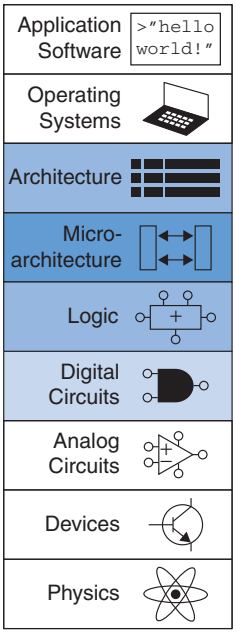
Microarchitecture

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 ARM architecture, which specifies the programmer's view of the ARM 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, ALUs, finite state machines (FSMs), memories, and other logic building blocks needed to implement an architecture. A particular architecture, such as ARM, 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 ARM processor consists of 16 32-bit registers and the status register. Any ARM 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 microarchitectures contain

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-800056-4.00007-8

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 consider only a subset of the ARM instruction set. Specifically, we handle the following instructions:

- Data-processing instructions: `ADD`, `SUB`, `AND`, `ORR` (with register and immediate addressing modes but no shifts)
- Memory instructions: `LDR`, `STR` (with positive immediate offset)
- Branches: `B`

These particular instructions were chosen because they are sufficient to write many interesting 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 ARM 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, registers, and status register). 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 five state elements: the program counter, register file, status register, and instruction and data memories.

In Figure 7.1, heavy lines are used to indicate 32-bit data busses. Medium lines are used to indicate narrower busses, such as the 4-bit address busses on the register file. Narrow lines indicate 1-bit buses, and blue lines are used for control signals, such as the register file write enable. We use this convention throughout the chapter to avoid cluttering diagrams with bus widths. Also, state elements usually have a reset input to put them into a known state at start-up. Again, to save clutter, this reset is not shown.

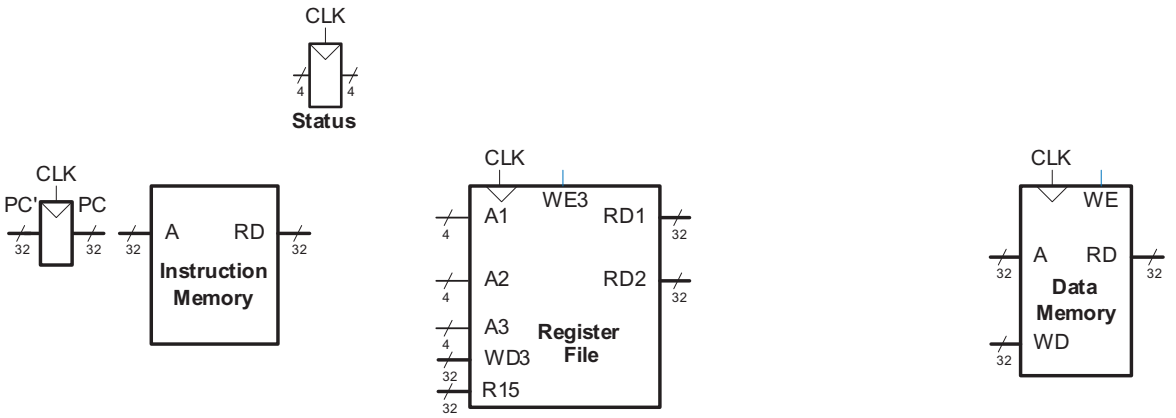


Figure 7.1 State elements of ARM processor

Although the *program counter* (PC) is logically part of the register file, it is read and written on every cycle independent of the normal register file operation and is more naturally built as a stand-alone 32-bit register. Its output, *PC*, points to the current instruction. Its input, *PC′*, indicates the address of the next instruction.

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 15-element × 32-bit register file holds registers R0–R14 and has an additional input to receive R15 from the PC. The register file has two read ports and one write port. The read ports take 4-bit address inputs, *A1* and *A2*, each specifying one of $2^4 = 16$ registers as source operands. They read the 32-bit register values onto read data outputs *RD1* and *RD2*, respectively. The write port takes a 4-bit address input, *A3*; a 32-bit write data input, *WD3*; a write enable input, *WE3*; and a clock. If the write enable is asserted, then the register file writes the data into the specified register on the rising edge of the clock. A read of R15 returns the value from the PC plus 8, and writes to R15 must be specially handled to update the PC because it is separate from the register file.

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 address *A* onto *RD*.

Resetting the PC

At the very least, the program counter must have a reset signal to initialize its value when the processor turns on. ARM processors normally initialize the PC to `0x00000000` on reset, and we start our programs there.

Treating the PC as part of the register file complicates the system design, and complexity ultimately means more gates and higher power consumption. Most other architectures treat the PC as a special register that is only updated by branches, not by ordinary data-processing instructions. As described in Section 6.7.6, ARM's 64-bit ARMv8 architecture also makes the PC a special register separate from the register file.

[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 OS can load a new program into memory. The multicycle microarchitecture described in Section 7.4 is more realistic in that it uses a combined memory for instructions and data that can be both read and written.

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

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.

7.1.3 Microarchitectures

In this chapter, we develop three microarchitectures for the ARM architecture: single-cycle, multicycle, and pipelined. They differ in the way that the state elements are connected together and in the amount of nonarchitectural state.

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 adding several nonarchitectural registers to hold intermediate results. The multicycle processor executes only one instruction at a time, but each instruction takes multiple clock cycles. The multicycle processor requires only a single memory, accessing it on one cycle to fetch the instruction and on another to read or write data. Therefore, 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

Intel processors have been pipelined since the 80486 was introduced in 1989. Nearly all RISC microprocessors are also pipelined. ARM processors have been pipelined since the original ARM1 in 1985. A pipelined ARM Cortex-M0 requires only about 12,000 logic gates, so in a modern integrated circuit it is so small that one needs a microscope to see it and the manufacturing cost is a fraction of a penny. Combined with memory and peripherals, a commercial Cortex-M0 chip such as the Freescale Kinetis still costs less than 50 cents. Thus, pipelined processors are replacing their slower multicycle siblings in even the most cost-sensitive applications.

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.

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

$$Execution\ Time = \left(\#instructions\right)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right) \qquad (7.1)$$

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

Dhrystone, CoreMark, and SPEC are three popular benchmarks. The first two are *synthetic benchmarks* comprising 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 SPEC CINT2006 benchmark from the Standard Performance Evaluation Corporation is composed of real programs, including h264ref (video compression), sjeng (an artificial intelligence chess player), hmmer (protein sequence analysis), and gcc (a C compiler). The benchmark is widely used for high-performance processors because it stresses the entire CPU in a representative way.

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 an ARM 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 on 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 have historically doubled transistor speeds every 4–6 years, 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 dial-up 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. We conclude by analyzing the performance of the single-cycle processor.

7.3.1 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 status register is part of the controller and will be omitted while we focus on the datapath.

The program counter contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. Figure 7.2 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*.

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

LDR

For the `LDR` instruction, the next step is to read the source register containing the base address. This register is specified in the *Rn* field of the instruction, $Instr_{19:16}$. These bits of the instruction are connected to the address input of one of the register file ports, *A1*, as shown in Figure 7.3. The register file reads the register value onto *RD1*.

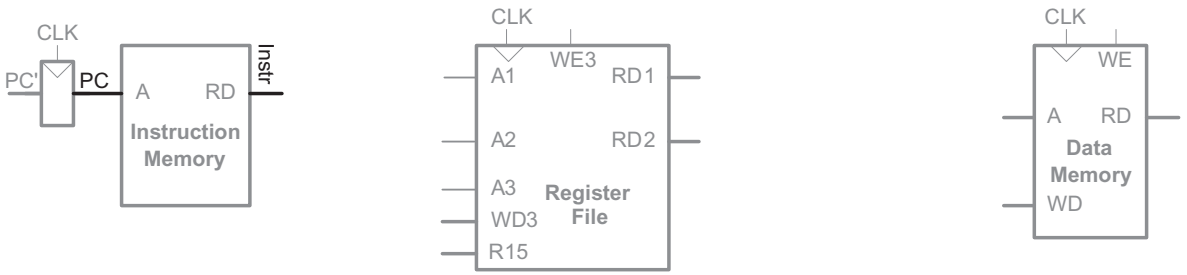


Figure 7.2 Fetch instruction from memory

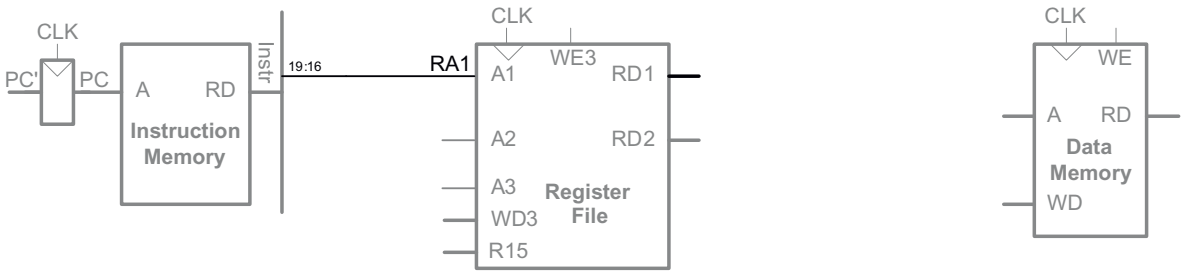


Figure 7.3 Read source operand from register file

The LDR instruction also requires an offset. The offset is stored in the immediate field of the instruction, $Instr_{11:0}$. It is an unsigned value, so it must be zero-extended to 32 bits, as shown in Figure 7.4. The 32-bit value is called *ExtImm*. Zero extension simply means prepending leading zeros: $ImmExt_{31:12} = 0$ and $ImmExt_{11:0} = Instr_{11:0}$.

The processor must add the base address to the offset to find the address to read from memory. Figure 7.5 introduces an ALU to perform this addition. The ALU receives two operands, *SrcA* and *SrcB*. *SrcA* comes from the register file, and *SrcB* comes from the extended immediate. The ALU can perform many operations, as was described in Section 5.2.4. The 2-bit *ALUControl* signal specifies the operation. The ALU generates a 32-bit *ALUResult*. For an LDR instruction, *ALUControl* should be set to 00 to perform addition. *ALUResult* is sent to the data memory as the address to read, as shown in Figure 7.5.

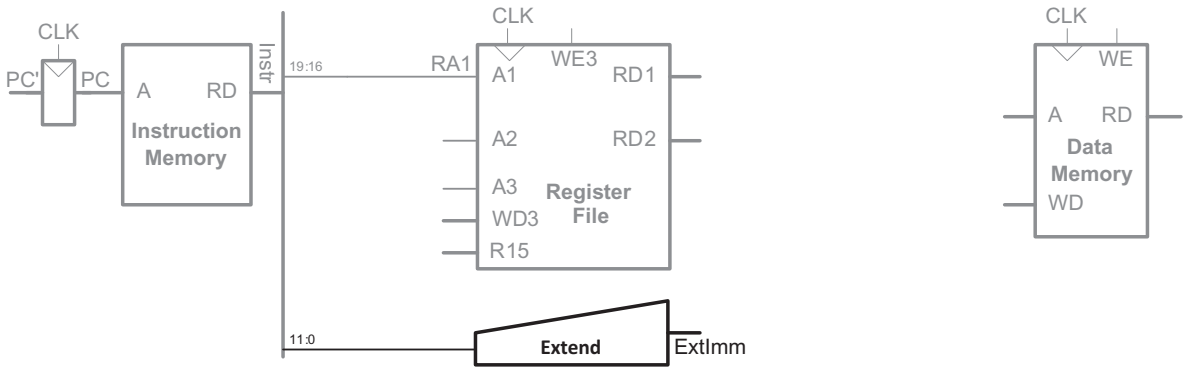


Figure 7.4 Zero-extend the immediate

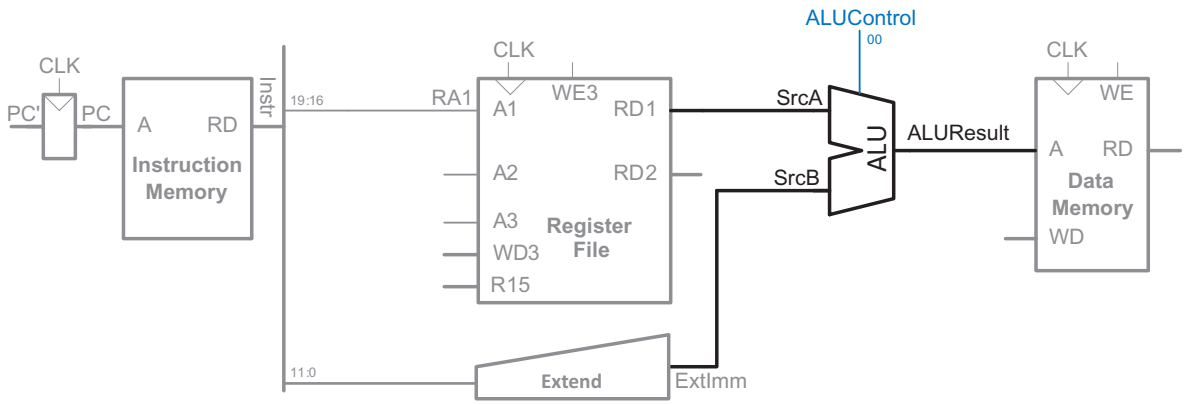


Figure 7.5 Compute memory address

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 7.6. Port 3 of the register file is the write port. The destination register for the `LDR` instruction is specified in the *Rd* field, $Instr_{15:12}$, which is connected to the port 3 address input, *A3*, of the register file. The *ReadData* bus is connected to the port 3 write data input, *WD3*, of the register file. A control signal called *RegWrite* is connected to the port 3 write enable input, *WE3*, and is asserted during an `LDR` 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.

While the instruction is being executed, the processor must compute the address of the next instruction, *PC′*. Because instructions are 32 bits (4 bytes), the next instruction is at *PC* + 4. Figure 7.7 uses an adder to

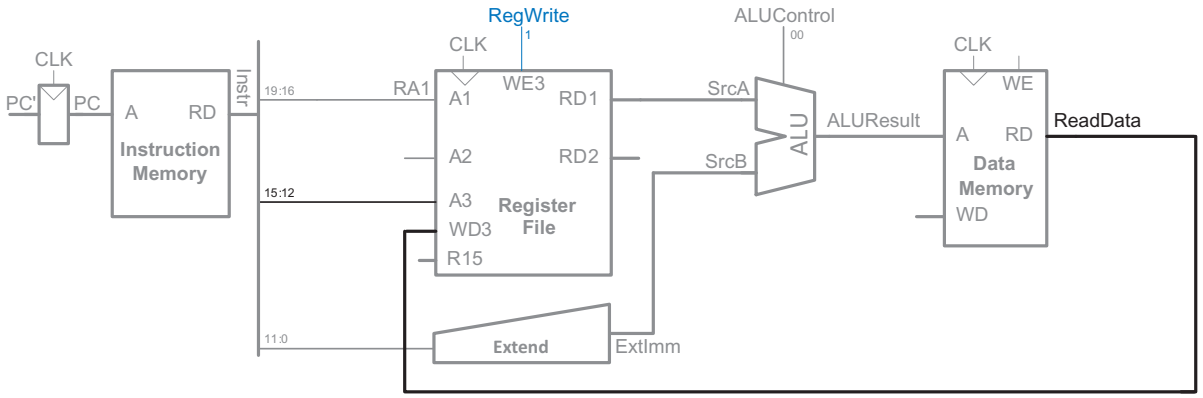


Figure 7.6 Write data back to register file

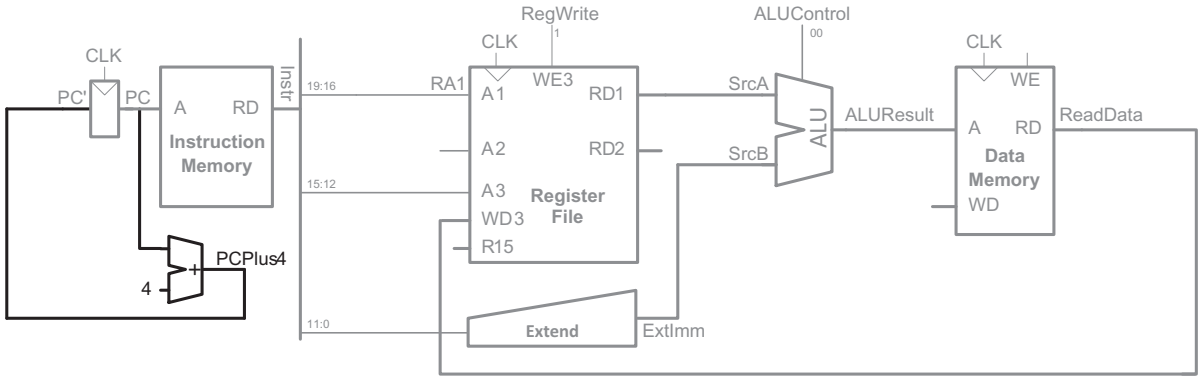


Figure 7.7 Increment program counter

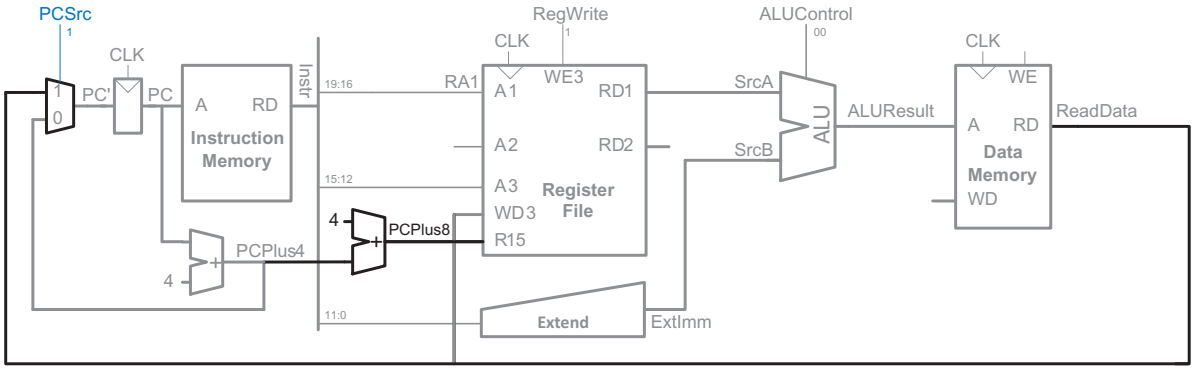


Figure 7.8 Read or write program counter as R15

increment the PC by 4. The new address is written into the program counter on the next rising edge of the clock. This completes the datapath for the LDR instruction, except for a sneaky case of the base or destination register being R15.

Recall from Section 6.4.6 that in the ARM architecture, reading register R15 returns $PC + 8$. Therefore, another adder is needed to further increment the PC and pass this sum to the *R15* port of the register file. Similarly, writing register R15 updates the PC. Therefore, *PC* may come from the result of the instruction (*ReadData*) rather than *PCPlus4*. A multiplexer chooses between these two possibilities. The *PCSrc* control signal is set to 0 to choose *PCPlus4* or 1 to choose *ReadData*. These PC-related features are highlighted in Figure 7.8.

STR

Next, let us extend the datapath to also handle the STR instruction. Like LDR, STR reads a base address from port 1 of the register file and zero-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.

The STR instruction also reads a second register from the register file and writes it to the data memory. Figure 7.9 shows the new connections for this function. The register is specified in the *Rd* field, $Instr_{15:12}$, which is connected to the *A2* port of the register file. The register value is read onto the *RD2* port. It is connected to the write data (*WD*) port of the data memory. The write enable port of the data memory, *WE*, is controlled by *MemWrite*. For an STR instruction: $MemWrite = 1$ to write the data to memory; $ALUControl = 00$ 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 that this *ReadData* is ignored because $RegWrite = 0$.

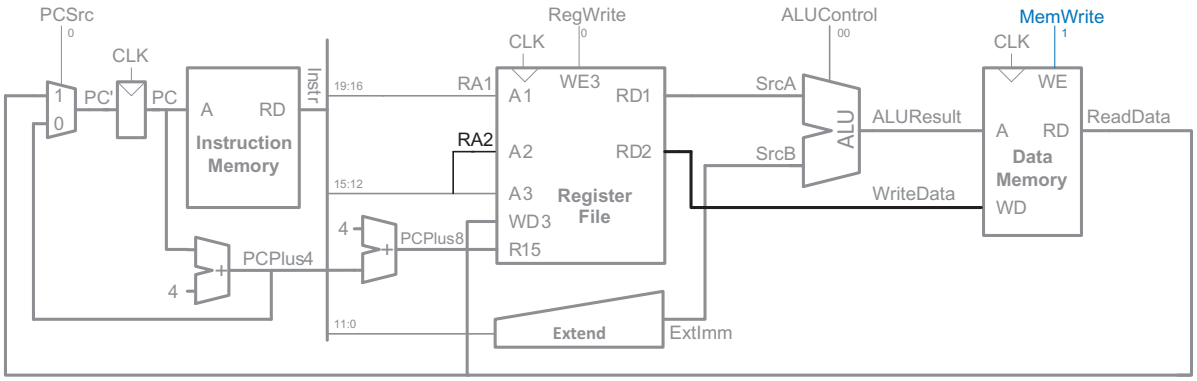


Figure 7.9 Write data to memory for STR instruction

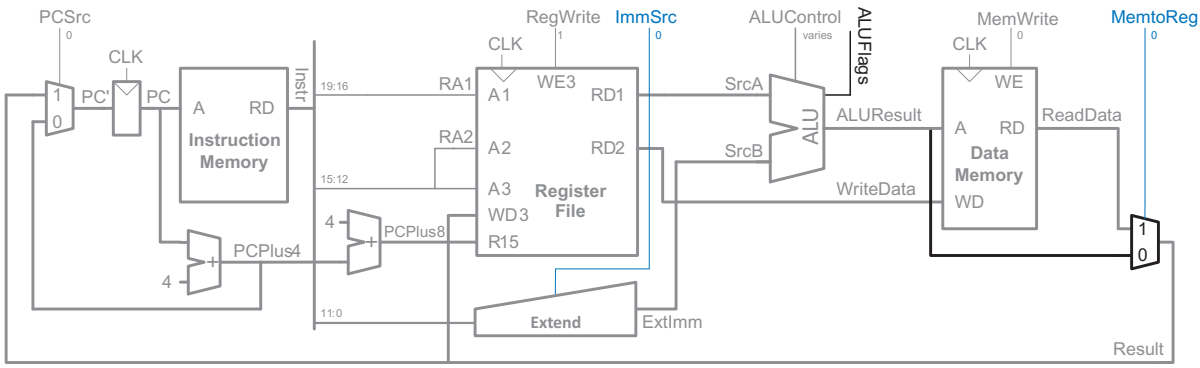


Figure 7.10 Datapath enhancements for data-processing instructions with immediate addressing

Data-Processing Instructions with Immediate Addressing

Next, consider extending the datapath to handle the data-processing instructions, ADD, SUB, AND, and ORR, using the immediate addressing mode. All of these instructions read a source register from the register file and an immediate from the low bits of the instruction, perform some ALU operation on them, and write the result back to a third register. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware using different *ALUControl* signals. As described in Section 5.2.4, *ALUControl* is 00 for ADD, 01 for SUB, 10 for AND, or 11 for ORR. The ALU also produces four flags, $ALUFlags_{3:0}$ (Zero, Negative, Carry, oVerflow), that are sent back to the controller.

Figure 7.10 shows the enhanced datapath handling data-processing instructions with an immediate second source. Like LDR, the datapath reads the first ALU source from port 1 of the register file and extends the immediate from the low bits of *Instr*. However, data-processing

instructions use only an 8-bit immediate rather than a 12-bit immediate. Therefore, we provide the *ImmSrc* control signal to the Extend block. When it is 0, *ExtImm* is zero-extended from $Instr_{7:0}$ for data-processing instructions. When it is 1, *ExtImm* is zero-extended from $Instr_{11:0}$ for `LDR` or `STR`.

For `LDR`, the register file received its write data from the data memory. However, data-processing instructions write *ALUResult* to the register file. Therefore, we add another multiplexer to choose between *ReadData* and *ALUResult*. We call its output *Result*. The multiplexer is controlled by another new signal, *MemtoReg*. *MemtoReg* is 0 for data-processing instructions to choose *Result* from *ALUResult*; it is 1 for `LDR` to choose *ReadData*. We do not care about the value of *MemtoReg* for `STR` because `STR` does not write the register file.

Data-Processing Instructions with Register Addressing

Data-processing instructions with register addressing receive their second source from *Rm*, specified by $Instr_{3:0}$, rather than from the immediate. Thus, we must add multiplexers on the inputs of the register file and ALU to select this second source register, as shown in Figure 7.11.

RA2 is chosen from the *Rd* field ($Instr_{15:12}$) for `STR` and the *Rm* field ($Instr_{3:0}$) for data-processing instructions with register addressing based on the *RegSrc* control signal. Similarly, based on the *ALUSrc* control signal, the second source to the ALU is selected from *ExtImm* for instructions using immediates and from the register file for data-processing instructions with register addressing.

B

Finally, we extend the datapath to handle the `B` instruction, as shown in Figure 7.12. The branch instruction adds a 24-bit immediate to $PC + 8$ and writes the result back to the PC. The immediate is multiplied by 4

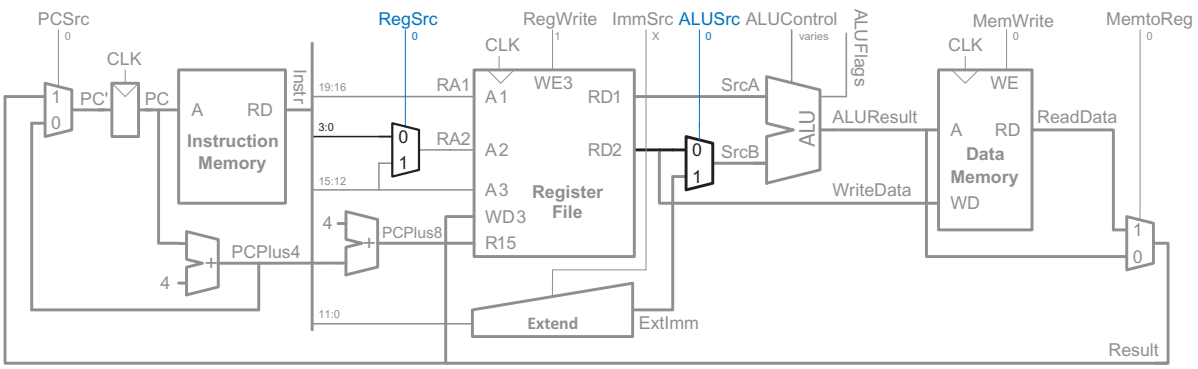


Figure 7.11 Datapath enhancements for data-processing instructions with register addressing

and sign extended. Therefore, the Extend logic needs yet another mode. *ImmSrc* is increased to 2 bits, with the encoding given in Table 7.1.

PC + 8 is read from the first port of the register file. Therefore, a multiplexer is needed to choose R15 as the *RA1* input. This multiplexer is controlled by another bit of *RegSrc*, choosing $Instr_{19:16}$ for most instructions but 15 for `B`.

MemtoReg is set to 0 and *PCSrc* is set to 1 to select the new PC from *ALUResult* for the branch.

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 connecting the state elements is systematically added. In the next section, we consider how to compute the control signals that direct the operation of our datapath.

7.3.2 Single-Cycle Control

The control unit computes the control signals based on the *cond*, *op*, and *funct* fields of the instruction ($Instr_{31:28}$, $Instr_{27:26}$, and $Instr_{25:20}$) as well as the flags and whether the destination register is the PC. The controller also stores the current status flags and updates them appropriately. Figure 7.13 shows the entire single-cycle processor with the control unit attached to the datapath.

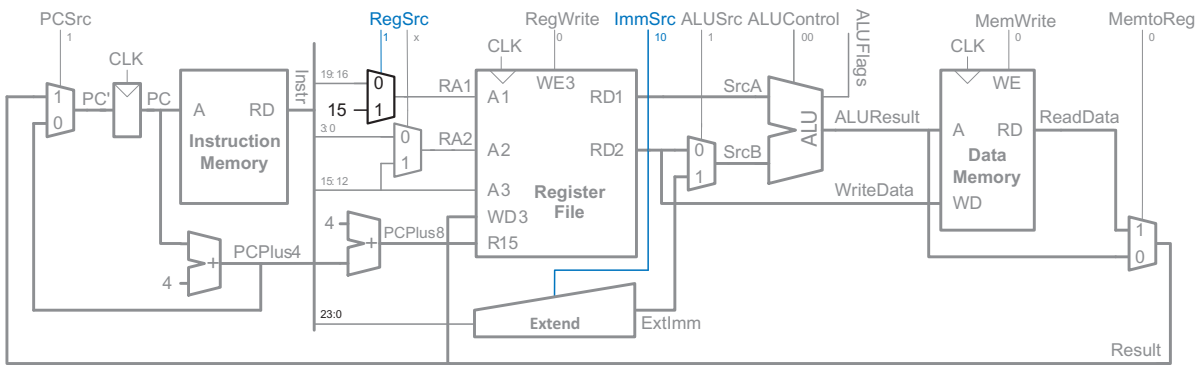


Figure 7.12 Datapath enhancements for `B` instruction

Table 7.1 ImmSrc Encoding

ImmSrc	ExtImm	Description
00	{24 0s} $Instr_{7:0}$	8-bit unsigned immediate for data-processing
01	{20 0s} $Instr_{11:0}$	12-bit unsigned immediate for `LDR/STR`
10	{6 $Instr_{23}$} $Instr_{23:0}$ 00	24-bit signed immediate multiplied by 4 for `B`

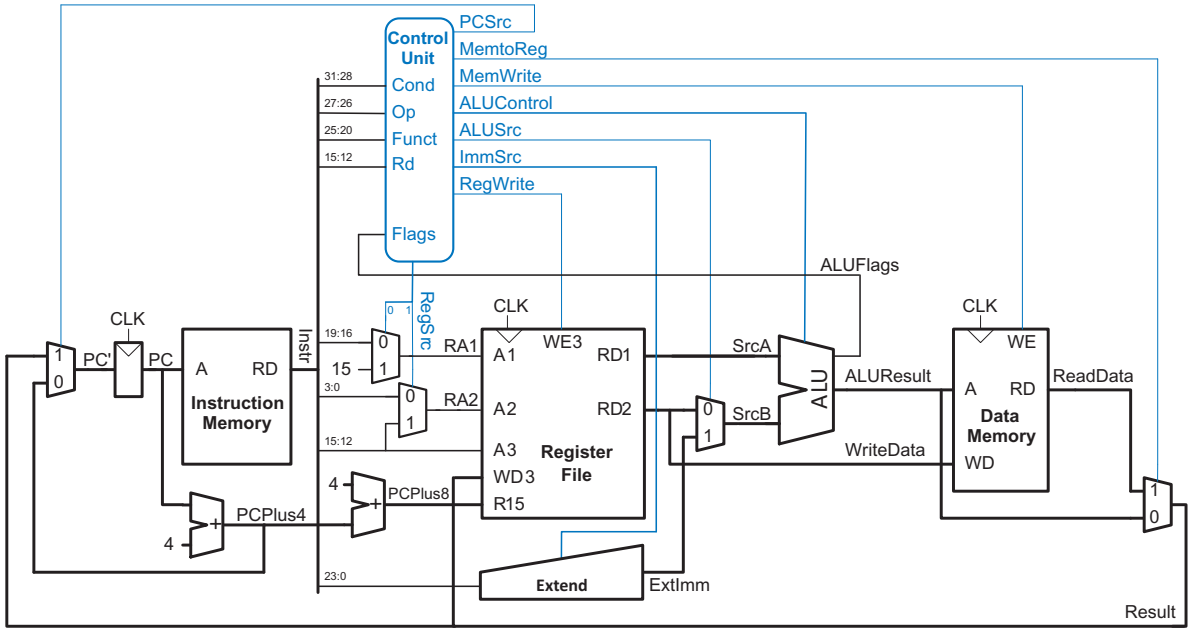


Figure 7.13 Complete single-cycle processor

Figure 7.14 shows a detailed diagram of the controller. We partition the controller into two main parts: the Decoder, which generates control signals based on *Instr*, and the Conditional Logic, which maintains the status flags and only enables updates to architectural state when the instruction should be conditionally executed. The Decoder, shown in Figure 7.14(b), is composed of a Main Decoder that produces most of the control signals, an ALU Decoder that uses the *Funct* field to determine the type of data-processing instruction, and PC Logic to determine whether the PC needs updating due to a branch or a write to R15.

The behavior of the Main Decoder is given by the truth table in Table 7.2. The Main Decoder determines the type of instruction: Data-Processing Register, Data-Processing Immediate, `STR`, `LDR`, or `B`. It produces the appropriate control signals to the datapath. It sends *MemtoReg*, *ALUSrc*, $ImmSrc_{1:0}$, and $RegSrc_{1:0}$ directly to the datapath. However, the write enables *MemW* and *RegW* must pass through the Conditional Logic before becoming datapath signals *MemWrite* and *RegWrite*. These write enables may be killed (reset to 0) by the Conditional Logic if the condition is not satisfied. The Main Decoder also generates the *Branch* and *ALUOp* signals, which are used within the controller to indicate that the instruction is `B` or data-processing, respectively. The logic for the Main Decoder can be developed from the truth table using your favorite techniques for combinational logic design.

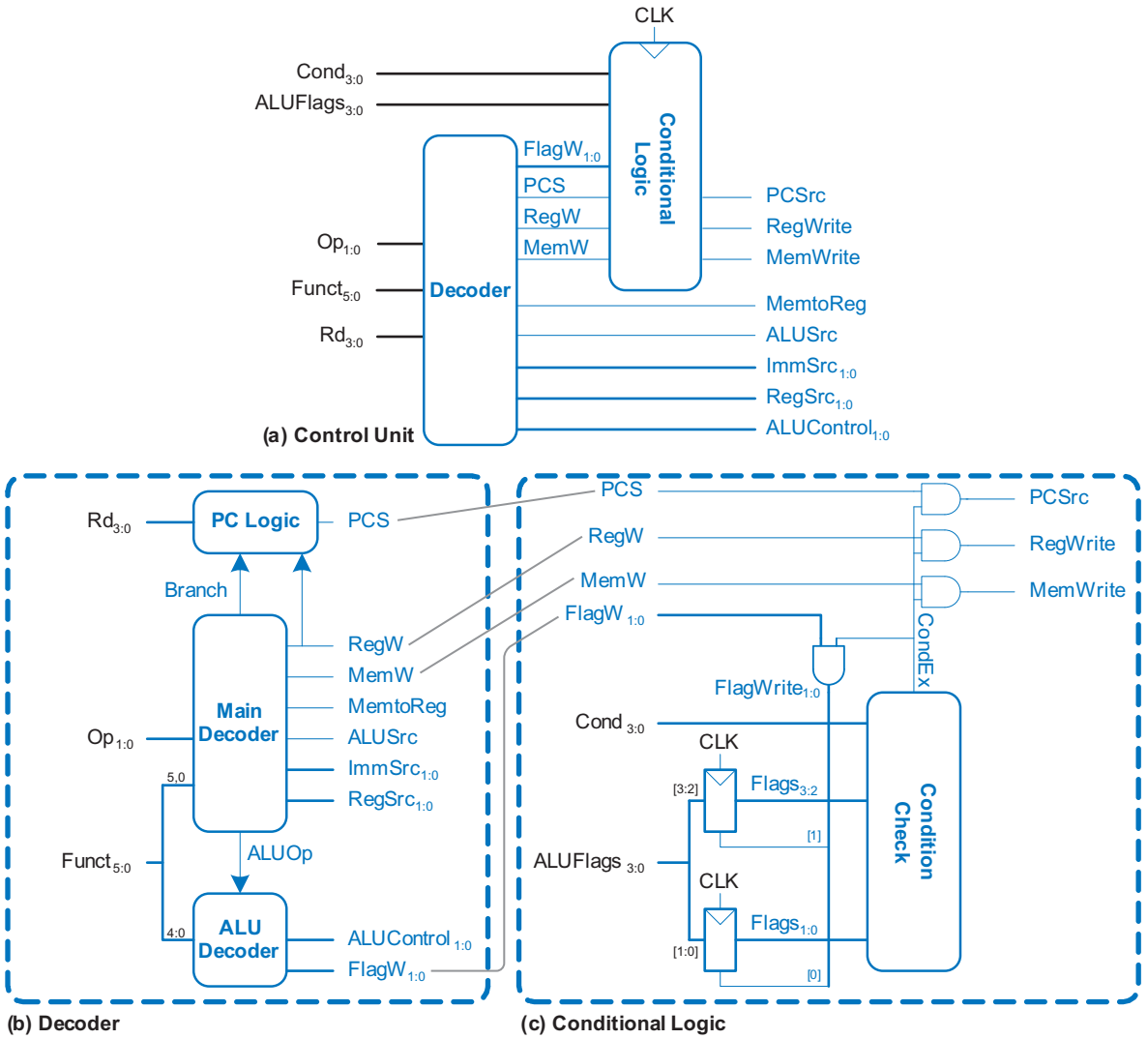


Figure 7.14 Single-cycle control unit

The behavior of the ALU Decoder is given by the truth tables in Table 7.3. For data-processing instructions, the ALU Decoder chooses *ALUControl* based on the type of instruction (`ADD`, `SUB`, `AND`, `ORR`). Moreover, it asserts *FlagW* to update the status flags when the *S*-bit is set. Note that `ADD` and `SUB` update all flags, whereas `AND` and `ORR` only update the *N* and *Z* flags, so two bits of *FlagW* are needed: $FlagW_1$ for updating *N* and *Z* ($Flags_{3:2}$), and $FlagW_0$ for updating *C* and *V* ($Flags_{1:0}$). $FlagW_{1:0}$ is killed by the Conditional Logic when the condition is not satisfied ($CondEx = 0$).

Table 7.2 Main Decoder truth table

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

Table 7.3 ALU Decoder truth table

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10

The PC Logic checks if the instruction is a write to R15 or a branch such that the PC should be updated. The logic is:

$$PCS = ((Rd == 15)\ \&\ RegW)\ |\ Branch$$

PCS may be killed by the Conditional Logic before it is sent to the datapath as *PCSrc*.

The Conditional Logic, shown in Figure 7.14(c), determines whether the instruction should be executed (*CondEx*) based on the *cond* field and the current values of the *N*, *Z*, *C*, and *V* flags ($Flags_{3:0}$), as was described in Table 6.3. If the instruction should not be executed, the write enables and *PCSrc* are forced to 0 so that the instruction does not change the architectural state. The Conditional Logic also updates some or all of the flags from the *ALUFlags* when *FlagW* is asserted by the ALU Decoder and the instruction's condition is satisfied (*CondEx* = 1).

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 `ORR` instruction with register addressing mode.

Solution: Figure 7.15 illustrates the control signals and flow of data during execution of the `ORR` instruction. The PC points to the memory location holding the instruction, and the instruction memory returns 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 $Instr_{19:16}$ and $Instr_{3:0}$, so *RegSrc* must be 00. *SrcB* should come from the second port of the register file (not *ExtImm*), so *ALUSrc* must be 0. The ALU performs a bitwise OR operation, so *ALUControl* must be 11. The result comes from the ALU, so *MemtoReg* is 0. The result is written to the register file, so *RegWrite* is 1. The instruction does not write memory, so *MemWrite* = 0.

The updating of *PC* with *PCPlus4* is shown with a heavy gray line. *PCSrc* is 0 to select the incremented *PC*.

Note that data certainly does flow through the nonhighlighted paths, but that the value of that data is unimportant for this instruction. For example, the immediate is extended and data is read from memory, but these values do not influence the next state of the system.

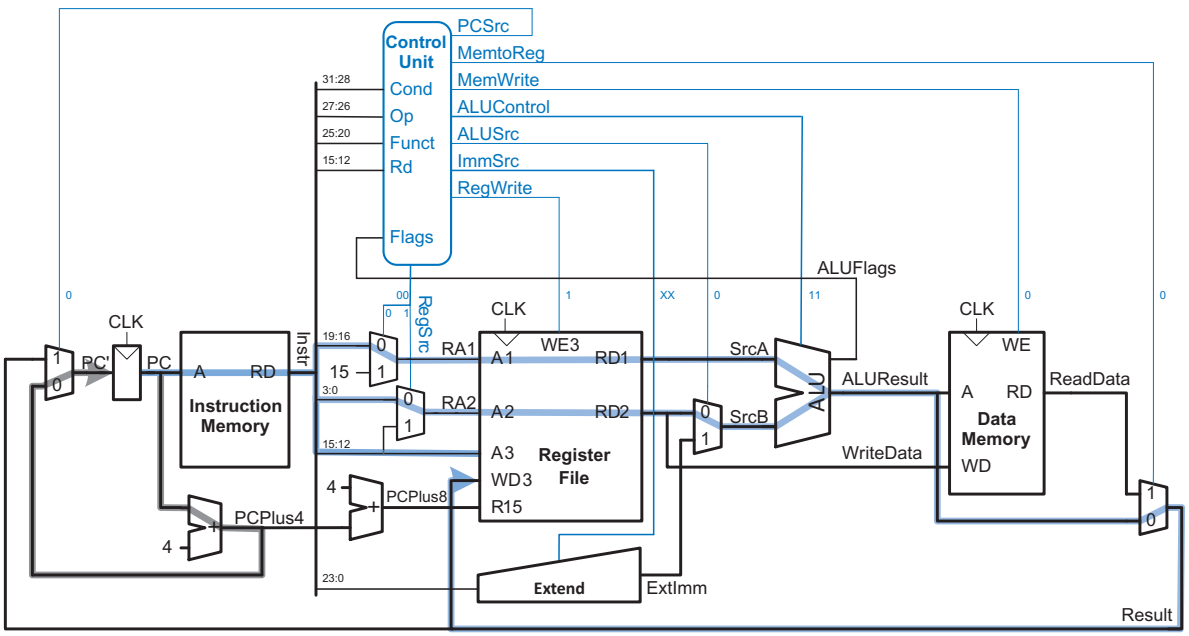


Figure 7.15 Control signals and data flow while executing an `ORR` instruction

7.3.3 More Instructions

We have considered a limited subset of the full ARM instruction set. In this section, we add support for the compare (CMP) instruction and for addressing modes in which the second source is a shifted register. These examples illustrate the principle of how to handle new instructions; with enough effort, you could extend the single-cycle processor to handle every ARM 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 CMP INSTRUCTION

The compare instruction, CMP, subtracts *SrcB* from *SrcA* and sets the flags but does not write the difference to a register. The datapath is already capable of this task. Determine the necessary changes to the controller to support CMP.

Solution: Introduce a new control signal called *NoWrite* to prevent writing *Rd* during CMP. (This signal would also be helpful for other instructions such as TST that do not write a register.) We extend the ALU Decoder to produce this signal and the *RegWrite* logic to accept it, as highlighted in blue in Figure 7.16. The enhanced ALU Decoder truth table is given in Table 7.4, with the new instruction and signal also highlighted.

Example 7.3 ENHANCED ADDRESSING MODE: REGISTERS WITH CONSTANT SHIFTS

So far, we assumed that data-processing instructions with register addressing did not shift the second source register. Enhance the single-cycle processor to support a shift by an immediate.

Solution: Insert a shifter before the ALU. Figure 7.17 shows the enhanced datapath. The shifter uses $Instr_{11:7}$ to specify the shift amount and $Instr_{6:5}$ to specify the shift type.

7.3.4 Performance Analysis

Each instruction in the single-cycle processor takes one clock cycle, so the CPI is 1. The critical paths for the LDR instruction are shown in Figure 7.18 with a heavy blue line. It starts with the PC loading a new address on the rising edge of the clock. The instruction memory reads the new instruction. The Main Decoder computes $RegSrc_0$, which drives the multiplexer to choose $Instr_{19:16}$ as *RA1*, and the register file reads this register as *SrcA*. While the register file is reading, the immediate field is zero-extended and selected at the *ALUSrc* multiplexer to determine *SrcB*. The ALU adds *SrcA* and *SrcB* to find the effective address. The data memory reads from this address. The *MemtoReg* multiplexer selects *ReadData*.

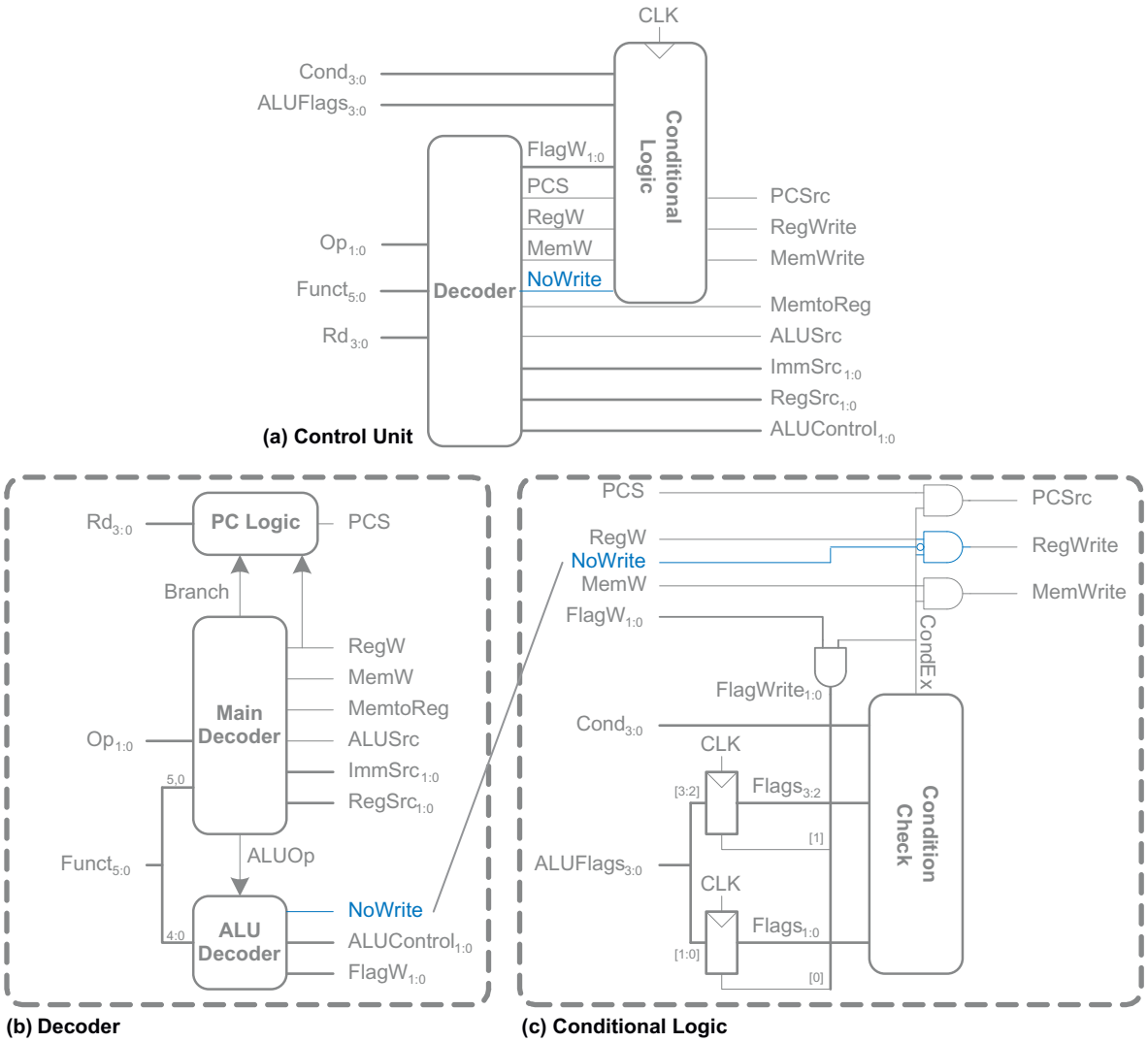


Figure 7.16 Controller modification for CMP

Finally, *Result* must set up at the register file before the next rising clock edge so that it can be properly written. Hence, the cycle time is:

$$T_{c1} = t_{pcq\_PC} + t_{mem} + t_{dec} + \max[t_{mux} + t_{RFread}, t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup} \tag{7.2}$$

We use the subscript 1 to distinguish this cycle time from that of subsequent processor designs. In most implementation technologies, the

Table 7.4 ALU Decoder truth table enhanced for CMP

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Notes	$ALUControl_{1:0}$	$FlagW_{1:0}$	*NoWrite*
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1010	1	CMP	01	11	1

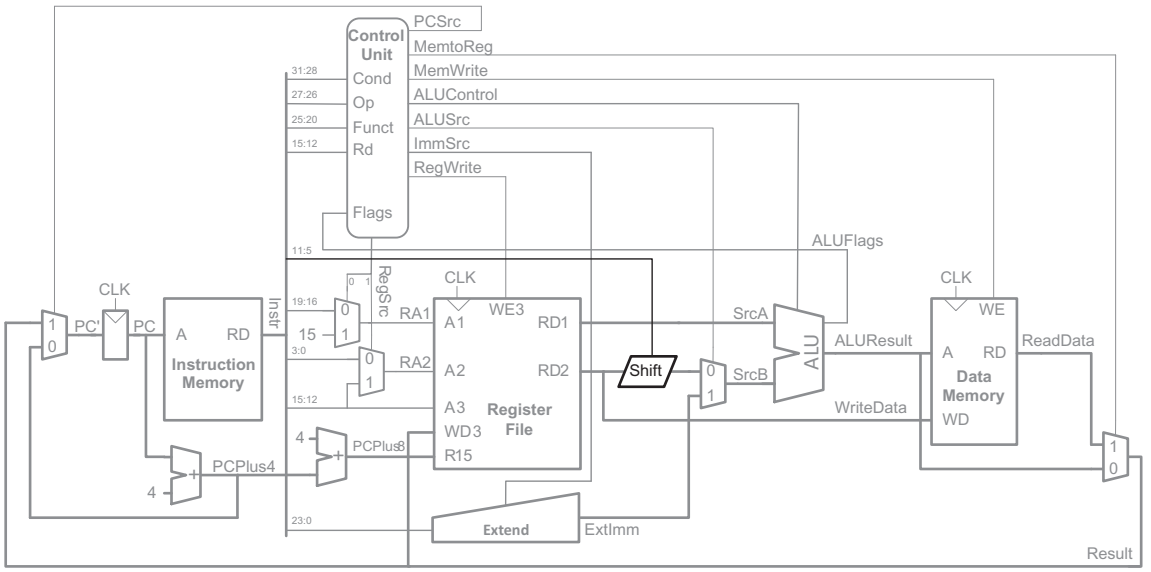


Figure 7.17 Enhanced datapath for register addressing with constant shifts

ALU, memory, and register file are substantially slower than other combinational blocks. Therefore, the cycle time simplifies to:

$$T_{c1} = t_{pcq\_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup} \quad (7.3)$$

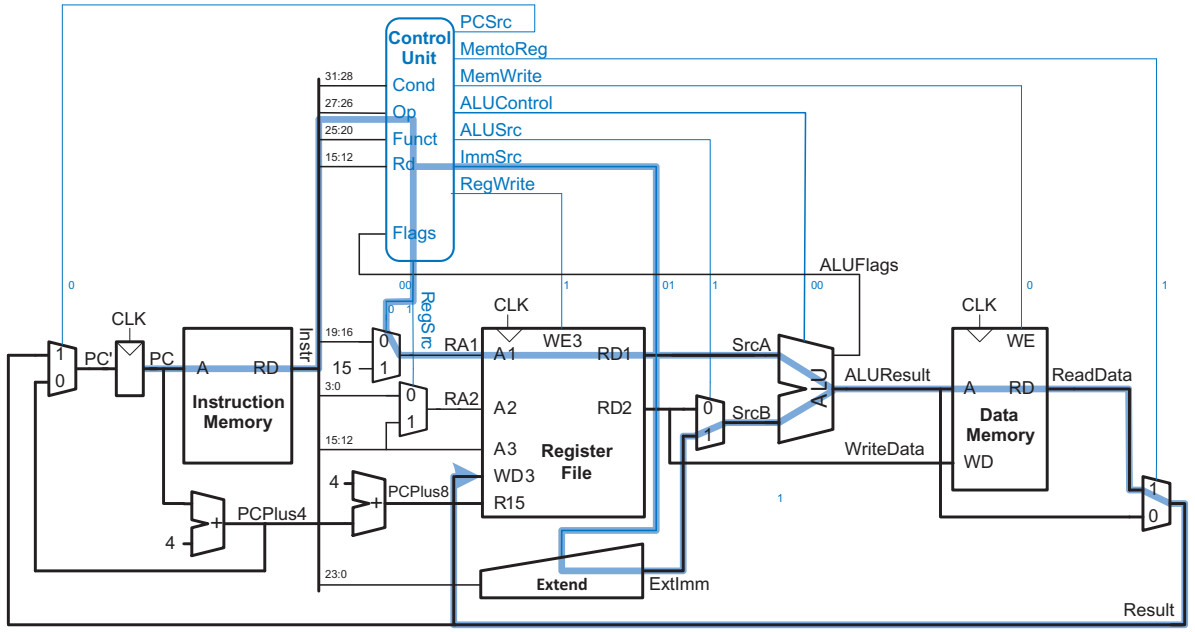


Figure 7.18 LDR **critical path**

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

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

Example 7.4 SINGLE-CYCLE PROCESSOR PERFORMANCE

Ben Bitdiddle is contemplating building the single-cycle processor in a 16-nm CMOS manufacturing process. He has determined that the logic elements have the delays given in Table 7.5. 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 $T_{c1} = 40 + 2(200) + 70 + 100 + 120 + 2(25) + 60 = 840$ ps. According to Equation 7.1, the total execution time is $T_1 = (100 \times 10^9$ instruction) (1 cycle/instruction) $(840 \times 10^{-12}$ s/cycle) = 84 seconds.

Table 7.5 Delay of circuit elements

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

7.4 MULTICYCLE PROCESSOR

The single-cycle processor has three notable weaknesses. First, it requires separate memories for instructions and data, whereas most processors only have a single external memory holding both instructions and data. Second, it requires a clock cycle long enough to support the slowest instruction (`LDR`), 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. In each short step, the processor can read or write the memory or register file or use the ALU. The instruction is read in one step and data can be read or written in a later step, so the processor can use a single memory for both. Different instructions use different numbers of steps, so simpler instructions can complete faster than more complex ones. And the processor needs only one adder, which is reused for different purposes on different steps.

We design a multicycle processor following the same procedure we used for the single-cycle processor. First, we construct a datapath by connecting the architectural state elements and memories with combinational logic. But, this time, we also add nonarchitectural state elements to hold intermediate results between the steps. Then, we design the controller. The controller produces different signals on different steps during execution of a single instruction, so now it is 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 7.19. In the single-cycle design, we used separate instruction and data memories because we needed to read the instruction memory and read or write the data memory all in one cycle. Now, we choose to use a combined memory for both instructions and data. This is more realistic, and it is feasible because we can read the instruction in one cycle, then read or write the data in a separate cycle. The PC and register file remain unchanged. 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 instruction memory. Figure 7.20 shows that the PC is 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.

LDR

As we did with the single-cycle processor, we first work out the datapath connections for the `LDR` instruction. After fetching `LDR`, the next step is

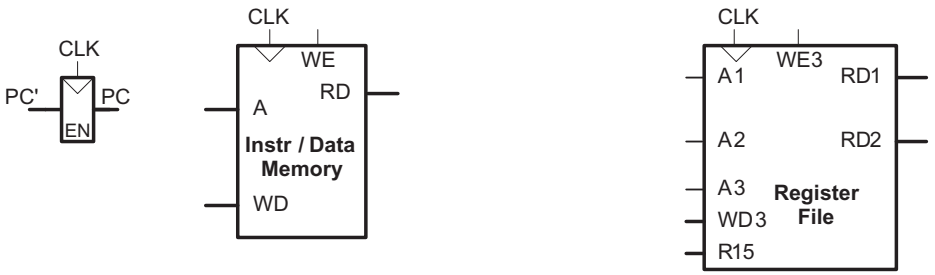


Figure 7.19 State elements with unified instruction/data memory

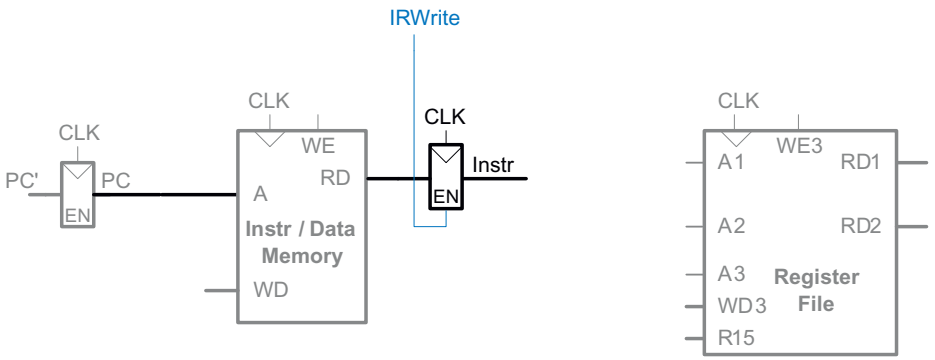


Figure 7.20 Fetch instruction from memory

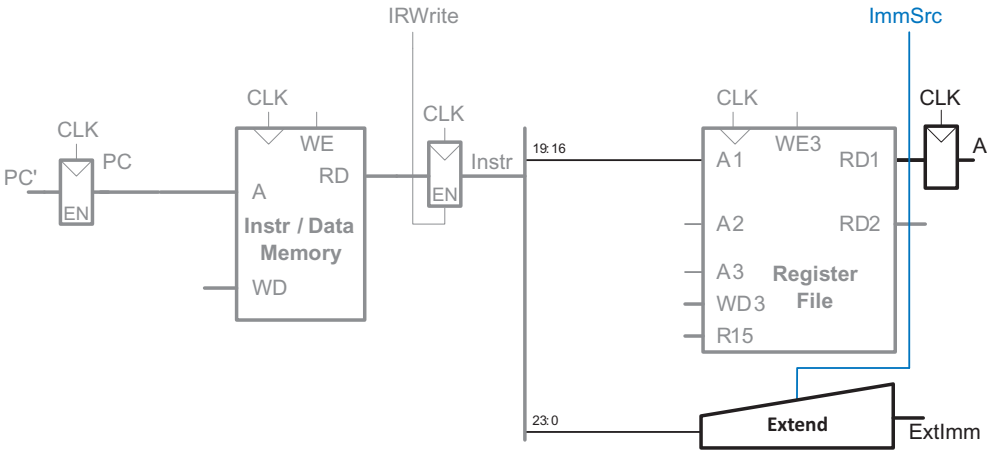


Figure 7.21 Read one source from register file and extend the second source from the immediate field

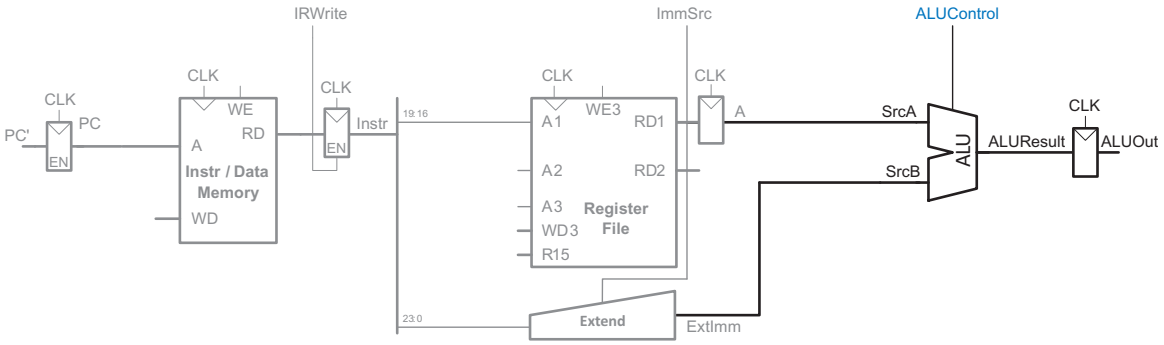


Figure 7.22 Add base address to offset

to read the source register containing the base address. This register is specified in the *Rn* field, $Instr_{19:16}$. These bits of the instruction are connected to address input *A1* of the register file, as shown in Figure 7.21. The register file reads the register into *RD1*. This value is stored in another nonarchitectural register, *A*.

The `LDR` instruction also requires a 12-bit offset, found in the immediate field of the instruction, $Instr_{11:0}$, which must be zero-extended to 32 bits, as shown in Figure 7.21. As in the single-cycle processor, the Extend block takes an *ImmSrc* control signal to specify an 8-, 12-, or 24-bit immediate to extend for various types of instructions. The 32-bit extended immediate is called *ExtImm*. To be consistent, we might store *ExtImm* in another nonarchitectural register. However, *ExtImm* is a combinational function of *Instr* and will not change while the current instruction is being processed, so there is no need to dedicate a register to hold the constant value.

The address of the load is the sum of the base address and offset. We use an ALU to compute this sum, as shown in Figure 7.22. *ALUControl*

should be set to 00 to perform the addition. *ALUResult* is stored in a nonarchitectural register called *ALUOut*.

The next step is to load the data from the calculated address in the memory. We add a multiplexer in front of the memory to choose the memory address, *Adr*, from either the PC or *ALUOut* based on the *AdrSrc* select, as shown in Figure 7.23. The data read from memory is stored in another nonarchitectural register, called *Data*. Note that the address multiplexer permits us to reuse the memory during the `LDR` instruction. On a first step, the address is taken from the PC to fetch the instruction. On a later step, the address is taken from *ALUOut* to load the data. Hence, *AdrSrc* must have different values on different steps. In Section 7.4.2, we develop the FSM controller that generates these sequences of control signals.

Finally, the data is written back to the register file, as shown in Figure 7.24. The destination register is specified by the *Rd* field of the instruction, $Instr_{15:12}$. The result comes from the *Data* register. Instead of

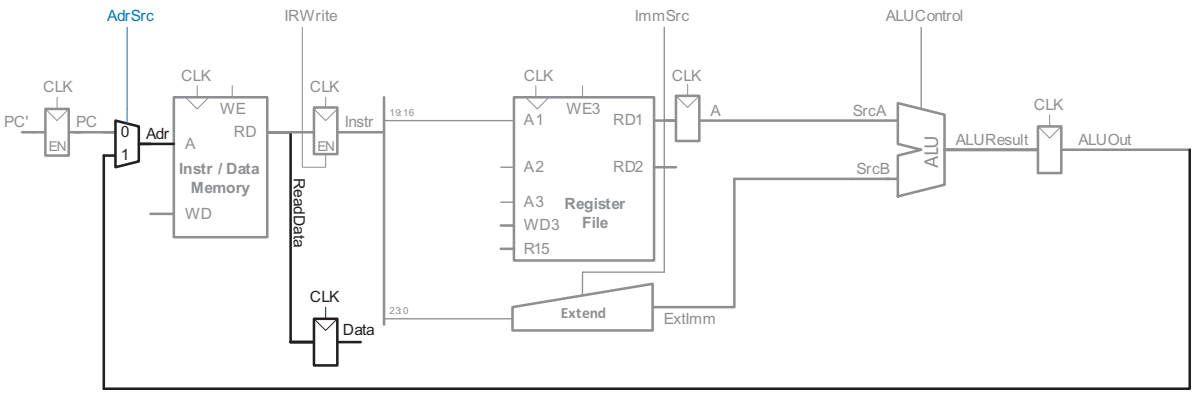


Figure 7.23 Load data from memory

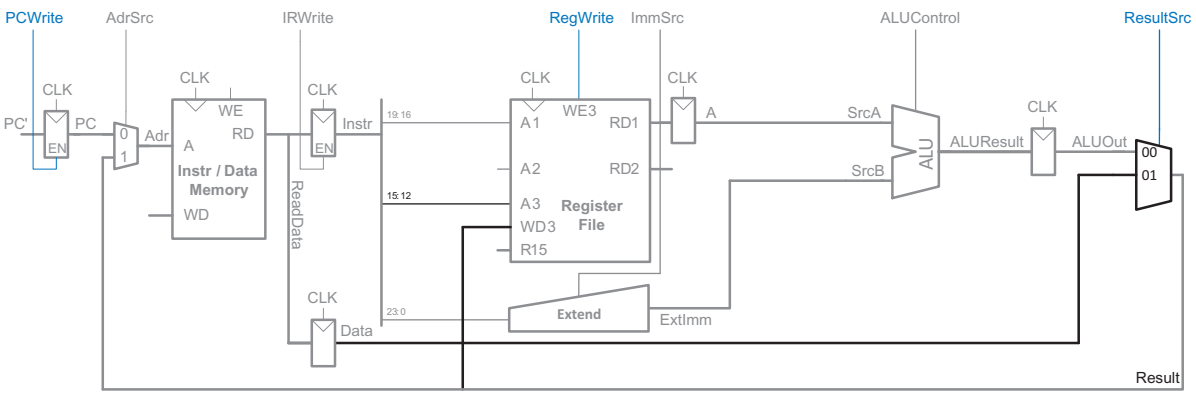


Figure 7.24 Write data back to register file

connecting the *Data* register directly to the register file *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 write port. This will be helpful because other instructions will need to write a result from the ALU. 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 old *PC*. In the single-cycle processor, a separate adder was needed. In the multicycle processor, we can use the existing ALU during the 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 7.25. A multiplexer controlled by *ALUSrcA* chooses either *PC* or register *A* as *SrcA*. Another multiplexer chooses either 4 or *ExtImm* as *SrcB*. To update the PC, the ALU adds *SrcA* (*PC*) to *SrcB* (4), and the result is written into the program counter. The *ResultSrc* multiplexer chooses this sum from *ALUResult* rather than *ALUOut*; this requires a third input. The *PCWrite* control signal enables the PC to be written only on certain cycles.

Again, we face the ARM architecture idiosyncrasy that reading R15 returns *PC* + 8 and writing R15 updates the PC. First, consider R15 reads. We already computed *PC* + 4 during the fetch step, and the sum is available in the PC register. Thus, during the second step, we obtain *PC* + 8 by adding four to the updated PC using the ALU. *ALUResult* is selected as the *Result* and fed to the R15 input port of the register file. Figure 7.26 shows the completed `LDR` datapath with this new connection. Thus, a read of R15, which also occurs during the second step, produces the value *PC* + 8 on the read data output of the register file. Writes to R15 require writing the PC register instead of the register file. Thus, in the final step of the instruction, *Result* must be routed to the PC register (instead of to the register file) and *PCWrite* must be asserted (instead of *RegWrite*). The datapath already accommodates this, so no datapath changes are required.

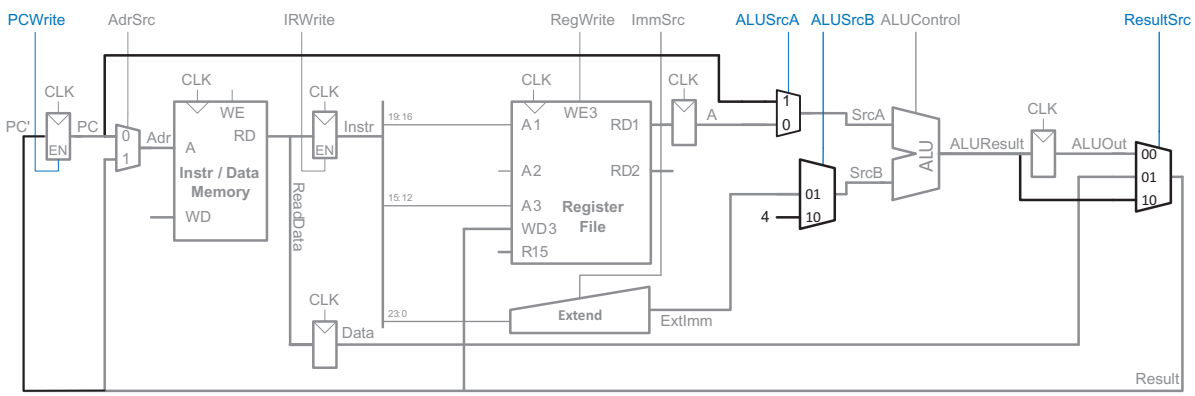


Figure 7.25 Increment PC by 4

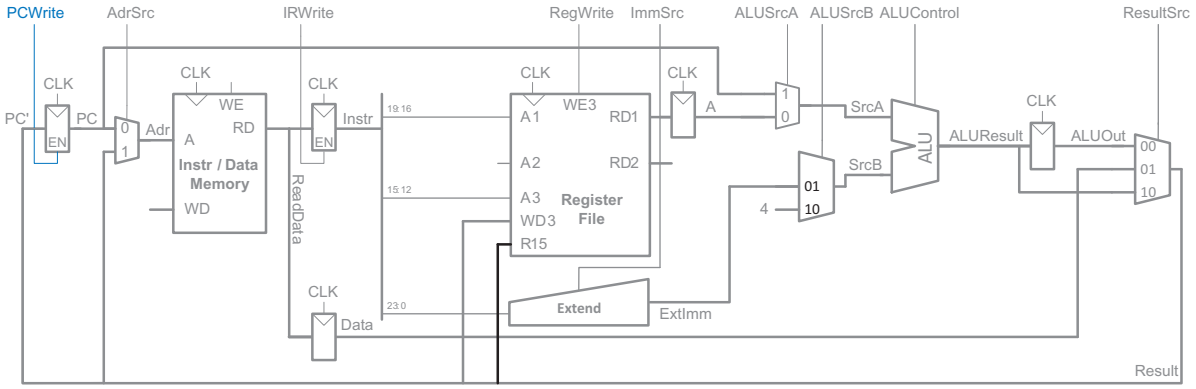


Figure 7.26 **Handle R15 reads and writes**

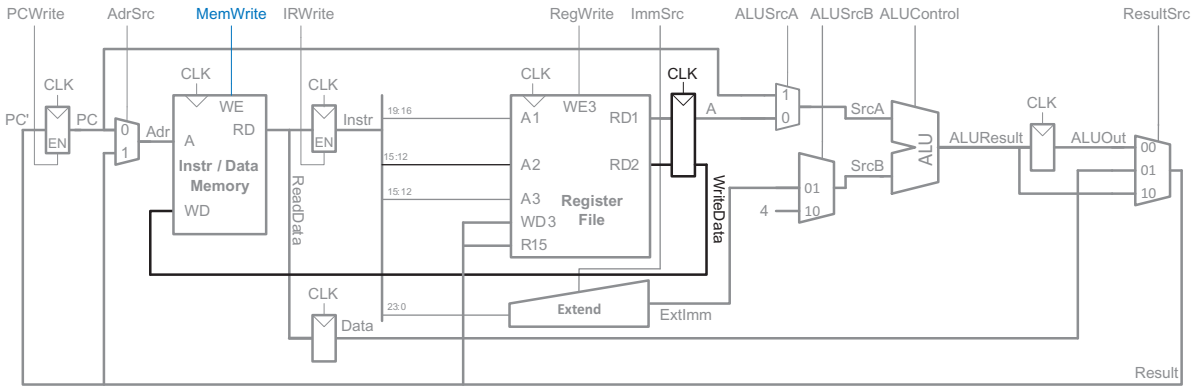


Figure 7.27 **Enhanced datapath for `STR` instruction**

STR

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

The only new feature of `STR` is that we must read a second register from the register file and write it into the memory, as shown in Figure 7.27. The register is specified in the *Rd* field of the instruction, $Instr_{15:12}$, which is connected to the second port of the register file. When the register is read, it is stored in a nonarchitectural register, *WriteData*. On the next step, it is sent to the write data port (*WD*) of the data memory to be written.

The memory receives the *MemWrite* control signal to indicate that the write should occur.

Data-Processing Instructions with Immediate Addressing

Data-processing instructions with immediate addressing read the first source from *Rn* and extend the second source from an 8-bit immediate. They operate on these two sources and then write the result back to the register file. The datapath already contains all the connections necessary for these steps. The ALU uses the *ALUControl* signal to determine the type of data-processing instruction to execute. The *ALUFlags* are sent back to the controller to update the *Status* register.

Data-Processing Instructions with Register Addressing

Data-processing instructions with register addressing select the second source from the register file. The register is specified in the *Rm* field, $Instr_{3:0}$, so we insert a multiplexer to choose this field as *RA2* for the register file. We also extend the *SrcB* multiplexer to accept the value read from the register file, as shown in Figure 7.28. Otherwise, the behavior is the same as for data-processing instructions with immediate addressing.

B

The branch instruction `B` reads $PC + 8$ and a 24-bit immediate, sums them, and adds the result to the PC. Recall from Section 6.4.6 that a read to R15 returns $PC + 8$, so we add a multiplexer to choose R15 as *RA1* for the register file, as shown in Figure 7.29. The rest of the hardware to perform the addition and write the PC is already present in the datapath.

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

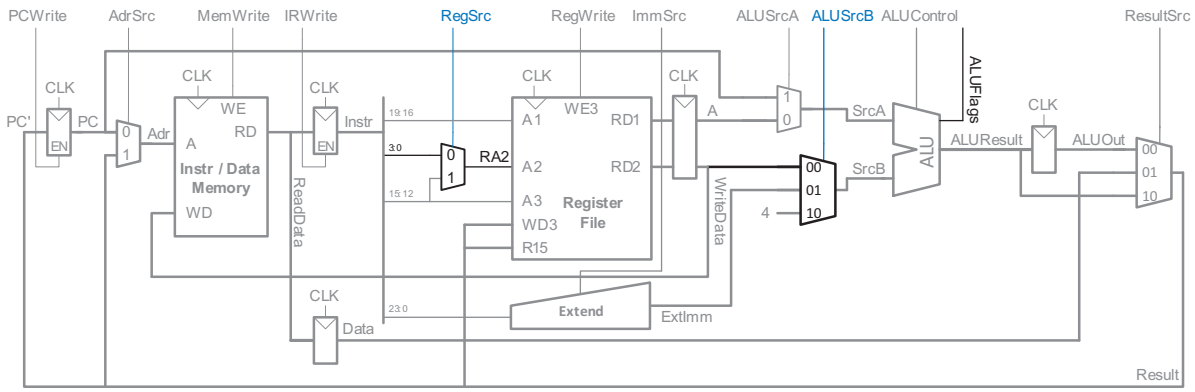


Figure 7.28 Enhanced datapath for data-processing instructions with register addressing

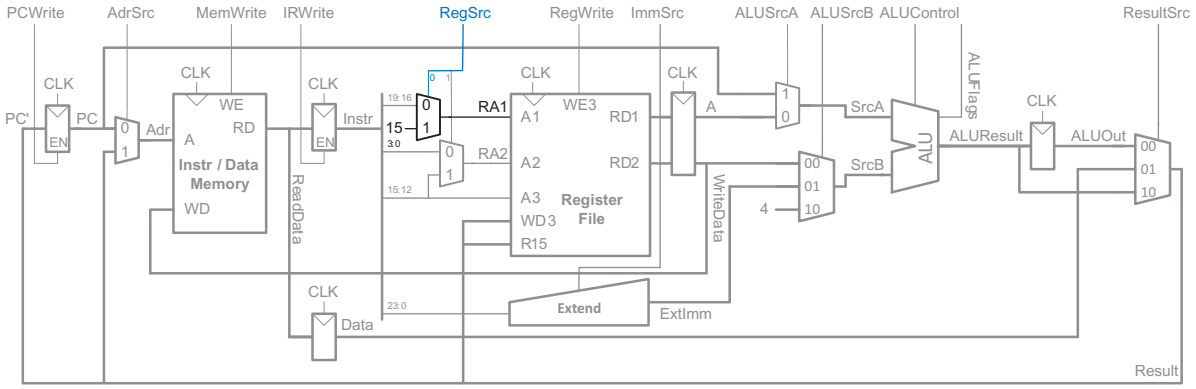


Figure 7.29 Enhanced datapath for the B instruction

of each step. In this way, the memory can be shared for instructions and data and the ALU can be reused several times, 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 *cond*, *op*, and *funct* fields of the instruction ($Instr_{31:28}$, $Instr_{27:26}$, and $Instr_{25:20}$) as well as the flags and whether the destination register is the PC. The controller also stores the current status flags and updates them appropriately. Figure 7.30 shows the entire multicycle processor with the control unit attached to the datapath. The datapath is shown in black and the control unit is shown in blue.

As in the single-cycle processor, the control unit is partitioned into Decoder and Conditional Logic blocks, as shown in Figure 7.31(a). The Decoder is decomposed further in Figure 7.31(b). The combinational Main Decoder of the single-cycle processor is replaced with a Main FSM in the multicycle processor to produce a sequence of control signals on the appropriate cycles. We design the Main FSM as a Moore machine so that the outputs are only a function of the current state. However, we will see during the state machine design that *ImmSrc* and *RegSrc* are a function of *Op* rather than the current state, so we also use a small Instruction Decoder to compute these signals, as will be described in Table 7.6. The ALU Decoder and PC Logic are identical to those in the single-cycle processor. The Conditional Logic is almost identical to that of the single-cycle processor. We add a *NextPC* signal to force a write to the PC when we compute $PC + 4$. We also delay *CondEx* by one cycle

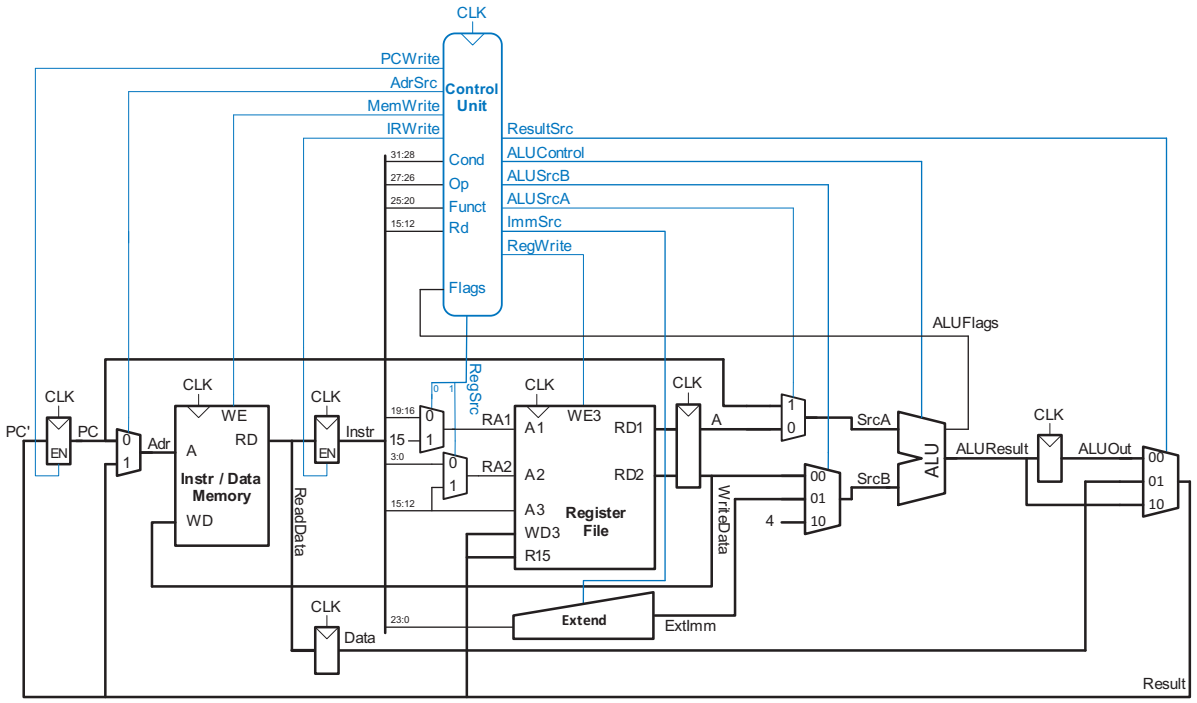


Figure 7.30 Complete multicycle processor

before sending it to *PCWrite*, *RegWrite*, and *MemWrite* so that updated condition flags are not seen until the end of an instruction. 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. Select signals are listed only when their value matters; otherwise, they are don't care. Enable signals (*RegW*, *MemW*, *IRWrite*, and *NextPC*) are listed only when they are asserted; otherwise, they are 0.

The first step for any instruction is to fetch the instruction from memory at the address held in the PC and to increment the PC to the next instruction. The FSM enters this Fetch state on reset. The control signals are shown in Figure 7.32. The data flow on this step is shown in Figure 7.33, with the instruction fetch highlighted in blue and the PC increment highlighted in gray. To read memory, $AdrSrc = 0$, so the address is taken from the PC. *IRWrite* is asserted to write the instruction into the instruction register, *IR*. Meanwhile, the PC should be incremented by 4 to point to the next instruction. Because the ALU is not being used for anything else, the processor can

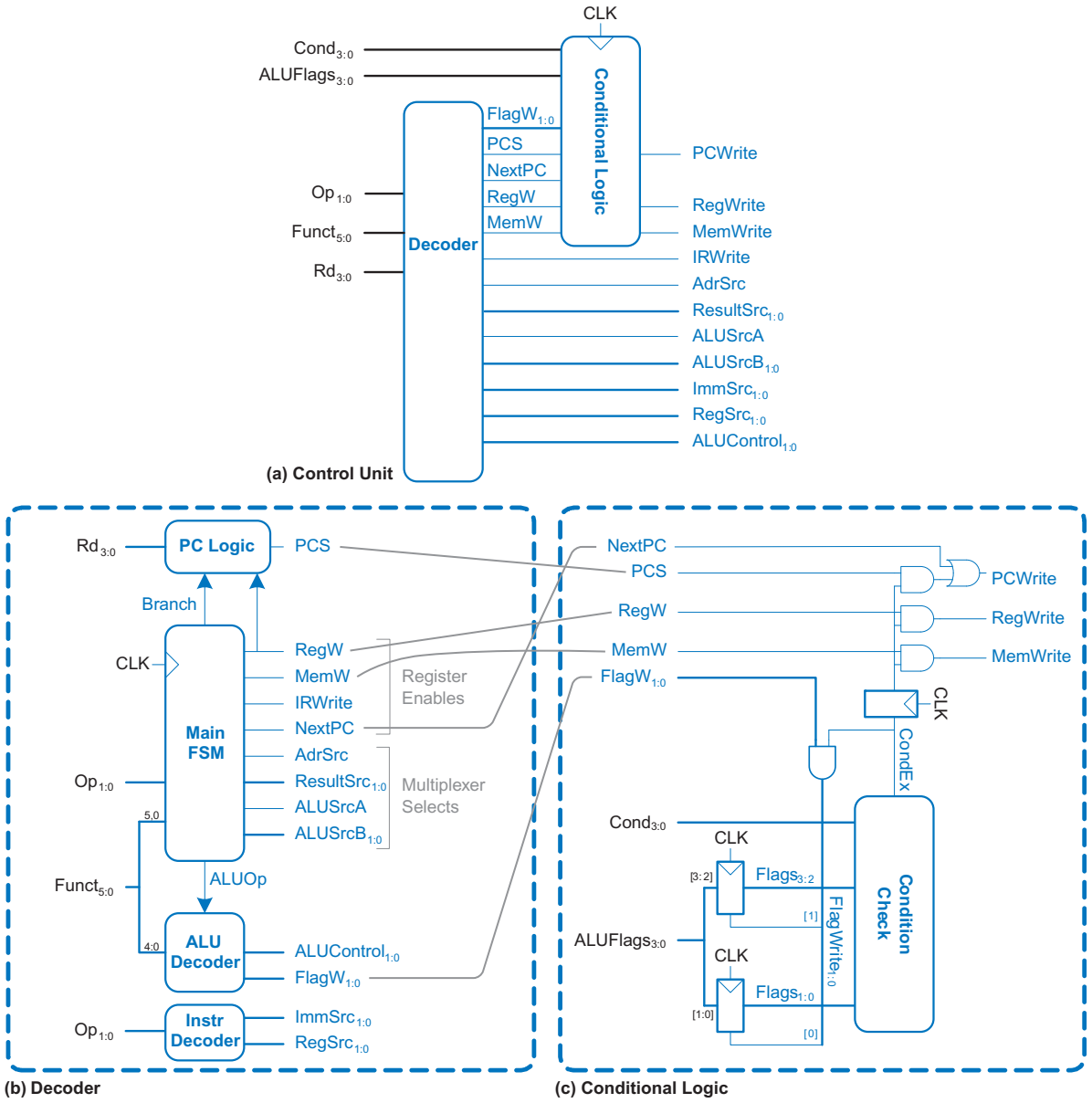


Figure 7.31 Multicycle control unit

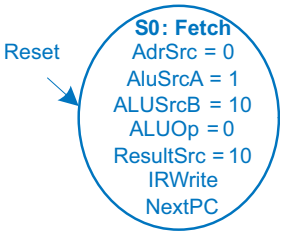


Figure 7.32 Fetch

Table 7.6 Instr Decoder logic for *RegSrc* and *ImmSrc*

Instruction	*Op*	$Funct_5$	$Funct_0$	$RegSrc_1$	$RegSrc_0$	$ImmSrc_{1:0}$
LDR	01	X	1	X	0	01
STR	01	X	0	1	0	01
DP immediate	00	1	X	X	0	00
DP register	00	0	X	0	0	00
B	10	X	X	X	1	10

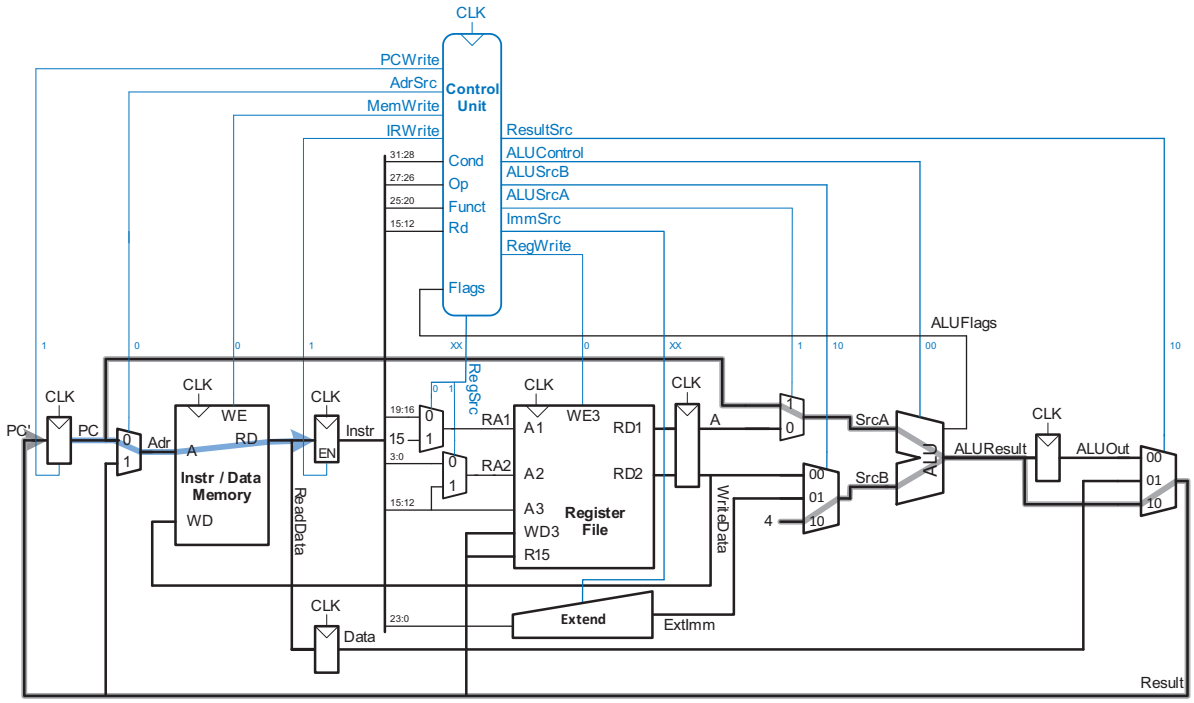


Figure 7.33 Data flow during the fetch step

use it to compute $PC + 4$ at the same time that it fetches the instruction. $ALUSrcA = 1$, so $SrcA$ comes from the PC. $ALUSrcB = 10$, so $SrcB$ is the constant 4. $ALUOp = 0$, so the ALU produces $ALUControl = 00$ to make the ALU add. To update the PC with $PC + 4$, $ResultSrc = 10$ to choose the $ALUResult$ and $NextPC = 1$ to enable $PCWrite$.

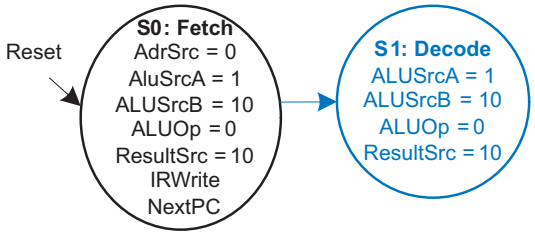


Figure 7.34 Decode

The second step is to read the register file and/or immediate and decode the instructions. The registers and immediate are selected based on *RegSrc* and *ImmSrc*, which are computed by the Instr Decoder based on *Instr*. $RegSrc_0$ should be 1 for branches to read $PC + 8$ as *SrcA*. $RegSrc_1$ should be 1 for stores to read the store value as *SrcB*. *ImmSrc* should be 00 for data-processing instructions to select an 8-bit immediate, 01 for loads and stores to select a 12-bit immediate, and 10 for branches to select a 24-bit immediate. Because the multicycle FSM is a Moore machine whose outputs depend only on the current state, the FSM cannot directly produce these selects that depend on *Instr*. The FSM could be organized as a Mealy machine whose outputs depend on *Instr* as well as the state, but this would be messy. Instead, we choose the simplest solution, which is to make these selects combinational functions of *Instr*, as given in Table 7.6. Taking advantage of don't cares, the Instr Decoder logic can be simplified to:

$$RegSrc_1 = (Op == 01)$$

$$RegSrc_0 = (Op == 10)$$

$$ImmSrc_{1:0} = Op$$

Meanwhile, the ALU is reused to compute $PC + 8$ by adding 4 more to the PC that was incremented in the Fetch step. Control signals are applied to select *PC* as the first ALU input ($ALUSrcA = 1$) and 4 as the second input ($ALUSrcB = 10$) and to perform addition ($ALUOp = 0$). This sum is selected as the *Result* ($ResultSrc = 10$) and provided to the *R15* input of the register file so that R15 reads as $PC + 8$. The FSM Decode step is shown in Figure 7.34 and the data flow is shown in Figure 7.35, highlighting the R15 computation and the register file read.

Now the FSM proceeds to one of several possible states, depending on *Op* and *Funct* that are examined during the Decode step. If the instruction is a memory load or store (`LDR` or `STR`, $Op = 01$), then the multicycle processor computes the address by adding the base address to the zero-extended offset. This requires $ALUSrcA = 0$ to select the base address from the register file and $ALUSrcB = 01$ to select *ExtImm*. $ALUOp = 0$ so the ALU adds. The effective address is stored in the

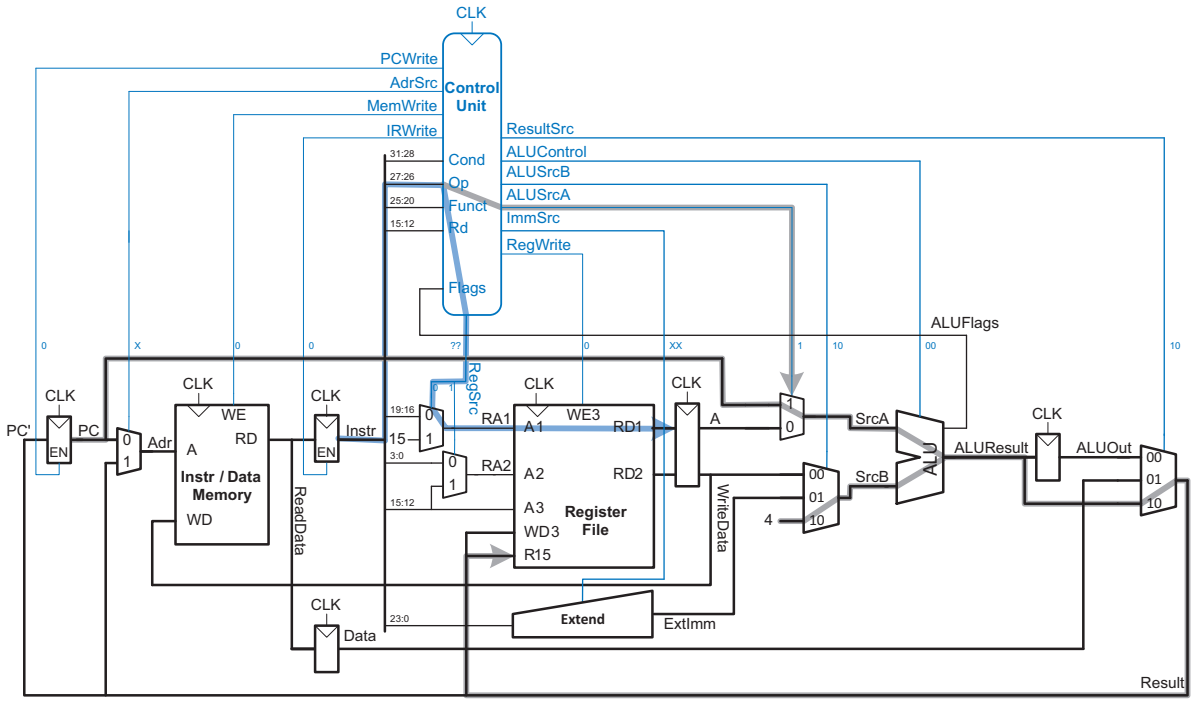


Figure 7.35 Data flow during the Decode step

ALUOut register for use on the next step. The FSM MemAdr state is shown in Figure 7.36 and the data flow is highlighted in Figure 7.37.

If the instruction is LDR ($Funct_0 = 1$), then the multicycle processor must next read data from the memory and write it to the register file. These two steps are shown in Figure 7.38. To read from the memory, $ResultSrc = 00$ and $AdrSrc = 1$ to select the memory address that was just computed and saved in *ALUOut*. This address in memory is read and saved in the *Data* register during the MemRead step. Then, in the memory writeback step MemWB, *Data* is written to the register file. $ResultSrc = 01$ to choose *Result* from *Data* and *RegW* is asserted to write the register file, completing the LDR instruction. Finally, the FSM returns to the Fetch state to start the next instruction. For these and subsequent steps, try to visualize the data flow on your own.

From the MemAdr state, if the instruction is STR ($Funct_0 = 0$), the data read from the second port of the register file is simply written to memory. In this MemWrite state, $ResultSrc = 00$ and $AdrSrc = 1$ to select the address computed in the MemAdr state and saved in *ALUOut*. *MemW* is asserted to write the memory. Again, the FSM returns to the Fetch state. The state is shown in Figure 7.39.

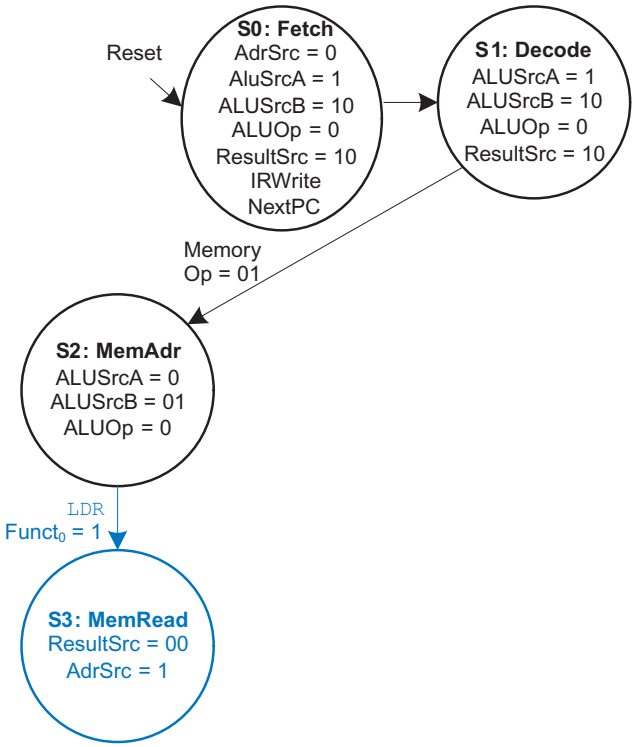


Figure 7.36 Memory address computation

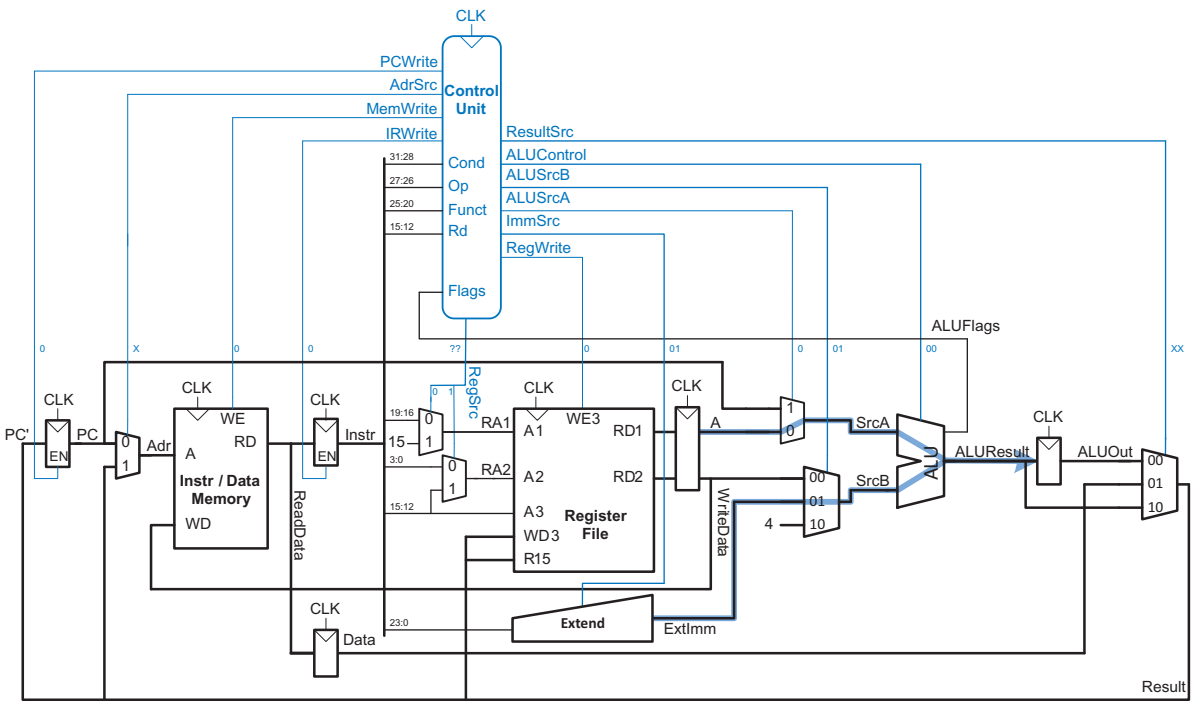


Figure 7.37 Data flow during memory address computation

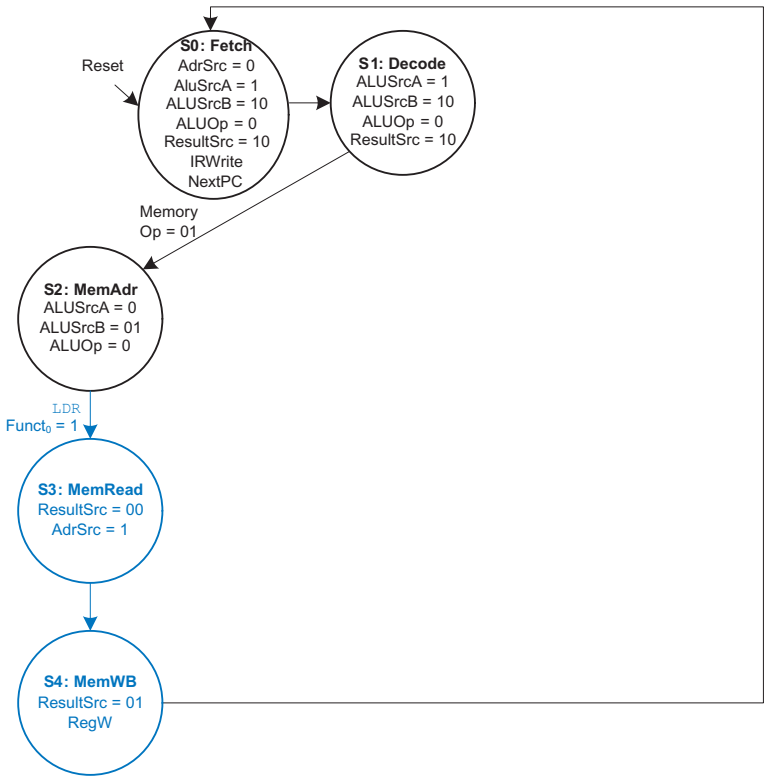


Figure 7.38 Memory read

For data-processing instructions ($Op = 00$), the multicycle processor must calculate the result using the ALU and write that result to the register file. The first source always comes from the register ($ALUSrcA = 0$). $ALUOp = 1$ so the ALU Decoder chooses the appropriate *ALUControl* for the specific instruction based on *cmd* ($Funct_{4:1}$). The second source comes from the register file for register instructions ($ALUSrcB = 00$) or from *ExtImm* for immediate instructions ($ALUSrcB = 01$). Thus, the FSM needs ExecuteR and ExecuteI states to cover these two possibilities. In either case, the data-processing instruction advances to the ALU Writeback state (ALUWB), in which the result is selected from *ALUOut* ($ResultSrc = 00$) and written to the register file ($RegW = 1$). All of these states are shown in Figure 7.40.

For a branch instruction, the processor must calculate the destination address ($PC + 8 +$ offset) and write it to the PC. During the Decode state, $PC + 8$ was already computed and read from the register file onto *RD1*. Therefore, during the Branch state, the controller uses $ALUSrcA = 0$ to choose R15 ($PC + 8$), $ALUSrcB = 01$ to choose *ExtImm*, and $ALUOp = 0$

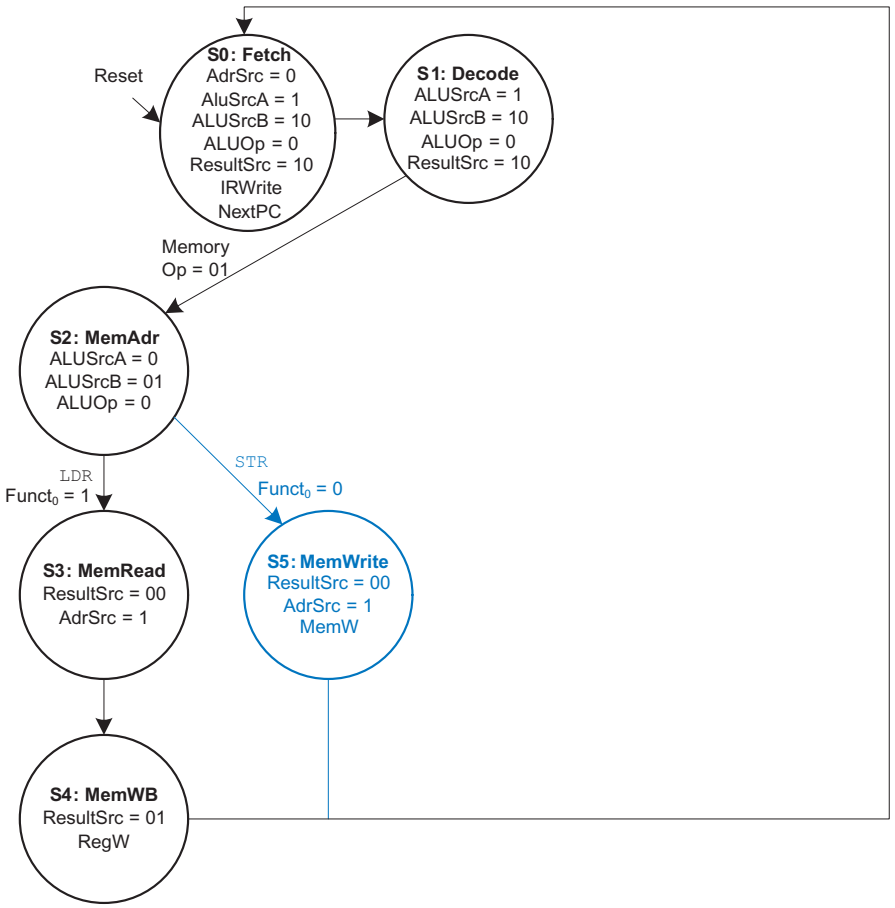


Figure 7.39 Memory write

to add. The *Result* multiplexer chooses *ALUResult* (*ResultSrc* = 10). *Branch* is asserted to write the result to the PC.

Putting these steps together, Figure 7.41 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.

7.4.3 Performance Analysis

The execution time of an instruction depends on both the number of cycles it uses and the cycle time. Whereas the single-cycle processor

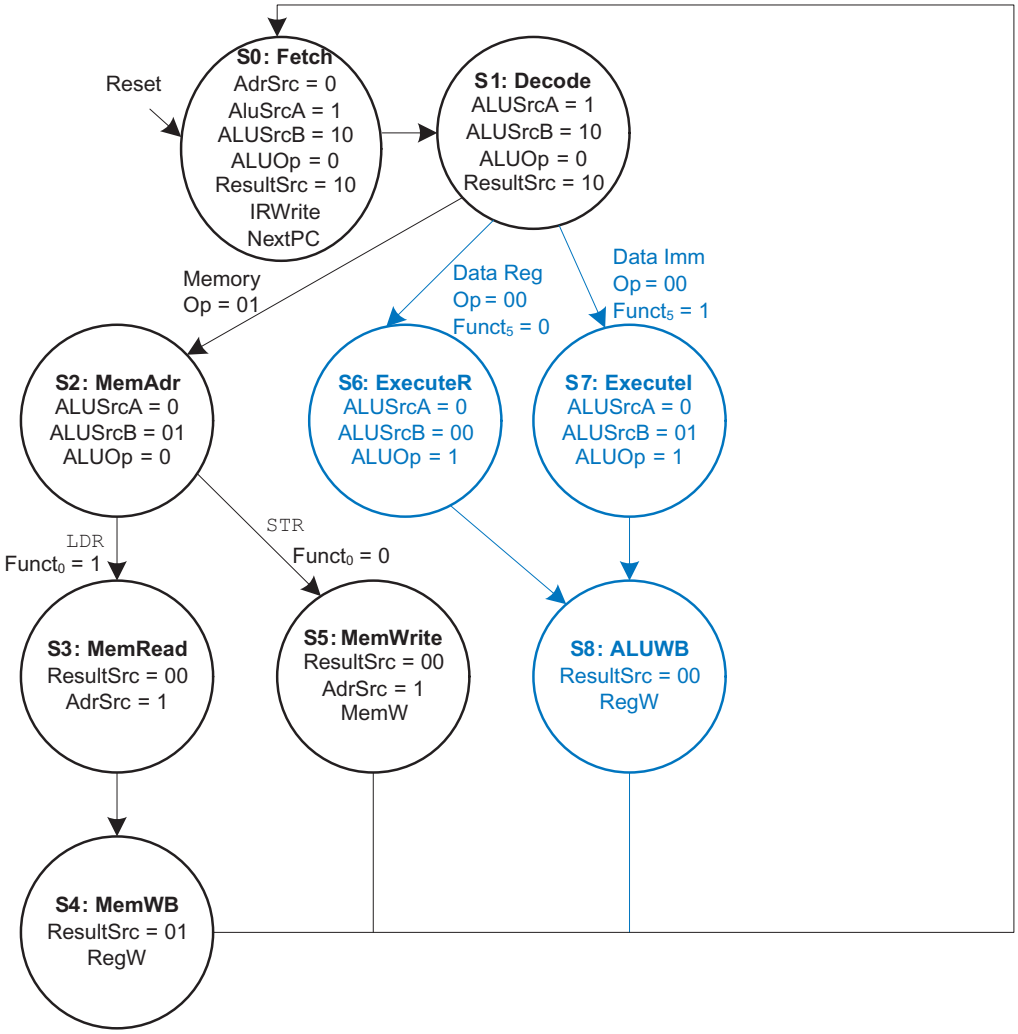
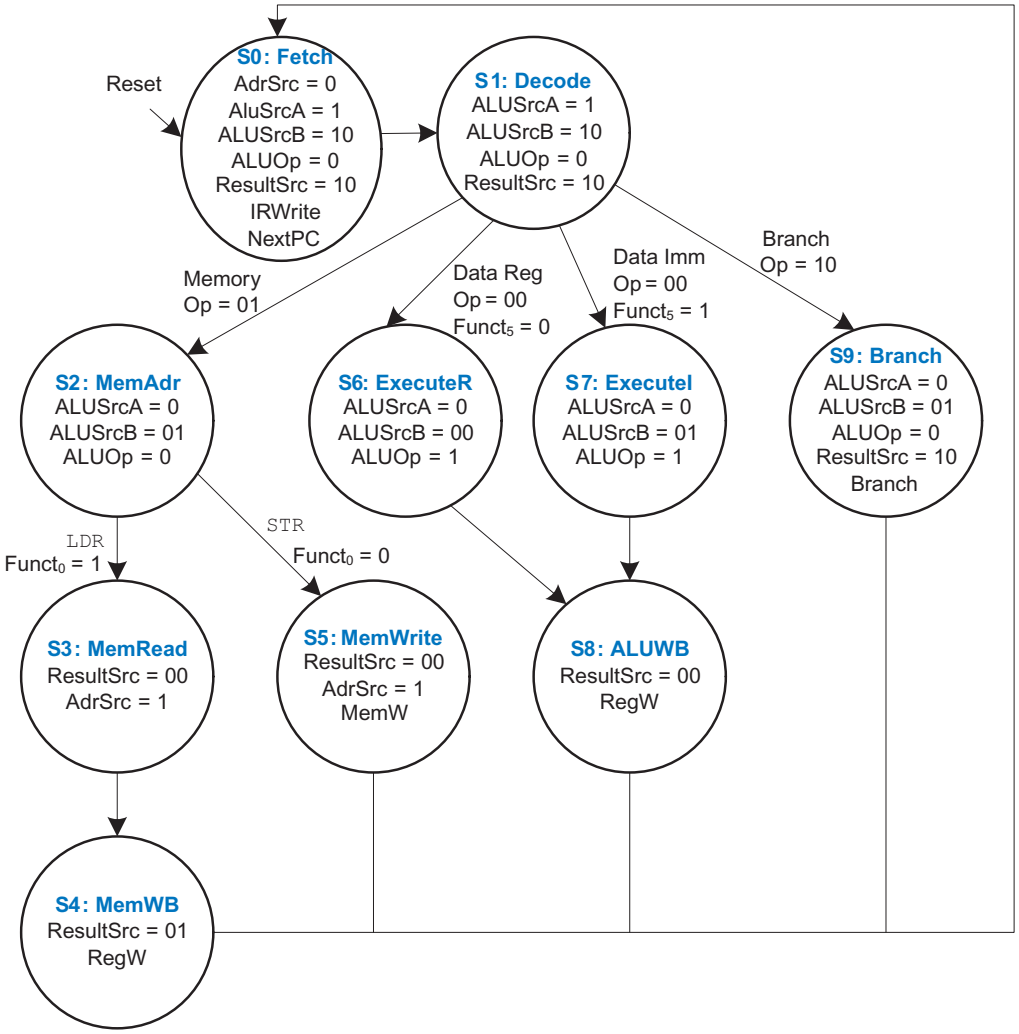


Figure 7.40 Data-processing

performed all instructions in one cycle, the multicycle processor uses varying numbers of cycles for the various instructions. However, the multicycle processor does less work in a single cycle and, thus, has a shorter cycle time.

The multicycle processor requires three cycles for branches, four for data-processing instructions and stores, and five for loads. The CPI depends on the relative likelihood that each instruction is used.



State	Datapath μOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PC +4
MemAdr	ALUOut ← Rn + Imm
MemRead	Data ← Mem[ALUOut]
MemWB	Rd ← Data
MemWrite	Mem[ALUOut] ← Rd
ExecuteR	ALUOut ← Rn op Rm
ExecuteI	ALUOut ← Rn op Imm
ALUWB	Rd ← ALUOut
Branch	PC ← R15 + offset

Figure 7.41 Complete multicycle control FSM

Example 7.5 MULTICYCLE PROCESSOR CPI

The SPECINT2000 benchmark consists of approximately 25% loads, 10% stores, 13% branches, and 52% data-processing instructions.[2] Determine the average CPI for this benchmark.

Solution: The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of the time that instruction is used. For this benchmark, average CPI = (0.13)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12. This is better than the worst-case CPI of 5, which would be required if all instructions took the same time.

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:

1. From the PC through the *SrcA* multiplexer, ALU, and result multiplexer to the *R15* port of the register file to the *A* register
2. From *ALUOut* through the *Result* and *Adr* muxes to read memory into the *Data* register

$$T_{c2} = t_{pcq} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup} \tag{7.4}$$

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

Example 7.6 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 16-nm CMOS manufacturing process with the delays given in Table 7.5. Help him compare each processor's execution time for 100 billion instructions from the SPECINT2000 benchmark (see Example 7.5).

Solution: According to Equation 7.4, the cycle time of the multicycle processor is $T_{c2} = 40 + 2(25) + 200 + 50 = 340$ ps. Using the CPI of 4.12 from Example 7.5, the total execution time is $T_2 = (100 \times 10^9$ instructions)(4.12 cycles/instruction)$(340 \times 10^{-12}$ s/cycle) = 140 seconds. According to Example 7.4, the single-cycle processor had a total execution time of 84 seconds.

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

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, `LDR`, was broken into five steps, the multicycle processor cycle time was not nearly improved five-fold. 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 almost five times faster. Hence, the latency of each instruction is ideally unchanged, but the throughput is ideally five-times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. Pipelining introduces some overhead, so the throughput will not be quite as high as we might ideally desire, but pipelining nevertheless gives such great advantage for so little cost that all modern high-performance microprocessors are pipelined.

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

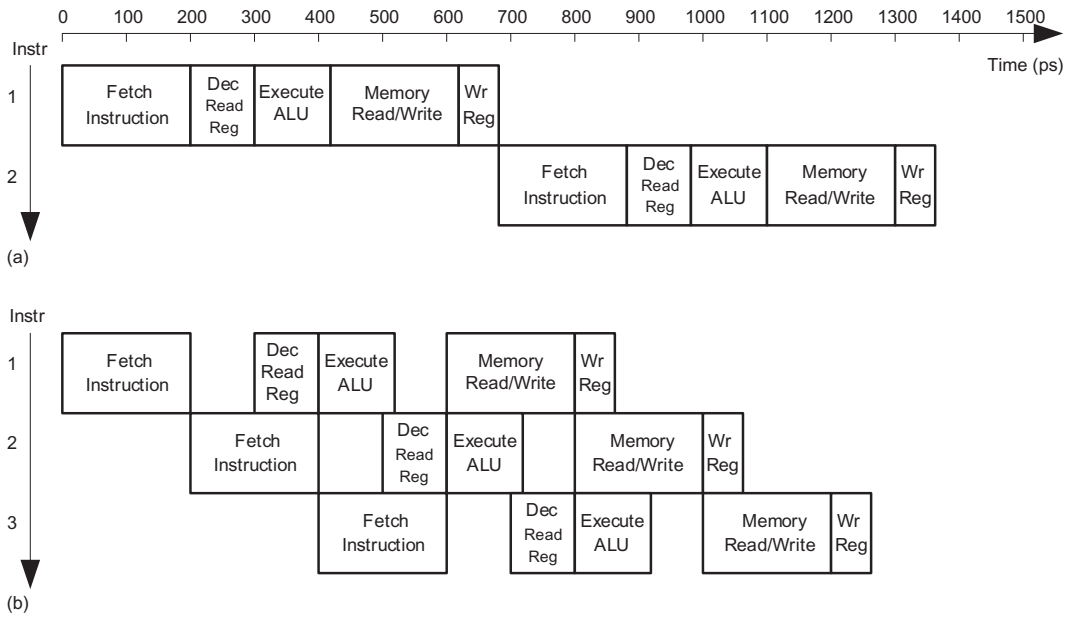


Figure 7.42 **Timing diagrams: (a) single-cycle processor and (b) pipelined processor**

Figure 7.42 shows a timing diagram comparing the single-cycle and pipelined processors. Time is on the horizontal axis, and instructions are on the vertical axis. The diagram assumes the logic element delays from Table 7.5 but ignores the delays of multiplexers and registers. In the single-cycle processor (Figure 7.42(a)), the first instruction is read from memory at time 0; next, the operands are read from the register file; and, then, the ALU executes the necessary computation. Finally, the data memory may be accessed, and the result is written back to the register file by 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 and a throughput of 1 instruction per 680 ps (1.47 billion instructions per second).

In the pipelined processor (Figure 7.42(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). 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. And so forth, until all the instructions complete. The instruction latency is $5 \times 200 = 1000$ ps. The throughput is 1 instruction per 200 ps (5 billion instructions per second). Because the stages are not perfectly balanced with equal amounts of logic, the latency is longer for the pipelined

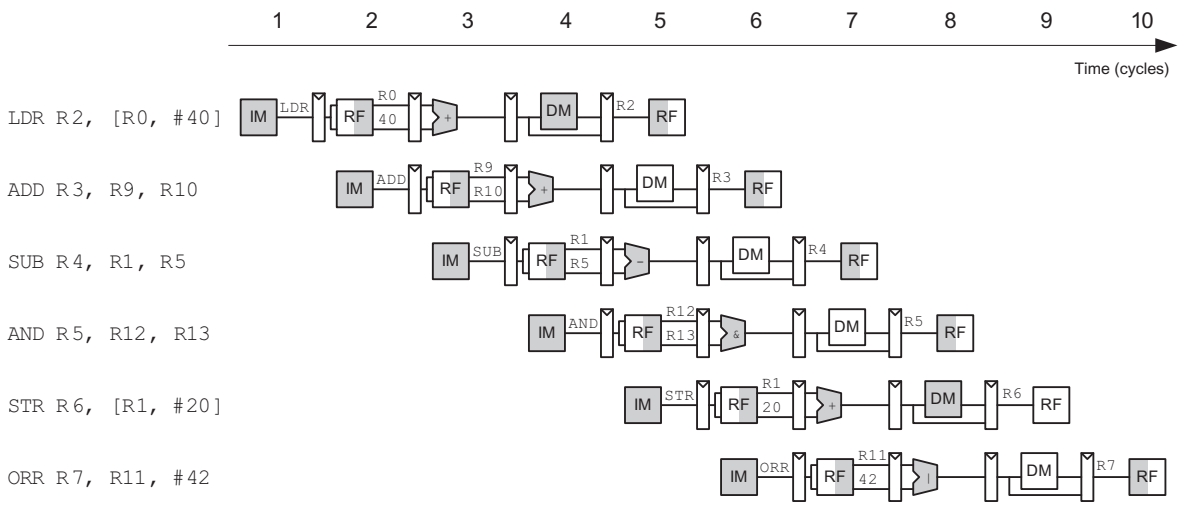


Figure 7.43 Abstract view of pipeline in operation

processor than for the single-cycle processor. Similarly, the throughput is not quite five-times as great for a five-stage pipeline as for the single-cycle processor. Nevertheless, the throughput advantage is substantial.

Figure 7.43 shows an abstracted view of the pipeline in operation in which each stage is represented pictorially. Each pipeline stage is represented with its major component—instruction memory (IM), register file (RF) read, ALU execution, data memory (DM), and register file write-back—to illustrate the flow of instructions through the pipeline. Reading across a row shows the clock cycles in which a particular instruction is in each stage. For example, the `SUB` instruction is fetched in cycle 3 and executed in cycle 5. Reading down a column shows what the various pipeline stages are doing on a particular cycle. For example, in cycle 6, the `ORR` instruction is being fetched from instruction memory, whereas R1 is being read from the register file, the ALU is computing R12 AND R13, the data memory is idle, and the register file is writing a sum to R3. Stages are shaded to indicate when they are used. For example, the data memory is used by `LDR` in cycle 4 and by `STR` in cycle 8. The instruction memory and ALU are used in every cycle. The register file is written by every instruction except `STR`. In the pipelined processor, the register file is written in the first part of a cycle and read in the second part, as suggested by the shading. This way, data can be written and read back within a single cycle.

A central challenge in pipelined systems is handling hazards that occur when the results of one instruction are needed by a subsequent instruction before the former instruction has completed. For example, if

the ADD in Figure 7.43 used R2 rather than R10, a hazard would occur because the R2 register has not been written by the LDR by the time it is read by the ADD. 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 7.44(a) shows the single-cycle datapath stretched out to leave room for the pipeline registers. Figure 7.44(b) shows the pipelined datapath formed by inserting four pipeline registers to separate the datapath into five stages. The stages and their boundaries are indicated in blue. Signals are given a suffix (F, D, E, M, or W) to indicate the stage in which they reside.

The register file is peculiar because it is read in the Decode stage and written in the Writeback stage. It is drawn in the Decode stage, but the write address and data come from the Writeback stage. This feedback will lead to pipeline hazards, which are discussed in Section 7.5.3. The register file in the pipelined processor writes on the falling edge of CLK so that it

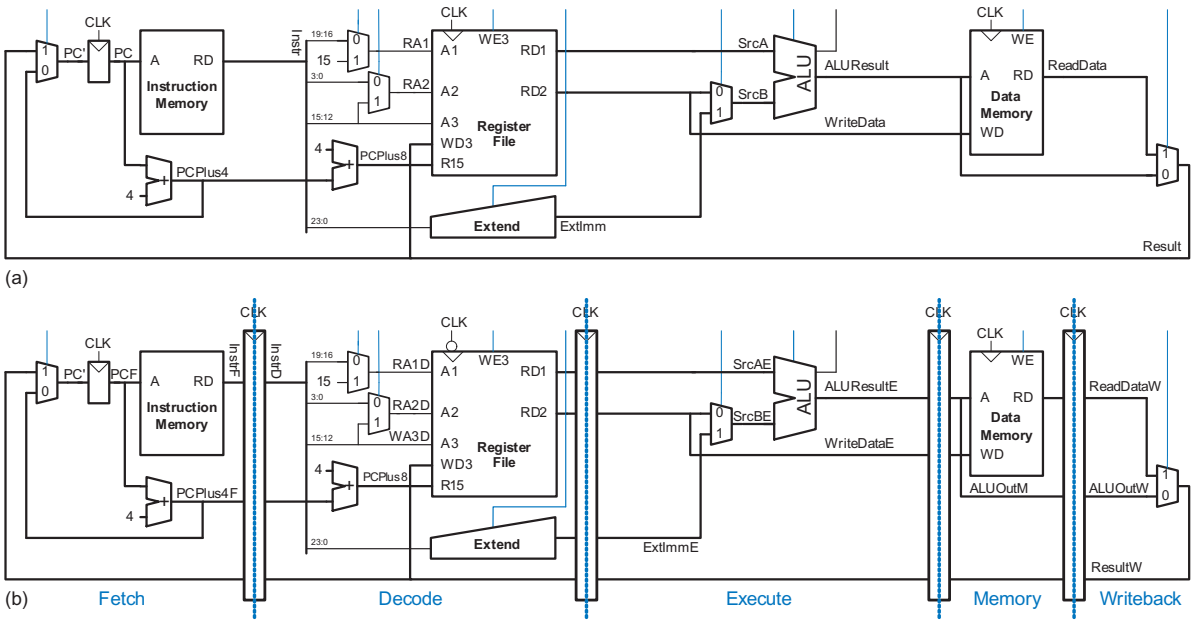


Figure 7.44 Datapaths: (a) single-cycle and (b) pipelined

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 7.44(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 write address comes from $InstrD_{15:12}$ (also known as *WA3D*), which is a Decode stage signal. In the pipeline diagram of Figure 7.43, during cycle 5, the result of the `LDR` instruction would be incorrectly written to R5 rather than R2.

Figure 7.45 shows a corrected datapath, with the modification in black. The *WA3* signal is now pipelined along through the Execution, Memory, and Writeback stages, so it remains in sync with the rest of the instruction. *WA3W* and *ResultW* are fed back together to the register file in the Writeback stage.

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

Figure 7.46 shows another optimization to save a 32-bit adder and register in the PC logic. Observe in Figure 7.45 that each time the program counter is incremented, *PCPlus4F* is simultaneously written to the PC and the pipeline register between the Fetch and Decode stages. Moreover, on the subsequent cycle, the value in both of these registers is incremented by 4 again. Thus, *PCPlus4F* for the instruction in the Fetch stage is logically equivalent to *PCPlus8D* for the instruction in the Decode stage. Sending this signal ahead saves the pipeline register and second adder.[3]

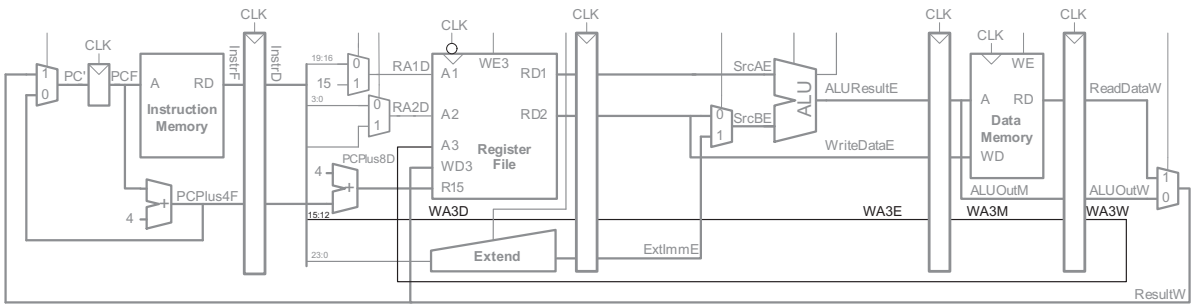


Figure 7.45 Corrected pipelined datapath

[3] There is a potential problem with this simplification when the PC is written with *ResultW* rather than *PCPlus4F*. However, this case is handled in Section 7.5.3 by flushing the pipeline, so *PCPlus8D* becomes a don't care and the pipeline still operates correctly.

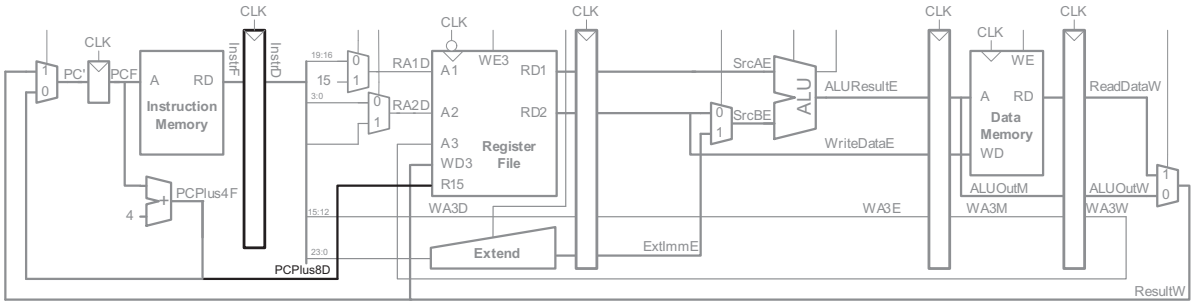


Figure 7.46 Optimized PC logic eliminating a register and adder

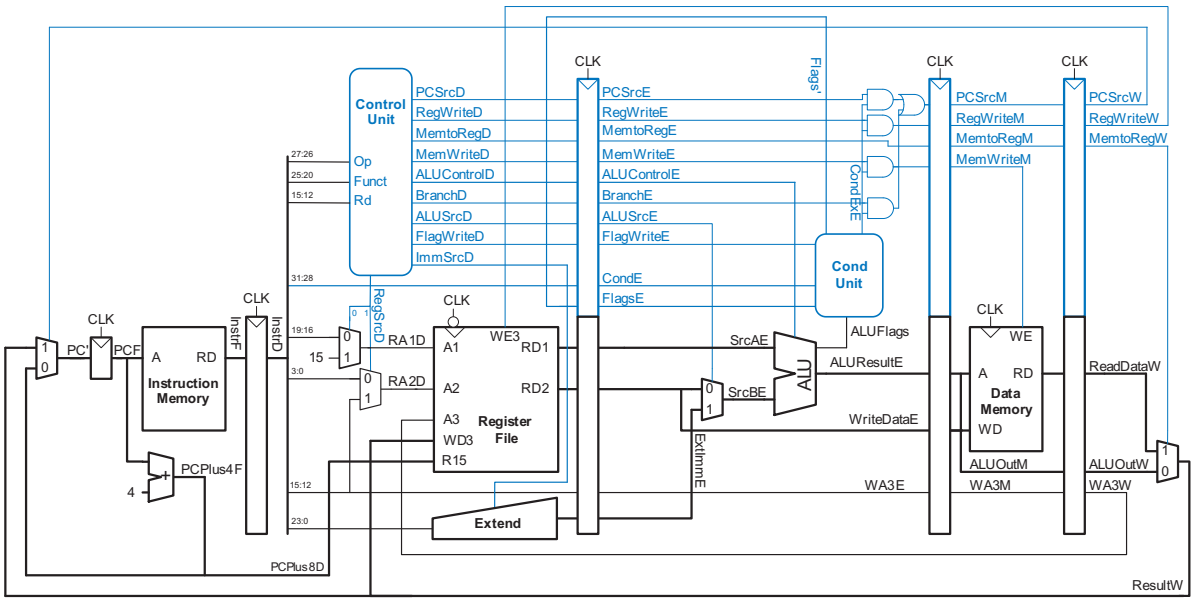


Figure 7.47 Pipelined processor with control

7.5.2 Pipelined Control

The pipelined processor takes the same control signals as the single-cycle processor and therefore uses the same control unit. The control unit examines the *Op* and *Funct* fields of the instruction in the Decode stage to produce the control signals, as was described in Section 7.3.2. These control signals must be pipelined along with the data so that they remain synchronized with the instruction. The control unit also examines the *Rd* field to handle writes to R15 (PC).

The entire pipelined processor with control is shown in Figure 7.47. *RegWrite* must be pipelined into the Writeback stage before it feeds back to the register file, just as *WA3* was pipelined in Figure 7.45.

7.5.3 Hazards

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

The register file can be read and written in the same cycle. The write takes place during the first half of the cycle and the read takes place 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 7.48 illustrates hazards that occur when one instruction writes a register (R1) and subsequent instructions read this register. This is called a *read after write* (*RAW*) *hazard*. The `ADD` instruction writes a result into R1 in the first half of cycle 5. However, the `AND` instruction reads R1 on cycle 3, obtaining the wrong value. The `ORR` instruction reads R1 on cycle 4, again obtaining the wrong value. The `SUB` instruction reads R1 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 R1. 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 `AND` instructions so that the dependent instruction does not read the result (R1) until it is available in the register file, as shown in Figure 7.49. Such a *software interlock* complicates programming as well as degrading performance, so it is not ideal.

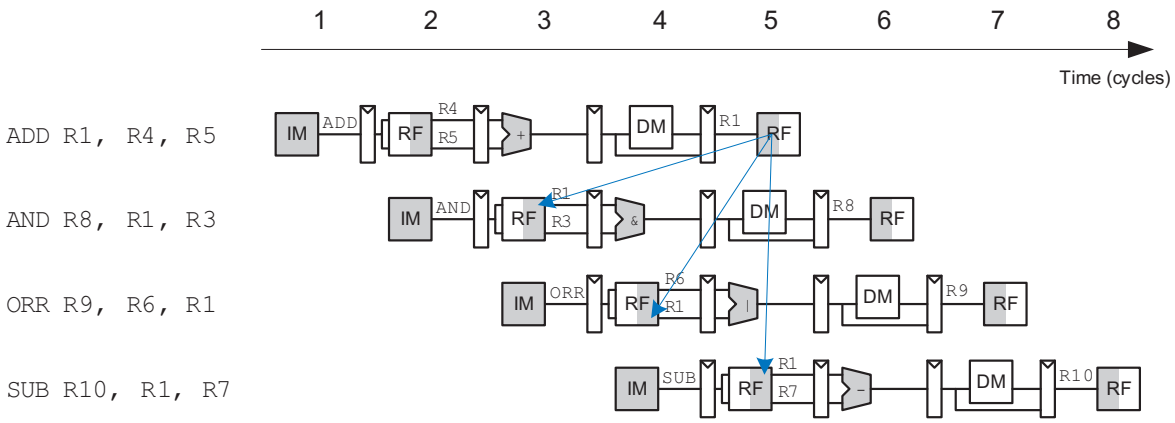


Figure 7.48 Abstract pipeline diagram illustrating hazards

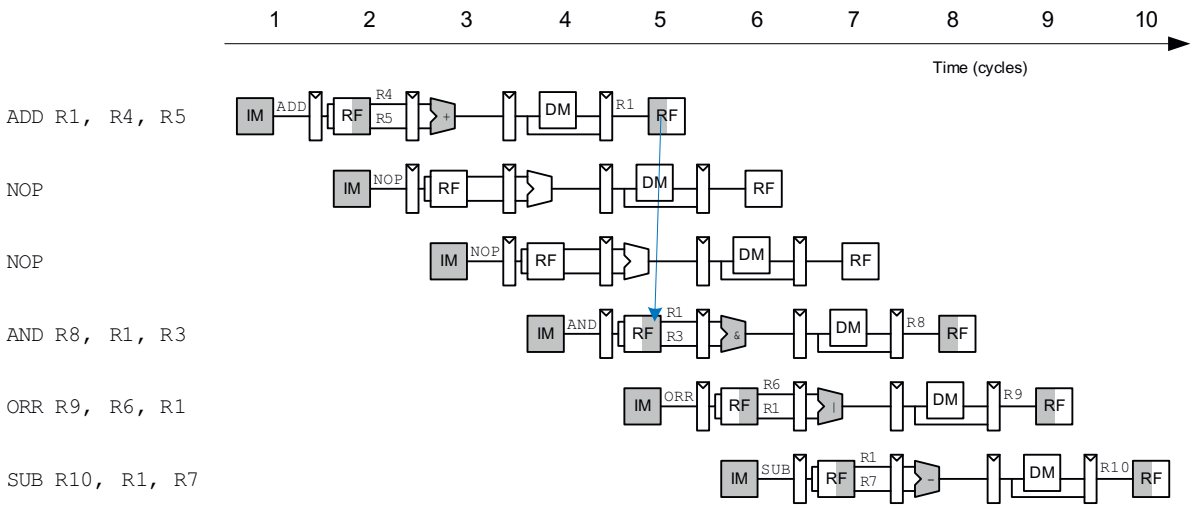


Figure 7.49 **Solving data hazard with** `NOP`

On closer inspection, observe from Figure 7.48 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.

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

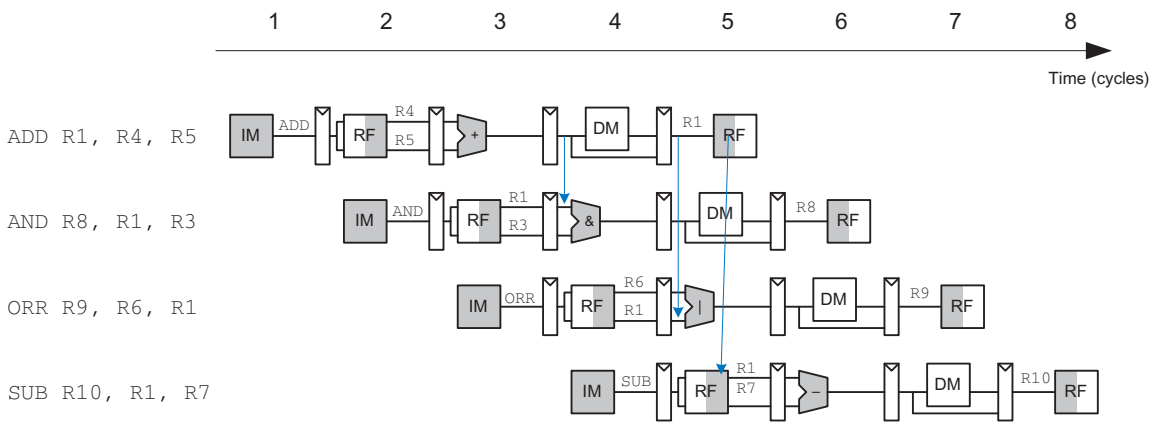


Figure 7.50 **Abstract pipeline diagram illustrating forwarding**

the Execute stage. This requires adding multiplexers in front of the ALU to select the operand from either the register file or the Memory or Writeback stage. Figure 7.50 illustrates this principle. In cycle 4, R1 is forwarded from the Memory stage of the `ADD` instruction to the Execute stage of the dependent `AND` instruction. In cycle 5, R1 is forwarded from the Writeback stage of the `ADD` instruction to the Execute stage of the dependent `ORR` instruction.

Forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or Writeback stage. Figure 7.51 modifies the pipelined processor to support forwarding. It adds a *Hazard Unit* and two *forwarding multiplexers*. The Hazard Unit receives four match signals from the datapath (abbreviated to *Match* in Figure 7.51) that indicate whether the source registers in the Execute stage match the destination registers in the Memory and Execute stages:

```
Match_1E_M = (RA1E == WA3M)
Match_1E_W = (RA1E == WA3W)
Match_2E_M = (RA2E == WA3M)
Match_2E_W = (RA2E == WA3W)
```

The Hazard Unit also receives the *RegWrite* signals from the Memory and Writeback stages to know whether the destination register will actually be written (e.g., the `STR` and `B` instructions do not write results to the register file and, hence, do not need to have their results forwarded).

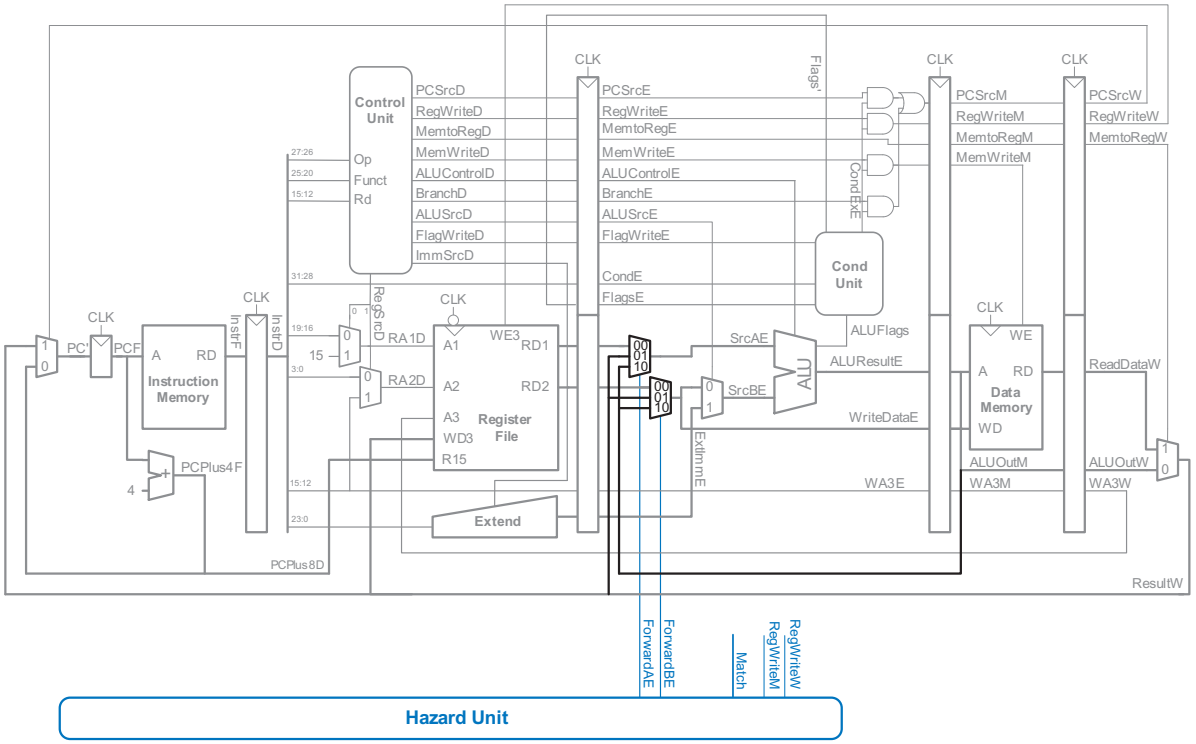


Figure 7.51 Pipelined processor with forwarding to solve hazards

Note that these signals are *connected by name*. In other words, rather than cluttering up the diagram with long wires running from the control signals at the top to the Hazard Unit at the bottom, the connections are indicated by a short stub of wire labeled with the control signal name to which it is connected. The Match signal logic and pipeline registers for *RA1E* and *RA2E* are also left out to limit clutter.

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 (*ALUOutM* or *ResultW*). It should forward from a stage if that stage will write a destination register and the destination register matches the source register. 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* is given here. The forwarding logic for *SrcBE* (*ForwardBE*) is identical except that it checks *Match_2E*.

```
if      (Match_1E_M • RegWriteM)  ForwardAE = 10;  // SrcAE = ALUOutM
else if (Match_1E_W • RegWriteW)  ForwardAE = 01;  // SrcAE = ResultW
else                              ForwardAE = 00;  // SrcAE from regfile
```

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 LDR 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 LDR instruction has a *two-cycle latency*, because a dependent instruction cannot use its result until two cycles later. Figure 7.52 shows this problem. The LDR instruction receives data from memory at the end of cycle 4. But the AND instruction needs that data as a source operand at the beginning of cycle 4. There is no way to solve this hazard with forwarding.

The alternative solution is to *stall* the pipeline, holding up operation until the data is available. Figure 7.53 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 (ORR) 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 LDR to the Execute stage of AND. Also in cycle 5, source R1 of the ORR 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*, and it behaves like

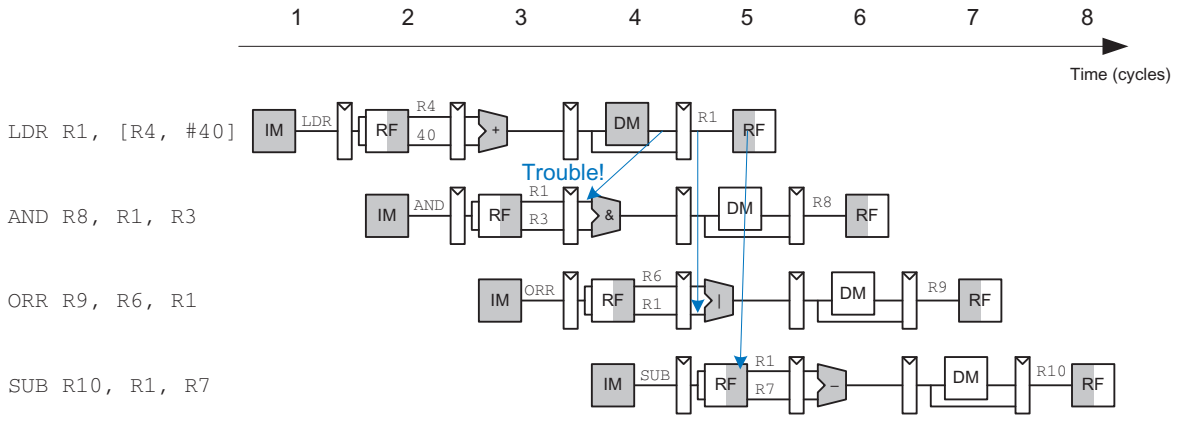


Figure 7.52 Abstract pipeline diagram illustrating trouble forwarding from LDR

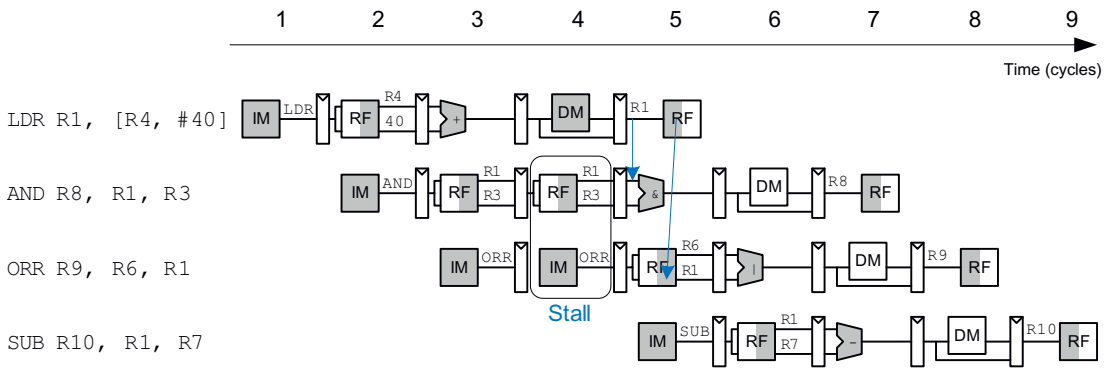


Figure 7.53 Abstract pipeline diagram illustrating stall to solve hazards

a NOP instruction. The bubble is introduced by zeroing out the Execute stage control signals during a Decode stall so that the bubble performs no action and changes no architectural state.

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

Figure 7.54 modifies the pipelined processor to add stalls for LDR data dependencies. The Hazard Unit examines the instruction in the Execute stage. If it is an LDR and its destination register (*WA3E*) matches either source operand of the instruction in the Decode stage (*RA1D* or *RA2D*), then that instruction must be stalled in the Decode stage until the source operand is ready.

Stalls are supported by adding enable inputs (*EN*) to the Fetch and Decode pipeline registers and a synchronous reset/clear (*CLR*) input to the Execute pipeline register. When an LDR stall occurs, *StallD* and *StallF* are asserted to force the Decode and Fetch stage pipeline registers to hold their old values. *FlushE* is also asserted to clear the contents of the Execute stage pipeline register, introducing a bubble.

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

```
Match_12D_E = (RA1D == WA3E) + (RA2D == WA3E)
LDRstall = Match_12D_E • MemtoRegE
StallF = StallD = FlushE = LDRstall
```

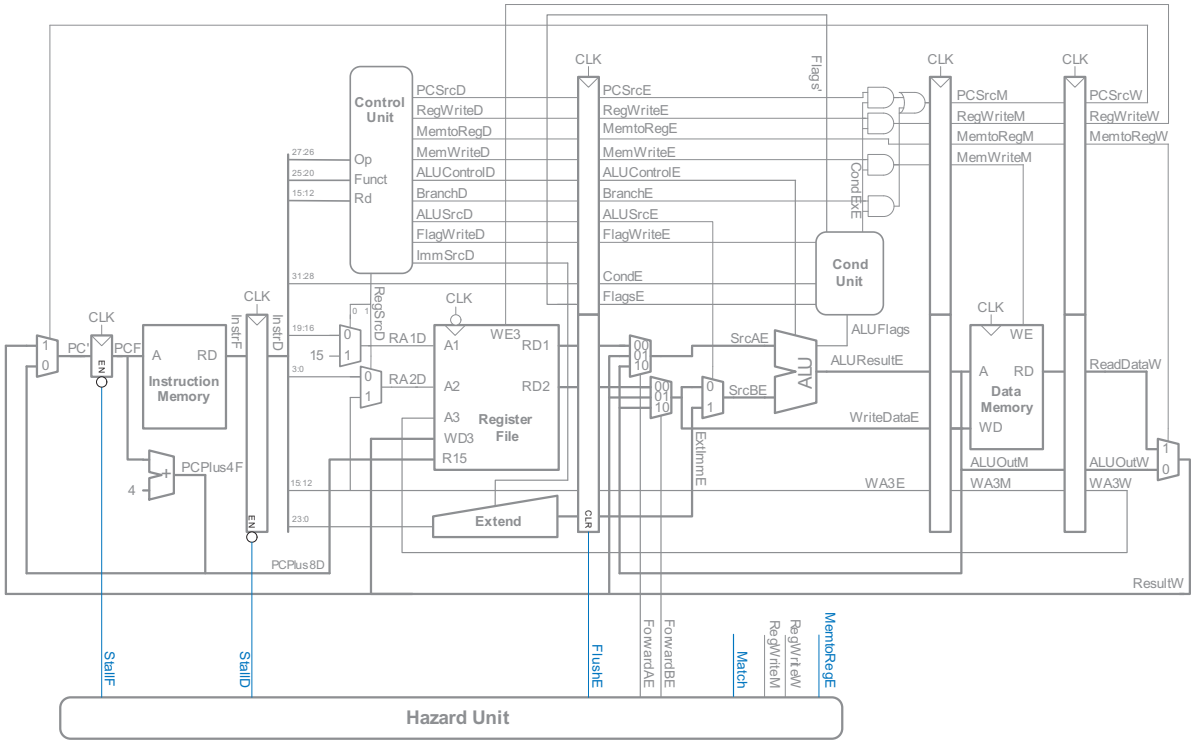


Figure 7.54 **Pipelined processor with stalls to solve `LDR` data hazard**

Solving Control Hazards

The `B` 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. Writes to R15 (PC) present a similar control hazard.

One mechanism for dealing with the control hazard is to stall the pipeline until the branch decision is made (i.e., *PCSrcW* is computed). Because the decision is made in the Writeback stage, the pipeline would have to be stalled for four cycles at every branch. This would severely degrade the system performance if it occurs often.

An alternative is to predict whether the branch will be taken and begin executing instructions based on the prediction. Once the branch decision is available, the processor can throw out the instructions if the prediction was wrong. In the pipeline presented so far (Figure 7.54), the processor predicts that branches are not taken and simply continues executing the program in order until *PCSrcW* is asserted to select the next *PC* from *ResultW* instead. If the branch should have been taken, then the

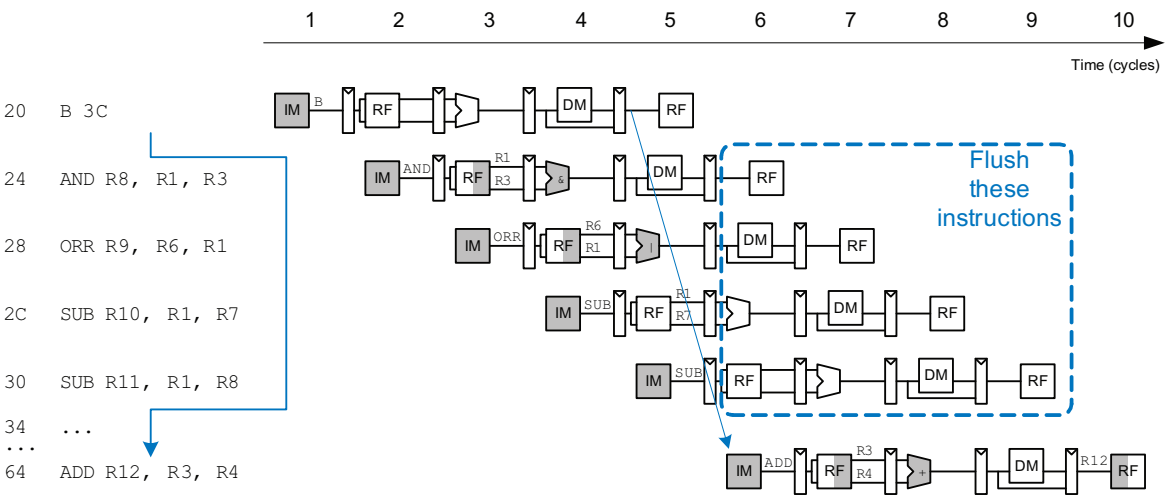


Figure 7.55 Abstract pipeline diagram illustrating flushing when a branch is taken

four 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 7.55 shows such a scheme in which a branch from address 0x20 to address 0x64 is taken. The PC is not written until cycle 5, by which point the `AND`, `ORR`, and both `SUB` instructions at addresses 0x24, 0x28, 0x2C, and 0x30 have already been fetched. These instructions must be flushed, and the `ADD` instruction is fetched from address 0x64 in cycle 6. This is somewhat of an improvement, but flushing so many instructions when the branch is taken still degrades performance.

We could reduce the branch misprediction penalty if the branch decision could be made earlier. Observe that the branch decision can be made in the Execute stage when the destination address has been computed and *CondEx* is known. Figure 7.56 shows the pipeline operation with the early branch decision being made in cycle 3. In cycle 4, the `AND` and `ORR` instructions are flushed and the `ADD` instruction is fetched. Now the branch misprediction penalty is reduced to only two instructions rather than four.

Figure 7.57 modifies the pipelined processor to move the branch decision earlier and handle control hazards. A branch multiplexer is added before the PC register to select the branch destination from *ALUResultE*. The *BranchTakenE* signal controlling this multiplexer is asserted on branches whose condition is satisfied. *PCSrcW* is now only asserted for writes to the PC, which still occur in the Writeback stage.

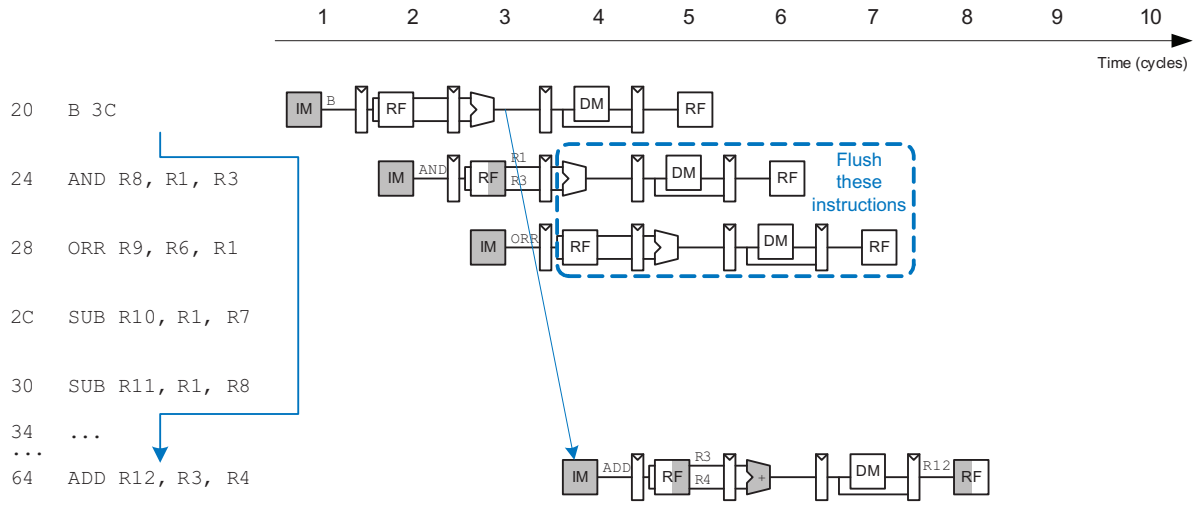


Figure 7.56 Abstract pipeline diagram illustrating earlier branch decision

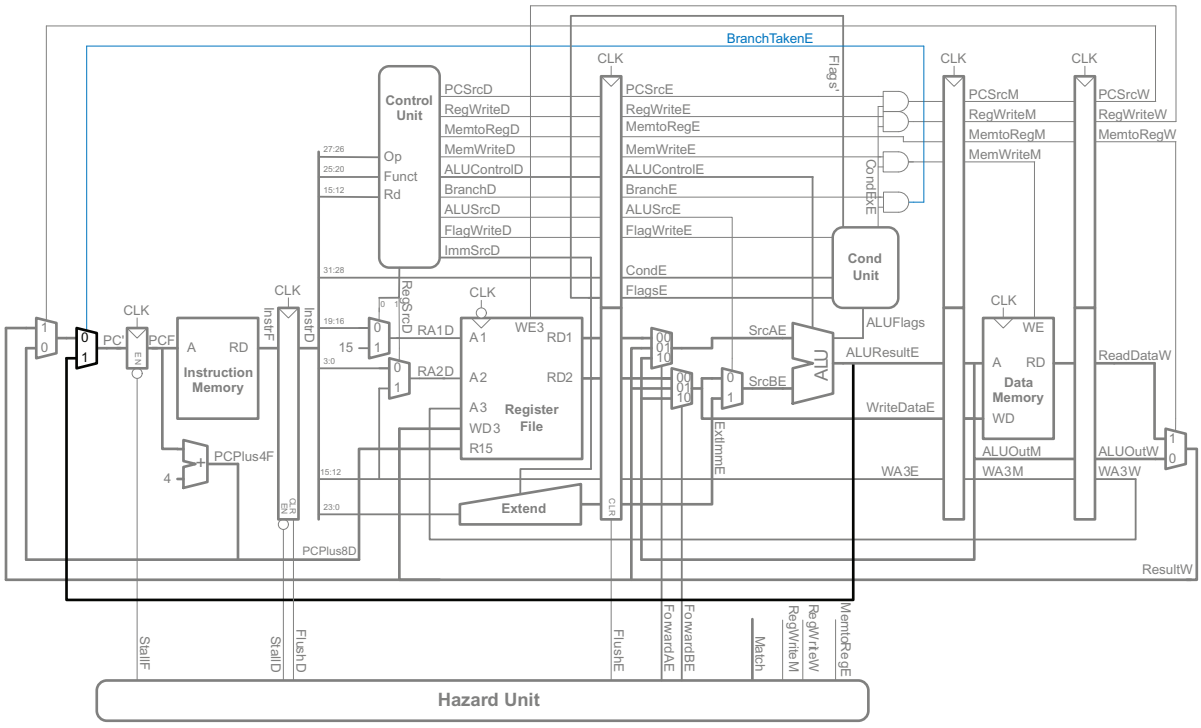


Figure 7.57 Pipelined processor handling branch control hazard

Finally, we must work out the stall and flush signals to handle branches and PC writes. It is common to goof this part of a pipelined processor design because the conditions are rather complicated. When a branch is taken, the subsequent two instructions must be flushed from the pipeline registers of the Decode and Execute stages. When a write to the PC is in the pipeline, the pipeline should be stalled until the write completes. This is done by stalling the Fetch stage. Recall that stalling one stage also requires flushing the next to prevent the instruction from being executed repeatedly. The logic to handle these cases is given here. *PCWrPending* is asserted when a PC write is in progress (in the Decode, Execute, or Memory stage). During this time, the Fetch stage is stalled and the Decode stage is flushed. When the PC write reaches the Writeback stage (*PCSrcW* asserted), *StallF* is released to allow the write to occur, but *FlushD* is still asserted so that the undesired instruction in the Fetch stage does not advance.

To reduce clutter, the Hazard Unit connections of *PCSrcD*, *PCSrcE*, *PCSrcM*, and *BranchTakenE* from the datapath are not shown in Figures 7.57 and 7.58.

```
PCWrPendingF = PCSrcD + PCSrcE + PCSrcM;
StallD = LDRstall;
StallF = LDRstall + PCWrPendingF;
FlushE = LDRstall + BranchTakenE;
FlushD = PCWrPendingF + PCSrcW + BranchTakenE;
```

Branches are very common, and even a two-cycle misprediction penalty still impacts performance. With a bit more work, the penalty could be reduced to one cycle for many branches. The destination address must be computed in the Decode stage as *PCBranchD* = *PCPlus8D* + *ExtImmD*. *BranchTakenD* must also be computed in the Decode stage based on *ALUFlagsE* generated by the previous instruction. This might increase the cycle time of the processor if these flags arrive late. These changes are left as an exercise to the reader (see Exercise 7.36).

Hazard Summary

In summary, RAW data hazards occur when an instruction depends on the result of another instruction that has not yet been written into the register file. The data hazards can be resolved by forwarding if the result is computed soon enough; otherwise, they require stalling the pipeline until the result is available. Control hazards occur when the decision of what instruction to fetch has not been made by the time the next instruction must be fetched. Control hazards are solved by predicting which instruction should be fetched and flushing the pipeline if the prediction is later determined to be wrong or by stalling the pipeline until the decision is made. Moving the decision as early as possible minimizes the number of instructions that are flushed on a misprediction. You may have observed

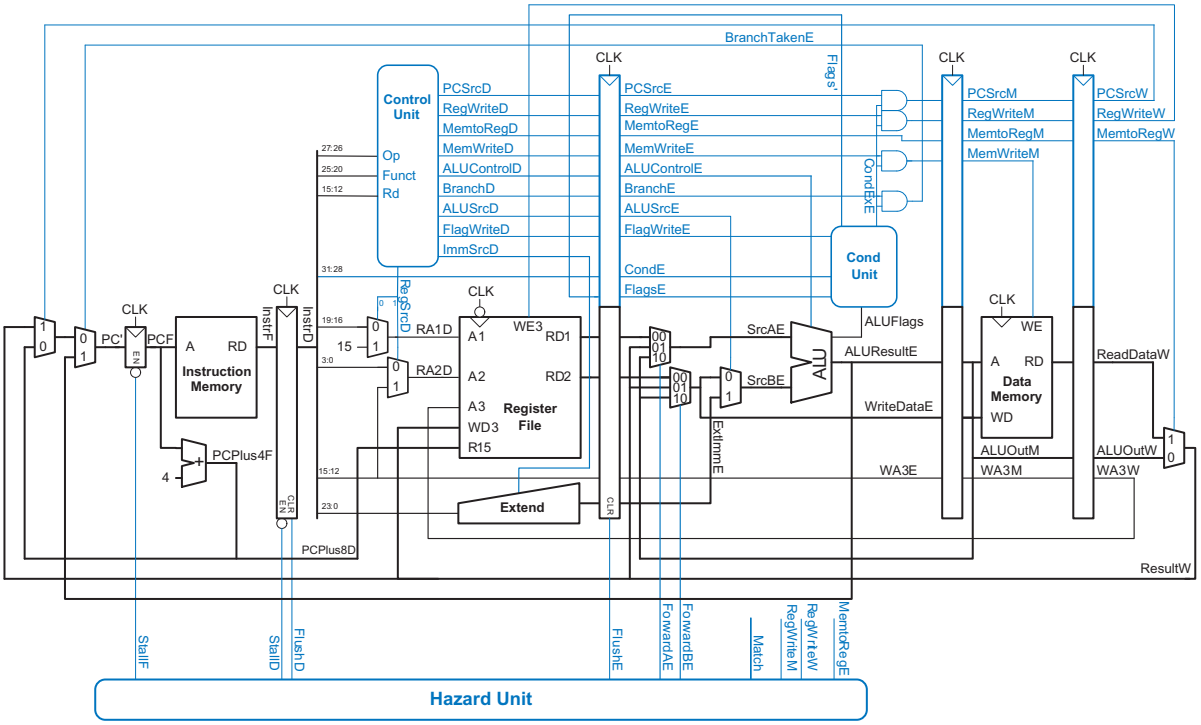


Figure 7.58 Pipelined processor with full hazard handling

by now that one of the challenges of designing a pipelined processor is to understand all the possible interactions between instructions and to discover all the hazards that may exist. Figure 7.58 shows the complete pipelined processor handling all of the hazards.

7.5.4 Performance Analysis

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

Example 7.7 PIPELINED PROCESSOR CPI

The SPECINT2000 benchmark considered in Example 7.5 consists of approximately 25% loads, 10% stores, 13% branches, and 52% data-processing 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 a flush. Ignore other hazards. Compute the average CPI of the pipelined processor.

Solution: The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of time that instruction is used. Loads take one clock cycle when there is no dependency and two cycles when the processor must stall for a dependency, so they have a CPI of $(0.6)(1) + (0.4)(2) = 1.4$. Branches take one clock cycle when they are predicted properly and three when they are not, so they have a CPI of $(0.5)(1) + (0.5)(3) = 2.0$. All other instructions have a CPI of 1. Hence, for this benchmark, average CPI $= (0.25)(1.4) + (0.1)(1) + (0.13)(2.0) + (0.52)(1) = 1.23$.

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

$$T_{c3} = max \begin{bmatrix} t_{pcq} + t_{mem} + t_{setup} & Fetch \\ 2(t_{RFread} + t_{setup}) & Decode \\ t_{pcq} + 2t_{mux} + t_{ALU} + t_{setup} & Execute \\ t_{pcq} + t_{mem} + t_{setup} & Memory \\ 2(t_{pcq} + t_{mux} + t_{RFsetup}) & Writeback \end{bmatrix} \quad (7.5)$$

Example 7.8 PROCESSOR PERFORMANCE COMPARISON

Ben Bitdiddle needs to compare the pipelined processor performance with that of the single-cycle and multicycle processors considered in Example 7.6. The logic delays were given in Table 7.5. Help Ben compare the execution time of 100 billion instructions from the SPECINT2000 benchmark for each processor.

Solution: According to Equation 7.5, the cycle time of the pipelined processor is $T_{c3} = \max[40 + 200 + 50,\ 2(100 + 50),\ 40 + 2(25) + 120 + 50,\ 40 + 200 + 50,\ 2(40 + 25 + 60)] = 300$ ps. According to Equation 7.1, the total execution time is $T_3 = (100 \times 10^9 \text{ instructions})(1.23 \text{ cycles/instruction})(300 \times 10^{-12} \text{ s/cycle}) = 36.9$ seconds. This compares with 84 seconds for the single-cycle processor and 140 seconds for the multicycle processor.

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

overhead limits the benefits one can hope to achieve from pipelining. The pipelined processor is similar in hardware requirements to the single-cycle processor, but it adds eight 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 processor supporting 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 and 7.40.

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 literal pools, a more complete processor must also be able to read data from the instruction memory. Chapter 8 will revisit memory systems, including the interaction of the caches with main memory.

The processor is composed of a datapath and a controller. The controller, in turn, is composed of the Decoder and the Conditional Logic. Figure 7.59 shows a block diagram of the single-cycle processor interfaced to external memories.

The HDL code is partitioned into several sections. Section 7.6.1 provides HDL for the single-cycle processor datapath and controller. Section 7.6.2 presents the generic building blocks, such as registers and multiplexers, which are used by any microarchitecture. Section 7.6.3 introduces the testbench and external memories. The HDL is available in electronic form on this book's website (see the Preface).

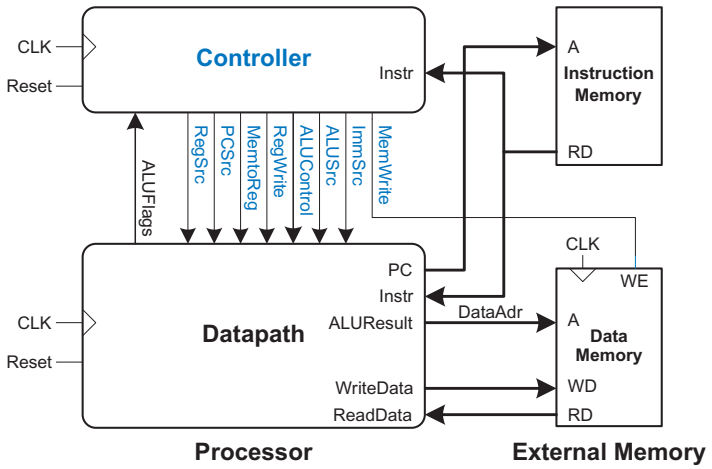


Figure 7.59 Single-cycle processor interfaced to external memory

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 arm(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  [3:0]  ALUFlags;
  logic         RegWrite,
                ALUSrc, MemtoReg, PCSrc;
  logic  [1:0]  RegSrc, ImmSrc, ALUControl;

  controller c(clk, reset, Instr[31:12], ALUFlags,
               RegSrc, RegWrite, ImmSrc,
               ALUSrc, ALUControl,
               MemWrite, MemtoReg, PCSrc);
  datapath dp(clk, reset,
              RegSrc, RegWrite, ImmSrc,
              ALUSrc, ALUControl,
              MemtoReg, PCSrc,
              ALUFlags, PC, Instr,
              ALUResult, WriteData, ReadData);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity arm is -- single cycle processor
  port(clk, reset:        in  STD_LOGIC;
       PC:                out STD_LOGIC_VECTOR(31 downto 0);
       Instr:             in  STD_LOGIC_VECTOR(31 downto 0);
       MemWrite:          out STD_LOGIC;
       ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
       ReadData:          in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of arm is
  component controller
  port(clk, reset:     in  STD_LOGIC;
       Instr:          in  STD_LOGIC_VECTOR(31 downto 12);
       ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
       RegSrc:         out STD_LOGIC_VECTOR(1 downto 0);
       RegWrite:       out STD_LOGIC;
       ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
       ALUSrc:         out STD_LOGIC;
       ALUControl:     out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:       out STD_LOGIC;
       MemtoReg:       out STD_LOGIC;
       PCSrc:          out STD_LOGIC);
  end component;
  component datapath
    port(clk, reset:      in     STD_LOGIC;
      RegSrc:             in     STD_LOGIC_VECTOR(1 downto 0);
      RegWrite:           in     STD_LOGIC;
      ImmSrc:             in     STD_LOGIC_VECTOR(1 downto 0);
      ALUSrc:             in     STD_LOGIC;
      ALUControl:         in     STD_LOGIC_VECTOR(1 downto 0);
      MemtoReg:           in     STD_LOGIC;
      PCSrc:              in     STD_LOGIC;
      ALUFlags:           out    STD_LOGIC_VECTOR(3 downto 0);
      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 component;
    signal RegWrite, ALUSrc, MemtoReg, PCSrc: STD_LOGIC;
    signal RegSrc, ImmSrc, ALUControl: STD_LOGIC_VECTOR
                                       (1 downto 0);
    signal ALUFlags: STD_LOGIC_VECTOR(3 downto 0);
begin
  cont: controller port map(clk, reset, Instr(31 downto 12),
                            ALUFlags, RegSrc, RegWrite,
                            ImmSrc, ALUSrc, ALUControl,
                            MemWrite, MemtoReg, PCSrc);
  dp: datapath port map(clk, reset, RegSrc, RegWrite, ImmSrc,
                        ALUSrc, ALUControl, MemtoReg, PCSrc,
                        ALUFlags, PC, Instr, ALUResult,
                        WriteData, ReadData);
end;
```

HDL Example 7.2 CONTROLLER

SystemVerilog

```
module controller(input  logic          clk, reset,
                  input  logic  [31:12] Instr,
                  input  logic  [3:0]   ALUFlags,
                  output logic  [1:0]   RegSrc,
                  output logic          RegWrite,
                  output logic  [1:0]   ImmSrc,
                  output logic          ALUSrc,
                  output logic  [1:0]   ALUControl,
                  output logic          MemWrite, MemtoReg,
                  output logic          PCSrc);
  logic [1:0] FlagW;
  logic       PCS, RegW, MemW;

  decoder dec(Instr[27:26], Instr[25:20], Instr[15:12],
              FlagW, PCS, RegW, MemW,
              MemtoReg, ALUSrc, ImmSrc, RegSrc, ALUControl);
  condlogic cl(clk, reset, Instr[31:28], ALUFlags,
               FlagW, PCS, RegW, MemW,
               PCSrc, RegWrite, MemWrite);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- single cycle control
  port(clk, reset:     in  STD_LOGIC;
       Instr:          in  STD_LOGIC_VECTOR(31 downto 12);
       ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
       RegSrc:         out STD_LOGIC_VECTOR(1 downto 0);
       RegWrite:       out STD_LOGIC;
       ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
       ALUSrc:         out STD_LOGIC;
       ALUControl:     out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:       out STD_LOGIC;
       MemtoReg:       out STD_LOGIC;
       PCSrc:          out STD_LOGIC);
end;

architecture struct of controller is
  component decoder
    port(Op:               in  STD_LOGIC_VECTOR(1 downto 0);
         Funct:            in  STD_LOGIC_VECTOR(5 downto 0);
         Rd:               in  STD_LOGIC_VECTOR(3 downto 0);
         FlagW:            out STD_LOGIC_VECTOR(1 downto 0);
         PCS, RegW, MemW:  out STD_LOGIC;
         MemtoReg, ALUSrc: out STD_LOGIC;
         ImmSrc, RegSrc:   out STD_LOGIC_VECTOR(1 downto 0);
         ALUControl:       out STD_LOGIC_VECTOR(1 downto 0));
  end component;
  component condlogic
    port(clk, reset:       in  STD_LOGIC;
         Cond:             in  STD_LOGIC_VECTOR(3 downto 0);
         ALUFlags:         in  STD_LOGIC_VECTOR(3 downto 0);
         FlagW:            in  STD_LOGIC_VECTOR(1 downto 0);
         PCS, RegW, MemW:  in  STD_LOGIC;
         PCSrc, RegWrite:  out STD_LOGIC;
         MemWrite:         out STD_LOGIC);
  end component;
  signal FlagW: STD_LOGIC_VECTOR(1 downto 0);
  signal PCS, RegW, MemW: STD_LOGIC;
begin
  dec: decoder port map(Instr(27 downto 26), Instr(25 downto 20),
                        Instr(15 downto 12), FlagW, PCS,
                        RegW, MemW, MemtoReg, ALUSrc, ImmSrc,
                        RegSrc, ALUControl);
  cl: condlogic port map(clk, reset, Instr(31 downto 28),
                         ALUFlags, FlagW, PCS, RegW, MemW,
                         PCSrc, RegWrite, MemWrite);
end;
```

HDL Example 7.3 DECODER

SystemVerilog

```
module decoder(input  logic [1:0] Op,
               input  logic [5:0] Funct,
               input  logic [3:0] Rd,
               output logic [1:0] FlagW,
               output logic       PCS, RegW, MemW,
               output logic       MemtoReg, ALUSrc,
               output logic [1:0] ImmSrc, RegSrc, ALUControl);

  logic [9:0] controls;
  logic       Branch, ALUOp;

  // Main Decoder
  always_comb
      casex(Op)
                             // Data-processing immediate
        2'b00: if (Funct[5]) controls = 10'b0000101001;
                             // Data-processing register
               else          controls = 10'b0000001001;
                             // LDR
        2'b01: if (Funct[0]) controls = 10'b0001111000;
                             // STR
               else          controls = 10'b1001110100;
                             // B
        2'b10:               controls = 10'b0110100010;
                             // Unimplemented
        default:             controls = 10'bx;
      endcase

  assign {RegSrc, ImmSrc, ALUSrc, MemtoReg,
    RegW, MemW, Branch, ALUOp} = controls;

  // ALU Decoder
  always_comb
  if (ALUOp) begin          // which DP Instr?
    case(Funct[4:1])
        4'b0100:  ALUControl = 2'b00;  // ADD
        4'b0010:  ALUControl = 2'b01;  // SUB
        4'b0000:  ALUControl = 2'b10;  // AND
        4'b1100:  ALUControl = 2'b11;  // ORR
        default:  ALUControl = 2'bx;   // unimplemented
    endcase

    // update flags if S bit is set (C & V only for arith)
    FlagW[1]      = Funct[0];
    FlagW[0]      = Funct[0] &
      (ALUControl == 2'b00 | ALUControl == 2'b01);
  end else begin
    ALUControl = 2'b00;  // add for non-DP instructions
    FlagW      = 2'b00;  // don't update Flags
  end

  // PC Logic
  assign PCS  = ((Rd == 4'b1111) & RegW) | Branch;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity decoder is -- main control decoder
  port(Op:               in  STD_LOGIC_VECTOR(1 downto 0);
       Funct:            in  STD_LOGIC_VECTOR(5 downto 0);
       Rd:               in  STD_LOGIC_VECTOR(3 downto 0);
       FlagW:            out STD_LOGIC_VECTOR(1 downto 0);
       PCS, RegW, MemW:  out STD_LOGIC;
       MemtoReg, ALUSrc: out STD_LOGIC;
       ImmSrc, RegSrc:   out STD_LOGIC_VECTOR(1 downto 0);
       ALUControl:       out STD_LOGIC_VECTOR(1 downto 0));
end;
architecture behave of decoder is
  signal controls:      STD_LOGIC_VECTOR(9 downto 0);
  signal ALUOp, Branch: STD_LOGIC;
  signal op2:           STD_LOGIC_VECTOR(3 downto 0);
begin
  op2 <= (Op, Funct(5), Funct(0));
  process(all) begin -- Main Decoder
    case? (op2) is
      when "000-"  => controls <= "0000001001";
      when "001-"  => controls <= "0000101001";
      when "01-0"  => controls <= "1001110100";
      when "01-1"  => controls <= "0001111000";
      when "10--"  => controls <= "0110100010";
      when others  => controls <= "----------";
    end case?;
  end process;

  (RegSrc, ImmSrc, ALUSrc, MemtoReg, RegW, MemW,
  Branch, ALUOp) <= controls;

  process(all) begin -- ALU Decoder
  if (ALUOp) then
    case Funct(4 downto 1) is
      when "0100" => ALUControl <= "00"; -- ADD
      when "0010" => ALUControl <= "01"; -- SUB
      when "0000" => ALUControl <= "10"; -- AND
      when "1100" => ALUControl <= "11"; -- ORR
      when others => ALUControl <= "--"; -- unimplemented
    end case;
    FlagW(1) <= Funct(0);
    FlagW(0) <= Funct(0) and (not ALUControl(1));
  else
    ALUControl <= "00";
    FlagW <= "00";
  end if;
  end process;

  PCS <= ((and Rd) and RegW) or Branch;
end;
```

HDL Example 7.4 CONDITIONAL LOGIC

SystemVerilog

```
module condlogic(input   logic         clk, reset,
                 input   logic  [3:0]  Cond,
                 input   logic  [3:0]  ALUFlags,
                 input   logic  [1:0]  FlagW,
                 input   logic         PCS, RegW, MemW,
                 output  logic         PCSrc, RegWrite,
                                       MemWrite);

  logic  [1:0]  FlagWrite;
  logic  [3:0]  Flags;
  logic         CondEx;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                       ALUFlags[3:2], Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                       ALUFlags[1:0], Flags[1:0]);

  // write controls are conditional
  condcheck cc(Cond,  Flags,  CondEx);
  assign FlagWrite =  FlagW  &  {2{CondEx}};
  assign RegWrite  =  RegW   &  CondEx;
  assign MemWrite  =  MemW   &  CondEx;
  assign PCSrc     =  PCS    &  CondEx;
endmodule

module condcheck(input  logic  [3:0]  Cond,
                 input  logic  [3:0]  Flags,
                 output logic         CondEx);

  logic neg, zero, carry, overflow, ge;

  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb
    case(Cond)
      4'b0000: CondEx = zero;             // EQ
      4'b0001: CondEx = ~zero;            // NE
      4'b0010: CondEx = carry;            // CS
      4'b0011: CondEx = ~carry;           // CC
      4'b0100: CondEx = neg;              // MI
      4'b0101: CondEx = ~neg;             // PL
      4'b0110: CondEx = overflow;         // VS
      4'b0111: CondEx = ~overflow;        // VC
      4'b1000: CondEx = carry & ~zero;    // HI
      4'b1001: CondEx = ~(carry & ~zero); // LS
      4'b1010: CondEx = ge;               // GE
      4'b1011: CondEx = ~ge;              // LT
      4'b1100: CondEx = ~zero & ge;       // GT
      4'b1101: CondEx = ~(~zero & ge);    // LE
      4'b1110: CondEx = 1'b1;             // Always
      default: CondEx = 1'bx;             // undefined
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condlogic is -- Conditional logic
  port(clk, reset:       in  STD_LOGIC;
       Cond:             in  STD_LOGIC_VECTOR(3 downto 0);
       ALUFlags:         in  STD_LOGIC_VECTOR(3 downto 0);
       FlagW:            in  STD_LOGIC_VECTOR(1 downto 0);
       PCS, RegW, MemW:  in  STD_LOGIC;
       PCSrc, RegWrite:  out STD_LOGIC;
       MemWrite:         out STD_LOGIC);
end;

architecture behave of condlogic is
  component condcheck
  port(Cond:       in  STD_LOGIC_VECTOR(3 downto 0);
       Flags:      in  STD_LOGIC_VECTOR(3 downto 0);
       CondEx:     out STD_LOGIC);
  end component;
  component flopenr 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 component;
  signal FlagWrite: STD_LOGIC_VECTOR(1 downto 0);
  signal Flags:     STD_LOGIC_VECTOR(3 downto 0);
  signal CondEx:    STD_LOGIC;
begin
  flagreg1: flopenr generic map(2)
    port map(clk, reset, FlagWrite(1),
             ALUFlags(3 downto 2), Flags(3 downto 2));
  flagreg0: flopenr generic map(2)
    port map(clk, reset, FlagWrite(0),
             ALUFlags(1 downto 0), Flags(1 downto 0));
  cc: condcheck port map(Cond, Flags, CondEx);

  FlagWrite  <=  FlagW and (CondEx, CondEx);
  RegWrite   <=  RegW  and CondEx;
  MemWrite   <=  MemW  and CondEx;
  PCSrc      <=  PCS   and CondEx;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condcheck is
  port(Cond:     in  STD_LOGIC_VECTOR(3 downto 0);
    Flags:       in  STD_LOGIC_VECTOR(3 downto 0);
    CondEx:      out STD_LOGIC);
end;

architecture behave of condcheck is
  signal neg, zero, carry, overflow, ge: STD_LOGIC;
begin
  (neg, zero, carry, overflow) <= Flags;
  ge <= (neg xnor overflow);

  process(all) begin -- Condition checking
    case Cond is
      when "0000"  =>  CondEx  <=  zero;
      when "0001"  =>  CondEx  <=  not zero;
      when "0010"  =>  CondEx  <=  carry;
      when "0011"  =>  CondEx  <=  not carry;
      when "0100"  =>  CondEx  <=  neg;
      when "0101"  =>  CondEx  <=  not neg;
      when "0110"  =>  CondEx  <=  overflow;
```

```
        when "0111"  =>  CondEx  <=  not overflow;
        when "1000"  =>  CondEx  <=  carry and (not zero);
        when "1001"  =>  CondEx  <=  not(carry and (not zero));
        when "1010"  =>  CondEx  <=  ge;
        when "1011"  =>  CondEx  <=  not ge;
        when "1100"  =>  CondEx  <=  (not zero) and ge;
        when "1101"  =>  CondEx  <=  not ((not zero) and ge);
        when "1110"  =>  CondEx  <=  '1';
        when others  =>  CondEx  <=  '-';
      end case;
    end process;
end;
```

HDL Example 7.5 DATAPATH

SystemVerilog

```
module datapath(input  logic         clk, reset,
                input  logic  [1:0]  RegSrc,
                input  logic         RegWrite,
                input  logic  [1:0]  ImmSrc,
                input  logic         ALUSrc,
                input  logic  [1:0]  ALUControl,
                input  logic         MemtoReg,
                input  logic         PCSrc,
                output logic  [3:0]  ALUFlags,
                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, PCPlus8;
  logic  [31:0] ExtImm, SrcA, SrcB, Result;
  logic  [3:0]  RA1, RA2;

  // next PC logic
  mux2 #(32)   pcmux(PCPlus4, Result, PCSrc, PCNext);
  flopr #(32)  pcreg(clk, reset, PCNext, PC);
  adder #(32)  pcadd1(PC, 32'b100, PCPlus4);
  adder #(32)  pcadd2(PCPlus4, 32'b100, PCPlus8);

  // register file logic
  mux2 #(4)  ra1mux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
  mux2 #(4)  ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
  regfile    rf(clk, RegWrite, RA1, RA2,
                Instr[15:12], Result, PCPlus8,
                SrcA, WriteData);
  mux2 #(32) resmux(ALUResult, ReadData, MemtoReg, Result);
  extend     ext(Instr[23:0], ImmSrc, ExtImm);

  // ALU logic
  mux2 #(32) srcbmux(WriteData, ExtImm, ALUSrc, SrcB);
  alu        alu(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity datapath is
  port(clk, reset:        in     STD_LOGIC;
       RegSrc:            in     STD_LOGIC_VECTOR(1 downto 0);
       RegWrite:          in     STD_LOGIC;
       ImmSrc:            in     STD_LOGIC_VECTOR(1 downto 0);
       ALUSrc:            in     STD_LOGIC;
       ALUControl:        in     STD_LOGIC_VECTOR(1 downto 0);
       MemtoReg:          in     STD_LOGIC;
       PCSrc:             in     STD_LOGIC;
       ALUFlags:          out    STD_LOGIC_VECTOR(3 downto 0);
       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 alu
  port(a, b:         in     STD_LOGIC_VECTOR(31 downto 0);
       ALUControl:   in     STD_LOGIC_VECTOR(1 downto 0);
       Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
       ALUFlags:     out    STD_LOGIC_VECTOR(3 downto 0));
  end component;
  component regfile
  port(clk:           in  STD_LOGIC;
       we3:           in  STD_LOGIC;
       ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
       wd3, r15:      in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component adder
  port(a, b:  in  STD_LOGIC_VECTOR(31 downto 0);
       y:     out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component extend
  port(Instr:   in  STD_LOGIC_VECTOR(23 downto 0);
       ImmSrc:  in  STD_LOGIC_VECTOR(1 downto 0);
       ExtImm:  out STD_LOGIC_VECTOR(31 downto 0));
  end component;
```

```
  component flopr generic(width: integer);
  port(clk, reset:  in  STD_LOGIC;
       d:           in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:           out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux2 generic(width: integer);
  port(d0, d1:  in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:       in  STD_LOGIC;
       y:       out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal PCNext, PCPlus4,
         PCPlus8:         STD_LOGIC_VECTOR(31 downto 0);
  signal ExtImm, Result:  STD_LOGIC_VECTOR(31 downto 0);
  signal SrcA, SrcB:      STD_LOGIC_VECTOR(31 downto 0);
  signal RA1, RA2:        STD_LOGIC_VECTOR(3 downto 0);
begin
  -- next PC logic
  pcmux:  mux2 generic map(32)
          port map(PCPlus4, Result, PCSrc, PCNext);
  pcreg: flopr generic map(32) port map(clk, reset, PCNext, PC);
  pcadd1: adder port map(PC, X"00000004", PCPlus4);
  pcadd2: adder port map(PCPlus4, X"00000004", PCPlus8);

  -- register file logic
  ra1mux: mux2 generic map (4)
    port map(Instr(19 downto 16), "1111", RegSrc(0), RA1);
  ra2mux: mux2 generic map (4) port map(Instr(3 downto 0),
                        Instr(15 downto 12), RegSrc(1), RA2);
  rf: regfile port map(clk, RegWrite, RA1, RA2,
                       Instr(15 downto 12), Result,
                       PCPlus8, SrcA, WriteData);
  resmux: mux2 generic map(32)
    port map(ALUResult, ReadData, MemtoReg, Result);
  ext: extend port map(Instr(23 downto 0), ImmSrc, ExtImm);

  -- ALU logic
  srcbmux: mux2 generic map(32)
    port map(WriteData, ExtImm, ALUSrc, SrcB);
  i_alu:  alu port map(SrcA, SrcB, ALUControl, ALUResult,
                       ALUFlags);
end;
```

7.6.2 Generic Building Blocks

This section contains generic building blocks that may be useful in any digital system, including a register file, adder, flip-flops, and a 2:1 multiplexer. The HDL for the ALU is left to Exercises 5.11 and 5.12.

HDL Example 7.6 REGISTER FILE

SystemVerilog

```
module regfile(input  logic          clk,
               input  logic          we3,
               input  logic  [3:0]   ra1, ra2, wa3,
               input  logic  [31:0]  wd3, r15,
               output logic  [31:0]  rd1, rd2);

  logic [31:0] rf[14:0];

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clock
  // register 15 reads PC + 8 instead

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

  assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
  assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity regfile is -- three-port register file
  port(clk:           in  STD_LOGIC;
       we3:           in  STD_LOGIC;
       ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
       wd3, r15:      in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array (31 downto 0) of
    STD_LOGIC_VECTOR(31 downto 0);
  signal mem: ramtype;
begin
  process(clk) begin
  if rising_edge(clk) then
    if we3 = '1' then mem(to_integer(wa3)) <= wd3;
    end if;
  end if;
  end process;
  process(all) begin
    if (to_integer(ra1) = 15) then rd1 <= r15;
    else rd1 <= mem(to_integer(ra1));
    end if;
    if (to_integer(ra2) = 15) then rd2 <= r15;
    else rd2 <= mem(to_integer(ra2));
    end if;
  end process;
end;
```

HDL Example 7.7 ADDER

SystemVerilog

```
module adder #(parameter WIDTH=8)
              (input  logic [WIDTH-1:0] a, b,
               output logic [WIDTH-1:0] y);

  assign y = a + b;
endmodule
```

VHDL

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

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

HDL Example 7.8 IMMEDIATE EXTENSION

SystemVerilog

```
module extend(input  logic  [23:0]  Instr,
              input  logic  [1:0]   ImmSrc,
              output logic  [31:0]  ExtImm);

  always_comb
    case(ImmSrc)
                // 8-bit unsigned immediate
      2'b00:    ExtImm = {24'b0, Instr[7:0]};
                // 12-bit unsigned immediate
      2'b01:    ExtImm = {20'b0, Instr[11:0]};
                // 24-bit two's complement shifted branch
      2'b10:    ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
      default: ExtImm = 32'bx; // undefined
  endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity extend is
  port(Instr:   in  STD_LOGIC_VECTOR(23 downto 0);
       ImmSrc:  in  STD_LOGIC_VECTOR(1 downto 0);
       ExtImm:  out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of extend is
begin
  process(all) begin
    case ImmSrc is
      when "00"  => ExtImm <= (X"000000", Instr(7 downto 0));
      when "01"  => ExtImm <= (X"00000", Instr(11 downto 0));
      when "10"  => ExtImm <= (Instr(23), Instr(23),
                               Instr(23), Instr(23),
                               Instr(23), Instr(23),
                               Instr(23 downto 0), "00");
      when others  =>  ExtImm  <=  X"--------";
    end case;
  end process;
end;
```

HDL Example 7.9 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;
entity flopr is -- flip-flop with synchronous reset
  generic(width: integer);
  port(clk, reset:  in  STD_LOGIC;
       d:           in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:           out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then  q <= (others => '0');
    elsif rising_edge(clk) then
         q <= d;
    end if;
  end process;
end;
```

HDL Example 7.10 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;
entity flopenr is -- flip-flop with enable and synchronous reset
  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) begin
    if reset then  q <= (others => '0');
    elsif rising_edge(clk) then
      if en then
        q <= d;
      end if;
    end if;
  end process;
end;
```

HDL Example 7.11 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 -- two-input multiplexer
  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;

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

7.6.3 Testbench

The testbench loads a program into the memories. The program in Figure 7.60 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 7 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 hexadecimal file called memfile.dat, which is loaded by the testbench during simulation. The file consists of the machine code for the instructions, one instruction per line. The testbench, top-level ARM module, and external memory HDL code are given in the following examples. The memories in this example hold 64 words each.

```
ADDR        PROGRAM             ; COMMENTS                      BINARY MACHINE CODE                          HEX CODE
00  MAIN    SUB R0, R15, R15    ; R0 = 0                        1110 000 0010 0 1111 0000 0000 0000 1111     E04F000F
04          ADD R2, R0, #5      ; R2 = 5                        1110 001 0100 0 0000 0010 0000 0000 0101     E2802005
08          ADD R3, R0, #12     ; R3 = 12                       1110 001 0100 0 0000 0011 0000 0000 1100     E280300C
0C          SUB R7, R3, #9      ; R7 = 3                        1110 001 0010 0 0011 0111 0000 0000 1001     E2437009
10          ORR R4, R7, R2      ; R4 = 3 OR 5 = 7               1110 000 1100 0 0111 0100 0000 0000 0010     E1874002
14          AND R5, R3, R4      ; R5 = 12 AND 7 = 4             1110 000 0000 0 0011 0101 0000 0000 0100     E0035004
18          ADD R5, R5, R4      ; R5 = 4 + 7 = 11               1110 000 0100 0 0101 0101 0000 0000 0100     E0855004
1C          SUBS R8, R5, R7     ; R8 = 11 - 3 = 8, set Flags    1110 000 0010 1 0101 1000 0000 0000 0111     E0558007
20          BEQ END             ; shouldn't be taken            0000 1010 0000  0000 0000 0000 0000 1100     0A00000C
24          SUBS R8, R3, R4     ; R8 = 12 - 7  = 5              1110 000 0010 1 0011 1000 0000 0000 0100     E0538004
28          BGE AROUND          ; should be taken               1010 1010 0000  0000 0000 0000 0000 0000     AA000000
2C          ADD R5, R0, #0      ; should be skipped             1110 001 0100 0 0000 0101 0000 0000 0000     E2805000
30 AROUND   SUBS R8, R7, R2     ; R8 = 3 - 5 = -2, set Flags    1110 000 0010 1 0111 1000 0000 0000 0010     E0578002
34          ADDLT R7, R5, #1    ; R7 = 11 + 1 = 12              1011 001 0100 0 0101 0111 0000 0000 0001     B2857001
38          SUB R7, R7, R2      ; R7 = 12 - 5 = 7               1110 000 0010 0 0111 0111 0000 0000 0010     E0477002
3C          STR R7, [R3, #84]   ; mem[12+84] = 7                1110 010 1100 0 0011 0111 0000 0101 0100     E5837054
40          LDR R2, [R0, #96]   ; R2 = mem[96] = 7              1110 010 1100 1 0000 0010 0000 0110 0000     E5902060
44          ADD R15, R15, R0    ; PC = PC+8 (skips next)        1110 000 0100 0 1111 1111 0000 0000 0000     E08FF000
48          ADD R2, R0, #14     ; shouldn't happen              1110 001 0100 0 0000 0010 0000 0000 0001     E280200E
4C          B END               ; always taken                  1110 1010 0000  0000 0000 0000 0000 0001     EA000001
50          ADD R2, R0, #13     ; shouldn't happen              1110 001 0100 0 0000 0010 0000 0000 0001     E280200D
54          ADD R2, R0, #10     ; shouldn't happen              1110 001 0100 0 0000 0010 0000 0000 0001     E280200A
58 END      STR R2, [R0, #100]  ; mem[100] = 7                  1110 010 1100 0 0000 0010 0000 0101 0100     E5802064
```

Figure 7.60 Assembly and machine code for test program

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
```

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;
```

```
  // check that 7 gets written to address 0x64
  // at end of program
  always @(negedge clk)
  begin
    if(MemWrite) begin
      if(DataAdr === 100 & WriteData === 7) begin
        $display("Simulation succeeded");
        $stop;
      end else if (DataAdr !== 96) begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

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

  -- check that 7 gets written to address 0x64
  -- 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) = 7) 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
  arm arm(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 -- top-level design for testing
  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 arm
    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
  i_arm: arm port map(clk, reset, PC, Instr, MemWrite, DataAdr,
                      WriteData, ReadData);
  i_imem: imem port map(PC, Instr);
  i_dmem: dmem port map(clk, MemWrite, DataAdr,
                        WriteData, ReadData);
end;
```

HDL Example 7.14 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 -- data memory
  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 clk'event and clk = '1' 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;
```

HDL Example 7.15 INSTRUCTION MEMORY

SystemVerilog

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

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

  initial
    $readmemh("memfile.dat",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;
entity imem is -- instruction memory
  port(a:   in  STD_LOGIC_VECTOR(31 downto 0);
       rd:  out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture behave of imem is -- instruction memory
begin
  process is
    file mem_file: TEXT;
    variable L: line;
    variable ch: character;
    variable i, index, result: integer;
    type ramtype is array (63 downto 0) of
                          STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- initialize memory from file
    for i in 0 to 63 loop -- set all contents low
      mem(i) := (others => '0');
    end loop;
```

```
    index := 0;
    FILE_OPEN(mem_file, "memfile.dat", READ_MODE);
    while not endfile(mem_file) loop
      readline(mem_file, L);
      result := 0;
      for i in 1 to 8 loop
        read(L, ch);
        if '0' <= ch and ch <= '9' then
          result := character'pos(ch) - character'pos('0');
        elsif 'a' <= ch and ch <= 'f' then
          result := character'pos(ch) - character'pos('a') + 10;
        elsif 'A' <= ch and ch <= 'F' then
          result := character'pos(ch) - character'pos('A') + 10;
        else report "Format error on line " & integer'image(index)
          severity error;
        end if;
        mem(index)(35-i*4 downto 32-i*4) :=
          to_std_logic_vector(result,4);
      end loop;
      index := index + 1;
    end loop;

    -- read memory
    loop
      rd <= mem(to_integer(a(7 downto 2)));
      wait on 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 speed-up techniques. The implementation details become quite complex, so we focus on the concepts. Hennessy & 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 is now 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

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 10–20 stages are now commonly used.

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). This sequencing overhead makes adding more pipeline stages give diminishing returns. Finally, adding more stages increases the cost because of the extra pipeline registers and hardware required to handle hazards.

Example 7.9

Consider building a pipelined processor by chopping up the single-cycle processor into N stages. The single-cycle processor has a propagation delay of 740 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 7.7 has a CPI of 1.23. 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 $T_c = (740/N + 90)$ ps. The CPI is $1.23 + 0.1(N–5)$. The time per instruction, or instruction time, is the product of the cycle time and the CPI. Figure 7.61 plots the cycle time and instruction time versus the number of stages. The instruction time has a minimum of 279 ps at $N = 8$ stages. This minimum is only slightly better than the 293 ps per instruction achieved with a five-stage pipeline.

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

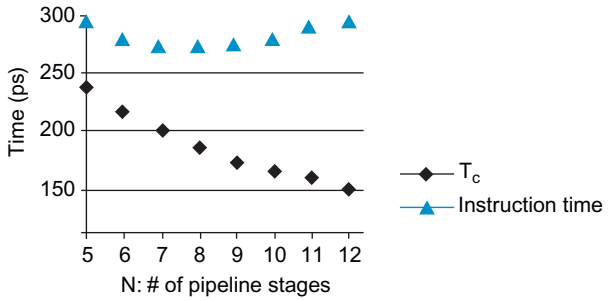


Figure 7.61 Cycle time and instruction time vs. the number of pipeline stages

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 MIPS 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, adding the base, displacement, and scaled index, 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 on more common, simpler instructions.

Computer architects 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 real instruction is decoded into one or more micro-ops. For example, if we defined *μops* resembling basic ARM instructions and some temporary registers T1 and T2 for holding intermediate results, then the x86 instruction could become seven *μops*:

```
ADD  T1, [EDX + 80] ; T1 <– EDX + 80
LSL  T2, EDI, 2     ; T2 <– EDI*2
ADD  T1, T2, T2     ; T1 <– EDX + 80 + EDI*2
LDR  T1, [T1]       ; T1 <– MEM[EDX + 80 + EDI*2]
LDR  T2, [ESP]      ; T2 <– MEM[ESP]
ADD  T1, T2, T1     ; T1 <– MEM[ESP] + MEM[EDX + 80 + EDI*2]
STR  T1, [ESP]      ; MEM[ESP]<– MEM[ESP] + MEM[EDX + 80 + EDI*2]
```

Although most ARM instructions are simple, some are decomposed into multiple micro-ops as well. For example, loads with postindexed addressing (such as `LDR R1, [R2], #4`) require a second write port on the register file. Data-processing instructions with register-shifted register addressing (such as `ORR R3, R4, R5, LSL R6`) require a third read port on the register file. Instead of providing a larger five-port register file,

the ARM datapath may decode these complex instructions into pairs of simpler instructions:

Complex Op	Micro-op Sequence
`LDR R1, [R2], #4`	`LDR R1, [R2]` `ADD R2, R2, #4`
`ORR R3, R4, R5 LSL R6`	`LSL T1, R5, R6` `ORR R3, R4, T1`

Although the programmer could have written the simpler instructions directly and the program may have run just as fast, a single complex instruction takes less memory than the pair of simpler instructions. Reading instructions from external memory can consume significant power, so the complex instruction also can save power. The ARM instruction set is so successful in part because of the architects' judicious choice of instructions that give better code density than pure RISC instructions sets such as MIPS, yet more efficient decoding than CISC instruction sets such as x86.

Microarchitects make the decision of whether to provide hardware to implement a complex operation directly or 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.

7.7.3 Branch Prediction

An ideal pipelined processor would have a CPI of 1.0. 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 when a program reaches the end of a loop and branches back to repeat the loop (e.g., in a for or while loop). Loops tend to be executed many times, so these backward branches are usually taken. The simplest form of branch prediction checks the direction of the branch and predicts that backward branches should be taken. This is called *static branch prediction*, because it does not depend on the history of the program.

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.

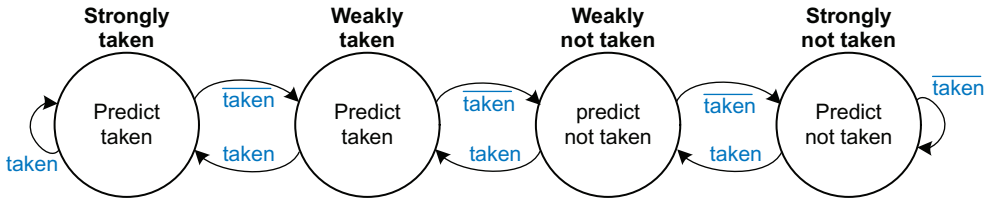


Figure 7.62 Two-bit branch predictor state transition diagram

To see the operation of dynamic branch predictors, consider the following loop from Code Example 6.17. The loop repeats 10 times, and the `BGE` out of the loop is taken only on the last iteration.

```
    MOV  R1, #0
    MOV  R0, #0
FOR
    CMP  R0, #10
    BGE  DONE
    ADD  R1, R1, R0
    ADD  R0, R0, #1
    B    FOR
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 `BGE` 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 7.62. 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 re-enters the "strongly not taken" state. In summary, a two-bit branch predictor mispredicts only the last branch of a loop.

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 Processor

A *superscalar processor* contains multiple copies of the datapath hardware to execute multiple instructions simultaneously. Figure 7.63 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 7.64 shows 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 7.65 shows a pipeline diagram running a program with data dependencies. The dependencies in the code are shown in blue. The `ADD` instruction is dependent on R8, which is produced by the `LDR` instruction, so it cannot be issued at the same time as `LDR`. The `ADD` instruction stalls for yet another cycle so that `LDR` can forward R8 to `ADD` in cycle 5. The other dependencies (between `SUB` and

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.

Our ARM pipelined processor is a scalar processor. Vector processors were popular for supercomputers in the 1980s and 1990s because they efficiently handled the long vectors of data common in scientific computations, and they are heavily used now in *graphics processing units* (*GPUs*). Modern high-performance microprocessors are superscalar, because issuing several independent instructions is more flexible than processing vectors.

However, modern processors also include hardware to handle short vectors of data that are common in multimedia and graphics applications. These are called *single instruction multiple data* (*SIMD*) units and are discussed in Section 6.7.5.

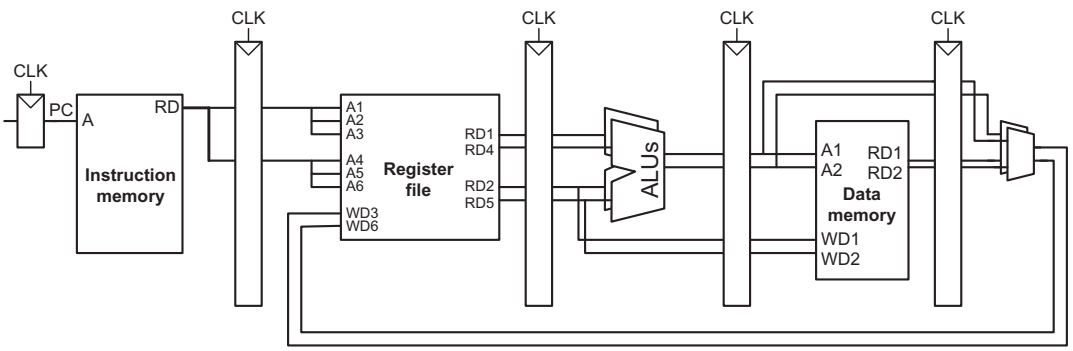


Figure 7.63 Superscalar datapath

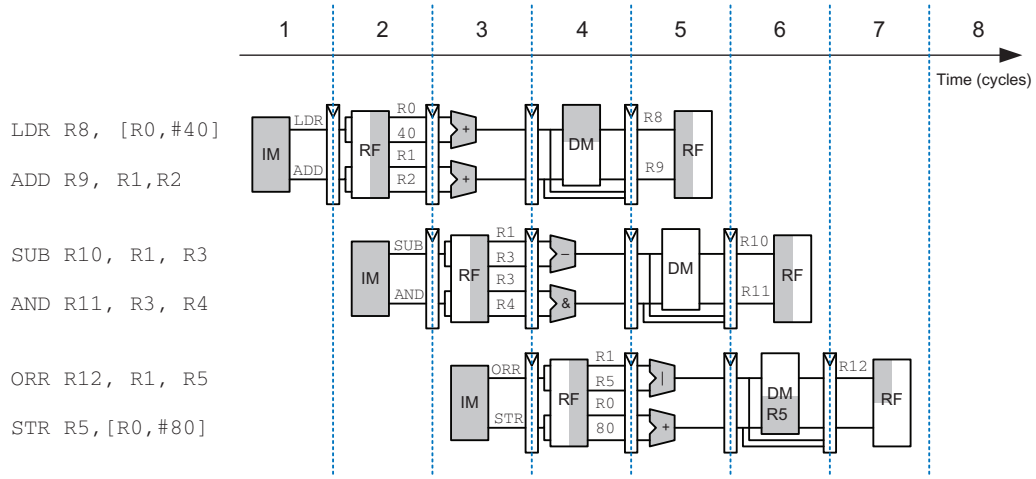


Figure 7.64 Abstract view of a superscalar pipeline in operation

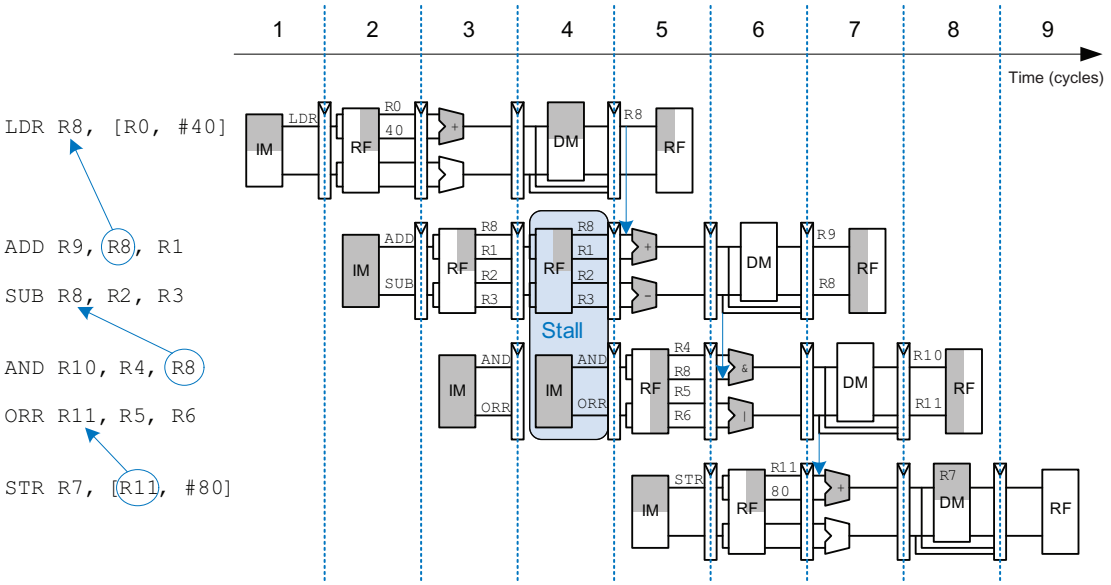


Figure 7.65 Program with data dependencies

AND based on R8, and between ORR and STR based on R11) 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 1.2.

Recall that parallelism comes in temporal and spatial forms. Pipelining is a case of temporal parallelism. 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.

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*, or begin executing, 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.

Consider running the same program from Figure 7.65 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 7.66 shows the data dependencies and the

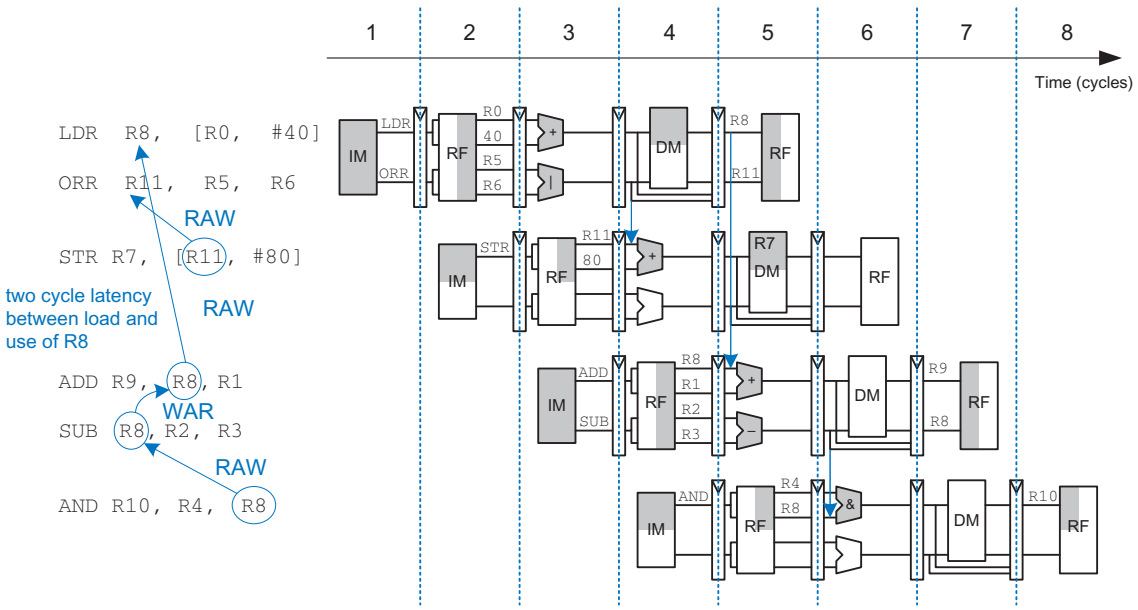


Figure 7.66 Out-of-order execution of a program with dependencies

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 LDR instruction issues.
  - The ADD, SUB, and AND instructions are dependent on LDR by way of R8, so they cannot issue yet. However, the ORR instruction is independent, so it also issues.
- Cycle 2
  - Remember that there is a two-cycle latency between issuing an LDR instruction and a dependent instruction, so ADD cannot issue yet because of the R8 dependence. SUB writes R8, so it cannot issue before ADD, lest ADD receive the wrong value of R8. AND is dependent on SUB.
  - Only the STR instruction issues.
- Cycle 3
  - On cycle 3, R8 is available, so the ADD issues. SUB issues simultaneously, because it will not write R8 until after ADD consumes R8.
- Cycle 4
  - The AND instruction issues. R8 is forwarded from SUB to AND.

The out-of-order processor issues the six instructions in four cycles, for an IPC of 1.5.

The dependence of ADD on LDR by way of R8 is a *read after write* (*RAW*) hazard. ADD must not read R8 until after LDR 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 STR on ORR by way of R11 and of AND on SUB by way of R8 are RAW dependencies.

The dependence between SUB and ADD by way of R8 is called a *write after read* (*WAR*) hazard or an *antidependence*. SUB must not write R8 before ADD reads R8, 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 R12 instead of R8, then the dependency would disappear and SUB could be issued before ADD. The ARM architecture only has 16 registers, so sometimes the programmer is forced to reuse a register and introduce a hazard just because all 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, `LDR` and `ADD` both write R8. The final value in R8 should come from `ADD` according to the order of the program. If an out-of-order processor attempted to execute `ADD` first, then a WAW hazard would occur.

```
LDR  R8,  [R3]
ADD  R8,  R1,  R2
```

WAW hazards are not essential either; again, they are artifacts caused by the programmer's 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 `LDR` instead of writing it to R8. This is called *squashing* the `LDR`.[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 T0–T19. The

[4] You might wonder why the `LDR` 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 `LDR` potentially may produce a Data Abort exception, so it must be issued to check for the exception, even though the result can be discarded.

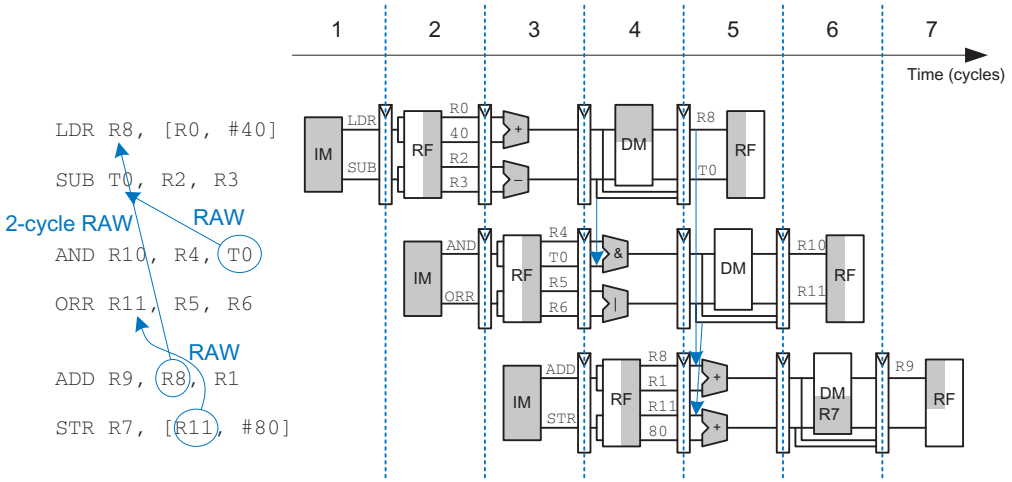


Figure 7.67 Out-of-order execution of a program using register renaming

programmer cannot 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 and ADD instructions based on reusing R8. The out-of-order processor could rename R8 to T0 for the SUB instruction. Then, SUB could be executed sooner, because T0 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, R8 must also be renamed to T0 in the AND instruction, because it refers to the result of SUB.

Figure 7.67 shows the same program from Figure 7.65 executing on an out-of-order processor with register renaming. R8 is renamed to T0 in SUB and AND to eliminate the WAR hazard. The constraints on issuing instructions are:

- Cycle 1
  - The LDR instruction issues.
  - The ADD instruction is dependent on LDR by way of R8, so it cannot issue yet. However, the SUB instruction is independent now that its destination has been renamed to T0, so SUB also issues.
- Cycle 2
  - Remember that there is a two-cycle latency between issuing an LDR instruction and a dependent instruction, so ADD cannot issue yet because of the R8 dependence.

  - The AND instruction is dependent on SUB, so it can issue. T0 is forwarded from SUB to AND.
  - The ORR instruction is independent, so it also issues.
- Cycle 3
  - On cycle 3, R8 is available, so the ADD issues.
  - R11 is also available, so STR 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 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 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 the threads fast enough, the user perceives all of the threads as running at the same time.

A 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 64 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 the execution units busy in a superscalar design, then another thread could issue instructions to the idle units.

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 is wasteful of 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. As of 2015, consumer applications used few threads at any given time, and a typical consumer might be expected to have 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 different microarchitectures that have different power, performance, and area trade-offs.

One heterogeneous strategy popularized by ARM is *big.LITTLE*, in which a system contains both energy-efficient and high-performance cores. "LITTLE" cores such as the Cortex-A53 are single-issue or dual-issue in-order processors with good energy efficiency that handle routine tasks. "big" cores such as the Cortex-A57 are more complex superscalar out-of-order cores delivering high performance for peak loads.

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 10–100x more efficient 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

Scientists searching for signs of extraterrestrial intelligence use the world's largest clustered multiprocessors to analyze radio telescope data for patterns that might be signs of life in other solar systems. The cluster, operational since 1999, consists of personal computers owned by more than 6 million volunteers around the world.

When a computer in the cluster is idle, it fetches a piece of the data from a centralized server, analyzes the data, and sends the results back to the server. You can volunteer your computer's idle time for the cluster by visiting `setiathome.berkeley.edu`.

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 cases that have more varying or special-purpose workloads.

Clusters

In *clustered* multiprocessors, each processor has its own local memory system. One type of cluster is a group of personal computers connected together on the network running software to jointly solve a large problem. Another type of cluster that has become very important is the *data center*, in which racks of computers and disks are networked together and share power and cooling. Major Internet companies including Google, Amazon, and Facebook have driven the rapid development of data centers to support millions of users around the world.

7.8 REAL-WORLD PERSPECTIVE: EVOLUTION OF ARM MICROARCHITECTURE*

DMIPS (Dhrystone millions of instructions per second) measures performance.

This section traces the development of the ARM architecture and microarchitecture since its inception in 1985. Table 7.7 summarizes the highlights, showing 10x improvement in IPC and 250x increase in

Table 7.7 Evolution of ARM processors

Microarchitecture	Year	Architecture	Pipeline Depth	DMIPS/MHz	Representative Frequency (MHz)	L1 Cache	Relative Size
ARM1	1985	v1	3	0.33	8	N/A	0.1
ARM6	1992	v3	3	0.65	30	4 KB unified	0.6
ARM7	1994	v4T	3	0.9	100	0–8 KB unified	1
ARM9E	1999	v5TE	5	1.1	300	0–16 KB I + D	3
ARM11	2002	v6	8	1.25	700	4–64 KB I + D	30
Cortex-A9	2009	v7	8	2.5	1000	16–64 KB I + D	100
Cortex-A7	2011	v7	8	1.9	1500	8–64 KB I + D	40
Cortex-A15	2011	v7	15	3.5	2000	32 KB I + D	240
Cortex-M0 +	2012	v7M	2	0.93	60–250	None	0.3
Cortex-A53	2012	v8	8	2.3	1500	8–64 KB I + D	50
Cortex-A57	2012	v8	15	4.1	2000	48 KB I + 32 KB D	300

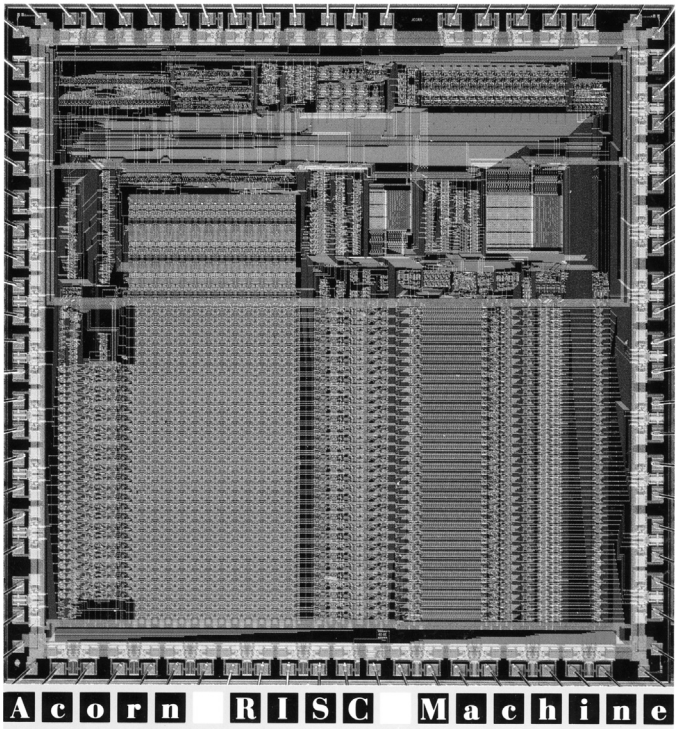


Figure 7.68 ARM1 die photograph
(Reproduced with permission from ARM. © 1985 ARM Ltd.)

frequency over three decades and eight revisions of the architecture. Frequency, area, and power will vary with manufacturing process and the goals, schedule, and capabilities of the design team. The representative frequencies are quoted for a fabrication process at the time of product introduction, so much of the frequency gain comes from transistors rather than microarchitecture. The relative size is normalized by the transistor feature size and can vary widely depending on cache size and other factors.

Figure 7.68 shows a die photograph of the ARM1 processor, which contained 25,000 transistors in a three-stage pipeline. If you count carefully, you can observe the 32 bits of the datapath at the bottom. The register file is on the left and the ALU is on the right. At the very left is the program counter; observe that the two least significant bits at the bottom are empty (tied to 0) and the six at the top are different because they are used for status bits. The controller sits on top of the datapath. Some of the rectangular blocks are PLAs implementing control logic. The rectangles around the edge are I/O pads, with tiny gold bond wires visible leading out of the picture.

In 1990, Acorn spun off the processor design team to establish a new company, Advanced RISC Machines (later named ARM Holdings), which began licensing the ARMv3 architecture. The ARMv3 architecture moved the status bits from the PC to the Current Program Status Register and extended the PC to 32 bits. Apple bought a major stake in ARM and used the ARM 610 in the Newton computer, the world's first Personal Digital Assistant (PDA) and one of the first commercial applications of handwriting recognition. Newton proved to be ahead of its time, but it laid the foundation for more successful PDAs and later for smart phones and tablets.

ARM achieved huge success with the ARM7 line in 1994, especially the ARM7TDMI, which became one of the mostly widely used RISC processors in embedded systems over the next 15 years. The ARM7TDMI used the ARMv4T instruction set, which introduced the Thumb instruction set for better code density and defined halfword and signed byte load and store instructions. TDMI stood for **T**humb, JTAG **D**ebug, fast **M**ultiply, and **I**n-Circuit Debug. The various debug features help programmers write code on the hardware and test it from a PC using a simple cable, an important advance at the time. ARM7 used a simple three-stage pipeline with Fetch, Decode, and Execute stages. The processor had a unified cache containing both instructions and data. Because the cache in a pipelined processor is usually busy every cycle fetching instructions, ARM7 stalled memory instructions in the Execute stage to make time for the cache to access the data. Figure 7.69 shows a block diagram of the processor. Rather than manufacturing a chip directly, ARM licensed the processor to other companies that put them into their larger system-on-chip (SoC). Customers could buy the processor as a hard macro (a complete and efficient but inflexible layout that could be dropped directly into a chip) or as a soft macro (Verilog code that could be synthesized by the customer). The ARM7 was used in a vast number of products, including mobile phones, the Apple iPod, Lego Mindstorms NXT, Nintendo game machines, and automobiles. Since then, nearly all mobile phones have been built around ARM processors.

The ARM9E line improved on ARM7 with a five-stage pipeline similar to the one described in this chapter, separate instruction and data caches, and new Thumb and digital signal processing instructions in the ARMv5TE architecture. Figure 7.70 shows a block diagram of the ARM9 containing many of the same components as we encountered in this chapter but adding the multiplier and shifter. The IA/ID/DA/DD signals are the Instruction and Data Address and Data busses to the memory system, and the IAreg is the PC. The next-generation ARM11 extended the pipeline further to eight stages to boost frequency and defined Thumb2 and SIMD instructions.

The ARMv7 instruction set added Advanced SIMD instructions operating on double- and quad-word registers. It also defined a v7-M variant

Sophie Wilson and Steve Furber together designed the ARM1.

Sophie Wilson (1957–) was born in Yorkshire, England, and studied Computer Science at the University of Cambridge. She designed the operating system and wrote the BBC Basic Interpreter for Acorn Computer, and then codesigned the ARM1 and subsequent processors through the ARM7. By 1999, she designed the Firepath SIMD digital signal processor and spun it off as a new company, which Broadcom acquired in 2001. She is presently a Senior Director at Broadcom Corporation and a Fellow of the Royal Society, the Royal Academy of Engineering, the British Computer Society, and the Women's Engineering Society.

(Photograph © Sophie Wilson. Reproduced with permission.)

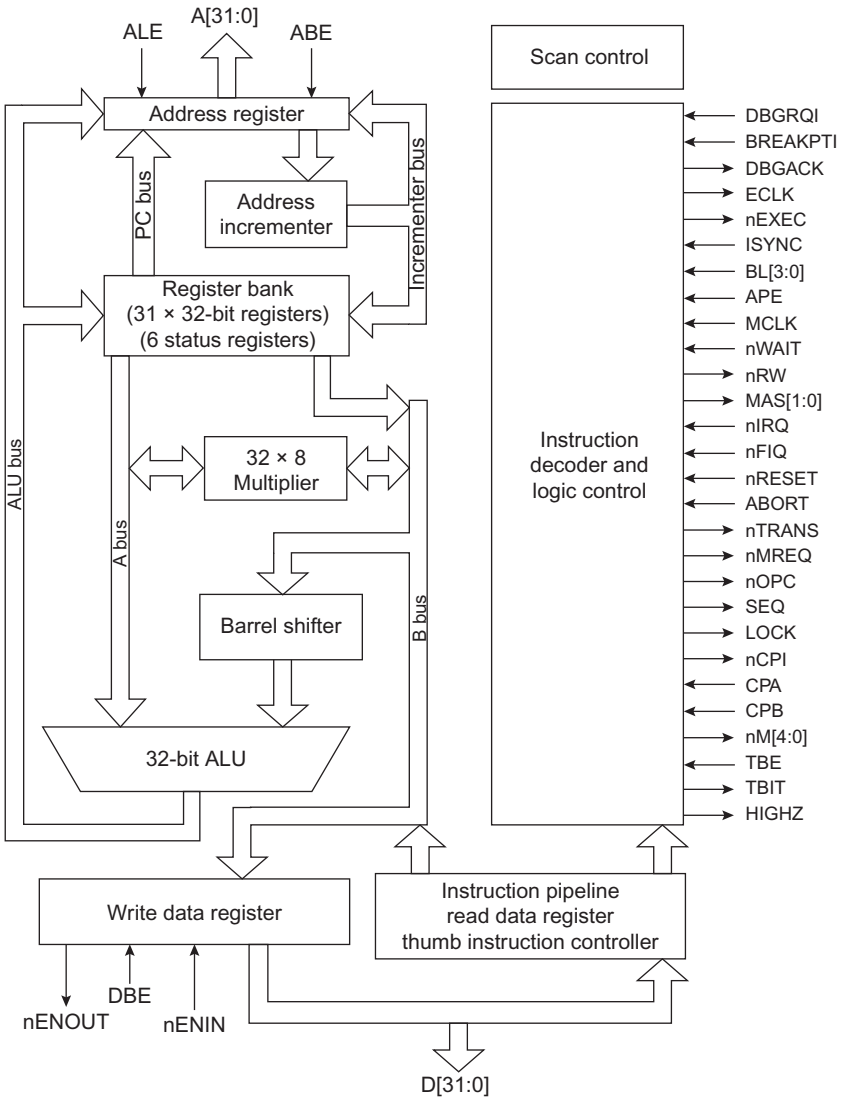


Figure 7.69 **ARM7 block diagram** (Reproduced with permission from ARM. © 1998 ARM Ltd.)

supporting only Thumb instructions. ARM introduced the Cortex-A and Cortex-M families of processors. The Cortex-A family of high-performance processors are now used in virtually all smart phones and tablets. The Cortex-M family, running the Thumb instruction set, are tiny and inexpensive microcontrollers used in embedded systems. For example, the Cortex-M0+ uses a two-stage pipeline and only 12,000 gates, compared with hundreds of thousands in an A-series processor. It costs well under a dollar as a stand-alone chip, or under a penny when integrated

Steve Furber (1953–) was born in Manchester, England, and received a PhD in aerodynamics from the University of Cambridge. He joined Acorn Computer, where he codesigned the BBC Micro and ARM1 microprocessor for Acorn Computer. In 1990, he joined the faculty of the University of Manchester, where his research has focused on asynchronous computing and neural systems.

(Photograph © 2012 The University of Manchester. Reproduced with permission.)

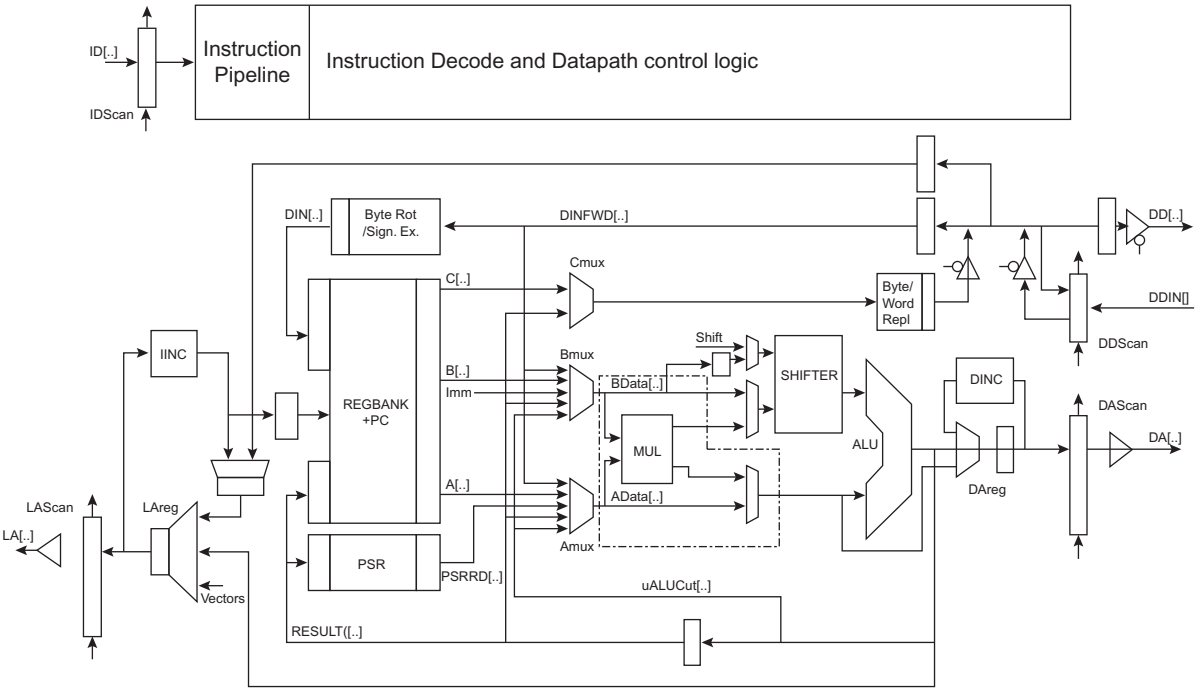


Figure 7.70 ARM9 block diagram

(Reproduced with permission from the ARM9TDMI Technical Reference Manual. © 1999 ARM Ltd.)

on a larger SoC. The power consumption is roughly 3 μW/MHz, so the processor powered by a watch battery could run continuously for nearly a year at 10 MHz.

Higher-end ARMv7 processors captured the cell phone and tablet markets. The Cortex-A9 was widely used in mobile phones, often as part of a dual-core SoC containing two Cortex-A9 processors, a graphics accelerator, a cellular modem, and other peripherals. Figure 7.71 shows a block diagram of the Cortex-A9. The processor decodes two instructions per cycle, performs register renaming, and issues them to out-of-order execution units.

Energy efficiency and performance are both critical for mobile devices, so ARM has been promoting the big.LITTLE architecture combining several high-performance "big" cores for peak workloads with energy-efficient "LITTLE" cores that handle most routine processes. For example, the Samsung Exynos 5 Octa in the Galaxy S5 phone contains four Cortex-A15 big cores running up to 2.1 GHz and four Cortex-A7 LITTLE cores running at up to 1.5 GHz. Figure 7.72 shows pipeline diagrams for the two types of cores. The Cortex-A7 is an in-order processor that can decode and issue up to one memory instruction and

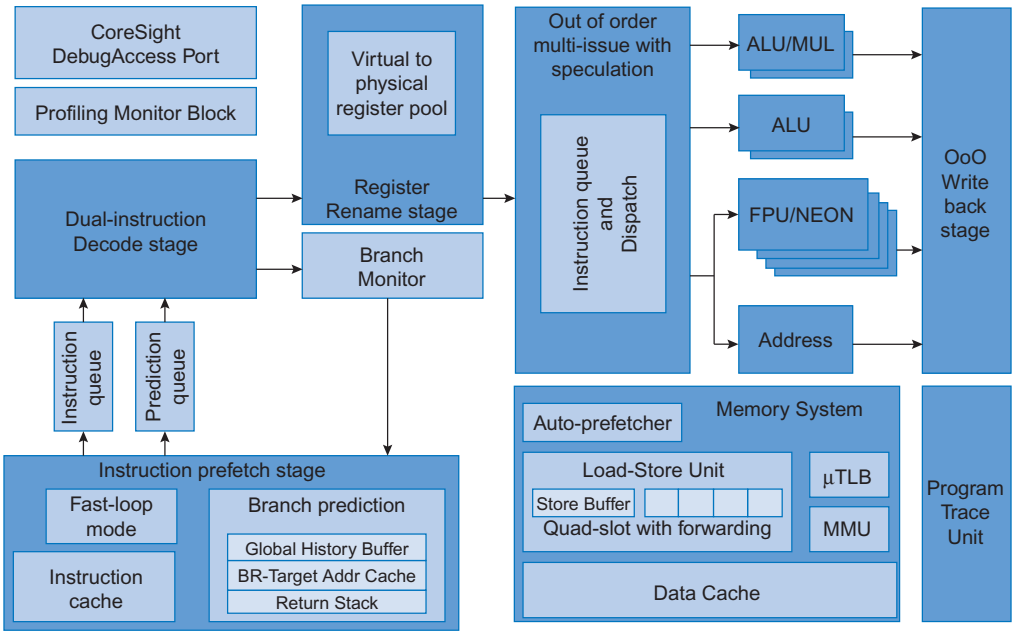


Figure 7.71 **Cortex-A9 block diagram**
(This image has been sourced by the authors and does not imply ARM endorsement.)

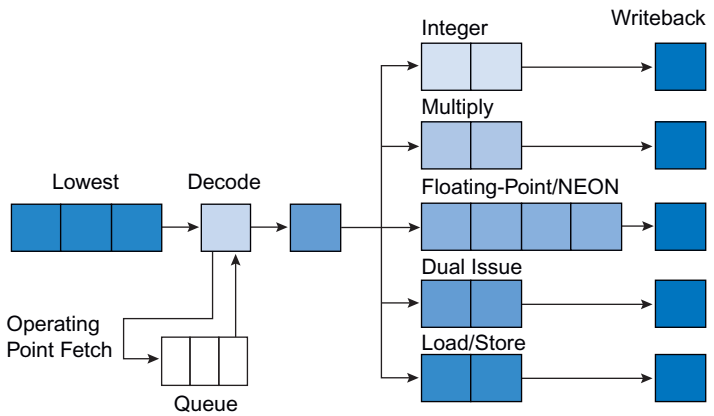
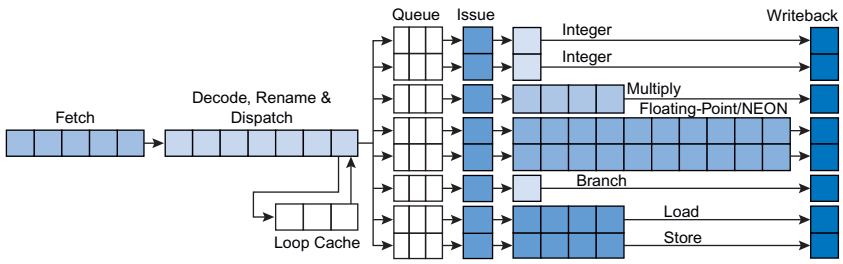


Figure 7.72 **Cortex-A7 and -A15 block diagrams**
(This image has been sourced by the authors and does not imply ARM endorsement.)



one other instruction each cycle. The Cortex-A15 is a much more complex out-of-order processor that can decode up to three instructions each cycle. The pipeline length almost doubles to handle the complexity and boost clock speed, so a more accurate branch predictor is necessary to compensate for the larger branch misprediction penalty. The Cortex-A15 delivers approximately 2.5x the performance of the Cortex-A7, but at 6x the power. Smart phones can only run the big cores briefly before the chip will begin to overheat and throttle itself back.

The ARMv8 architecture is a streamlined 64-bit architecture. ARM's Cortex-A53 and -A57 have pipelines similar to the Cortex-A7 and -A15, respectively, but boost the registers and datapaths to 64 bits to handle ARMv8. Apple popularized the 64-bit architecture in 2013, when it introduced its own implementation in the iPhone and iPad.

7.9 SUMMARY

This chapter has described three ways to build processors, each with different performance 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 2 and 3), the application of many of the building blocks (described in Chapter 5), and the implementation of the ARM 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 used 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. Each of these units is built from logic blocks, which can be built from gates, which in turn can be built from transistors using the techniques developed in the first five chapters.