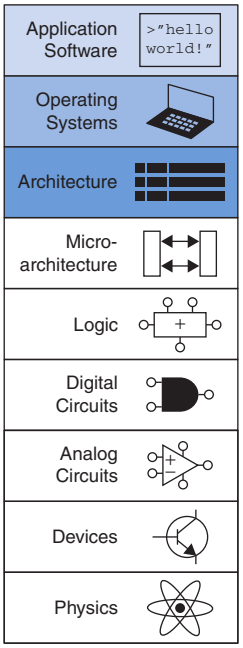
Architecture

6



6.1 INTRODUCTION

The previous chapters introduced digital design principles and building blocks. In this chapter, we jump up a few levels of abstraction to define the architecture of a computer. The *architecture* is the programmer's view of a computer. It is defined by the instruction set (language) and operand locations (registers and memory). Many different architectures exist, such as ARM, x86, MIPS, SPARC, and PowerPC.

The first step in understanding any computer architecture is to learn its language. The words in a computer's language are called *instructions*. The computer's vocabulary is called the *instruction set*. All programs running on a computer use the same instruction set. Even complex software applications, such as word processing and spreadsheet applications, are eventually compiled into a series of simple instructions such as add, subtract, and branch. Computer instructions indicate both the operation to perform and the operands to use. The operands may come from memory, from registers, or from the instruction itself.

Computer hardware understands only 1's and 0's, so instructions are encoded as binary numbers in a format called machine language. Just as we use letters to encode human language, computers use binary numbers to encode machine language. The ARM architecture represents each instruction as a 32-bit word. Microprocessors are digital systems that read and execute machine language instructions. However, humans consider reading machine language to be tedious, so we prefer to represent the instructions in a symbolic format called assembly language.

The instruction sets of different architectures are more like different dialects than different languages. Almost all architectures define basic instructions, such as add, subtract, and branch, that operate on memory or registers. Once you have learned one instruction set, understanding others is fairly straightforward.

A computer architecture does not define the underlying hardware implementation. Often, many different hardware implementations of a single architecture exist. For example, Intel and Advanced Micro Devices (AMD) both sell various microprocessors belonging to the same x86 architecture. They all can run the same programs, but they use different underlying hardware and therefore offer trade-offs in performance, price, and power. Some microprocessors are optimized for high-performance servers, whereas others are optimized for long battery life in laptop computers. The specific arrangement of registers, memories, ALUs, and other building blocks to form a microprocessor is called the microarchitecture and will be the subject of Chapter 7. Often, many different microarchitectures exist for a single architecture.

In this text, we introduce the ARM architecture. This architecture was first developed in the 1980s by Acorn Computer Group, which spun off Advanced RISC Machines Ltd., now known as ARM. Over 10 billion ARM processors are sold every year. Almost all cell phones and tablets contain multiple ARM processors. The architecture is used in everything from pinball machines to cameras to robots to cars to rack-mounted servers. ARM is unusual in that it does not sell processors directly, but rather licenses other companies to build its processors, often as part of a larger system-on-chip. For example, Samsung, Altera, Apple, and Qualcomm all build ARM processors, either using microarchitectures purchased from ARM or microarchitectures developed internally under license from ARM. We choose to focus on ARM because it is a commercial leader and because the architecture is clean, with few idiosyncrasies. We start by introducing assembly language instructions, operand locations, and common programming constructs, such as branches, loops, array manipulations, and function calls. We then describe how the assembly language translates into machine language and show how a program is loaded into memory and executed.

The "ARM architecture" we describe is ARM version 4 (ARMv4), which forms the core of the instruction set. Section 6.7 summarizes new features in versions 5–8 of the architecture. The ARM *Architecture Reference Manual* (ARM), available online, is the authoritative definition of the architecture.

Throughout the chapter, we motivate the design of the ARM architecture using four principles articulated by David Patterson and John Hennessy in their text *Computer Organization and Design*: (1) regularity supports simplicity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises.

6.2 ASSEMBLY LANGUAGE

Assembly language is the human-readable representation of the computer's native language. Each assembly language instruction specifies both the operation to perform and the operands on which to operate. We introduce simple arithmetic instructions and show how these operations are written in assembly language. We then define the ARM instruction operands: registers, memory, and constants.

This chapter assumes that you already have some familiarity with a high-level programming language such as C, C++, or Java.

(These languages are practically identical for most of the examples in this chapter, but where they differ, we will use C.) Appendix C provides an introduction to C for those with little or no prior programming experience.

We used Keil's ARM Microcontroller Development Kit (MDK-ARM) to compile, assemble, and simulate the example assembly code in this chapter. The MDK-ARM is a free development tool that comes with a complete ARM compiler. Labs available on this textbook's companion site (see Preface) show how to install and use this tool to write, compile, simulate, and debug both C and assembly programs.

6.2.1 Instructions

The most common operation computers perform is addition. Code Example 6.1 shows code for adding variables `b` and `c` and writing the result to `a`. The code is shown on the left in a high-level language (using the syntax of C, C++, and Java) and then rewritten on the right in ARM assembly language. Note that statements in a C program end with a semicolon.

Code Example 6.1 ADDITION

High-Level Code	ARM Assembly Code
`a = b + c;`	`ADD a, b, c`

The first part of the assembly instruction, `ADD`, is called the *mnemonic* and indicates what operation to perform. The operation is performed on `b` and `c`, the *source operands*, and the result is written to `a`, the *destination operand*.

Mnemonic (pronounced ni-mon-ik) comes from the Greek word μιμνΕσκεσταιι, to remember. The assembly language mnemonic is easier to remember than a machine language pattern of 0's and 1's representing the same operation.

Code Example 6.2 SUBTRACTION

High-Level Code	ARM Assembly Code
`a = b - c;`	`SUB a, b, c`

Code Example 6.2 shows that subtraction is similar to addition. The instruction format is the same as the `ADD` instruction except for the operation specification, `SUB`. This consistent instruction format is an example of the first design principle:

Design Principle 1: Regularity supports simplicity.

Instructions with a consistent number of operands—in this case, two sources and one destination—are easier to encode and handle in hardware. More complex high-level code translates into multiple ARM instructions, as shown in Code Example 6.3.

In the high-level language examples, single-line comments begin with `//` and continue until the end of the line. Multiline comments begin with `/*` and end with `*/`. In ARM assembly language, only single-line comments

Code Example 6.3 MORE COMPLEX CODE

High-Level Code

```
a = b + c - d;    // single-line comment
                  /* multiple-line
                     comment */
```

ARM Assembly Code

```
ADD t, b, c  ; t = b + c
SUB a, t, d  ; a = t - d
```

are used. They begin with a semicolon (`;`) and continue until the end of the line. The assembly language program in Code Example 6.3 requires a temporary variable `t` to store the intermediate result. Using multiple assembly language instructions to perform more complex operations is an example of the second design principle of computer architecture:

Design Principle 2: Make the common case fast.

The ARM instruction set makes the common case fast by including only simple, commonly used instructions. The number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small, and fast. More elaborate operations that are less common are performed using sequences of multiple simple instructions. Thus, ARM is a *reduced instruction set computer* (RISC) architecture. Architectures with many complex instructions, such as Intel's x86 architecture, are *complex instruction set computers* (CISC). For example, x86 defines a "string move" instruction that copies a string (a series of characters) from one part of memory to another. Such an operation requires many, possibly even hundreds, of simple instructions in a RISC machine. However, the cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions.

A RISC architecture minimizes the hardware complexity and the necessary instruction encoding by keeping the set of distinct instructions small. For example, an instruction set with 64 simple instructions would need $\log_2 64 = 6$ bits to encode the operation. An instruction set with 256 complex instructions would need $\log_2 256 = 8$ bits of encoding per instruction. In a CISC machine, even though the complex instructions may be used only rarely, they add overhead to all instructions, even the simple ones.

6.2.2 Operands: Registers, Memory, and Constants

An instruction operates on *operands*. In Code Example 6.1, the variables `a`, `b`, and `c` are all operands. But computers operate on 1's and 0's, not variable names. The instructions need a physical location from which to retrieve the binary data. Operands can be stored in registers or memory, or they may be constants stored in the instruction itself. Computers use

various locations to hold operands in order to optimize for speed and data capacity. Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data. Additional data must be accessed from memory, which is large but slow. ARM (prior to ARMv8) is called a 32-bit architecture because it operates on 32-bit data.

Version 8 of the ARM architecture has been extended to 64 bits, but we will focus on the 32-bit version in this book.

Registers

Instructions need to access operands quickly so that they can run fast. But operands stored in memory take a long time to retrieve. Therefore, most architectures specify a small number of registers that hold commonly used operands. The ARM architecture uses 16 registers, called the *register set* or *register file*. The fewer the registers, the faster they can be accessed. This leads to the third design principle:

Design Principle 3: Smaller is faster.

Looking up information from a small number of relevant books on your desk is a lot faster than searching for the information in the stacks at a library. Likewise, reading data from a small register file is faster than reading it from a large memory. A register file is typically built from a small SRAM array (see Section 5.5.3).

Code Example 6.4 shows the `ADD` instruction with register operands. ARM register names are preceded by the letter 'R'. The variables `a`, `b`, and `c` are arbitrarily placed in R0, R1, and R2. The name R1 is pronounced "register 1" or "R1" or "register R1". The instruction adds the 32-bit values contained in R1 (`b`) and R2 (`c`) and writes the 32-bit result to R0 (`a`). Code Example 6.5 shows ARM assembly code using a register, R4, to store the intermediate calculation of `b + c`:

Code Example 6.4 REGISTER OPERANDS

High-Level Code	ARM Assembly Code
`a = b + c;`	`; R0 = a, R1 = b, R2 = c` `ADD R0, R1, R2 ; a = b + c`

Code Example 6.5 TEMPORARY REGISTERS

High-Level Code	ARM Assembly Code

`a = b + c - d;`	`; R0 = a, R1 = b, R2 = c, R3 = d; R4 = t`
	`ADD R4, R1, R2 ; t = b + c`
	`SUB R0, R4, R3 ; a = t - d`

Example 6.1 TRANSLATING HIGH-LEVEL CODE TO ASSEMBLY LANGUAGE

Translate the following high-level code into ARM assembly language. Assume variables a–c are held in registers R0–R2 and f–j are in R3–R7.

```
a = b - c;
f = (g + h) - (i + j);
```

Solution: The program uses four assembly language instructions.

```
; ARM assembly code
; R0 = a, R1 = b, R2 = c, R3 = f, R4 = g, R5 = h, R6 = i, R7 = j
  SUB R0, R1, R2     ; a = b - c
  ADD R8, R4, R5     ; R8 = g + h
  ADD R9, R6, R7     ; R9 = i + j
  SUB R3, R8, R9     ; f = (g + h) - (i + j)
```

The Register Set

Table 6.1 lists the name and use for each of the 16 ARM registers. R0–R12 are used for storing variables; R0–R3 also have special uses during procedure calls. R13–R15 are also called SP, LR, and PC, and they will be described later in this chapter.

Constants/Immediates

In addition to register operations, ARM instructions can use constant or *immediate* operands. These constants are called immediates, because their values are immediately available from the instruction and do not require a register or memory access. Code Example 6.6 shows the ADD instruction adding an immediate to a register. In assembly code, the immediate is preceded by the # symbol and can be written in decimal or hexadecimal. Hexadecimal constants in ARM assembly language start with 0x, as they

Table 6.1 ARM register set

Name	Use
R0	Argument / return value / temporary variable
R1–R3	Argument / temporary variables
R4–R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Code Example 6.6 IMMEDIATE OPERANDS

High-Level Code	ARM Assembly Code

`a = a + 4;`
`b = a - 12;`
	`; R7 = a, R8 = b`
	`ADD R7, R7, #4 ; a = a + 4`
	`SUB R8, R7, #0xC ; b = a - 12`

Code Example 6.7 INITIALIZING VALUES USING IMMEDIATES

High-Level Code	ARM Assembly Code

`i = 0;`
`x = 4080;`
	`; R4 = i, R5 = x`
	`MOV R4, #0 ; i = 0`
	`MOV R5, #0xFF0 ; x = 4080`

do in C. Immediates are unsigned 8- to 12-bit numbers with a peculiar encoding described in Section 6.4.

The move instruction (`MOV`) is a useful way to initialize register values. Code Example 6.7 initializes the variables `i` and `x` to 0 and 4080, respectively. MOV can also take a register source operand. For example, `MOV R1, R7` copies the contents of register R7 into R1.

Memory

If registers were the only storage space for operands, we would be confined to simple programs with no more than 15 variables. However, data can also be stored in memory. Whereas the register file is small and fast, memory is larger and slower. For this reason, frequently used variables are kept in registers. In the ARM architecture, instructions operate exclusively on registers, so data stored in memory must be moved to a register before it can be processed. By using a combination of memory and registers, a program can access a large amount of data fairly quickly. Recall from Section 5.5 that memories are organized as an array of data words. The ARM architecture uses 32-bit memory addresses and 32-bit data words.

ARM uses a *byte-addressable* memory. That is, each byte in memory has a unique address, as shown in Figure 6.1(a). A 32-bit word consists of four 8-bit bytes, so each word address is a multiple of 4. The most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. Both the 32-bit word address and the data value in Figure 6.1(b) are given in hexadecimal. For example, data word 0xF2F1AC07 is stored at memory address 4. By convention, memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top.

ARM provides the *load register* instruction, `LDR`, to read a data word from memory into a register. Code Example 6.8 loads memory word 2 into `a` (R7). In C, the number inside the brackets is the *index* or word number,

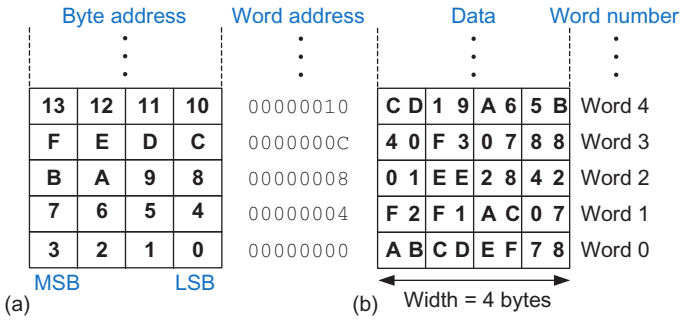


Figure 6.1 **ARM byte-addressable memory showing: (a) byte address and (b) data**

Code Example 6.8 READING MEMORY

High-Level Code	ARM Assembly Code

`a = mem[2];`	`; R7 = a`
	`MOV R5, #0 ; base address = 0`
	`LDR R7, [R5, #8] ; R7 <= data at memory address (R5+8)`

A read from the base address (i.e., index 0) is a special case that requires no offset in the assembly code. For example, a memory read from the base address held in R5 is written as `LDR R3, [R5]`.

which we discuss further in Section 6.3.6. The `LDR` instruction specifies the memory address using a *base register* (R5) and an *offset* (8). Recall that each data word is 4 bytes, so word number 1 is at address 4, word number 2 is at address 8, and so on. The word address is four times the word number. The memory address is formed by adding the contents of the base register (R5) and the offset. ARM offers several modes for accessing memory, as will be discussed in Section 6.3.6.

After the load register instruction (`LDR`) is executed in Code Example 6.8, R7 holds the value 0x01EE2842, which is the data value stored at memory address 8 in Figure 6.1.

ARM uses the *store register* instruction, `STR`, to write a data word from a register into memory. Code Example 6.9 writes the value 42 from register R9 into memory word 5.

Byte-addressable memories are organized in a big-endian or little-endian fashion, as shown in Figure 6.2. In both formats, a 32-bit word's most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. Word addresses are the same in both formats and refer to the same four bytes. Only the addresses of bytes within a word

ARMv4 requires *word-aligned addresses* for `LDR` and `STR`, that is, a word address that is divisible by four. Since ARMv6, this alignment restriction can be removed by setting a bit in the ARM system control register, but performance of *unaligned* loads is usually worse. Some architectures, such as x86, allow non-word-aligned data reads and writes, but others, such as MIPS, require strict alignment for simplicity. Of course, byte addresses for load byte and store byte, `LDRB` and `STRB` (discussed in Section 6.3.6), need not be word aligned.

Code Example 6.9 WRITING MEMORY

High-Level Code	ARM Assembly Code

`mem[5] = 42;`	`MOV R1, #0 ; base address = 0`
	`MOV R9, #42`
	`STR R9, [R1, #0x14] ; value stored at memory address (R1+20) = 42`

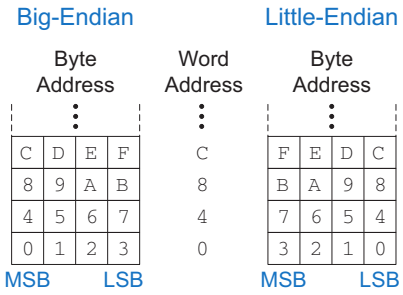
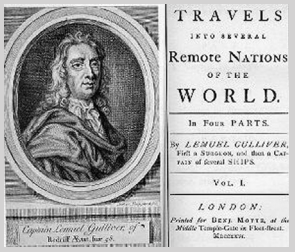


Figure 6.2 Big-endian and little-endian memory addressing

differ. In *big-endian* machines, bytes are numbered starting with 0 at the big (most significant) end. In *little-endian* machines, bytes are numbered starting with 0 at the little (least significant) end.

IBM's PowerPC (formerly found in Macintosh computers) uses big-endian addressing. Intel's x86 architecture (found in PCs) uses little-endian addressing. ARM prefers little-endian but provides support in some versions for *bi-endian* data addressing, which allows data loads and stores in either format. The choice of endianness is completely arbitrary but leads to hassles when sharing data between big-endian and little-endian computers. In examples in this text, we use little-endian format whenever byte ordering matters.



The terms big-endian and little-endian come from Jonathan Swift's *Gulliver's Travels*, first published in 1726 under the pseudonym of Isaac Bickerstaff. In his stories, the Lilliputian king required his citizens (the Little-Endians) to break their eggs on the little end. The Big-Endians were rebels who broke their eggs on the big end.

These terms were first applied to computer architectures by Danny Cohen in his paper "On Holy Wars and a Plea for Peace" published on April Fools Day, 1980 (*USC/ISI IEN 137*). (Photo courtesy of The Brotherton Collection, Leeds University Library.)

6.3 PROGRAMMING

Software languages such as C or Java are called high-level programming languages because they are written at a more abstract level than assembly language. Many high-level languages use common software constructs such as arithmetic and logical operations, conditional execution, if/else statements, for and while loops, array indexing, and function calls. See Appendix C for more examples of these constructs in C. In this section, we explore how to translate these high-level constructs into ARM assembly code.

6.3.1 Data-processing Instructions

The ARM architecture defines a variety of *data-processing* instruction (often called logical and arithmetic instructions in other architectures). We introduce these instructions briefly here because they are necessary to implement higher-level constructs. Appendix B provides a summary of ARM instructions.

Logical Instructions

ARM *logical operations* include `AND`, `ORR` (OR), `EOR` (XOR), and `BIC` (bit clear). These each operate bitwise on two sources and write the result

Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
`AND R3, R1, R2`	R3	0100 0110	1010 0001	0000 0000	0000 0000
`ORR R4, R1, R2`	R4	1111 1111	1111 1111	1111 0001	1011 0111
`EOR R5, R1, R2`	R5	1011 1001	0101 1110	1111 0001	1011 0111
`BIC R6, R1, R2`	R6	0000 0000	0000 0000	1111 0001	1011 0111
`MVN R7, R2`	R7	0000 0000	0000 0000	1111 1111	1111 1111

Figure 6.3 Logical operations

to a destination register. The first source is always a register and the second source is either an immediate or another register. Another logical operation, `MVN` (MoVe and Not), performs a bitwise NOT on the second source (an immediate or register) and writes the result to the destination register. Figure 6.3 shows examples of these operations on the two source values 0x46A1F1B7 and 0xFFFF0000. The figure shows the values stored in the destination register after the instruction executes.

The bit clear (`BIC`) instruction is useful for masking bits (i.e., forcing unwanted bits to 0). `BIC R6, R1, R2` computes R1 AND NOT R2. In other words, `BIC` clears the bits that are asserted in R2. In this case, the top two bytes of R1 are cleared or *masked*, and the unmasked bottom two bytes of R1, 0xF1B7, are placed in R6. Any subset of register bits can be masked.

The `ORR` instruction is useful for combining bitfields from two registers. For example, 0x347A0000 `ORR` 0x000072FC = 0x347A72FC.

Shift Instructions

Shift instructions shift the value in a register left or right, dropping bits off the end. The rotate instruction rotates the value in a register right by up to 31 bits. We refer to both shift and rotate generically as shift operations. ARM shift operations are `LSL` (logical shift left), `LSR` (logical shift right), `ASR` (arithmetic shift right), and `ROR` (rotate right). There is no `ROL` instruction because left rotation can be performed with a right rotation by a complementary amount.

As discussed in Section 5.2.5, left shifts always fill the least significant bits with 0's. However, right shifts can be either logical (0's shift into the most significant bits) or arithmetic (the sign bit shifts into the most significant bits). The amount by which to shift can be an immediate or a register.

Figure 6.4 shows the assembly code and resulting register values for `LSL`, `LSR`, `ASR`, and `ROR` when shifting by an immediate value. R5 is shifted by the immediate amount, and the result is placed in the destination register.

	Source register			
R5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
`LSL R0, R5, #7`	R0	1000 1110	0000 1000	0111 0011	1000 0000
`LSR R1, R5, #17`	R1	0000 0000	0000 0000	0111 1111	1000 1110
`ASR R2, R5, #3`	R2	1111 1111	1110 0011	1000 0010	0001 1100
`ROR R3, R5, #21`	R3	1110 0000	1000 0111	0011 1111	1111 1000

Figure 6.4 Shift instructions with immediate shift amounts

	Source registers			
R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code		Result			
`LSL R4, R8, R6`	R4	0110 1110	0111 0000	0000 0000	0000 0000
`ROR R5, R8, R6`	R5	1100 0001	0110 1110	0111 0000	1000 0001

Figure 6.5 Shift instructions with register shift amounts

Shifting a value left by N is equivalent to multiplying it by 2^N. Likewise, arithmetically shifting a value right by N is equivalent to dividing it by 2^N, as discussed in Section 5.2.5. Logical shifts are also used to extract or assemble bitfields.

Figure 6.5 shows the assembly code and resulting register values for shift operations where the shift amount is held in a register, R6. This instruction uses the *register-shifted register* addressing mode, where one register (R8) is shifted by the amount (20) held in a second register (R6).

Multiply Instructions*

Multiplication is somewhat different from other arithmetic operations. Multiplying two 32-bit numbers produces a 64-bit product. The ARM architecture provides *multiply instructions* that result in a 32-bit or 64-bit product. Multiply (`MUL`) multiplies two 32-bit numbers and produces a 32-bit result. `MUL R1, R2, R3` multiplies the values in R2 and R3 and places the least significant bits of the product in R1; the most significant 32 bits of the product are discarded. This instruction is useful for multiplying small numbers whose result fits in 32 bits. `UMULL` (unsigned multiply long) and `SMULL` (signed multiply long) multiply two 32-bit numbers and produce a 64-bit product. For example, `UMULL R1, R2, R3, R4` performs an unsigned multiply of R3 and R4. The least significant 32 bits of the product is placed in R1 and the most significant 32 bits are placed in R2.

Each of these instructions also has a multiply-accumulate variant, `MLA`, `SMLAL`, and `UMLAL`, that adds the product to a running 32- or 64-bit sum. These instructions can boost the math performance in applications such as matrix multiplication and signal processing consisting of repeated multiplies and adds.

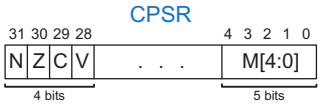


Figure 6.6 Current Program Status Register (CPSR)

The least significant five bits of the CPSR are *mode* bits and will be described in Section 6.6.3.

Other useful instructions for comparing two values are `CMN`, `TST`, and `TEQ`. Each instruction performs an operation, updates the condition flags, and discards the result. `CMN` (compare negative) compares the first source to the negative of the second source by adding the two sources. As will be shown in Section 6.4, ARM instructions only encode positive immediates. So, `CMN R2, #20` is used instead of `CMP R2, #-20`. `TST` (test) ANDs the source operands. It is useful for checking if some portion of the register is zero or nonzero. For example, `TST R2, #0xFF` would set the *Z* flag if the low byte of R2 is 0. `TEQ` (test if equal) checks for equivalence by XOR-ing the sources. Thus, the *Z* flag is set when they are equal and the *N* flag is set when the signs are different.

6.3.2 Condition Flags

Programs would be boring if they could only run in the same order every time. ARM instructions optionally set *condition flags* based on whether the result is negative, zero, etc. Subsequent instructions then execute *conditionally*, depending on the state of those condition flags. The ARM condition flags, also called *status flags*, are negative (*N*), zero (*Z*), carry (*C*), and overflow (*V*), as listed in Table 6.2. These flags are set by the ALU (see Section 5.2.4) and are held in the top 4 bits of the 32-bit *Current Program Status Register* (*CPSR*), as shown in Figure 6.6.

The most common way to set the status bits is with the compare (`CMP`) instruction, which subtracts the second source operand from the first and sets the condition flags based on the result. For example, if the numbers are equal, the result will be zero and the *Z* flag is set. If the first number is an unsigned value that is higher than or the same as the second, the subtraction will produce a carry out and the *C* flag is set.

Subsequent instructions can conditionally execute depending on the state of the flags. The instruction mnemonic is followed by a *condition mnemonic* that indicates when to execute. Table 6.3 lists the 4-bit condition field (*cond*), the condition mnemonic, name, and the state of the condition flags that result in instruction execution (CondEx). For example, suppose a program performs `CMP R4, R5`, and then `ADDEQ R1, R2, R3`. The compare sets the *Z* flag if R4 and R5 are equal, and the `ADDEQ` executes only if the *Z* flag is set. The *cond* field will be used in machine language encodings in Section 6.4.

Table 6.2 Condition flags

Flag	Name	Description
N	Negative	Instruction result is negative, i.e., bit 31 of the result is 1
Z	Zero	Instruction result is zero
C	Carry	Instruction causes a carry out
V	oVerflow	Instruction causes an overflow

Table 6.3 Condition mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

Condition mnemonics differ for signed and unsigned comparison. For example, ARM provides two forms of greater than or equal comparison: HS (CS) is used for unsigned numbers and GE for signed. For unsigned numbers, $A - B$ will produce a carry out (C) when $A \geq B$. For signed numbers, $A - B$ will make N and V either both 0 or both 1 when $A \geq B$. Figure 6.7 highlights the difference between HS and GE comparisons with two examples using 4-bit numbers for ease of interpretation.

	Unsigned	Signed
A = 1001_2	A = 9	A = –7
B = 0010_2	B = 2	B = 2

A – B: 1001 + 1110 = 10111 (a)

$NZCV = 0011_2$

HS: TRUE

GE: FALSE

	Unsigned	Signed
A = 0101_2	A = 5	A = 5
B = 1101_2	B = 13	B = –3

A – B: 0101 + 0011 = 1000 (b)

$NZCV = 1001_2$

HS: FALSE

GE: TRUE

Figure 6.7 Signed vs. unsigned comparison: HS vs. GE

Other data-processing instructions will set the condition flags when the instruction mnemonic is followed by "S." For example, `SUBS R2, R3, R7` will subtract R7 from R3, put the result in R2, and set the condition flags. Table B.5 in Appendix B summarizes which condition flags are influenced by each instruction. All data-processing instructions will affect the N and Z flags based on whether the result is zero or has the most significant bit set. `ADDS` and `SUBS` also influence V and C, and shifts influence C.

Code Example 6.10 shows instructions that execute conditionally. The first instruction, `CMP R2, R3`, executes unconditionally and sets the condition flags. The remaining instructions execute conditionally, depending on the values of the condition flags. Suppose R2 and R3 contain the values 0x80000000 and 0x00000001. The compare computes R2 – R3 = 0x80000000 – 0x00000001 = 0x80000000 + 0xFFFFFFFF = 0x7FFFFFFF with a carry out ($C = 1$). The sources had opposite signs and the sign of the result differs from the sign of the first source, so the result overflows ($V = 1$). The remaining flags (N and Z) are 0. `ANDHS` executes

Code Example 6.10 CONDITIONAL EXECUTION

ARM Assembly Code

```
CMP   R2, R3
ADDEQ R4, R5, #78
ANDHS R7, R8, R9
ORRMI R10, R11, R12
EORLT R12, R7, R10
```

because $C = 1$. `EORLT` executes because N is 0 and V is 1 (see Table 6.3). Intuitively, `ANDHS` and `EORLT` execute because R2 $\geq$ R3 (unsigned) and R2 $<$ R3 (signed), respectively. `ADDEQ` and `ORRMI` do not execute because the result of R2 – R3 is not zero (i.e., R2 $\neq$ R3) or negative.

6.3.3 Branching

An advantage of a computer over a calculator is its ability to make decisions. A computer performs different tasks depending on the input. For example, if/else statements, switch/case statements, while loops, and for loops all conditionally execute code depending on some test.

One way to make decisions is to use conditional execution to ignore certain instructions. This works well for simple if statements where a small number of instructions are ignored, but it is wasteful for if statements with many instructions in the body, and it is insufficient to handle loops. Thus, ARM and most other architectures use *branch instructions* to skip over sections of code or repeat code.

A program usually executes in sequence, with the program counter (PC) incrementing by 4 after each instruction to point to the next instruction. (Recall that instructions are 4 bytes long and ARM is a byte-addressed architecture.) Branch instructions change the program counter. ARM includes two types of branches: a simple *branch* (`B`) and *branch and link* (`BL`). `BL` is used for function calls and is discussed in Section 6.3.7. Like other ARM instructions, branches can be unconditional or conditional. Branches are also called *jumps* in some architectures.

Code Example 6.11 shows unconditional branching using the branch instruction `B`. When the code reaches the `B TARGET` instruction, the branch is *taken*. That is, the next instruction executed is the `SUB` instruction just after the *label* called `TARGET`.

Assembly code uses labels to indicate instruction locations in the program. When the assembly code is translated into machine code, these labels are translated into instruction addresses (see Section 6.4.3). ARM assembly labels cannot be reserved words, such as instruction mnemonics. Most programmers indent their instructions but not the labels, to help

Code Example 6.11 UNCONDITIONAL BRANCHING

ARM Assembly Code

```
  ADD R1, R2, #17    ; R1 = R2 + 17
  B   TARGET         ; branch to TARGET
  ORR R1, R1, R3     ; not executed
  AND R3, R1, #0xFF  ; not executed

TARGET
  SUB R1, R1, #78    ; R1 = R1 – 78
```

Code Example 6.12 CONDITIONAL BRANCHING

ARM Assembly Code

```
  MOV R0, #4         ; R0 = 4
  ADD R1, R0, R0     ; R1 = R0 + R0 = 8
  CMP R0, R1         ; set flags based on R0–R1 = –4. NZCV = 1000
  BEQ THERE          ; branch not taken (Z != 1)
  ORR R1, R1, #1     ; R1 = R1 OR 1 = 9

THERE
  ADD R1, R1, #78    ; R1 = R1 + 78 = 87
```

make labels stand out. The ARM compiler makes this a requirement: labels must not be indented, and instructions must be preceded by white space. Some compilers, including GCC, require a colon after the label.

Branch instructions can execute conditionally based on the condition mnemonics listed in Table 6.3. Code Example 6.12 illustrates the use of `BEQ`, branching dependent on equality ($Z = 1$). When the code reaches the `BEQ` instruction, the Z condition flag is 0 (i.e., `R0` $\neq$ `R1`), so the branch is *not taken*. That is, the next instruction executed is the `ORR` instruction.

6.3.4 Conditional Statements

if, if/else, and switch/case statements are conditional statements commonly used in high-level languages. They each conditionally execute a *block* of code consisting of one or more statements. This section shows how to translate these high-level constructs into ARM assembly language.

if Statements

An if statement executes a block of code, the *if block*, only when a condition is met. Code Example 6.13 shows how to translate an if statement into ARM assembly code.

Code Example 6.13 IF STATEMENT

High-Level Code

```
if (apples == oranges)
  f = i + 1;

f = f - i;
```

ARM Assembly Code

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
  CMP R0, R1       ; apples == oranges ?
  BNE L1           ; if not equal, skip if block
  ADD R2, R3, #1   ; if block: f = i + 1
L1
  SUB R2, R2, R3   ; f = f - i
```

Recall that != is an inequality comparison and == is an equality comparison in the high-level code.

The assembly code for the if statement tests the opposite condition of the one in the high-level code. In Code Example 6.13, the high-level code tests for `apples == oranges`. The assembly code tests for `apples != oranges` using `BNE` to skip the if block if the condition is **not** satisfied. Otherwise, `apples == oranges`, the branch is not taken, and the if block is executed.

Because any instruction can be conditionally executed, the ARM assembly code for Code Example 6.13 could also be written more compactly as shown below.

```
CMP   R0, R1          ; apples == oranges ?
ADDEQ R2, R3, #1      ; f = i + 1 on equality (i.e., Z = 1)
SUB   R2, R2, R3      ; f = f - i
```

This solution with conditional execution is shorter and also faster because it involves one fewer instruction. Moreover, we will see in Section 7.5.3 that branches sometimes introduce extra delay, whereas conditional execution is always fast. This example shows the power of conditional execution in the ARM architecture.

In general, when a block of code has a single instruction, it is better to use conditional execution rather than branch around it. As the block becomes longer, the branch becomes valuable because it avoids wasting time fetching instructions that will not be executed.

if/else Statements

if/else statements execute one of two blocks of code depending on a condition. When the condition in the if statement is met, the *if block* is executed. Otherwise, the *else block* is executed. Code Example 6.14 shows an example if/else statement.

Like if statements, if/else assembly code tests the opposite condition of the one in the high-level code. In Code Example 6.14, the high-level code tests for `apples == oranges`, and the assembly code tests for `apples != oranges`. If that opposite condition is TRUE, `BNE` skips the if block and executes the else block. Otherwise, the if block executes and finishes with an unconditional branch (`B`) past the else block.

Code Example 6.14 IF/ELSE STATEMENT

High-Level Code

```
if (apples == oranges)
  f = i + 1;


else
  f = f - i;
```

ARM Assembly Code

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
  CMP R0, R1        ; apples == oranges?
  BNE L1            ; if not equal, skip if block
  ADD R2, R3, #1    ; if block: f = i + 1
  B   L2            ; skip else block
L1
  SUB R2, R2, R3    ; else block: f = f - i
L2
```

Again, because any instruction can conditionally execute and because the instructions within the if block do not change the condition flags, the ARM assembly code for Code Example 6.14 could also be written much more succinctly as:

```
CMP   R0, R1        ; apples == oranges?
ADDEQ R2, R3, #1    ; f = i + 1 on equality (i.e., Z = 1)
SUBNE R2, R2, R3    ; f = f - i on not equal (i.e., Z = 0)
```

switch/case Statements*

switch/case statements execute one of several blocks of code depending on the conditions. If no conditions are met, the *default block* is executed. A case statement is equivalent to a series of *nested* if/else statements. Code Example 6.15 shows two high-level code snippets with the same

Code Example 6.15 SWITCH/CASE STATEMENT

High-Level Code

```
switch (button) {
  case 1:   amt = 20;  break;


  case 2:   amt = 50;  break;


  case 3:   amt = 100; break;


  default: amt = 0;
}
// equivalent function using
// if/else statements
  if      (button == 1) amt = 20;
  else if (button == 2) amt = 50;
  else if (button == 3) amt = 100;
  else                  amt = 0;
```

ARM Assembly Code

```
; R0 = button, R1 = amt
  CMP   R0, #1          ; is button 1 ?
  MOVEQ R1, #20         ; amt = 20 if button is 1
  BEQ   DONE            ; break

  CMP   R0, #2          ; is button 2 ?
  MOVEQ R1, #50         ; amt = 50 if button is 2
  BEQ   DONE            ; break

  CMP   R0, #3          ; is button 3?
  MOVEQ R1, #100        ; amt = 100 if button is 3
  BEQ   DONE            ; break

  MOV   R1, #0          ; default amt = 0
DONE
```

functionality: they calculate whether to dispense $20, $50, or $100 from an ATM (automatic teller machine) depending on the button pressed. The ARM assembly implementation is the same for both high-level code snippets.

6.3.5 Getting Loopy

Loops repeatedly execute a block of code depending on a condition. while loops and for loops are common loop constructs used by high-level languages. This section shows how to translate them into ARM assembly language, taking advantage of conditional branching.

while Loops

while loops repeatedly execute a block of code until a condition is *not* met. The while loop in Code Example 6.16 determines the value of `x` such that $2^x = 128$. It executes seven times, until `pow` = 128.

The `int` data type in C refers to a word of data representing a two's complement integer. ARM uses 32-bit words, so an `int` represents a number in the range $[-2^{31}, 2^{31} - 1]$.

Like if/else statements, the assembly code for while loops tests the opposite condition of the one in the high-level code. If that opposite condition is TRUE (in this case, R0 == 128), the while loop is finished. If not (R0 ≠128), the branch isn't taken and the loop body executes.

Code Example 6.16 WHILE LOOP

High-Level Code

```
int pow = 1;
int x = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

```
; R0 = pow, R1 = x
  MOV R0, #1      ; pow = 1
  MOV R1, #0      ; x = 0

WHILE
  CMP R0, #128    ; pow != 128 ?
  BEQ DONE        ; if pow == 128, exit loop
  LSL R0, R0, #1  ; pow = pow * 2
  ADD R1, R1, #1  ; x = x + 1
  B   WHILE       ; repeat loop
DONE
```

In Code Example 6.16, the while loop compares `pow` to 128 and exits the loop if it is equal. Otherwise it doubles `pow` (using a left shift), increments `x`, and branches back to the start of the while loop.

for Loops

It is very common to initialize a variable before a while loop, check that variable in the loop condition, and change that variable each time through the while loop. for loops are a convenient shorthand that combines the initialization, condition check, and variable change in one place. The format of the for loop is:

```
for (initialization; condition; loop operation)
  statement
```

Code Example 6.17 FOR LOOP

High-Level Code

```
int i;
int sum = 0;


for (i = 0; i < 10; i = i + 1) {
  sum = sum + i;
}
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV R1, #0        ; sum = 0
  MOV R0, #0        ; i = 0          loop initialization

FOR
  CMP R0, #10       ; i < 10 ?       check condition
  BGE DONE          ; if (i >= 10) exit loop
  ADD R1, R1, R0    ; sum = sum + i  loop body
  ADD R0, R0, #1    ; i = i + 1      loop operation
  B   FOR           ; repeat loop
DONE
```

The initialization code executes before the for loop begins. The condition is tested at the beginning of each loop. If the condition is not met, the loop exits. The loop operation executes at the end of each loop.

Code Example 6.17 adds the numbers from 0 to 9. The loop variable, in this case `i`, is initialized to 0 and is incremented at the end of each loop iteration. The for loop executes as long as `i` is less than 10. Note that this example also illustrates relative comparisons. The loop checks the < condition to continue, so the assembly code checks the opposite condition, >=, to exit the loop.

Loops are especially useful for accessing large amounts of similar data stored in memory, which is discussed next.

6.3.6 Memory

For ease of storage and access, similar data can be grouped together into an *array*. An array stores its contents at sequential data addresses in memory. Each array element is identified by a number called its *index*. The number of elements in the array is called the *length* of the array.

Figure 6.8 shows a 200-element array of scores stored in memory. Code Example 6.18 is a grade inflation algorithm that adds 10 points to each of the scores. Note that the code for initializing the scores array is not shown. The index into the array is a variable (`i`) rather than a constant, so we must multiply it by 4 before adding it to the base address.

ARM can *scale* (multiply) the index, add it to the base address, and load from memory in a single instruction. Instead of the `LSL` and `LDR` instruction sequence in Code Example 6.18, we can use a single instruction:

```
LDR R3, [R0, R1, LSL #2]
```

R1 is scaled (shifted left by two) then added to the base address (R0). Thus, the memory address is R0 + (R1 × 4).

Figure 6.8 Memory holding `scores[200]` starting at base address 0x14000000

Code Example 6.18 ACCESSING ARRAYS USING A FOR LOOP

High-Level Code

```
int i;
int scores[200];
...

for (i = 0; i < 200; i = i + 1)

  scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
; initialization code      ...
  MOV R0, #0x14000000      ; R0 = base address
  MOV R1, #0               ; i = 0

LOOP
  CMP R1, #200             ; i < 200?
  BGE L3                   ; if i ≥ 200, exit loop
  LSL R2, R1, #2           ; R2 = i * 4
  LDR R3, [R0, R2]         ; R3 = scores[i]
  ADD R3, R3, #10          ; R3 = scores[i] + 10
  STR R3, [R0, R2]         ; scores[i] = scores[i] + 10
  ADD R1, R1, #1           ; i = i + 1
  B   LOOP                 ; repeat loop
L3
```

In addition to scaling the index register, ARM provides offset, pre-indexed, and post-indexed addressing to enable dense and efficient code for array accesses and function calls. Table 6.4 gives examples of each indexing mode. In each case, the base register is R1 and the offset is R2. The offset can be subtracted by writing –R2. The offset may also be an immediate in the range of 0–4095 that can be added (e.g., #20) or subtracted (e.g., #–20).

Offset addressing calculates the address as the base register ± the offset; the base register is unchanged. *Pre-indexed addressing* calculates the address as the base register ± the offset and updates the base register to this new address. *Post-indexed addressing* calculates the address as the base register only and then, after accessing memory, the base register is updated to the base register ± the offset. We have seen many examples of offset indexing mode. Code Example 6.19 shows the for loop from Code Example 6.18 rewritten to use post-indexing, eliminating the `ADD` to increment `i`.

Table 6.4 ARM indexing modes

Mode	ARM Assembly	Address	Base Register
Offset	`LDR R0, [R1, R2]`	R1 + R2	Unchanged
Pre-index	`LDR R0, [R1, R2]!`	R1 + R2	R1 = R1 + R2
Post-index	`LDR R0, [R1], R2`	R1	R1 = R1 + R2

Code Example 6.19 FOR LOOP USING POST-INDEXING

High-Level Code

```
int i;
int scores[200];
...

for (i = 0; i < 200; i = i + 1)
        scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address
; initialization code    ...
  MOV R0, #0x14000000    ; R0 = base address
  ADD R1, R0, #800       ; R1 = base address + (200*4)

LOOP
  CMP R0, R1             ; reached end of array?
  BGE L3                 ; if yes, exit loop
  LDR R2, [R0]           ; R2 = scores[i]
  ADD R2, R2, #10        ; R2 = scores[i] + 10
  STR R2, [R0], #4       ; scores[i] = scores[i] + 10
                         ; then R0 = R0 + 4
  B   LOOP               ; repeat loop
  L3
```

Bytes and Characters

Numbers in the range [–128, 127] can be stored in a single byte rather than an entire word. Because there are much fewer than 256 characters on an English language keyboard, English characters are often represented by bytes. The C language uses the type `char` to represent a byte or character.

Other programming languages, such as Java, use different character encodings, most notably Unicode. Unicode uses 16 bits to represent each character, so it supports accents, umlauts, and Asian languages. For more information, see www.unicode.org.

Early computers lacked a standard mapping between bytes and English characters, so exchanging text between computers was difficult. In 1963, the American Standards Association published the *American Standard Code for Information Interchange* (*ASCII*), which assigns each text character a unique byte value. Table 6.5 shows these character encodings for printable characters. The ASCII values are given in hexadecimal. Lowercase and uppercase letters differ by 0x20 (32).

ARM provides load byte (`LDRB`), load signed byte (`LDRSB`), and store byte (`STRB`) to access individual bytes in memory. `LDRB` zero-extends the byte, whereas `LDRSB` sign-extends the byte to fill the entire 32-bit register. `STRB` stores the least significant byte of the 32-bit register into the specified byte address in memory. All three are illustrated in Figure 6.9, with

`LDRH`, `LDRSH`, and `STRH` are similar, but access 16-bit *halfwords*.

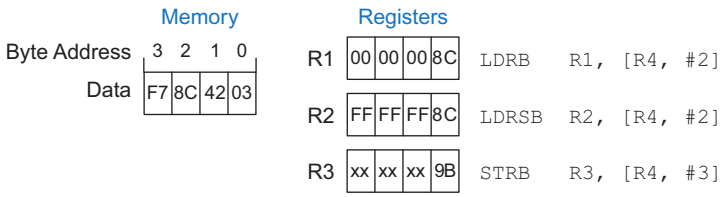


Figure 6.9 Instructions for loading and storing bytes

ASCII codes developed from earlier forms of character encoding. Beginning in 1838, telegraph machines used Morse code, a series of dots (.) and dashes (–), to represent characters. For example, the letters A, B, C, and D were represented as – , – ... , – . – . , and – .. , respectively. The number of dots and dashes varied with each letter. For efficiency, common letters used shorter codes.

In 1874, Jean-Maurice-Emile Baudot invented a 5-bit code called the Baudot code. For example, A, B, C, and D were represented as 00011, 11001, 01110, and 01001.

However, the 32 possible encodings of this 5-bit code were not sufficient for all the English characters, but 8-bit encoding was. Thus, as electronic communication became prevalent, 8-bit ASCII encoding emerged as the standard.

Table 6.5 ASCII encodings

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	‘	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

the base address R4 being 0. `LDRB` loads the byte at memory address 2 into the least significant byte of R1 and fills the remaining register bits with 0. `LDRSB` loads this byte into R2 and sign-extends the byte into the upper 24 bits of the register. `STRB` stores the least significant byte of R3 (0x9B) into memory byte 3; it replaces 0xF7 with 0x9B. The more significant bytes of R3 are ignored.

A series of characters is called a *string*. Strings have a variable length, so programming languages must provide a way to determine the length or end of the string. In C, the null character (0x00) signifies the end of a string. For example, Figure 6.10 shows the string “Hello!” (0x48 65 6C 6C 6F 21 00) stored in memory. The string is seven bytes long

Example 6.2 USING LDRB AND STRB TO ACCESS A CHARACTER ARRAY

The following high-level code converts a 10-entry array of characters from lowercase to uppercase by subtracting 32 from each array entry. Translate it into ARM assembly language. Remember that the address difference between array elements is now 1 byte, not 4 bytes. Assume that R0 already holds the base address of `chararray`.

```
// high-level code
// chararray[10] declared and initialized earlier
int i;

for (i = 0; i < 10; i = i + 1)
  chararray[i] = chararray[i] - 32;
```

Solution:

```
; ARM assembly code
; R0 = base address of chararray (initialized earlier), R1 = i
            MOV   R1, #0            ; i = 0
LOOP        CMP   R1, #10           ; i < 10 ?
            BGE   DONE              ; if (i >=10), exit loop
            LDRB  R2, [R0, R1]      ; R2 = mem[R0+R1] = chararray[i]
            SUB   R2, R2, #32       ; R2 = chararray[i] - 32
            STRB  R2, [R0, R1]      ; chararray[i] = R2
            ADD   R1, R1, #1        ; i = i + 1
            B     LOOP              ; repeat loop
DONE
```

and extends from address 0x1522FFF0 to 0x1522FFF6. The first character of the string (H = 0x48) is stored at the lowest byte address (0x1522FFF0).

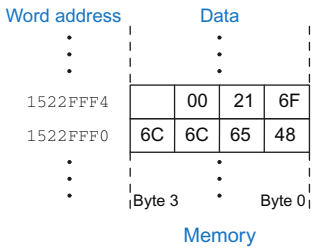


Figure 6.10 **The string "Hello!" stored in memory**

6.3.7 Function Calls

High-level languages support *functions* (also called *procedures* or *subroutines*) to reuse common code and to make a program more modular and readable. Functions have inputs, called *arguments*, and an output, called the *return value*. Functions should calculate the return value and cause no other unintended side effects.

When one function calls another, the calling function, the *caller*, and the called function, the *callee*, must agree on where to put the arguments and the return value. In ARM, the caller conventionally places up to four arguments in registers R0–R3 before making the function call,

and the callee places the return value in register R0 before finishing. By following this convention, both functions know where to find the arguments and return value, even if the caller and callee were written by different people.

The callee must not interfere with the behavior of the caller. This means that the callee must know where to return to after it completes and it must not trample on any registers or memory needed by the caller. The caller stores the return address in the link register LR at the same time it jumps to the callee using the branch and link instruction (`BL`). The callee must not overwrite any architectural state or memory that the caller is depending on. Specifically, the callee must leave the *saved registers* (R4–R11, and LR) and the *stack*, a portion of memory used for temporary variables, unmodified.

This section shows how to call and return from a function. It shows how functions access input arguments and the return value and how they use the stack to store temporary variables.

Function Calls and Returns

ARM uses the branch and link instruction (`BL`) to call a function and moves the link register to the PC (`MOV PC, LR`) to return from a function. Code Example 6.20 shows the `main` function calling the `simple` function. `main` is the caller, and `simple` is the callee. The `simple` function is called with no input arguments and generates no return value; it just returns to the caller. In Code Example 6.20, instruction addresses are given to the left of each ARM instruction in hexadecimal.

`BL` (branch and link) and `MOV PC, LR` are the two essential instructions needed for a function call and return. `BL` performs two tasks: it stores the *return address* of the next instruction (the instruction

Code Example 6.20 simple FUNCTION CALL

High-Level Code

```
int main() {

  simple();
  ...
}

// void means the function returns no value
void simple() {
  return;
}
```

ARM Assembly Code

```
0x00008000 MAIN     ...
...                 ...
0x00008020          BL  SIMPLE    ; call the simple function
...


0x0000902C  SIMPLE  MOV PC, LR    ; return
```

after BL) in the link register (LR), and it branches to the target instruction.

In Code Example 6.20, the main function calls the simple function by executing the branch and link instruction (BL). BL branches to the SIMPLE label and stores 0x00008024 in LR. The simple function returns immediately by executing the instruction MOV PC, LR, copying the return address from the LR back to the PC. The main function then continues executing at this address (0x00008024).

Remember that PC and LR are alternative names for R15 and R14, respectively. ARM is unusual in that PC is part of the register set, so a function return can be done with a MOV instruction. Many other instruction sets keep the PC in a special register and use a special return or jump instruction to return from functions.

These days, ARM compilers do a function return using BX LR. The BX branch and exchange instruction is like a branch, but it also can transition between the standard ARM instruction set and the Thumb instruction set described in Section 6.7.1. This chapter doesn't use the Thumb or BX instructions and thus sticks with the ARMv4 MOV PC, LR method.

We will see in Chapter 7 that treating the PC as an ordinary register complicates the implementation of the processor.

Input Arguments and Return Values

The simple function in Code Example 6.20 receives no input from the calling function (main) and returns no output. By ARM convention, functions use R0–R3 for input arguments and R0 for the return value. In Code Example 6.21, the function diffofsums is called with four arguments and returns one result. result is a local variable, which we choose to keep in R4.

According to ARM convention, the calling function, main, places the function arguments from left to right into the input registers, R0–R3. The called function, diffofsums, stores the return value in the return register, R0. When a function with more than four arguments is called, the additional input arguments are placed on the stack, which we discuss next.

Code Example 6.21 FUNCTION CALL WITH ARGUMENTS AND RETURN VALUES

High-Level Code

```
int main() {
  int y;
  . . .
  y = diffofsums(2, 3, 4, 5);
  . . .
}



int diffofsums(int f, int g, int h, int i) {
  int result;

  result = (f + g) - (h + i);
  return result;
}
```

ARM Assembly Code

```
; R4 = y
MAIN
  . . .
  MOV R0, #2       ; argument 0 = 2
  MOV R1, #3       ; argument 1 = 3
  MOV R2, #4       ; argument 2 = 4
  MOV R3, #5       ; argument 3 = 5
  BL  DIFFOFSUMS   ; call function
  MOV R4, R0       ; y = returned value
  . . .

; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1   ; R8 = f + g
  ADD R9, R2, R3   ; R9 = h + i
  SUB R4, R8, R9   ; result = (f + g) - (h + i)
  MOV R0, R4       ; put return value in R0
  MOV PC, LR       ; return to caller
```

Code Example 6.21 has some subtle errors. Code Examples 6.22–6.25 show improved versions of the program.

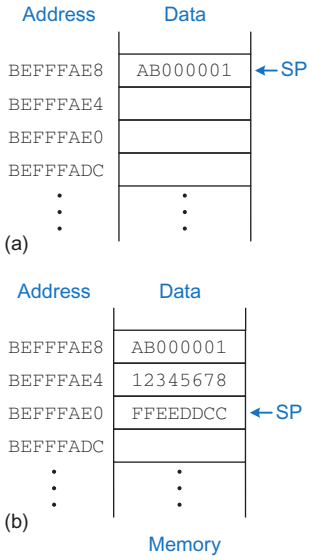


Figure 6.11 **The stack (a) before expansion and (b) after two-word expansion**

The stack is typically stored upside down in memory such that the top of the stack is actually the lowest address and the stack grows downward toward lower memory addresses. This is called a *descending stack*. ARM also allows for *ascending stacks* that grow up toward higher memory addresses. The stack pointer typically points to the topmost element on the stack; this is called a *full stack*. ARM also allows for *empty stacks* in which SP points one word beyond the top of the stack. The ARM *Application Binary Interface* (ABI) defines a standard way in which functions pass variables and use the stack so that libraries developed by different compilers can interoperate. It specifies a *full descending* stack, which we will use in this chapter.

The Stack

The stack is memory that is used to save information within a function. The stack expands (uses more memory) as the processor needs more scratch space and contracts (uses less memory) when the processor no longer needs the variables stored there. Before explaining how functions use the stack to store temporary values, we explain how the stack works.

The stack is a last-in-first-out (LIFO) queue. Like a stack of dishes, the last item *pushed* onto the stack (the top dish) is the first one that can be *popped* off. Each function may allocate stack space to store local variables but must deallocate it before returning. The *top of the stack* is the most recently allocated space. Whereas a stack of dishes grows up in space, the ARM stack grows down in memory. The stack expands to lower memory addresses when a program needs more scratch space.

Figure 6.11 shows a picture of the stack. The stack pointer, SP (R13), is an ordinary ARM register that, by convention, *points* to the *top of the stack*. A pointer is a fancy name for a memory address. SP points to (gives the address of) data. For example, in Figure 6.11(a), the stack pointer, SP, holds the address value 0XBEFFFAE8 and points to the data value 0xAB000001.

The stack pointer (SP) starts at a high memory address and decrements to expand as needed. Figure 6.11(b) shows the stack expanding to allow two more data words of temporary storage. To do so, SP decrements by eight to become 0xBEFFFAE0. Two additional data words, 0x12345678 and 0xFFEEDDCC, are temporarily stored on the stack.

One of the important uses of the stack is to save and restore registers that are used by a function. Recall that a function should calculate a return value but have no other unintended side effects. In particular, it should not modify any registers besides R0, the one containing the return value. The `diffofsums` function in Code Example 6.21 violates this rule because it modifies R4, R8, and R9. If `main` had been using these registers before the call to `diffofsums`, their contents would have been corrupted by the function call.

To solve this problem, a function saves registers on the stack before it modifies them, then restores them from the stack before it returns. Specifically, it performs the following steps:

1. Makes space on the stack to store the values of one or more registers
2. Stores the values of the registers on the stack
3. Executes the function using the registers
4. Restores the original values of the registers from the stack
5. Deallocates space on the stack

Code Example 6.22 shows an improved version of `diffofsums` that saves and restores R4, R8, and R9. Figure 6.12 shows the stack before, during, and after a call to the `diffofsums` function from Code Example 6.22. The stack starts at 0xBEF0F0FC. `diffofsums` makes room for three words on the stack by decrementing the stack pointer SP by 12. It then stores the current values held in R4, R8, and R9 in the newly allocated space. It executes the rest of the function, changing the values in these three registers. At the end of the function, `diffofsums` restores the values of these registers from the stack, deallocates its stack space, and returns. When the function returns, R0 holds the result, but there

Code Example 6.22 FUNCTION SAVING REGISTERS ON THE STACK

ARM Assembly Code

```
;R4 = result
DIFFOFSUMS
  SUB   SP, SP, #12      ; make space on stack for 3 registers
  STR   R9, [SP, #8]     ; save R9 on stack
  STR   R8, [SP, #4]     ; save R8 on stack
  STR   R4, [SP]         ; save R4 on stack

  ADD  R8, R0, R1        ; R8 = f + g
  ADD  R9, R2, R3        ; R9 = h + i
  SUB  R4, R8, R9        ; result = (f + g) - (h + i)
  MOV  R0, R4            ; put return value in R0

  LDR   R4, [SP]         ; restore R4 from stack
  LDR   R8, [SP, #4]     ; restore R8 from stack
  LDR   R9, [SP, #8]     ; restore R9 from stack
  ADD   SP, SP, #12      ; deallocate stack space

  MOV   PC, LR           ; return to caller
```

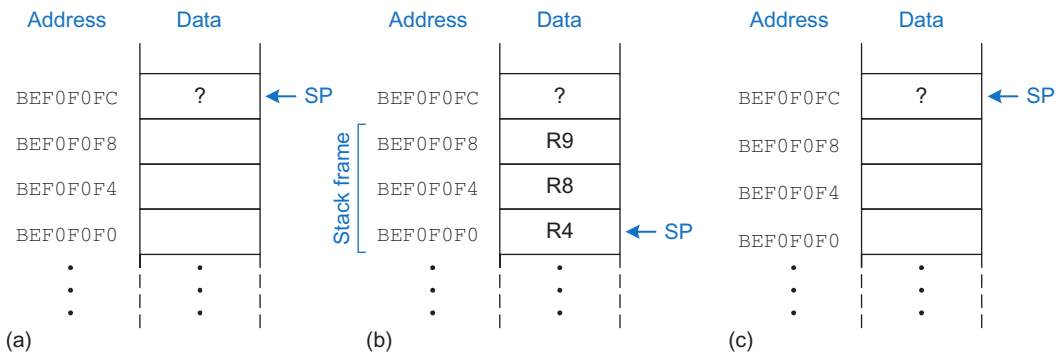


Figure 6.12 The stack: (a) before, (b) during, and (c) after the `diffofsums` function call

are no other side effects: R4, R8, R9, and SP have the same values as they did before the function call.

The stack space that a function allocates for itself is called its *stack frame*. `diffofsums`'s stack frame is three words deep. The principle of modularity tells us that each function should access only its own stack frame, not the frames belonging to other functions.

Loading and Storing Multiple Registers

Saving and restoring registers on the stack is such a common operation that ARM provides Load Multiple and Store Multiple instructions (`LDM` and `STM`) that are optimized to this purpose. Code Example 6.23 rewrites `diffofsums` using these instructions. The stack holds exactly the same information as in the previous example, but the code is much shorter.

Code Example 6.23 SAVING AND RESTORING MULTIPLE REGISTERS

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
   STMFD  SP!, {R4, R8, R9}        ; push R4/8/9 on full descending stack

   ADD    R8, R0, R1               ; R8 = f + g
   ADD    R9, R2, R3               ; R9 = h + i
   SUB    R4, R8, R9               ; result = (f + g) - (h + i)
   MOV    R0, R4                   ; put return value in R0

   LDMFD  SP!, {R4, R8, R9}        ; pop R4/8/9 off full descending stack

   MOV    PC, LR                   ; return to caller
```

`LDM` and `STM` come in four flavors for full and empty descending and ascending stacks (`FD`, `ED`, `FA`, `EA`). The `SP!` in the instructions indicates to store the data relative to the stack pointer and to update the stack pointer after the store or load. `PUSH` and `POP` are synonyms for `STMFD SP!, {regs}` and `LDMFD SP!, {regs}`, respectively, and are the preferred way to save registers on the conventional full descending stack.

Preserved Registers

Code Examples 6.22 and 6.23 assume that all of the used registers (R4, R8, and R9) must be saved and restored. If the calling function does not use those registers, the effort to save and restore them is wasted. To avoid this waste, ARM divides registers into *preserved* and *nonpreserved* categories. The preserved registers include R4–R11. The nonpreserved registers are R0–R3 and R12. SP and LR (R13 and R14)

must also be preserved. A function must save and restore any of the preserved registers that it wishes to use, but it can change the nonpreserved registers freely.

Code Example 6.24 shows a further improved version of `diffofsums` that saves only R4 on the stack. It also illustrates the preferred `PUSH` and `POP` synonyms. The code reuses the nonpreserved argument registers R1 and R3 to hold the intermediate sums when those arguments are no longer necessary.

Code Example 6.24 REDUCING THE NUMBER OF PRESERVED REGISTERS

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
   PUSH {R4}                    ; save R4 on stack
   ADD  R1, R0, R1              ; R1 = f + g
   ADD  R3, R2, R3              ; R3 = h + i
   SUB  R4, R1, R3              ; result = (f + g) - (h + i)
   MOV  R0, R4                  ; put return value in R0
   POP  {R4}                    ; pop R4 off stack
   MOV  PC, LR                  ; return to caller
```

`PUSH` (and `POP`) save (and restore) registers on the stack in order of register number from low to high, with the lowest numbered register placed at the lowest memory address, regardless of the order listed in the assembly instruction. For example, `PUSH {R8, R1, R3}` will store R1 at the lowest memory address, then R3 and finally R8 at the next higher memory addresses on the stack.

Remember that when one function calls another, the former is the caller and the latter is the callee. The callee must save and restore any preserved registers that it wishes to use. The callee may change any of the nonpreserved registers. Hence, if the caller is holding active data in a nonpreserved register, the caller needs to save that nonpreserved register before making the function call and then needs to restore it afterward. For these reasons, preserved registers are also called *callee-save*, and nonpreserved registers are called *caller-save*.

Table 6.6 summarizes which registers are preserved. R4–R11 are generally used to hold local variables within a function, so they must be saved. LR must also be saved, so that the function knows where to return.

Table 6.6 Preserved and nonpreserved registers

Preserved	Nonpreserved
Saved registers: R4–R11	Temporary register: R12
Stack pointer: SP (R13)	Argument registers: R0–R3
Return address: LR (R14)	Current Program Status Register
Stack above the stack pointer	Stack below the stack pointer

The convention of which registers are preserved or not preserved is part of the Procedure Call Standard for the ARM Architecture, rather than of the architecture itself. Alternate procedure call standards exist.

R0–R3 and R12 are used to hold temporary results. These calculations typically complete before a function call is made, so they are not preserved, and it is rare that the caller needs to save them.

R0–R3 are often overwritten in the process of calling a function. Hence, they must be saved by the caller if the caller depends on any of its own arguments after a called function returns. R0 certainly should not be preserved, because the callee returns its result in this register. Recall that the Current Program Status Register (CPSR) holds the condition flags. It is not preserved across function calls.

The stack above the stack pointer is automatically preserved as long as the callee does not write to memory addresses above SP. In this way, it does not modify the stack frame of any other functions. The stack pointer itself is preserved, because the callee deallocates its stack frame before returning by adding back the same amount that it subtracted from SP at the beginning of the function.

The astute reader or an optimizing compiler may notice that the local variable `result` is immediately returned without being used for anything else. Hence, we can eliminate the variable and simply store it in the return register R0, eliminating the need to push and pop R4 and to move `result` from R4 to R0. Code Example 6.25 shows this even further optimized `diffofsums`.

Nonleaf Function Calls

A function that does not call others is called a *leaf function*; `diffofsums` is an example. A function that does call others is called a *nonleaf function*. As mentioned, nonleaf functions are somewhat more complicated because they may need to save nonpreserved registers on the stack before they call another function and then restore those registers afterward. Specifically:

Caller save rule: Before a function call, the caller must save any nonpreserved registers (R0–R3 and R12) that it needs after the call. After the call, it must restore these registers before using them.

Callee save rule: Before a callee disturbs any of the preserved registers (R4–R11 and LR), it must save the registers. Before it returns, it must restore these registers.

Code Example 6.25 OPTIMIZED `diffofsums` FUNCTION CALL

ARM Assembly Code

```
DIFFOFSUMS
  ADD  R1, R0, R1    ; R1 = f + g
  ADD  R3, R2, R3    ; R3 = h + i
  SUB  R0, R1, R3    ; return (f + g) - (h + i)
  MOV  PC, LR        ; return to caller
```

Code Example 6.26 demonstrates a nonleaf function `f1` and a leaf function `f2` including all the necessary saving and preserving of registers. Suppose `f1` keeps `i` in R4 and `x` in R5. `f2` keeps `r` in R4. `f1` uses preserved registers R4, R5, and LR, so it initially pushes them on the stack according to the callee save rule. It uses R12 to hold the intermediate result `(a – b)` so that it does not need to preserve another register for this calculation. Before calling `f2`, `f1` pushes R0 and R1 onto the stack according to the caller save rule because these are nonpreserved registers that `f2` might change and that `f1` will still need after the call. Although R12 is also a nonpreserved register that `f2` could overwrite, `f1` no longer needs R12 and doesn't have to save it. `f1` then passes the argument to `f2` in R0, makes the function call, and uses the result in R0. `f1` then restores R0 and R1 because it still needs them. When `f1` is done, it puts the return value in R0, restores preserved registers R4, R5, and LR, and returns. `f2` saves and restores R4 according to the callee save rule.

A nonleaf function overwrites LR when it calls another function using `BL`. Thus, a nonleaf function must always save LR on its stack and restore it before returning.

Code Example 6.26 NONLEAF FUNCTION CALL

High-Level Code

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}

int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; R0 = a, R1 = b, R4 = i, R5 = x
F1
   PUSH {R4, R5, LR}    ; save preserved registers used by f1
   ADD  R5, R0, R1      ; x = (a + b)
   SUB  R12, R0, R1     ; temp = (a - b)
   MUL  R5, R5, R12     ; x = x * temp = (a + b) * (a - b)
   MOV  R4, #0          ; i = 0
FOR
   CMP  R4, R0          ; i < a?
   BGE  RETURN          ; no: exit loop
   PUSH {R0, R1}        ; save nonpreserved registers
   ADD  R0, R1, R4      ; argument is b + i
   BL   F2              ; call f2(b+i)
   ADD  R5, R5, R0      ; x = x + f2(b+i)
   POP  {R0, R1}        ; restore nonpreserved registers
   ADD  R4, R4, #1      ; i++
   B    FOR             ; continue for loop
RETURN
   MOV  R0, R5          ; return value is x
   POP  {R4, R5, LR}    ; restore preserved registers
   MOV  PC, LR          ; return from f1

; R0 = p, R4 = r
F2
   PUSH {R4}            ; save preserved registers used by f2
   ADD  R4, R0, 5       ; r = p + 5
   ADD  R0, R4, R0      ; return value is r + p
   POP  {R4}            ; restore preserved registers
   MOV  PC, LR          ; return from f2
```

On careful inspection, one might note that `f2` does not modify R1, so `f1` did not need to save and restore it. However, a compiler cannot always easily ascertain which nonpreserved registers may be disturbed during a function call. Hence, a simple compiler will always make the caller save and restore any nonpreserved registers that it needs after the call.

An optimizing compiler could observe that `f2` is a leaf procedure and could allocate `r` to a nonpreserved register, avoiding the need to save and restore R4.

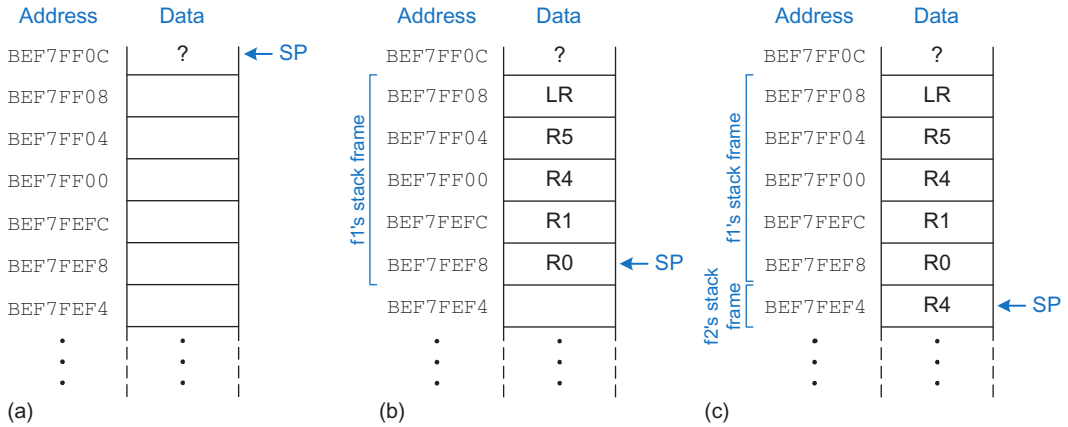


Figure 6.13 **The stack: (a) before function calls, (b) during `f1`, and (c) during `f2`**

Figure 6.13 shows the stack during execution of `f1`. The stack pointer originally starts at 0xBEF7FF0C.

Recursive Function Calls

A *recursive function* is a nonleaf function that calls itself. Recursive functions behave as both caller and callee and must save both preserved and nonpreserved registers. For example, the factorial function can be written as a recursive function. Recall that $factorial(n) = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$. The factorial function can be rewritten recursively as $factorial(n) = n \times factorial(n - 1)$, as shown in Code Example 6.27. The factorial of 1 is simply 1. To conveniently refer to program addresses, we show the program starting at address 0x8500.

According to the callee save rule, `factorial` is a nonleaf function and must save LR. According to the caller save rule, `factorial` will need `n` after calling itself, so it must save R0. Hence, it pushes both registers onto the stack at the start. It then checks whether `n` ≤ 1. If so, it puts the return value of 1 in R0, restores the stack pointer, and returns to the caller. It does not have to reload LR and R0 in this case, because they were never modified. If `n` > 1, the function recursively calls `factorial(n - 1)`. It then restores the value of `n` and the link register (LR) from the stack, performs the multiplication, and returns this result. Notice that the function cleverly restores `n` into R1, so as not to overwrite the returned value. The multiply instruction (`MUL R0, R1, R0`) multiplies `n` (R1) and the returned value (R0) and puts the result in R0.

Code Example 6.27 `factorial` RECURSIVE FUNCTION CALL

High-Level Code

```
int factorial(int n) {
  if (n <= 1)
    return 1;



  else
    return (n * factorial(n - 1));
}
```

ARM Assembly Code

```
0x8500  FACTORIAL  PUSH  {R0, LR}   ; push n and LR on stack
0x8504             CMP   R0, #1     ; R0 <= 1?
0x8508             BGT   ELSE       ; no: branch to else
0x850C             MOV   R0, #1     ; otherwise, return 1
0x8510             ADD   SP, SP, #8 ; restore SP
0x8514             MOV   PC, LR     ; return
0x8518  ELSE       SUB   R0, R0, #1 ; n = n - 1
0x851C             BL    FACTORIAL  ; recursive call
0x8520             POP   {R1, LR}   ; pop n (into R1) and LR
0x8524             MUL   R0, R1, R0 ; R0 = n * factorial(n - 1)
0x8528             MOV   PC, LR     ; return
```

Figure 6.14 shows the stack when executing `factorial(3)`. For illustration, we show SP initially pointing to 0xBEFF0FF0, as shown in Figure 6.14(a). The function creates a two-word stack frame to hold `n` (R0) and LR. On the first invocation, `factorial` saves R0 (holding `n` = 3) at 0xBEFF0FE8 and LR at 0xBEFF0FEC, as shown in Figure 6.14(b). The function then changes `n` to 2 and recursively calls `factorial(2)`, making LR hold 0x8520. On the second invocation, it saves R0 (holding n=2) at 0xBEFF0FE0 and LR at 0xBEFF0FE4. This time, we know that LR contains 0x8520. The function then changes `n` to 1 and recursively calls `factorial(1)`. On the third invocation, it saves R0 (holding `n` = 1) at

For clarity, we will always save registers at the start of a procedure call. An optimizing compiler might observe that there is no need to save R0 and LR when $n \leq 1$, and thus push registers only in the `ELSE` portion of the function.

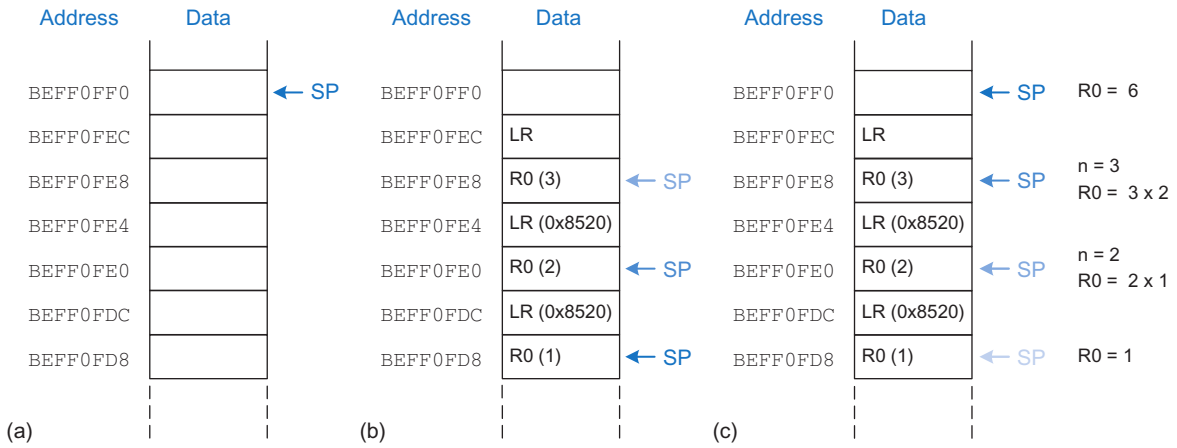


Figure 6.14 **Stack: (a) before, (b) during, and (c) after factorial function call with `n` = 3**

0xBEFF0FD8 and LR at 0xBEFF0FDC. This time, LR again contains 0x8520. The third invocation of `factorial` returns the value 1 in R0 and deallocates the stack frame before returning to the second invocation. The second invocation restores `n` (into R1) to 2, restores LR to 0x8520 (it happened to already have this value), deallocates the stack frame, and returns R0 = 2 × 1 = 2 to the first invocation. The first invocation restores `n` (into R1) to 3, restores LR to the return address of the caller, deallocates the stack frame, and returns R0 = 3 × 2 = 6. Figure 6.14(c) shows the stack as the recursively called functions return. When `factorial` returns to the caller, the stack pointer is in its original position (0xBEFF0FF0), none of the contents of the stack above the pointer have changed, and all of the preserved registers hold their original values. R0 holds the return value, 6.

Additional Arguments and Local Variables*

Functions may have more than four input arguments and may have too many local variables to keep in preserved registers. The stack is used to store this information. By ARM convention, if a function has more than four arguments, the first four are passed in the argument registers as usual. Additional arguments are passed on the stack, just above SP. The caller must expand its stack to make room for the additional arguments. Figure 6.15(a) shows the caller's stack for calling a function with more than four arguments.

A function can also declare local variables or arrays. Local variables are declared within a function and can be accessed only within that function. Local variables are stored in R4–R11; if there are too many local variables, they can also be stored in the function's stack frame. In particular, local arrays are stored on the stack.

Figure 6.15(b) shows the organization of a callee's stack frame. The stack frame holds the temporary registers and link register (if they need to be saved because of a subsequent function call), and any of the saved

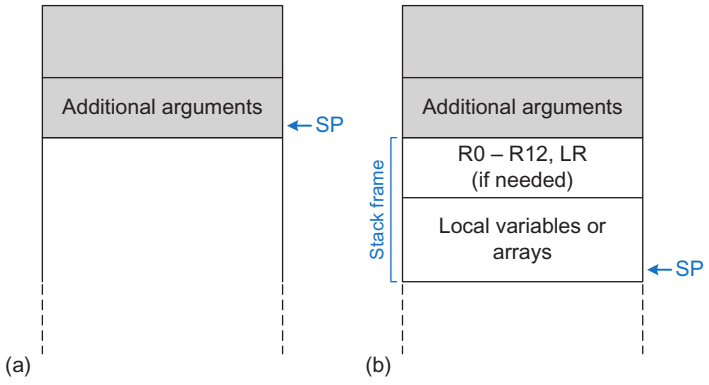


Figure 6.15 **Stack usage: (a) before and (b) after call**

registers that the function will modify. It also holds local arrays and any excess local variables. If the callee has more than four arguments, it finds them in the caller's stack frame. Accessing additional input arguments is the one exception in which a function can access stack data not in its own stack frame.

6.4 MACHINE LANGUAGE

Assembly language is convenient for humans to read. However, digital circuits understand only 1's and 0's. Therefore, a program written in assembly language is translated from mnemonics to a representation using only 1's and 0's called *machine language*. This section describes ARM machine language and the tedious process of converting between assembly and machine language.

ARM uses 32-bit instructions. Again, regularity supports simplicity, and the most regular choice is to encode all instructions as words that can be stored in memory. Even though some instructions may not require all 32 bits of encoding, variable-length instructions would add complexity. Simplicity would also encourage a single instruction format, but that is too restrictive. However, this issue allows us to introduce the last design principle:

Design Principle 4: Good design demands good compromises.

ARM makes the compromise of defining three main instruction formats: *Data-processing*, *Memory*, and *Branch*. This small number of formats allows for some regularity among instructions, and thus simpler decoder hardware, while also accommodating different instruction needs. Data-processing instructions have a first source register, a second source that is either an immediate or a register, possibly shifted, and a destination register. The Data-processing format has several variations for these second sources. Memory instructions have three operands: a base register, an offset that is either an immediate or an optionally shifted register, and a register that is the destination on an `LDR` and another source on an `STR`. Branch instructions take one 24-bit immediate branch offset. This section discusses these ARM instruction formats and shows how they are encoded into binary. Appendix B provides a quick reference for all the ARMv4 instructions.

6.4.1 Data-processing Instructions

The data-processing instruction format is the most common. The first source operand is a register. The second source operand can be an immediate or an optionally shifted register. A third register is the destination. Figure 6.16 shows the data-processing instruction format. The 32-bit instruction has six fields: *cond*, *op*, *funct*, *Rn*, *Rd*, and *Src2*.

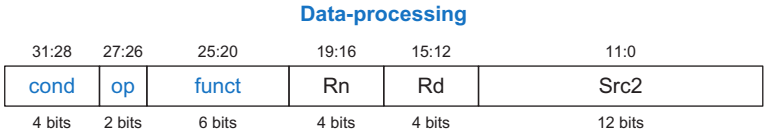


Figure 6.16 Data-processing instruction format

The operation the instruction performs is encoded in the fields highlighted in blue: *op* (also called the opcode or operation code) and *funct* or function code; the *cond* field encodes conditional execution based on flags described in Section 6.3.2. Recall that $cond = 1110_2$ for unconditional instructions. *op* is 00_2 for data-processing instructions.

The operands are encoded in the three fields: *Rn*, *Rd*, and *Src2*. *Rn* is the first source register and *Src2* is the second source; *Rd* is the destination register.

Figure 6.17 shows the format of the *funct* field and the three variations of *Src2* for data-processing instructions. *funct* has three subfields: *I*, *cmd*, and *S*. The *I*-bit is 1 when *Src2* is an immediate. The *S*-bit is 1 when the instruction sets the condition flags. For example, `SUBS R1, R9, #11` has $S = 1$. *cmd* indicates the specific data-processing instruction, as given in Table B.1 in Appendix B. For example, *cmd* is 4 (0100_2) for `ADD` and 2 (0010_2) for `SUB`.

Rd is short for "register destination." *Rn* and *Rm* unintuitively indicate the first and second register sources.

Three variations of *Src2* encoding allow the second source operand to be (1) an immediate, (2) a register (*Rm*) optionally shifted by a constant (*shamt5*), or (3) a register (*Rm*) shifted by another register (*Rs*). For the latter two encodings of *Src2*, *sh* encodes the type of shift to perform, as will be shown in Table 6.8.

Data-processing instructions have an unusual immediate representation involving an 8-bit unsigned immediate, *imm8*, and a 4-bit rotation, *rot*. *imm8* is rotated right by $2 \times rot$ to create a 32-bit constant. Table 6.7 gives example rotations and resulting 32-bit constants for the 8-bit immediate 0xFF. This representation is valuable because it

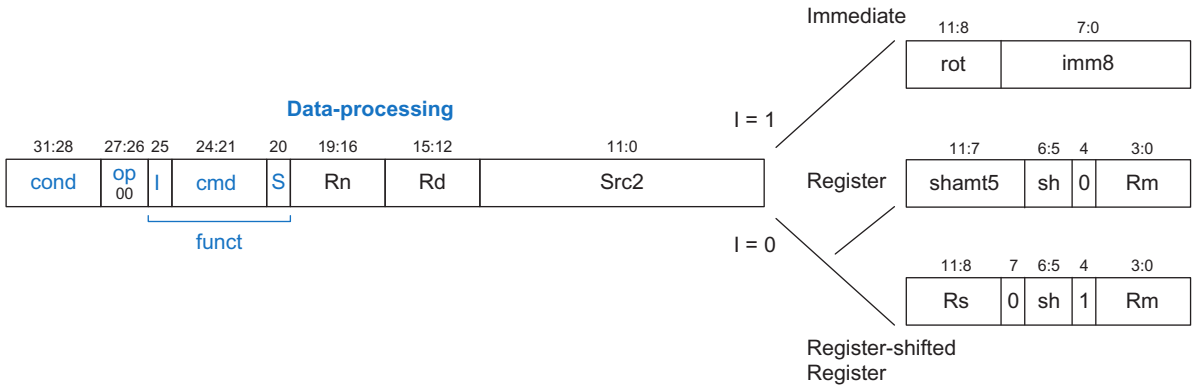


Figure 6.17 Data-processing instruction format showing the *funct* field and *Src2* variations

Table 6.7 Immediate rotations and resulting 32-bit constant for *imm8* = 0xFF

rot	32-bit Constant
0000	0000 0000 0000 0000 0000 0000 1111 1111
0001	1100 0000 0000 0000 0000 0000 0011 1111
0010	1111 0000 0000 0000 0000 0000 0000 1111
…	…
1111	0000 0000 0000 0000 0000 0011 1111 1100

If an immediate has multiple possible encodings, the representation with the smallest rotation value *rot* is used. For example, #12 would be represented as (*rot*, *imm8*) = (0000, 00001100), not (0001, 00110000).

permits many useful constants, including small multiples of any power of two, to be packed into a small number of bits. Section 6.6.1 describes how to generate arbitrary 32-bit constants.

Figure 6.18 shows the machine code for `ADD` and `SUB` when *Src2* is a register. The easiest way to translate from assembly to machine code is to write out the values of each field and then convert these values to binary. Group the bits into blocks of four to convert to hexadecimal to make the machine language representation more compact. Beware that the destination is the first register in an assembly language instruction, but it is the second register field (*Rd*) in the machine language instruction. *Rn* and *Rm* are the first and second source operands, respectively. For example, the assembly instruction `ADD R5, R6, R7` has *Rn* = 6, *Rd* = 5, and *Rm* = 7.

Figure 6.19 shows the machine code for `ADD` and `SUB` with an immediate and two register operands. Again, the destination is the first

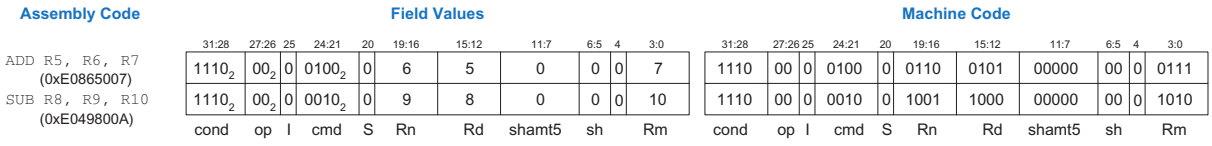


Figure 6.18 Data-processing instructions with three register operands

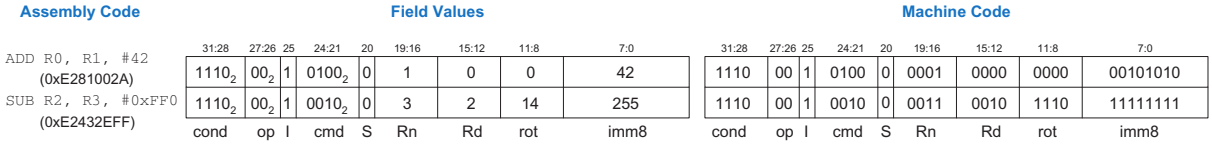


Figure 6.19 Data-processing instructions with an immediate and two register operands

register in an assembly language instruction, but it is the second register field (*Rd*) in the machine language instruction. The immediate of the `ADD` instruction (42) can be encoded in 8 bits, so no rotation is needed (*imm8* = 42, *rot* = 0). However, the immediate of `SUB R2, R3, 0xFF0` cannot be encoded directly using the 8 bits of *imm8*. Instead, *imm8* is 255 (0xFF), and it is rotated right by 28 bits (*rot* = 14). This is easiest to interpret by remembering that the right rotation by 28 bits is equivalent to a left rotation by 32−28 = 4 bits.

Shifts are also data-processing instructions. Recall from Section 6.3.1 that the amount by which to shift can be encoded using either a 5-bit immediate or a register.

Figure 6.20 shows the machine code for logical shift left (`LSL`) and rotate right (`ROR`) with immediate shift amounts. The *cmd* field is 13 (1101_2) for all shift instruction, and the shift field (*sh*) encodes the type of shift to perform, as given in Table 6.8. *Rm* (i.e., R5) holds the 32-bit value to be shifted, and *shamt5* gives the number of bits to shift. The shifted result is placed in *Rd*. *Rn* is not used and should be 0.

Figure 6.21 shows the machine code for `LSR` and `ASR` with the shift amount encoded in the least significant 8 bits of *Rs* (R6 and R12). As

Table 6.8 *sh* field encodings

Instruction	sh	Operation
`LSL`	00_2	Logical shift left
`LSR`	01_2	Logical shift right
`ASR`	10_2	Arithmetic shift right
`ROR`	11_2	Rotate right

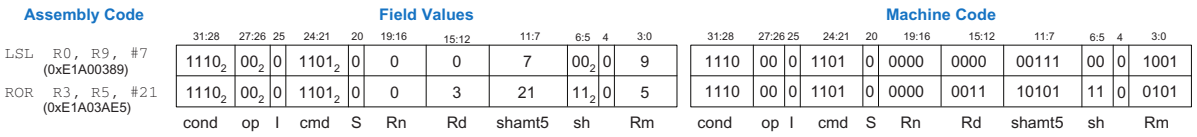


Figure 6.20 Shift instructions with immediate shift amounts

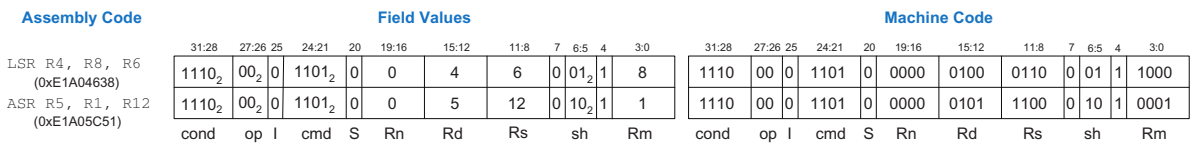


Figure 6.21 Shift instructions with register shift amounts

Table 6.9 Offset type control bits for memory instructions

Bit	Meaning	
	$\overline{I}$	U
0	Immediate offset in Src2	Subtract offset from base
1	Register offset in Src2	Add offset to base

before, *cmd* is 13 (1101_2), *sh* encodes the type of shift, *Rm* holds the value to be shifted, and the shifted result is placed in *Rd*. This instruction uses the *register-shifted register* addressing mode, where one register (*Rm*) is shifted by the amount held in a second register (*Rs*). Because the least significant 8 bits of *Rs* are used, *Rm* can be shifted by up to 255 positions. For example, if *Rs* holds the value 0xF001001C, the shift amount is 0x1C (28). A logical shift by more than 31 bits pushes all the bits off the end and produces all 0's. Rotate is cyclical, so a rotate by 50 bits is equivalent to a rotate by 18 bits.

6.4.2 Memory Instructions

Memory instructions use a format similar to that of data-processing instructions, with the same six overall fields: *cond*, *op*, *funct*, *Rn*, *Rd*, and *Src2*, as shown in Figure 6.22. However, memory instructions use a different *funct* field encoding, have two variations of *Src2*, and use an *op* of 01_2. *Rn* is the base register, *Src2* holds the offset, and *Rd* is the destination register in a load or the source register in a store. The offset is either a 12-bit unsigned immediate (*imm12*) or a register (*Rm*) that is optionally shifted by a constant (*shamt5*). *funct* is composed of six control bits: $\overline{I}$, *P*, *U*, *B*, *W*, and *L*. The $\overline{I}$ (immediate) and *U* (add) bits determine whether the offset is an immediate or register and whether it should be added or subtracted, according to Table 6.9. The *P* (pre-index) and *W* (writeback) bits specify the index mode according to Table 6.10. The *L* (load) and *B* (byte) bits specify the type of memory operation according to Table 6.11.

Table 6.10 Index mode control bits for memory instructions

P	W	Index Mode
0	0	Post-index
0	1	Not supported
1	0	Offset
1	1	Pre-index

Table 6.11 Memory operation type control bits for memory instructions

L	B	Instruction
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

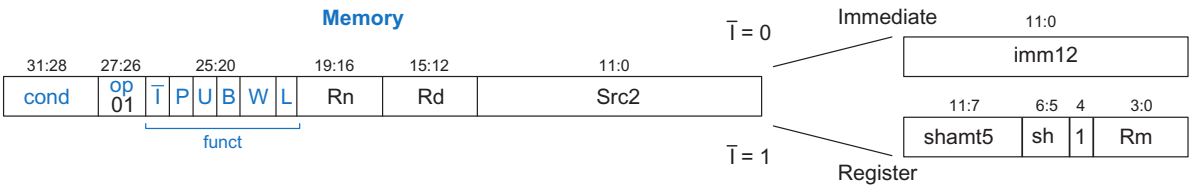


Figure 6.22 Memory instruction format for LDR, STR, LDRB, and STRB

Example 6.3 TRANSLATING MEMORY INSTRUCTIONS INTO MACHINE LANGUAGE

Translate the following assembly language statement into machine language.

```
STR R11, [R5], #-26
```

Notice the counterintuitive encoding of post-indexing mode.

Solution: `STR` is a memory instruction, so it has an *op* of 01_2. According to Table 6.11, $L = 0$ and $B = 0$ for `STR`. The instruction uses post-indexing, so according to Table 6.10, $P = 0$ and $W = 0$. The immediate offset is subtracted from the base, so $\overline{I} = 0$ and $U = 0$. Figure 6.23 shows each field and the machine code. Hence, the machine language instruction is 0xE405B01A.

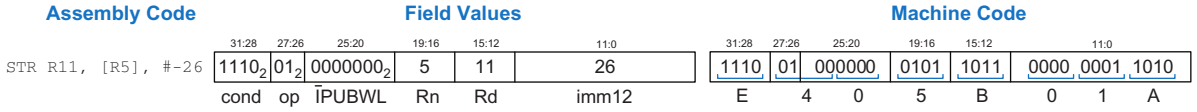


Figure 6.23 **Machine code for the memory instruction of Example 6.3**

6.4.3 Branch Instructions

Branch instructions use a single 24-bit signed immediate operand, *imm24*, as shown in Figure 6.24. As with data-processing and memory instructions, branch instructions begin with a 4-bit condition field and a 2-bit *op*, which is 10_2. The *funct* field is only 2 bits. The upper bit of *funct* is always 1 for branches. The lower bit, *L*, indicates the type of branch operation: 1 for `BL` and 0 for `B`. The remaining 24-bit two's complement *imm24* field is used to specify an instruction address relative to PC + 8.

Code Example 6.28 shows the use of the branch if less than (`BLT`) instruction and Figure 6.25 shows the machine code for that instruction. The *branch target address* (*BTA*) is the address of the next instruction to execute if the branch is taken. The `BLT` instruction in Figure 6.25 has a BTA of 0x80B4, the instruction address of the `THERE` label.

The 24-bit immediate field gives the number of instructions between the BTA and PC + 8 (two instructions past the branch). In this case, the value in the immediate field (*imm24*) of `BLT` is 3 because the BTA (0x80B4) is three instructions past PC + 8 (0x80A8).

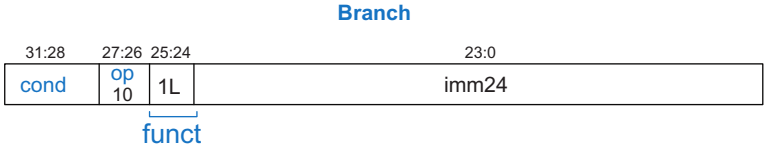


Figure 6.24 **Branch instruction format**

Code Example 6.28 CALCULATING THE BRANCH TARGET ADDRESS

ARM Assembly Code

```
0x80A0          BLT THERE
0x80A4          ADD R0, R1, R2
0x80A8          SUB R0, R0, R9
0x80AC          ADD SP, SP, #8
0x80B0          MOV PC, LR
0x80B4 THERE    SUB R0, R0, #1
0x80B8          ADD R3, R3, #0x5
```

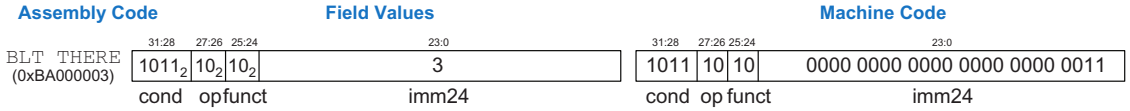


Figure 6.25 Machine code for branch if less than (BLT)

The processor calculates the BTA from the instruction by sign-extending the 24-bit immediate, shifting it left by 2 (to convert words to bytes), and adding it to PC + 8.

Example 6.4 CALCULATING THE IMMEDIATE FIELD FOR PC-RELATIVE ADDRESSING

Calculate the immediate field and show the machine code for the branch instruction in the following assembly program.

```
0x8040 TEST   LDRB  R5, [R0, R3]
0x8044        STRB  R5, [R1, R3]
0x8048        ADD   R3, R3, #1
0x8044        MOV   PC, LR
0x8050        BL    TEST
0x8054        LDR   R3, [R1], #4
0x8058        SUB   R4, R3, #9
```

Solution: Figure 6.26 shows the machine code for the branch and link instruction (BL). Its branch target address (0x8040) is six instructions behind PC + 8 (0x8058), so the immediate field is -6.

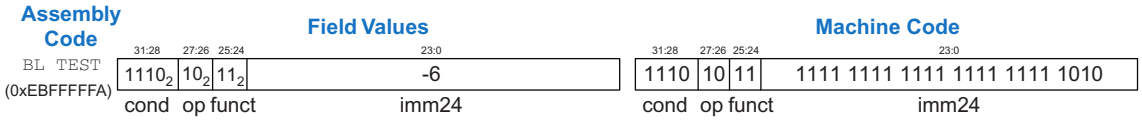


Figure 6.26 BL machine code

ARM is unusual among RISC architectures in that it allows the second source operand to be shifted in register and base addressing modes. This requires a shifter in series with the ALU in the hardware implementation but significantly reduces code length in common programs, especially array accesses. For example, in an array of 32-bit data elements, the array index must be left-shifted by 2 to compute the byte offset into the array. Any type of shift is permitted, but left shifts for multiplication are most common.

6.4.4 Addressing Modes

This section summarizes the modes used for addressing instruction operands. ARM uses four main modes: register, immediate, base, and PC-relative addressing. Most other architectures provide similar addressing modes, so understanding these modes helps you easily learn other assembly languages. Register and base addressing have several submodes described below. The first three modes (register, immediate, and base addressing) define modes of reading and writing operands. The last mode (PC-relative addressing) defines a mode of writing the program counter (PC). Table 6.12 summarizes and gives examples of each addressing mode.

Data-processing instructions use register or immediate addressing, in which the first source operand is a register and the second is a register or immediate, respectively. ARM allows the second register to be optionally shifted by an amount specified in an immediate or a third register. Memory instructions use base addressing, in which the base address comes from a register and the offset comes from an immediate, a register, or a register shifted by an immediate. Branches use PC-relative addressing in which the branch target address is computed by adding an offset to PC + 8.

6.4.5 Interpreting Machine Language Code

To interpret machine language, one must decipher the fields of each 32-bit instruction word. Different instructions use different formats, but all

Table 6.12 ARM operand addressing modes

Operand Addressing Mode	Example	Description
Register		
Register-only	`ADD R3, R2, R1`	R3 ← R2 + R1
Immediate-shifted register	`SUB R4, R5, R9, LSR #2`	R4 ← R5 – (R9 >> 2)
Register-shifted register	`ORR R0, R10, R2, ROR R7`	R0 ← R10 | (R2 ROR R7)
Immediate	`SUB R3, R2, #25`	R3 ← R2 – 25
Base		
Immediate offset	`STR R6, [R11, #77]`	mem[R11+77] ← R6
Register offset	`LDR R12, [R1, –R5]`	R12 ← mem[R1 – R5]
Immediate-shifted register offset	`LDR R8, [R9, R2, LSL #2]`	R8 ← mem[R9 + (R2 << 2)]
PC-Relative	`B LABEL1`	Branch to `LABEL1`

formats start with a 4-bit condition field and a 2-bit *op*. The best place to begin is to look at the *op*. If it is 00_2, then the instruction is a data-processing instruction; if it is 01_2, then the instruction is a memory instruction; if it is 10_2, then it is a branch instruction. Based on that, the rest of the fields can be interpreted.

Example 6.5 TRANSLATING MACHINE LANGUAGE TO ASSEMBLY LANGUAGE

Translate the following machine language code into assembly language.

```
0xE0475001
0xE5949010
```

Solution: First, we represent each instruction in binary and look at bits 27:26 to find the *op* for each instruction, as shown in Figure 6.27. The *op* fields are 00_2 and 01_2, indicating a data-processing and memory instruction, respectively. Next, we look at the *funct* field of each instruction.

The *cmd* field of the data-processing instruction is 2 (0010_2) and the *I*-bit (bit 25) is 0, indicating that it is a `SUB` instruction with a register *Src2*. *Rd* is 5, *Rn* is 7, and *Rm* is 1.

The *funct* field for the memory instruction is 011001_2. $B = 0$ and $L = 1$, so this is an `LDR` instruction. $P = 1$ and $W = 0$, indicating offset addressing. $\overline{I} = 0$, so the offset is an immediate. $U = 1$, so the offset is added. Thus, it is a load register instruction with an immediate offset that is added to the base register. *Rd* is 9, *Rn* is 4, and *imm12* is 16. Figure 6.27 shows the assembly code equivalent of the two machine instructions.

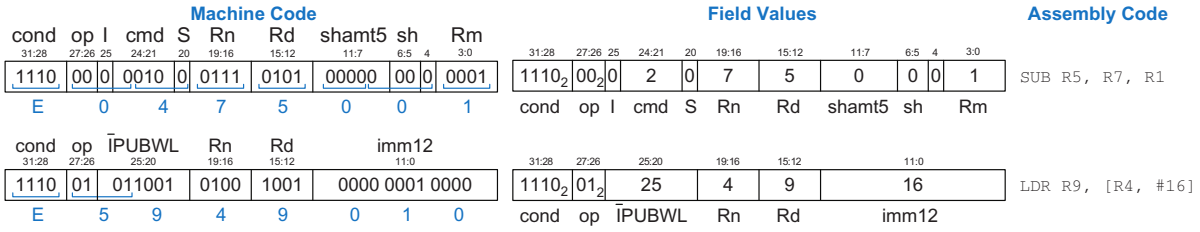


Figure 6.27 Machine code to assembly code translation

6.4.6 The Power of the Stored Program

A program written in machine language is a series of 32-bit numbers representing the instructions. Like other binary numbers, these instructions can be stored in memory. This is called the *stored program* concept, and it is a key reason why computers are so powerful. Running a different

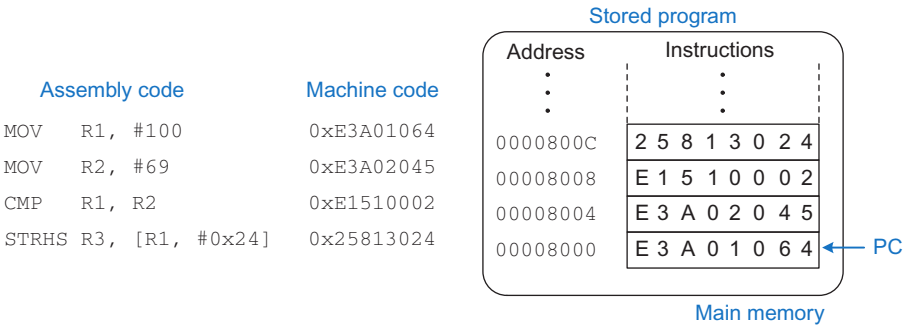


Figure 6.28 Stored program

program does not require large amounts of time and effort to reconfigure or rewire hardware; it only requires writing the new program to memory. In contrast to dedicated hardware, the stored program offers general-purpose computing. In this way, a computer can execute applications ranging from a calculator to a word processor to a video player simply by changing the stored program.

Instructions in a stored program are retrieved, or *fetched*, from memory and executed by the processor. Even large, complex programs are simply a series of memory reads and instruction executions.

Figure 6.28 shows how machine instructions are stored in memory. In ARM programs, the instructions are normally stored starting at low addresses, in this case 0x00008000. Remember that ARM memory is byte-addressable, so 32-bit (4-byte) instruction addresses advance by 4 bytes, not 1.

To run or execute the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in a 32-bit register called the program counter (PC), which is register R15. For historical reasons, a read to the PC returns the address of the current instruction plus 8.

To execute the code in Figure 6.28, the PC is initialized to address 0x00008000. The processor fetches the instruction at that memory address and executes the instruction, 0xE3A01064 (`MOV R1, #100`). The processor then increments the PC by 4 to 0x00008004, fetches and executes that instruction, and repeats.

The *architectural state* of a microprocessor holds the state of a program. For ARM, the architectural state includes the register file and status registers. If the operating system (OS) saves the architectural state at some point in the program, it can interrupt the program, do something else, and then restore the state such that the program continues properly, unaware that it was ever interrupted. The architectural state is also of great importance when we build a microprocessor in Chapter 7.

Ada Lovelace, 1815–1852. A British mathematician who wrote the first computer program. It calculated the Bernoulli numbers using Charles Babbage's Analytical Engine. She was the daughter of the poet Lord Byron.

6.5 LIGHTS, CAMERA, ACTION: COMPILING, ASSEMBLING, AND LOADING*

Until now, we have shown how to translate short high-level code snippets into assembly and machine code. This section describes how to compile and assemble a complete high-level program and how to load the program into memory for execution. We begin by introducing an example ARM *memory map*, which defines where code, data, and stack memory are located.

Figure 6.29 shows the steps required to translate a program from a high-level language into machine language and to start executing that program. First, a *compiler* translates the high-level code into assembly code. The *assembler* translates the assembly code into machine code and puts it in an object file. The *linker* combines the machine code with code from libraries and other files and determines the proper branch addresses and variable locations to produce an entire executable program. In practice, most compilers perform all three steps of compiling, assembling, and linking. Finally, the *loader* loads the program into memory and starts execution. The remainder of this section walks through these steps for a simple program.

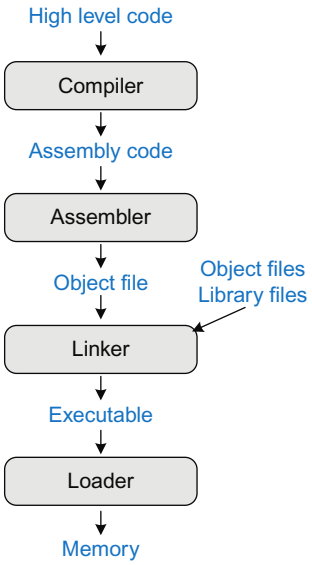


Figure 6.29 Steps for translating and starting a program

6.5.1 The Memory Map

With 32-bit addresses, the ARM address space spans 2^{32} bytes (4 GB). Word addresses are multiples of 4 and range from 0 to 0xFFFFFFFC. Figure 6.30 shows an example memory map. The ARM architecture divides the address space into five parts or segments: the text segment,

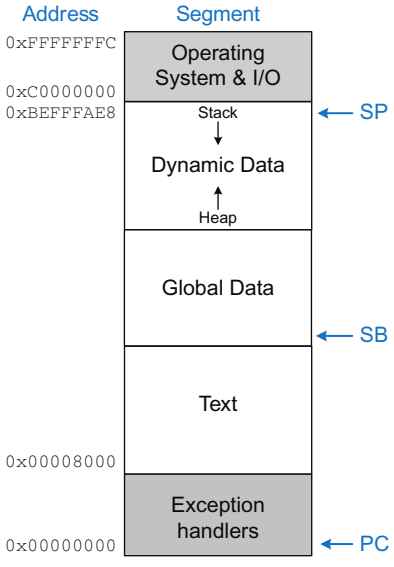


Figure 6.30 Example ARM memory map

We present an example ARM memory map here; however, in ARM, the memory map is somewhat flexible. While the exception vector table must be located at 0x0 and memory-mapped I/O is typically located at the high memory addresses, the user can define where the text (code and constant data), stack, and global data are placed. Moreover, at least historically, most ARM systems have less than 4 GB of memory.

global data segment, dynamic data segment, and segments for exception handlers, the operating system (OS) and input/output (I/O). The following sections describe each segment.

The Text Segment

The *text segment* stores the machine language program. ARM also calls this the *read-only* (RO) *segment*. In addition to code, it may include literals (constants) and read-only data.

The Global Data Segment

The *global data segment* stores global variables that, in contrast to local variables, can be accessed by all functions in a program. Global variables are allocated in memory before the program begins executing. ARM also calls this the *read/write* (RW) *segment*. Global variables are typically accessed using a *static base* register that points to the start of the global segment. ARM conventionally uses R9 as the static base pointer (SB).

The Dynamic Data Segment

The *dynamic data segment* holds the stack and the heap. The data in this segment is not known at start-up but is dynamically allocated and deallocated throughout the execution of the program.

Upon start-up, the operating system sets up the stack pointer (SP) to point to the top of the stack. The stack typically grows downward, as shown here. The stack includes temporary storage and local variables, such as arrays, that do not fit in the registers. As discussed in Section 6.3.7, functions also use the stack to save and restore registers. Each stack frame is accessed in last-in-first-out order.

The *heap* stores data that is allocated by the program during runtime. In C, memory allocations are made by the `malloc` function; in C++ and Java, `new` is used to allocate memory. Like a heap of clothes on a dorm room floor, heap data can be used and discarded in any order. The heap typically grows upward from the bottom of the dynamic data segment.

If the stack and heap ever grow into each other, the program's data can become corrupted. The memory allocator tries to ensure that this never happens by returning an out-of-memory error if there is insufficient space to allocate more dynamic data.

The Exception Handler, OS, and I/O Segments

The lowest part of the ARM memory map is reserved for the exception vector table and exception handlers, starting at address 0x0 (see Section 6.6.3). The highest part of the memory map is reserved for the operating system and memory-mapped I/O (see Section 9.2).

Grace Hopper, 1906–1992. Graduated from Yale University with a Ph.D. in mathematics. Developed the first compiler while working for the Remington Rand Corporation and was instrumental in developing the COBOL programming language. As a naval officer, she received many awards, including a World War II Victory Medal and the National Defense Service Medal.

6.5.2 Compilation

A compiler translates high-level code into assembly language. The examples in this section are based on GCC, a popular and widely used free compiler, running on the Raspberry Pi single-board computer

Code Example 6.29 COMPILING A HIGH-LEVEL PROGRAM

High-Level Code

```
int f, g, y; // global variables


int sum(int a, int b) {
  return (a + b);
}


int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}
```

ARM Assembly Code

```
  .text
  .global sum
  .type   sum, %function
sum:
  add     r0, r0, r1
  bx      lr
  .global main
  .type   main, %function
main:
  push    {r3, lr}
  mov     r0, #2
  ldr     r3, .L3
  str     r0, [r3, #0]
  mov     r1, #3
  ldr     r3, .L3+4
  str     r1, [r3, #0]
  bl      sum
  ldr     r3, .L3+8
  str     r0, [r3, #0]
  pop     {r3, pc}
.L3:
  .word   f
  .word   g
  .word   y
```

In Code Example 6.29, global variables are accessed using two memory instructions: one to load the *address* of the variable, and a second to read or write the variable. The addresses of the global variables are placed after the code, starting at label `.L3`. `LDR R3, .L3` loads the *address* of `f` into `R3`, and `STR R0, [R3, #0]` writes to `f`; `LDR R3, .L3+4` loads the *address* of `g` into `R3`, and `STR R1, [R3, #0]` writes to `g`, and so on. Section 6.6.1 describes this assembly code construct further.

(see Section 9.3). Code Example 6.29 shows a simple high-level program with three global variables and two functions, along with the assembly code produced by GCC.

To compile, assemble, and link a C program named `prog.c` with GCC, use the command:

```
gcc -O1 -g prog.c -o prog
```

This command produces an executable output file called `prog`. The `-O1` flag asks the compiler to perform basic optimizations rather than producing grossly inefficient code. The `-g` flag tells the compiler to include debugging information in the file.

To see the intermediate steps, we can use GCC's `-S` flag to compile but not assemble or link.

```
gcc -O1 -S prog.c -o prog.s
```

The output, `prog.s`, is rather verbose, but the interesting parts are shown in Code Example 6.29. Note that GCC requires labels to be followed by a colon. The GCC output is in lowercase and has other assembler directives not discussed here. Observe that `sum` returns using the `BX` instruction rather than `MOV PC, LR`. Also, observe that GCC elected to save and restore R3 even though it is not one of the preserved registers. The addresses of the global variables will be stored in a table starting at label .L3.

6.5.3 Assembling

An assembler turns the assembly language code into an *object file* containing machine language code. GCC can create the object file from either `prog.s` or directly from `prog.c` using

```
gcc -c prog.s -o prog.o
```

or

```
gcc -O1 -g -c prog.c -o prog.o
```

The assembler makes two passes through the assembly code. On the first pass, the assembler assigns instruction addresses and finds all the symbols, such as labels and global variable names. The names and addresses of the symbols are kept in a symbol table. On the second pass through the code, the assembler produces the machine language code. Addresses for labels are taken from the symbol table. The machine language code and symbol table are stored in the object file.

We can *disassemble* the object file using the `objdump` command to see the assembly language code beside the machine language code. If the code was originally compiled with `–g`, the disassembler also shows the corresponding lines of C code:

```
objdump -S prog.o
```

The following shows the disassembly of section .text:

```
00000000 <sum>:
int sum(int a, int b) {
    return (a + b);
}
  0:  e0800001   add  r0, r0, r1
  4:  e12fff1e   bx  lr

00000008 <main>:

int f, g, y; // global variables

int sum(int a, int b);

int main(void) {
  8:   e92d4008   push  {r3, lr}
       f = 2;
  c:   e3a00002   mov   r0, #2
 10:   e59f301c   ldr   r3, [pc, #28]  ; 34 <main+0x2c>
 14:   e5830000   str   r0, [r3]
       g = 3;
 18:   e3a01003   mov   r1, #3
 1c:   e59f3014   ldr   r3, [pc, #20]  ; 38 <main+0x30>
 20:   e5831000   str   r1, [r3]
       y = sum(f,g);
 24:   ebffffe   bl    0 <sum>
```

Recall from Section 6.4.6 that a read to PC returns the address of the current instruction plus 8. So, `LDR R3, [PC, #28]` loads `f`'s address, which is just after the code at: (PC + 8) + 28 = (0x10 + 0x8) + 0x1C = 0x34.

```
  28:  e59f300c   ldr   r3, [pc, #12]  ; 3c <main+0x34>
  2c:  e5830000   str   r0, [r3]
  return y;
}
  30:  e8bd8008   pop  {r3, pc}
  ...
```

We can also view the symbol table from the object file using `objdump` with the `-t` flag. The interesting parts are shown below. Observe that the `sum` function starts at address 0 and has a size of 8 bytes. `main` starts at address 8 and has size 0x38. The global variable symbols `f`, `g`, and `h` are listed and are 4 bytes each, but they have not yet been assigned addresses.

```
objdump -t prog.o

SYMBOL TABLE:
00000000 l d .text  00000000 .text
00000000 l d .data  00000000 .data
00000000 g F .text  00000008 sum
00000008 g F .text  00000038 main
00000004   O *COM*  00000004 f
00000004   O *COM*  00000004 g
00000004   O *COM*  00000004 y
```

6.5.4 Linking

Most large programs contain more than one file. If the programmer changes only one of the files, it would be wasteful to recompile and reassemble the other files. In particular, programs often call functions in library files; these library files almost never change. If a file of high-level code is not changed, the associated object file need not be updated. Also, a program typically involves some start-up code to initialize the stack, heap, and so forth, that must be executed before calling the `main` function.

The job of the linker is to combine all of the object files and the start-up code into one machine language file called the *executable* and assign addresses for global variables. The linker relocates the data and instructions in the object files so that they are not all on top of each other. It uses the information in the symbol tables to adjust the code based on the new label and global variable addresses. Invoke GCC to link the object file using:

```
gcc prog.o -o prog
```

We can again disassemble the executable using:

```
objdump -S -t prog
```

The start-up code is too lengthy to show, but our program begins at address 0x8390 in the text segment and the global variables are assigned addresses

starting at 0x10570 in the global segment. Notice the `.word` assembler directives defining the addresses of the global variables `f`, `g`, and `y`.

```
00008390 <sum>:

int sum(int a, int b) {
  return (a + b);
}
  8390:  e0800001   add  r0, r0, r1
  8394:  e12fff1e   bx  lr

00008398 <main>:

int f, g, y; // global variables

int sum(int a, int b);

int main(void) {
  8398:  e92d4008   push  {r3, lr}
  f = 2;
  839c:  e3a00002   mov  r0, #2
  83a0:  e59f301c   ldr  r3, [pc, #28]  ; 83c4 <main+0x2c>
  83a4:  e5830000   str  r0, [r3]
  g = 3;
  83a8:  e3a01003   mov  r1, #3
  83ac:  e59f3014   ldr  r3, [pc, #20]  ; 83c8 <main+0x30>
  83b0:  e5831000   str  r1, [r3]
  y = sum(f,g);
  83b4:  ebfffff5   bl  8390 <sum>
  83b8:  e59f300c   ldr  r3, [pc, #12]  ; 83cc <main+0x34>
  83bc:  e5830000   str  r0, [r3]
  return y;
}
  83c0:  e8bd8008   pop   {r3, pc}
  83c4:  00010570   .word  0x00010570
  83c8:  00010574   .word  0x00010574
  83cc:  00010578   .word  0x00010578
```

The instruction `LDR R3, [PC, #28]` in the executable loads from address (PC + 8) + 28 = (0x83A0 + 0x8) + 0x1C = 0x83C4. This memory address contains the value 0x10570, the location of global variable `f`.

The executable also contains an updated symbol table with the relocated addresses of the functions and global variables.

```
SYMBOL TABLE:
000082e4 l  d  .text  00000000    .text
00010564 l  d  .data  00000000    .data
00008390 g    F .text  00000008    sum
00008398 g    F .text  00000038    main
00010570 g    O .bss   00000004    f
00010574 g    O .bss   00000004    g
00010578 g    O .bss   00000004    y
```

6.5.5 Loading

The operating system loads a program by reading the text segment of the executable file from a storage device (usually the hard disk) into the text segment of memory. The operating system jumps to the beginning of the program to begin executing. Figure 6.31 shows the memory map at the beginning of program execution.

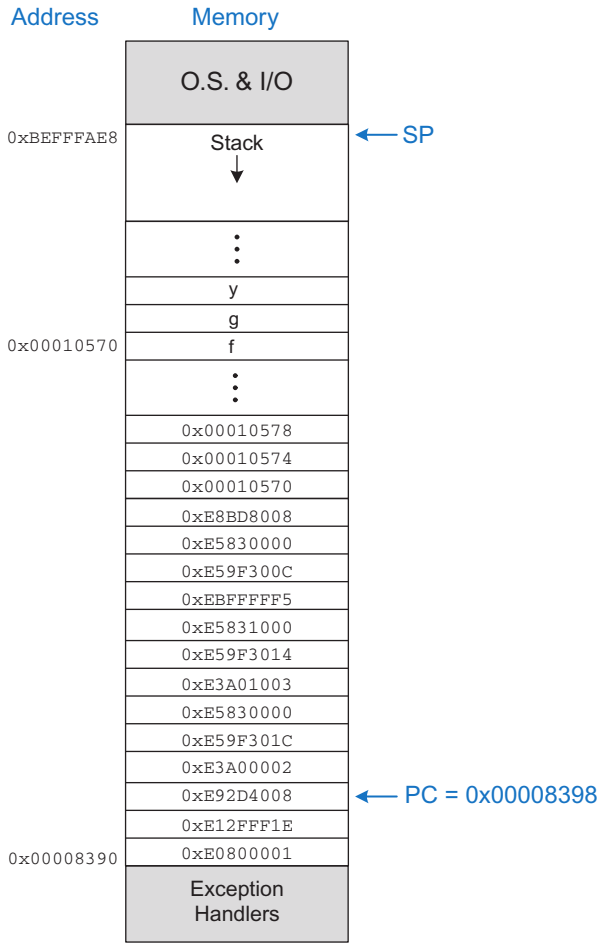


Figure 6.31 Executable loaded in memory

6.6 ODDS AND ENDS*

This section covers a few optional topics that do not fit naturally elsewhere in the chapter. These topics include loading 32-bit literals, NOPs, and exceptions.

6.6.1 Loading Literals

Many programs need to load 32-bit literals, such as constants or addresses. `MOV` only accepts a 12-bit source, so the `LDR` instruction is used to load these numbers from a *literal pool* in the text segment. ARM assemblers accept loads of the form

```
LDR Rd, =literal
LDR Rd, =label
```

The first loads a 32-bit constant specified by `literal`, and the second loads the address of a variable or pointer in the program specified by `label`. In both cases, the value to load is kept in a *literal pool*, which is a portion of the text segment containing literals. The literal pool must be less than 4096 bytes from the `LDR` instruction so that the load can be performed as `LDR Rd, [PC, #offset_to_literal]`. The program must be careful to branch around the literal pool because executing literals would be nonsensical or worse.

Code Example 6.30 illustrates loading a literal. As shown in Figure 6.32, suppose the `LDR` instruction is at address 0x8110 and the literal is at 0x815C. Remember that reading the PC returns the address 8 bytes beyond the current instruction being executed. Hence, when the `LDR` is executed, reading the PC returns 0x8118. Thus, the `LDR` uses an offset of 0x44 to find the literal pool: `LDR R1, [PC, #0x44]`.

Code Example 6.30 LARGE IMMEDIATE USING A LITERAL POOL

High-level code	ARM Assembly Code

`int a = 0x2B9056F;`	`; R1 = a`
	`LDR R1, =0x2B9056F`
	`...`

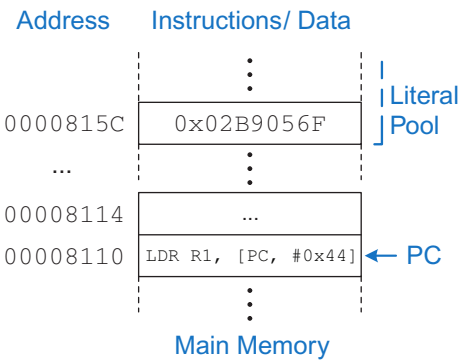


Figure 6.32 Example literal pool

Pseudoinstructions are not actually part of the instruction set but are shorthand for instructions or instruction sequences that are commonly used by programmers and compilers. The assembler translates pseudoinstructions into one or more actual instructions.

6.6.2 NOP

NOP is a mnemonic for "no operation" and is pronounced "no op." It is a *pseudoinstruction* that does nothing. The assembler translates it to `MOV R0, R0` (0xE1A00000). NOPs are useful to, among other things, achieve some delay or align instructions.

6.6.3 Exceptions

An *exception* is like an unscheduled function call that branches to a new address. Exceptions may be caused by hardware or software. For example, the processor may receive notification that the user pressed a key on a keyboard. The processor may stop what it is doing, determine which key was pressed, save it for future reference, and then resume the program that was running. Such a hardware exception triggered by an input/output (I/O) device such as a keyboard is often called an *interrupt*. Alternatively, the program may encounter an error condition such as an undefined instruction. The program then branches to code in the operating system (OS), which may choose to either emulate the unimplemented instruction or terminate the offending program. Software exceptions are sometimes called *traps*. A particularly important form of a trap is a *system call*, whereby the program invokes a function in the OS running at a higher privilege level. Other causes of exceptions include reset and attempts to read nonexistent memory.

Like any other function call, an exception must save the return address, jump to some address, do its work, clean up after itself, and return to the program where it left off. Exceptions use a *vector* table to determine where to jump to the *exception handler* and use *banked registers* to maintain extra copies of key registers so that they will not corrupt the registers in the active program. Exceptions also change the *privilege level* of the program, allowing the exception handler to access protected parts of memory.

Execution Modes and Privilege Levels

An ARM processor can operate in one of several execution modes with different privilege levels. The different modes allow an exception to take place in an exception handler without corrupting state; for example, an interrupt could occur while the processor is executing operating system code in Supervisor mode, and a subsequent Abort exception could occur if the interrupt attempted to access an invalid memory address. The exception handlers would eventually return and resume the supervisor code. The mode is specified in the bottom bits of the Current Program Status Register (CPSR), as was shown in Figure 6.6. Table 6.13 lists execution modes and their encodings. User mode operates at privilege level PL0, which is unable to access protected portions of memory such as the operating system code. The other modes operate at privilege level PL1, which can access all system resources. Privilege levels are important so that buggy or malicious user code cannot corrupt other programs or crash or infect the system.

Exception Vector Table

When an exception occurs, the processor branches to an offset in the *exception vector table*, depending on the cause of the exception. Table 6.14 describes the vector table, which is normally located starting at address 0x00000000 in memory. For example, when an interrupt occurs, the processor branches to address 0x00000018. Similarly, on

Table 6.13 ARM execution modes

Mode	$CPSR_{4:0}$
User	10000
Supervisor	10011
Abort	10111
Undefined	11011
Interrupt (IRQ)	10010
Fast Interrupt (FIQ)	10001

Table 6.14 Exception vector table

Exception	Address	Mode
Reset	0x00	Supervisor
Undefined Instruction	0x04	Undefined
Supervisor Call	0x08	Supervisor
Prefetch Abort (instruction fetch error)	0x0C	Abort
Data Abort (data load or store error)	0x10	Abort
Reserved	0x14	N/A
Interrupt	0x18	IRQ
Fast Interrupt	0x1C	FIQ

ARM also supports a High Vectors mode in which the exception vector table starts at address 0xFFFF0000. For example, the system may boot using a vector table in ROM at address 0x00000000. Once the system starts up, the OS may write an updated vector table in RAM at 0xFFFF0000 and put the system into High Vectors mode.

power-up, the processor branches to address 0x00000000. Each exception vector offset typically contains a branch instruction to an exception handler, code that handles the exception and then either exits or returns to the user code.

Banked Registers

Before an exception changes the PC, it must save the return address in the LR so that the exception handler knows where to return. However, it must take care not to disturb the value already in the LR, which the program will need later. Therefore, the processor maintains a bank of different registers to use as LR during each of the execution modes. Similarly, the exception handler must not disturb the status register bits.

Hence, a bank of *saved program status registers* (SPSRs) is used to hold a copy of the CPSR during exceptions.

If an exception takes place while a program is manipulating its stack frame, the frame might be in an unstable state (e.g., data has been written onto the stack but the stack pointer is not yet pointing to the top of stack). Hence, each execution mode also uses its own stack and banked copy of SP pointing to the top of its stack. Memory must be reserved for each execution mode's stack and banked versions of the stack pointers must be initialized at start-up.

The first thing that an exception handler must do is to push all of the registers it might change onto the stack. This takes some time. ARM has a *fast interrupt* execution mode FIQ in which R8–R12 are also banked. Thus, the exception handler can immediately begin without saving these registers.

Exception Handling

Now that we have defined execution modes, exception vectors, and banked registers, we can define what occurs during an exception. Upon detecting an exception, the processor:

1. Stores the CPSR into the banked SPSR
2. Sets the execution mode and privilege level based on the type of exception
3. Sets *interrupt mask* bits in the CPSR so that the exception handler will not be interrupted
4. Stores the return address into the banked LR
5. Branches to the exception vector table based on the type of exception

The processor then executes the instruction in the exception vector table, typically a branch to the exception handler. The handler usually pushes other registers onto its stack, takes care of the exception, and pops the registers back off the stack. The exception handler returns using the `MOVS PC, LR` instruction, a special flavor of `MOV` that performs the following cleanup:

1. Copies the banked SPSR to the CPSR to restore the status register
2. Copies the banked LR (possibly adjusted for certain exceptions) to the PC to return to the program where the exception occurred
3. Restores the execution mode and privilege level

Exception-Related Instructions

Programs operate at a low privilege level, whereas the operating system has a higher privilege level. To transition between levels in a controlled way, the program places arguments in registers and issues a *supervisor call* (`SVC`) instruction, which generates an exception and raises the

privilege level. The OS examines the arguments and performs the requested function, and then returns to the program.

The OS and other code operating at PL1 can access the banked registers for the various execution modes using the `MRS` (move to register from special register) and `MSR` (move to special register from register) instructions. For example, at boot time, the OS will use these instructions to initialize the stacks for exception handlers.

Start-up

On start-up, the processor jumps to the reset vector and begins executing *boot loader* code in supervisor mode. The boot loader typically configures the memory system, initializes the stack pointer, and reads the OS from disk; then it begins a much longer boot process in the OS. The OS eventually will load a program, change to unprivileged user mode, and jump to the start of the program.

6.7 EVOLUTION OF ARM ARCHITECTURE

The ARM1 processor was first developed by Acorn Computer in Britain for the BBC Micro computers in 1985 as an upgrade to the 6502 microprocessor used in many personal computers of the era. It was followed within the year by the ARM2, which went into production in the Acorn Archimedes computer. ARM was an acronym for *Acorn RISC Machine*. The product implemented Version 2 of the ARM instruction set (ARMv2). The address bus was only 26 bits, and the upper 6 bits of the 32-bit PC were used to hold status bits. The architecture included almost all of the instructions described in this chapter, including data-processing, most loads and stores, branches, and multiplies.

As of ARMv7, the CPSR is called the Application Program Status Register (APSR).

ARM soon extended the address bus to a full 32 bits, moving the status bits into a dedicated Current Program Status Register (CPSR). ARMv4, introduced in 1993, added halfword loads and stores and provided both signed and unsigned halfword and byte loads. This is the core of the modern ARM instruction set, and is what we have covered in this chapter.

The ARM instruction set has seen many enhancements described in subsequent sections. The highly successful ARM7TDMI processor in 1995 introduced the 16-bit Thumb instruction set in ARMv4T to improve code density. ARMv5TE added digital signal processing (DSP) and optional floating-point instructions. ARMv6 added multimedia instructions and enhanced the Thumb instruction set. ARMv7 improved the floating-point and multimedia instructions, renaming them Advanced SIMD. ARMv8 introduced a completely new 64-bit architecture. Various other system programming instructions have been introduced as the architecture has evolved.

6.7.1 Thumb Instruction Set

Thumb instructions are 16 bits long to achieve higher code density; they are identical to regular ARM instructions but generally have limitations, including that they:

- Access only the bottom eight registers
- Reuse a register as both a source and destination
- Support shorter immediates
- Lack conditional execution
- Always write the status flags

Almost all ARM instructions have Thumb equivalents. Because the instructions are less powerful, more are required to write an equivalent program. However, the instructions are half as long, giving overall Thumb code size of about 65% of the ARM equivalent. The Thumb instruction set is valuable not only to reduce the size and cost of code storage memory, but also to allow for an inexpensive 16-bit bus to instruction memory and to reduce the power consumed by fetching instructions from the memory.

ARM processors have an instruction set state register, ISETSTATE, that includes a T bit to indicate whether the processor is in normal mode (T = 0) or Thumb mode (T = 1). This mode determines how instructions should be fetched and interpreted. The `BX` and `BLX` branch instructions toggle the T bit to enter or exit Thumb mode.

Thumb instruction encoding is more complex and irregular than ARM instructions to pack as much useful information as possible into 16-bit halfwords. Figure 6.33 shows encodings for common Thumb instructions. The upper bits specify the type of instruction. Data-processing instructions typically specify two registers, one of which is both the first source and the destination. They always write the status flags. Adds, subtracts, and shifts can specify a short immediate. Conditional branches specify a 4-bit condition code and a short offset, whereas unconditional branches allow a longer offset. Note that `BX` takes a 4-bit register identifier so that it can access the link register LR. Special forms of `LDR`, `STR`, `ADD`, and `SUB` are defined to operate relative to the stack pointer SP (to access the stack frame during function calls). Another special form of `LDR` loads relative to the PC (to access a literal pool). Forms of `ADD` and `MOV` can access all 16 registers. `BL` always requires two halfwords to specify a 22-bit destination.

ARM subsequently refined the Thumb instruction set and added a number of 32-bit Thumb-2 instructions to boost performance of common operations and to allow any program to be written in Thumb mode.

The irregular Thumb instruction set encoding and variable-length instructions (1 or 2 halfwords) are characteristic of 16-bit processor architectures that must pack a large amount of information into a short instruction word. The irregularity complicates instruction decoding.

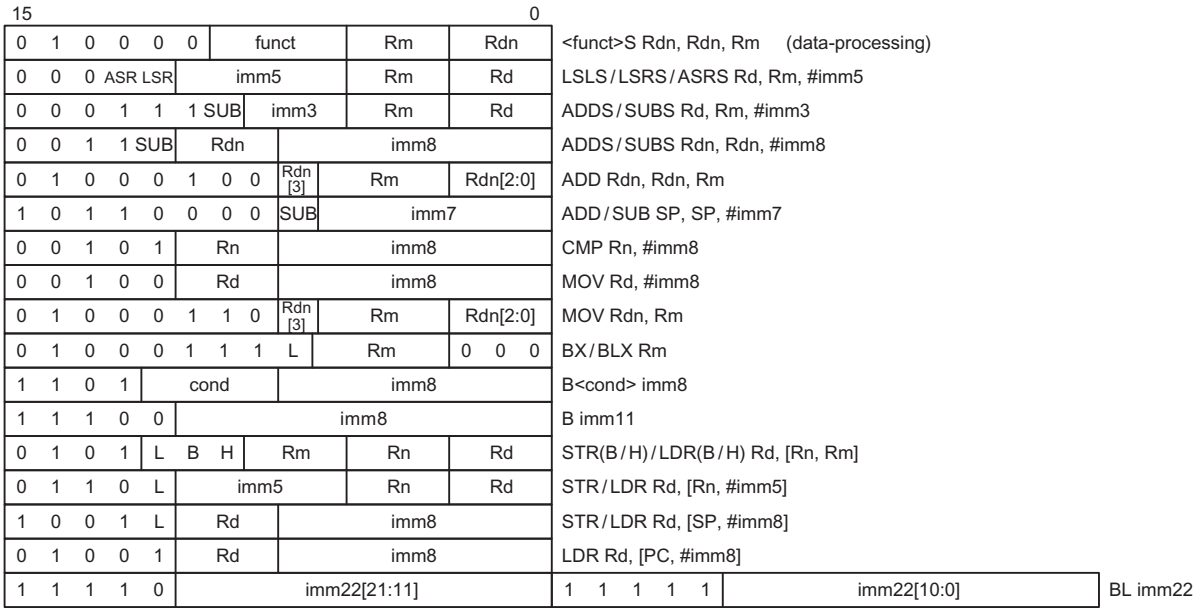


Figure 6.33 Thumb instruction encoding examples

Thumb-2 instructions are identified by their most significant 5 bits being 11101, 11110, or 11111. The processor then fetches a second halfword containing the remainder of the instruction. The Cortex-M series of processors operates exclusively in Thumb state.

6.7.2 DSP Instructions

The Fast Fourier Transform (FFT), the most common DSP algorithm, is both complicated and performance-critical. The DSP instructions in computer architectures are intended to perform efficient FFTs, especially on 16-bit fractional data.

The basic multiply instructions, listed in Appendix B, are part of ARMv4. ARMv5TE added the saturating math instructions and packed and fractional multiplies to support DSP algorithms.

Digital signal processors (DSPs) are designed to efficiently handle signal processing algorithms such as the Fast Fourier Transform (FFT) and Finite/Infinite Impulse Response filters (FIR/IIR). Common applications include audio and video encoding and decoding, motor control, and speech recognition. ARM provides a number of DSP instructions for these purposes. DSP instructions include multiply, add, and multiply-accumulate (MAC)—multiply and add the result to a running sum: sum = sum + src1 × src2. MAC is a distinguishing feature separating DSP instruction sets from regular instruction sets. It is very commonly used in DSP algorithms and doubles the performance relative to separate multiply and add instructions. However, MAC requires specifying an extra register to hold the running sum.

DSP instructions often operate on short (16-bit) data representing samples read from a sensor by an analog-to-digital converter. However, the intermediate results are held to greater precision (e.g., 32 or 64 bits)

Table 6.15 DSP data types

Type	Sign Bit	Integer Bits	Fractional Bits
short	1	15	0
unsigned short	0	16	0
long	1	31	0
unsigned long	0	32	0
long long	1	63	0
unsigned long long	0	64	0
Q15	1	0	15
Q31	1	0	31

or saturated to prevent overflow. In *saturated arithmetic*, results larger than the most positive number are treated as the most positive, and results smaller than the most negative are treated as the most negative. For example, in 32-bit arithmetic, results greater than $2^{31} - 1$ saturate at $2^{31} - 1$, and results less than -2^{31} saturate at -2^{31}. Common DSP data types are given in Table 6.15. Two's complement numbers are indicated as having one sign bit. The 16-, 32-, and 64-bit types are also known as *half*, *single*, and *double* precision, not to be confused with single and double-precision floating-point numbers. For efficiency, two half-precision numbers are packed in a single 32-bit word.

The *integer* types come in signed and unsigned flavors with the sign bit in the msb. *Fractional* types (Q15 and Q31) represent a signed fractional number; for example, Q31 spans the range $[-1, 1-2^{-31}]$ with a step of 2^{-31} between consecutive numbers. These types are not defined in the C standard but are supported by some libraries. Q31 can be converted to Q15 by truncation or rounding. In truncation, the Q15 result is just the upper half. In rounding, 0x00008000 is added to the Q31 value and then the result is truncated. When a computation involves many steps, rounding is useful because it avoids accumulating multiple small truncation errors into a significant error.

ARM added a *Q* flag to the status registers to indicate that overflow or saturation has occurred in DSP instructions. For applications where accuracy is critical, the program can clear the *Q* flag before a computation, do the computation in single-precision, and check the *Q* flag afterward. If it is set, overflow occurred and the computation can be repeated in double precision if necessary.

Saturated arithmetic is an important way to gracefully degrade accuracy in DSP algorithms. Commonly, single-precision arithmetic is sufficient to handle most inputs, but pathological cases can overflow the single-precision range. An overflow causes an abrupt sign change to a radically wrong answer, which may appear to the user as a click in an audio stream or a strangely colored pixel in a video stream. Going to double-precision arithmetic prevents overflow but degrades performance and increases power consumption in the typical case. Saturated arithmetic clips the overflow at the maximum or minimum value, which is usually close to the desired value and causes little inaccuracy.

Addition and subtraction are performed identically no matter which format is used. However, multiplication depends on the type. For example, with 16-bit numbers, the number 0xFFFF is interpreted as 65535 for unsigned short, −1 for short, and -2^{-15} for Q15 numbers. Hence, 0xFFFF × 0xFFFF has a very different value for each representation (4,294,836,225; 1; and 2^{-30}, respectively). This leads to different instructions for signed and unsigned multiplication.

A Q15 number A can be viewed as $a \times 2^{-15}$, where a is its interpretation in the range $[-2^{15}, 2^{15}-1]$ as a signed 16-bit number. Hence, the product of two Q15 numbers is:

$$A \times B = a \times b \times 2^{-30} = 2 \times a \times b \times 2^{-31}$$

This means that to multiply two Q15 numbers and get a Q31 result, do ordinary signed multiplication and then double the product. The product can then be truncated or rounded to put it back into Q15 format if necessary.

The rich assortment of multiply and multiply-accumulate instructions are summarized in Table 6.16. MACs require up to four registers: *RdHi*, *RdLo*, *Rn*, and *Rm*. For double-precision operations, *RdHi* and *RdLo* hold the most and least significant 32 bits, respectively. For example, `UMLAL RdLo, RdHi, Rn, Rm` computes {*RdHi*, *RdLo*} = {*RdHi*, *RdLo*} + *Rn* × *Rm*. Half-precision multiplies come in various flavors denoted in braces to choose the operands from the top or bottom half of the word, and in *dual* forms where both the top and bottom halves are multiplied. MACs involving half-precision inputs and a single-precision accumulator (`SMLA*`, `SMLAW*`, `SMUAD`, `SMUSD`, `SMLAD`, `SMLSD`) will set the *Q* flag if the accumulator overflows. The most significant word (MSW) multiplies also come in forms with an `R` suffix that round rather than truncate.

The DSP instructions also include saturated add (`QADD`) and subtract (`QSUB`) of 32-bit words that saturate the results instead of overflowing. They also include `QDADD` and `QDSUB`, which double the second operand before adding/subtracting it to/from the first with saturation; we will shortly find these valuable in fractional MACs. They set the *Q* flag if saturation occurs.

Finally, the DSP instructions include `LDRD` and `STRD` that load and store an even/odd pair of registers in a 64-bit memory double word. These instructions increase the efficiency of moving double-precision values between memory and registers.

Table 6.17 summarizes how to use the DSP instructions to multiply or MAC various types of data. The examples assume halfword data is in the bottom half of a register and that the top half is zero; use the T flavor of `SMUL` when the data is in the top instead. The result is stored in R2, or in {R3, R2} for double-precision. Fractional operations (Q15/Q31) double the result using saturated adds to prevent overflow when multiplying −1 × −1.

Table 6.16 Multiply and multiply-accumulate instructions

Instruction	Function	Description
Ordinary 32-bit multiplication works for both signed and unsigned		
MUL	32 = 32 × 32	Multiply
MLA	32 = 32 + 32 × 32	Multiply-accumulate
MLS	32 = 32 − 32 × 32	Multiply-subtract
unsigned long long = unsigned long × unsigned long		
UMULL	64 = 32 × 32	Unsigned multiply long
UMLAL	64 = 64 + 32 × 32	Unsigned multiply-accumulate long
UMAAL	64 = 32 + 32 × 32 + 32	Unsigned multiply-accumulate-add long
long long = long × long		
SMULL	64 = 32 × 32	Signed multiply long
SMLAL	64 = 64 + 32 × 32	Signed multiply-accumulate long
Packed arithmetic: short × short		
SMUL{BB/BT/TB/TT}	32 = 16 × 16	Signed multiply {bottom/top}
SMLA{BB/BT/TB/TT}	32 = 32 + 16 × 16	Signed multiply-accumulate {bottom/top}
SMLAL{BB/BT/TB/TT}	64 = 64 + 16 × 16	Signed multiply-accumulate long {bottom/top}
Fractional multiplication (Q31 / Q15)		
SMULW{B/T}	32 = (32 × 16) >> 16	Signed multiply word-halfword {bottom/top}
SMLAW{B/T}	32 = 32 + (32 × 16) >> 16	Signed multiply-add word-halfword {bottom/top}
SMMUL{R}	32 = (32 × 32) >> 32	Signed MSW multiply {round}
SMMLA{R}	32 = 32 + (32 × 32) >> 32	Signed MSW multiply-accumulate {round}
SMMLS{R}	32 = 32 − (32 × 32) >> 32	Signed MSW multiply-subtract {round}
long or long long = short × short + short × short		
SMUAD	32 = 16 × 16 + 16 × 16	Signed dual multiply-add
SMUSD	32 = 16 × 16 − 16 × 16	Signed dual multiply-subtract
SMLAD	32 = 32 + 16 × 16 + 16 × 16	Signed multiply-accumulate dual
SMLSD	32 = 32 + 16 × 16 − 16 × 16	Signed multiply-subtract dual
SMLALD	64 = 64 + 16 × 16 + 16 × 16	Signed multiply-accumulate long dual
SMLSLD	64 = 64 + 16 × 16 − 16 × 16	Signed multiply-subtract long dual

Table 6.17 Multiply and MAC code for various data types

First Operand (R0)	Second Operand (R1)	Product (R3/R2)	Multiply	MAC

short	short	short	`SMULBB R2, R0, R1`
			`LDR R3, =0x0000FFFF`
			`AND R2, R3, R2`
				`SMLABB R2, R0, R1`
				`LDR R3, =0x0000FFFF`
				`AND R2, R3, R2`

short	short	long	`SMULBB R2, R0, R1`	`SMLABB R2, R0, R1, R2`

short	short	long long	`MOV R2, #0`
			`MOV R3, #0`
			`SMLALBB R2, R3, R0, R1`
				`SMLALBB R2, R3, R0, R1`

long	short	long	`SMULWB R2, R0, R1`	`SMLAWB R2, R0, R1, R2`
long	long	long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
long	long	long long	`SMULL R2, R3, R0, R1`	`SMLAL R2, R3, R0, R1`

unsigned short	unsigned short	unsigned short	`MUL R2, R0, R1`
			`LDR R3, =0x0000FFFF`
			`AND R2, R3, R2`
				`MLA R2, R0, R1, R2`
				`LDR R3, =0x0000FFFF`
				`AND R2, R3, R2`

unsigned short	unsigned short	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned short	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned long	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned long	unsigned long long	`UMULL R2, R3, R0, R1`	`UMLAL R2, R3, R0, R1`

Q15	Q15	Q15	`SMULBB R2, R0, R1`
			`QADD R2, R2, R2`
			`LSR R2, R2, #16`
				`SMLABB R2, R0, R1, R2`
				`SSAT R2, 16, R2`

Q15	Q15	Q31	`SMULBB R2, R0, R1` `QADD R2, R2, R2`	`SMULBB R3, R0, R1` `QDADD R2, R2, R3`
Q31	Q15	Q31	`SMULWB R2, R0, R1` `QADD R2, R2, R2`	`SMULWB R3, R0, R1` `QDADD R2, R2, R3`
Q31	Q31	Q31	`SMMUL R2, R0, R1` `QADD R2, R2, R2`	`SMMUL R3, R0, R1` `QDADD R2, R2, R3`

6.7.3 Floating-Point Instructions

Floating-point is more flexible than the fixed-point numbers favored in DSP and makes programming easier. Floating-point is widely used in graphics, scientific applications, and control algorithms. Floating-point arithmetic can be performed with a series of ordinary data-processing instructions but is faster and consumes less power using dedicated floating-point instructions and hardware.

The ARMv5 instruction set includes optional floating-point instructions. These instructions access at least 16 64-bit double-precision registers separate from the ordinary registers. These registers can also be treated as pairs of 32-bit single-precision registers. The registers are named D0–D15 as double-precision or S0–S31 as single-precision. For example, `VADD.F32 S2, S0, S1` and `VADD.F64 D2, D0, D1` perform single and double-precision floating-point adds, respectively. Floating-point instructions, listed in Table 6.18, are suffixed with `.F32` or `.F64` to indicate single- or double-precision floating-point.

Table 6.18 ARM floating-point instructions

Instruction	Function
`VABS Rd, Rm`	Rd = |Rm|
`VADD Rd, Rn, Rm`	Rd = Rn + Rm
`VCMP Rd, Rm`	Compare and set floating-point status flags
`VCVT Rd, Rm`	Convert between int and float
`VDIV Rd, Rn, Rm`	Rd = Rn / Rm
`VMLA Rd, Rn, Rm`	Rd = Rd + Rn * Rm
`VMLS Rd, Rn, Rm`	Rd = Rd − Rn * Rm
`VMOV Rd, Rm or #const`	Rd = Rm or constant
`VMUL Rd, Rn, Rm`	Rd = Rn * Rm
`VNEG Rd, Rm`	Rd = −Rm
`VNMLA Rd, Rn, Rm`	Rd = −(Rd + Rn * Rm)
`VNMLS Rd, Rn, Rm`	Rd = −(Rd − Rn * Rm)
`VNMUL Rd, Rn, Rm`	Rd = −Rn * Rm
`VSQRT Rd, Rm`	Rd = sqrt(Rm)
`VSUB Rd, Rn, Rm`	Rd = Rn − Rm

The MRC and MCR instructions are used to transfer data between the ordinary registers and the floating-point coprocessor registers.

ARM defines the Floating-Point Status and Control Register (FPSCR). Like the ordinary status register, it holds *N*, *Z*, *C*, and *V* flags for floating-point operations. It also specifies rounding modes, exceptions, and special conditions such as overflow, underflow, and divide-by-zero. The VMRS and VMSR instructions transfer information between a regular register and the FPSCR.

6.7.4 Power-Saving and Security Instructions

Battery-powered devices save power by spending most of their time in sleep mode. ARMv6K introduced instructions to support such power savings. The wait for interrupt (WFI) instruction allows the processor to enter a low-power state until an interrupt occurs. The system may generate interrupts based on user events (such as touching a screen) or on a periodic timer. The wait for event (WFE) instruction is similar but is helpful in multiprocessor systems (see Section 7.7.8) so that a processor can go to sleep until notified by another processor. It wakes up either during an interrupt or when another processor sends an event using the SEV instruction.

ARMv7 enhances the exception handling to support virtualization and security. In *virtualization*, multiple operating systems can run concurrently on the same processor, unaware of each other's existence. A hypervisor switches between the operating systems. The hypervisor operates at privilege level PL2. It is invoked with a hypervisor trap exception. With *security* extensions, the processor defines a secure state with limited means of entry and restricted access to secure portions of memory. Even if an attacker compromises the operating system, the secure kernel may resist tampering. For example, the secure kernel may be used to disable a stolen phone or to enforce digital rights management such that a user can't duplicate copyrighted content.

6.7.5 SIMD Instructions

The term SIMD (pronounced "sim-dee") stands for *single instruction multiple data*, in which a single instruction acts on multiple pieces of data in parallel. A common application of SIMD is to perform many short arithmetic operations at once, especially for graphics processing. This is also called *packed* arithmetic.

Short data elements often appear in graphics processing. For example, a pixel in a digital photo may use 8 bits to store each of the red, green, and blue color components. Using an entire 32-bit word to process one of these components wastes the upper 24 bits. Moreover,

when the components from 16 adjacent pixels are packed into a 128-bit quadword, the processing can be performed 16 times faster. Similarly, coordinates in a 3-dimensional graphics space are generally represented with 32-bit (single-precision) floating-point numbers. Four of these coordinates can be packed into a 128-bit quadword.

Most modern architectures offer SIMD arithmetic operations with wide SIMD registers packing multiple narrower operands. For example, the ARMv7 Advanced SIMD instructions share the registers from the floating-point unit. Moreover, these registers can also be paired to act as eight 128-bit quad words Q0–Q7. The registers pack together several 8-, 16-, 32-, or 64-bit integer or floating-point values. The instructions are suffixed with `.I8`, `.I16`, `.I32`, `.I64`, `.F32`, or `.F64` to indicate how the registers should be treated.

Figure 6.34 shows the `VADD.I8 D2, D1, D0` vector add instruction operating on eight pairs of 8-bit integers packed into 64-bit double words. Similarly `VADD.I32 Q2, Q1, Q0` adds four pairs of 32-bit integers packed into 128-bit quad words and `VADD.F32, D2, D1, D0` adds two pairs of 32-bit single-precision floating-point numbers packed into 64-bit double words. Performing packed arithmetic requires modifying the ALU to eliminate carries between the smaller data elements. For example, a carry out of $a_0 + b_0$ must not affect the result of $a_1 + b_1$.

Advanced SIMD instructions begin with `V`. They include the following categories:

- Basic arithmetic functions also defined for floating-point
- Loads and stores of multiple elements, including deinterleaving and interleaving
- Bitwise logical operations
- Comparisons
- Many flavors of shifts, additions, and subtractions with and without saturation
- Many flavors of multiply and MAC
- Miscellaneous instructions

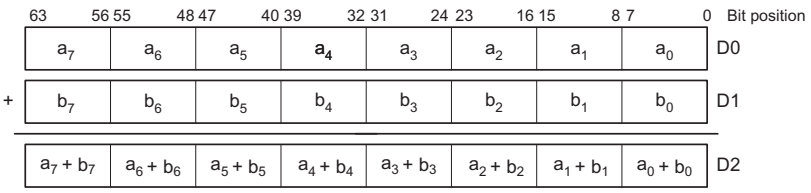


Figure 6.34 Packed arithmetic: eight simultaneous 8-bit additions

ARMv6 also defined a more limited set of SIMD instructions operating on the regular 32-bit registers. These include 8- and 16-bit addition and subtraction, and instructions to efficiently pack and unpack bytes and halfwords into a word. These instructions are useful to manipulate 16-bit data in DSP code.

6.7.6 64-bit Architecture

32-bit architectures allow a program to directly access at most 2^{32} bytes = 4 GB of memory. Large computer servers led the transition to 64-bit architectures that can access vast amounts of memory. Personal computers and then mobile devices followed. 64-bit architectures can sometimes be faster as well because they move more information with a single instruction.

Many architectures simply extend their general-purpose registers from 32 to 64 bits, but ARMv8 introduced a new instruction set as well to streamline idiosyncrasies. The classic instruction set lacks enough general-purpose registers for complex programs, forcing costly movement of data between registers and memory. Keeping the PC in R15 and SP in R13 also complicates the processor implementation, and programs often need a register containing the value 0.

The ARMv8 instructions are still 32 bits long and the instruction set looks very much like ARMv7, but with some problems cleaned up. In ARMv8, the register file is expanded to 31 64-bit registers (called X0–X30) and the PC and SP are no longer part of the general-purpose registers. X30 serves as the link register. Note that there is no X31 register; instead, it is called the zero register (ZR) and is hardwired to 0. Data-processing instructions can operate on 32- or 64-bit values, whereas loads and stores always use 64-bit addresses. To make room for the extra bits to specify source and destination registers, the condition field is removed from most instructions. However, branches can still be conditional. ARMv8 also streamlines exception handling, doubles the number of advanced SIMD registers, and adds instructions for AES and SHA cryptography. The instruction encodings are rather complex and do not classify into a handful of categories.

On reset, ARMv8 processors boot in 64-bit mode. The processor can drop into 32-bit mode by setting a bit in a system register and invoking an exception. It returns to 64-bit mode when the exception returns.

6.8 ANOTHER PERSPECTIVE: x86 ARCHITECTURE

Almost all personal computers today use x86 architecture microprocessors. x86, also called IA-32, is a 32-bit architecture originally developed by Intel. AMD also sells x86 compatible microprocessors.

The x86 architecture has a long and convoluted history dating back to 1978, when Intel announced the 16-bit 8086 microprocessor. IBM selected the 8086 and its cousin, the 8088, for IBM's first personal computers. In 1985, Intel introduced the 32-bit 80386 microprocessor, which was backward compatible with the 8086, so it could run software developed for earlier PCs. Processor architectures compatible with the 80386 are called x86 processors. The Pentium, Core, and Athlon processors are well known x86 processors.

Various groups at Intel and AMD over many years have shoehorned more instructions and capabilities into the antiquated architecture. The result is far less elegant than ARM. However, software compatibility is far more important than technical elegance, so x86 has been the *de facto* PC standard for more than two decades. More than 100 million x86 processors are sold every year. This huge market justifies more than $5 billion of research and development annually to continue improving the processors.

x86 is an example of a Complex Instruction Set Computer (CISC) architecture. In contrast to RISC architectures such as ARM, each CISC instruction can do more work. Programs for CISC architectures usually require fewer instructions. The instruction encodings were selected to be more compact, so as to save memory, when RAM was far more expensive than it is today; instructions are of variable length and are often less than 32 bits. The trade-off is that complicated instructions are more difficult to decode and tend to execute more slowly.

This section introduces the x86 architecture. The goal is not to make you into an x86 assembly language programmer, but rather to illustrate some of the similarities and differences between x86 and ARM. We think it is interesting to see how x86 works. However, none of the material in this section is needed to understand the rest of the book. Major differences between x86 and ARM are summarized in Table 6.19.

Table 6.19 Major differences between ARM and x86

Feature	ARM	x86
# of registers	15 general purpose	8, some restrictions on purpose
# of operands	3–4 (2–3 sources, 1 destination)	2 (1 source, 1 source/destination)
operand location	registers or immediates	registers, immediates, or memory
operand size	32 bits	8, 16, or 32 bits
condition flags	yes	yes
instruction types	simple	simple and complicated
instruction encoding	fixed, 4 bytes	variable, 1–15 bytes

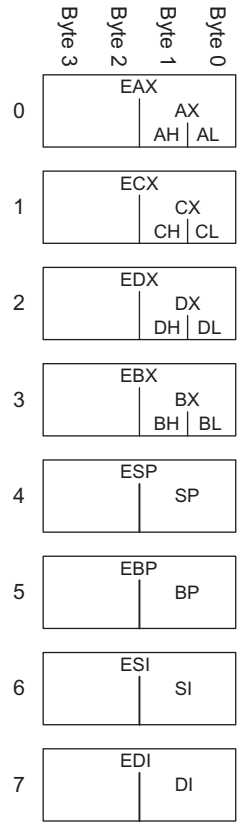


Figure 6.35 x86 registers

6.8.1 x86 Registers

The 8086 microprocessor provided eight 16-bit registers. It could separately access the upper and lower eight bits of some of these registers. When the 32-bit 80386 was introduced, the registers were extended to 32 bits. These registers are called EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI. For backward compatibility, the bottom 16 bits and some of the bottom 8-bit portions are also usable, as shown in Figure 6.35.

The eight registers are almost, but not quite, general purpose. Certain instructions cannot use certain registers. Other instructions always put their results in certain registers. Like SP in ARM, ESP is normally reserved for the stack pointer.

The x86 program counter is called the EIP (the *extended instruction pointer*). Like the ARM PC, it advances from one instruction to the next or can be changed with branch and function call instructions.

6.8.2 x86 Operands

ARM instructions always act on registers or immediates. Explicit load and store instructions are needed to move data between memory and the registers. In contrast, x86 instructions may operate on registers, immediates, or memory. This partially compensates for the small set of registers.

ARM instructions generally specify three operands: two sources and one destination. x86 instructions specify only two operands. The first is a source. The second is both a source and the destination. Hence, x86 instructions always overwrite one of their sources with the result. Table 6.20 lists the combinations of operand locations in x86. All combinations are possible except memory to memory.

Table 6.20 Operand locations

Source/ Destination	Source	Example	Meaning
register	register	`add EAX, EBX`	`EAX <– EAX + EBX`
register	immediate	`add EAX, 42`	`EAX <– EAX + 42`
register	memory	`add EAX, [20]`	`EAX <– EAX + Mem[20]`
memory	register	`add [20], EAX`	`Mem[20] <– Mem[20] + EAX`
memory	immediate	`add [20], 42`	`Mem[20] <– Mem[20] + 42`

Table 6.21 Memory addressing modes

Example	Meaning	Comment
`add EAX, [20]`	`EAX <– EAX + Mem[20]`	displacement
`add EAX, [ESP]`	`EAX <– EAX + Mem[ESP]`	base addressing
`add EAX, [EDX+40]`	`EAX <– EAX + Mem[EDX+40]`	base + displacement
`add EAX, [60+EDI*4]`	`EAX <– EAX + Mem[60+EDI*4]`	displacement + scaled index
`add EAX, [EDX+80+EDI*2]`	`EAX <– EAX + Mem[EDX+80+EDI*2]`	base + displacement + scaled index

Table 6.22 Instructions acting on 8-, 16-, or 32-bit data

Example	Meaning	Data Size
`add AH, BL`	`AH <– AH + BL`	8-bit
`add AX, –1`	`AX <– AX + 0xFFFF`	16-bit
`add EAX, EDX`	`EAX <– EAX + EDX`	32-bit

Like ARM, x86 has a 32-bit memory space that is byte-addressable. However, x86 supports a wider variety of memory indexing modes. Memory locations are specified with any combination of a *base register*, *displacement*, and a *scaled index register*. Table 6.21 illustrates these combinations. The displacement can be an 8-, 16-, or 32-bit value. The scale multiplying the index register can be 1, 2, 4, or 8. The base + displacement mode is equivalent to the ARM base addressing mode for loads and stores. Like ARM, x86 also provides a scaled index. In x86, the scaled index provides an easy way to access arrays or structures of 2-, 4-, or 8-byte elements without having to issue a sequence of instructions to generate the address.

While ARM always acts on 32-bit words, x86 instructions can operate on 8-, 16-, or 32-bit data. Table 6.22 illustrates these variations.

6.8.3 Status Flags

x86, like many CISC architectures, uses condition flags (also called *status flags*) to make decisions about branches and to keep track of carries and arithmetic overflow. x86 uses a 32-bit register, called EFLAGS, that stores the status flags. Some of the bits of the EFLAGS register are given in Table 6.23. Other bits are used by the operating system.

ARM's use of condition flags sets it apart from other RISC architectures.

Table 6.23 Selected EFLAGS

Name	Meaning
CF (Carry Flag)	Carry out generated by last arithmetic operation. Indicates overflow in unsigned arithmetic. Also used for propagating the carry between words in multiple-precision arithmetic
ZF (Zero Flag)	Result of last operation was zero
SF (Sign Flag)	Result of last operation was negative (msb = 1)
OF (Overflow Flag)	Overflow of two's complement arithmetic

The architectural state of an x86 processor includes EFLAGS as well as the eight registers and the EIP.

6.8.4 x86 Instructions

x86 has a larger set of instructions than ARM. Table 6.24 describes some of the general purpose instructions. x86 also has instructions for floating-point arithmetic and for arithmetic on multiple short data elements packed into a longer word. D indicates the destination (a register or memory location), and S indicates the source (a register, memory location, or immediate).

Note that some instructions always act on specific registers. For example, 32×32-bit multiplication always takes one of the sources from EAX and always puts the 64-bit result in EDX and EAX. LOOP always stores the loop counter in ECX. PUSH, POP, CALL, and RET use the stack pointer, ESP.

Conditional jumps check the flags and branch if the appropriate condition is met. They come in many flavors. For example, JZ jumps if the zero flag (*ZF*) is 1. JNZ jumps if the zero flag is 0. Like ARM, the jumps usually follow an instruction, such as the compare instruction (CMP), that sets the flags. Table 6.25 lists some of the conditional jumps and how they depend on the flags set by a prior compare operation.

6.8.5 x86 Instruction Encoding

The x86 instruction encodings are truly messy, a legacy of decades of piecemeal changes. Unlike ARMv4, whose instructions are uniformly 32 bits, x86 instructions vary from 1 to 15 bytes, as shown in Figure 6.36.[1]

[1] It is possible to construct 17-byte instructions if all the optional fields are used. However, x86 places a 15-byte limit on the length of legal instructions.

Table 6.24 Selected x86 instructions

Instruction	Meaning	Function
ADD/SUB	add/subtract	D = D + S / D = D − S
ADDC	add with carry	D = D + S + CF
INC/DEC	increment/decrement	D = D + 1 / D = D − 1
CMP	compare	Set flags based on D − S
NEG	negate	D = − D
AND/OR/XOR	logical AND/OR/XOR	D = D op S
NOT	logical NOT	$D = \overline{D}$
IMUL/MUL	signed/unsigned multiply	EDX:EAX = EAX × D
IDIV/DIV	signed/unsigned divide	EDX:EAX/D EAX = Quotient; EDX = Remainder
SAR/SHR	arithmetic/logical shift right	D = D >>> S / D = D >> S
SAL/SHL	left shift	D = D << S
ROR/ROL	rotate right/left	Rotate D by S
RCR/RCL	rotate right/left with carry	Rotate CF and D by S
BT	bit test	CF = D[S] (the S*th* bit of D)
BTR/BTS	bit test and reset/set	CF = D[S]; D[S] = 0 / 1
TEST	set flags based on masked bits	Set flags based on D AND S
MOV	move	D = S
PUSH	push onto stack	ESP = ESP −4; Mem[ESP] = S
POP	pop off stack	D = MEM[ESP]; ESP = ESP + 4
CLC, STC	clear/set carry flag	CF = 0 / 1
JMP	unconditional jump	relative jump: EIP = EIP + S absolute jump: EIP = S
Jcc	conditional jump	if (flag) EIP = EIP + S
LOOP	loop	ECX = ECX −1 if (ECX ≠ 0) EIP = EIP + imm
CALL	function call	ESP = ESP −4; MEM[ESP] = EIP; EIP = S
RET	function return	EIP = MEM[ESP]; ESP = ESP + 4

Table 6.25 Selected branch conditions

Instruction	Meaning	Function after `CMP D, S`
`JZ/JE`	jump if `ZF = 1`	jump if `D = S`
`JNZ/JNE`	jump if `ZF = 0`	jump if `D ≠ S`
`JGE`	jump if `SF = OF`	jump if `D ≥ S`
`JG`	jump if `SF = OF` and `ZF = 0`	jump if `D >S`
`JLE`	jump if `SF ≠ OF` or `ZF = 1`	jump if `D ≤ S`
`JL`	jump if `SF ≠ OF`	jump if `D <S`
`JC/JB`	jump if `CF = 1`	
`JNC`	jump if `CF = 0`	
`JO`	jump if `OF = 1`	
`JNO`	jump if `OF = 0`	
`JS`	jump if `SF = 1`	
`JNS`	jump if `SF = 0`	

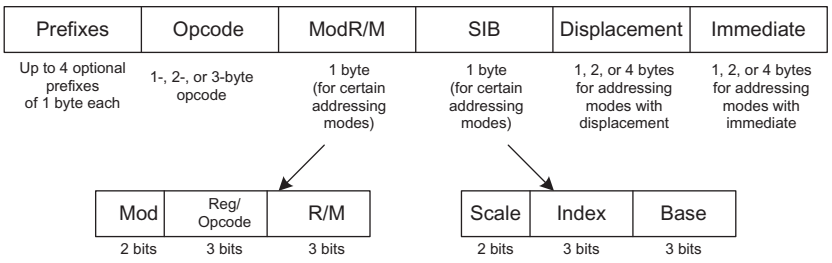


Figure 6.36 x86 instruction encodings

The *opcode* may be 1, 2, or 3 bytes. It is followed by four optional fields: *ModR/M*, *SIB*, *Displacement*, and *Immediate*. *ModR/M* specifies an addressing mode. *SIB* specifies the scale, index, and base registers in certain addressing modes. *Displacement* indicates a 1-, 2-, or 4-byte displacement in certain addressing modes. And *Immediate* is a 1-, 2-, or 4-byte constant for instructions using an immediate as the source operand. Moreover, an instruction can be preceded by up to four optional byte-long prefixes that modify its behavior.

The *ModR/M* byte uses the 2-bit *Mod* and 3-bit *R/M* field to specify the addressing mode for one of the operands. The operand can come from

one of the eight registers, or from one of 24 memory addressing modes. Due to artifacts in the encodings, the ESP and EBP registers are not available for use as the base or index register in certain addressing modes. The *Reg* field specifies the register used as the other operand. For certain instructions that do not require a second operand, the *Reg* field is used to specify three more bits of the *opcode*.

In addressing modes using a scaled index register, the *SIB* byte specifies the index register and the scale (1, 2, 4, or 8). If both a base and index are used, the *SIB* byte also specifies the base register.

ARM fully specifies the instruction in the *cond*, *op*, and *funct* fields of the instruction. x86 uses a variable number of bits to specify different instructions. It uses fewer bits to specify more common instructions, decreasing the average length of the instructions. Some instructions even have multiple opcodes. For example, `add AL, imm8` performs an 8-bit add of an immediate to AL. It is represented with the 1-byte opcode, 0x04, followed by a 1-byte immediate. The A register (AL, AX, or EAX) is called the *accumulator*. On the other hand, `add D, imm8` performs an 8-bit add of an immediate to an arbitrary destination, *D* (memory or a register). It is represented with the 1-byte *opcode* 0x80 followed by one or more bytes specifying *D*, followed by a 1-byte immediate. Many instructions have shortened encodings when the destination is the accumulator.

In the original 8086, the *opcode* specified whether the instruction acted on 8- or 16-bit operands. When the 80386 introduced 32-bit operands, no new opcodes were available to specify the 32-bit form. Instead, the same opcode was used for both 16- and 32-bit forms. An additional bit in the *code segment descriptor* used by the OS specifies which form the processor should choose. The bit is set to 0 for backward compatibility with 8086 programs, defaulting the *opcode* to 16-bit operands. It is set to 1 for programs to default to 32-bit operands. Moreover, the programmer can specify prefixes to change the form for a particular instruction. If the *prefix* 0x66 appears before the *opcode*, the alternative size operand is used (16 bits in 32-bit mode, or 32 bits in 16-bit mode).

6.8.6 Other x86 Peculiarities

The 80286 introduced *segmentation* to divide memory into segments of up to 64 KB in length. When the OS enables segmentation, addresses are computed relative to the beginning of the segment. The processor checks for addresses that go beyond the end of the segment and indicates an error, thus preventing programs from accessing memory outside their own segment. Segmentation proved to be a hassle for programmers and is not used in modern versions of the Windows operating system.

Intel and Hewlett-Packard jointly developed a new 64-bit architecture called IA-64 in the mid 1990's. It was designed from a clean slate, bypassing the convoluted history of x86, taking advantage of 20 years of new research in computer architecture, and providing a 64-bit address space. However, IA-64 has yet to become a market success. Most computers needing the large address space now use the 64-bit extensions of x86.

x86 contains string instructions that act on entire strings of bytes or words. The operations include moving, comparing, or scanning for a specific value. In modern processors, these instructions are usually slower than performing the equivalent operation with a series of simpler instructions, so they are best avoided.

As mentioned earlier, the 0x66 *prefix* is used to choose between 16- and 32-bit operand sizes. Other prefixes include ones used to lock the bus (to control access to shared variables in a multiprocessor system), to predict whether a branch will be taken or not, and to repeat the instruction during a string move.

The bane of any architecture is to run out of memory capacity. With 32-bit addresses, x86 can access 4 GB of memory. This was far more than the largest computers had in 1985, but by the early 2000's it had become limiting. In 2003, AMD extended the address space and register sizes to 64 bits, calling the enhanced architecture AMD64. AMD64 has a compatibility mode that allows it to run 32-bit programs unmodified while the OS takes advantage of the bigger address space. In 2004, Intel gave in and adopted the 64-bit extensions, renaming them Extended Memory 64 Technology (EM64T). With 64-bit addresses, computers can access 16 exabytes (16 billion GB) of memory.

For those curious about more details of the x86 architecture, the x86 Intel Architecture Software Developer's Manual is freely available on Intel's Web site.

6.8.7 The Big Picture

ARM strikes a balance between simple instructions and dense code by including features such as condition flags and shifted register operands. Thease features make ARM code more compact than other RISC architectures.

This section has given a taste of some of the differences between the ARM RISC architecture and the x86 CISC architecture. x86 tends to have shorter programs, because a complex instruction is equivalent to a series of simple ARM instructions and because the instructions are encoded to minimize memory use. However, the x86 architecture is a hodgepodge of features accumulated over the years, some of which are no longer useful but must be kept for compatibility with old programs. It has too few registers, and the instructions are difficult to decode. Merely explaining the instruction set is difficult. Despite all these failings, x86 is firmly entrenched as the dominant computer architecture for PCs, because the value of software compatibility is so great and because the huge market justifies the effort required to build fast x86 microprocessors.

6.9 SUMMARY

To command a computer, you must speak its language. A computer architecture defines how to command a processor. Many different computer architectures are in widespread commercial use today, but once

you understand one, learning others is much easier. The key questions to ask when approaching a new architecture are:

- What is the data word length?
- What are the registers?
- How is memory organized?
- What are the instructions?

ARM is a 32-bit architecture because it operates on 32-bit data. The ARM architecture has 16 registers which include 15 general-purpose registers and the PC. In principle, any of the general-purpose registers can be used in any code. However, by convention, certain registers are reserved for certain purposes for ease of programming and so that functions written by different programmers can communicate easily. For example, R14 (the link register LR) holds the return address after a `BL` instruction, and R0–R3 hold the arguments of a function. ARM has a byte-addressable memory system with 32-bit addresses. Instructions are 32 bits long and are word-aligned for efficient access. This chapter discussed the most commonly used ARM instructions.

The power of defining a computer architecture is that a program written for any given architecture can run on many different implementations of that architecture. For example, programs written for the Intel Pentium processor in 1993 will generally still run (and run much faster) on the Intel Xeon or AMD Phenom processors in 2015.

In the first part of this book, we learned about the circuit and logic levels of abstraction. In this chapter, we jumped up to the architecture level. In the next chapter, we study microarchitecture, the arrangement of digital building blocks that implement a processor architecture. Microarchitecture is the link between hardware and software engineering. And, we believe it is one of the most exciting topics in all of engineering: You will learn to build your own microprocessor!

Exercises

Exercise 6.1 Give three examples from the ARM architecture of each of the architecture design principles: (1) regularity supports simplicity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises. Explain how each of your examples exhibits the design principle.

Exercise 6.2 The ARM architecture has a register set that consists of 16 32-bit registers. Is it possible to design a computer architecture without a register set? If so, briefly describe the architecture, including the instruction set. What are advantages and disadvantages of this architecture over the ARM architecture?

Exercise 6.3 Consider memory storage of a 32-bit word stored at memory word 42 in a byte-addressable memory.

(a) What is the byte address of memory word 42?

(b) What are the byte addresses that memory word 42 spans?

(c) Draw the number 0xFF223344 stored at word 42 in both big-endian and little-endian machines. Clearly label the byte address corresponding to each data byte value.

Exercise 6.4 Repeat Exercise 6.3 for memory storage of a 32-bit word stored at memory word 15 in a byte-addressable memory.

Exercise 6.5 Explain how the following ARM program can be used to determine whether a computer is big-endian or little-endian:

```
MOV  R0, #100
LDR  R1, =0xABCD876     ; R1 = 0xABCD876
STR  R1, [R0]
LDRB R2, [R0, #1]
```

Exercise 6.6 Write the following strings using ASCII encoding. Write your final answers in hexadecimal.

(a) SOS

(b) Cool

(c) boo!

Exercise 6.7 Repeat Exercise 6.6 for the following strings.

(a) howdy

(b) lions

(c) To the rescue!

Exercise 6.8 Show how the strings in Exercise 6.6 are stored in a byte-addressable memory on a little-endian machine starting at memory address 0x00001050C. Clearly indicate the memory address of each byte.

Exercise 6.9 Repeat Exercise 6.8 for the strings in Exercise 6.7.

Exercise 6.10 Convert the following ARM assembly code into machine language. Write the instructions in hexadecimal.

```
MOV R10, #63488
LSL R9, R6, #7
STR R4, [R11, R8]
ASR R6, R7, R3
```

Exercise 6.11 Repeat Exercise 6.10 for the following ARM assembly code:

```
ADD R8, R0, R1
LDR R11, [R3, #4]
SUB R5, R7, #0x58
LSL R3, R2, #14
```

Exercise 6.12 Consider data-processing instructions with an immediate *Src2*.

(a) Which instructions from Exercise 6.10 are in this format?

(b) Write out the 12-bit immediate field (*imm12*) of the instructions from part (a), then write them as 32-bit immediates.

Exercise 6.13 Repeat Exercise 6.12 for the instructions in Exercise 6.11.

Exercise 6.14 Convert the following program from machine language into ARM assembly language. The numbers on the left are the instruction addresses in memory, and the numbers on the right give the instruction at that address. Then reverse engineer a high-level program that would compile into this assembly language routine and write it. Explain in words what the program does. R0 and R1 are the input, and they initially contain positive numbers, `a` and `b`. At the end of the program, R0 is the output.

```
0x00008008  0xE3A02000
0x0000800C  0xE1A03001
0x00008010  0xE1510000
0x00008014  0x8A000002
0x00008018  0xE2822001
0x0000801C  0xE0811003
0x00008020  0xEAFFFFFA
0x00008024  0xE1A00002
```

Exercise 6.15 Repeat Exercise 6.14 for the following machine code. R0 and R1 are the inputs. R0 contains a 32-bit number and R1 is the address of a 32-element array of characters (char).

```
0x00008104  0xE3A0201F
0x00008108  0xE1A03230
0x0000810C  0xE2033001
0x00008110  0xE4C13001
0x00008114  0xE2522001
0x00008118  0x5AFFFFFA
0x0000811C  0xE1A0F00E
```

Exercise 6.16 The `NOR` instruction is not part of the ARM instruction set, because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: R0 = R1 NOR R2. Use as few instructions as possible.

Exercise 6.17 The `NAND` instruction is not part of the ARM instruction set, because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: R0 = R1 NAND R2. Use as few instructions as possible.

Exercise 6.18 Consider the following high-level code snippets. Assume the (signed) integer variables `g` and `h` are in registers R0 and R1, respectively.

(i)
```
if (g >= h)
  g = g + h;
else
  g = g - h;
```

(ii)
```
if (g < h)
  h = h + 1;
else
  h = h * 2;
```

(a) Write the code snippets in ARM assembly language assuming conditional execution is available for branch instructions only. Use as few instructions as possible (within these parameters).

(b) Write the code snippets in ARM assembly language with conditional execution available for all instructions. Use as few instructions as possible.

(c) Compare the difference in code density (i.e., number of instructions) between (a) and (b) for each code snippet and discuss any advantages or disadvantages.

Exercise 6.19 Repeat Exercise 6.18 for the following code snippets.

(i)
```
if (g > h)
  g = g + 1;
else
  h = h - 1;
```

(ii)
```
if (g <= h)
  g = 0;
else
  h = 0;
```

Exercise 6.20 Consider the following high-level code snippet. Assume that the base addresses of `array1` and `array2` are held in R1 and R2 and that `array2` is initialized before it is used.

```
int i;
int array1[100];
int array2[100];
...
for (i=0; i<100; i=i+1)
    array1[i] = array2[i];
```

(a) Write the code snippet in ARM assembly without using pre- or post-indexing or a scaled register. Use as few instructions as possible (given the constraints).

(b) Write the code snippet in ARM assembly with pre- or post-indexing and a scaled register available. Use as few instructions as possible.

(c) Compare the difference in code density (i.e., number of instructions) between (a) and (b). Discuss any advantages or disadvantages.

Exercise 6.21 Repeat Exercise 6.20 for the following high-level code snippet. Assume that `temp` is initialized before it is used and that R3 holds the base address of `temp`.

```
int i;
int temp[100];
...
for (i=0; i<100; i=i+1)
  temp[i] = temp[i] * 128;
```

Exercise 6.22 Consider the following two code snippets. Assume R1 holds `i` and that R0 holds the base address of the `vals` array.

(i)
```
int i;
int vals[200];

for (i=0; i < 200; i=i+1)
  vals[i] = i;
```

(ii)
```
int i;
int vals[200];
for (i=199; i >= 0; i = i-1)
  vals[i] = i;
```

(a) Are the code snippets functionally equivalent?

(b) Write each code snippet using ARM assembly language. Use as few instructions as possible.

(c) Discuss any advantages or disadvantages of one construct over the other.

Exercise 6.23 Repeat Exercise 6.22 for the following high-level code snippets. Assume R1 holds `i`, R0 holds the base address of the `nums` array, and that the array is initialized before use.

(i)
```
int i;
int nums[10];
...
for (i=0; i < 10; i=i+1)
  nums[i] = nums[i]/2;
```

(ii)
```
int i;
int nums[10];
...
for (i=9; i >= 0; i = i-1)
  nums[i] = nums[i]/2;
```

This simple string copy function has a serious flaw: it has no way of knowing that `dst` has enough space to receive `src`. If a malicious programmer were able to execute `strcpy` with a long string `src`, the programmer might be able to write bytes all over memory, possibly even modifying code stored in subsequent memory locations. With some cleverness, the modified code might take over the machine. This is called a buffer overflow attack; it is employed by several nasty programs, including the infamous Blaster worm, which caused an estimated $525 million in damages in 2003.

Exercise 6.24 Write a function in a high-level language for `int find42(int array[], int size)`. `size` specifies the number of elements in `array`, and `array` specifies the base address of the `array`. The function should return the index number of the first array entry that holds the value 42. If no array entry is 42, it should return the value –1.

Exercise 6.25 The high-level function `strcpy` copies the character string `src` to the character string `dst`.

```
// C code
void strcpy(char dst[], char src[]) {
  int i = 0;
  do {
    dst[i] = src[i];
  } while (src[i++]);
}
```

(a) Implement the `strcpy` function in ARM assembly code. Use R4 for `i`.

(b) Draw a picture of the stack before, during, and after the `strcpy` function call.

Assume SP = 0xBEFFF000 just before `strcpy` is called.

Exercise 6.26 Convert the high-level function from Exercise 6.24 into ARM assembly code.

Exercise 6.27 Consider the ARM assembly code below. `func1`, `func2`, and `func3` are non-leaf functions. `func4` is a leaf function. The code is not shown for each function, but the comments indicate which registers are used within each function.

```
0x00091000 func1 ...   ; func1 uses R4-R10
0x00091020 BL func2
. . .
0x00091100 func2 ...   ; func2 uses R0-R5
0x0009117C BL func3
. . .
0x00091400 func3 ...   ; func3 uses R3, R7-R9
0x00091704 BL func4
. . .
0x00093008 func4 ...   ; func4 uses R11-R12
0x00093118 MOV PC, LR
```

(a) How many words are the stack frames of each function?

(b) Sketch the stack after `func4` is called. Clearly indicate which registers are stored where on the stack and mark each of the stack frames. Give values where possible.

Exercise 6.28 Each number in the Fibonacci series is the sum of the previous two numbers. Table 6.26 lists the first few numbers in the series, $fib(n)$.

(a) What is $fib(n)$ for $n = 0$ and $n = -1$?

(b) Write a function called `fib` in a high-level language that returns the Fibonacci number for any nonnegative value of n. Hint: You probably will want to use a loop. Clearly comment your code.

(c) Convert the high-level function of part (b) into ARM assembly code. Add comments after every line of code that explain clearly what it does. Use the Keil MDK-ARM simulator to test your code on $fib(9)$. (See the Preface for how to install the Keil MDK-ARM simulator.)

Table 6.26 Fibonacci series

n	1	2	3	4	5	6	7	8	9	10	11	…
$fib(n)$	1	1	2	3	5	8	13	21	34	55	89	…

Exercise 6.29 Consider Code Example 6.27. For this exercise, assume *factorial*(*n*) is called with input argument $n = 5$.

(a) What value is in R0 when `factorial` returns to the calling function?

(b) Suppose you replace the instructions at addresses 0x8500 and 0x8520 with `PUSH {R0, R1}` and `POP {R1, R2}`, respectively. Will the program:
(1) enter an infinite loop but not crash;
(2) crash (cause the stack to grow or shrink beyond the dynamic data segment or the PC to jump to a location outside the program);
(3) produce an incorrect value in R0 when the program returns to loop (if so, what value?); or
(4) run correctly despite the deleted lines?

(c) Repeat part (b) with the following instruction modifications:
(i) replace the instructions at addresses 0x8500 and 0x8520 with `PUSH {R3, LR}` and `POP {R3, LR}`, respectively.
(ii) replace the instructions at addresses 0x8500 and 0x8520 with `PUSH {LR}` and `POP {LR}`, respectively.
(iii) delete the instruction at address 0x8510.

Exercise 6.30 Ben Bitdiddle is trying to compute the function $f(a, b) = 2a + 3b$ for nonnegative b. He goes overboard in the use of function calls and recursion and produces the following high-level code for functions `f` and `g`.

```
// high-level code for functions f and g
int f(int a, int b) {
  int j;

  j = a;

  return j + a + g(b);
}
int g(int x) {
  int k;
  k = 3;

  if (x == 0) return 0;
  else return k + g(x - 1);
}
```

Ben then translates the two functions into assembly language as follows. He also writes a function, `test`, that calls the function `f(5, 3)`.

```
; ARM assembly code
; f: R0 = a, R1 = b, R4 = j;
; g: R0 = x, R4 = k

0x00008000 test   MOV  R0, #5           ; a = 5
0x00008004        MOV  R1, #3           ; b = 3
0x00008008        BL   f                ; call f(5, 3)
0x0000800C loop   B    loop             ; and loop forever
0x00008010 f      PUSH {R1,R0,LR,R4}    ; save registers on stack
0x00008014        MOV  R4, R0           ; j = a
0x00008018        MOV  R0, R1           ; place b as argument for g
0x0000801C        BL   g                ; call g(b)
0x00008020        MOV  R2, R0           ; place return value in R2
0x00008024        POP  {R1,R0}          ; restore a and b after call
0x00008028        ADD  R0, R2, R0       ; R0 = g(b) + a
0x0000802C        ADD  R0, R0, R4       ; R0 = (g(b) + a) + j
0x00008030        POP  {R4,LR}          ; restore R4, LR
0x00008034        MOV  PC, LR           ; return
0x00008038 g      PUSH {R4,LR}          ; save registers on stack
0x0000803C        MOV  R4, #3           ; k = 3
0x00008040        CMP  R0, #0           ; x == 0?
0x00008044        BNE  else             ; branch when not equal
0x00008048        MOV  R0, #0           ; if equal, return value = 0
0x0000804C        B    done             ; and clean up
0x00008050 else   SUB  R0, R0, #1       ; x = x - 1
0x00008054        BL   g                ; call g(x - 1)
0x00008058        ADD  R0, R0, R4       ; R0 = g(x - 1) + k
0x0000805C done   POP  {R4,LR}          ; restore R0,R4,LR from stack
0x00008060        MOV  PC, LR           ; return
```

You will probably find it useful to make drawings of the stack similar to the one in Figure 6.14 to help you answer the following questions.

(a) If the code runs starting at `test`, what value is in R0 when the program gets to `loop`? Does his program correctly compute $2a + 3b$?

(b) Suppose Ben changes the instructions at addresses 0x00008010 and 0x00008030 to `PUSH {R1,R0,R4}` and `POP {R4}`, respectively. Will the program
   (1) enter an infinite loop but not crash;
   (2) crash (cause the stack to grow beyond the dynamic data segment or the PC to jump to a location outside the program);
   (3) produce an incorrect value in R0 when the program returns to `loop` (if so, what value?), or
   (4) run correctly despite the deleted lines?

(c) Repeat part (b) when the following instructions are changed. Note that labels aren't changed, only instructions.
   (i) instructions at 0x00008010 and 0x00008024 change to `PUSH {R1,LR, R4}` and `POP {R1}`, respectively.
   (ii) instructions at 0x00008010 and 0x00008024 change to `PUSH {R0,LR, R4}` and `POP {R0}`, respectively.
   (iii) instructions at 0x00008010 and 0x00008030 change to `PUSH {R1,R0, LR}` and `POP {LR}`, respectively.
   (iv) instructions at 0x00008010, 0x00008024, and 0x00008030 are deleted.
   (v) instructions at 0x00008038 and 0x0000805C change to `PUSH {R4}` and `POP {R4}`, respectively.
   (vi) instructions at 0x00008038 and 0x0000805C change to `PUSH {LR}` and `POP {LR}`, respectively.
   (vii) instructions at 0x00008038 and 0x0000805C are deleted.

Exercise 6.31 Convert the following branch instructions into machine code. Instruction addresses are given to the left of each instruction.

(a)
```
0x0000A000       BEQ LOOP
0x0000A004       ...
0x0000A008       ...
0x0000A00C LOOP  ...
```

(b)
```
0x00801000       BGE DONE
...
0x00802040 DONE  ...
```

(c)
```
0x0000B10C BACK  ...
...              ...
0x0000D000       BHI BACK
```

(d)
```
0x00103000       BL  FUNC
...              ...
0x0011147C FUNC  ...
```

(e)
```
0x00008004 L1    ...
...              ...
0x0000F00C       B L1
```

Exercise 6.32 Consider the following ARM assembly language snippet. The numbers to the left of each instruction indicate the instruction address.

```
0x000A0028 FUNC1       MOV R4, R1
0x000A002C             ADD R5, R3, R5, LSR #2
0x000A0030             SUB R4, R0, R3, ROR R4
0x000A0034             BL  FUNC2
...                    ...
0x000A0038 FUNC2       LDR R2, [R0, #4]
0x000A003C             STR R2, [R1, -R2]
```

```
0x000A0040              CMP R3, #0
0x000A0044              BNE ELSE
0x000A0048              MOV PC, LR
0x000A004C ELSE         SUB R3, R3, #1
0x000A0050              B FUNC2
```

(a) Translate the instruction sequence into machine code. Write the machine code instructions in hexadecimal.

(b) List the addressing mode used at each line of code.

Exercise 6.33 Consider the following C code snippet.

```
// C code
void setArray(int num) {
  int i;
  int array[10];

  for (i = 0; i < 10; i = i + 1)
    array[i] = compare(num, i);
}
int compare(int a, int b) {
  if (sub(a, b) >= 0)
    return 1;
  else
    return 0;
}
int sub(int a, int b) {
  return a - b;
}
```

(a) Implement the C code snippet in ARM assembly language. Use R4 to hold the variable `i`. Be sure to handle the stack pointer appropriately. The array is stored on the stack of the `setArray` function (see the end of Section 6.3.7).

(b) Assume `setArray` is the first function called. Draw the status of the stack before calling `setArray` and during each function call. Indicate the names of registers and variables stored on the stack, mark the location of SP, and clearly mark each stack frame.

(c) How would your code function if you failed to store LR on the stack?

Exercise 6.34 Consider the following high-level function.

```
// C code
int f(int n, int k) {
  int b;

  b = k + 2;
  if (n == 0) b = 10;
  else b = b + (n * n) + f(n - 1, k + 1);
  return b * k;
}
```

(a) Translate the high-level function `f` into ARM assembly language. Pay particular attention to properly saving and restoring registers across function calls and using the ARM preserved register conventions. Clearly comment your code. You can use the ARM `MUL` instruction. The function starts at instruction address 0x00008100. Keep local variable `b` in R4.

(b) Step through your function from part (a) by hand for the case of `f(2, 4)`. Draw a picture of the stack similar to the one in Figure 6.14, and assume that SP is equal to 0xBFF00100 when `f` is called. Write the register name and data value stored at each location in the stack and keep track of the stack pointer value (SP). Clearly mark each stack frame. You might also find it useful to keep track of the values in R0, R1, and R4 throughout execution. Assume that when `f` is called, R4 = 0xABCD and LR = 0x00008010. What is the final value of R0?

Exercise 6.35 Give an example of the worst case for a forward branch (i.e., a branch to a higher instruction address). The worst case is when the branch cannot branch far. Show instructions and instruction addresses.

Exercise 6.36 The following questions examine the limitations of the branch instruction, `B`. Give your answer in number of instructions relative to the branch instruction.

(a) In the worst case, how far can `B` branch forward (i.e., to higher addresses)? (The worst case is when the branch instruction cannot branch far.) Explain using words and examples, as needed.

(b) In the best case, how far can `B` branch forward? (The best case is when the branch instruction can branch the farthest.) Explain.

(c) In the worst case, how far can `B` branch backward (to lower addresses)? Explain.

(d) In the best case, how far can `B` branch backward? Explain.

Exercise 6.37 Explain why it is advantageous to have a large immediate field, *imm24*, in the machine format for the branch instructions, `B` and `BL`.

Exercise 6.38 Write assembly code that branches to an instruction 32 Minstructions from the first instruction. Recall that 1 Minstruction = 2^{20} instructions = 1,048,576 instructions. Assume that your code begins at address 0x00008000. Use a minimum number of instructions.

Exercise 6.39 Write a function in high-level code that takes a 10-entry array of 32-bit integers stored in little-endian format and converts it to big-endian format. After writing the high-level code, convert it to ARM assembly code. Comment all your code and use a minimum number of instructions.

Exercise 6.40 Consider two strings: `string1` and `string2`.

(a) Write high-level code for a function called `concat` that concatenates (joins together) the two strings: `void concat(char string1[], char string2[], char stringconcat[])`. The function does not return a value. It concatenates `string1` and `string2` and places the resulting string in `stringconcat`. You may assume that the character array `stringconcat` is large enough to accommodate the concatenated string.

(b) Convert the function from part (a) into ARM assembly language.

Exercise 6.41 Write an ARM assembly program that adds two positive single-precision floating point numbers held in R0 and R1. Do not use any of the ARM floating-point instructions. You need not worry about any of the encodings that are reserved for special purposes (e.g., 0, NANs, etc.) or numbers that overflow or underflow. Use the Keil MDK-ARM simulator to test your code. (See the Preface for how to install the Keil MDK-ARM simulator.) You will need to manually set the values of R0 and R1 to test your code. Demonstrate that your code functions reliably.

Exercise 6.42 Consider the following ARM program. Assume the instructions are placed starting at memory address 0x8400 and that `L1` is at memory address 0x10024.

```
; ARM assembly code
MAIN
   PUSH {LR}
   LDR  R2, =L1  ; this is translated into a PC-relative load
   LDR  R0, [R2]
   LDR  R1, [R2, #4]
   BL   DIFF
   POP  {LR}
   MOV  PC, LR
DIFF
   SUB  R0, R0, R1
   MOV  PC, LR
   ...
L1
```

(a) First show the instruction address next to each assembly instruction.

(b) Describe the symbol table: i.e., list the address of each of the labels.

(c) Convert all instructions into machine code.

(d) How big (how many bytes) are the data and text segments?

(e) Sketch a memory map showing where data and instructions are stored, similar to Figure 6.31.

Exercise 6.43 Repeat Exercise 6.42 for the following ARM code. Assume the instructions are placed starting at memory address 0x8534 and that `L2` is at memory address 0x1305C.

```
; ARM assembly code
MAIN
  PUSH {R4,LR}
  MOV  R4, #15
  LDR  R3, =L2  ; this is translated into a PC-relative load
  STR  R4, [R3]
  MOV  R1, #27
  STR  R1, [R3, #4]
  LDR  R0, [R3]
  BL   GREATER
  POP  {R4,LR}
  MOV  PC, LR
GREATER
  CMP  R0, R1
  MOV  R0, #0
  MOVGT R0, #1
  MOV  PC, LR
  ...
L2
```

Exercise 6.44 Name two ARM instructions that can increase code density (i.e., decrease the number of instructions in a program). Give examples of each, showing equivalent ARM assembly code with and without using the instructions.

Exercise 6.45 Explain the advantages and disadvantages of conditional execution.

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs (but are usually open to any assembly language).

Question 6.1 Write ARM assembly code for swapping the contents of two registers, R0 and R1. You may not use any other registers.

Question 6.2 Suppose you are given an array of both positive and negative integers. Write ARM assembly code that finds the subset of the array with the largest sum. Assume that the array's base address and the number of array elements are in R0 and R1, respectively. Your code should place the resulting subset of the array starting at the base address in R2. Write code that runs as fast as possible.

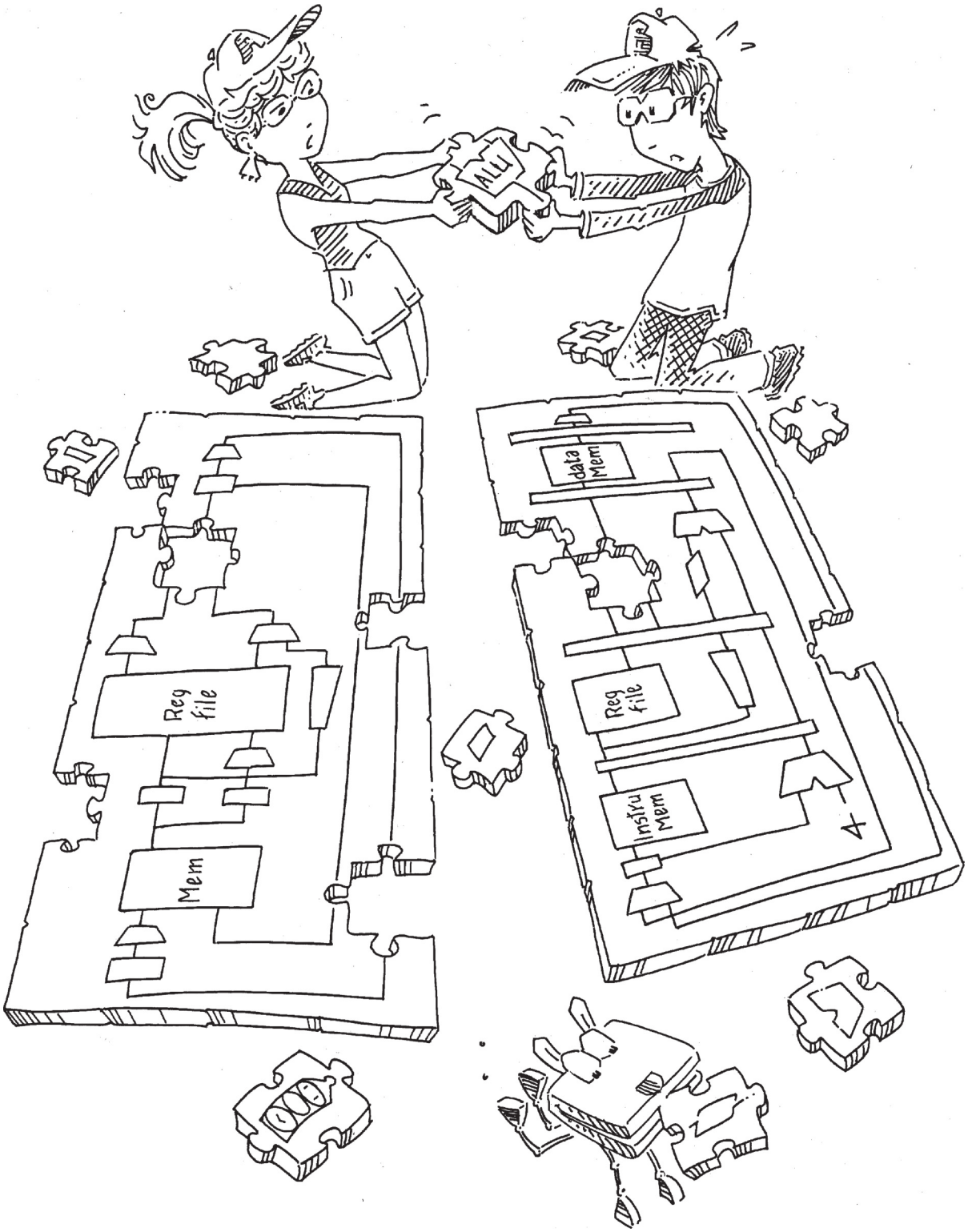
Question 6.3 You are given an array that holds a C string. The string forms a sentence. Design an algorithm for reversing the words in the sentence and storing the new sentence back in the array. Implement your algorithm using ARM assembly code.

Question 6.4 Design an algorithm for counting the number of 1's in a 32-bit number. Implement your algorithm using ARM assembly code.

Question 6.5 Write ARM assembly code to reverse the bits in a register. Use as few instructions as possible. Assume the register of interest is R3.

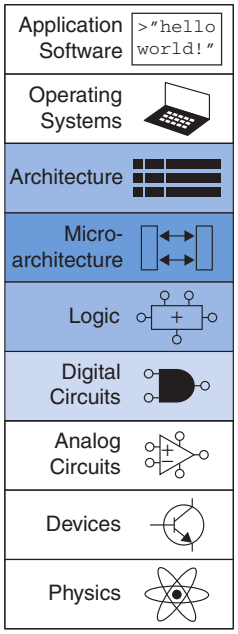
Question 6.6 Write ARM assembly code to test whether overflow occurs when R2 and R3 are added. Use a minimum number of instructions.

Question 6.7 Design an algorithm for testing whether a given string is a palindrome. (Recall that a palindrome is a word that is the same forward and backward. For example, the words "wow" and "racecar" are palindromes.) Implement your algorithm using ARM assembly code


ALU
Reg file
Mem
data Mem
Reg file
Instru Mem
4

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
© 2016 Elsevier Inc. All rights reserved.

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.

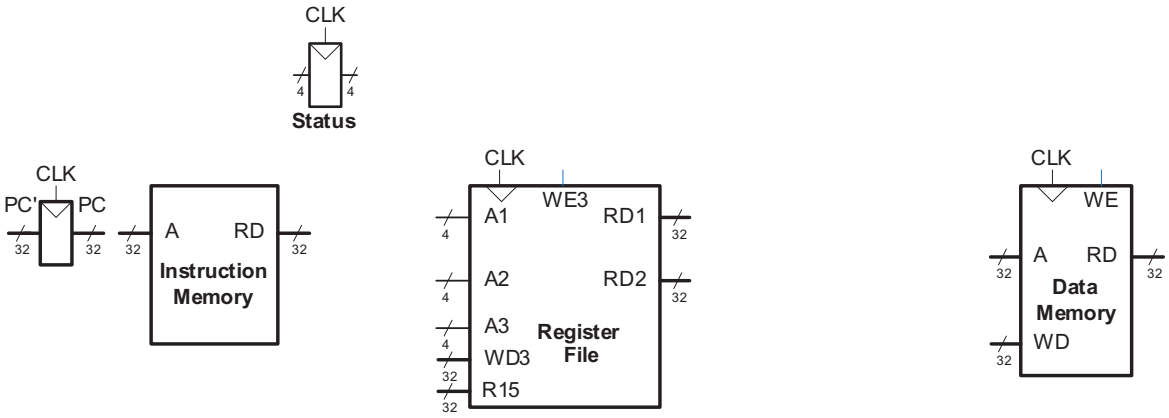


Figure 7.1 State elements of ARM processor

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*.

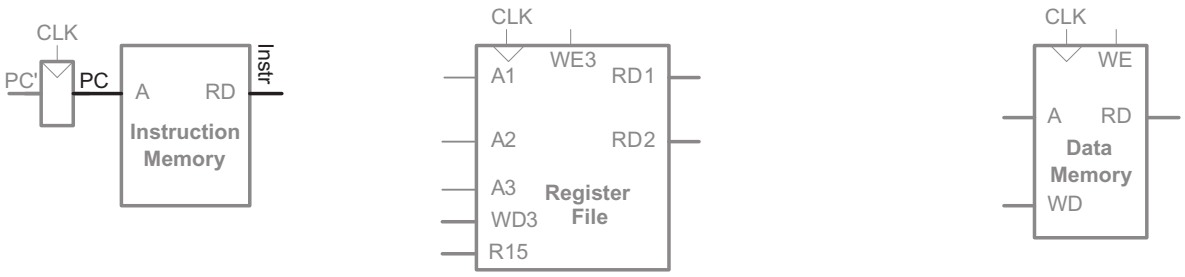


Figure 7.2 Fetch instruction from memory

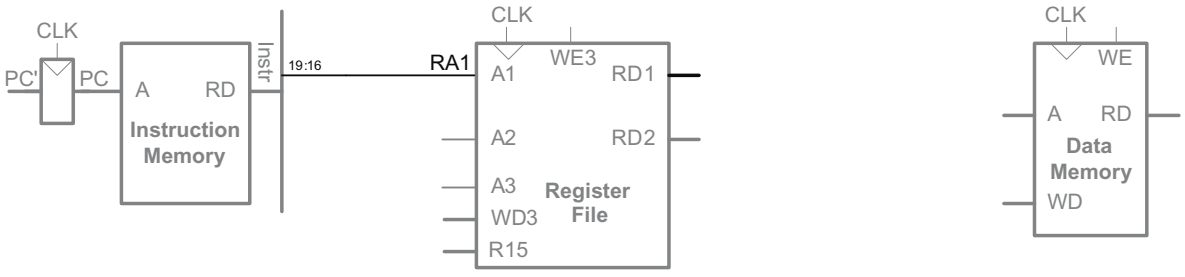


Figure 7.3 Read source operand from register file

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.

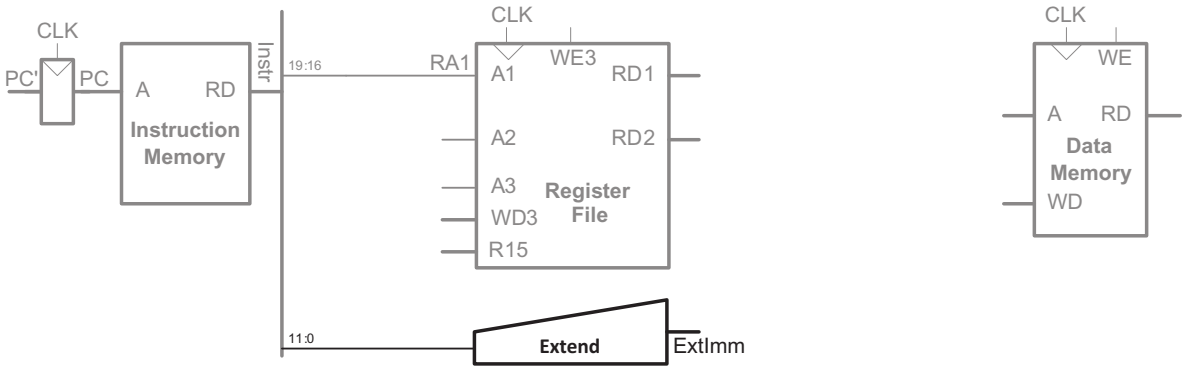


Figure 7.4 Zero-extend the immediate

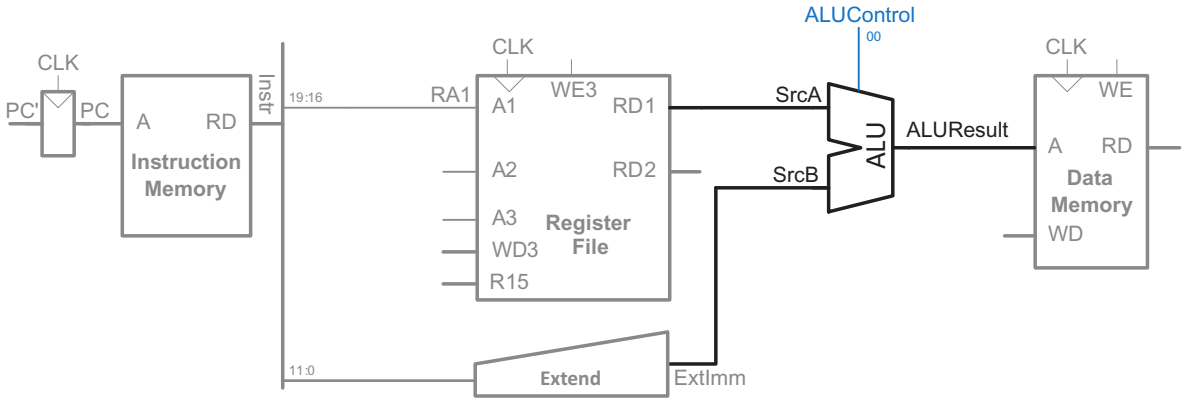


Figure 7.5 Compute memory address

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

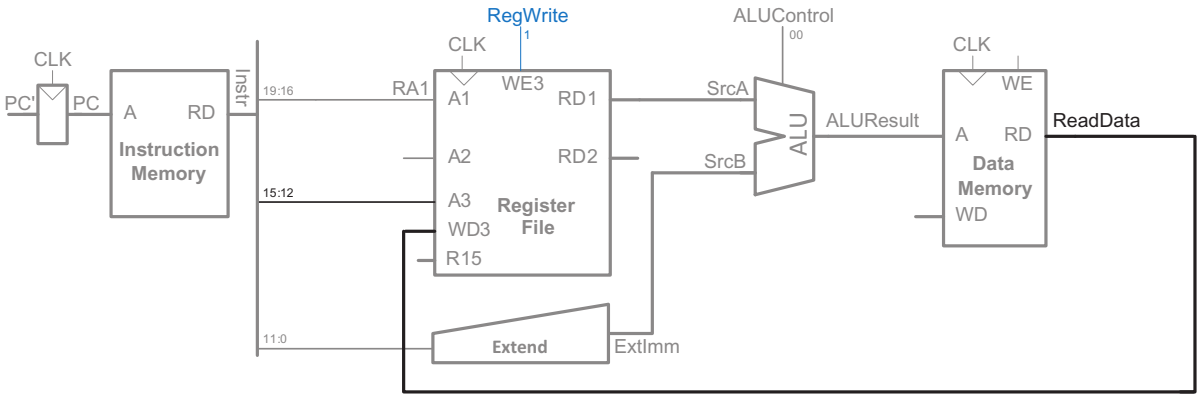


Figure 7.6 Write data back to register file

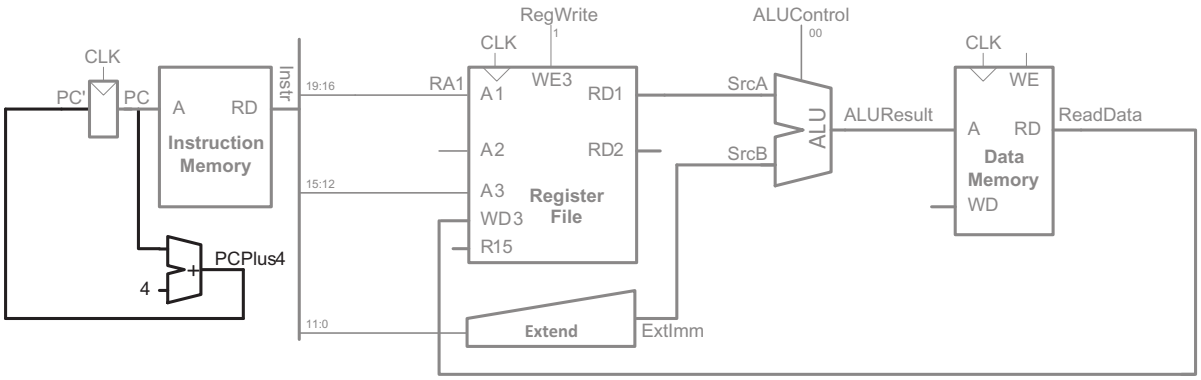


Figure 7.7 Increment program counter

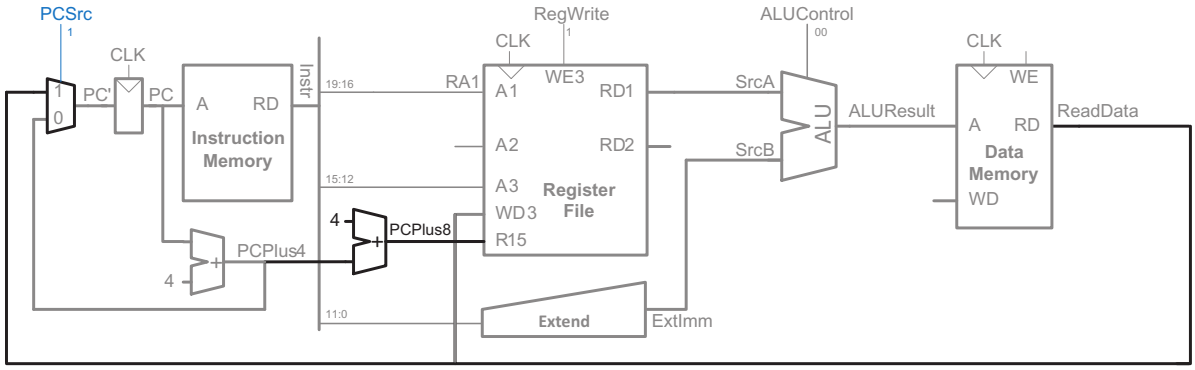


Figure 7.8 Read or write program counter as R15

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 $RegWrite = 0$.

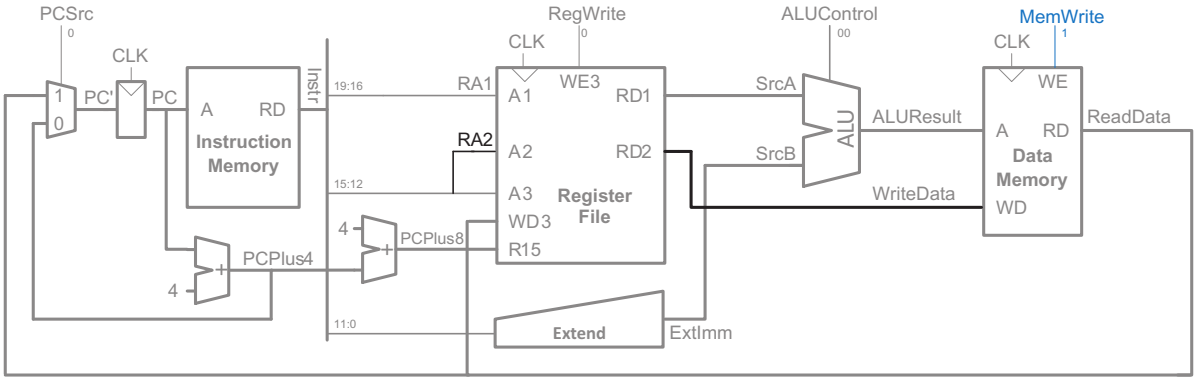


Figure 7.9 Write data to memory for STR instruction

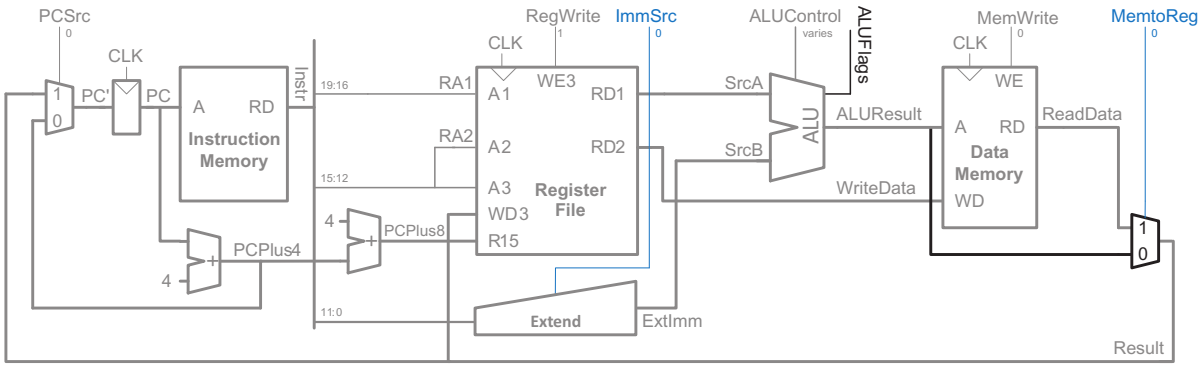


Figure 7.10 Datapath enhancements for data-processing instructions with immediate addressing

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

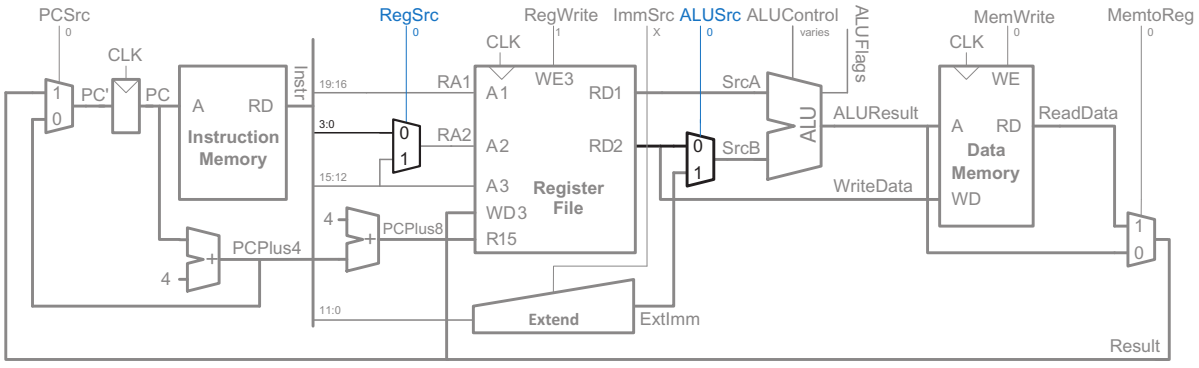


Figure 7.11 Datapath enhancements for data-processing instructions with register addressing

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.

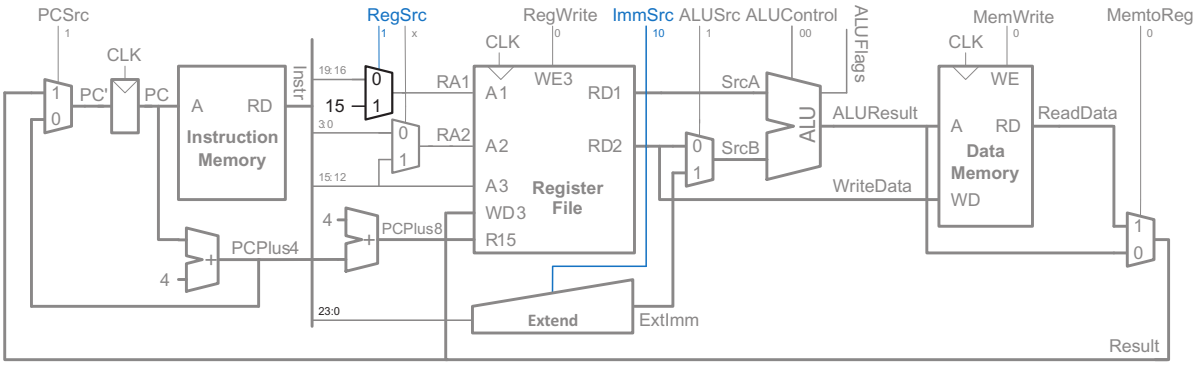


Figure 7.12 Datapath enhancements for `B` instruction

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`

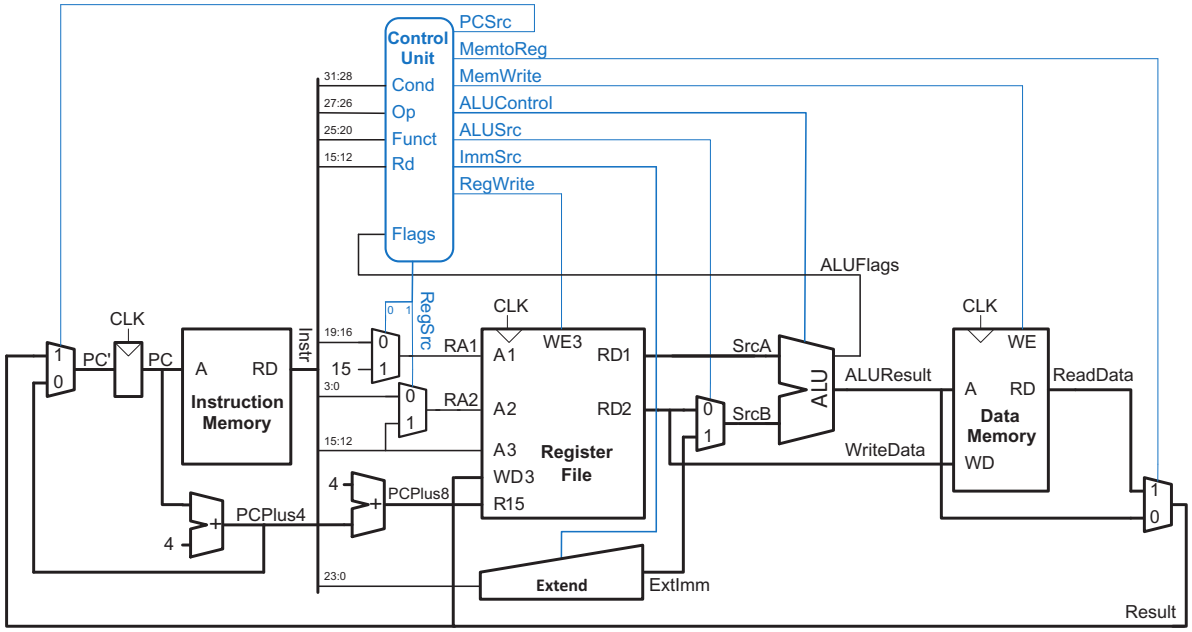


Figure 7.13 Complete single-cycle processor

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.

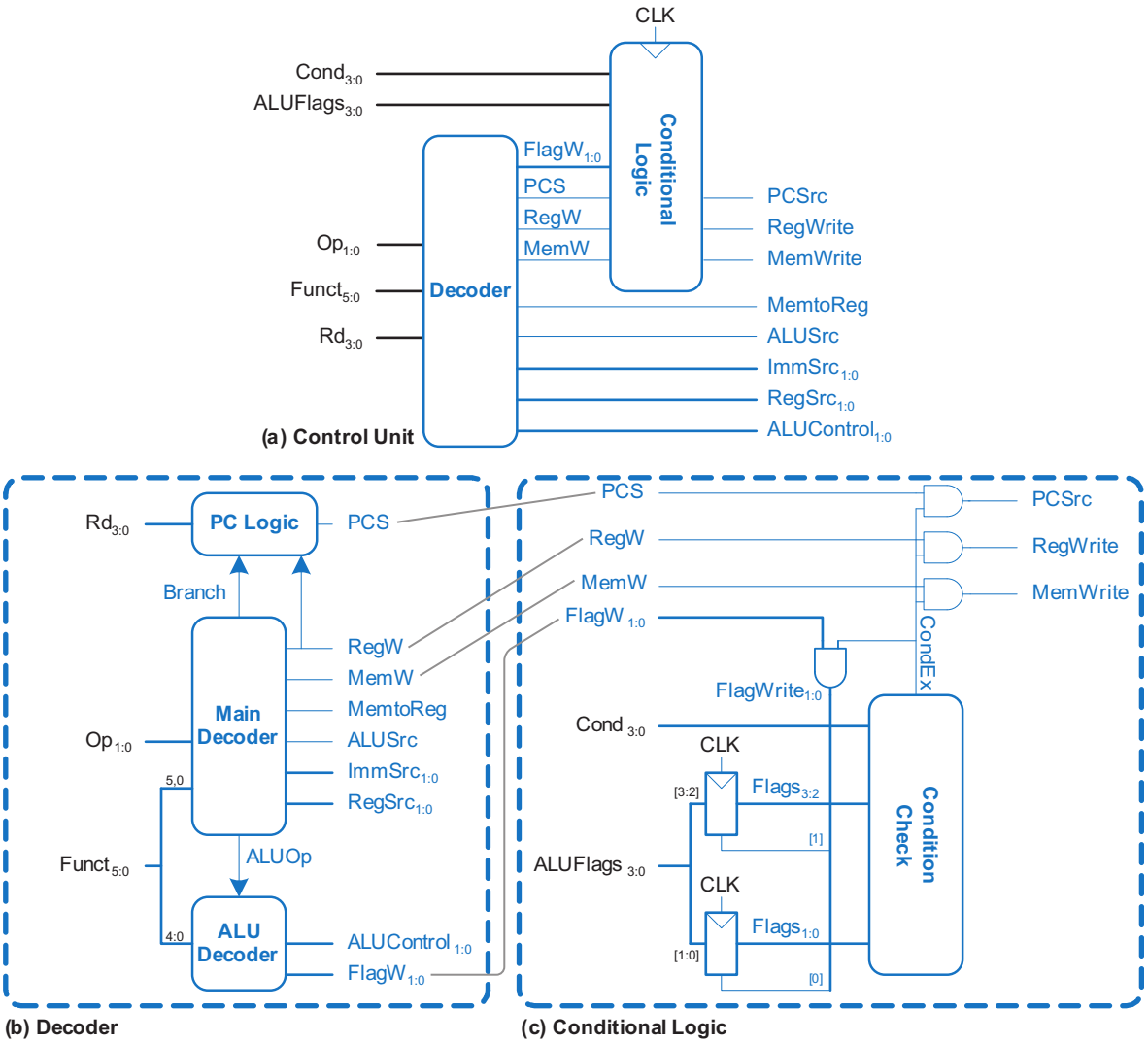


Figure 7.14 Single-cycle control unit

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.

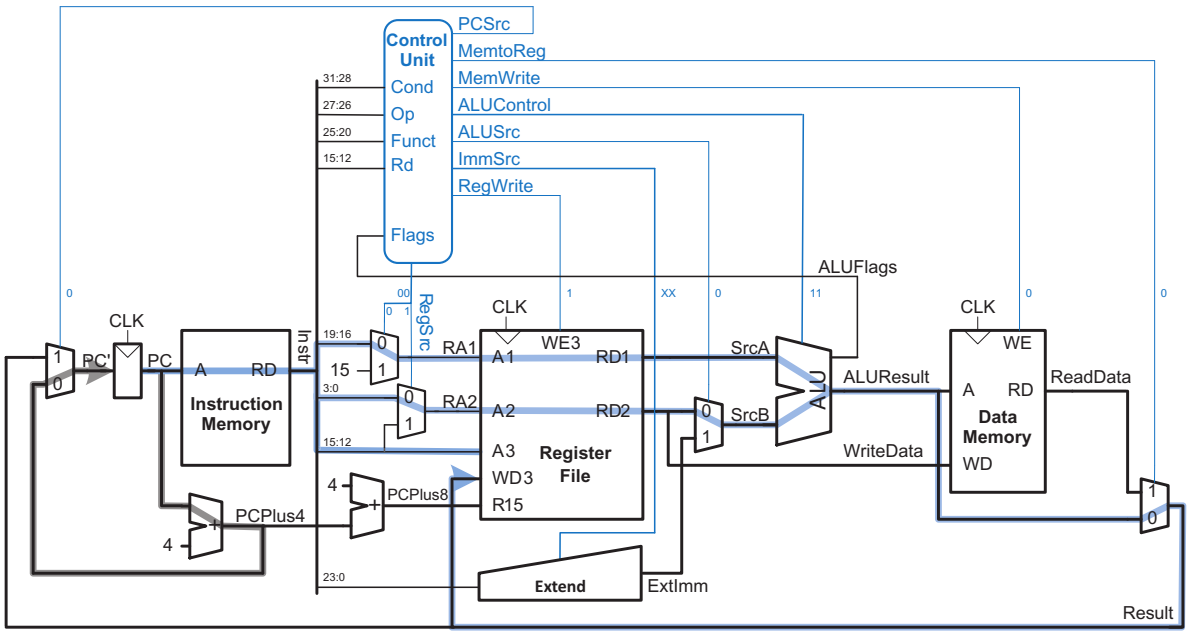


Figure 7.15 Control signals and data flow while executing an `ORR` instruction

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*.

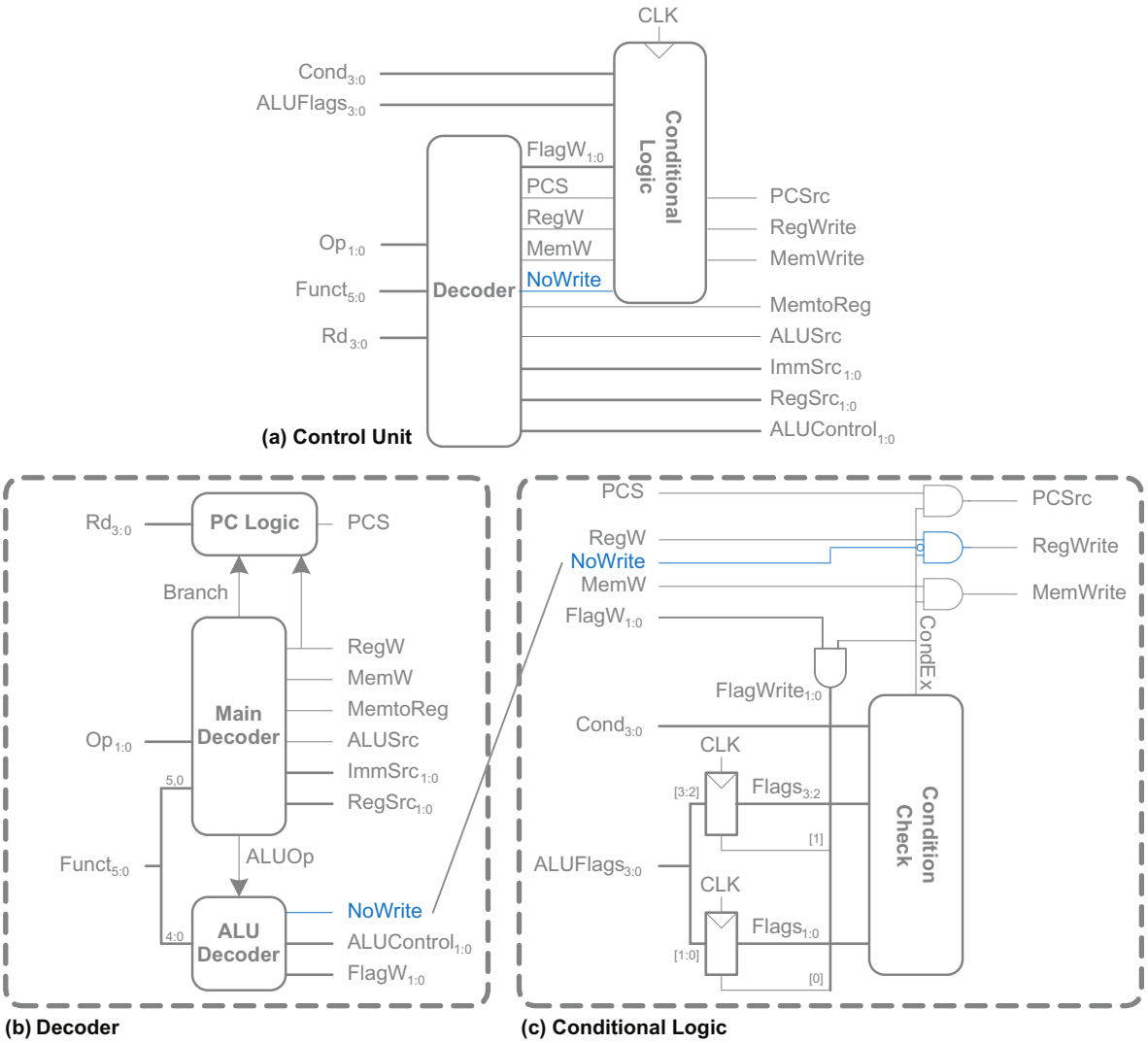


Figure 7.16 Controller modification for `CMP`

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

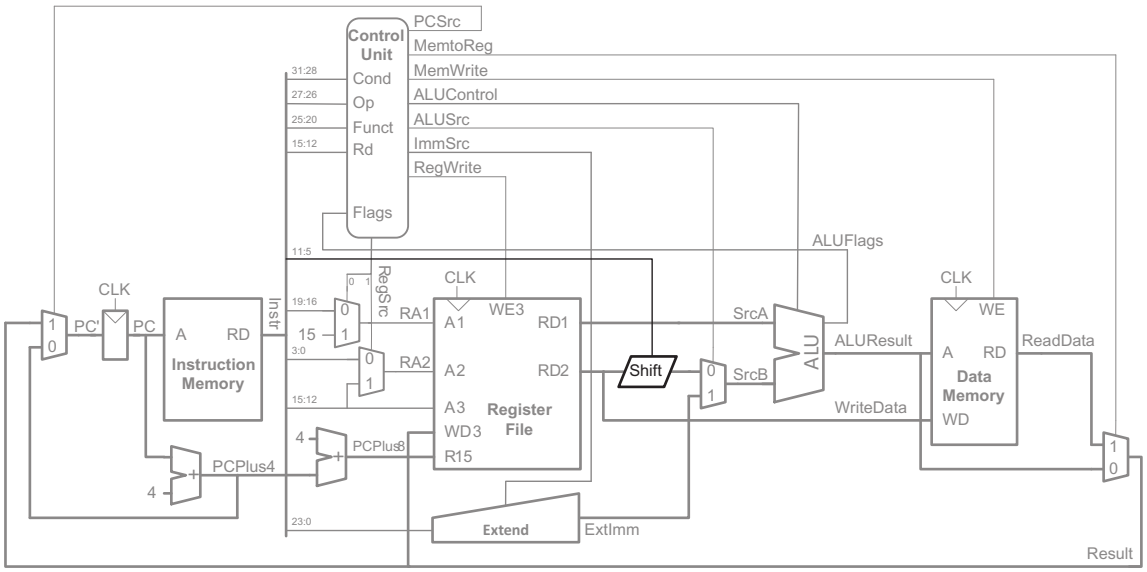


Figure 7.17 Enhanced datapath for register addressing with constant shifts

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

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

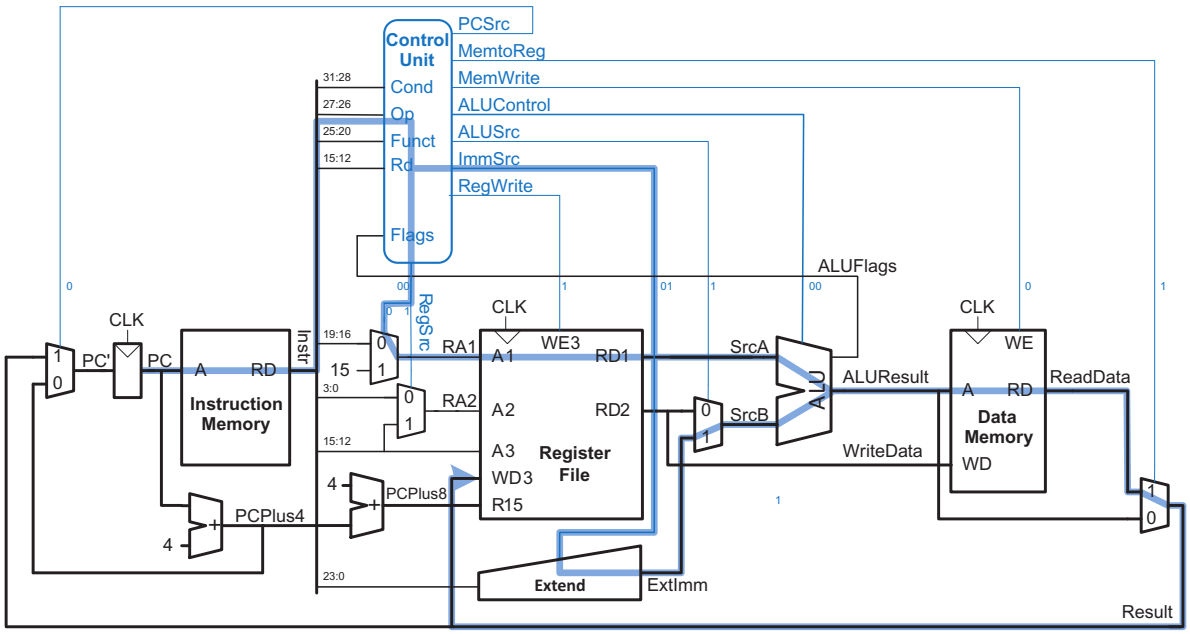


Figure 7.18 LDR **critical path**

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

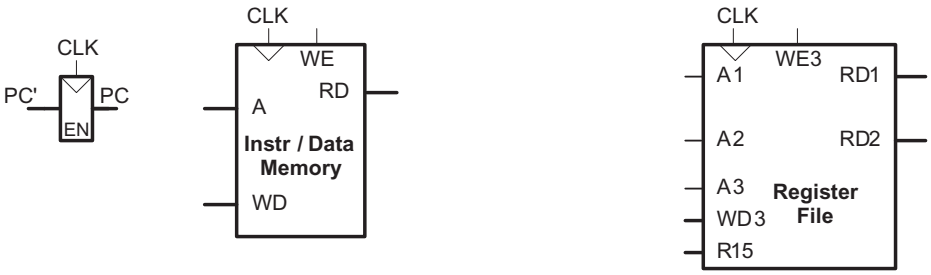


Figure 7.19 State elements with unified instruction/data memory

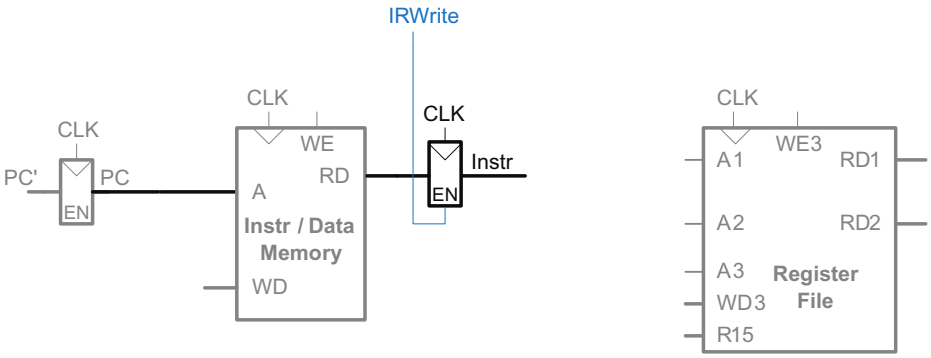


Figure 7.20 Fetch instruction from memory

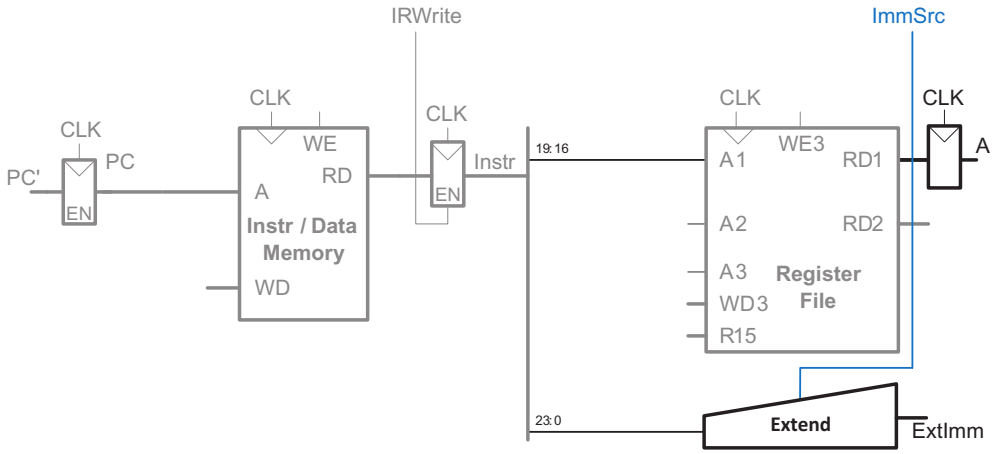


Figure 7.21 Read one source from register file and extend the second source from the immediate field

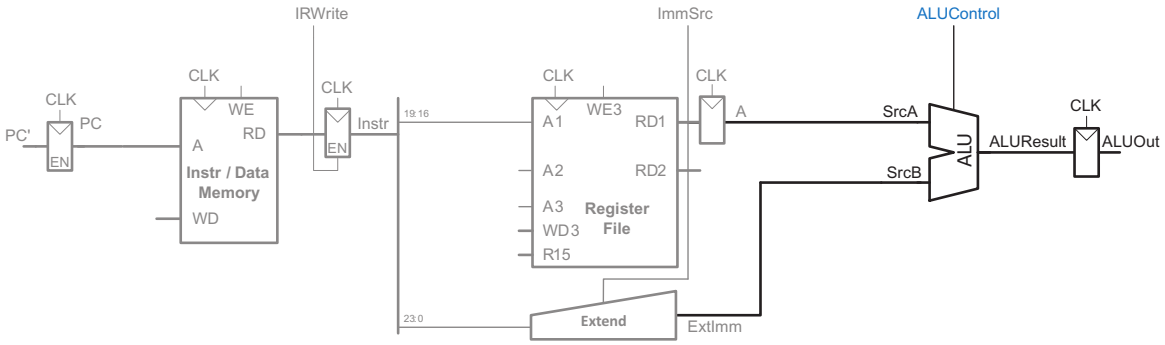


Figure 7.22 Add base address to offset

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

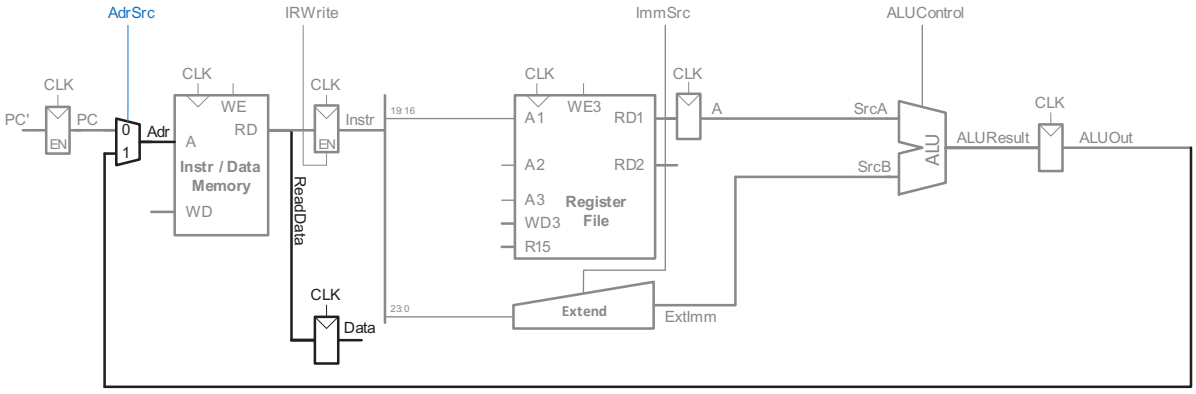


Figure 7.23 Load data from memory

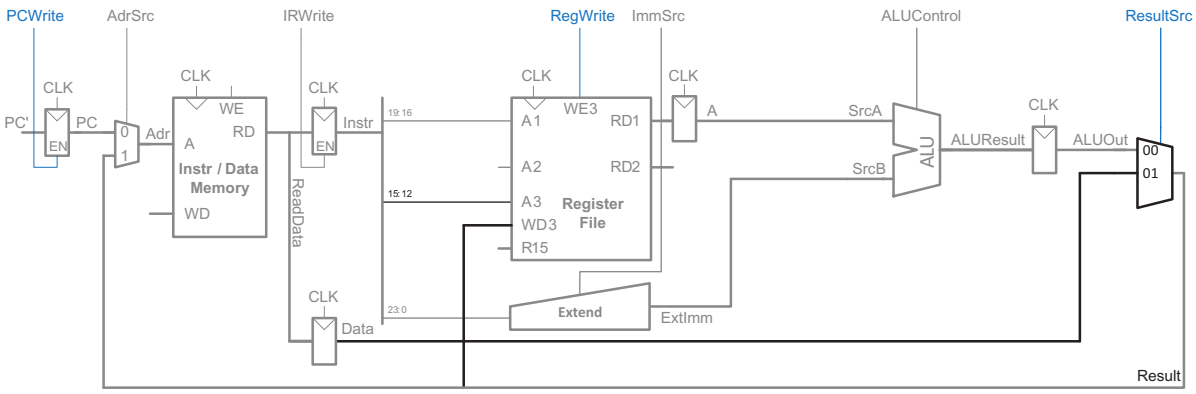


Figure 7.24 Write data back to register file

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.

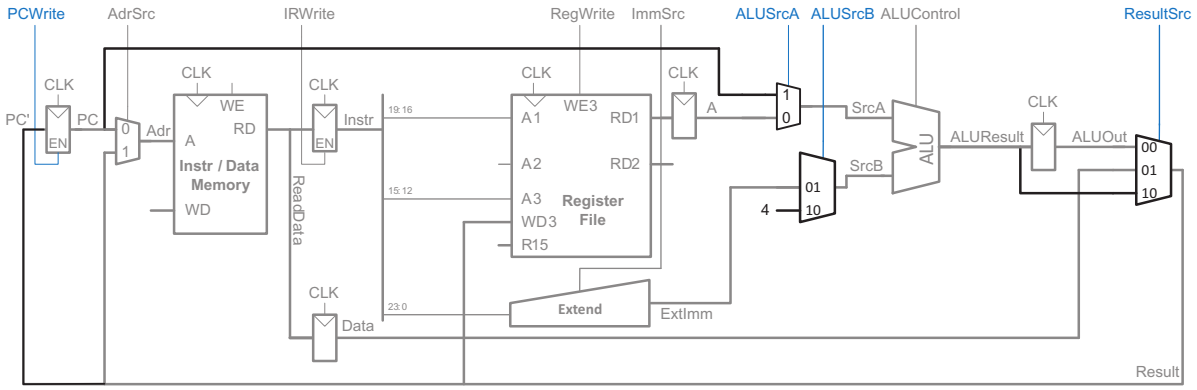


Figure 7.25 Increment PC by 4

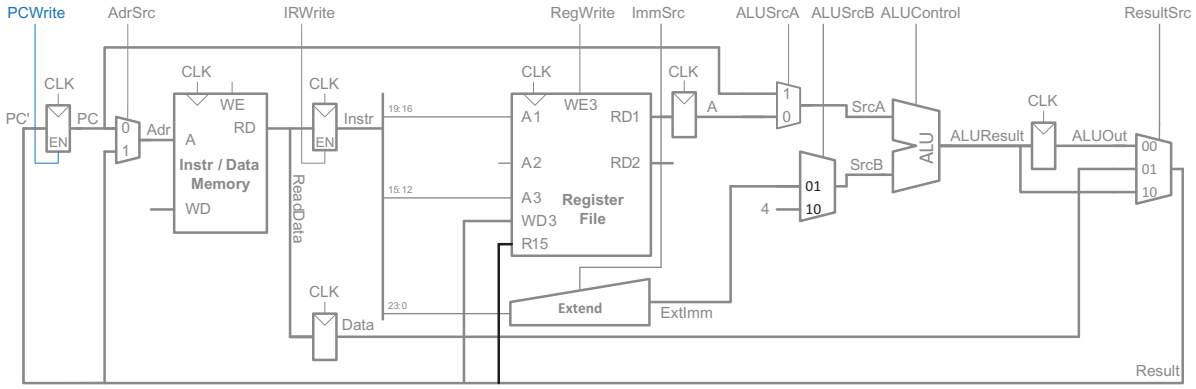


Figure 7.26 Handle R15 reads and writes

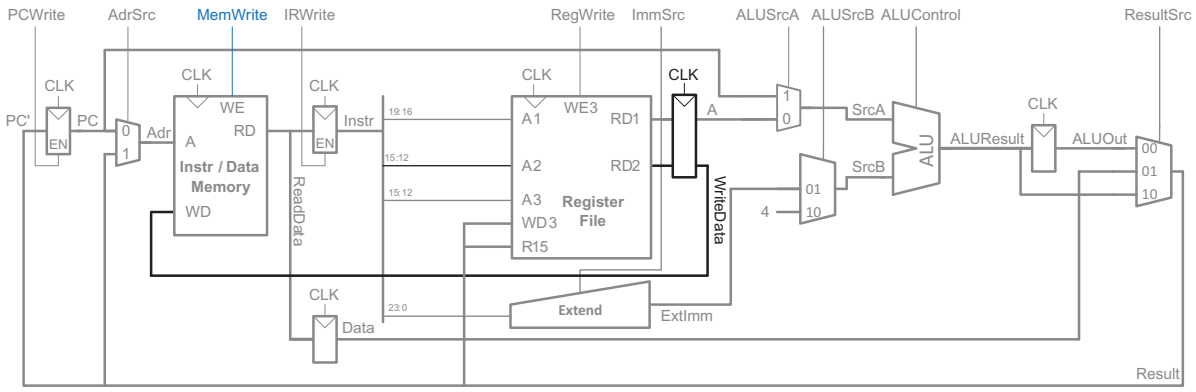


Figure 7.27 Enhanced datapath for STR instruction

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

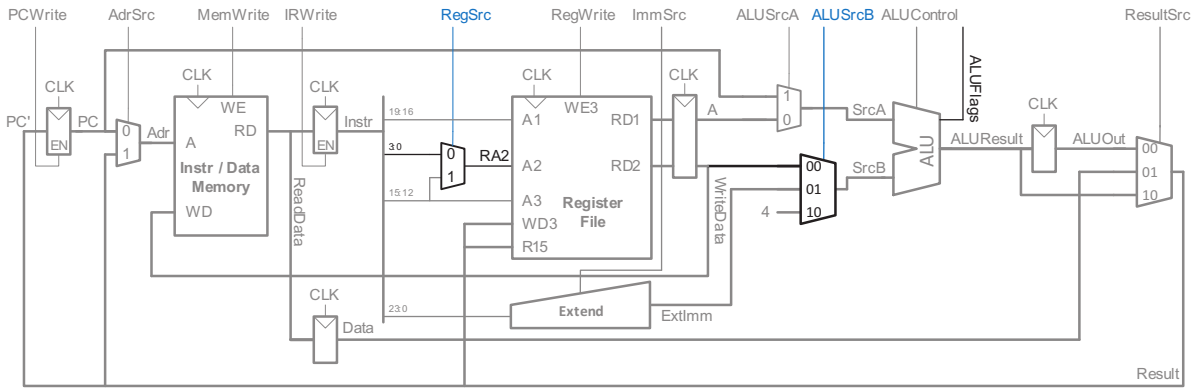


Figure 7.28 Enhanced datapath for data-processing instructions with register addressing

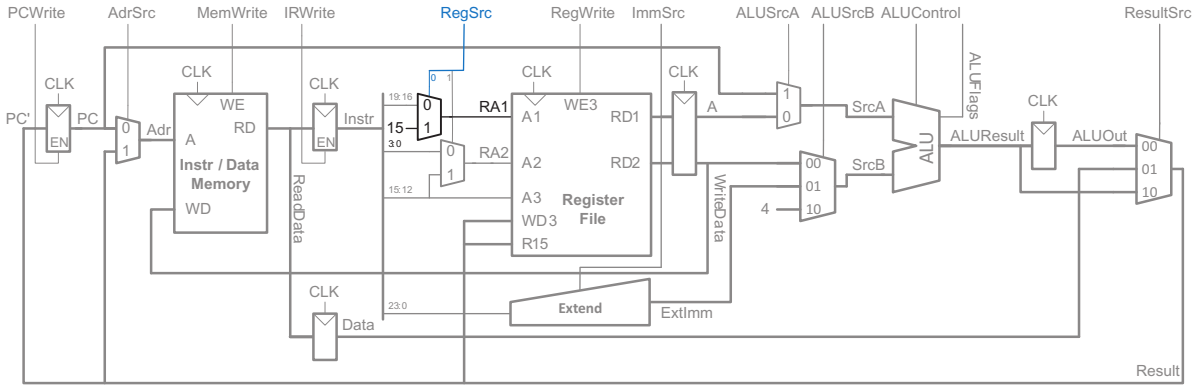


Figure 7.29 Enhanced datapath for the B instruction

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

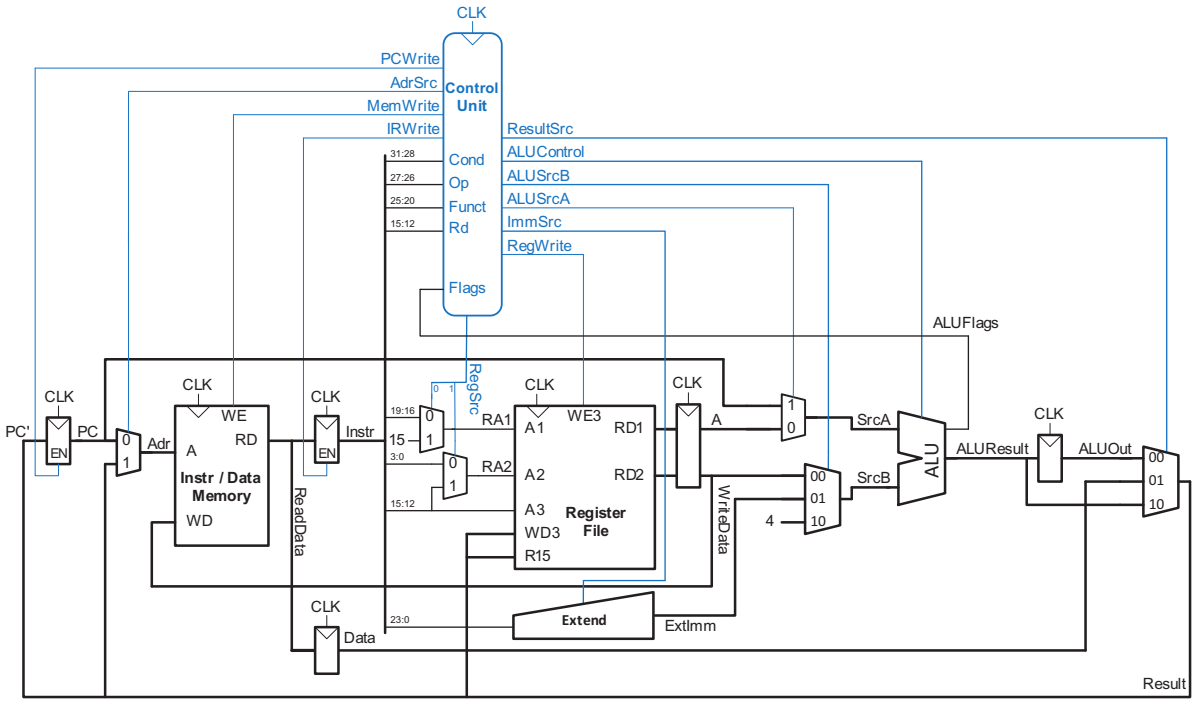


Figure 7.30 Complete multicycle processor

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

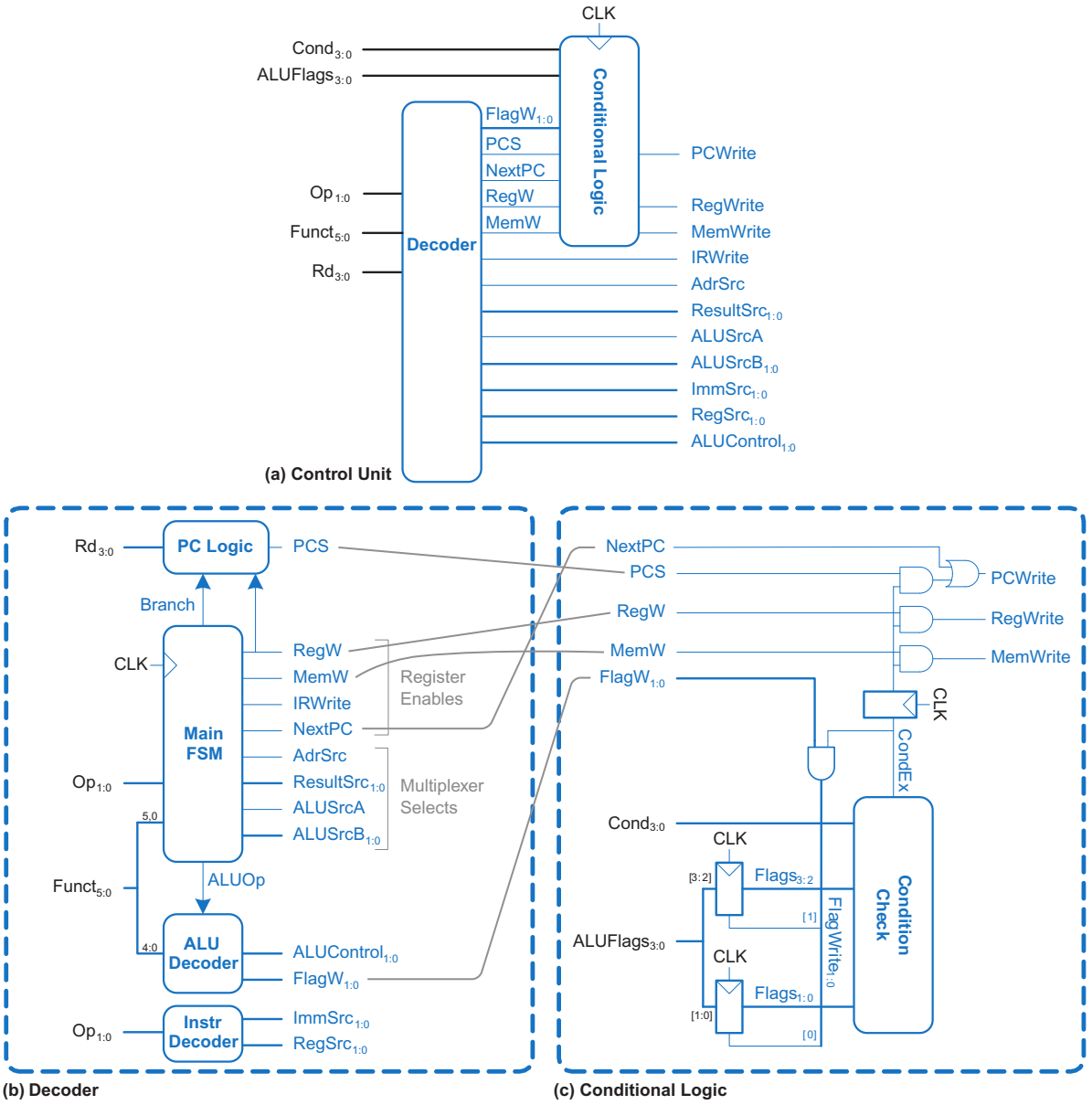


Figure 7.31 Multicycle control unit

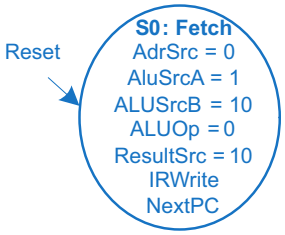


Figure 7.32 Fetch

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

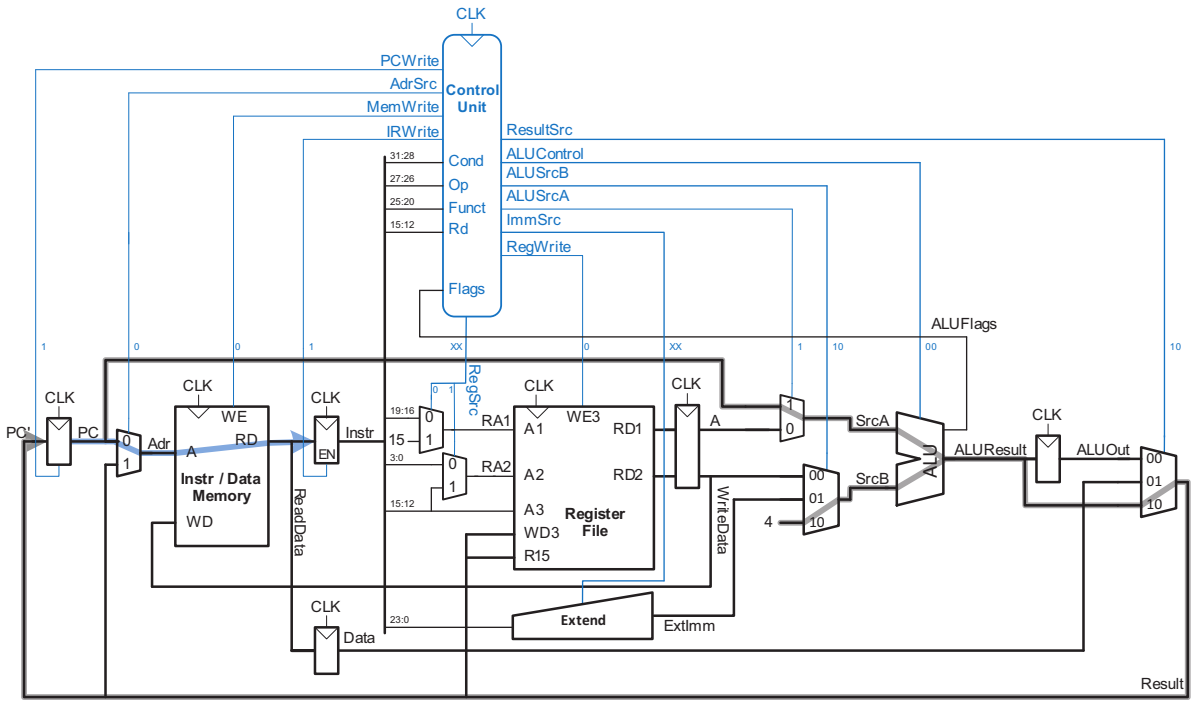


Figure 7.33 Data flow during the fetch step

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 $PCWrite$.

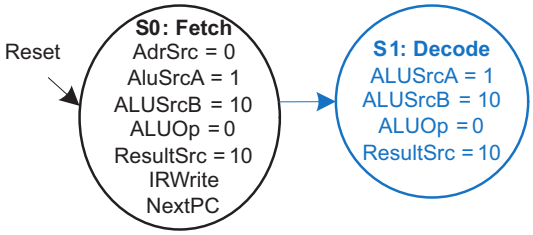


Figure 7.34 Decode

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

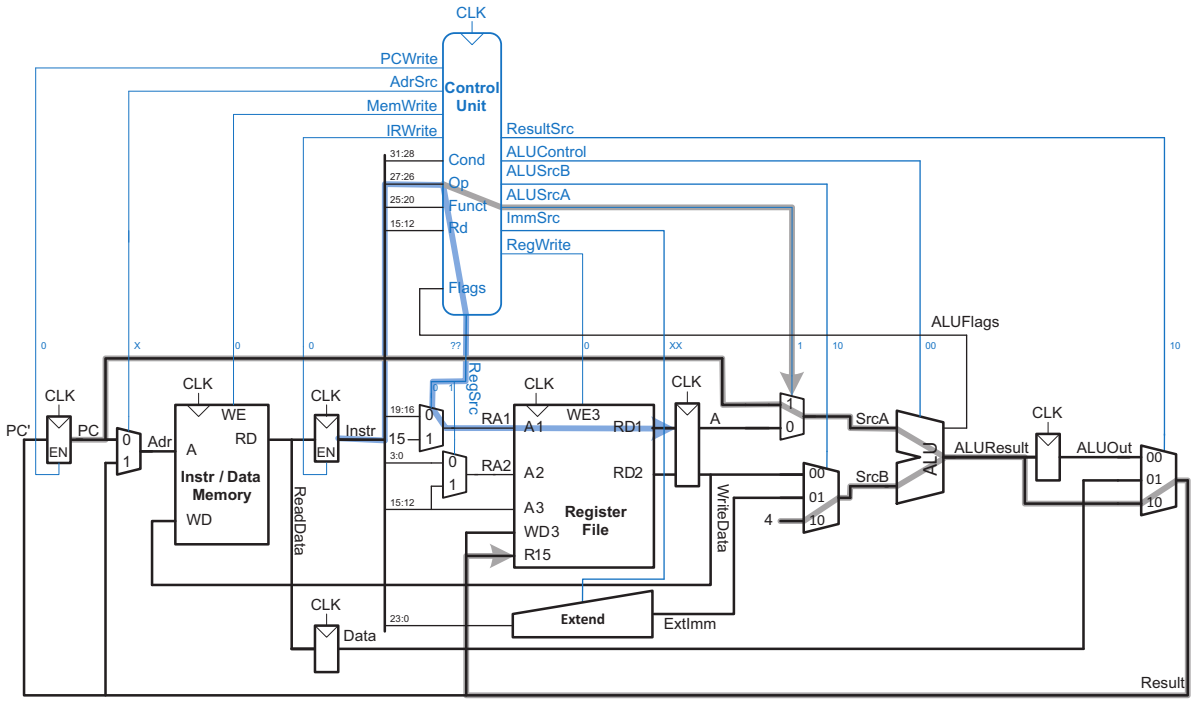


Figure 7.35 Data flow during the Decode step

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.

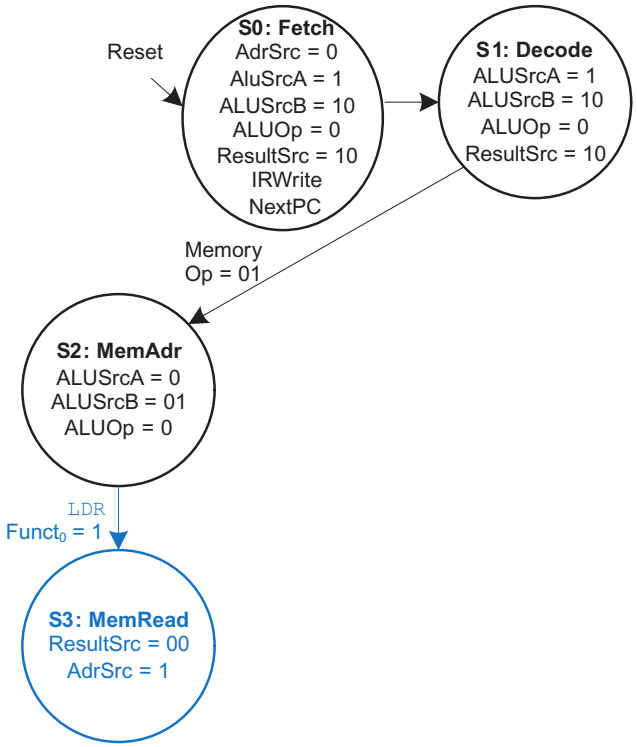


Figure 7.36 Memory address computation

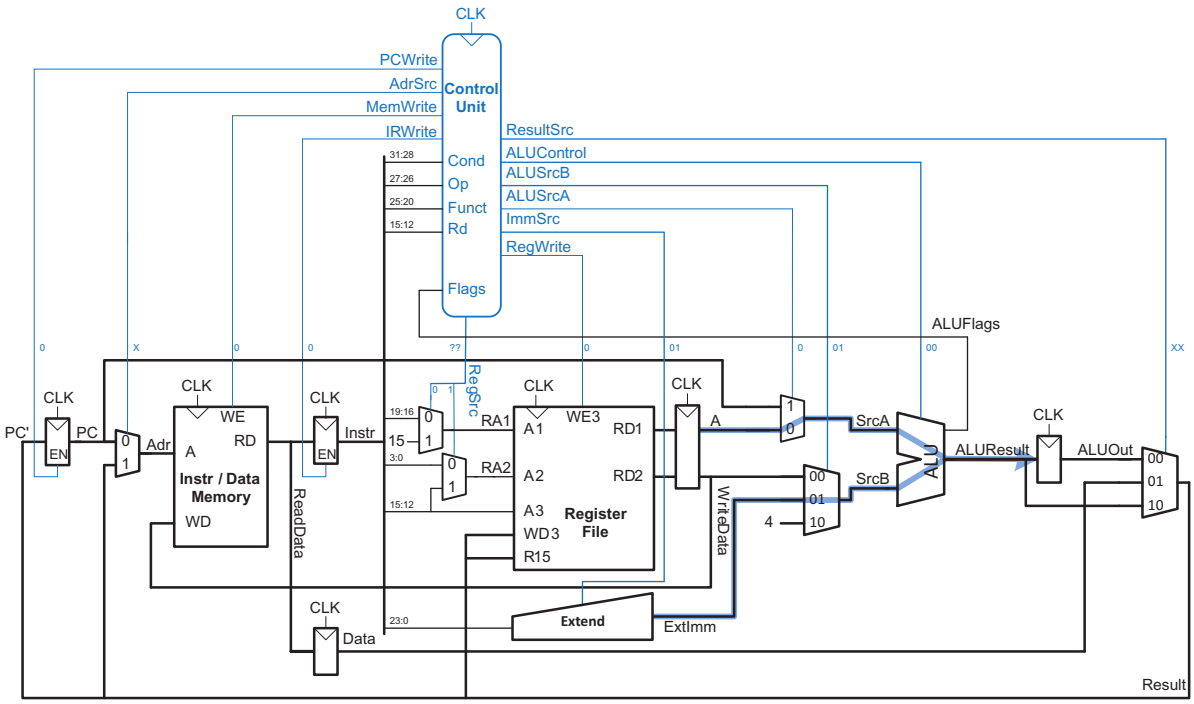


Figure 7.37 Data flow during memory address computation

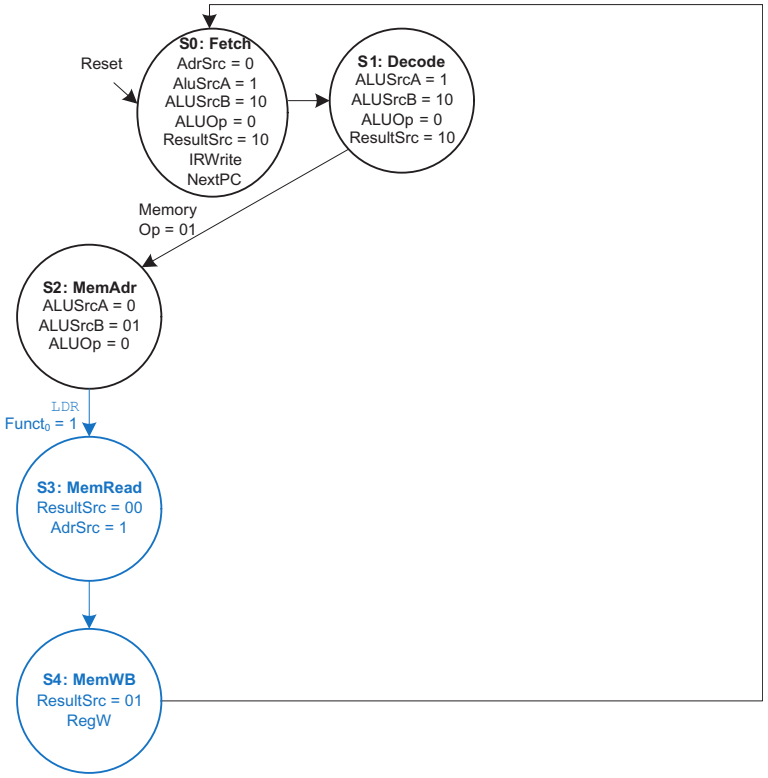


Figure 7.38 Memory read

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 $ALUOp = 0$

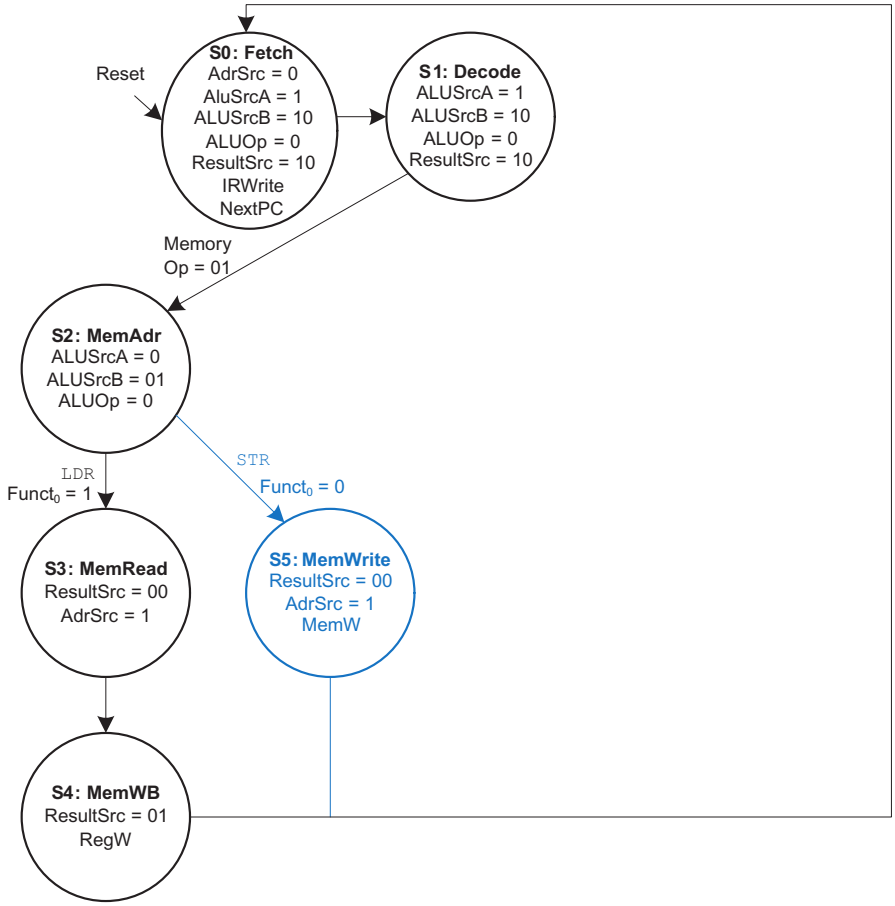


Figure 7.39 Memory write

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

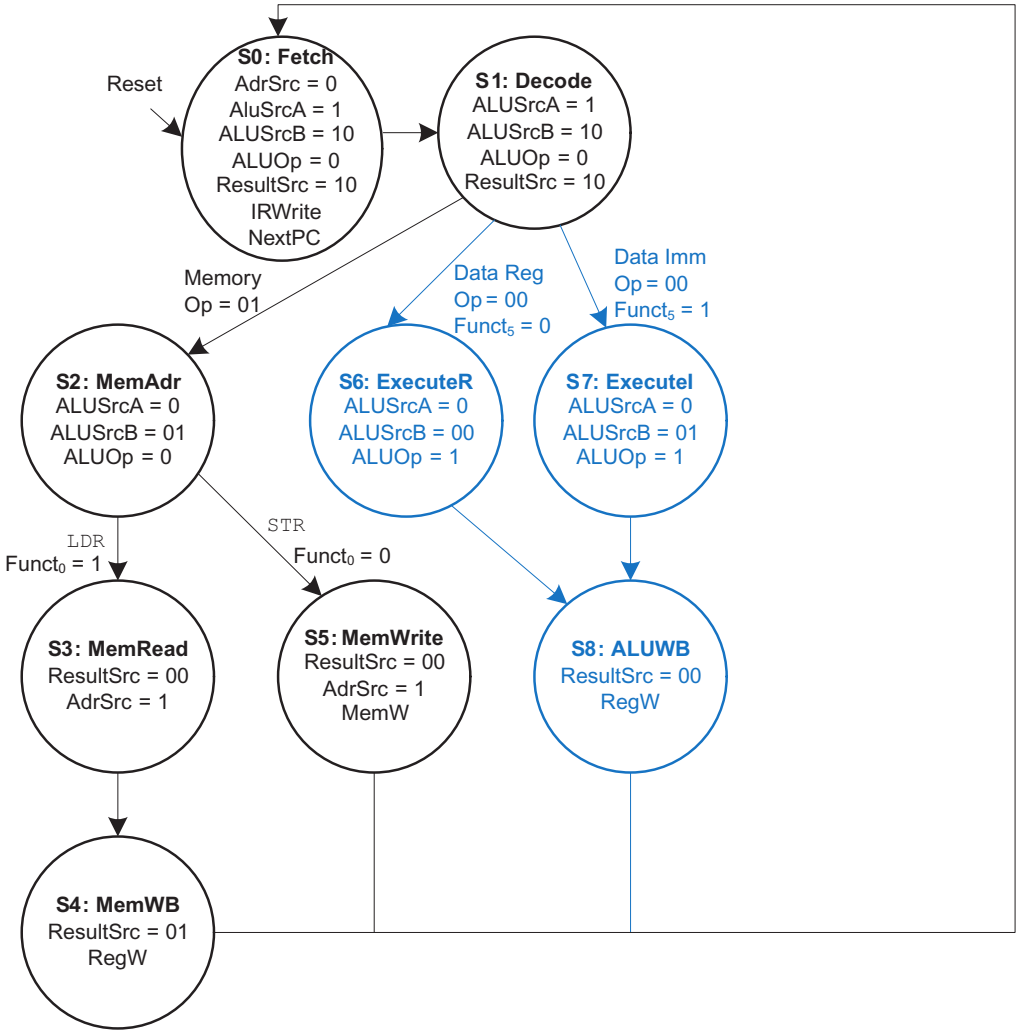
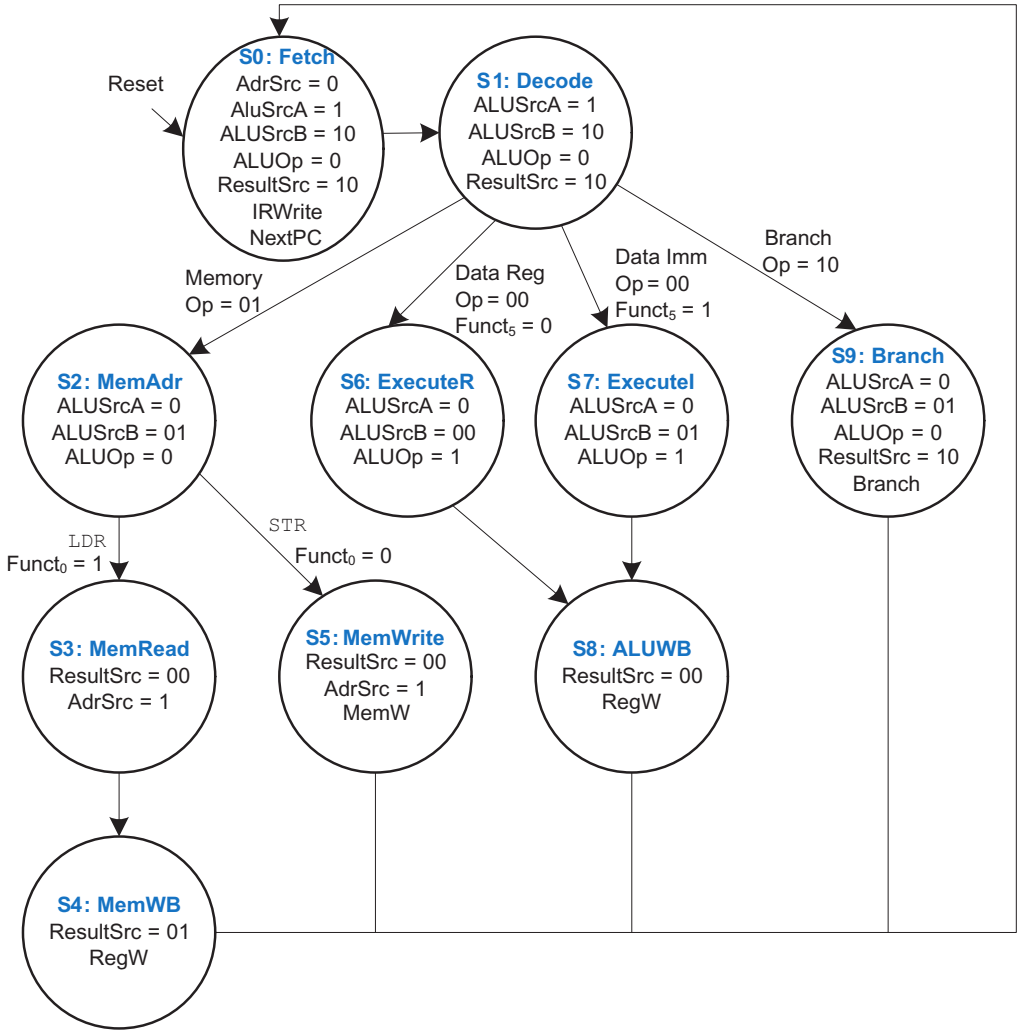


Figure 7.40 Data-processing

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.

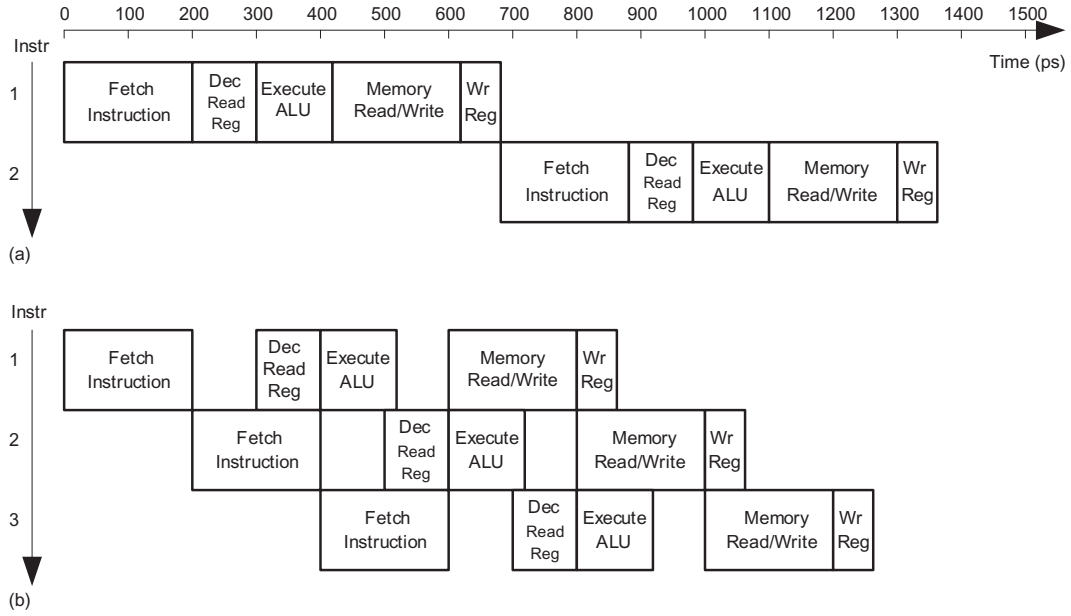


Figure 7.42 Timing diagrams: (a) single-cycle processor and (b) pipelined processor

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

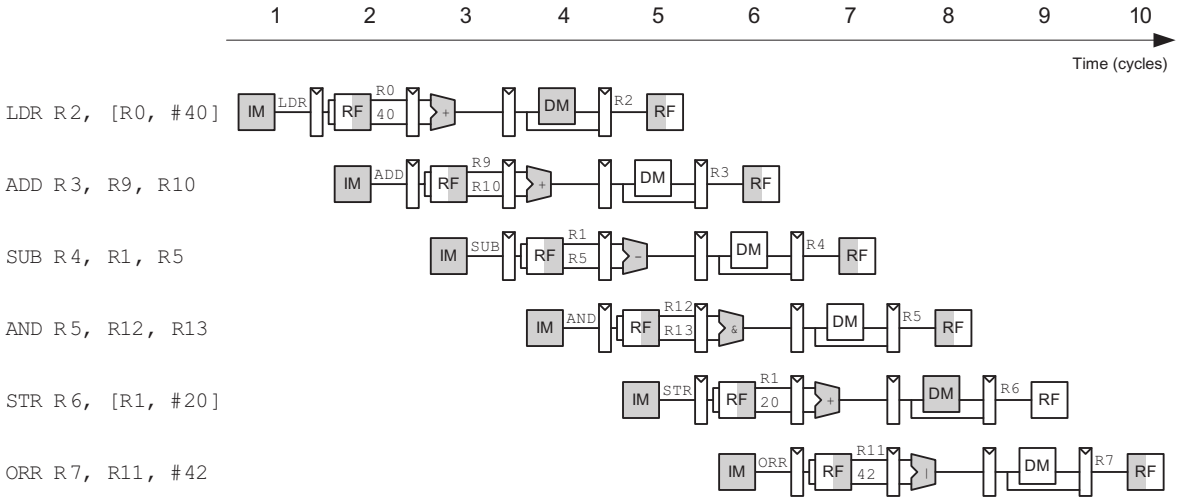


Figure 7.43 Abstract view of pipeline in operation

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

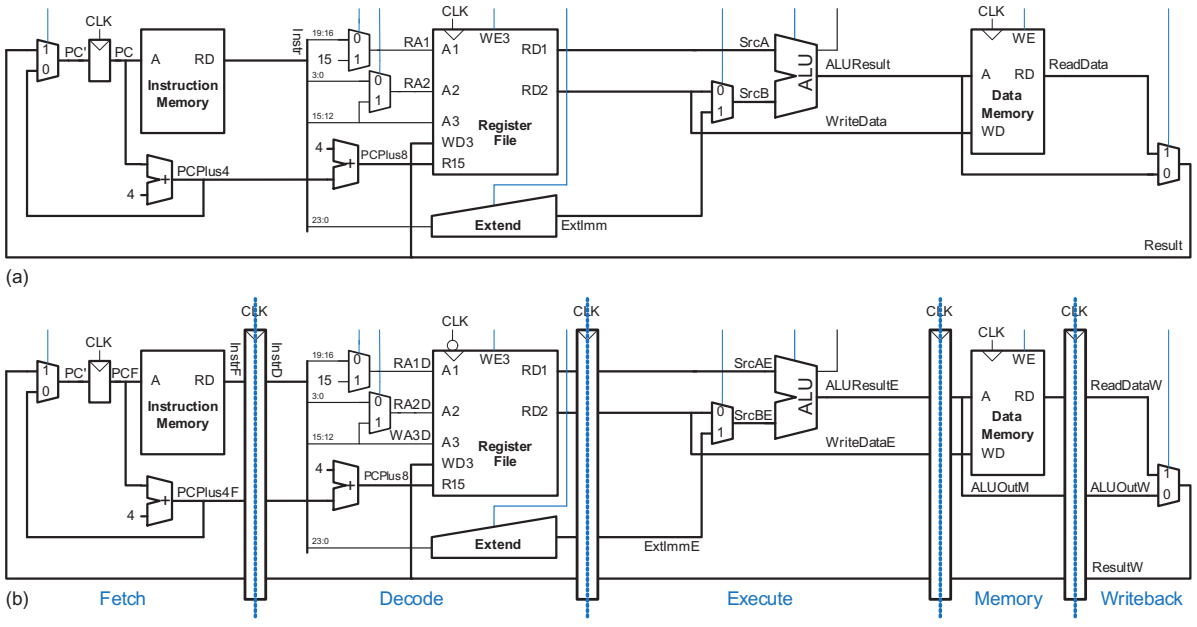


Figure 7.44 Datapaths: (a) single-cycle and (b) pipelined

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]

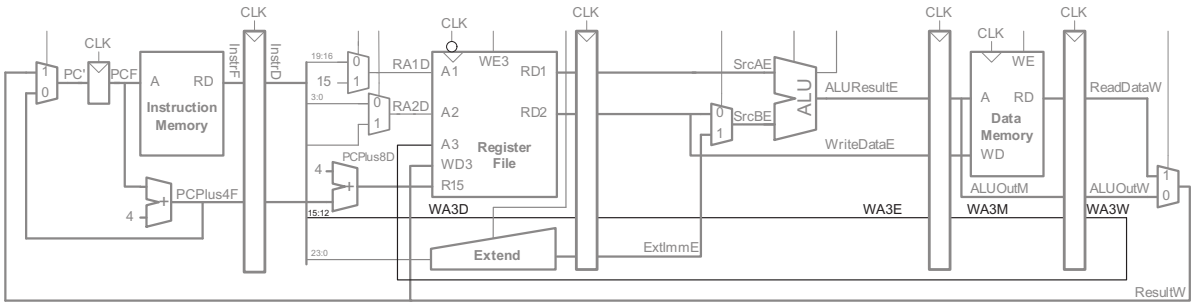


Figure 7.45 Corrected pipelined datapath

[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.

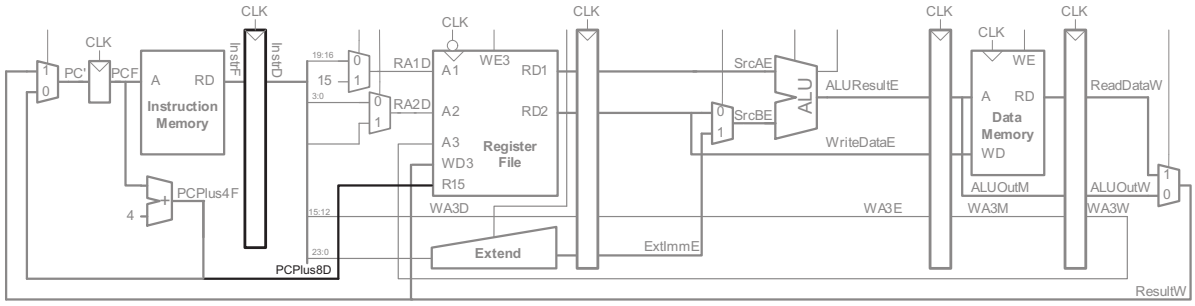


Figure 7.46 Optimized PC logic eliminating a register and adder

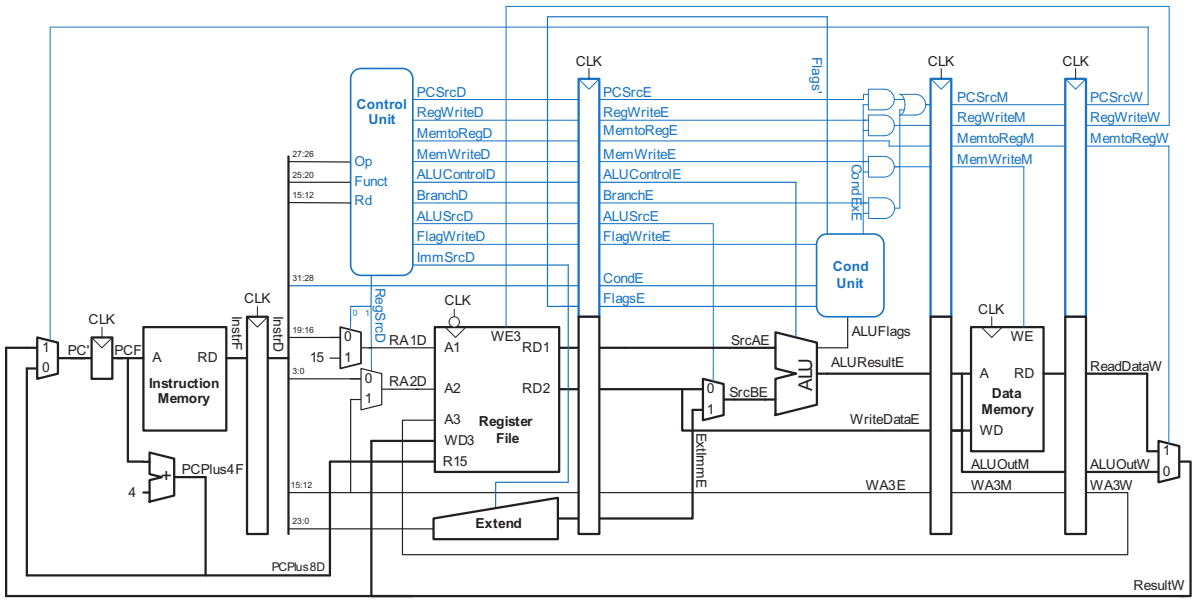


Figure 7.47 Pipelined processor with control

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.

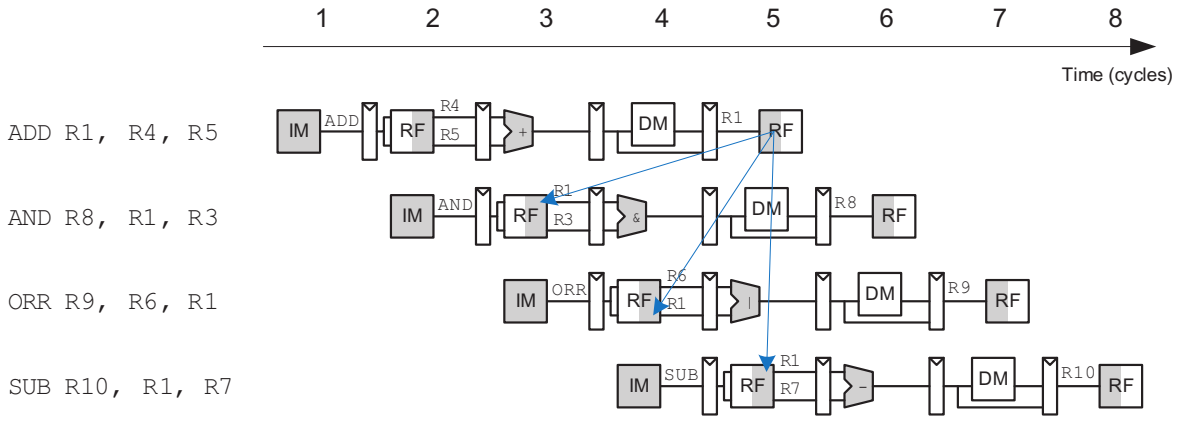


Figure 7.48 Abstract pipeline diagram illustrating hazards

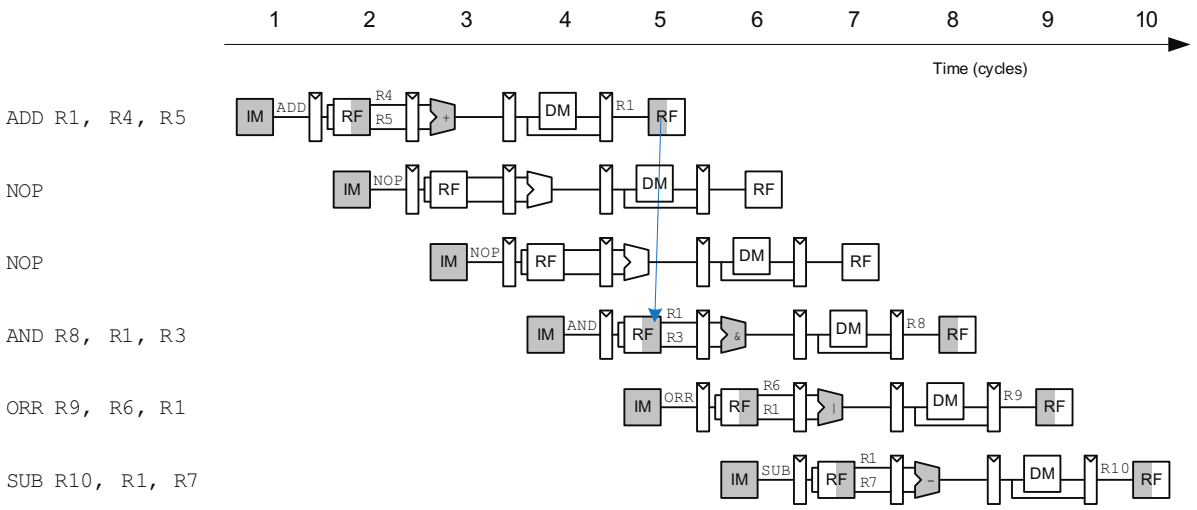


Figure 7.49 **Solving data hazard with** `NOP`

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

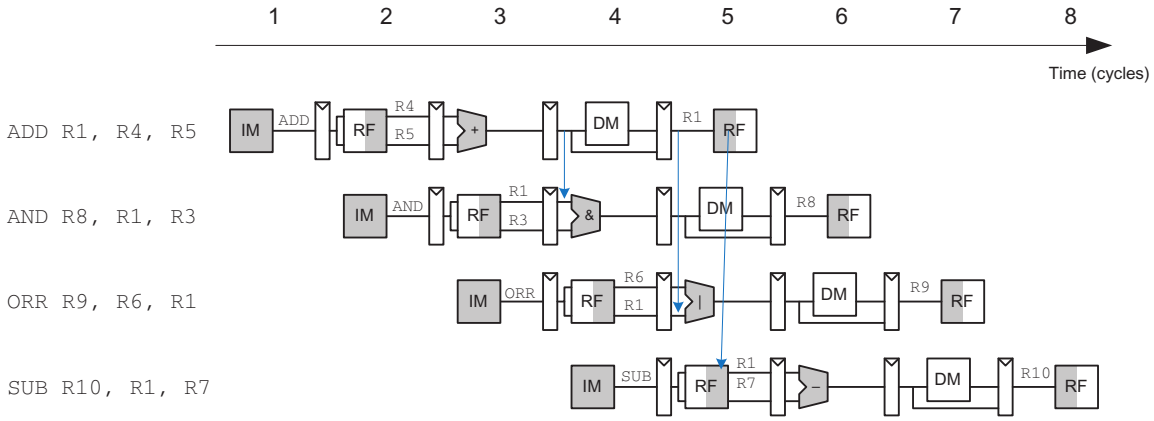


Figure 7.50 Abstract pipeline diagram illustrating forwarding

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).

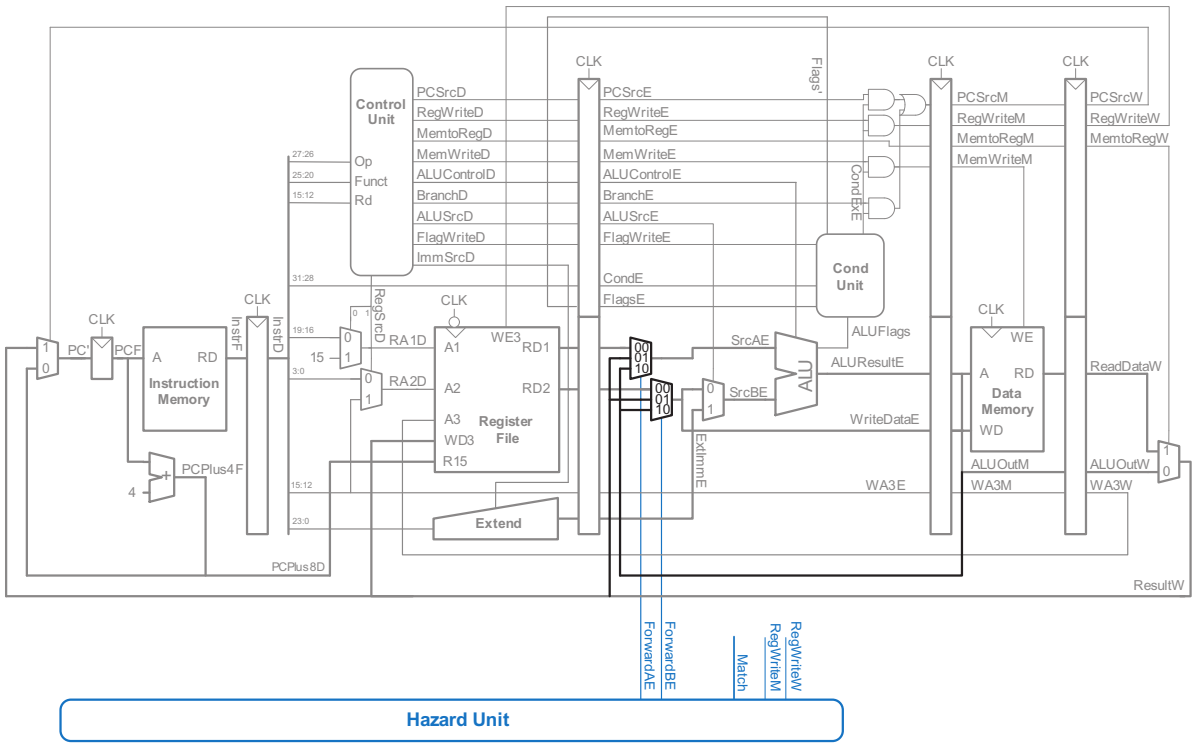


Figure 7.51 Pipelined processor with forwarding to solve hazards

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

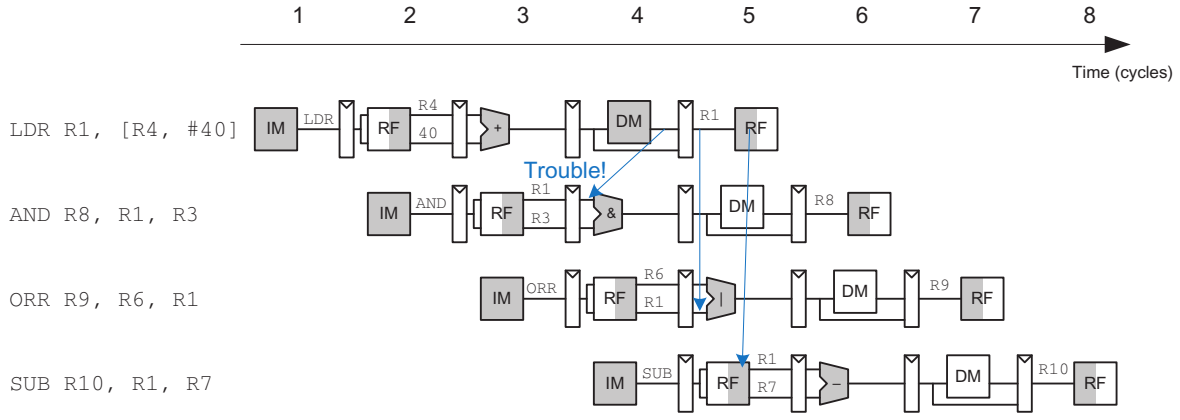


Figure 7.52 Abstract pipeline diagram illustrating trouble forwarding from LDR

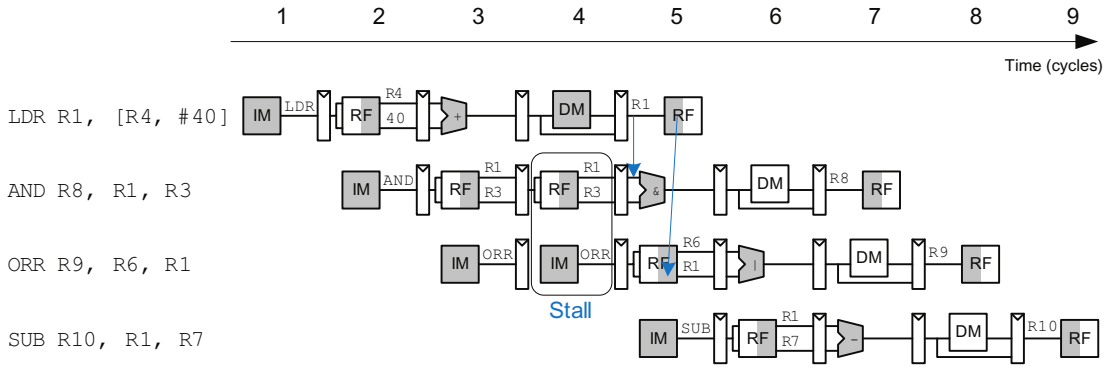


Figure 7.53 Abstract pipeline diagram illustrating stall to solve hazards

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
```

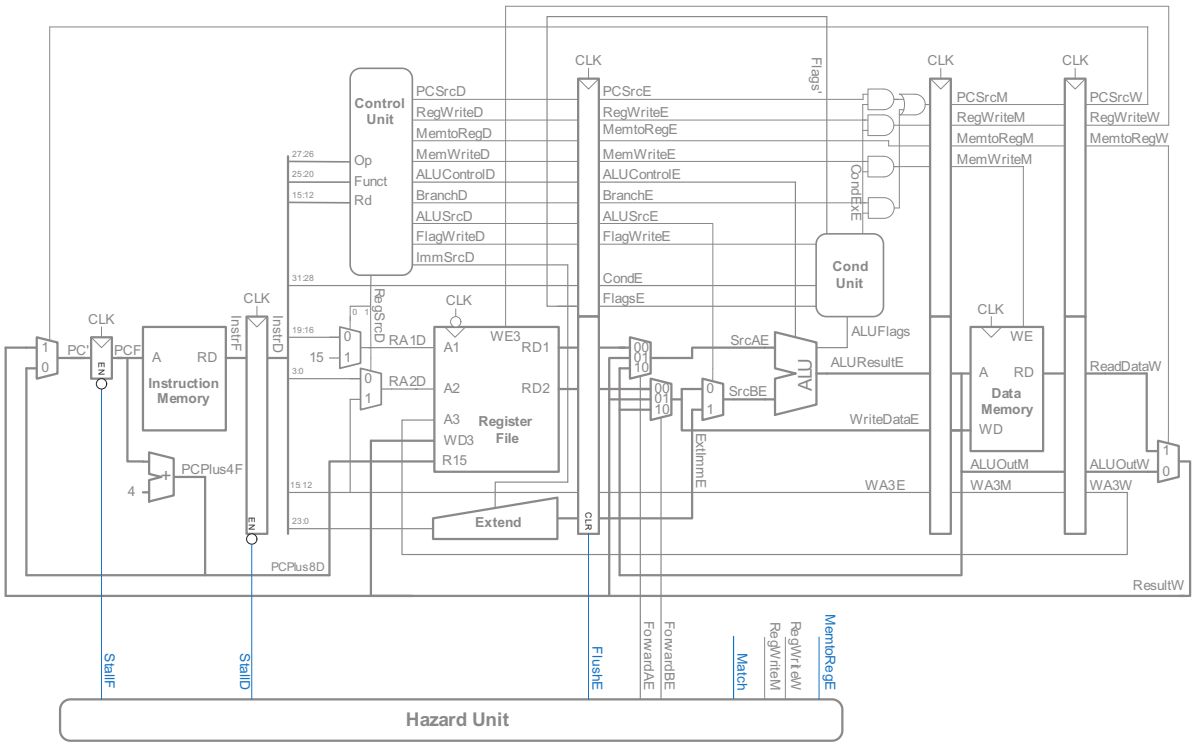


Figure 7.54 **Pipelined processor with stalls to solve** `LDR` **data hazard**

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

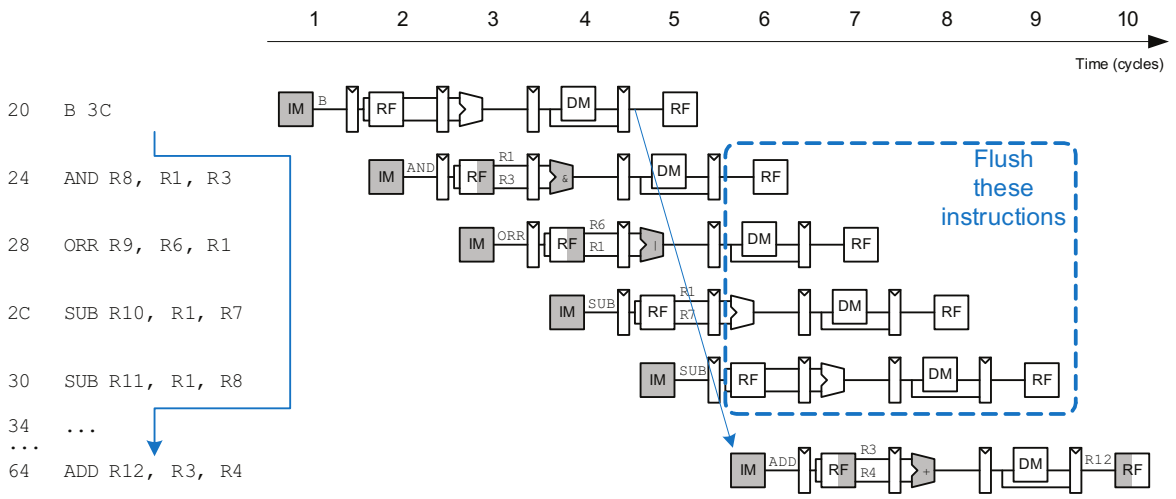


Figure 7.55 Abstract pipeline diagram illustrating flushing when a branch is taken

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.

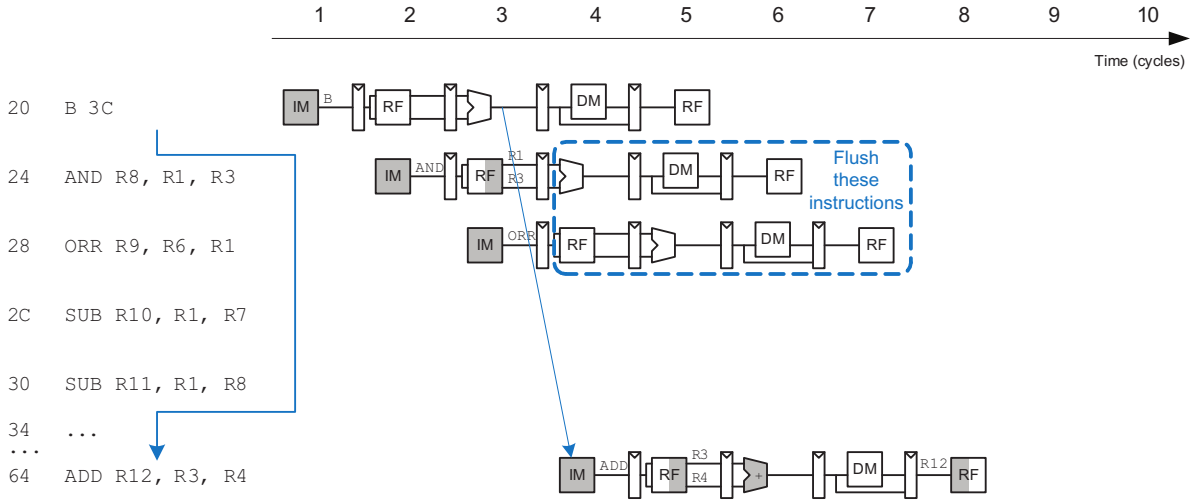


Figure 7.56 Abstract pipeline diagram illustrating earlier branch decision

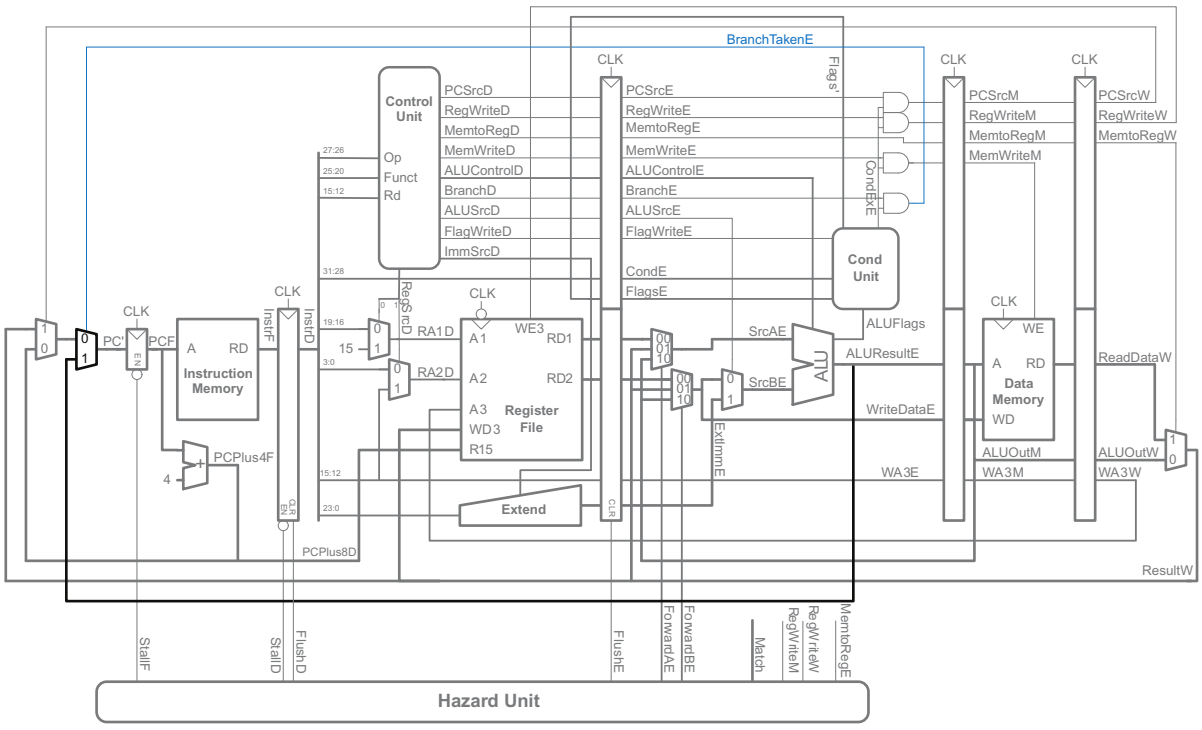


Figure 7.57 Pipelined processor handling branch control hazard

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

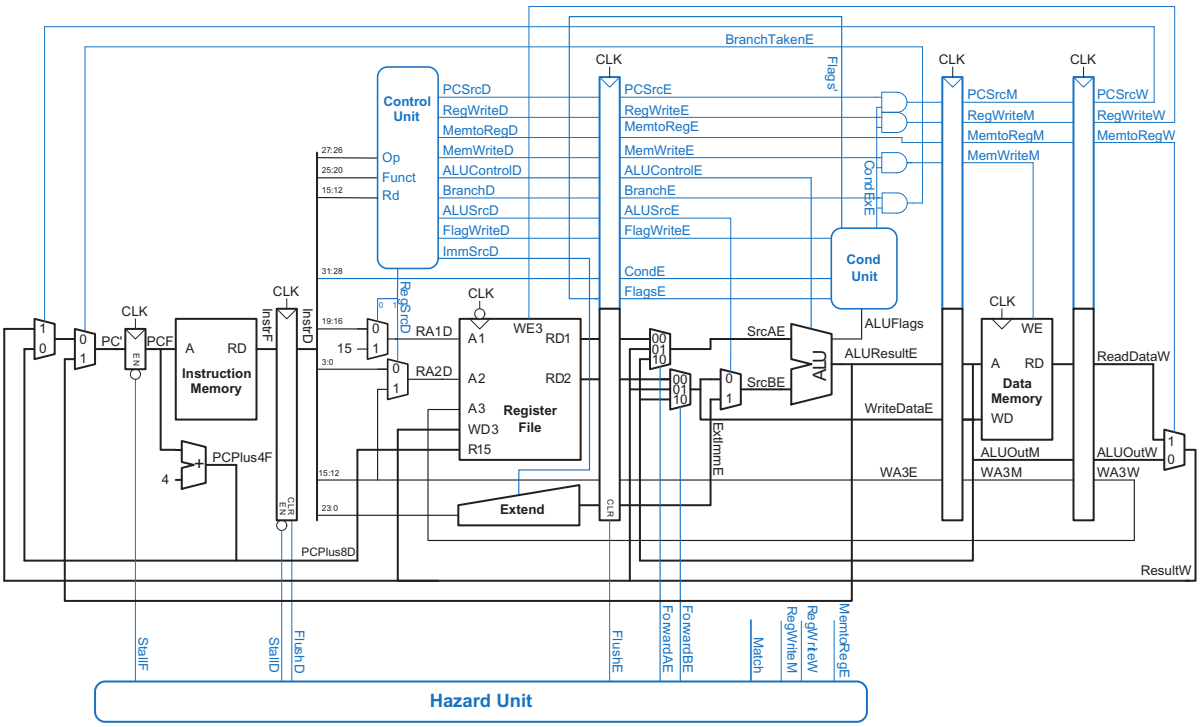


Figure 7.58 Pipelined processor with full hazard handling

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).

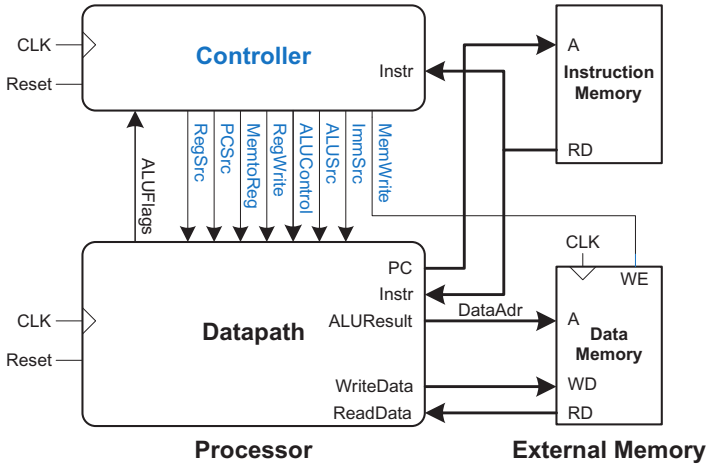


Figure 7.59 Single-cycle processor interfaced to external memory

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 "00-0"  => 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, DataAadr: 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.

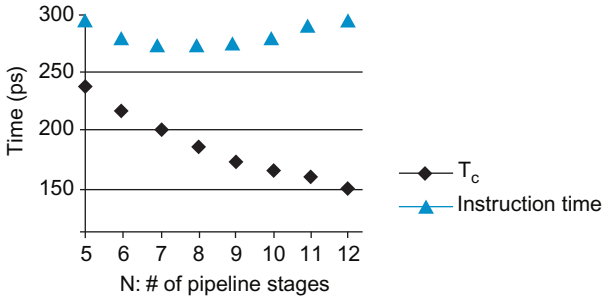


Figure 7.61 Cycle time and instruction time vs. the number of pipeline stages

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.

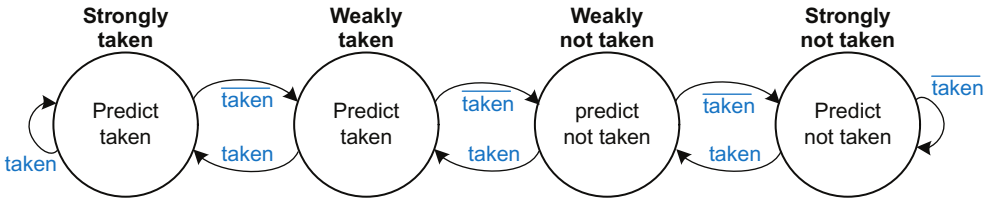


Figure 7.62 Two-bit branch predictor state transition diagram

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.

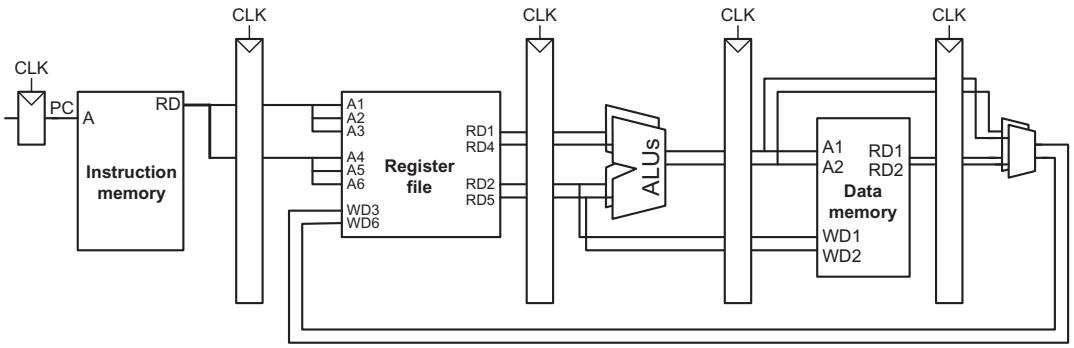


Figure 7.63 Superscalar datapath

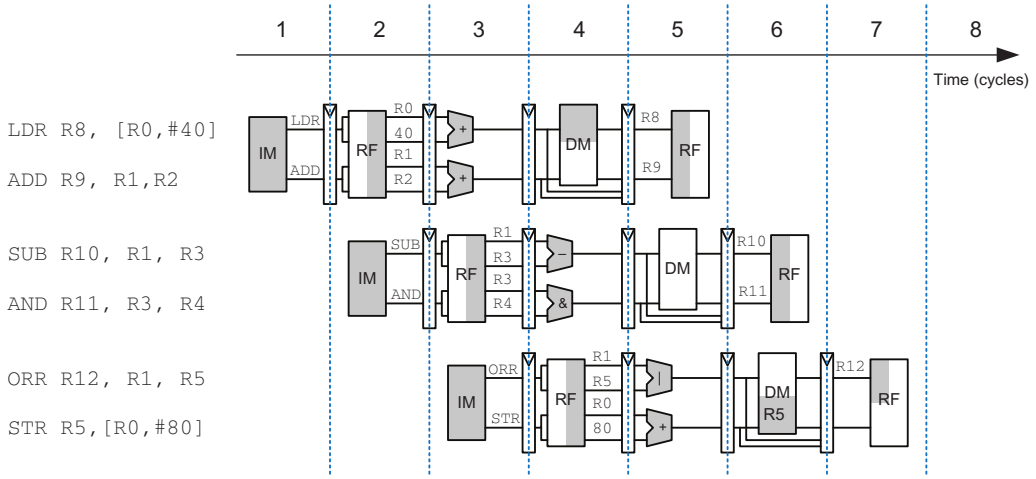


Figure 7.64 Abstract view of a superscalar pipeline in operation

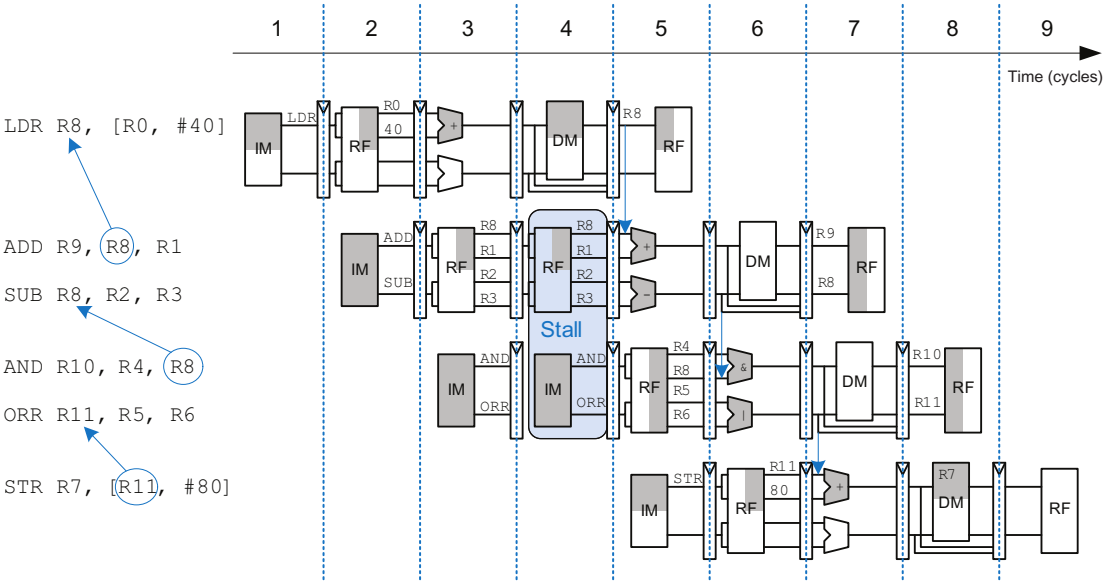


Figure 7.65 Program with data dependencies

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

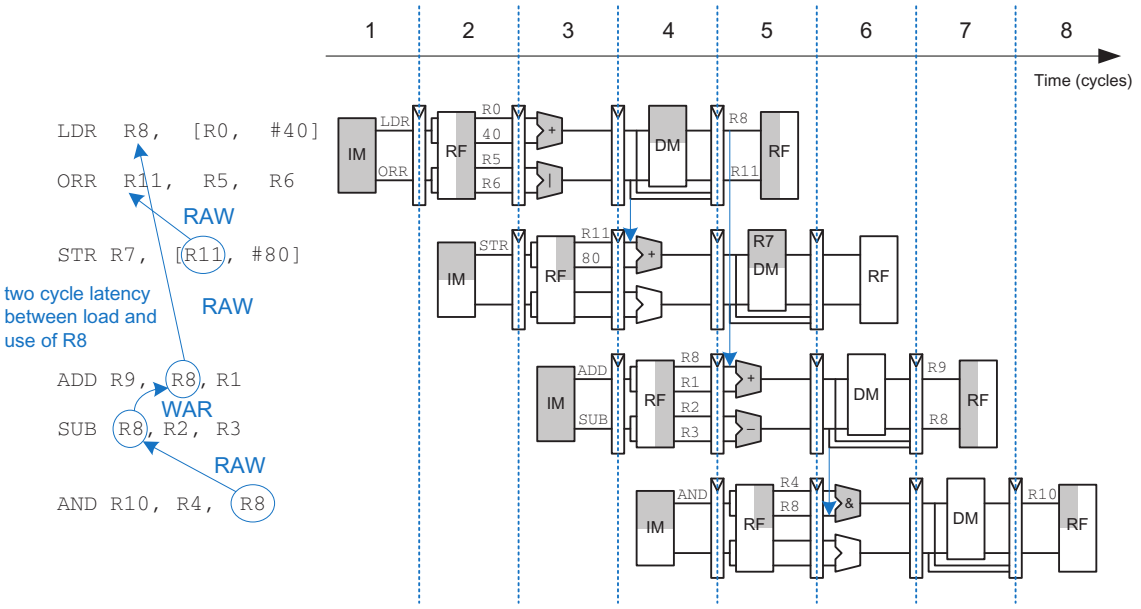


Figure 7.66 Out-of-order execution of a program with dependencies

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.

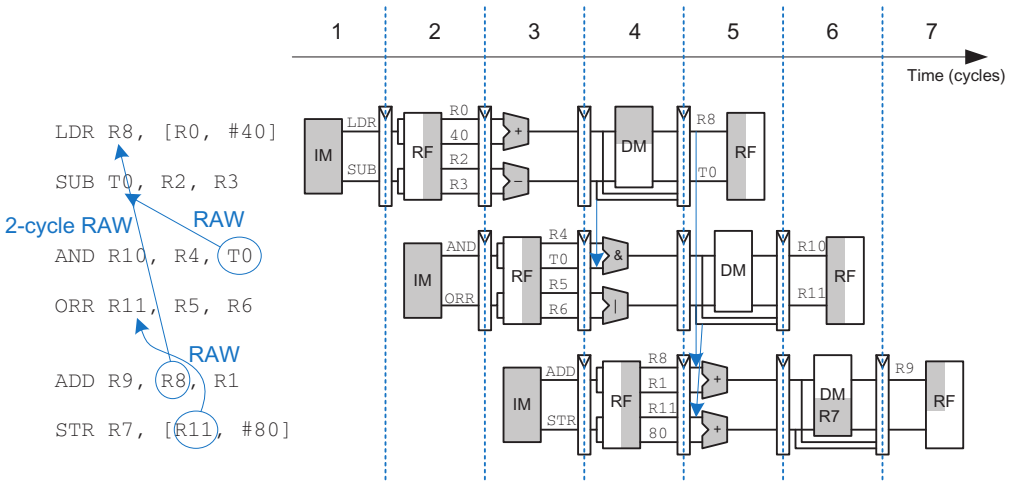


Figure 7.67 Out-of-order execution of a program using register renaming

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

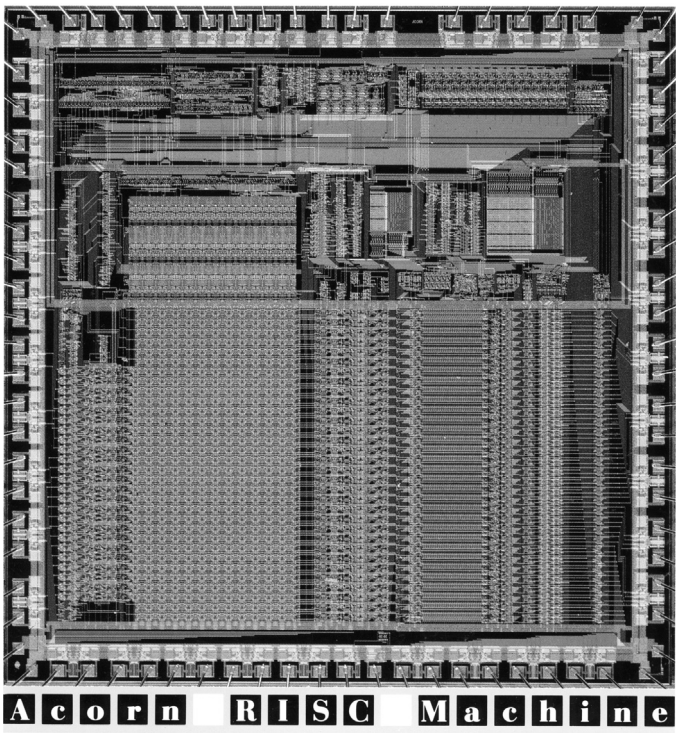


Figure 7.68 ARM1 die photograph
(Reproduced with permission from ARM. © 1985 ARM Ltd.)

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.)

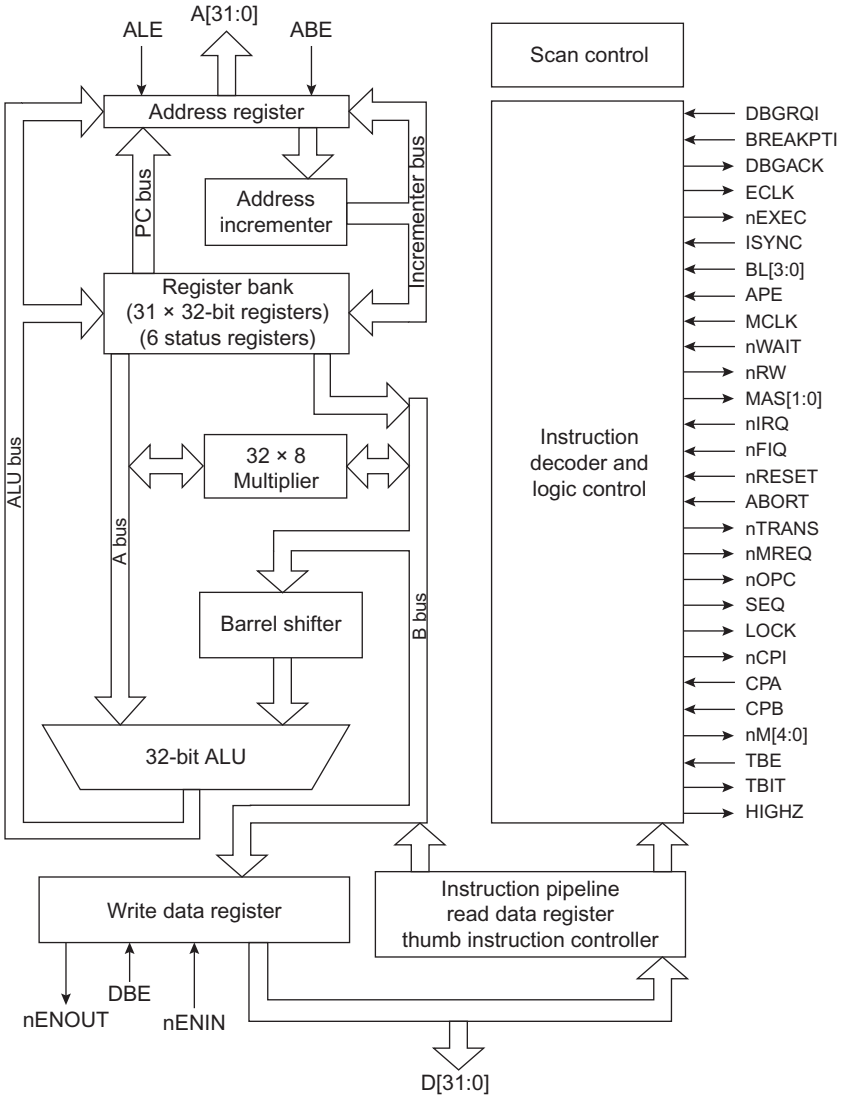


Figure 7.69 **ARM7 block diagram** (Reproduced with permission from ARM. © 1998 ARM Ltd.)

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.)

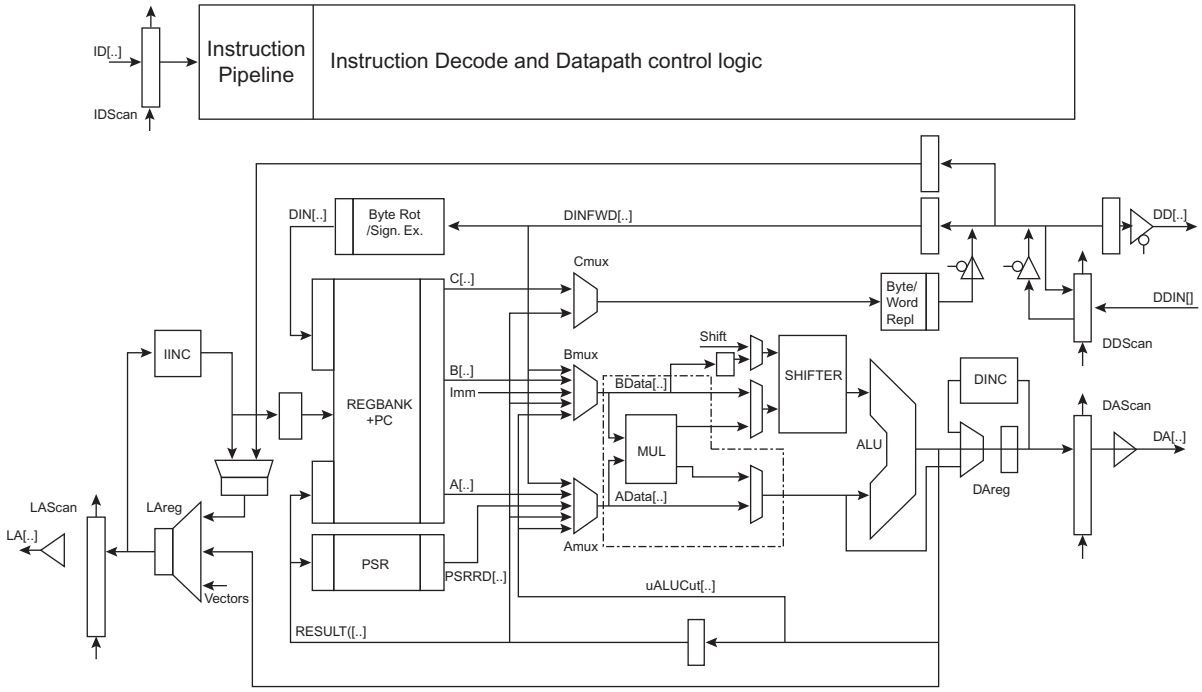


Figure 7.70 ARM9 block diagram

(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

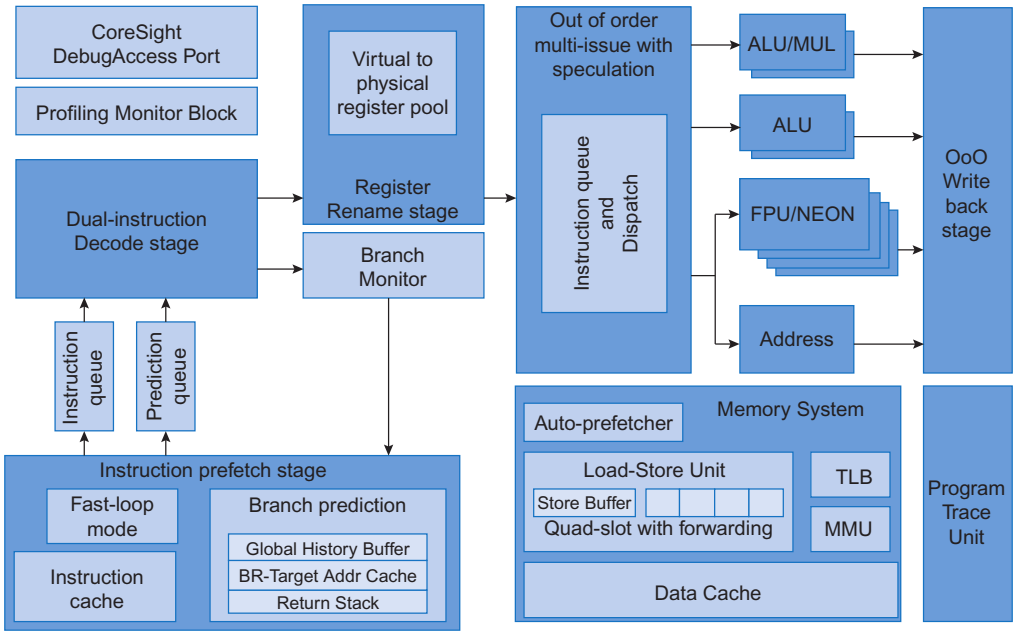


Figure 7.71 **Cortex-A9 block diagram**
(This image has been sourced by the authors and does not imply ARM endorsement.)

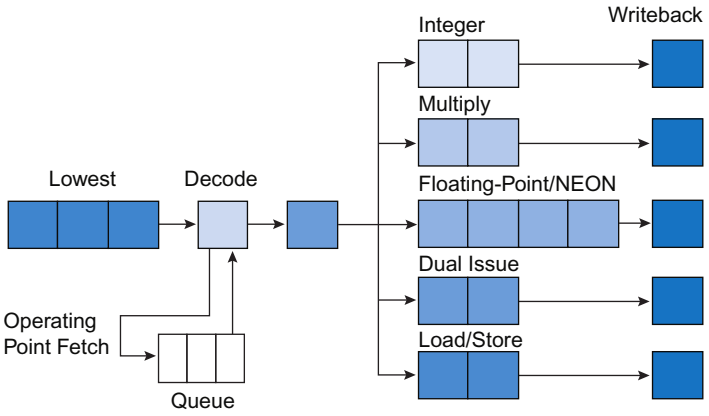
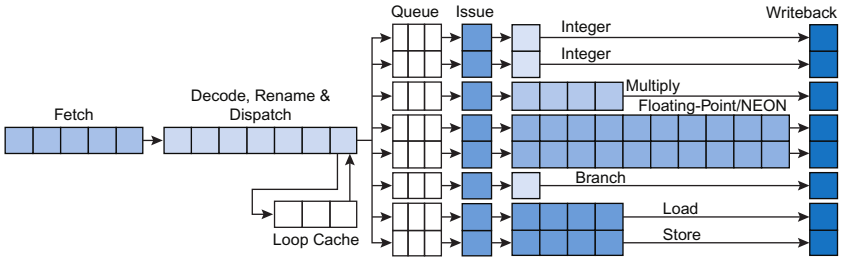


Figure 7.72 **Cortex-A7 and -A15 block diagrams**
(This image has been sourced by the authors and does not imply ARM endorsement.)



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.

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

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

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

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

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

Exercises

Exercise 7.1 Suppose that one of the following control signals in the single-cycle ARM processor has a *stuck-at-0 fault*, meaning that the signal is always 0, regardless of its intended value. What instructions would malfunction? Why?

(a) *RegW*

(b) *ALUOp*

(c) *MemW*

Exercise 7.2 Repeat Exercise 7.1, assuming that the signal has a stuck-at-1 fault.

Exercise 7.3 Modify the single-cycle ARM processor to implement one of the following instructions. See Appendix B for a definition of the instructions. Mark up a copy of Figure 7.13 to indicate the changes to the datapath. Name any new control signals. Mark up a copy of Tables 7.2 and 7.3 to show the changes to the Main Decoder and ALU Decoder. Describe any other changes that are required.

(a) `TST`

(b) `LSL`

(c) `CMN`

(d) `ADC`

Exercise 7.4 Repeat Exercise 7.3 for the following ARM instructions.

(a) `EOR`

(b) `LSR`

(c) `TEQ`

(d) `RSB`

Exercise 7.5 ARM includes `LDR` with post-indexing, which updates the base register after completing the load. `LDR Rd,[Rn],Rm` is equivalent to the following two instructions:

```
LDR  Rd,[Rn]
ADD  Rn,Rn,Rm
```

Repeat Exercise 7.3 for `LDR` with post-indexing. Is it possible to add the instruction without modifying the register file?

Exercise 7.6 ARM includes `LDR` with pre-indexing, which updates the base register after completing the load. `LDR Rd,[Rn,Rm]!` is equivalent to the following two instructions:

```
LDR  Rd,[Rn, Rm]
ADD  Rn,Rn,Rm
```

Repeat Exercise 7.3 for `LDR` with pre-indexing. Is it possible to add the instruction without modifying the register file?

Exercise 7.7 Your friend is a crack circuit designer. She has offered to redesign one of the units in the single-cycle ARM processor to have half the delay. Using the delays from Table 7.5, which unit should she work on to obtain the greatest speedup of the overall processor, and what would the cycle time of the improved machine be?

Exercise 7.8 Consider the delays given in Table 7.5. Ben Bitdiddle builds a prefix adder that reduces the ALU delay by 20 ps. If the other element delays stay the same, find the new cycle time of the single-cycle ARM processor and determine how long it takes to execute a benchmark with 100 billion instructions.

Exercise 7.9 Modify the HDL code for the single-cycle ARM processor, given in Section 7.6.1, to handle one of the new instructions from Exercise 7.3. Enhance the testbench, given in Section 7.6.3, to test the new instruction.

Exercise 7.10 Repeat Exercise 7.9 for the new instructions from Exercise 7.4.

Exercise 7.11 Suppose one of the following control signals in the multicycle ARM processor has a stuck-at-0 fault, meaning that the signal is always 0, regardless of its intended value. What instructions would malfunction? Why?

(a) $RegSrc_1$

(b) *AdrSrc*

(c) *NextPC*

Exercise 7.12 Repeat Exercise 7.11, assuming that the signal has a stuck-at-1 fault.

Exercise 7.13 Modify the multicycle ARM processor to implement one of the following instructions. See Appendix B for a definition of the instructions. Mark up a copy of Figure 7.30 to indicate the changes to the datapath. Name any new control signals. Mark up a copy of Figure 7.41 to show the changes to the controller FSM. Describe any other changes that are required.

(a) `ASR`

(b) `TST`

(c) `SBC`

(d) `ROR`

Exercise 7.14 Repeat Exercise 7.13 for the following ARM instructions.

(a) `BL`

(b) `LDR` (with positive or negative immediate offset)

(c) `LDRB` (with positive immediate offset only)

(d) `BIC`

Exercise 7.15 Repeat Exercise 7.5 for the multicycle ARM processor. Show the changes to the multicycle datapath and control FSM. Is it possible to add the instruction without modifying the register file?

Exercise 7.16 Repeat Exercise 7.6 for the multicycle ARM processor. Show the changes to the multicycle datapath and control FSM. Is it possible to add the instruction without modifying the register file?

Exercise 7.17 Repeat Excercise 7.7 for the multicycle ARM processor. Assume the instruction mix of Example 7.5.

Exercise 7.18 Repeat Exercise 7.8 for the multicycle ARM processor. Assume the instruction mix of Example 7.5.

Exercise 7.19 Your friend, the crack circuit designer, has offered to redesign one of the units in the multicycle ARM processor to be much faster. Using the delays from Table 7.5, which unit should she work on to obtain the greatest speedup of the overall processor? How fast should it be? (Making it faster than necessary is a waste of your friend's effort.) What is the cycle time of the improved processor?

Exercise 7.20 Goliath Corp claims to have a patent on a three-ported register file. Rather than fighting Goliath in court, Ben Bitdiddle designs a new register file that has only a single read/write port (like the combined instruction and data memory). Redesign the ARM multicycle datapath and controller to use his new register file.

Exercise 7.21 Suppose the multicycle ARM processor has the component delays given in Table 7.5. Alyssa P. Hacker designs a new register file that has 40% less power but twice as much delay. Should she switch to the slower but lower power register file for her multicycle processor design?

Exercise 7.22 What is the CPI of the redesigned multicycle ARM processor from Exercise 7.20? Use the instruction mix from Example 7.5.

Exercise 7.23 How many cycles are required to run the following program on the multicycle ARM processor? What is the CPI of this program?

```
    MOV  R0, #5             ; result = 5
    MOV  R1, #0             ; R1 = 0
L1
    CMP  R0, R1
    BEQ  DONE               ; if result > 0, loop
    SUB  R0, R0, #1         ; result = result-1
    B    L1
DONE
```

Exercise 7.24 Repeat Exercise 7.23 for the following program.

```
    MOV  R0, #0             ; i = 0
    MOV  R1, #0             ; sum = 0
    MOV  R2, #10            ; R2 = 10

LOOP
    CMP  R2, R0             ; R2 == R0?
    BEQ  L2
    ADD  R1, R1, R0         ; sum = sum + i
    ADD  R0, R0, #1         ; increment i
    B
    LOOP
L2
```

Exercise 7.25 Write HDL code for the multicycle ARM processor. The processor should be compatible with the following top-level module. The `mem` module is used to hold both instructions and data. Test your processor using the testbench from Section 7.6.3.

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

  logic [31:0] ReadData;

  // instantiate processor and shared memory
  arm arm(clk, reset, MemWrite, Adr,
          WriteData, ReadData);
  mem mem(clk, MemWrite, Adr, WriteData, ReadData);
endmodule

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

  logic [31:0] RAM[63:0];
  initial
      $readmemh("memfile.dat",RAM);
```

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

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

Exercise 7.26 Extend your HDL code for the multicycle ARM processor from Exercise 7.25 to handle one of the new instructions from Exercise 7.14. Enhance the testbench to test the new instruction.

Exercise 7.27 Repeat Exercise 7.26 for one of the new instructions from Exercise 7.13.

Exercise 7.28 The pipelined ARM processor is running the following code snippet. Which registers are being written, and which are being read on the fifth cycle? Recall that the pipelined ARM processor has a Hazard Unit.

```
MOV  R1, #42
SUB  R0, R1, #5
LDR  R3, [R0, #18]
STR  R4, [R1, #63]
ORR  R2, R0, R3
```

Exercise 7.29 Repeat Exercise 7.28 for the following ARM code snippet.

```
ADD  R0, R4, R5
SUB  R1, R6, R7
AND  R2, R0, R1
ORR  R3, R2, R5
LSL  R4, R2, R3
```

Exercise 7.30 Using a diagram similar to Figure 7.53, show the forwarding and stalls needed to execute the following instructions on the pipelined ARM processor.

```
ADD  R0, R4, R9
SUB  R0, R0, R2
LDR  R1, [R0, #60]
AND  R2, R1, R0
```

Exercise 7.31 Repeat Exercise 7.30 for the following instructions.

```
ADD  R0, R11, R5
LDR  R2, [R1, #45]
SUB  R5, R0, R2
AND  R5, R2, R5
```

Exercise 7.32 How many cycles are required for the pipelined ARM processor to issue all of the instructions for the program in Exercise 7.24? What is the CPI of the processor on this program?

Exercise 7.33 Repeat Exercise 7.32 for the instructions of the program in Exercise 7.23.

Exercise 7.34 Explain how to extend the pipelined ARM processor to handle the `EOR` instruction.

Exercise 7.35 Explain how to extend the pipelined processor to handle the `CMN` instruction.

Exercise 7.36 Section 7.5.3 points out that the pipelined processor performance might be better if branches take place during the Decode stage rather than the Execute stage. Show how to modify the pipelined processor from Figure 7.58 to branch in the Decode stage. How do the stall, flush, and forwarding signals change? Redo Examples 7.7 and 7.8 to find the new CPI, cycle time, and overall time to execute the program.

Exercise 7.37 Your friend, the crack circuit designer, has offered to redesign one of the units in the pipelined ARM processor to be much faster. Using the delays from Table 7.5, which unit should she work on to obtain the greatest speedup of the overall processor? How fast should it be? (Making it faster than necessary is a waste of your friend's effort.) What is the cycle time of the improved processor?

Exercise 7.38 Consider the delays from Table 7.5. Now suppose that the ALU were 20% faster. Would the cycle time of the pipelined ARM processor change? What if the ALU were 20% slower?

Exercise 7.39 Suppose the ARM pipelined processor is divided into 10 stages of 400 ps each, including sequencing overhead. Assume the instruction mix of Example 7.7. Also assume that 50% of the loads are immediately followed by an instruction that uses the result, requiring six stalls, and that 30% of the branches are mispredicted. The target address of a branch instruction is not computed until the end of the second stage. Calculate the average CPI and execution time of computing 100 billion instructions from the SPECINT2000 benchmark for this 10-stage pipelined processor.

Exercise 7.40 Write HDL code for the pipelined ARM processor. The processor should be compatible with the top-level module from HDL Example 7.13. It should support the seven instructions described in this chapter: `ADD`, `SUB`, `AND`, `ORR` (with register and immediate addressing modes but no shifts), `LDR`, `STR` (with positive immediate offset), and `B`. Test your design using the testbench from HDL Example 7.12.

Exercise 7.41 Design the Hazard Unit shown in Figure 7.58 for the pipelined ARM processor. Use an HDL to implement your design. Sketch the hardware that a synthesis tool might generate from your HDL.

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 7.1 Explain the advantages of pipelined microprocessors?

Question 7.2 If additional pipeline stages allow a processor to go faster, why don't processors have 100 pipeline stages?

Question 7.3 Describe what a hazard is in a microprocessor and explain ways in which it can be resolved. What are the pros and cons of each way?

Question 7.4 Describe the concept of a superscalar processor and its pros and cons?