resultmux(ALUResult, ReadData, PCPlus4,
                        ResultSrc, Result);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity datapath is
  port(clk, reset:          in     STD_LOGIC;
       ResultSrc:           in     STD_LOGIC_VECTOR(1  downto 0);
       PCSrc, ALUSrc:       in     STD_LOGIC;
       RegWrite:            in     STD_LOGIC;
       ImmSrc:              in     STD_LOGIC_VECTOR(1  downto 0);
       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);
       ALUResult, WriteData: buffer STD_LOGIC_VECTOR(31 downto 0);
       ReadData:            in     STD_LOGIC_VECTOR (31 downto 0));
end;

architecture struct of datapath is
  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 adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:    out STD_LOGIC_VECTOR(31 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;
  component mux3 generic(width: integer);
    port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:          in  STD_LOGIC_VECTOR(1 downto 0);
         y:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component regfile
    port(clk:        in  STD_LOGIC;
         we3:        in  STD_LOGIC;
         a1, a2, a3: 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 extend
    port(instr:  in  STD_LOGIC_VECTOR(31 downto 7);
         immsrc: in  STD_LOGIC_VECTOR(1  downto 0);
         immext: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component alu
    port(a, b:       in     STD_LOGIC_VECTOR(31 downto 0);
         ALUControl: in     STD_LOGIC_VECTOR(2  downto 0);
         ALUResult:  buffer STD_LOGIC_VECTOR(31 downto 0);
         Zero:       out    STD_LOGIC);
  end component;

  signal PCNext, PCPlus4, PCTarget: STD_LOGIC_VECTOR(31 downto 0);
  signal ImmExt:                    STD_LOGIC_VECTOR(31 downto 0);
  signal SrcA, SrcB:                STD_LOGIC_VECTOR(31 downto 0);
  signal Result:                    STD_LOGIC_VECTOR(31 downto 0);
begin
  -- next PC logic
  pcreg: flopr generic map(32) port map(clk, reset, PCNext, PC);
  pcadd4: adder port map(PC, X"00000004", PCPlus4);
  pcaddbranch: adder port map(PC, ImmExt, PCTarget);
  pcmux: mux2 generic map(32) port map(PCPlus4, PCTarget, PCSrc,
                                       PCNext);
  -- register file logic
  rf: regfile port map(clk, RegWrite, Instr(19 downto 15),
                       Instr(24 downto 20), Instr(11 downto 7),
                       Result, SrcA, WriteData);
  ext: extend port map(Instr(31 downto 7), ImmSrc, ImmExt);
  -- ALU logic
  srcbmux: mux2 generic map(32) port map(WriteData, ImmExt,
                                         ALUSrc, SrcB);
  mainalu: alu port map(SrcA, SrcB, ALUControl, ALUResult, Zero);
  resultmux: mux3 generic map(32) port map(ALUResult, ReadData,
                                           PCPlus4, ResultSrc,
                                           Result);
end;
```

7.6.2 Generic Building Blocks

This section contains generic building blocks that may be useful in any digital system, including an adder, flip-flops, and a 2:1 multiplexer. The register file appeared in HDL Example 5.8. The HDL for the ALU is left to Exercises 5.11 through 5.14.

HDL Example 7.6 ADDER

SystemVerilog

```
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.NUMERIC_STD_UNSIGNED.all;

entity adder is
  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 7.7 EXTEND UNIT

SystemVerilog

```
module extend(input  logic [31:7] instr,
              input  logic [1:0]  immsrc,
              output logic [31:0] immext);

  always_comb
    case(immsrc)
               // I-type
      2'b00:   immext = {{20{instr[31]}}, instr[31:20]};
               // S-type (stores)
      2'b01:   immext = {{20{instr[31]}}, instr[31:25],
                          instr[11:7]};
               // B-type (branches)
      2'b10:   immext = {{20{instr[31]}}, instr[7],
                          instr[30:25], instr[11:8], 1'b0};
               // J-type (jal)
      2'b11:   immext = {{12{instr[31]}}, instr[19:12],
                          instr[20], instr[30:21], 1'b0};
      default: immext = 32'bx; // undefined
    endcase
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity extend is
  port(instr:  in  STD_LOGIC_VECTOR(31 downto 7);
       immsrc: in  STD_LOGIC_VECTOR(1  downto 0);
       immext: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of extend is
begin
  process(instr, immsrc) begin
    case immsrc is
      -- I-type
      when "00" =>
        immext <= (31 downto 12 => instr(31)) & instr(31 downto 20);
      -- S-types (stores)
      when "01" =>
        immext <= (31 downto 12 => instr(31)) &
                  instr(31 downto 25) & instr(11 downto 7);
      -- B-type (branches)
      when "10" =>
        immext <= (31 downto 12 => instr(31)) & instr(7) & instr(30
                  downto 25) & instr(11 downto 8) & '0';
      -- J-type (jal)
      when "11" =>
        immext <= (31 downto 20 => instr(31)) &
                  instr(19 downto 12) & instr(20) &
                  instr(30 downto 21) & '0';
      when others =>
        immext <= (31 downto 0  => '-');
    end case;
  end process;
end;
```

HDL Example 7.8 RESETTABLE FLIP-FLOP

SystemVerilog

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

  always_ff @(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
  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 <= (others => '0');
    elsif rising_edge(clk) then q <= d;
    end if;
  end process;
end;
```

HDL Example 7.9 RESETTABLE FLIP-FLOP WITH ENABLE

SystemVerilog

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

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

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity flopenr is
  generic(width: integer);
  port(clk, reset, en: 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 flopenr is
begin
  process(clk, reset, en) begin
    if reset = '1' then                        q <= (others => '0');
    elsif rising_edge(clk) and en = '1' then q <= d;
    end if;
  end process;
end;
```

HDL Example 7.10 2:1 MULTIPLEXER

SystemVerilog

```
module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  generic(width: integer := 8);
  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;

architecture behave of mux2 is
begin
  y <= d1 when s = '1' else d0;
end;
```

HDL Example 7.11 3:1 MULTIPLEXER

SystemVerilog

```
module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux3 is
  generic(width: integer := 8);
  port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:          in  STD_LOGIC_VECTOR(1 downto 0);
       y:          out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux3 is
begin
  process(d0, d1, d2, s) begin
    if    (s = "00") then y <= d0;
    elsif (s = "01") then y <= d1;
    elsif (s = "10") then y <= d2;
    end if;
  end process;
end;
```

```
# riscvtest.s
# Sarah.Harris@unlv.edu
# David_Harris@hmc.edu
# 27 Oct 2020
#
# Test the RISC-V processor:
#   add, sub, and, or, slt, addi, lw, sw, beq, jal
# If successful, it should write the value 25 to address 100
#       RISC-V Assembly         Description               Address   Machine Code
main:   addi x2, x0, 5          # x2 = 5                  0         00500113
        addi x3, x0, 12         # x3 = 12                 4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3       8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7       C         0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4     10        0041F2B3
        add  x5, x5, x4         # x5 = 4 + 7 = 11         14        004282B3
        beq  x5, x7, end        # shouldn't be taken      18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0       1C        0041A233
        beq  x4, x0, around     # should be taken         20        00020463
        addi x5, x0, 0          # shouldn't execute       24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5) = 1        28        0023A233
        add  x7, x4, x5         # x7 = (1 + 11) = 12      2C        005203B3
        sub  x7, x7, x2         # x7 = (12 - 5) = 7       30        402383B3
        sw   x7, 84(x3)         # [96] = 7                34        0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 7           38        06002103
        add  x9, x2, x5         # x9 = (7 + 11) = 18      3C        005104B3
        jal  x3, end            # jump to end, x3 = 0x44  40        008001EF
        addi x2, x0, 1          # shouldn't execute       44        00100113
end:    add  x2, x2, x9         # x2 = (7 + 18) = 25      48        00910133
        sw   x2, 0x20(x3)       # [100] = 25              4C        0221A023
done:   beq  x2, x2, done       # infinite loop           50        00210063
```

Figure 7.64 riscvtest.s

```
00500113
00C00193
FF718393
0023E233
0041F2B3
004282B3
02728863
0041A233
00020463
00000293
0023A233
005203B3
402383B3
0471AA23
06002103
005104B3
008001EF
00100113
00910133
0221A023
00210063
```

Figure 7.65 riscvtest.txt

7.6.3 Testbench

The testbench loads a program into the memories. The program in Figure 7.64 exercises all of the instructions by performing a computation that should produce the correct result only if all of the instructions are functioning correctly. Specifically, the program will write the value 25 to address 100 if it runs correctly, but it is unlikely to do so if the hardware is buggy. This is an example of *ad hoc* testing.

The machine code is stored in a text file called riscvtest.txt (Figure 7.65) which is loaded by the testbench during simulation. The file consists of the machine code for the instructions written in hexadecimal, one instruction per line.

The testbench, top-level RISC-V module (that instantiates the RISC-V processor and memories), and external memory HDL code are given in the following examples. The testbench instantiates the top-level module being tested and generates a periodic clock and a reset at the start of the simulation. It checks for memory writes and reports success if the correct value (25) is written to address 100. The memories in this example hold 64 32-bit words each.

HDL Example 7.12 TESTBENCH

SystemVerilog

```
module testbench();

  logic        clk;
  logic        reset;
  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
  end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
  end

  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr === 100 & WriteData === 25) begin
          $display("Simulation succeeded");
          $stop;
        end else if (DataAdr !== 96) begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity testbench is
end;

architecture test of testbench is
  component top
    port(clk, reset:         in  STD_LOGIC;
         WriteData, DataAdr: out STD_LOGIC_VECTOR(31 downto 0);
         MemWrite:           out STD_LOGIC);
  end component;

  signal WriteData, DataAdr:   STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset, MemWrite: STD_LOGIC;
begin
  -- instantiate device to be tested
  dut: top port map(clk, reset, WriteData, DataAdr, MemWrite);

  -- Generate clock with 10 ns period
  process begin
     clk <= '1';
     wait for 5 ns;
     clk <= '0';
     wait for 5 ns;
  end process;

  -- Generate reset for first two clock cycles
  process begin
    reset <= '1';
    wait for 22 ns;
    reset <= '0';
    wait;
  end process;

  -- check that 25 gets written to address 100 at end of program
  process(clk) begin
    if(clk'event and clk = '0' and MemWrite = '1') then
      if(to_integer(DataAdr) = 100 and
         to_integer(writedata) = 25) then
        report "NO ERRORS: Simulation succeeded" severity
        failure;
      elsif (DataAdr /= 96) then
        report "Simulation failed" severity failure;
      end if;
    end if;
  end process;
end;
```

HDL Example 7.13 TOP-LEVEL MODULE

SystemVerilog

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);

  logic [31:0] PC, Instr, ReadData;

  // instantiate processor and memories
  riscvsingle rvsingle(clk, reset, PC, Instr, MemWrite,
                       DataAdr, WriteData, ReadData);
  imem imem(PC, Instr);
  dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity top is
  port(clk, reset:        in     STD_LOGIC;
       WriteData, DataAdr: buffer STD_LOGIC_VECTOR(31 downto 0);
       MemWrite:          buffer STD_LOGIC);
end;

architecture test of top is
  component riscvsingle
    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;
      ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
      ReadData:            in  STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component imem
  port(a:  in  STD_LOGIC_VECTOR(31 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component dmem
  port(clk, we: in  STD_LOGIC;
    a, wd:      in  STD_LOGIC_VECTOR(31 downto 0);
    rd:         out STD_LOGIC_VECTOR(31 downto 0));
  end component;

  signal PC, Instr, ReadData: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- instantiate processor and memories
  rvsingle: riscvsingle port map(clk, reset, PC, Instr,
                                 MemWrite, DataAdr,
                                 WriteData, ReadData);
  imem1: imem port map(PC, Instr);
  dmem1: dmem port map(clk, MemWrite, DataAdr, WriteData,
                       ReadData);
end;
```

HDL Example 7.14 INSTRUCTION MEMORY

SystemVerilog

```
module imem(input  logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
    $readmemh("riscvtest.txt",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
use ieee.std_logic_textio.all;

entity imem is
  port(a:  in  STD_LOGIC_VECTOR(31 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of imem is
type ramtype is array(63 downto 0) of
                      STD_LOGIC_VECTOR(31 downto 0);
```

```
  -- initialize memory from file
  impure function init_ram_hex return ramtype is
  file text_file : text open read_mode is "riscvtest.txt";
    variable text_line : line;
    variable ram_content : ramtype;
    variable i : integer := 0;
  begin
    for i in 0 to 63 loop -- set all contents low
      ram_content(i) := (others => '0');
    end loop;
    while not endfile(text_file) loop -- set contents from file
      readline(text_file, text_line);
      hread(text_line, ram_content(i));
      i := i + 1;
    end loop;

    return ram_content;
  end function;

  signal mem : ramtype := init_ram_hex;
  begin
  -- read memory
  process(a) begin
    rd <= mem(to_integer(a(31 downto 2)));
  end process;
end;
```

HDL Example 7.15 DATA MEMORY

SystemVerilog

```
module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity dmem is
  port(clk, we: in  STD_LOGIC;
       a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
       rd:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of dmem is
begin
  process is
    type ramtype is array (63 downto 0) of
                   STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- read or write memory
    loop
      if rising_edge(clk) then
        if (we = '1') then mem(to_integer(a(7 downto 2))) := wd;
        end if;
      end if;
      rd <= mem(to_integer(a(7 downto 2)));
      wait on clk, a;
    end loop;
  end process;
end;
```

7.7 ADVANCED MICROARCHITECTURE*

High-performance microprocessors use a wide variety of techniques to run programs faster. Recall that the time required to run a program is proportional to the period of the clock and to the number of clock cycles per instruction (CPI). Thus, to increase performance, we would like to speed up the clock and/or reduce the CPI. This section surveys some existing speedup techniques. The implementation details become quite complex, so we focus on the concepts. Hennessy and Patterson's *Computer Architecture* text is a definitive reference if you want to fully understand the details.

Advances in integrated circuit manufacturing have steadily reduced transistor sizes. Smaller transistors are faster and generally consume less power. Thus, even if the microarchitecture does not change, the clock frequency can increase because all the gates are faster. Moreover, smaller transistors enable placing more transistors on a chip. Microarchitects use the additional transistors to build more complicated processors or to put more processors on a chip. Unfortunately, power consumption increases with the number of transistors and the speed at which they operate (see Section 1.8). Power consumption has become an essential concern. Microprocessor designers have a challenging task juggling the trade-offs among speed, power, and cost for chips with billions of transistors in some of the most complex systems that humans have ever built.

7.7.1 Deep Pipelines

In the late 1990's and early 2000's, microprocessors were marketed largely based on clock frequency ($f = 1/T_c$). This pushed microprocessors to use very deep pipelines (20–31 stages on the Pentium 4) to maximize the clock frequency, even if the benefits to overall performance were questionable. Power is proportional to clock frequency and increases with the number of pipeline registers, so now that power consumption is so important, pipeline depths are shorter.

Aside from advances in manufacturing, the easiest way to speed up the clock is to chop the pipeline into more stages. Each stage contains less logic, so it can run faster. This chapter has considered a classic five-stage pipeline, but 8 to 20 stages are now commonly used. For example, the SweRV EH1 core, the open-source commercial RISC-V processor developed by Western Digital, has nine pipeline stages.

The maximum number of pipeline stages is limited by pipeline hazards, sequencing overhead, and cost. Longer pipelines introduce more dependencies. Some of the dependencies can be solved by forwarding but others require stalls, which increase the CPI. The pipeline registers between each stage have sequencing overhead from their setup time and clk-to-Q delay (as well as clock skew). Due to this sequencing overhead, adding more pipeline stages gives diminishing returns. Finally, adding more stages increases the cost because of the extra pipeline registers and hardware required to handle hazards.

Example 7.11 DEEP PIPELINES

Consider building a pipelined processor by chopping up the single-cycle processor into N stages. The single-cycle processor has a propagation delay of 750 ps through the combinational logic. The sequencing overhead of a register is 90 ps. Assume that the combinational delay can be arbitrarily divided into any number of stages and that pipeline hazard logic does not increase the delay. The five-stage pipeline in Example 7.9 has a CPI of 1.25. Assume that each additional stage increases the CPI by 0.1 because of branch mispredictions and other pipeline hazards. How many pipeline stages should be used to make the processor execute programs as fast as possible?

Solution The cycle time for an N-stage pipeline is $T_c = [(750/N) + 90]$ ps. The CPI is $1.25 + 0.1(N - 5)$, where $N \geq 5$. The time per instruction (i.e., instruction time) is the product of the cycle time T_c and the CPI. Figure 7.66 plots the cycle time and instruction time versus the number of stages. The instruction time has a minimum of 281 ps at $N = 8$ stages. This minimum is only slightly better than the 295 ps per instruction achieved with a five-stage pipeline, and the curve is almost flat between 7 to 10 stages.

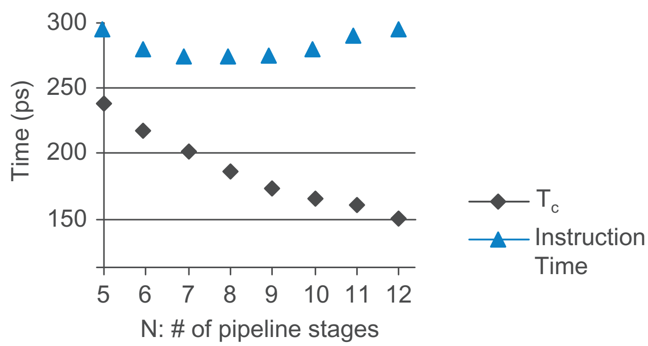


Figure 7.66 Cycle time and instruction time vs. the number of pipeline stages

7.7.2 Micro-Operations

Recall our design principles of "regularity supports simplicity" and "make the common case fast." Pure reduced instruction set computer (RISC) architectures such as RISC-V contain only simple instructions, typically those that can be executed in a single cycle on a simple, fast datapath with a three-ported register file, single ALU, and single data memory access like the ones we have developed in this chapter. Complex instruction set computer (CISC) architectures generally include instructions requiring more registers, more additions, or more than one memory access per instruction. For example, the x86 instruction `ADD [ESP],[EDX+80+EDI*2]` involves reading the three registers (`ESP`, `EDX`, and `EDI`), adding the base (`EDX`), displacement (80), and scaled index (EDI*2), reading two memory locations, summing their values, and writing the result back to memory. A microprocessor that could perform all of these functions at once would be unnecessarily slow when executing more common, simpler instructions.

Microarchitects make the decision of whether to provide hardware to implement a complex operation directly or to break it into micro-op sequences. They make similar decisions about other options described later in this section. These choices lead to different points in the performance-power-cost design space.

Computer architects of CISC processors make the common case fast by defining a set of simple *micro-operations* (also known as *micro-ops* or μops) that can be executed on simple datapaths. Each CISC instruction is decoded into one or more micro-ops. For example, if we defined μops resembling RISC-V instructions, and used temporary registers t1 and t2 to hold intermediate results, then the x86 instruction above could become six μops:

```
slli t2, EDI, 1   # t2 = EDI*2
add  t1, EDX, t2  # t1 = EDX + EDI*2
lw   t1, 80(t1)   # t1 = MEM[EDX + EDI*2 + 80]
lw   t2, 0(ESP)   # t2 = MEM[ESP]
add  t1, t2, t1   # t1 = MEM[ESP] + MEM[EDX + EDI*2 + 80]
sw   t1, 0(ESP)   # MEM[ESP] = MEM[ESP] + MEM[EDX + EDI*2 + 80]
```

7.7.3 Branch Prediction

An ideal pipelined processor would have a CPI of 1. The branch misprediction penalty is a major reason for increased CPI. As pipelines get deeper, branches are resolved later in the pipeline. Thus, the branch misprediction penalty gets larger because all the instructions issued after the mispredicted branch must be flushed. To address this problem, most pipelined processors use a *branch predictor* to guess whether the branch should be taken. Recall that our pipeline from Section 7.5.3 simply predicted that branches are never taken.

Some branches occur at the beginning of a loop to check a condition and branch past the loop when the condition is no longer met (e.g., in *for* and *while* loops). Loops tend to execute many times, so these forward branches are usually not taken. Other branches occur when a program reaches the end of a loop and branches back to repeat the loop (e.g., in a *do/while* loop). Again, because loops tend to execute many times, these backward branches are usually taken. The simplest form of branch prediction checks the direction of the branch and predicts that backward branches are taken and forward branches are not. This is called *static branch prediction*, because it does not depend on the history of the program.

However, branches, especially forward branches, are difficult to predict without knowing more about the specific program. Therefore, most processors use *dynamic branch predictors*, which use the history of program execution to guess whether a branch should be taken. Dynamic branch predictors maintain a table of the last several hundred (or thousand) branch instructions that the processor has executed. The table, called a *branch target buffer*, includes the destination of the branch and a history of whether the branch was taken.

To see the operation of dynamic branch predictors, consider the following loop from Code Example 6.20. The loop repeats 10 times, and the branch out of the loop (`bge s0, t0, done`) is taken only on the last iteration.

```
      addi s1, zero, 0   # s1 = sum = 0
      addi s0, zero, 0   # s0 = i = 0
      addi t0, zero, 10  # t0 = 10
for:
      bge  s0, t0, done  # i >= 10?
      add  s1, s1, s0    # sum = sum + i
      addi s0, s0, 1     # i = i + 1
      j    for           # repeat loop
done:
```

A *one-bit dynamic branch predictor* remembers whether the branch was taken the last time and predicts that it will do the same thing the next time. While the loop is repeating, it remembers that the `beq` was not taken last time and predicts that it should not be taken next time. This is a correct prediction until the last branch of the loop, when the branch does get taken. Unfortunately, if the loop is run again, the branch predictor remembers that the last branch was taken. Therefore, it incorrectly predicts that the branch should be taken when the loop is first run again. In summary, a 1-bit branch predictor mispredicts the first and last branches of a loop.

A *two-bit dynamic branch predictor* solves this problem by having four states: Strongly Taken, Weakly Taken, Weakly Not Taken, and Strongly Not Taken, as shown in Figure 7.67. When the loop is repeating, it enters the Strongly Not Taken state and predicts that the branch should not be taken next time. This is correct until the last branch of the loop, which is taken and moves the predictor to the Weakly Not Taken state. When the loop is first run again, the branch predictor correctly predicts that the branch should not be taken and reenters the Strongly Not Taken state. In summary, a two-bit branch predictor mispredicts only the last branch of a loop. It is called a two-bit branch predictor because it requires two bits to encode the four states.

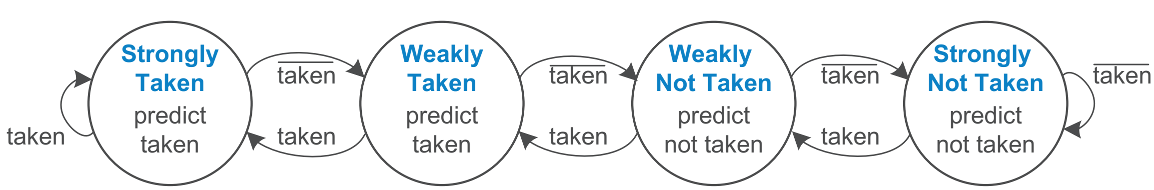


Figure 7.67 Two-bit branch predictor state transition diagram

Western Digital's RISC-V SweRV EH1 is a two-way superscalar core.

A *scalar* processor acts on one piece of data at a time. A *vector* processor acts on several pieces of data with a single instruction. A *superscalar* processor issues several instructions at a time, each of which operates on one piece of data.

Scalar processors are classified as single-instruction single-data (SISD) machines. Vector processors and graphics processors (GPUs: graphics processing units) are single-instruction multiple-data (SIMD) machines. Multiprocessors, such as multicore processors, are classified as multiple-instruction multiple-data (MIMD) machines. Typically, MIMD machines have a single program running that uses all or a subset of the cores. This style of programming is called single-program multiple-data (SPMD), but multiprocessors can be programmed in other ways as well.

The branch predictor operates in the Fetch stage of the pipeline so that it can determine which instruction to execute on the next cycle. When it predicts that the branch should be taken, the processor fetches the next instruction from the branch destination stored in the branch target buffer.

As one can imagine, branch predictors may be used to track even more history of the program to increase the accuracy of predictions. Good branch predictors achieve better than 90% accuracy on typical programs.

7.7.4 Superscalar Processors

A *superscalar processor* contains multiple copies of the datapath hardware to execute multiple instructions simultaneously. Figure 7.68 shows a block diagram of a two-way superscalar processor that fetches and executes two instructions per cycle. The datapath fetches two instructions at a time from the instruction memory. It has a six-ported register file to read four source operands and write two results back in each cycle. It also contains two ALUs and a two-ported data memory to execute the two instructions at the same time.

Figure 7.69 is a pipeline diagram illustrating the two-way superscalar processor executing two instructions on each cycle. For this program, the processor has a CPI of 0.5. Designers commonly refer to the reciprocal of the CPI as the *instructions per cycle*, or IPC. This processor has an IPC of 2 on this program.

Executing many instructions simultaneously is difficult because of dependencies. For example, Figure 7.70 shows a pipeline diagram running a program with data dependencies. The dependencies in the code are indicated in blue. The `add` instruction is dependent on `s8`, which is

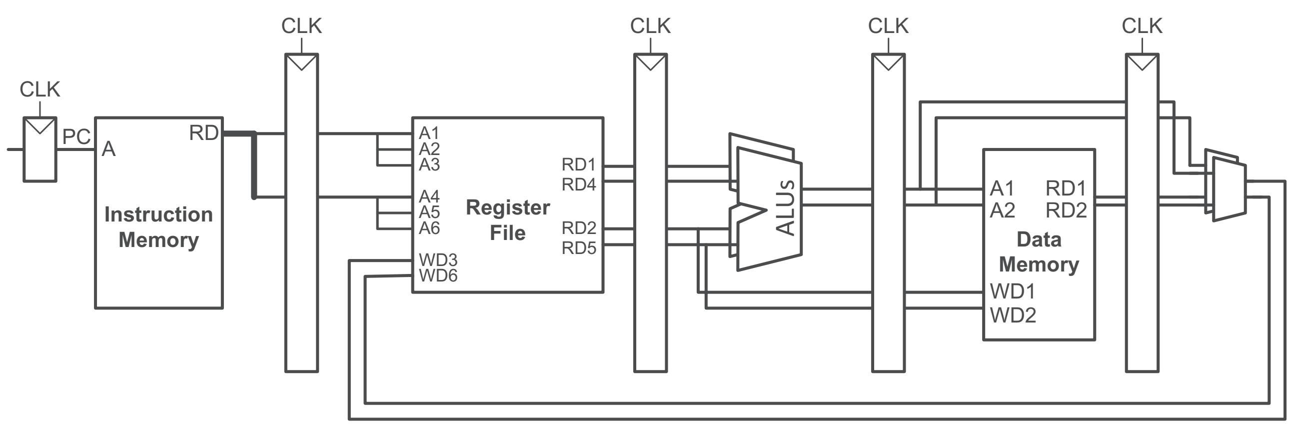


Figure 7.68 Superscalar datapath

produced by the `lw` instruction, so it cannot be issued at the same time as `lw`. The `add` instruction stalls for yet another cycle so that `lw` can forward `s8` to `add` in cycle 5. The other dependencies (between `sub` and `and` based on `s8`, and between `or` and `sw` based on `s11`) are handled by forwarding results produced in one cycle to be consumed in the next. This program requires five cycles to issue six instructions, for an IPC of 6/5 = 1.2.

Recall that parallelism comes in temporal and spatial forms. Pipelining is a case of temporal parallelism. Using multiple execution units is a case of spatial parallelism. Superscalar processors exploit both forms of parallelism to squeeze out performance far exceeding that of our single-cycle and multicycle processors.

Commercial processors may be three-, four-, or even six-way superscalar. They must handle control hazards such as branches as well as data hazards. Unfortunately, real programs have many dependencies, so wide superscalar processors rarely fully utilize all of the execution units. Moreover, the large number of execution units and complex forwarding networks consume vast amounts of circuitry and power.

Our RISC-V pipelined processor is a scalar processor. Vector processors were popular for supercomputers in the 1980's and 1990's because they efficiently handled the long vectors of data common in scientific computations, and they are heavily used now in GPUs. Processing vectors, or SIMD in general, is an example of *data-level parallelism*, where multiple data can be operated on in parallel. Modern high-performance microprocessors are superscalar, because issuing several independent instructions is more flexible than processing vectors. However, modern processors also include hardware, called SIMD units, to handle short vectors of data that are common in multimedia and graphics applications. RISC-V includes the vector (V) extension to support vector operations.

7.7.5 Out-of-Order Processor

To cope with the problem of dependencies, an out-of-order processor looks ahead across many instructions to issue independent instructions as rapidly as possible. The instructions can issue in a different order than that written by the programmer as long as dependencies are honored so that the program produces the intended result.

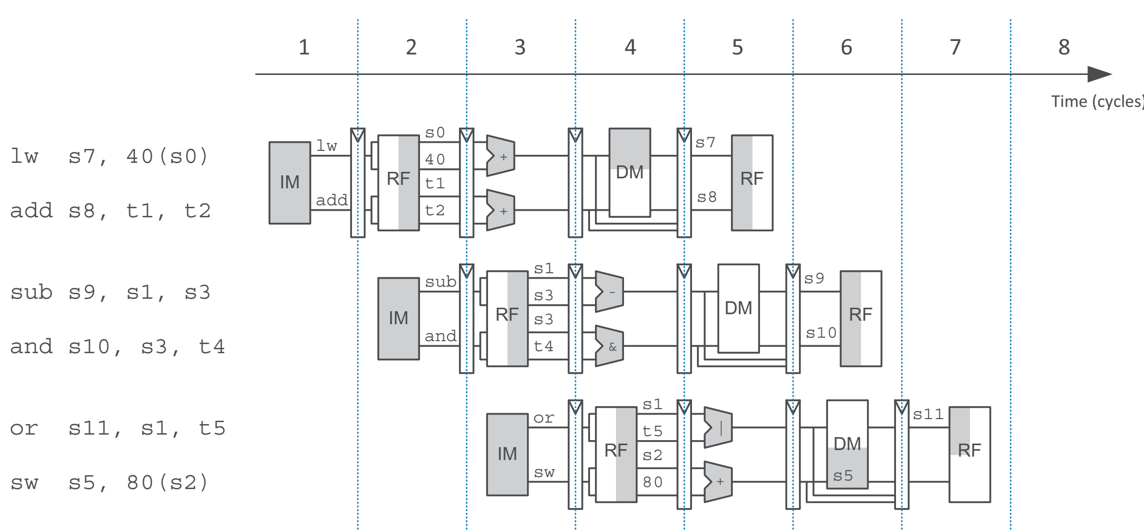


Figure 7.69 Abstract view of a superscalar pipeline in operation

Consider running the same program from Figure 7.70 on a two-way superscalar out-of-order processor. The processor can issue up to two instructions per cycle from anywhere in the program, as long as dependencies are observed. Figure 7.71 shows the data dependencies and the

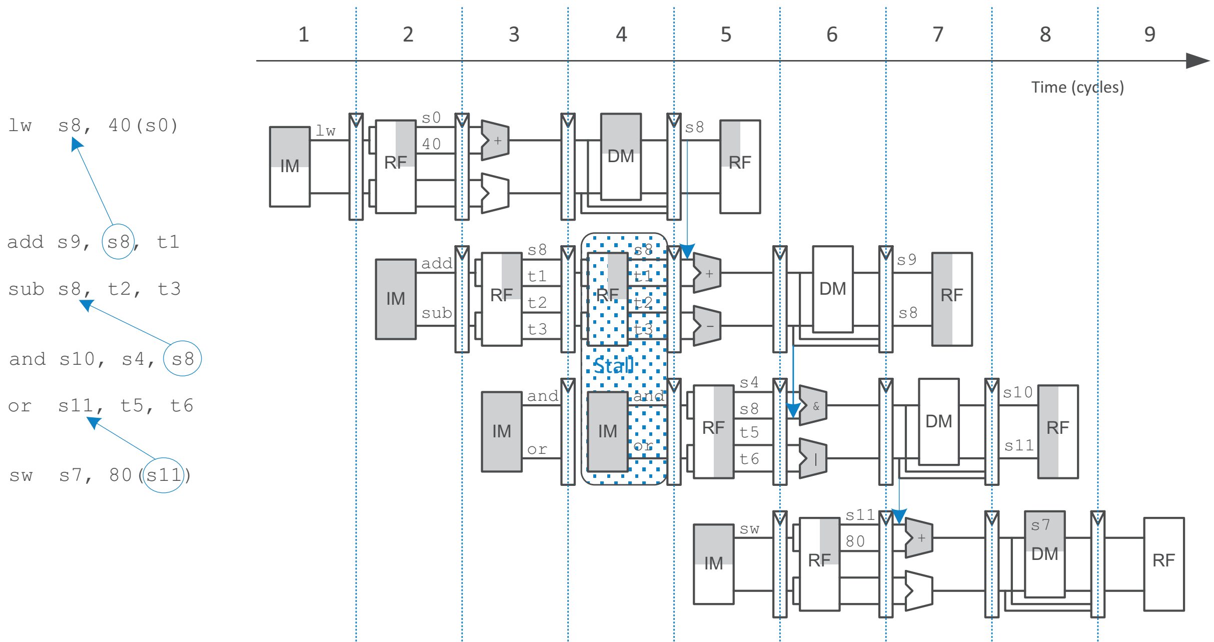


Figure 7.70 Program with data dependencies

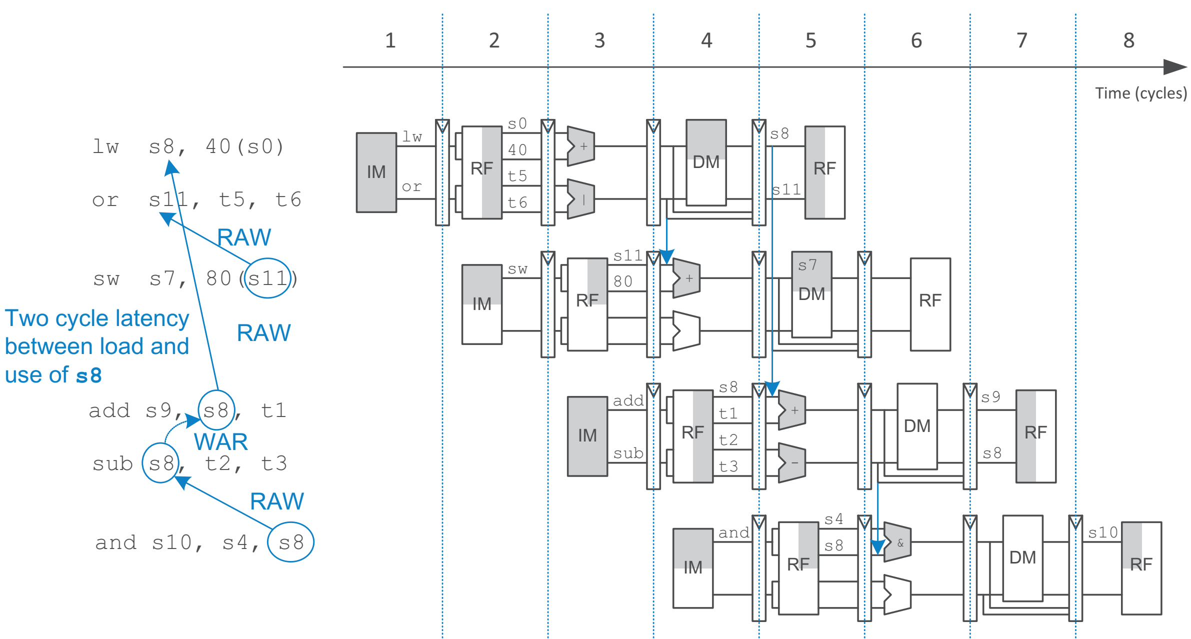


Figure 7.71 Out-of-order execution of a program with dependencies

operation of the processor. The classifications of dependencies as RAW and WAR will be discussed soon. The constraints on issuing instructions are:

Cycle 1

- The `lw` instruction issues.
- The `add`, `sub`, and `and` instructions are dependent on `lw` by way of `s8`, so they cannot issue yet. However, the `or` instruction is independent, so it also issues.

Cycle 2

- Remember that a two-cycle latency exists between issuing `lw` and a dependent instruction, so `add` cannot issue yet because of the `s8` dependence. `sub` writes `s8`, so it cannot issue before `add`, lest `add` receive the wrong value of `s8`. `and` is dependent on `sub`.
- Only the `sw` instruction issues.

Cycle 3

- On cycle 3, `s8` is available (or, rather, will be when `add` needs it), so the `add` issues. `sub` issues simultaneously, because it will not write `s8` until after `add` consumes (i.e., reads) it.

Cycle 4

- The `and` instruction issues. `s8` is forwarded from `sub` to `and`.

The out-of-order processor issues the six instructions in four cycles, for an IPC of 6/4 = 1.5. The dependence of `add` on `lw` by way of `s8` is a *read after write* (*RAW*) hazard. `add` must not read `s8` until after `lw` has written it. This is the type of dependency we are accustomed to handling in the pipelined processor. It inherently limits the speed at which the program can run, even if infinitely many execution units are available. Similarly, the dependence of `sw` on `or` by way of `s11` and of `and` on `sub` by way of `s8` are RAW dependencies.

The dependence between `sub` and `add` by way of `s8` is called a *write after read* (*WAR*) hazard or an *antidependence*. `sub` must not write `s8` before `add` reads `s8`, so that `add` receives the correct value according to the original order of the program. WAR hazards could not occur in the simple pipeline, but they may happen in an out-of-order processor if the dependent instruction (in this case, `sub`) is moved too early.

A WAR hazard is not essential to the operation of the program. It is merely an artifact of the programmer's choice to use the same register for two unrelated instructions. If the `sub` instruction had written `s3` instead of `s8`, then the dependency would disappear and `sub` could be issued before `add`. The RISC-V architecture has only 31 registers, so sometimes the programmer is forced to reuse a register and introduce a hazard just because all of the other registers are in use.

A third type of hazard, not shown in the program, is called a *write after write* (*WAW*) hazard or an *output dependence*. A WAW hazard occurs if an instruction attempts to write a register after a subsequent instruction has already written it. The hazard would result in the wrong value being written to the register. For example, in the following code, `lw` and `add` both write `s7`. The final value in `s7` should come from `add` according to the order of the program. If an out-of-order processor attempted to execute `add` first and then `lw`, a WAW hazard would occur.

```
lw  s7, 0(t3)
add s7, s1, t2
```

WAW hazards are not essential either; again, they are artifacts caused by the programmer using the same destination register for two unrelated instructions. If the `add` instruction were issued first, then the program could eliminate the WAW hazard by discarding the result of the `lw` instead of writing it to `s7`. This is called *squashing* the `lw`.[4]

Out-of-order processors use a table to keep track of instructions waiting to issue. The table, sometimes called a *scoreboard*, contains information about the dependencies. The size of the table determines how many instructions can be considered for issue. On each cycle, the processor examines the table and issues as many instructions as it can, limited by the dependencies and by the number of execution units (e.g., ALUs, memory ports) that are available.

The *instruction-level parallelism* (ILP) is the number of instructions that can be executed simultaneously for a particular program and microarchitecture. Theoretical studies have shown that the ILP can be quite large for out-of-order microarchitectures with perfect branch predictors and enormous numbers of execution units. However, practical processors seldom achieve an ILP greater than two or three, even with six-way superscalar datapaths with out-of-order execution.

7.7.6 Register Renaming

Out-of-order processors use a technique called *register renaming* to eliminate WAR and WAW hazards. Register renaming adds some nonarchitectural renaming registers to the processor. For example, a processor might add 20 renaming registers, called `r0` to `r19`. The programmer cannot

[4] You might wonder why the `lw` needs to be issued at all. The reason is that out-of-order processors must guarantee that all of the same exceptions occur that would have occurred if the program had been executed in its original order. The `lw` potentially may produce a load address misaligned exception or load access fault, so it must be issued to check for the exception, even though the result can be discarded.

use these registers directly, because they are not part of the architecture. However, the processor is free to use them to eliminate hazards.

For example, in the previous section, a WAR hazard occurred between the `sub s8, t2, t3` and `add s9, s8, t1` instructions based on reusing `s8`. The out-of-order processor could rename `s8` to `r0` for the `sub` instruction. Then, `sub` could be executed sooner, because `r0` has no dependency on the `add` instruction. The processor keeps a table of which registers were renamed so that it can consistently rename registers in subsequent dependent instructions. In this example, `s8` must also be renamed to `r0` in the `and` instruction, because it refers to the result of `sub`. Figure 7.72 shows the same program from Figure 7.71 executing on an out-of-order processor with register renaming. `s8` is renamed to `r0` in `sub` and `and` to eliminate the WAR hazard. The constraints on issuing instructions are:

Cycle 1

- The `lw` instruction issues.
- The `add` instruction is dependent on `lw` by way of `s8`, so it cannot issue yet. However, the `sub` instruction is independent now that its destination has been renamed to `r0`, so `sub` also issues.

Cycle 2

- Remember that a two-cycle latency must exist between issuing `lw` and a dependent instruction, so `add` cannot issue yet because of the `s8` dependence.

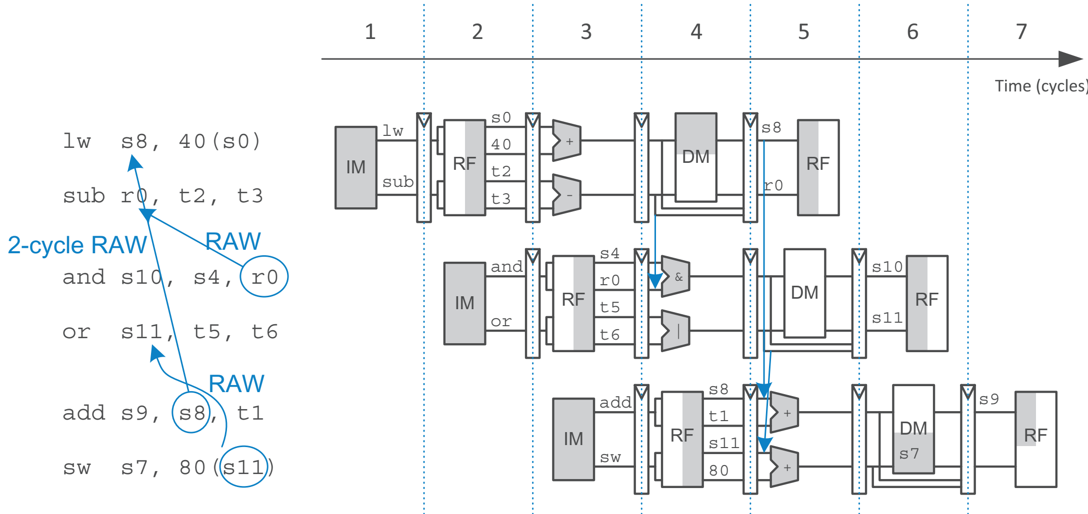


Figure 7.72 Out-of-order execution of a program using register renaming

- The `and` instruction is dependent on `sub`, so it can issue. `r0` is forwarded from `sub` to `and`.
- The `or` instruction is independent, so it also issues.

Cycle 3

- On cycle 3, `s8` is available, so the `add` issues.
- `s11` is also available, so `sw` issues.

The out-of-order processor with register renaming issues the six instructions in three cycles, for an IPC of 2.

7.7.7 Multithreading

Because the instruction-level parallelism (ILP) of real programs tends to be fairly low, adding more execution units to a superscalar or out-of-order processor gives diminishing returns. Another problem, discussed in Chapter 8, is that memory is much slower than the processor. Most loads and stores access a smaller and faster memory, called a *cache*. However, when the instructions or data are not available in the cache, the processor may stall for 100 or more cycles while retrieving the information from the main memory. Multithreading is a technique that helps keep a processor with many execution units busy even if the ILP of a program is low or the program is stalled waiting for memory.

To explain multithreading, we need to define a few new terms. A program running on a computer is called a *process*. Computers can run multiple processes simultaneously; for example, you can play music on a PC while surfing the web and running a virus checker. Each process consists of one or more *threads* that also run simultaneously. For example, a word processor may have one thread handling the user typing, a second thread spell-checking the document while the user works, and a third thread printing the document. In this way, the user does not have to wait, for example, for a document to finish printing before being able to type again. The degree to which a process can be split into multiple threads that can run simultaneously defines its level of *thread-level parallelism* (TLP).

In a conventional processor, the threads only give the illusion of running simultaneously. The threads actually take turns being executed on the processor under control of the operating system (OS). When one thread's turn ends, the OS saves its architectural state, loads the architectural state of the next thread, and starts executing that next thread. This procedure is called *context switching*. As long as the processor switches through all threads fast enough, the user perceives all of the threads as running at the same time. RISC-V dedicates one of its 32 registers, the thread pointer register, `tp` (i.e., `x4`), to point to (hold the address of) a thread's local memory.

A *hardware multithreaded* processor contains more than one copy of its architectural state so that more than one thread can be active at a time. For example, if we extended a processor to have four program counters and 128 registers, four threads could be available at one time. If one thread stalls while waiting for data from main memory, then the processor could context switch to another thread without any delay, because the program counter and registers are already available. Moreover, if one thread lacks sufficient parallelism to keep all execution units busy in a superscalar design, then another thread could issue instructions to the idle units. Switching between threads can either be fine-grained or coarse-grained. *Fine-grained* multithreading switches between threads on each instruction and must be supported by hardware multithreading. *Coarse-grained* multithreading switches out a thread only on expensive stalls, such as long memory accesses due to cache misses.

Multithreading does not improve the performance of an individual thread, because it does not increase the ILP. However, it does improve the overall throughput of the processor, because multiple threads can use processor resources that would have been idle when executing a single thread. Multithreading is also relatively inexpensive to implement because it replicates only the PC and register file, not the execution units and memories.

7.7.8 Multiprocessors

With contributions from Matthew Watkins

Modern processors have enormous numbers of transistors available. Using them to increase the pipeline depth or to add more execution units to a superscalar processor gives little performance benefit and wastes power. Around the year 2005, computer architects made a major shift to building multiple copies of the processor on the same chip; these copies are called *cores*.

A *multiprocessor* system consists of multiple processors and a method for communication between the processors. Three common classes of multiprocessors include *symmetric* (or *homogeneous*) multiprocessors, *heterogeneous* multiprocessors, and *clusters*.

Symmetric Multiprocessors

Symmetric multiprocessors include two or more identical processors sharing a single main memory. The multiple processors may be separate chips or multiple cores on the same chip.

Multiprocessors can be used to run more threads simultaneously or to run a particular thread faster. Running more threads simultaneously is easy; the threads are simply divided up among the processors. Unfortunately, typical PC users need to run only a small number of threads at any given time. Running a particular thread faster is much more challenging. The programmer must divide the existing thread into multiple threads to execute on each processor. This becomes tricky when the processors need to communicate with each other. One of the major challenges for computer designers and programmers is to effectively use large numbers of processor cores.

Symmetric multiprocessors have a number of advantages. They are relatively simple to design because the processor can be designed once and then replicated multiple times to increase performance. Programming for and executing code on a symmetric multiprocessor is also relatively straightforward because any program can run on any processor in the system and achieve approximately the same performance.

Heterogeneous Multiprocessors

Unfortunately, continuing to add more and more symmetric cores is not guaranteed to provide continued performance improvement. Most consumer applications use only a few threads at any given time, and consumers typically have only a couple of applications actually computing simultaneously. Although this is enough to keep dual-core and quad-core systems busy, unless programs start incorporating significantly more parallelism, continuing to add more cores beyond this point will provide diminishing benefits. As an added issue, because general-purpose processors are designed to provide good average performance, they are generally not the most power-efficient option for performing a given operation. This energy inefficiency is especially important in highly power-constrained systems, such as mobile phones.

Heterogeneous multiprocessors aim to address these issues by incorporating different types of cores and/or specialized hardware in a single system. Each application uses those resources that provide the best performance, or power-performance ratio, for that application. Because transistors are fairly plentiful these days, the fact that not every application will make use of every piece of hardware is of lesser concern.

Heterogeneous systems can take a number of forms. A heterogeneous system can incorporate cores with the same architecture but different microarchitectures, each with different power, performance, and area trade-offs. The RISC-V architecture was specifically designed with the aim of supporting a range of processor implementations, from low-cost embedded processors to high-performance multiprocessors. Another heterogeneous strategy is accelerators, in which a system contains special-purpose hardware optimized for performance or energy efficiency on specific types of tasks. For example, a mobile system-on-chip (SoC) presently may contain dedicated accelerators for graphics processing, video, wireless communication, real-time tasks, and cryptography. These accelerators can be 10 to 100 times more efficient (in performance, cost, and area) than general-purpose processors for the same tasks. Digital signal processors are another class of accelerators. These processors have a specialized instruction set optimized for math-intensive tasks.

Heterogeneous systems are not without their drawbacks. They add complexity in terms of both designing the different heterogeneous elements and the additional programming effort to decide when and how to make use of the varying resources. Symmetric and heterogeneous systems both have their places in modern systems. Symmetric multiprocessors are good for situations like large data centers that have lots of thread-level parallelism available. Heterogeneous systems are good for systems that have more varying or special-purpose workloads, such as mobile devices.

Clusters

In contrast to the other multiprocessors, processors in *clustered* multiprocessor systems each have their own local memory system instead of sharing memory. One type of cluster is a group of personal computers connected on a network and running software to jointly solve a large problem. The computers communicate using message passing instead of via shared memory. A large-scale computer cluster that has become increasingly important is the *data center*, also called *warehouse-scale computers* (WSCs), in which racks of computers and disks are networked and share power and cooling. Such systems typically include 50,000 to 100,000 computers, or *servers*, and cost $150 million.[5] Major Internet companies—including Google, Amazon, and Facebook—have driven the rapid development of data centers to support millions of users around the world. One major advantage of such clusters is that single computers can be swapped out as needed due to failures or upgrades.

In recent years, traditional servers owned by various companies are being replaced by *cloud computing*, where a smaller company rents a part of a WSC from such companies as Google and Amazon. Similarly, instead of an application running completely on a handheld device, such as a smartphone or tablet, generally called a *personal mobile device* (*PMD*), part of the application may run on the cloud to speed up computation and make data storage more efficient. This is called *software as a service* (*SaaS*). A common example of SaaS is a web search, where the database is stored on a WSC. Companies that rent cloud or web services demand both privacy (protection from other software running on the cloud) and performance. Both are realized using a virtual machine, which emulates an entire computer, including its operating system, running on a physical machine that may itself be running a

[5] D. Patterson and J. Hennessy, *Computer Organization and Design, The Hardware-Software Interface: RISC-V Edition*, Morgan Kaufmann, © 2018.

different operating system. Several virtual machines may run on one physical machine at once, with resources such as memory and I/O either partitioned or shared in time. This allows providers such as Amazon Web Services (AWS) to efficiently use physical resources, provide protection between virtual machines, and migrate virtual machines off of nonworking or low-performing computers. The *hypervisor*, also called the *virtual machine monitor* (VMM), is the software that runs the virtual machine and that maps virtual resources to physical resources. The hypervisor performs the functions typically performed by the operating system, such as managing I/O, CPU resources, and memory. The hypervisor runs between the *host* (the underlying physical hardware platform) and the operating system it is emulating. Instruction set architectures that allow the hypervisor to run directly on the hardware (as opposed to in software) are called *virtualizable*. This allows for more efficient, higher-performance virtual machines. Examples of virtualizable architectures include x86 (as of 2005), RISC-V, and IBM 370. ARMv7 and MIPS architectures are not virtualizable, but ARM did introduce virtualization extensions in 2013 with the introduction of ARMv8.

Cloud computing is also a critical part of Internet of Things (IoT) applications, where devices such as speakers, phones, and sensors connect through a network such as Bluetooth or Wi-Fi. Example IoT applications include connecting headphones to a smartphone using Bluetooth or connecting Alexa or Siri using a Wi-Fi connection. The low-cost devices (headphones, Google Home for Google Assistant, or Echo Dot for Alexa) connect through a network to higher-power servers that can stream music or, in the case of Siri and Alexa, perform speech recognition, query databases, and perform computations.

7.8 REAL-WORLD PERSPECTIVE: EVOLUTION OF RISC-V MICROARCHITECTURE*

Although RISC-V is an open-source architecture, not microarchitecture, many open-source hardware implementations are emerging, including Western Digital's SweRV cores, the SweRVolf SoC, and the PULP (Parallel Ultra Low Power) Platform. The ever-increasing RISC-V hardware implementations and supporting tools are referred to as the RISC-V ecosystem. See the Preface for information on how to access and use some of these open-source tools and hardware.

This section traces the development of RISC-V microarchitectures since RISC-V's inception in 2010. Because the base instruction set was fully described only recently, in 2017, many RISC-V chips are in development but only few are on the market as of 2021. But that is expected to change quickly as supporting tools and development cycles mature.

Most existing processor implementations are in low-level or embedded processors, but high-performance chips are on the horizon. RISC-V International (riscv.org) provides an ever-growing list of cores and SoC platforms. RISC-V commercial cores are found in SiFive's HiFive development board, Western Digital hard drives, and NVIDIA GPUs, amongst others.

As of 2021, two notable commercial RISC-V processors are SiFive's Freedom E310 core and Western Digital's open-source SweRV core, which comes in three versions. The Freedom E310 is a low-cost embedded processor used in SiFive's HiFive and Sparkfun's RED-V development boards. It runs RV32IMAC (RV32I with multiply/divide [M], atomic memory accesses [A], and compressed instructions [C] extensions) and has 8 KB of program memory, 8 KB of mask ROM for boot code, 16 KB of data SRAM, and a 16-KB two-way set-associative instruction cache. It also includes JTAG, SPI, I2C, and UART interfaces as well as a QSPI flash memory interface. The processor runs at 320 MHz and is a single-issue, in-order core with a 5-stage pipeline that has the same stages described in this chapter. Figure 7.73 shows a block diagram of the FE310-G002 processor found on the HiFive 1 Rev B board.

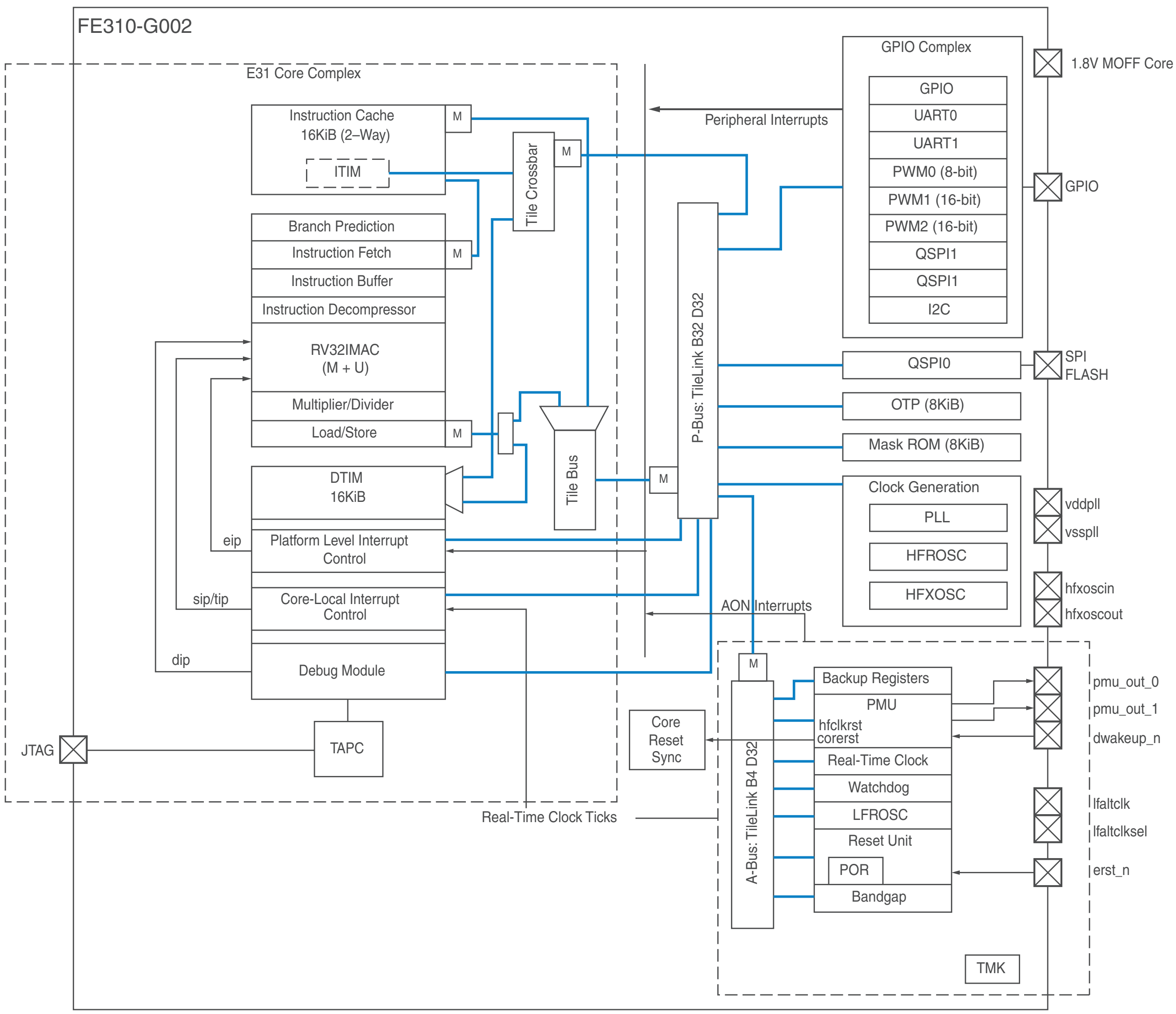


Figure 7.73 Freedom E310-G002 Block Diagram (Courtesy of SiFive Inc., *SiFive FE310-G002 Preliminary Datasheet v1p0*, ©2019)

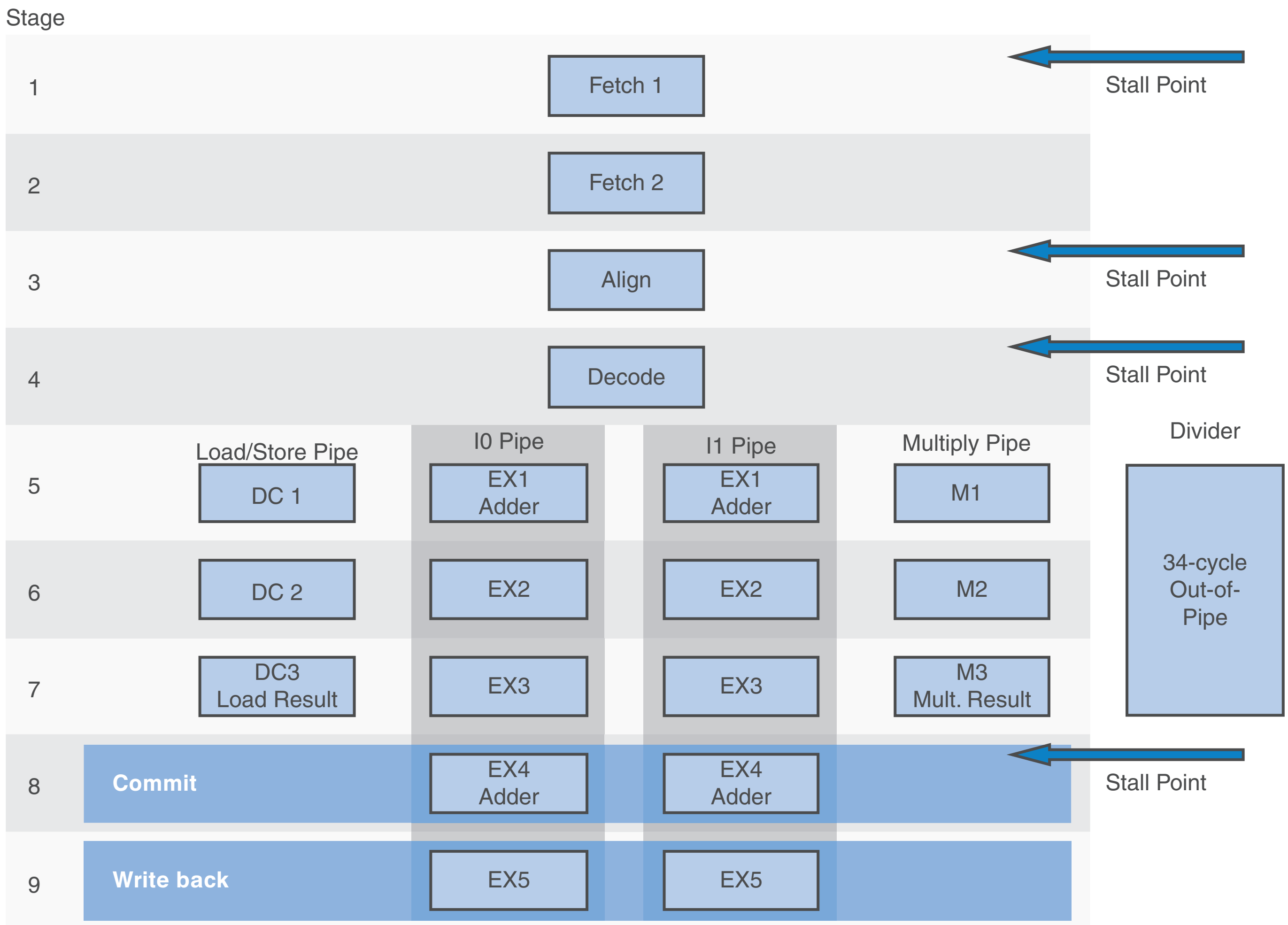


Figure 7.74 **SweRV EH1 9-stage pipeline**

(Courtesy of Western Digital Corporation, *RISC-V SweRV*^{TM} *EH1 Programmer's Reference Manual*, ©2020)

The source code (SystemVerilog) for Western Digital's open-source cores is available for download at https://github.com/chipsalliance/Cores-SweRV

The Western Digital SweRV core comes in three open-source versions: EH1, EH2, and EL2. The EH1 is a 32-bit, two-way superscalar core with a nine-stage pipeline and some support for out-of-order execution. These cores implement the RV32IMC instruction set, which includes the 32-bit base instruction set and compressed (C) and multiply/divide (M) extensions. It has a target frequency of 1 GHz using a 28-nm chip manufacturing process. The HDL can also be synthesized onto an FPGA. The EH2 core adds dual threading to the EH1. The EL2 core is a lower-performance processor targeted to embedded systems. Figure 7.74 shows the nine EH1 pipeline stages, which start with two fetch, one align, and one decode stage. The decode stage decodes up to two instructions. After this, the pipeline

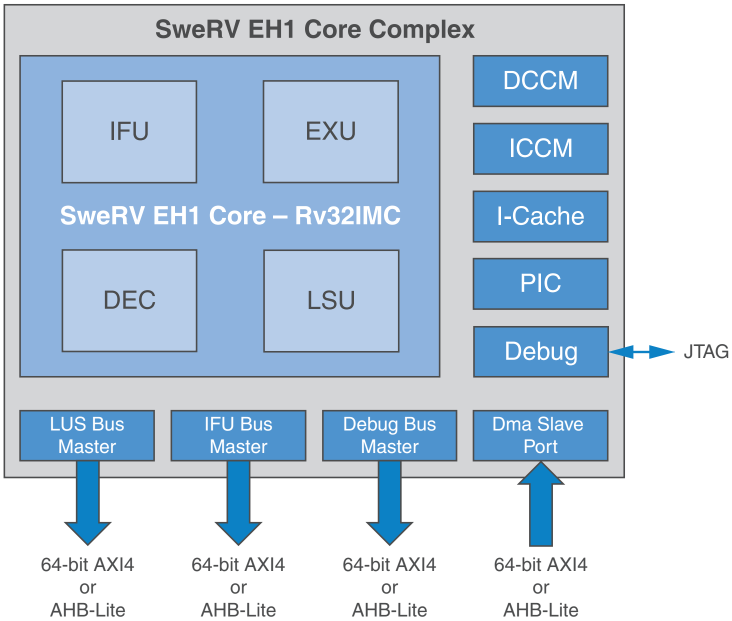


Figure 7.75 SweRV EH1 Core Complex (Courtesy of Western Digital Corporation, *RISC-V SweRV™ EH1 Programmer's Reference Manual*, ©2020)

splits into five parallel paths: a load/store path, two paths for integer instructions (such as `add`, `sub`, and `xor`), one path for multiply, and one path for divide—that is out of the pipeline because of its 34-cycle delay. The last two pipeline stages are the commit and writeback stages. The commit stage is required because of out-of-order execution and store buffers. The final stage, the writeback stage, writes results back to the register file, if needed.

Figure 7.75 shows a block diagram of the SweRV EH1 Core Complex. It includes the processor (labeled SweRV EH1 Core in the figure), instruction cache (I-Cache), data and instruction memories (DCCM and ICCM—data and instruction closely coupled memories), programmable interrupt controller (PIC), JTAG debugger interface, and memory/debug interfaces that can be configured as AXI4 or AHB-Lite busses. The processor consists of instruction fetch (IFU), decode (DEC), execute (EXU), and load/store (LSU) units. The IFU encompasses both pipeline fetch stages; DEC includes the align and decode stages; and EXU encompasses all other stages except the load/store pipeline, which is in the load/store unit (LSU). The system includes a 4-way set-associative instruction cache that can be configured as 16 to 256 KiB. The DCCM and ICCM are called *closely coupled* memories because they are low-latency on-chip memories, and they can be configured as 4 to 512 KiB.

Robert Golla is a Senior Fellow at Western Digital and was responsible for the architecture of Western Digital's EH1, EL2, and EH2 RISC-V open-source embedded processor cores. He also architected Oracle's T4, M7, M8, and M9 out-of-order cores, Sun's N2 multithreaded core, and Motorola's embedded e500 core and contributed to Cyrix's next generation x86 M3, NXP's low-power 603 and 603e, and IBM's POWER1 and POWER2 microprocessors. He has over 50 patents related to microprocessor design.

The RISC-V FPGA (RVfpga) course from Imagination Technologies shows how to target the SweRV EH1 core to an FPGA and how to compile and run programs on the RISC-V core. This free course also provides labs and exercises that show how to expand the core to add peripherals, how to understand and modify the pipeline, superscalar execution, branch predictor, and memory hierarchy, and how to use features such as timers, interrupts, and performance counters. These labs and materials are freely available from the Imagination Technologies University Program at https://university.imgtec.com/rvfpga/.

7.9 SUMMARY

This chapter has described three ways to build processors, each with different performance, area, and cost trade-offs. We find this topic almost magical: how can such a seemingly complicated device as a microprocessor actually be simple enough to fit in a half-page schematic? Moreover, the inner workings, so mysterious to the uninitiated, are actually reasonably straightforward.

The microarchitectures have drawn together almost every topic covered in the text so far. Piecing together the microarchitecture puzzle illustrates the principles introduced in previous chapters, including the design of combinational and sequential circuits (covered in Chapters 1 through 3), the application of many of the building blocks (described in Chapter 5), and the implementation of the RISC-V architecture (introduced in Chapter 6). The microarchitectures can be described in a few pages of HDL using the techniques from Chapter 4.

Building the microarchitectures has also heavily relied on our techniques for managing complexity. The microarchitectural abstraction forms the link between the logic and architecture abstractions, forming the crux of this book on digital design and computer architecture. We also use the abstractions of block diagrams and HDL to succinctly describe the arrangement of components. The microarchitectures exploit regularity and modularity, reusing a library of common building blocks such as ALUs, memories, multiplexers, and registers. Hierarchy is used in numerous ways. The microarchitectures are partitioned into the datapath and control units, which themselves are partitioned into smaller units. Each of these units is built from logic blocks, which can be built from gates, which, in turn, can be built from transistors—all of which use the techniques developed in the first five chapters.

This chapter has compared single-cycle, multicycle, and pipelined microarchitectures for the RISC-V processor. All three microarchitectures implement the same subset of the RISC-V instruction set and have the same architectural state. The single-cycle processor is the most straightforward and has a CPI of 1.

The multicycle processor uses a variable number of shorter steps to execute instructions. It thus can reuse the ALU, rather than requiring several adders, and includes a unified memory. However, it does require several nonarchitectural registers to store results between steps. The multicycle design, in principle, could be faster because not all instructions must be equally long. In practice, it is generally slower, because it is limited by the slowest steps and by the sequencing overhead in each step.

The pipelined processor divides the single-cycle processor into five relatively fast pipeline stages. It adds pipeline registers between the stages to separate the five instructions that are simultaneously executing. It nominally has a CPI of 1, but hazards force stalls or flushes that increase the CPI slightly. Hazard resolution also costs some extra hardware and design complexity. The clock period ideally could be five times shorter than that of the single-cycle processor. In practice, it is not that short because it is limited by the slowest stage and by the sequencing overhead in each stage. Nevertheless, pipelining provides substantial performance benefits. All modern high-performance microprocessors use pipelining today.

Although the microarchitectures in this chapter implement only a subset of the RISC-V architecture, we have seen that supporting more instructions involves straightforward enhancements of the datapath and controller.

A major limitation of this chapter is that we have assumed an ideal memory system that is fast and large enough to store the entire program and data. In reality, large fast memories are prohibitively expensive. The next chapter shows how to get most of the benefits of a large fast memory by using a small fast memory that holds the most commonly used information and one or more larger but slower memories holding the rest of the information.