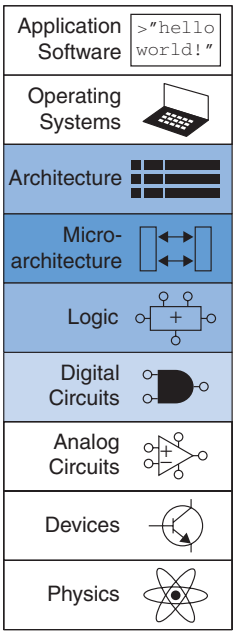
Microarchitecture

7



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.

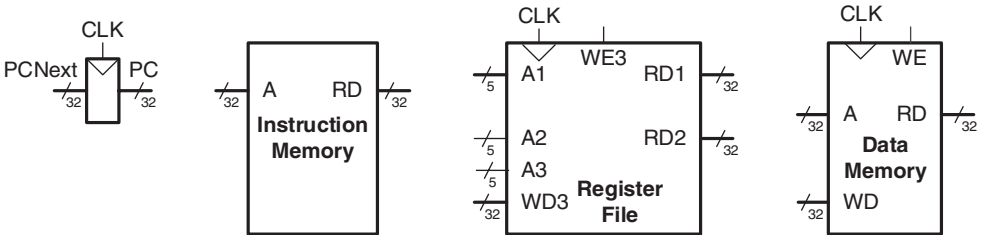


Figure 7.1 State elements of a RISC-V processor

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.

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.

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.

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.

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.

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 | 10 = 0110_2 | 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

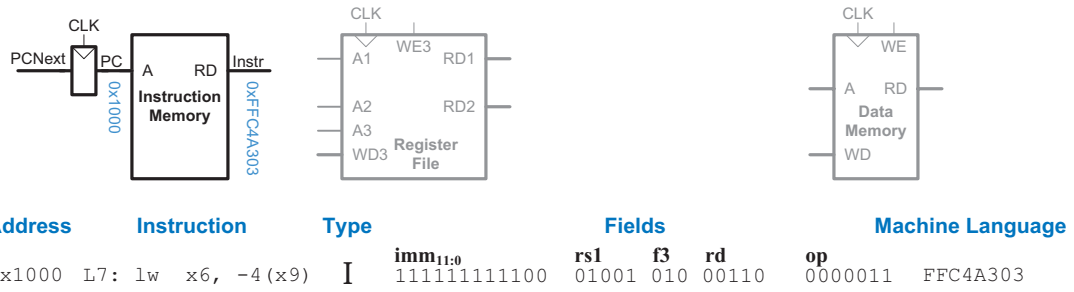


Figure 7.3 Fetch instruction from memory

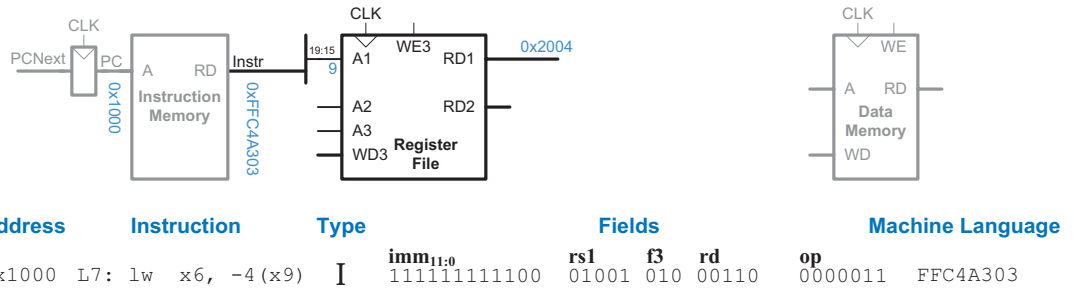


Figure 7.4 Read source operand from register file

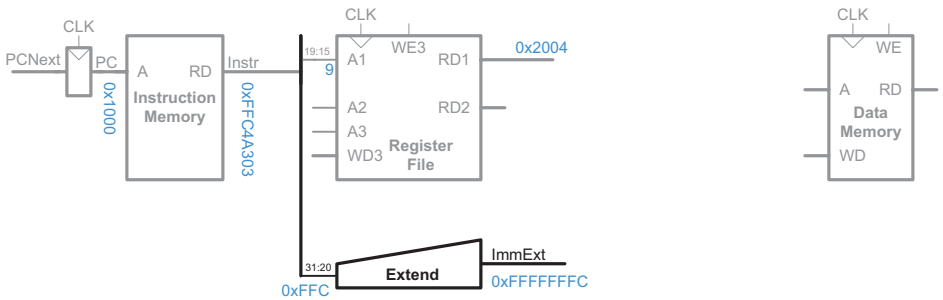
Instr is `lw`, 0xFFC4A303, as also shown at the bottom of Figure 7.3. These sample values are annotated in light blue on the diagram.

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`

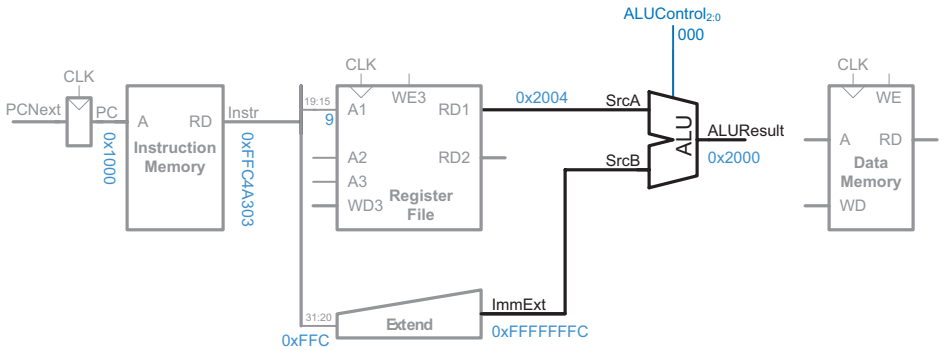
For the `lw` instruction, the next step is to read the source register containing the base address. Recall that `lw` is an I-type instruction, and the base register is specified in the **rs1** field of the instruction, $Instr_{19:15}$. These bits of the instruction connect to the *A1* address input of 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
			$\text{imm}_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.5 Sign-extend the immediate

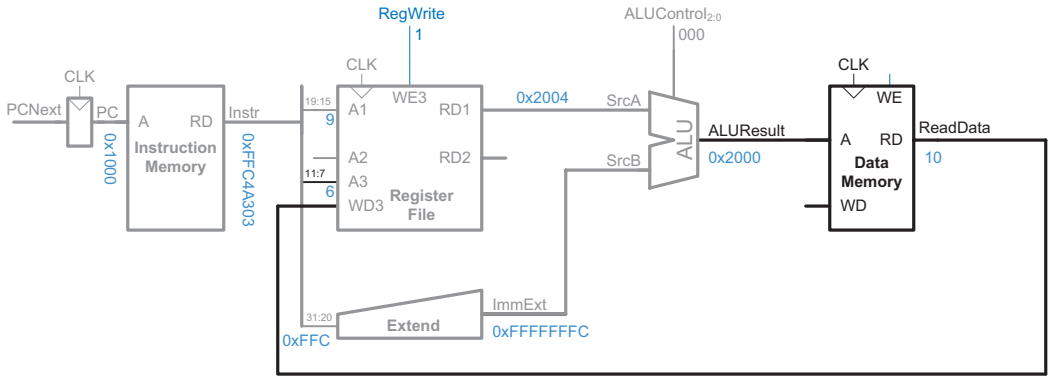


Address	Instruction	Type	Fields					Machine Language
			$\text{imm}_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

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



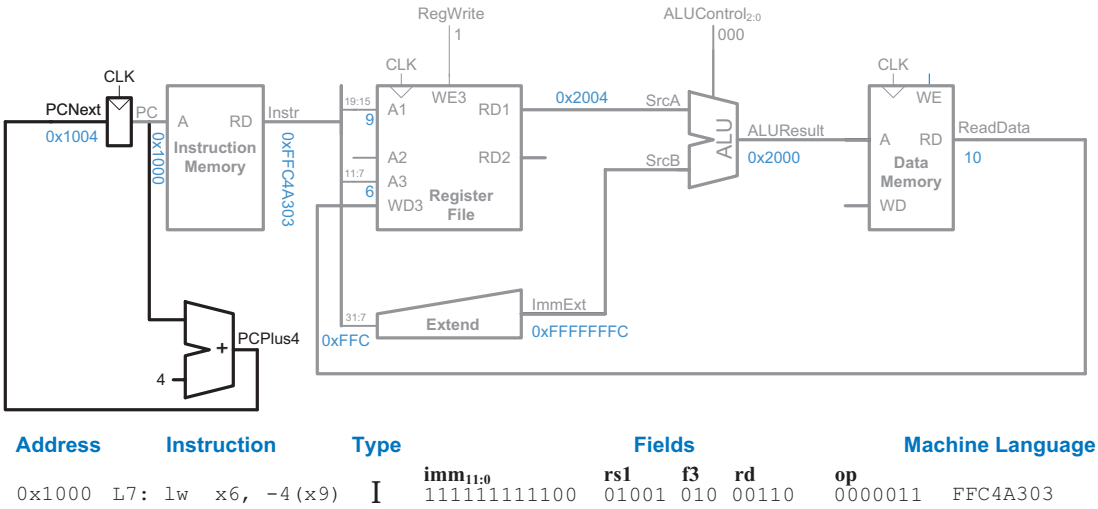
Address	Instruction	Type	Fields					Machine Language
			$\text{imm}_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

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.



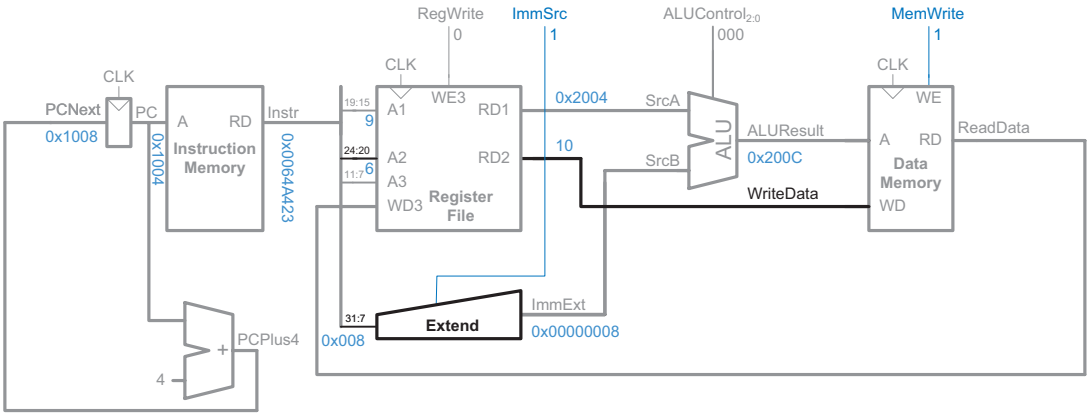
Address	Instruction	Type	Fields					Machine Language
			$\mathbf{imm_{11:0}}$	**rs1**	**f3**	**rd**	**op**	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

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	Fields					Machine Language
			$\mathbf{imm_{11:5}}$	**rs2**	**rs1**	**f3**	$\mathbf{imm_{4:0}}$	**op**	
0x1004	`sw x6, 8(x9)`	S	0000000	00110	01001	010	01000	0100011	0064A423

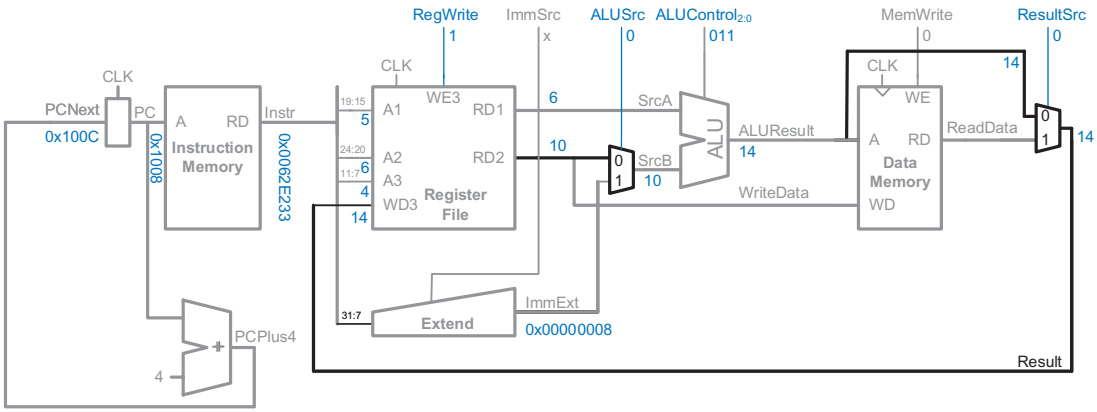
Figure 7.9 Write data to memory for `sw` instruction

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).

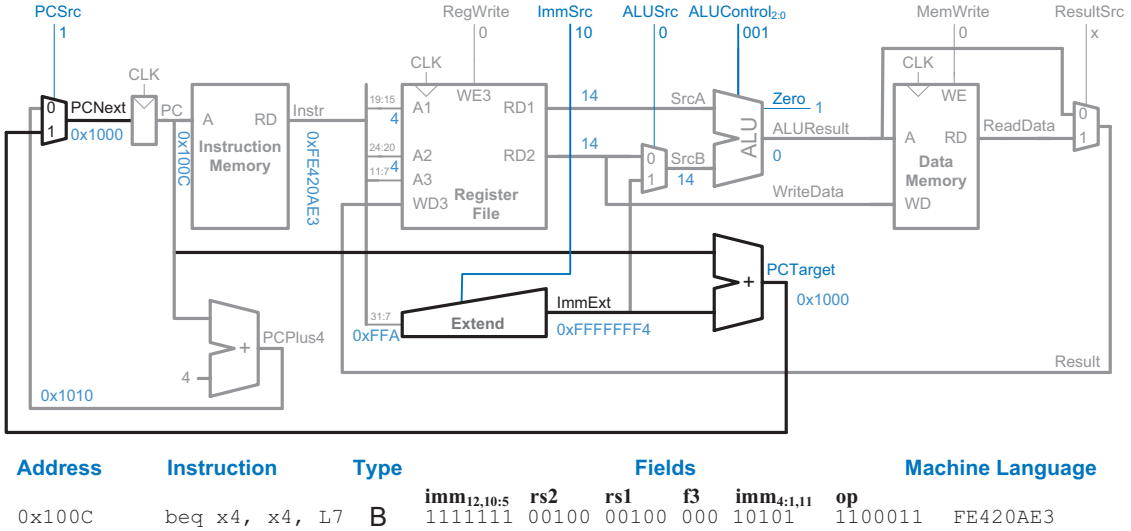
The branch offset is a 13-bit signed immediate stored in the 12-bit immediate field of the B-type instruction. Thus, the Extend logic needs yet another mode to choose the proper immediate. *ImmSrc* is increased to 2 bits, using the encoding from Table 7.1. *ImmExt* is now either the

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

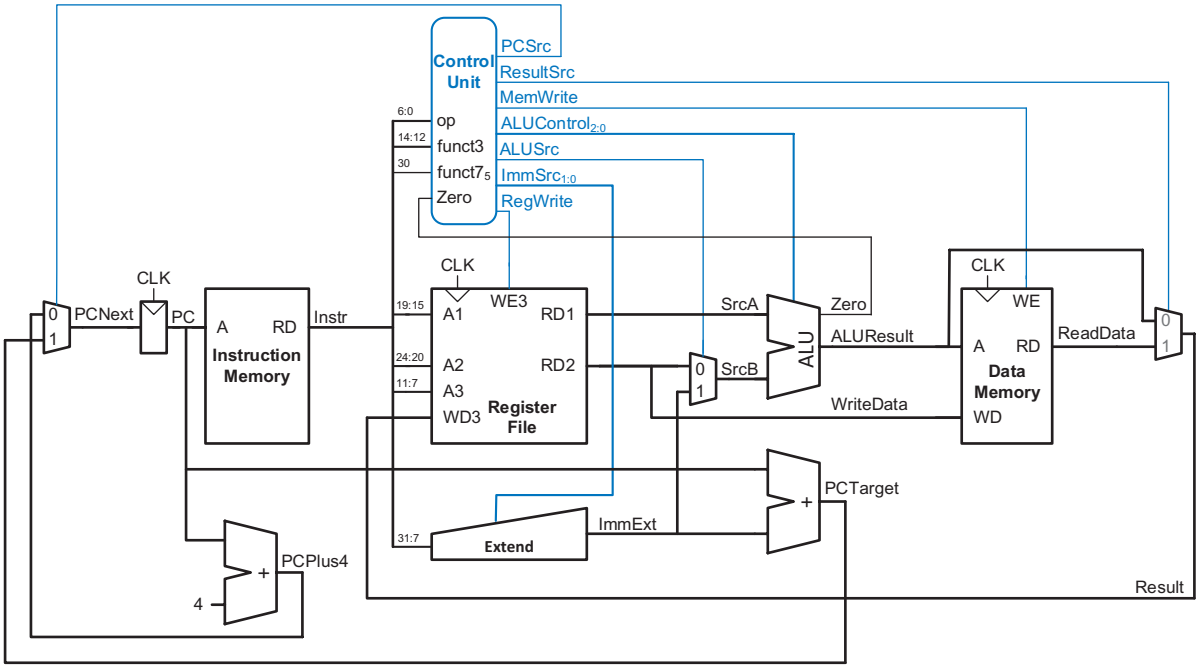


Figure 7.12 Complete single-cycle processor

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.

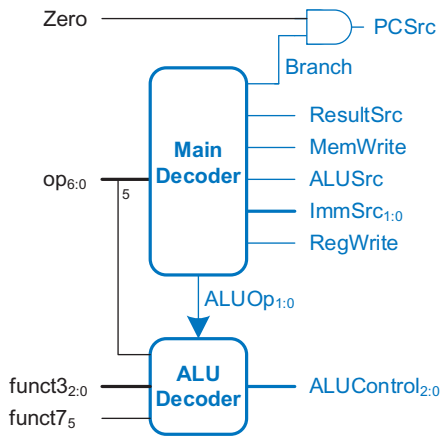


Figure 7.13 Single-cycle processor control unit

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 $\mathbf{funct7}_5$ and $\mathbf{op}_5$ 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 $\mathbf{op}_5$ and $\mathbf{funct7}_5$ 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 $\mathbf{funct7}_5$ 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 $\mathbf{op}_5$ and $\mathbf{funct7}_5$ 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.

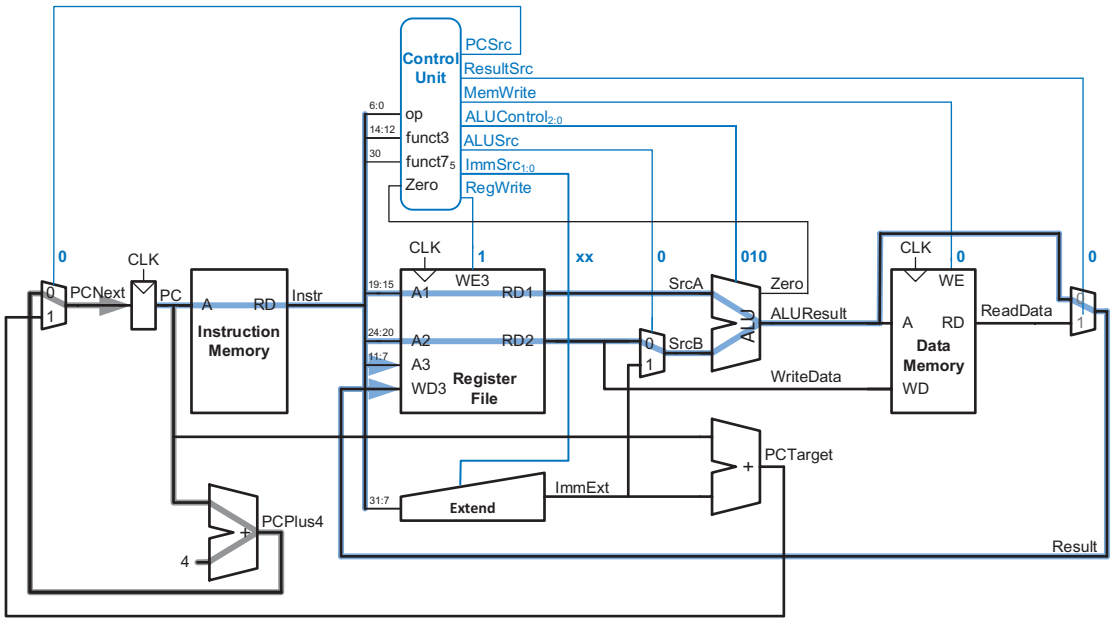


Figure 7.14 Control signals and data flow while executing an `and` instruction

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.

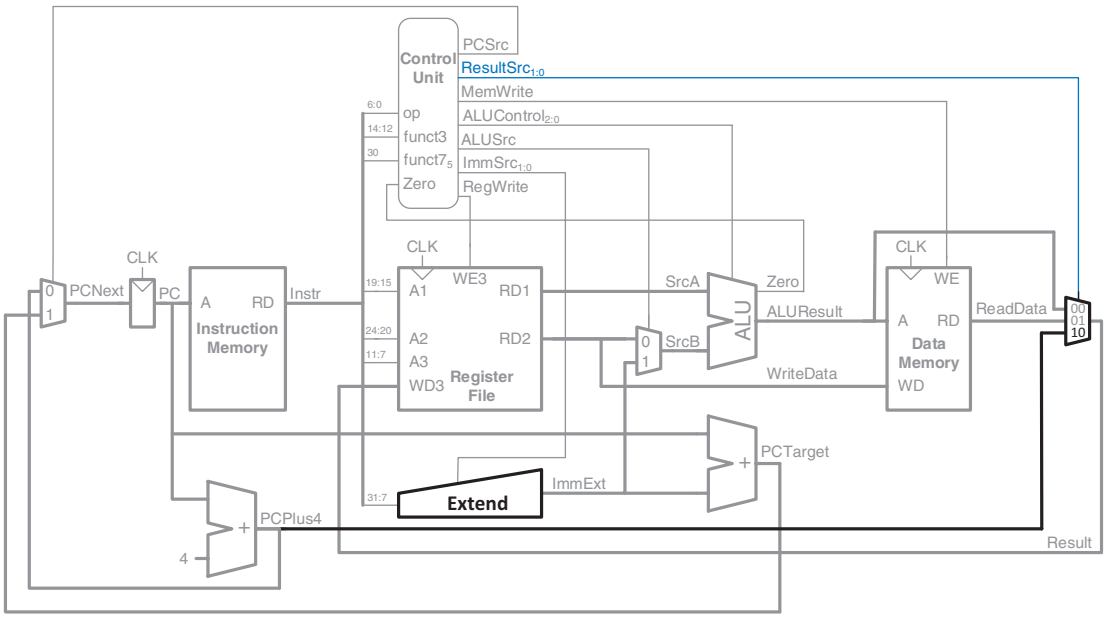


Figure 7.15 Enhanced datapath for `jal`

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

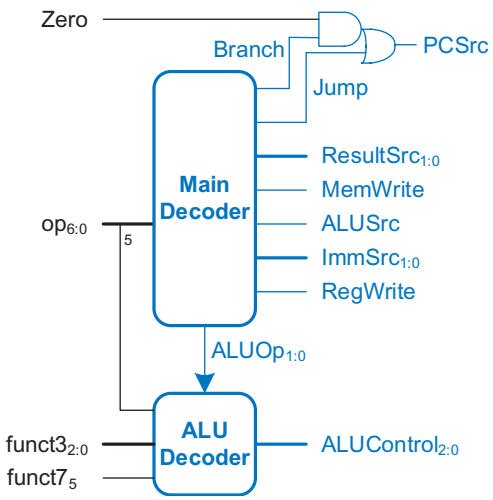


Figure 7.16 Enhanced control unit for `jal`

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

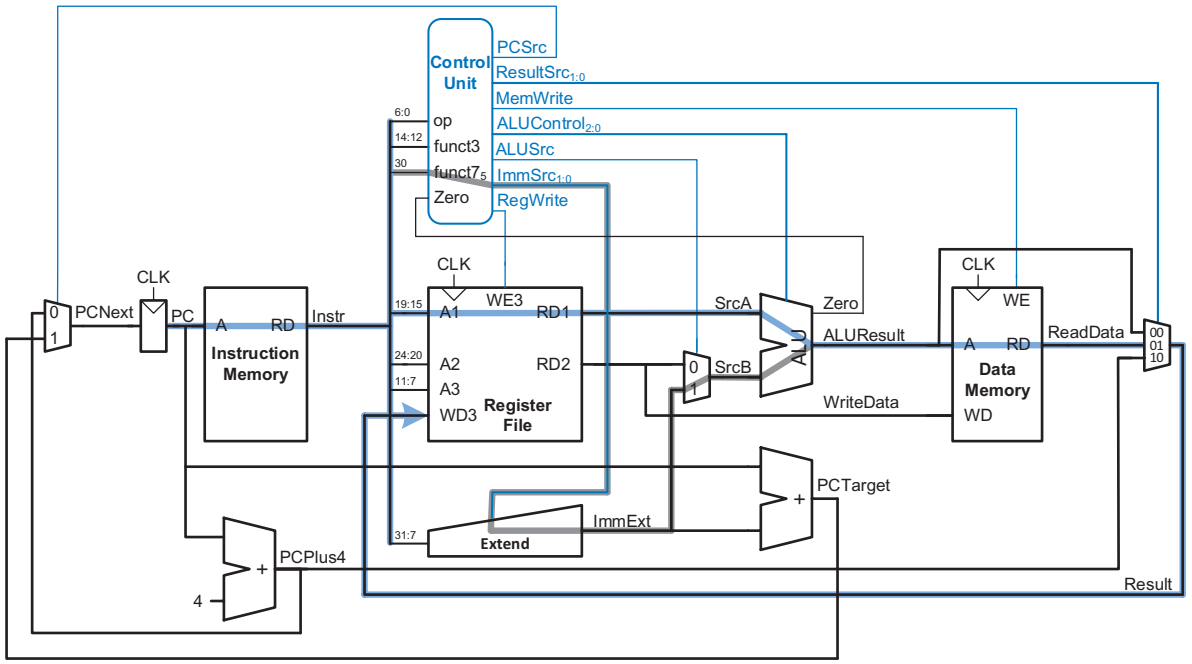


Figure 7.17 Critical path for `lw`

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

$$T_{c\_single} = t_{pcq\_PC} + t_{mem} + \max[t_{RFread}, \; t_{dec} + t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup} \quad (7.2)$$

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:

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

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 $T_{single} = (100 \times 10^9 \text{ instruction})\ (1 \text{ cycle/instruction})\ (750 \times 10^{-12} \text{ s/cycle}) = 75$ 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