called B and the least significant N bits are called C. By choosing appropriate values of B, C, and k, the funnel shifter can perform any type of shift or rotate. Explain what these values should be in terms of A, *shamt*, and N for

(a) logical right shift of A by *shamt*.

(b) arithmetic right shift of A by *shamt*.

(c) left shift of A by *shamt*.

(d) right rotate of A by *shamt*.

(e) left rotate of A by *shamt*.

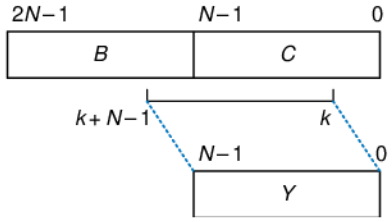


Figure 5.64 Funnel shifter

Exercise 5.18 Find the critical path for the 4 × 4 multiplier from Figure 5.18 in terms of an AND gate delay (t_{AND}) and a full adder delay (t_{FA}). What is the delay of an $N \times N$ multiplier built in the same way?

Exercise 5.19 Design a multiplier that handles two's complement numbers.

Exercise 5.20 A *sign extension unit* extends a two's complement number from M to N ($N > M$) bits by copying the most significant bit of the input into the upper bits of the output (see Section 1.4.6). It receives an M-bit input, A, and produces an N-bit output, Y. Sketch a circuit for a sign extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

Exercise 5.21 A *zero extension unit* extends an unsigned number from M to N bits ($N > M$) by putting zeros in the upper bits of the output. Sketch a circuit for a zero extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

Exercise 5.22 Compute $111001.000_2/001100.000_2$ in binary using the standard division algorithm from elementary school. Show your work.

Exercise 5.23 What is the range of numbers that can be represented by the following number systems?

(a) 24-bit unsigned fixed-point numbers with 12 integer bits and 12 fraction bits

(b) 24-bit sign and magnitude fixed-point numbers with 12 integer bits and 12 fraction bits

(c) 24-bit two's complement fixed-point numbers with 12 integer bits and 12 fraction bits

Exercise 5.24 Express the following base 10 numbers in 16-bit fixed-point sign/magnitude format with eight integer bits and eight fraction bits. Express your answer in hexadecimal.

(a) −13.5625

(b) 42.3125

(c) −17.15625

Exercise 5.25 Express the base 10 numbers in Exercise 5.24 in 16-bit fixed-point two's complement format with eight integer bits and eight fraction bits. Express your answer in hexadecimal.

Exercise 5.26 Express the base 10 numbers in Exercise 5.24 in IEEE 754 single-precision floating-point format. Express your answer in hexadecimal.

Exercise 5.27 Convert the following two's complement binary fixed-point numbers to base 10.

(a) 0101.1000

(b) 1111.1111

(c) 1000.0000

Exercise 5.28 When adding two floating-point numbers, the number with the smaller exponent is shifted. Why is this? Explain in words and give an example to justify your explanation.

Exercise 5.29 Add the following IEEE 754 single-precision floating-point numbers.

(a) C0D20004 + 72407020

(b) C0D20004 + 40DC0004

(c) (5FBE4000 + 3FF80000) + DFDE4000
(Why is the result counterintuitive? Explain.)

Exercise 5.30 Expand the steps in section 5.3.2 for performing floating-point addition to work for negative as well as positive floating-point numbers.

Exercise 5.31 Consider IEEE 754 single-precision floating-point numbers.

(a) How many numbers can be represented by IEEE 754 single-precision floating-point format? You need not count $\pm\infty$ or NaN.

(b) How many additional numbers could be represented if $\pm\infty$ and NaN were not represented?

(c) Explain why $\pm\infty$ and NaN are given special representations.

Exercise 5.32 Consider the following decimal numbers: 245 and 0.0625.

(a) Write the two numbers using single-precision floating-point notation. Give your answers in hexadecimal.

(b) Perform a magnitude comparison of the two 32-bit numbers from part (a). In other words, interpret the two 32-bit numbers as two's complement numbers and compare them. Does the integer comparison give the correct result?

(c) You decide to come up with a new single-precision floating-point notation. Everything is the same as the IEEE 754 single-precision floating-point standard, except that you represent the exponent using two's complement instead of a bias. Write the two numbers using your new standard. Give your answers in hexadecimal.

(e) Does integer comparison work with your new floating-point notation from part (d)?

(f) Why is it convenient for integer comparison to work with floating-point numbers?

Exercise 5.33 Design a single-precision floating-point adder using your favorite HDL. Before coding the design in an HDL, sketch a schematic of your design. Simulate and test your adder to prove to a skeptic that it functions correctly. You may consider positive numbers only and use round toward zero (truncate). You may also ignore the special cases given in Table 5.2.

Exercise 5.34 In this problem, you will explore the design of a 32-bit floating-point multiplier. The multiplier has two 32-bit floating-point inputs and produces a 32-bit floating-point output. You may consider positive numbers only

and use round toward zero (truncate). You may also ignore the special cases given in Table 5.2.

(a) Write the steps necessary to perform 32-bit floating-point multiplication.

(b) Sketch the schematic of a 32-bit floating-point multiplier.

(c) Design a 32-bit floating-point multiplier in an HDL. Simulate and test your multiplier to prove to a skeptic that it functions correctly.

Exercise 5.35 In this problem, you will explore the design of a 32-bit prefix adder.

(a) Sketch a schematic of your design.

(b) Design the 32-bit prefix adder in an HDL. Simulate and test your adder to prove that it functions correctly.

(c) What is the delay of your 32-bit prefix adder from part (a)? Assume that each two-input gate delay is 100 ps.

(d) Design a pipelined version of the 32-bit prefix adder. Sketch the schematic of your design. How fast can your pipelined prefix adder run? Make the design run as fast as possible.

(e) Design the pipelined 32-bit prefix adder in an HDL.

Exercise 5.36 An incrementer adds 1 to an N-bit number. Build an 8-bit incrementer using half adders.

Exercise 5.37 Build a 32-bit synchronous *Up/Down counter*. The inputs are *Reset* and *Up*. When *Reset* is 1, the outputs are all 0. Otherwise, when $Up = 1$, the circuit counts up, and when $Up = 0$, the circuit counts down.

Exercise 5.38 Design a 32-bit counter that adds 4 at each clock edge. The counter has reset and clock inputs. Upon reset, the counter output is all 0.

Exercise 5.39 Modify the counter from Exercise 5.38 such that the counter will either increment by 4 or load a new 32-bit value, D, on each clock edge, depending on a control signal, *PCSrc*. When $PCSrc = 1$, the counter loads the new value D.

Exercise 5.40 An N-bit *Johnson counter* consists of an N-bit shift register with a reset signal. The output of the shift register (S_{out}) is inverted and fed back to the input (S_{in}). When the counter is reset, all of the bits are cleared to 0.

(a) Show the sequence of outputs, $Q_{3:0}$, produced by a 4-bit Johnson counter starting immediately after the counter is reset.

(b) How many cycles elapse until an N-bit Johnson counter repeats its sequence? Explain.

(c) Design a decimal counter using a 5-bit Johnson counter, ten AND gates, and inverters. The decimal counter has a clock, a reset, and ten one-hot outputs, $Y_{9:0}$. When the counter is reset, Y_0 is asserted. On each subsequent cycle, the next output should be asserted. After ten cycles, the counter should repeat. Sketch a schematic of the decimal counter.

(d) What advantages might a Johnson counter have over a conventional counter?

Exercise 5.41 Write the HDL for a 4-bit scannable flip-flop like the one shown in Figure 5.37. Simulate and test your HDL module to prove that it functions correctly.

Exercise 5.42 The English language has a good deal of redundancy that allows us to reconstruct garbled transmissions. Binary data can also be transmitted in redundant form to allow error correction. For example, the number 0 could be coded as 00000 and the number 1 could be coded as 11111. The value could then be sent over a noisy channel that might flip up to two of the bits. The receiver could reconstruct the original data because a 0 will have at least three of the five received bits as 0's; similarly a 1 will have at least three 1's.

(a) Propose an encoding to send 00, 01, 10, or 11 encoded using five bits of information such that all errors that corrupt one bit of the encoded data can be corrected. Hint: the encodings 00000 and 11111 for 00 and 11, respectively, will not work.

(b) Design a circuit that receives your five-bit encoded data and decodes it to 00, 01, 10, or 11, even if one bit of the transmitted data has been changed.

(c) Suppose you wanted to change to an alternative 5-bit encoding. How might you implement your design to make it easy to change the encoding without having to use different hardware?

Exercise 5.43 Flash EEPROM, simply called Flash memory, is a fairly recent invention that has revolutionized consumer electronics. Research and explain how Flash memory works. Use a diagram illustrating the floating gate. Describe how a bit in the memory is programmed. Properly cite your sources.

Exercise 5.44 The extraterrestrial life project team has just discovered aliens living on the bottom of Mono Lake. They need to construct a circuit to classify the aliens by potential planet of origin based on measured features

available from the NASA probe: greenness, brownness, sliminess, and ugliness. Careful consultation with xenobiologists leads to the following conclusions:

- If the alien is green and slimy or ugly, brown, and slimy, it might be from Mars.
- If the critter is ugly, brown, and slimy, or green and neither ugly nor slimy, it might be from Venus.
- If the beastie is brown and neither ugly nor slimy or is green and slimy, it might be from Jupiter.

Note that this is an inexact science; for example, a life form which is mottled green and brown and is slimy but not ugly might be from either Mars or Jupiter.

(a) Program a $4 \times 4 \times 3$ PLA to identify the alien. You may use dot notation.

(b) Program a 16×3 ROM to identify the alien. You may use dot notation.

(c) Implement your design in an HDL.

Exercise 5.45 Implement the following functions using a single 16×3 ROM. Use dot notation to indicate the ROM contents.

(a) $X = AB + B\overline{C}D + \overline{A}\overline{B}$

(b) $Y = AB + BD'$

(c) $Z = A + B + C + D$

Exercise 5.46 Implement the functions from Exercise 5.45 using an $4 \times 8 \times 3$ PLA. You may use dot notation.

Exercise 5.47 Specify the size of a ROM that you could use to program each of the following combinational circuits. Is using a ROM to implement these functions a good design choice? Explain why or why not.

(a) a 16-bit adder/subtractor with C_{in} and C_{out}

(b) an 8×8 multiplier

(c) a 16-bit priority encoder (see Exercise 2.25)

Exercise 5.48 Consider the ROM circuits in Figure 5.65. For each row, can the circuit in column I be replaced by an equivalent circuit in column II by proper programming of the latter's ROM?

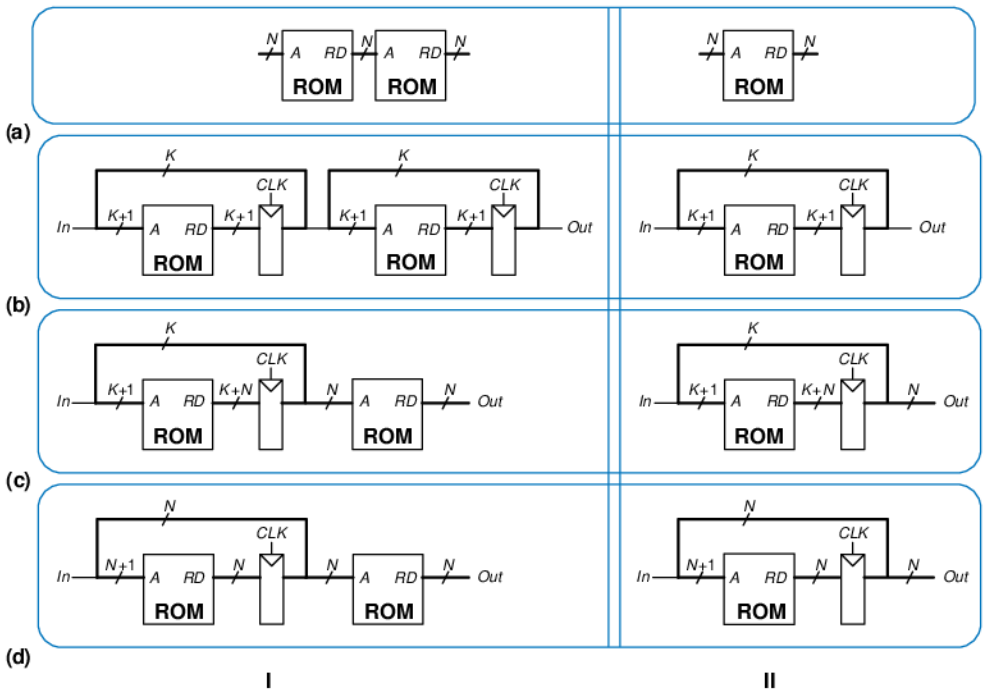


Figure 5.65 ROM circuits

Exercise 5.49 Give an example of a nine-input function that can be performed using only one Spartan FPGA CLB. Give an example of an eight-input function that cannot be performed using only one CLB.

Exercise 5.50 How many Spartan FPGA CLBs are required to perform each of the following functions? Show how to configure one or more CLBs to perform the function. You should be able to do this by inspection, without performing logic synthesis.

(a) The combinational function from Exercise 2.7(c).

(b) The combinational function from Exercise 2.9(c).

(c) The two-output function from Exercise 2.15.

(d) The function from Exercise 2.24.

(e) A four-input priority encoder (see Exercise 2.25).

(f) An eight-input priority encoder (see Exercise 2.25).

(g) A 3:8 decoder.

(h) A 4-bit carry propagate adder (with no carry in or out).

(i) The FSM from Exercise 3.19.

(j) The Gray code counter from Exercise 3.24.

Exercise 5.51 Consider the Spartan CLB shown in Figure 5.58. It has the following specifications: $t_{pd} = t_{cd} = 2.7$ ns per CLB; $t_{\text{setup}} = 3.9$ ns, $t_{\text{hold}} = 0$ ns, and $t_{pcq} = 2.8$ ns for all flip-flops.

(a) What is the minimum number of Spartan CLBs required to implement the FSM of Figure 3.26?

(b) Without clock skew, what is the fastest clock frequency at which this FSM will run reliably?

(c) With 5 ns of clock skew, what is the fastest frequency at which the FSM will run reliably?

Exercise 5.52 You would like to use an FPGA to implement an M&M sorter with a color sensor and motors to put red candy in one jar and green candy in another. The design is to be implemented as an FSM using a Spartan XC3S200 FPGA, a chip from the Spartan 3 series family. It is considerably faster than the original Spartan FPGA. According to the data sheet, the FPGA has timing characteristics shown in Table 5.5. Assume that the design is small enough that wire delay is negligible.

Table 5.5 Spartan 3 XC3S200 timing

Name	Value (ns)
t_{pcq}	0.72
t_{setup}	0.53
t_{hold}	0
t_{pd} (per CLB)	0.61
t_{skew}	0

You would like your FSM to run at 100 MHz. What is the maximum number of CLBs on the critical path? What is the fastest speed at which the FSM will run?

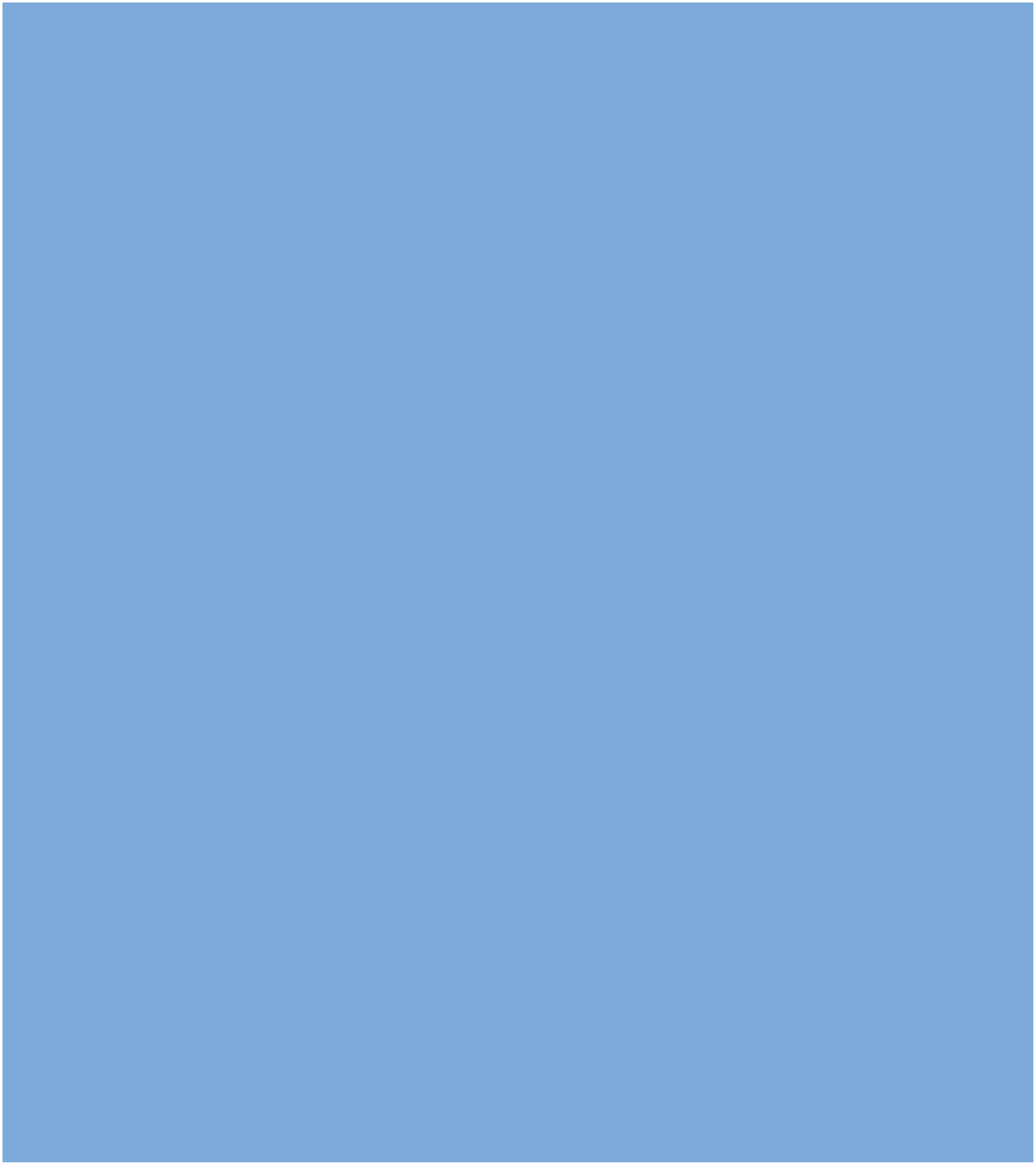
Interview Questions

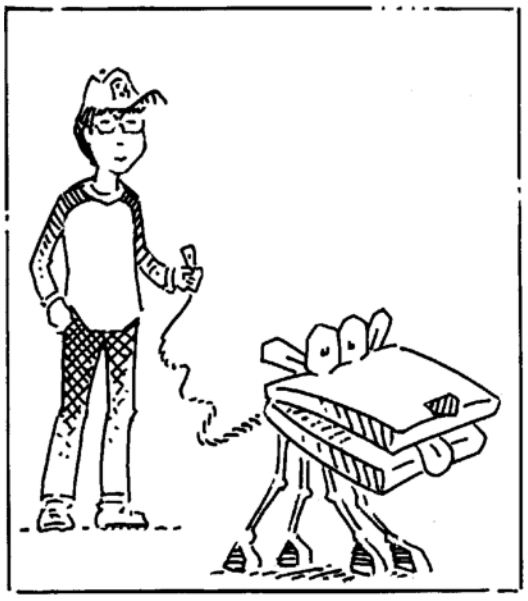
The following exercises present questions that have been asked at interviews for digital design jobs.

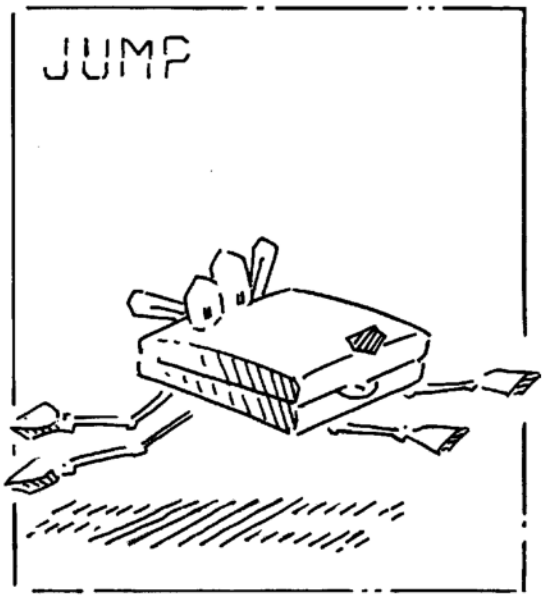
Question 5.1 What is the largest possible result of multiplying two unsigned N-bit numbers?

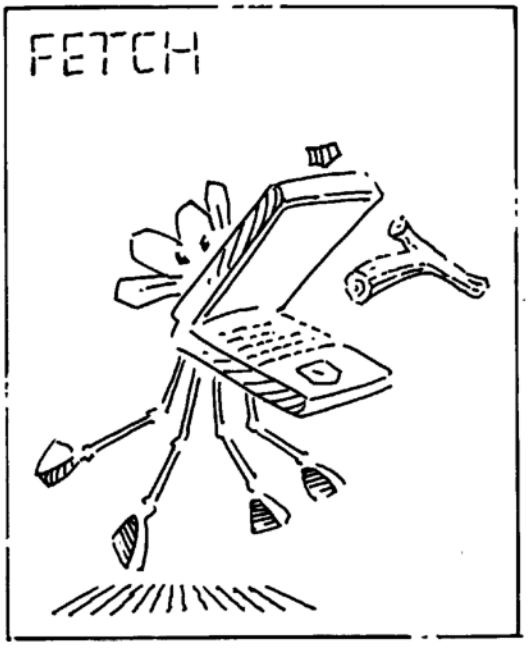
Question 5.2 *Binary coded decimal* (*BCD*) representation uses four bits to encode each decimal digit. For example 42_{10} is represented as 01000010_{BCD}. Explain in words why processors might use BCD representation.

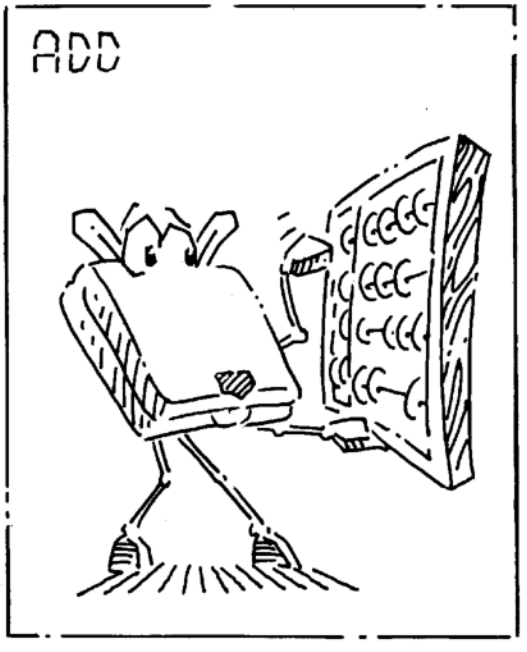
Question 5.3 Design hardware to add two 8-bit unsigned BCD numbers (see Question 5.2). Sketch a schematic for your design, and write an HDL module for the BCD adder. The inputs are A, B, and C_{in}, and the outputs are S and C_{out}. C_{in} and C_{out} are 1-bit carries, and A, B, and S are 8-bit BCD numbers.






JUMP


FETCH


ADD

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 (see Figure 1.1). 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 IA-32, 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 jump. 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. 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 jump, that operate on memory or registers. Once you have learned one instruction set, understanding others is fairly straightforward.

What is the best architecture to study when first learning the subject?

Commercially successful architectures such as IA-32 are satisfying to study because you can use them to write programs on real computers. Unfortunately, many of these architectures are full of warts and idiosyncrasies accumulated over years of haphazard development by different engineering teams, making the architectures difficult to understand and implement.

Many textbooks teach imaginary architectures that are simplified to illustrate the key concepts.

We follow the lead of David Patterson and John Hennessy in their text, *Computer Organization and Design*, by focusing on the MIPS architecture. Hundreds of millions of MIPS microprocessors have shipped, so the architecture is commercially very important. Yet it is a clean architecture with little odd behavior. At the end of this chapter, we briefly visit the IA-32 architecture to compare and contrast it with MIPS.

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) sell various microprocessors belonging to the same IA-32 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 MIPS architecture that was first developed by John Hennessy and his colleagues at Stanford in the 1980s. MIPS processors are used by, among others, Silicon Graphics, Nintendo, and Cisco. We start by introducing the basic instructions, operand locations, and machine language formats. We then introduce more instructions used in common programming constructs, such as branches, loops, array manipulations, and procedure calls.

Throughout the chapter, we motivate the design of the MIPS architecture using four principles articulated by Patterson and Hennessy: (1) simplicity favors regularity; (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 MIPS instruction operands: registers, memory, and constants.

We assume 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.)

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 program 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 MIPS assembly language. Note that statements in a C program end with a semicolon.

Code Example 6.1 ADDITION

High-Level Code	MIPS Assembly Code
`a = b + c;`	`add a, b, c`

Code Example 6.2 SUBTRACTION

High-Level Code	MIPS Assembly Code
`a = b - c;`	`sub 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 μμνΕσκεστηαι, 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 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: Simplicity favors regularity.

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 MIPS 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 assembly language, only single-line comments are used. They begin with `#` 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

Code Example 6.3 MORE COMPLEX CODE

High-Level Code	MIPS Assembly Code

`a = b + c - d;  // single-line comment`
`/* multiple-line`
`comment */`
	`sub t, c, d  # t = c - d`
	`add a, b, t  # a = b + t`

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 MIPS 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, MIPS is a *reduced instruction set computer* (*RISC*) architecture. Architectures with many complex instructions, such as Intel's IA-32 architecture, are *complex instruction set computers* (*CISC*). For example, IA-32 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, 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. MIPS is called a 32-bit architecture because it operates on 32-bit data. (The MIPS architecture has been extended to 64 bits in commercial products, but we will consider only the 32-bit form 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 MIPS architecture uses 32 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 set of registers (for example, 32) is faster than reading it from 1000 registers or a large memory. A small register file is typically built from a small SRAM array (see Section 5.5.3). The SRAM array uses a small decoder and bitlines connected to relatively few memory cells, so it has a shorter critical path than a large memory does.

Code Example 6.4 shows the `add` instruction with register operands. MIPS register names are preceded by the `$` sign. The variables `a`, `b`, and `c` are arbitrarily placed in `$s0`, `$s1`, and `$s2`. The name `$s1` is pronounced "register s1" or "dollar s1". The instruction adds the 32-bit values contained in `$s1` (`b`) and `$s2` (`c`) and writes the 32-bit result to `$s0` (`a`).

MIPS generally stores variables in 18 of the 32 registers: `$s0` – `$s7`, and `$t0` – `$t9`. Register names beginning with `$s` are called *saved* registers. Following MIPS convention, these registers store variables such as `a`, `b`, and `c`. Saved registers have special connotations when they are used with procedure calls (see Section 6.4.6). Register names beginning with `$t` are called *temporary* registers. They are used for storing temporary variables. Code Example 6.5 shows MIPS assembly code using a temporary register, `$t0`, to store the intermediate calculation of `c – d`.

Code Example 6.4 REGISTER OPERANDS

High-Level Code	MIPS Assembly Code
`a = b + c;`	`# $s0 = a, $s1 = b, $s2 = c` `add $s0, $s1, $s2  # a = b + c`

Code Example 6.5 TEMPORARY REGISTERS

High-Level Code	MIPS Assembly Code

`a = b + c - d;`	`# $s0 = a, $s1 = b, $s2 = c, $s3 = d`
	
	`sub $t0, $s2, $s3  # t = c - d`
	`add $s0, $s1, $t0  # a = b + t`

Example 6.1 TRANSLATING HIGH-LEVEL CODE TO ASSEMBLY LANGUAGE

Translate the following high-level code into assembly language. Assume variables `a-c` are held in registers `$s0-$s2` and `f-j` are in `$s3-$s7`.

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

Solution: The program uses four assembly language instructions.

```
# MIPS assembly code
# $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 = g, $s5 = h,
# $s6 = i, $s7 = j
  sub $s0, $s1, $s2   # a = b - c
  add $t0, $s4, $s5   # $t0 = g + h
  add $t1, $s6, $s7   # $t1 = i + j
  sub $s3, $t0, $t1   # f = (g + h) - (i + j)
```

The Register Set

The MIPS architecture defines 32 registers. Each register has a name and a number ranging from 0 to 31. Table 6.1 lists the name, number, and use for each register. `$0` always contains the value 0 because this constant is so frequently used in computer programs. We have also discussed the `$s` and `$t` registers. The remaining registers will be described throughout this chapter.

Table 6.1 MIPS register set

Name	Number	Use
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2–3	procedure return values
`$a0-$a3`	4–7	procedure arguments
`$t0-$t7`	8–15	temporary variables
`$s0-$s7`	16–23	saved variables
`$t8-$t9`	24–25	temporary variables
`$k0-$k1`	26–27	operating system (OS) temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	procedure return address

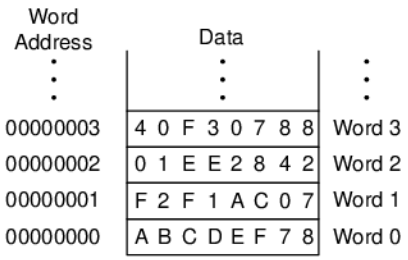


Figure 6.1 Word-addressable memory

Memory

If registers were the only storage space for operands, we would be confined to simple programs with no more than 32 variables. However, data can also be stored in memory. When compared to the register file, memory has many data locations, but accessing it takes a longer amount of time. Whereas the register file is small and fast, memory is large and slow. For this reason, commonly used variables are kept in registers. By using a combination of memory and registers, a program can access a large amount of data fairly quickly. As described in Section 5.5, memories are organized as an array of data words. The MIPS architecture uses 32-bit memory addresses and 32-bit data words.

MIPS uses a byte-addressable memory. That is, each byte in memory has a unique address. However, for explanation purposes only, we first introduce a word-addressable memory, and afterward describe the MIPS byte-addressable memory.

Figure 6.1 shows a memory array that is *word-addressable*. That is, each 32-bit data word has a unique 32-bit address. Both the 32-bit word address and the 32-bit data value are written in hexadecimal in Figure 6.1. For example, data 0xF2F1AC07 is stored at memory address 1. Hexadecimal constants are written with the prefix 0x. By convention, memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top.

MIPS uses the *load word* instruction, `lw`, to read a data word from memory into a register. Code Example 6.6 loads memory word 1 into `$s3`.

The `lw` instruction specifies the *effective address* in memory as the sum of a *base address* and an *offset*. The base address (written in parentheses in the instruction) is a register. The offset is a constant (written before the parentheses). In Code Example 6.6, the base address

Code Example 6.6 READING WORD-ADDRESSABLE MEMORY

Assembly Code

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  lw $s3, 1($0)     # read memory word 1 into $s3
```

Code Example 6.7 WRITING WORD-ADDRESSABLE MEMORY

Assembly Code

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  sw   $s7, 5($0)     # write $s7 to memory word 5
```

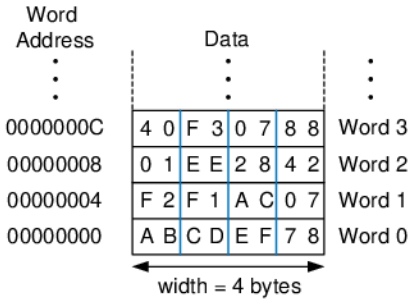


Figure 6.2 **Byte-addressable memory**

is `$0`, which holds the value 0, and the offset is 1, so the `lw` instruction reads from memory address (`$0` + 1) = 1. After the load word instruction (`lw`) is executed, `$s3` holds the value 0xF2F1AC07, which is the data value stored at memory address 1 in Figure 6.1.

Similarly, MIPS uses the *store word* instruction, `sw`, to write a data word from a register into memory. Code Example 6.7 writes the contents of register `$s7` into memory word 5. These examples have used `$0` as the base address for simplicity, but remember that any register can be used to supply the base address.

The previous two code examples have shown a computer architecture with a word-addressable memory. The MIPS memory model, however, is byte-addressable, *not* word-addressable. Each data byte has a unique address. A 32-bit word consists of four 8-bit bytes. So each word address is a multiple of 4, as shown in Figure 6.2. Again, both the 32-bit word address and the data value are given in hexadecimal.

Code Example 6.8 shows how to read and write words in the MIPS byte-addressable memory. The word address is four times the word number. The MIPS assembly code reads words 0, 2, and 3 and writes words 1, 8, and 100. The offset can be written in decimal or hexadecimal.

The MIPS architecture also provides the `lb` and `sb` instructions that load and store single bytes in memory rather than words. They are similar to `lw` and `sw` and will be discussed further in Section 6.4.5.

Byte-addressable memories are organized in a *big-endian* or *little-endian* fashion, as shown in Figure 6.3. In both formats, the *most significant byte* (*MSB*) is on the left and the *least significant byte* (*LSB*) is on the right. In big-endian machines, bytes are numbered starting with 0

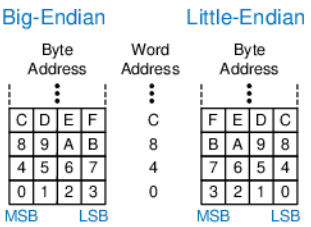


Figure 6.3 **Big- and little-endian memory addressing**

Code Example 6.8 ACCESSING BYTE-ADDRESSABLE MEMORY

MIPS Assembly Code

```
lw $s0, 0($0)       # read data word 0 (0xABCDEF78) into $s0
lw $s1, 8($0)       # read data word 2 (0x01EE2842) into $s1
lw $s2, 0xC($0)     # read data word 3 (0x40F30788) into $s2
sw $s3, 4($0)       # write $s3 to data word 1
sw $s4, 0x20($0)    # write $s4 to data word 8
sw $s5, 400($0)     # write $s5 to data word 100
```

at the big (most significant) end. In little-endian machines, bytes are numbered starting with 0 at the little (least significant) end. Word addresses are the same in both formats and refer to the same four bytes. Only the addresses of bytes within a word differ.

Example 6.2 BIG- AND LITTLE-ENDIAN MEMORY

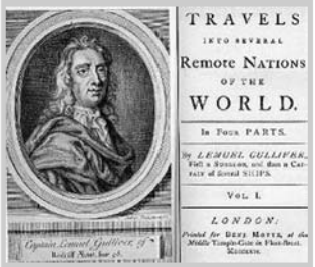
Suppose that `$s0` initially contains 0x23456789. After the following program is run on a big-endian system, what value does `$s0` contain? In a little-endian system? `lb $s0, 1($0)` loads the data at byte address `(1 + $0) = 1` into the least significant byte of `$s0`. `lb` is discussed in detail in Section 6.4.5.

```
sw $s0, 0($0)
lb $s0, 1($0)
```

Solution: Figure 6.4 shows how big- and little-endian machines store the value 0x23456789 in memory word 0. After the load byte instruction, `lb $s0, 1($0)`, `$s0` would contain 0x00000045 on a big-endian system and 0x00000067 on a little-endian system.

Figure 6.4 Big-endian and little-endian data storage

IBM's PowerPC (formerly found in Macintosh computers) uses big-endian addressing. Intel's IA-32 architecture (found in PCs) uses little-endian addressing. Some MIPS processors are little-endian, and some are big-endian.[1] 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 will 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.

The 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 The Brotherton Collection, IEEDS University Library.)

[1] SPIM, the MIPS simulator that comes with this text, uses the endianness of the machine it is run on. For example, when using SPIM on an Intel IA-32 machine, the memory is little-endian. With an older Macintosh or Sun SPARC machine, memory is big-endian.

In the MIPS architecture, word addresses for `lw` and `sw` must be *word aligned*. That is, the address must be divisible by 4. Thus, the instruction `lw $s0, 7($0)` is an illegal instruction. Some architectures, such as IA-32, allow non-word-aligned data reads and writes, but MIPS requires strict alignment for simplicity. Of course, byte addresses for load byte and store byte, `lb` and `sb`, need not be word aligned.

Constants/Immediates

Load word and store word, `lw` and `sw`, also illustrate the use of *constants* in MIPS instructions. These constants are called *immediates*, because their values are immediately available from the instruction and do not require a register or memory access. Add immediate, `addi`, is another common MIPS instruction that uses an immediate operand. `addi` adds the immediate specified in the instruction to a value in a register, as shown in Code Example 6.9.

The immediate specified in an instruction is a 16-bit two's complement number in the range [−32768, 32767]. Subtraction is equivalent to adding a negative number, so, in the interest of simplicity, there is no `subi` instruction in the MIPS architecture.

Recall that the `add` and `sub` instructions use three register operands. But the `lw`, `sw`, and `addi` instructions use two register operands and a constant. Because the instruction formats differ, `lw` and `sw` instructions violate design principle 1: simplicity favors regularity. However, this issue allows us to introduce the last design principle:

Code Example 6.9 IMMEDIATE OPERANDS

High-Level Code	MIPS Assembly Code

`a = a + 4;`
`b = a − 12;`
	`# $s0 = a, $s1 = b`
	`addi $s0, $s0, 4 # a = a + 4`
	`addi $s1, $s0, −12 # b = a − 12`

Design Principle 4: Good design demands good compromises.

A single instruction format would be simple but not flexible. The MIPS instruction set makes the compromise of supporting three instruction formats. One format, used for instructions such as `add` and `sub`, has three register operands. Another, used for instructions such as `lw` and `addi`, has two register operands and a 16-bit immediate. A third, to be discussed later, has a 26-bit immediate and no registers. The next section discusses the three MIPS instruction formats and shows how they are encoded into binary.

6.3 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*.

MIPS uses 32-bit instructions. Again, simplicity favors regularity, 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 too much complexity. Simplicity would also encourage a single instruction format, but, as already mentioned, that is too restrictive. MIPS makes the compromise of defining three instruction formats: R-type, I-type, and J-type. This small number of formats allows for some regularity among all the types, and thus simpler hardware, while also accommodating different instruction needs, such as the need to encode large constants in the instruction. *R-type* instructions operate on three registers. *I-type* instructions operate on two registers and a 16-bit immediate. *J-type* (jump) instructions operate on one 26-bit immediate. We introduce all three formats in this section but leave the discussion of J-type instructions for Section 6.4.2.

6.3.1 R-type Instructions

The name R-type is short for *register-type*. R-type instructions use three registers as operands: two as sources, and one as a destination. Figure 6.5 shows the R-type machine instruction format. The 32-bit instruction has six fields: `op`, `rs`, `rt`, `rd`, `shamt`, and `funct`. Each field is five or six bits, as indicated.

The operation the instruction performs is encoded in the two fields highlighted in blue: `op` (also called `opcode` or operation code) and `funct` (also called the function). All R-type instructions have an `opcode` of 0. The specific R-type operation is determined by the `funct` field. For example, the `opcode` and `funct` fields for the `add` instruction are 0 (000000_2) and 32 (100000_2), respectively. Similarly, the `sub` instruction has an `opcode` and `funct` field of 0 and 34.

The operands are encoded in the three fields: `rs`, `rt`, and `rd`. The first two registers, `rs` and `rt`, are the source registers; `rd` is the destination

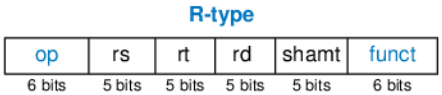


Figure 6.5 R-type machine instruction format

rs is short for "register source." rt comes after rs alphabetically and usually indicates the second register source.

register. The fields contain the register numbers that were given in Table 6.1. For example, $s0 is register 16.

The fifth field, shamt, is used only in shift operations. In those instructions, the binary value stored in the 5-bit shamt field indicates the amount to shift. For all other R-type instructions, shamt is 0.

Figure 6.6 shows the machine code for the R-type instructions add and sub. Notice that the destination is the first register in an assembly language instruction, but it is the third register field (rd) in the machine language instruction. For example, the assembly instruction add $s0, $s1, $s2 has rs = $s1 (17), rt = $s2 (18), and rd = $s0 (16).

Tables B.1 and B.2 in Appendix B define the opcode values for all MIPS instructions and the funct field values for R-type instructions.

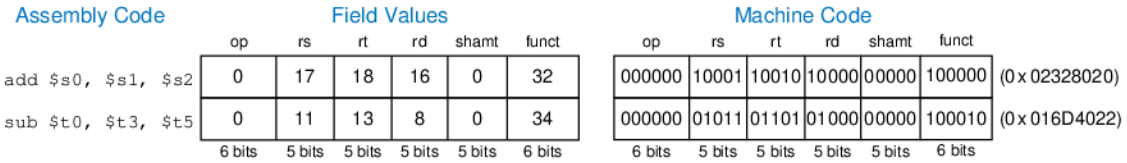


Figure 6.6 Machine code for R-type instructions

Example 6.3 TRANSLATING ASSEMBLY LANGUAGE TO MACHINE LANGUAGE

Translate the following assembly language statement into machine language.

```
add $t0, $s4, $s5
```

Solution: According to Table 6.1, $t0, $s4, and $s5 are registers 8, 20, and 21. According to Tables B.1 and B.2, add has an opcode of 0 and a funct code of 32. Thus, the fields and machine code are given in Figure 6.7. The easiest way to write the machine language in hexadecimal is to first write it in binary, then look at consecutive groups of four bits, which correspond to hexadecimal digits (indicated in blue). Hence, the machine language instruction is 0x02954020.

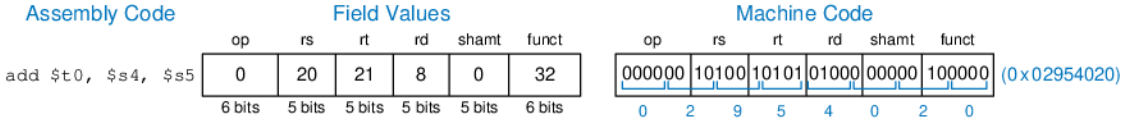


Figure 6.7 Machine code for the R-type instruction of Example 6.3

6.3.2 I-Type Instructions

The name I-type is short for *immediate-type*. I-type instructions use two register operands and one immediate operand. Figure 6.8 shows the I-type machine instruction format. The 32-bit instruction has four fields: `op`, `rs`, `rt`, and `imm`. The first three fields, `op`, `rs`, and `rt`, are like those of R-type instructions. The `imm` field holds the 16-bit immediate.

The operation is determined solely by the `opcode`, highlighted in blue. The operands are specified in the three fields, `rs`, `rt`, and `imm`. `rs` and `imm` are always used as source operands. `rt` is used as a destination for some instructions (such as `addi` and `lw`) but as another source for others (such as `sw`).

Figure 6.9 shows several examples of encoding I-type instructions. Recall that negative immediate values are represented using 16-bit two's complement notation. `rt` is listed first in the assembly language instruction when it is used as a destination, but it is the second register field in the machine language instruction.

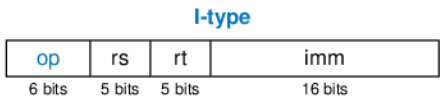


Figure 6.8 I-type instruction format

Assembly Code	op	rs	rt	imm	op	rs	rt	imm	Machine Code
`addi $s0, $s1, 5`	8	17	16	5	001000	10001	10000	0000 0000 0000 0101	(0x22300005)
`addi $t0, $s3, -12`	8	19	8	–12	001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
`lw $t2, 32($0)`	35	0	10	32	100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
`sw $s1, 4($t1)`	43	9	17	4	101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
	6 bits	5 bits	5 bits	16 bits	6 bits	5 bits	5 bits	16 bits	

Field Values | Machine Code

Figure 6.9 Machine code for I-type instructions

Example 6.4 TRANSLATING I-TYPE ASSEMBLY INSTRUCTIONS INTO MACHINE CODE

Translate the following I-type instruction into machine code.

```
lw $s3, -24($s4)
```

Solution: According to Table 6.1, `$s3` and `$s4` are registers 19 and 20, respectively. Table B.1 indicates that `lw` has an `opcode` of 35. `rs` specifies the base address, `$s4`, and `rt` specifies the destination register, `$s3`. The immediate, `imm`, encodes the 16-bit offset, `-24`. Thus, the fields and machine code are given in Figure 6.10.

Figure 6.10 Machine code for the I-type instruction

I-type instructions have a 16-bit immediate field, but the immediates are used in 32-bit operations. For example, `lw` adds a 16-bit offset to a 32-bit base register. What should go in the upper half of the 32 bits? For positive immediates, the upper half should be all 0's, but for negative immediates, the upper half should be all 1's. Recall from Section 1.4.6 that this is called *sign extension*. An N-bit two's complement number is sign-extended to an M-bit number ($M > N$) by copying the sign bit (most significant bit) of the N-bit number into all of the upper bits of the M-bit number. Sign-extending a two's complement number does not change its value.

Most MIPS instructions sign-extend the immediate. For example, `addi`, `lw`, and `sw` do sign extension to support both positive and negative immediates. An exception to this rule is that logical operations (`andi`, `ori`, `xori`) place 0's in the upper half; this is called *zero extension* rather than sign extension. Logical operations are discussed further in Section 6.4.1.

6.3.3 J-type Instructions

The name J-type is short for *jump-type*. This format is used only with jump instructions (see Section 6.4.2). This instruction format uses a single 26-bit address operand, `addr`, as shown in Figure 6.11. Like other formats, J-type instructions begin with a 6-bit `opcode`. The remaining bits are used to specify an address, `addr`. Further discussion and machine code examples of J-type instructions are given in Sections 6.4.2 and 6.5.

6.3.4 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 formats start with a 6-bit `opcode` field. Thus, the best place to begin is to look at the `opcode`. If it is 0, the instruction is R-type; otherwise it is I-type or J-type.

Figure 6.11 J-type instruction format

Example 6.5 TRANSLATING MACHINE LANGUAGE TO ASSEMBLY LANGUAGE

Translate the following machine language code into assembly language.

```
0x2237FFF1
0x02F34022
```

Solution: First, we represent each instruction in binary and look at the six most significant bits to find the `opcode` for each instruction, as shown in Figure 6.12. The `opcode` determines how to interpret the rest of the bits. The `opcodes` are 001000_2 (8_{10}) and 000000_2 (0_{10}), indicating an `addi` and R-type instruction, respectively. The `funct` field of the R-type instruction is 100010_2 (34_{10}), indicating that it is a `sub` instruction. Figure 6.12 shows the assembly code equivalent of the two machine instructions.

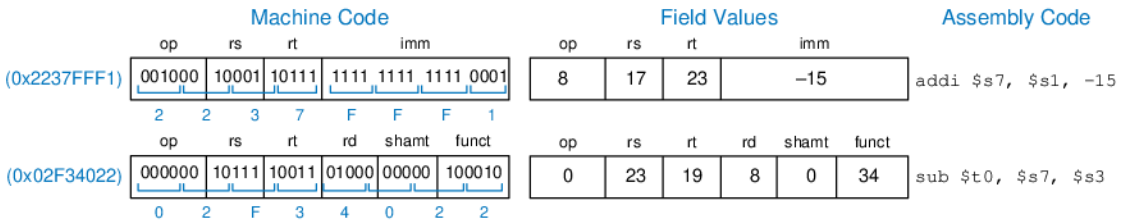


Figure 6.12 Machine code to assembly code translation

6.3.5 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 program does not require large amounts of time and effort to reconfigure or rewire hardware; it only requires writing the new program to memory. Instead of 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 simplified to a series of memory reads and instruction executions.

Figure 6.13 shows how machine instructions are stored in memory. In MIPS programs, the instructions are normally stored starting at address 0x00400000. Remember that MIPS memory is byte addressable, so 32-bit (4-byte) instruction addresses advance by 4 bytes, not 1.

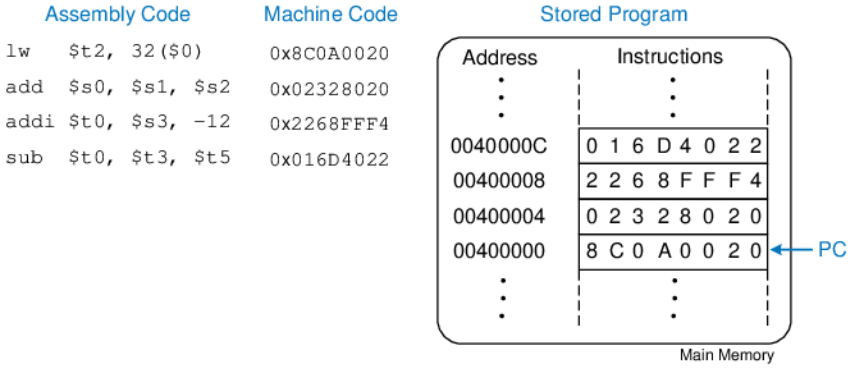


Figure 6.13 Stored program

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). The PC is separate from the 32 registers shown previously in Table 6.1.

To execute the code in Figure 6.13, the operating system sets the PC to address 0x00400000. The processor reads the instruction at that memory address and executes the instruction, 0x8C0A0020. The processor then increments the PC by 4, to 0x00400004, fetches and executes that instruction, and repeats.

The *architectural state* of a microprocessor holds the state of a program. For MIPS, the architectural state consists of the register file and PC. If the operating system saves the architectural state at some point in the program, it can interrupt the program, do something else, 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. Wrote the first computer program. It calculated the Bernoulli numbers using Charles Babbage's Analytical Engine. She was the only legitimate child of the poet Lord Byron.

6.4 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, `if/else` statements, `for` and `while` loops, array indexing, and procedure calls. In this section, we explore how to translate these high-level constructs into MIPS assembly code.

6.4.1 Arithmetic/Logical Instructions

The MIPS architecture defines a variety of arithmetic and logical instructions. We introduce these instructions briefly here, because they are necessary to implement higher-level constructs.

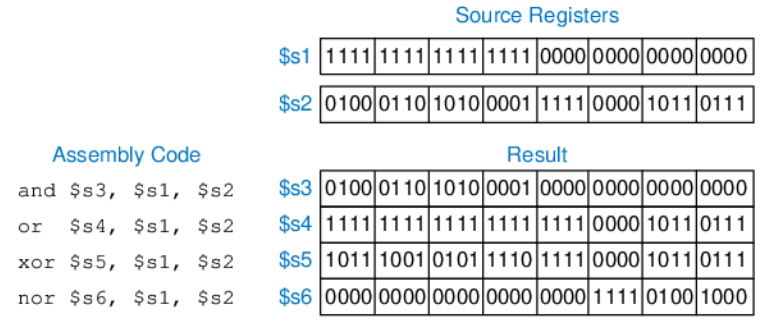


Figure 6.14 Logical operations

Logical Instructions

MIPS logical operations include `and`, `or`, `xor`, and `nor`. These R-type instructions operate bit-by-bit on two source registers and write the result to the destination register. Figure 6.14 shows examples of these operations on the two source values 0xFFFF0000 and 0x46A1F0B7. The figure shows the values stored in the destination register, `rd`, after the instruction executes.

The `and` instruction is useful for *masking* bits (i.e., forcing unwanted bits to 0). For example, in Figure 6.14, 0xFFFF0000 AND 0x46A1F0B7 = 0x46A10000. The `and` instruction masks off the bottom two bytes and places the unmasked top two bytes of `$s2`, 0x46A1, in `$s3`. Any subset of register bits can be masked.

The `or` instruction is useful for combining bits from two registers. For example, 0x347A0000 OR 0x000072FC = 0x347A72FC, a combination of the two values.

MIPS does not provide a NOT instruction, but A NOR $0 = NOT A, so the NOR instruction can substitute.

Logical operations can also operate on immediates. These I-type instructions are `andi`, `ori`, and `xori`. `nori` is not provided, because the same functionality can be easily implemented using the other instructions, as will be explored in Exercise 6.11. Figure 6.15 shows examples of the `andi`, `ori`, and `xori` instructions. The figure gives the values of

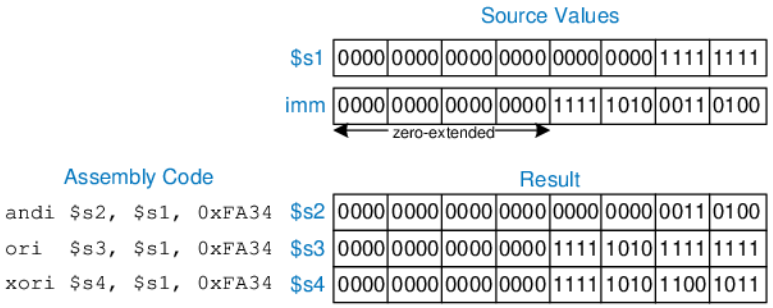


Figure 6.15 Logical operations with immediates

the source register and immediate, and the value of the destination register, `rt`, after the instruction executes. Because these instructions operate on a 32-bit value from a register and a 16-bit immediate, they first zero-extend the immediate to 32 bits.

Shift Instructions

Shift instructions shift the value in a register left or right by up to 31 bits. Shift operations multiply or divide by powers of two. MIPS shift operations are `sll` (shift left logical), `srl` (shift right logical), and `sra` (shift right arithmetic).

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). Figure 6.16 shows the machine code for the R-type instructions `sll`, `srl`, and `sra`. `rt` (i.e., `$s1`) holds the 32-bit value to be shifted, and `shamt` gives the amount by which to shift (4). The shifted result is placed in `rd`.

Figure 6.17 shows the register values for the shift instructions `sll`, `srl`, and `sra`. 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.

MIPS also has variable-shift instructions: `sllv` (shift left logical variable), `srlv` (shift right logical variable), and `srav` (shift right arithmetic variable). Figure 6.18 shows the machine code for these instructions.

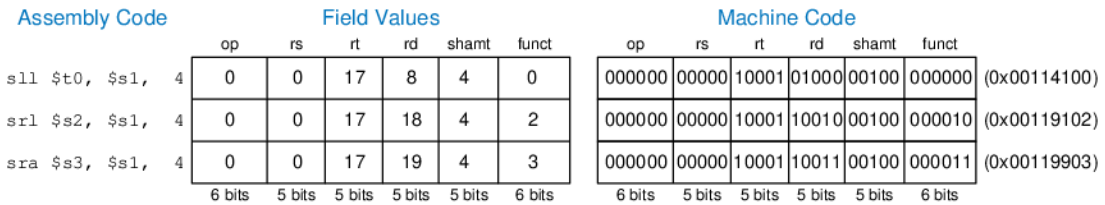


Figure 6.16 Shift instruction machine code

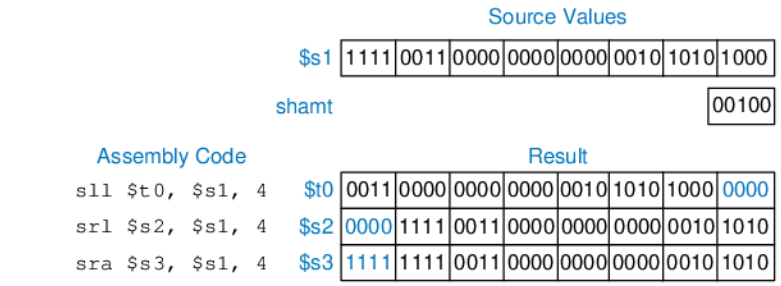


Figure 6.17 Shift operations

Assembly Code	Field Values						Machine Code						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
sllv $s3, $s1, $s2	0	18	17	19	0	4	000000	10010	10001	10011	00000	000100	(0x02519804)
srlv $s4, $s1, $s2	0	18	17	20	0	6	000000	10010	10001	10100	00000	000110	(0x0251A006)
srav $s5, $s1, $s2	0	18	17	21	0	7	000000	10010	10001	10101	00000	000111	(0x0251A807)
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Figure 6.18 Variable-shift instruction machine code

Source Values								
$s1	1111	0011	0000	0100	0000	0010	1010	1000
$s2	0000	0000	0000	0000	0000	0000	0000	**1000**

Assembly Code	Result								
sllv $s3, $s1, $s2	$s3	0000	0100	0000	0010	1010	1000	0000	0000
srlv $s4, $s1, $s2	$s4	0000	0000	1111	0011	0000	0100	0000	0010
srav $s5, $s1, $s2	$s5	1111	1111	1111	0011	0000	0100	0000	0010

Figure 6.19 Variable-shift operations

`rt` (i.e., `$s1`) holds the value to be shifted, and the five least significant bits of `rs` (i.e., `$s2`) give the amount to shift. The shifted result is placed in `rd`, as before. The `shamt` field is ignored and should be all 0's. Figure 6.19 shows register values for each type of variable-shift instruction.

Generating Constants

The `addi` instruction is helpful for assigning 16-bit constants, as shown in Code Example 6.10.

Code Example 6.10 16-BIT CONSTANT

High-Level Code

```
int a = 0x4f3c;
```

MIPS Assembly code

```
# $s0 = a
  addi $s0, $0, 0x4f3c   # a = 0x4f3c
```

Code Example 6.11 32-BIT CONSTANT

High-Level Code

```
int a = 0x6d5e4f3c;
```

MIPS Assembly Code

```
# $s0 = a
  lui $s0, 0x6d5e        # a = 0x6d5e0000
  ori $s0, $s0, 0x4f3c   # a = 0x6d5e4f3c
```

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

To assign 32-bit constants, use a load upper immediate instruction (lui) followed by an or immediate (ori) instruction, as shown in Code Example 6.11. lui loads a 16-bit immediate into the upper half of a register and sets the lower half to 0. As mentioned earlier, ori merges a 16-bit immediate into the lower half.

Multiplication and Division Instructions*

Multiplication and division are somewhat different from other arithmetic operations. Multiplying two 32-bit numbers produces a 64-bit product. Dividing two 32-bit numbers produces a 32-bit quotient and a 32-bit remainder.

hi and lo are not among the usual 32 MIPS registers, so special instructions are needed to access them. mfhi $s2 (move from hi) copies the value in hi to $s2. mflo $s3 (move from lo) copies the value in lo to $s3. hi and lo are technically part of the architectural state; however, we generally ignore these registers in this book.

The MIPS architecture has two special-purpose registers, hi and lo, which are used to hold the results of multiplication and division. mult $s0, $s1 multiplies the values in $s0 and $s1. The 32 most significant bits are placed in hi and the 32 least significant bits are placed in lo. Similarly, div $s0, $s1 computes $s0/$s1. The quotient is placed in lo and the remainder is placed in hi.

6.4.2 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, case statements, while loops, and for loops all conditionally execute code depending on some test.

To sequentially execute instructions, the program counter increments by 4 after each instruction. *Branch* instructions modify the program counter to skip over sections of code or to go back to repeat previous code. *Conditional branch* instructions perform a test and branch only if the test is TRUE. *Unconditional branch* instructions, called *jumps*, always branch.

Conditional Branches

The MIPS instruction set has two conditional branch instructions: branch if equal (beq) and branch if not equal (bne). beq branches when the values in two registers are equal, and bne branches when they are not equal. Code Example 6.12 illustrates the use of beq. Note that branches are written as beq $rs, $rt, imm, where $rs is the first source register. This order is reversed from most I-type instructions.

When the program in Code Example 6.12 reaches the branch if equal instruction (beq), the value in $s0 is equal to the value in $s1, so the branch is *taken*. That is, the next instruction executed is the add instruction just after the *label* called target. The two instructions directly after the branch and before the label are not executed.

Assembly code uses labels to indicate instruction locations in the program. When the assembly code is translated into machine code, these

Code Example 6.12 CONDITIONAL BRANCHING USING `beq`

MIPS Assembly Code

```
  addi  $s0, $0, 4        # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1        # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2       # $s1 = 1 << 2 = 4
  beq   $s0, $s1, target  # $s0 == $s1, so branch is taken
  addi  $s1, $s1, 1       # not executed
  sub   $s1, $s1, $s0     # not executed

target:
  add   $s1, $s1, $s0     # $s1 = 4 + 4 = 8
```

labels are translated into instruction addresses (see Section 6.5). MIPS assembly labels are followed by a (`:`) and cannot use reserved words, such as instruction mnemonics. Most programmers indent their instructions but not the labels, to help make labels stand out.

Code Example 6.13 shows an example using the branch if not equal instruction (`bne`). In this case, the branch is *not taken* because `$s0` is equal to `$s1`, and the code continues to execute directly after the `bne` instruction. All instructions in this code snippet are executed.

Code Example 6.13 CONDITIONAL BRANCHING USING `bne`

MIPS Assembly Code

```
  addi   $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi   $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll    $s1, $s1, 2         # $s1 = 1 << 2 = 4
  bne    $s0, $s1, target    # $s0 == $s1, so branch is not taken
  addi   $s1, $s1, 1         # $s1 = 4 + 1 = 5
  sub    $s1, $s1, $s0       # $s1 = 5 - 4 = 1

target:
  add    $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Jump

A program can unconditionally branch, or *jump*, using the three types of jump instructions: jump (`j`), jump and link (`jal`), and jump register (`jr`). Jump (`j`) jumps directly to the instruction at the specified label. Jump and link (`jal`) is similar to `j` but is used by procedures to save a return address, as will be discussed in Section 6.4.6. Jump register (`jr`) jumps to the address held in a register. Code Example 6.14 shows the use of the jump instruction (`j`).

`j` and `jal` are J-type instructions. `jr` is an R-type instruction that uses only the `rs` operand.

After the `j target` instruction, the program in Code Example 6.14 unconditionally continues executing the `add` instruction at the label `target`. All of the instructions between the jump and the label are skipped.

Code Example 6.14 UNCONDITIONAL BRANCHING USING `j`

MIPS Assembly Code

```
  addi   $s0, $0, 4     # $s0 = 4
  addi   $s1, $0, 1     # $s1 = 1
  j      target         # jump to target
  addi   $s1, $s1, 1    # not executed
  sub    $s1, $s1, $s0  # not executed
target:
  add    $s1, $s1, $s0  # $s1 = 1 + 4 = 5
```

Code Example 6.15 UNCONDITIONAL BRANCHING USING `jr`

MIPS Assembly Code

```
0x00002000   addi   $s0, $0, 0x2010  # $s0 = 0x2010
0x00002004   jr     $s0              # jump to 0x00002010
0x00002008   addi   $s1, $0, 1       # not executed
0x0000200c   sra    $s1, $s1, 2      # not executed
0x00002010   lw     $s3, 44 ($s1)    # executed after jr instruction
```

Code Example 6.15 shows the use of the jump register instruction (`jr`). Instruction addresses are given to the left of each instruction. `jr $s0` jumps to the address held in `$s0`, 0x00002010.

6.4.3 Conditional Statements

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

If Statements

An `if` statement executes a block of code, the *if block*, only when a condition is met. Code Example 6.16 shows how to translate an `if` statement into MIPS assembly code.

Code Example 6.16 `if` STATEMENT

High-Level Code

```
if (i == j)
  f = g + h;

f = f - i;
```

MIPS Assembly Code

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
  bne $s3,  $s4, L1     # if i != j, skip if block
  add $s0,  $s1, $s2    # if block: f = g + h
L1:
  sub $s0,  $s0, $s3    # f = f - i
```

The assembly code for the `if` statement tests the opposite condition of the one in the high-level code. In Code Example 6.16, the high-level code tests for `i == j`, and the assembly code tests for `i != j`. The `bne` instruction branches (skips the `if` block) when `i != j`. Otherwise, `i == j`, the branch is not taken, and the `if` block is executed as desired.

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 6.17 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. For example, in Code Example 6.17, the high-level code tests for `i == j`. The assembly code tests for the opposite condition (`i != j`). If that opposite condition is TRUE, `bne` skips the `if` block and executes the `else` block. Otherwise, the `if` block executes and finishes with a jump instruction (`j`) to jump past the `else` block.

Code Example 6.17 `if/else` STATEMENT

High-Level Code

```
if (i == j)
  f = g + h;

else
  f = f - i;
```

MIPS Assembly Code

```
# $s0 = f, $s1 = g,   $s2 = h, $s3 = i, $s4 = j
  bne $s3, $s4, else     # if i ! = j, branch to else
  add $s0, $s1, $s2      # if block: f = g + h
  j   L2                 # skip past the else block
else:
  sub $s0, $s0, $s3      # else block: f = f - i
L2:
```

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 6.18 shows two high-level code snippets with the same functionality: they calculate the fee for an *ATM* (*automatic teller machine*) withdrawal of $20, $50, or $100, as defined by `amount`. The MIPS assembly implementation is the same for both high-level code snippets.

6.4.4 Getting Loopy

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

Code Example 6.18 `switch/case` STATEMENT

High-Level Code

```
switch (amount) {
  case 20: fee = 2; break;

  case 50: fee = 3; break;

  case 100: fee = 5; break;

  default: fee = 0;
}

// equivalent function using if/else statements
  if      (amount == 20)  fee = 2;
  else if (amount == 50)  fee = 3;
  else if (amount == 100) fee = 5;
  else                    fee = 0;
```

MIPS Assembly Code

```
# $s0 = amount, $s1 = fee
case20:
  addi $t0, $0,  20      # $t0 = 20
  bne  $s0, $t0, case50  # i == 20? if not,
                         #   skip to case50
  addi $s1, $0,  2       # if so, fee = 2
  j    done              # and break out of case
case50:
  addi $t0, $0,  50      # $t0 = 50
  bne  $s0, $t0, case100 # i == 50? if not,
                         #   skip to case100
  addi $s1, $0,  3       # if so, fee = 3
  j    done              # and break out of case
case100:
  addi $t0, $0,  100     # $t0 = 100
  bne  $s0, $t0, default # i == 100? if not,
                         #   skip to default
  addi $s1, $0,  5       # if so, fee = 5
  j    done              # and break out of case
default:
  add  $s1, $0, $0       # charge = 0

done:
```

While Loops

`while` loops repeatedly execute a block of code until a condition is *not* met. The `while` loop in Code Example 6.19 determines the value of `x` such that $2^x = 128$. It executes seven times, until `pow` = 128.

Like `if/else` statements, the assembly code for `while` loops tests the opposite condition of the one given in the high-level code. If that opposite condition is TRUE, the `while` loop is finished.

Code Example 6.19 `while` LOOP

High-Level Code

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


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

MIPS Assembly Code

```
# $s0 = pow, $s1 = x
  addi  $s0, $0, 1      # pow = 1
  addi  $s1, $0, 0      # x = 0

  addi  $t0, $0, 128    # t0 = 128 for comparison
while:
  beq   $s0, $t0, done  # if pow == 128, exit while
  sll   $s0, $s0, 1     # pow = pow * 2
  addi  $s1, $s1, 1     # x = x + 1
  j     while
done:
```

In Code Example 6.19, 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 jumps back to the start of the `while` loop.

For Loops

`for` loops, like `while` loops, repeatedly execute a block of code until a condition is *not* met. However, `for` loops add support for a *loop variable*, which typically keeps track of the number of loop executions. A general format of the `for` loop is

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

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 6.20 adds the numbers from 0 to 9. The loop variable, `i`, is initialized to 0 and is incremented at the end of each loop iteration. At the beginning of each iteration, the `for` loop executes only when `i` is not equal to 10. Otherwise, the loop is finished. In this case, the `for` loop executes 10 times. `for` loops can be implemented using a `while` loop, but the `for` loop is often convenient.

Magnitude Comparison

So far, the examples have used `beq` and `bne` to perform equality or inequality comparisons and branches. MIPS provides the *set less than* instruction, `slt`, for magnitude comparison. `slt` sets `rd` to 1 when `rs` < `rt`. Otherwise, `rd` is 0.

Code Example 6.20 `for` LOOP

High-Level Code

```
int sum = 0;



for (i = 0; i ! = 10; i = i + 1) {
  sum = sum + i;


}

// equivalent to the following while loop
int sum = 0;
int i = 0;
while (i != 10) {
  sum = sum + i;
  i = i + 1;
}
```

MIPS Assembly Code

```
# $s0 = i, $s1 = sum
  add   $s1, $0, $0      # sum = 0
  addi  $s0, $0, 0       # i   = 0
  addi  $t0, $0, 10      # $t0 = 10
for:
  beq   $s0, $t0, done   # if i == 10, branch to done
  add   $s1, $s1, $s0    # sum = sum + i
  addi  $s0, $s0, 1      # increment i
  j     for
done:
```

Example 6.6 LOOPS USING `slt`

The following high-level code adds the powers of 2 from 1 to 100. Translate it into assembly language.

```
// high-level code

int sum = 0;
for (i = 1; i < 101; i = i * 2)
  sum = sum + i;
```

Solution: The assembly language code uses the set less than (`slt`) instruction to perform the less than comparison in the `for` loop.

```
# MIPS assembly code

# $s0 = i, $s1 = sum
  addi $s1, $0, 0    # sum = 0
  addi $s0, $0, 1    # i = 1
  addi $t0, $0, 101 # $t0 = 101

loop:
  slt $t1, $s0, $t0     # if (i < 101) $t1 = 1, else $t1 = 0
  beq $t1, $0, done     # if $t1 == 0 (i >= 101), branch to done
  add $s1, $s1, $s0     # sum = sum + i
  sll $s0, $s0, 1       # i = i * 2
  j   loop

done:
```

Exercise 6.12 explores how to use `slt` for other magnitude comparisons including greater than, greater than or equal, and less than or equal.

6.4.5 Arrays

Arrays are useful for accessing large amounts of similar data. An array is organized as 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 *size* of the array. This section shows how to access array elements in memory.

Array Indexing

Figure 6.20 shows an array of five integers stored in memory. The *index* ranges from 0 to 4. In this case, the array is stored in a processor's main memory starting at *base address* 0x10007000. The base address gives the address of the first array element, `array[0]`.

Code Example 6.21 multiplies the first two elements in `array` by 8 and stores them back in the array.

The first step in accessing an array element is to load the base address of the array into a register. Code Example 6.21 loads the base address

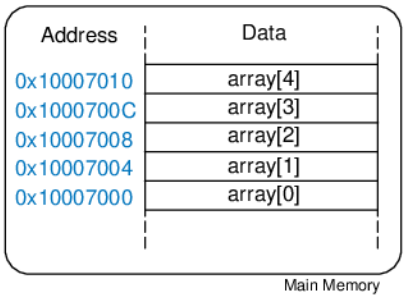


Figure 6.20 Five-entry array with base address of 0x10007000

Code Example 6.21 ACCESSING ARRAYS

High-Level Code

```
int array [5];

array[0] = array[0] * 8;

array[1] = array[1] * 8;
```

MIPS Assembly Code

```
# $s0 = base address of array
  lui   $s0, 0x1000      # $s0 = 0x10000000
  ori   $s0, $s0, 0x7000 # $s0 = 0x10007000

  lw    $t1, 0($s0)      # $t1 = array[0]
  sll   $t1, $t1, 3      # $t1 = $t1 << 3 = $t1 * 8
  sw    $t1, 0($s0)      # array[0] = $t1

  lw    $t1, 4($s0)      # $t1 = array[1]
  sll   $t1, $t1, 3      # $t1 = $t1 << 3 = $t1 * 8
  sw    $t1, 4($s0)      # array[1] = $t1
```

into `$s0`. Recall that the load upper immediate (`lui`) and or immediate (`ori`) instructions can be used to load a 32-bit constant into a register.

Code Example 6.21 also illustrates why `lw` takes a base address and an offset. The base address points to the start of the array. The offset can be used to access subsequent elements of the array. For example, `array[1]` is stored at memory address 0x10007004 (one word or four bytes after `array[0]`), so it is accessed at an offset of 4 past the base address.

You might have noticed that the code for manipulating each of the two array elements in Code Example 6.21 is essentially the same except for the index. Duplicating the code is not a problem when accessing two array elements, but it would become terribly inefficient for accessing all of the elements in a large array. Code Example 6.22 uses a `for` loop to multiply by 8 all of the elements of a 1000-element array stored at a base address of 0x23B8F000.

Figure 6.21 shows the 1000-element array in memory. The index into the array is now a variable (`i`) rather than a constant, so we cannot take advantage of the immediate offset in `lw`. Instead, we compute the address of the ith element and store it in `$t0`. Remember that each array element is a word but that memory is byte addressed, so the offset from

Code Example 6.22 ACCESSING ARRAYS USING A `for` LOOP

High-Level Code

```
int i;
int array[1000];




for (i=0; i < 1000; i = i + 1) {



  array[i] = array[i] * 8;



}
```

MIPS Assembly Code

```
# $s0 = array base address, $s1 = i
# initialization code
  lui   $s0, 0x23B8       # $s0 = 0x23B80000
  ori   $s0, $s0, 0xF000  # $s0 = 0x23B8F000
  addi  $s1, $0           # i = 0
  addi  $t2, $0, 1000     # $t2 = 1000

loop:
  slt   $t0, $s1, $t2     # i < 1000?
  beq   $t0, $0, done     # if not then done
  sll   $t0, $s1, 2       # $t0 = i * 4 (byte offset)
  add   $t0, $t0, $s0     # address of array[i]
  lw    $t1, 0($t0)       # $t1 = array[i]
  sll   $t1, $t1, 3       # $t1 = array[i] * 8
  sw    $t1, 0($t0)       # array[i] = array[i] * 8
  addi  $s1, $s1, 1       # i = i + 1
  j     loop              # repeat
done:
```

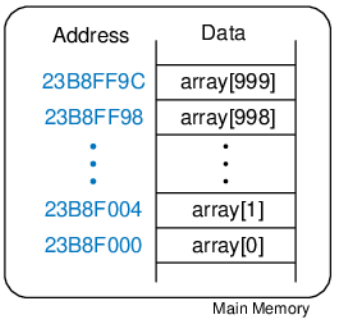


Figure 6.21 Memory holding array[1000] starting at base address 0x23B8F000

the base address is `i * 4`. Shifting left by 2 is a convenient way to multiply by 4 in MIPS assembly language. This example readily extends to an array of any size.

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.

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 6.2 shows these character encodings for printable characters. The ASCII values are given in hexadecimal. Lower-case and upper-case letters differ by 0x20 (32).

Other program 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.

MIPS provides load byte and store byte instructions to manipulate bytes or characters of data: load byte unsigned (`lbu`), load byte (`lb`), and store byte (`sb`). All three are illustrated in Figure 6.22.

Table 6.2 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		

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.

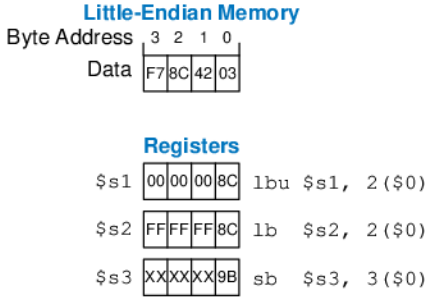


Figure 6.22 Instructions for loading and storing bytes

Load byte unsigned (`lbu`) zero-extends the byte, and load byte (`lb`) sign-extends the byte to fill the entire 32-bit register. Store byte (`sb`) stores the least significant byte of the 32-bit register into the specified byte address in memory. In Figure 6.22, `lbu` loads the byte at memory address 2 into the least significant byte of `$s1` and fills the remaining register bits with 0. `lb` loads the sign-extended byte at memory address `2` into `$s2`. `sb` stores the least significant byte of `$s3` into memory byte `3`; it replaces 0xF7 with 0x9B. The more significant bytes of `$s3` are ignored.

Example 6.7 USING `lb` AND `sb` TO ACCESS A CHARACTER ARRAY

The following high-level code converts a ten-entry array of characters from lower-case to upper-case by subtracting 32 from each array entry. Translate it into MIPS assembly language. Remember that the address difference between array elements is now 1 byte, not 4 bytes. Assume that `$s0` already holds the base address of `chararray`.

```
// high-level code

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

Solution:

```
# MIPS assembly code
# $s0 = base address of chararray, $s1 = i

          addi $s1, $0, 0      # i = 0
          addi $t0, $0, 10     # $t0 = 10
loop:     beq  $t0, $s1, done  # if i == 10, exit loop
          add  $t1, $s1, $s0   # $t1 = address of chararray[i]
          lb   $t2, 0($t1)     # $t2 = array[i]
          addi $t2, $t2, -32   # convert to upper case: $t1 = $t1 - 32
          sb   $t2, 0($t1)     # store new value in array:
                               # chararray[i] = $t1
          addi $s1, $s1, 1     # i = i + 1
          j    loop            # repeat
done:
```

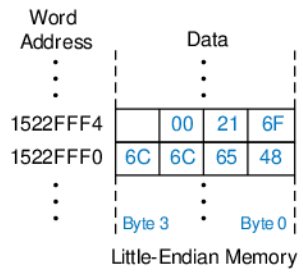


Figure 6.23 The string "Hello!" stored in memory

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 6.23 shows the string "Hello!" (0x48 65 6C 6C 6F 21 00) stored in memory. The string is seven bytes long and extends from address 0x1522FFF0 to 0x1522FFF6. The first character of the string (H = 0x48) is stored at the lowest byte address (0x1522FFF0).

6.4.6 Procedure Calls

High-level languages often use *procedures* (also called *functions*) to reuse frequently accessed code and to make a program more readable. Procedures have inputs, called *arguments*, and an output, called the *return value*. Procedures should calculate the return value and cause no other unintended side effects.

When one procedure calls another, the calling procedure, the *caller*, and the called procedure, the *callee*, must agree on where to put the arguments and the return value. In MIPS, the caller conventionally places up to four arguments in registers `$a0-$a3` before making the procedure call, and the callee places the return value in registers `$v0-$v1` before finishing. By following this convention, both procedures 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 function of the caller. Briefly, 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 `$ra` at the same time it jumps to the callee using the jump and link instruction (`jal`). The callee must not overwrite any architectural state or memory that the caller is depending on. Specifically, the callee must leave the saved registers, `$s0-$s7`, `$ra`, and the *stack*, a portion of memory used for temporary variables, unmodified.

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

Procedure Calls and Returns

MIPS uses the *jump and link* instruction (`jal`) to call a procedure and the *jump register* instruction (`jr`) to return from a procedure. Code Example 6.23 shows the `main` procedure calling the `simple` procedure. `main` is the caller, and `simple` is the callee. The `simple` procedure is called with no input arguments and generates no return value; it simply returns to the caller. In Code Example 6.23, instruction addresses are given to the left of each MIPS instruction in hexadecimal.

Code Example 6.23 `simple` PROCEDURE CALL

High-Level Code

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

MIPS Assembly Code

```
0x00400200 main:   jal simple # call procedure
0x00400204         ...


0x00401020 simple: jr $ra     # return
```

Jump and link (`jal`) and jump register (`jr $ra`) are the two essential instructions needed for a procedure call. `jal` performs two functions: it stores the address of the *next* instruction (the instruction after `jal`) in the return address register (`$ra`), and it jumps to the target instruction.

In Code Example 6.23, the `main` procedure calls the `simple` procedure by executing the jump and link (`jal`) instruction. `jal` jumps to the `simple` label and stores 0x00400204 in `$ra`. The `simple` procedure returns immediately by executing the instruction `jr $ra`, jumping to the instruction address held in `$ra`. The main procedure then continues executing at this address, 0x00400204.

Input Arguments and Return Values

The `simple` procedure in Code Example 6.23 is not very useful, because it receives no input from the calling procedure (`main`) and returns no output. By MIPS convention, procedures use `$a0-$a3` for input arguments and `$v0-$v1` for the return value. In Code Example 6.24, the procedure `diffofsums` is called with four arguments and returns one result.

According to MIPS convention, the calling procedure, `main`, places the procedure arguments, from left to right, into the input registers, `$a0-$a3`. The called procedure, `diffofsums`, stores the return value in the return register, `$v0`.

A procedure that returns a 64-bit value, such as a double-precision floating point number, uses both return registers, `$v0` and `$v1`. When a procedure with more than four arguments is called, the additional input arguments are placed on the stack, which we discuss next.

Code Example 6.24 has some subtle errors. Code Examples 6.25 and 6.26 on page 323 show improved versions of the program.

Code Example 6.24 PROCEDURE 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;
}
```

MIPS Assembly Code

```
# $s0 = y

main:
  . . .
  addi $a0, $0, 2   # argument 0 = 2
  addi $a1, $0, 3   # argument 1 = 3
  addi $a2, $0, 4   # argument 2 = 4
  addi $a3, $0, 5   # argument 3 = 5
  jal diffofsums    # call procedure
  add $s0, $v0, $0  # y = returned value
  . . .

# $s0 = result
diffofsums:
  add $t0, $a0, $a1 # $t0 = f + g
  add $t1, $a2, $a3 # $t1 = h + i
  sub $s0, $t0, $t1 # result = (f + g) - (h + i)
  add $v0, $s0, $0  # put return value in $v0
  jr  $ra           # return to caller
```

The Stack

The *stack* is memory that is used to save local variables within a procedure. 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 procedures use the stack to store temporary variables, 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 pulled (*popped*) off. Each procedure 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 MIPS stack grows *down* in memory. The stack expands to lower memory addresses when a program needs more scratch space.

Figure 6.24 shows a picture of the stack. The *stack pointer*, `$sp`, is a special MIPS register that points to the top of the stack. A *pointer* is a fancy name for a memory address. It points to (gives the address of) data. For example, in Figure 6.24(a) the stack pointer, `$sp`, holds the address value 0x7FFFFFFC and points to the data value 0x12345678. `$sp` points to the top of the stack, the lowest accessible memory address on the stack. Thus, in Figure 6.24(a), the stack cannot access memory below memory word 0x7FFFFFFC.

The stack pointer (`$sp`) starts at a high memory address and decrements to expand as needed. Figure 6.24(b) shows the stack expanding to allow two more data words of temporary storage. To do so, `$sp` decrements by 8 to become 0x7FFFFFF4. Two additional data words, 0xAABBCCDD and 0x11223344, 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 procedure. Recall that a procedure should calculate a return value but have no other unintended side effects. In particular, it should not modify any registers besides the one containing the return value, `$v0`. The `diffofsums` procedure in Code Example 6.24 violates this rule because it modifies `$t0`, `$t1`, and `$s0`. If `main` had been using `$t0`, `$t1`, or `$s0` before the call to `diffofsums`, the contents of these registers would have been corrupted by the procedure call.

To solve this problem, a procedure 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 procedure using the registers.
4. Restores the original values of the registers from the stack.
5. Deallocates space on the stack.

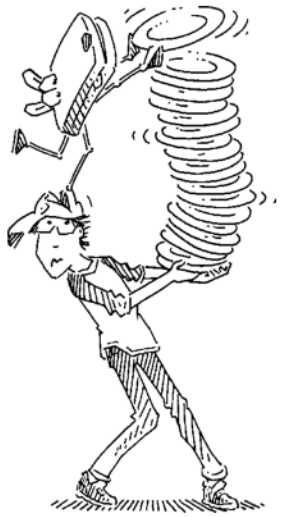


Figure 6.24 The stack

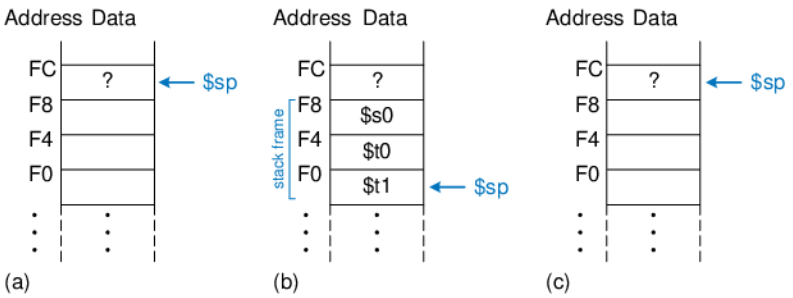


Figure 6.25 The stack (a) before, (b) during, and (c) after `diffofsums` procedure call

Code Example 6.25 shows an improved version of `diffofsums` that saves and restores `$t0`, `$t1`, and `$s0`. The new lines are indicated in blue. Figure 6.25 shows the stack before, during, and after a call to the `diffofsums` procedure from Code Example 6.25. `diffofsums` makes room for three words on the stack by decrementing the stack pointer (`$sp`) by 12. It then stores the current values of `$s0`, `$t0`, and `$t1` in the newly allocated space. It executes the rest of the procedure, changing the values in these three registers. At the end of the procedure, `diffofsums` restores the values of `$s0`, `$t0`, and `$t1` from the stack, deallocates its stack space, and returns. When the procedure returns, `$v0` holds the result, but there are no other side effects: `$s0`, `$t0`, `$t1`, and `$sp` have the same values as they did before the procedure call.

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

Preserved Registers

Code Example 6.25 assumes that temporary registers `$t0` and `$t1` must be saved and restored. If the calling procedure does not use those registers, the effort to save and restore them is wasted. To avoid this waste, MIPS divides registers into *preserved* and *nonpreserved* categories. The preserved registers include `$s0-$s7` (hence their name, *saved*). The nonpreserved registers include `$t0-$t9` (hence their name, *temporary*). A procedure must save and restore any of the preserved registers that it wishes to use, but it can change the nonpreserved registers freely.

Code Example 6.26 shows a further improved version of `diffofsums` that saves only `$s0` on the stack. `$t0` and `$t1` are nonpreserved registers, so they need not be saved.

Remember that when one procedure 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

Code Example 6.25 PROCEDURE SAVING REGISTERS ON THE STACK

MIPS Assembly Code

```
# $s0 = result
diffofsums:
  addi  $sp, $sp, -12 # make space on stack to store three registers
  sw    $s0, 8($sp)   # save $s0 on stack
  sw    $t0, 4($sp)   # save $t0 on stack
  sw    $t1, 0($sp)   # save $t1 on stack
  add   $t0, $a0, $a1 # $t0 = f + g
  add   $t1, $a2, $a3 # $t1 = h + i
  sub   $s0, $t0, $t1 # result = (f + g) - (h + i)
  add   $v0, $s0, $0  # put return value in $v0
  lw    $t1, 0($sp)   # restore $t1 from stack
  lw    $t0, 4($sp)   # restore $t0 from stack
  lw    $s0, 8($sp)   # restore $s0 from stack
  addi  $sp, $sp, 12  # deallocate stack space
  jr    $ra           # return to caller
```

Code Example 6.26 PROCEDURE SAVING PRESERVED REGISTERS ON THE STACK

MIPS Assembly Code

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4   # make space on stack to store one register
  sw   $s0, 0($sp)    # save $s0 on stack
  add  $t0, $a0, $a1  # $t0 = f + g
  add  $t1, $a2, $a3  # $t1 = h + i
  sub  $s0, $t0, $t1  # result = (f + g) - (h + i)
  add  $v0, $s0, $0   # put return value in $v0
  lw   $s0, 0($sp)    # restore $s0 from stack
  addi $sp, $sp, 4    # deallocate stack space
  jr   $ra            # return to caller
```

nonpreserved register, the caller needs to save that nonpreserved register before making the procedure 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 6.3 summarizes which registers are preserved. `$s0-$s7` are generally used to hold local variables within a procedure, so they must be saved. `$ra` must also be saved, so that the procedure knows where to return. `$t0-$t9` are used to hold temporary results before they are assigned to local variables. These calculations typically complete before a procedure call is made, so they are not preserved, and it is rare that the caller needs to save them. `$a0-$a3` are often overwritten in the process of calling a procedure. Hence, they must be saved by the caller if the caller depends on any of its own arguments after a called procedure returns. `$v0-$v1` certainly should not be preserved, because the callee returns its result in these registers.

Table 6.3 Preserved and nonpreserved registers

Preserved	Nonpreserved
Saved registers: `$s0-$s7`	Temporary registers: `$t0-$t9`
Return address: `$ra`	Argument registers: `$a0-$a3`
Stack pointer: `$sp`	Return value registers: `$v0-$v1`
Stack above the stack pointer	Stack below the stack pointer

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

Recursive Procedure Calls

A procedure that does not call others is called a *leaf* procedure; an example is `diffofsums`. A procedure that does call others is called a *nonleaf* procedure. As mentioned earlier, nonleaf procedures are somewhat more complicated because they may need to save nonpreserved registers on the stack before they call another procedure, and then restore those registers afterward. Specifically, the caller saves any nonpreserved registers (`$t0-$t9` and `$a0-$a3`) that are needed after the call. The callee saves any of the preserved registers (`$s0-$s7` and `$ra`) that it intends to modify.

A *recursive* procedure is a nonleaf procedure that calls itself. The factorial function can be written as a recursive procedure call. Recall that $factorial(n) = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$. The factorial function can be rewritten recursively as $factorial(n) = n \times factorial(n-1)$. The factorial of 1 is simply 1. Code Example 6.27 shows the factorial function written as a recursive procedure. To conveniently refer to program addresses, we assume that the program starts at address 0x90.

The `factorial` procedure might modify `$a0` and `$ra`, so it saves them on the stack. It then checks whether `n` < 2. If so, it puts the return value of 1 in `$v0`, restores the stack pointer, and returns to the caller. It does not have to reload `$ra` and `$a0` in this case, because they were never modified. If `n` > 1, the procedure recursively calls `factorial(n-1)`. It then restores the value of `n` (`$a0`) and the return address (`$ra`) from the stack, performs the multiplication, and returns this result. The multiply instruction (`mul $v0, $a0, $v0`) multiplies `$a0` and `$v0` and places the result in `$v0`. It is discussed further in Section 6.7.1.

Code Example 6.27 `factorial` RECURSIVE PROCEDURE CALL

High-Level Code

```
int factorial (int n) {


  if (n <= 1)
      return 1;


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

MIPS Assembly Code

```
0x90  factorial:  addi  $sp, $sp, -8   # make room on stack
0x94              sw    $a0, 4($sp)    # store $a0
0x98              sw    $ra, 0($sp)    # store $ra
0x9C              addi  $t0, $0, 2     # $t0 = 2
0xA0              slt   $t0, $a0, $t0  # a <= 1 ?
0xA4              beq   $t0, $0, else  # no: goto else
0xA8              addi  $v0, $0, 1     # yes: return 1
0xAC              addi  $sp, $sp, 8    # restore $sp
0xB0              jr    $ra            # return
0xB4        else: addi  $a0, $a0, -1   # n = n - 1
0xB8              jal   factorial      # recursive call
0xBC              lw    $ra, 0($sp)    # restore $ra
0xC0              lw    $a0, 4($sp)    # restore $a0
0xC4              addi  $sp, $sp, 8    # restore $sp
0xC8              mul   $v0, $a0, $v0  # n * factorial (n-1)
0xCC              jr    $ra            # return
```

Figure 6.26 shows the stack when executing `factorial(3)`. We assume that `$sp` initially points to 0xFC, as shown in Figure 6.26(a). The procedure creates a two-word stack frame to hold `$a0` and `$ra`. On the first invocation, `factorial` saves `$a0` (holding `n` = 3) at 0xF8 and `$ra` at 0xF4, as shown in Figure 6.26(b). The procedure then changes `$a0` to `n` = 2 and recursively calls `factorial(2)`, making `$ra` hold 0xBC. On the second invocation, it saves `$a0` (holding `n` = 2) at 0xF0 and `$ra` at 0xEC. This time, we know that `$ra` contains 0xBC. The procedure then changes `$a0` to `n` = 1 and recursively calls `factorial(1)`. On the third invocation, it saves `$a0` (holding `n` = 1) at 0xE8 and `$ra` at 0xE4. This time, `$ra` again contains 0xBC. The third invocation of

Figure 6.26 **Stack during `factorial` procedure call when `n` = 3: (a) before call, (b) after last recursive call, (c) after return**

`factorial` returns the value 1 in `$v0` and deallocates the stack frame before returning to the second invocation. The second invocation restores `n` to 2, restores `$ra` to 0xBC (it happened to already have this value), deallocates the stack frame, and returns `$v0` = 2 × 1 = 2 to the first invocation. The first invocation restores `n` to 3, restores `$ra` to the return address of the caller, deallocates the stack frame, and returns `$v0` = 3 × 2 = 6. Figure 6.26(c) shows the stack as the recursively called procedures return. When `factorial` returns to the caller, the stack pointer is in its original position (0xFC), none of the contents of the stack above the pointer have changed, and all of the preserved registers hold their original values. `$v0` holds the return value, 6.

Additional Arguments and Local Variables*

Procedures may have more than four input arguments and local variables. The stack is used to store these temporary values. By MIPS convention, if a procedure 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 6.27(a) shows the caller's stack for calling a procedure with more than four arguments.

A procedure can also declare local variables or arrays. *Local* variables are declared within a procedure and can be accessed only within that procedure. Local variables are stored in `$s0`-`$s7`; if there are too many local variables, they can also be stored in the procedure's stack frame. In particular, local arrays are stored on the stack.

Figure 6.27(b) shows the organization of a callee's stack frame. The frame holds the procedure's own arguments (if it calls other procedures), the return address, and any of the saved registers that the procedure will

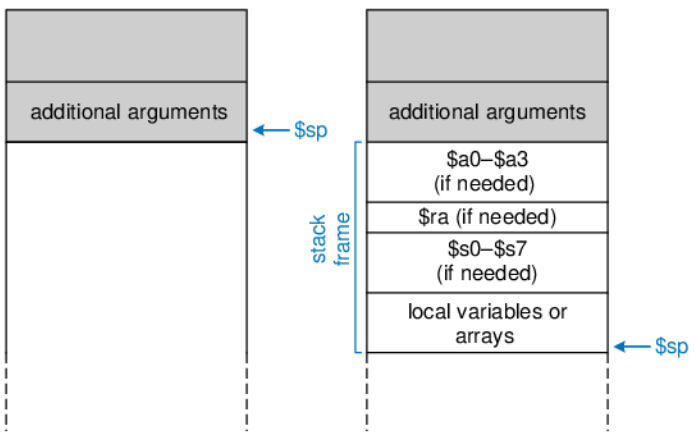


Figure 6.27 Stack usage: (left) before call, (right) after call

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 procedure can access stack data not in its own stack frame.

6.5 ADDRESSING MODES

MIPS uses five *addressing modes*: register-only, immediate, base, PC-relative, and pseudo-direct. The first three modes (register-only, immediate, and base addressing) define modes of reading and writing operands. The last two (PC-relative and pseudo-direct addressing) define modes of writing the program counter, `PC`.

Register-Only Addressing

Register-only addressing uses registers for all source and destination operands. All R-type instructions use register-only addressing.

Immediate Addressing

Immediate addressing uses the 16-bit immediate along with registers as operands. Some I-type instructions, such as add immediate (`addi`) and load upper immediate (`lui`), use immediate addressing.

Base Addressing

Memory access instructions, such as load word (`lw`) and store word (`sw`), use *base addressing*. The effective address of the memory operand is found by adding the base address in register `rs` to the sign-extended 16-bit offset found in the immediate field.

PC-relative Addressing

Conditional branch instructions use *PC-relative addressing* to specify the new value of the `PC` if the branch is taken. The signed offset in the immediate field is added to the `PC` to obtain the new `PC`; hence, the branch destination address is said to be *relative* to the current `PC`.

Code Example 6.28 shows part of the `factorial` procedure from Code Example 6.27. Figure 6.28 shows the machine code for the `beq` instruction. The *branch target address* (*BTA*) is the address of the next instruction to execute if the branch is taken. The `beq` instruction in Figure 6.28 has a BTA of 0xB4, the instruction address of the `else` label.

The 16-bit immediate field gives the number of instructions between the BTA and the instruction *after* the branch instruction (the instruction at `PC`+4). In this case, the value in the immediate field of `beq` is 3 because the BTA (0xB4) is 3 instructions past `PC`+4 (0xA8).

The processor calculates the BTA from the instruction by sign-extending the 16-bit immediate, multiplying it by 4 (to convert words to bytes), and adding it to `PC`+4.

Code Example 6.28 CALCULATING THE BRANCH TARGET ADDRESS

MIPS Assembly Code

```
0xA4          beq  $t0, $0, else
0xA8          addi $v0, $0, 1
0xAC          addi $sp, $sp, 8
0xB0          jr   $ra
0xB4    else: addi $a0, $a0, -1
0xB8          jal  factorial
```

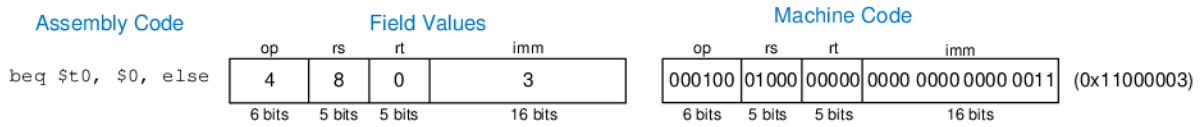


Figure 6.28 Machine code for `beq`

Example 6.8 CALCULATING THE IMMEDIATE FIELD FOR PC-RELATIVE ADDRESSING

Calculate the immediate field and show the machine code for the branch not equal (`bne`) instruction in the following program.

```
# MIPS assembly code
0x40 loop: add  $t1, $a0, $s0
0x44       lb   $t1, 0($t1)
0x48       add  $t2, $a1, $s0
0x4C       sb   $t1, 0($t2)
0x50       addi $s0, $s0, 1
0x54       bne  $t1, $0, loop
0x58       lw   $s0, 0($sp)
```

Solution: Figure 6.29 shows the machine code for the `bne` instruction. Its branch target address, 0x40, is 6 instructions behind `PC`+4 (0x58), so the immediate field is −6.

Figure 6.29 `bne` machine code

Pseudo-Direct Addressing

In *direct addressing*, an address is specified in the instruction. The jump instructions, `j` and `jal`, ideally would use direct addressing to specify a

Code Example 6.29 CALCULATING THE JUMP TARGET ADDRESS

MIPS Assembly Code

```
0x0040005C          jal     sum
...
0x004000A0  sum:    add     $v0, $a0, $a1
```

32-bit *jump target address* (*JTA*) to indicate the instruction address to execute next.

Unfortunately, the J-type instruction encoding does not have enough bits to specify a full 32-bit JTA. Six bits of the instruction are used for the `opcode`, so only 26 bits are left to encode the JTA. Fortunately, the two least significant bits, $JTA_{1:0}$, should always be 0, because instructions are word aligned. The next 26 bits, $JTA_{27:2}$, are taken from the `addr` field of the instruction. The four most significant bits, $JTA_{31:28}$, are obtained from the four most significant bits of `PC`+4. This addressing mode is called *pseudo-direct*.

Code Example 6.29 illustrates a `jal` instruction using pseudo-direct addressing. The JTA of the `jal` instruction is 0x004000A0. Figure 6.30 shows the machine code for this `jal` instruction. The top four bits and bottom two bits of the JTA are discarded. The remaining bits are stored in the 26-bit address field (`addr`).

The processor calculates the JTA from the J-type instruction by appending two 0's and prepending the four most significant bits of `PC`+4 to the 26-bit address field (`addr`).

Because the four most significant bits of the JTA are taken from `PC`+4, the jump range is limited. The range limits of branch and jump instructions are explored in Exercises 6.23 to 6.26. All J-type instructions, `j` and `jal`, use pseudo-direct addressing.

Note that the jump register instruction, `jr`, is *not* a J-type instruction. It is an R-type instruction that jumps to the 32-bit value held in register `rs`.

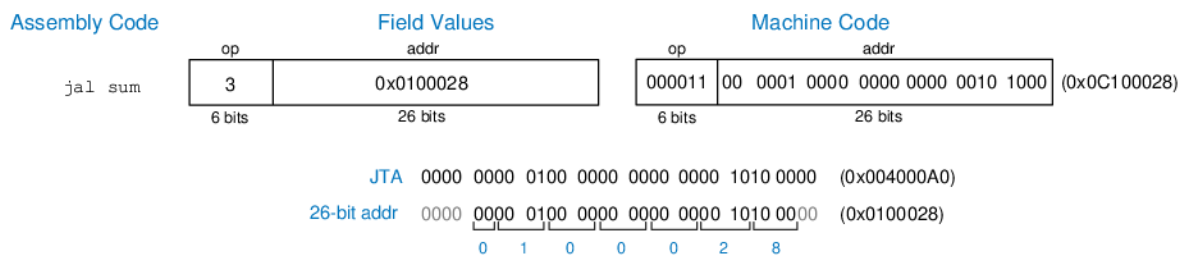


Figure 6.30 `jal` **machine code**

6.6 LIGHTS, CAMERA, ACTION: COMPILING, ASSEMBLING, AND LOADING

Up 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 the MIPS *memory map*, which defines where code, data, and stack memory are located. We then show the steps of code execution for a sample program.

6.6.1 The Memory Map

With 32-bit addresses, the MIPS *address space* spans 2^{32} bytes = 4 gigabytes (GB). Word addresses are divisible by 4 and range from 0 to 0xFFFFFFFC. Figure 6.31 shows the MIPS memory map. The MIPS architecture divides the address space into four parts or *segments*: the text segment, global data segment, dynamic data segment, and reserved segments. The following sections describes each segment.

The Text Segment

The *text segment* stores the machine language program. It is large enough to accommodate almost 256 MB of code. Note that the four most significant bits of the address in the text space are all 0, so the `j` instruction can directly jump to any address in the program.

The Global Data Segment

The *global data segment* stores global variables that, in contrast to local variables, can be seen by all procedures in a program. Global variables

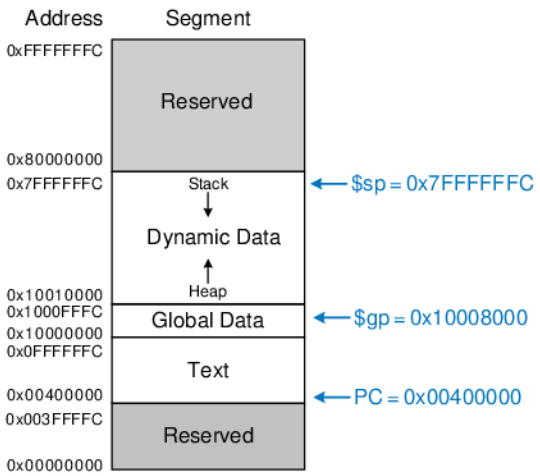


Figure 6.31 MIPS memory map

are defined at *start-up*, before the program begins executing. These variables are declared outside the main procedure in a C program and can be accessed by any procedure. The global data segment is large enough to store 64 KB of global variables.

Global variables are accessed using the global pointer (`$gp`), which is initialized to 0x100080000. Unlike the stack pointer (`$sp`), `$gp` does not change during program execution. Any global variable can be accessed with a 16-bit positive or negative offset from `$gp`. The offset is known at assembly time, so the variables can be efficiently accessed using base addressing mode with constant offsets.

The Dynamic Data Segment

The *dynamic data segment* holds the stack and the *heap*. The data in this segment are not known at start-up but are dynamically allocated and deallocated throughout the execution of the program. This is the largest segment of memory used by a program, spanning almost 2 GB of the address space.

As discussed in Section 6.4.6, the stack is used to save and restore registers used by procedures and to hold local variables such as arrays. The stack grows downward from the top of the dynamic data segment (0x7FFFFFFC) and is accessed in last-in-first-out (LIFO) 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 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.

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 Defence Service Medal.

The Reserved Segments

The *reserved segments* are used by the operating system and cannot directly be used by the program. Part of the reserved memory is used for interrupts (see Section 7.7) and for memory-mapped I/O (see Section 8.5).

6.6.2 Translating and Starting a Program

Figure 6.32 shows the steps required to translate a program from a high-level language into machine language and to start executing that program. First, the high-level code is compiled into assembly code. The assembly code is assembled into machine code in an *object file*. The linker combines the machine code with object code from libraries and other files to produce an entire executable program. In practice, most compilers perform all three steps of compiling, assembling, and linking. Finally, the loader loads

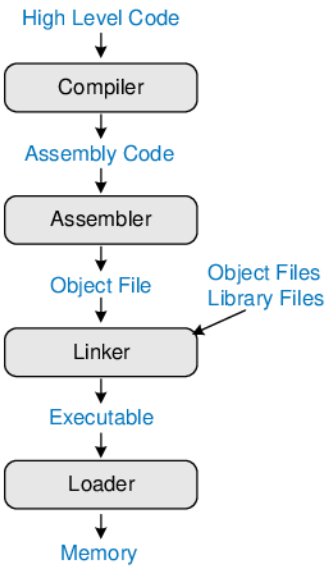


Figure 6.32 Steps for translating and starting a program

the program into memory and starts execution. The remainder of this section walks through these steps for a simple program.

Step 1: Compilation

A compiler translates high-level code into assembly language. Code Example 6.30 shows a simple high-level program with three global

Code Example 6.30 COMPILING A HIGH-LEVEL PROGRAM

High-Level Code

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



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



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

MIPS Assembly Code

```
.data
f:
g:
y:

.text
main:
  addi  $sp, $sp, -4  # make stack frame
  sw    $ra, 0($sp)   # store $ra on stack
  addi  $a0, $0, 2    # $a0 = 2
  sw    $a0, f        # f = 2
  addi  $a1, $0, 3    # $a1 = 3
  sw    $a1, g        # g = 3
  jal   sum           # call sum procedure
  sw    $v0, y        # y = sum (f, g)
  lw    $ra, 0($sp)   # restore $ra from stack
  addi  $sp, $sp, 4   # restore stack pointer
  jr    $ra           # return to operating system

sum:
  add   $v0, $a0, $a1 # $v0 = a + b
  jr    $ra           # return to caller
```

variables and two procedures, along with the assembly code produced by a typical compiler. The `.data` and `.text` keywords are *assembler directives* that indicate where the text and data segments begin. Labels are used for global variables `f`, `g`, and `y`. Their storage location will be determined by the assembler; for now, they are left as symbols in the code.

Step 2: Assembling

The assembler turns the assembly language code into an *object file* containing machine language code. 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 code after the first assembler pass is shown here.

```
0x00400000  main:  addi $sp, $sp, -4
0x00400004         sw   $ra, 0($sp)
0x00400008         addi $a0, $0, 2
0x0040000C         sw   $a0, f
0x00400010         addi $a1, $0, 3
0x00400014         sw   $a1, g
0x00400018         jal  sum
0x0040001C         sw   $v0, y
0x00400020         lw   $ra, 0($sp)
0x00400024         addi $sp, $sp, 4
0x00400028         jr   $ra
0x0040002C  sum:   add  $v0, $a0, $a1
0x00400030         jr   $ra
```

The names and addresses of the symbols are kept in a *symbol table*, as shown in Table 6.4 for this code. The symbol addresses are filled in after the first pass, when the addresses of labels are known. Global variables are assigned storage locations in the global data segment of memory, starting at memory address 0x10000000.

On the second pass through the code, the assembler produces the machine language code. Addresses for the global variables and labels are taken from the symbol table. The machine language code and symbol table are stored in the object file.

Table 6.4 Symbol table

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Step 3: 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 procedures 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.

The job of the linker is to combine all of the object files into one machine language file called the *executable*. 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 addresses of global variables and of labels that are relocated.

In our example, there is only one object file, so no relocation is necessary. Figure 6.33 shows the executable file. It has three sections: the executable file header, the text segment, and the data segment. The executable file header reports the text size (code size) and data size (amount of globally declared data). Both are given in units of bytes. The text segment gives the instructions and the addresses where they are to be stored.

The figure shows the instructions in human-readable format next to the machine code for ease of interpretation, but the executable file includes only machine instructions. The data segment gives the address of each global variable. The global variables are addressed with respect to the base address given by the global pointer, `$gp`. For example, the

Executable file header	Text Size	Data Size	
	0x34 (52 bytes)	0xC (12 bytes)	
Text segment	**Address**	**Instruction**	
	0x00400000	0x23BDFFFC	addi $sp, $sp, –4
	0x00400004	0xAFBF0000	sw $ra, 0 ($sp)
	0x00400008	0x20040002	addi $a0, $0, 2
	0x0040000C	0xAF848000	sw $a0, 0x8000 ($gp)
	0x00400010	0x20050003	addi $a1, $0, 3
	0x00400014	0xAF858004	sw $a1, 0x8004 ($gp)
	0x00400018	0x0C10000B	jal 0x0040002C
	0x0040001C	0xAF828008	sw $v0, 0x8008 ($gp)
	0x00400020	0x8FBF0000	lw $ra, 0 ($sp)
	0x00400024	0x23BD0004	addi $sp, $sp, –4
	0x00400028	0x03E00008	jr $ra
	0x0040002C	0x00851020	add $v0, $a0, $a1
	0x00400030	0x03E00008	jr $ra
Data segment	**Address**	**Data**	
	0x10000000	f	
	0x10000004	g	
	0x10000008	y	

Figure 6.33 Executable

first store instruction, `sw $a0, 0x8000($gp)`, stores the value 2 to the global variable `f`, which is located at memory address 0x10000000. Remember that the offset, 0x8000, is a 16-bit signed number that is sign-extended and added to the base address, `$gp`. So, `$gp` + 0x8000 = 0x10008000 + 0xFFFF8000 = 0x10000000, the memory address of variable `f`.

Step 4: 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 sets `$gp` to 0x10008000 (the middle of the global data segment) and `$sp` to 0x7FFFFFFC (the top of the dynamic data segment), then performs a `jal 0x00400000` to jump to the beginning of the program. Figure 6.34 shows the memory map at the beginning of program execution.

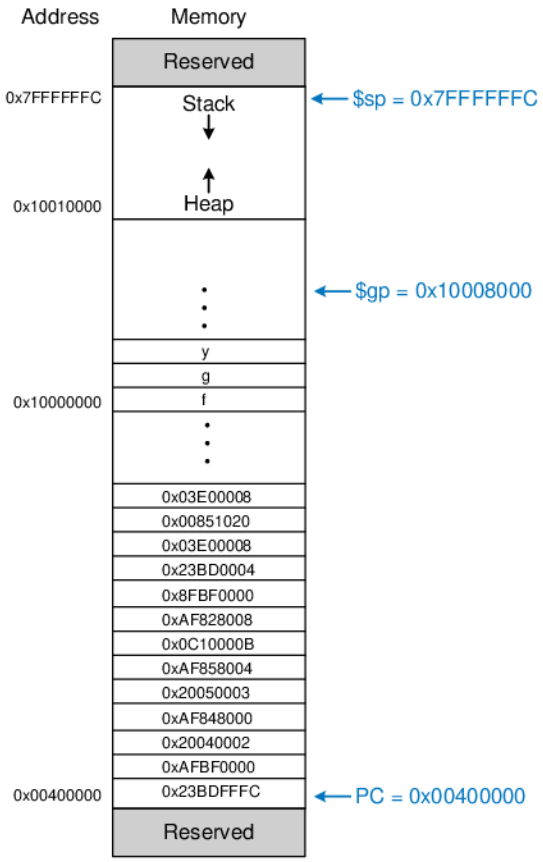


Figure 6.34 Executable loaded in memory

6.7 ODDS AND ENDS*

This section covers a few optional topics that do not fit naturally elsewhere in the chapter. These topics include pseudoinstructions, exceptions, signed and unsigned arithmetic instructions, and floating-point instructions.

6.7.1 Pseudoinstructions

If an instruction is not available in the MIPS instruction set, it is probably because the same operation can be performed using one or more existing MIPS instructions. Remember that MIPS is a reduced instruction set computer (RISC), so the instruction size and hardware complexity are minimized by keeping the number of instructions small.

However, MIPS defines *pseudoinstructions* that are not actually part of the instruction set but are commonly used by programmers and compilers. When converted to machine code, pseudoinstructions are translated into one or more MIPS instructions.

Table 6.5 gives examples of pseudoinstructions and the MIPS instructions used to implement them. For example, the load immediate pseudoinstruction (`li`) loads a 32-bit constant using a combination of `lui` and `ori` instructions. The `multiply` pseudoinstruction (`mul`) provides a three-operand multiply, multiplying two registers and putting the 32 least significant bits of the result into a third register. The no operation pseudoinstruction (`nop`, pronounced "no op") performs no operation. The `PC` is incremented by 4 upon its execution. No other registers or memory values are altered. The machine code for the `nop` instruction is 0x00000000.

Some pseudoinstructions require a temporary register for intermediate calculations. For example, the pseudoinstruction `beq $t2,` $\text{imm}_{15:0}$`, Loop` compares `$t2` to a 16-bit immediate, $\text{imm}_{15:0}$. This pseudoinstruction

Table 6.5 Pseudoinstructions

Pseudoinstruction	Corresponding MIPS Instructions
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`mul $s0, $s1, $s2`	`mult $s1, $s2` `mflo $s0`
`clear $t0`	`add $t0, $0, $0`
`move $s1, $s2`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

Table 6.6 Pseudoinstruction using `$at`

Pseudoinstruction	Corresponding MIPS Instructions
`beq $t2, imm15:0, Loop`	`addi $at, $0, imm15:0` `beq $t2, $at, Loop`

requires a temporary register in which to store the 16-bit immediate. Assemblers use the assembler register, `$at`, for such purposes. Table 6.6 shows how the assembler uses `$at` in converting a pseudoinstruction to real MIPS instructions. We leave it as Exercise 6.31 to implement other pseudoinstructions such as rotate left (`rol`) and rotate right (`ror`).

6.7.2 Exceptions

An *exception* is like an unscheduled procedure call that jumps 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, 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 jumps to code in the *operating system* (*OS*), which may choose to terminate the offending program. Software exceptions are sometimes called *traps*. Other causes of exceptions include division by zero, attempts to read nonexistent memory, hardware malfunctions, debugger breakpoints, and arithmetic overflow (see Section 6.7.3).

The processor records the cause of an exception and the value of the `PC` at the time the exception occurs. It then jumps to the *exception handler* procedure. The exception handler is code (usually in the OS) that examines the cause of the exception and responds appropriately (by reading the keyboard on a hardware interrupt, for example). It then returns to the program that was executing before the exception took place. In MIPS, the exception handler is always located at 0x80000180. When an exception occurs, the processor always jumps to this instruction address, regardless of the cause.

The MIPS architecture uses a special-purpose register, called the `Cause` register, to record the cause of the exception. Different codes are used to record different exception causes, as given in Table 6.7. The exception handler code reads the `Cause` register to determine how to handle the exception. Some other architectures jump to a different exception handler for each different cause instead of using a `Cause` register.

MIPS uses another special-purpose register called the Exception Program Counter (`EPC`) to store the value of the `PC` at the time an exception

Table 6.7 Exception cause codes

Exception	Cause
hardware interrupt	0x00000000
system call	0x00000020
breakpoint/divide by 0	0x00000024
undefined instruction	0x00000028
arithmetic overflow	0x00000030

takes place. The processor returns to the address in `EPC` after handling the exception. This is analogous to using `$ra` to store the old value of the `PC` during a `jal` instruction.

The `EPC` and `Cause` registers are not part of the MIPS register file. The `mfc0` (move from coprocessor 0) instruction copies these and other special-purpose registers into one of the general purpose registers. Coprocessor 0 is called the *MIPS processor control*; it handles interrupts and processor diagnostics. For example, `mfc0 $t0, Cause` copies the `Cause` register into `$t0`.

The `syscall` and `break` instructions cause traps to perform system calls or debugger breakpoints. The exception handler uses the `EPC` to look up the instruction and determine the nature of the system call or breakpoint by looking at the fields of the instruction.

`$k0` and `$k1` are included in the MIPS register set. They are reserved by the OS for exception handling. They do not need to be saved and restored during exceptions.

In summary, an exception causes the processor to jump to the exception handler. The exception handler saves registers on the stack, then uses `mfc0` to look at the cause and respond accordingly. When the handler is finished, it restores the registers from the stack, copies the return address from `EPC` to `$k0` using `mfc0`, and returns using `jr $k0`.

6.7.3 Signed and Unsigned Instructions

Recall that a binary number may be signed or unsigned. The MIPS architecture uses two's complement representation of signed numbers. MIPS has certain instructions that come in signed and unsigned flavors, including addition and subtraction, multiplication and division, set less than, and partial word loads.

Addition and Subtraction

Addition and subtraction are performed identically whether the number is signed or unsigned. However, the interpretation of the results is different.

As mentioned in Section 1.4.6, if two large signed numbers are added together, the result may incorrectly produce the opposite sign. For example, adding the following two huge positive numbers gives a negative

result: 0x7FFFFFFF + 0x7FFFFFFF = 0xFFFFFFFE = −2. Similarly, adding two huge negative numbers gives a positive result, 0x80000001 + 0x80000001 = 0x00000002. This is called arithmetic *overflow*.

The C language ignores arithmetic overflows, but other languages, such as Fortran, require that the program be notified. As mentioned in Section 6.7.2, the MIPS processor takes an exception on arithmetic overflow. The program can decide what to do about the overflow (for example, it might repeat the calculation with greater precision to avoid the overflow), then return to where it left off.

MIPS provides signed and unsigned versions of addition and subtraction. The signed versions are `add`, `addi`, and `sub`. The unsigned versions are `addu`, `addiu`, and `subu`. The two versions are identical except that signed versions trigger an exception on overflow, whereas unsigned versions do not. Because C ignores exceptions, C programs technically use the unsigned versions of these instructions.

Multiplication and Division

Multiplication and division behave differently for signed and unsigned numbers. For example, as an unsigned number, 0xFFFFFFFF represents a large number, but as a signed number it represents −1. Hence, 0xFFFFFFFF × 0xFFFFFFFF would equal 0xFFFFFFFE00000001 if the numbers were unsigned but 0x0000000000000001 if the numbers were signed.

Therefore, multiplication and division come in both signed and unsigned flavors. `mult` and `div` treat the operands as signed numbers. `multu` and `divu` treat the operands as unsigned numbers.

Set Less Than

Set less than instructions can compare either two registers (`slt`) or a register and an immediate (`slti`). Set less than also comes in signed (`slt` and `slti`) and unsigned (`sltu` and `sltiu`) versions. In a signed comparison, 0x80000000 is less than any other number, because it is the most negative two's complement number. In an unsigned comparison, 0x80000000 is greater than 0x7FFFFFFF but less than 0x80000001, because all numbers are positive.

Beware that `sltiu` sign-extends the immediate before treating it as an unsigned number. For example, `sltiu $s0, $s1, 0x8042` compares `$s1` to 0xFFFF8042, treating the immediate as a large positive number.

Loads

As described in Section 6.4.5, byte loads come in signed (`lb`) and unsigned (`lbu`) versions. `lb` sign-extends the byte, and `lbu` zero-extends the byte to fill the entire 32-bit register. Similarly, MIPS provides signed and unsigned half-word loads (`lh` and `lhu`), which load two bytes into the lower half and sign- or zero-extend the upper half of the word.

6.7.4 Floating-Point Instructions

The MIPS architecture defines an optional floating-point coprocessor, known as coprocessor 1. In early MIPS implementations, the floating-point coprocessor was a separate chip that users could purchase if they needed fast floating-point math. In most recent MIPS implementations, the floating-point coprocessor is built in alongside the main processor.

MIPS defines 32 32-bit floating-point registers, `$f0-$f31`. These are separate from the ordinary registers used so far. MIPS supports both single- and double-precision IEEE floating point arithmetic. Double-precision (64-bit) numbers are stored in pairs of 32-bit registers, so only the 16 even-numbered registers (`$f0, $f2, $f4, . . . , $f30`) are used to specify double-precision operations. By convention, certain registers are reserved for certain purposes, as given in Table 6.8.

Floating-point instructions all have an `opcode` of 17 (10001_2). They require both a `funct` field and a `cop` (coprocessor) field to indicate the type of instruction. Hence, MIPS defines the *F-type* instruction format for floating-point instructions, shown in Figure 6.35. Floating-point instructions come in both single- and double-precision flavors. `cop` = 16 (10000_2) for single-precision instructions or 17 (10001_2) for double-precision instructions. Like R-type instructions, F-type instructions have two source operands, `fs` and `ft`, and one destination, `fd`.

Instruction precision is indicated by `.s` and `.d` in the mnemonic. Floating-point arithmetic instructions include addition (`add.s, add.d`), subtraction (`sub.sz, sub.d`), multiplication (`mul.s, mul.d`), and division (`div.s, div.d`) as well as negation (`neg.s, neg.d`) and absolute value (`abs.s, abs.d`).

Table 6.8 MIPS floating-point register set

Name	Number	Use
`$fv0-$fv1`	0, 2	procedure return values
`$ft0-$ft3`	4, 6, 8, 10	temporary variables
`$fa0-$fa1`	12, 14	procedure arguments
`$ft4-$ft5`	16, 18	temporary variables
`$fs0-$fs5`	20, 22, 24, 26, 28, 30	saved variables

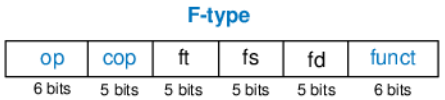


Figure 6.35 F-type machine instruction format

Floating-point branches have two parts. First, a compare instruction is used to set or clear the *floating-point condition flag* (`fpcond`). Then, a conditional branch checks the value of the flag. The compare instructions include equality (`c.seq.s`/`c.seq.d`), less than (`c.lt.s`/`c.lt.d`), and less than or equal to (`c.le.s`/`c.le.d`). The conditional branch instructions are `bc1f` and `bc1t` that branch if `fpcond` is FALSE or TRUE, respectively. Inequality, greater than or equal to, and greater than comparisons are performed with `seq`, `lt`, and `le`, followed by `bc1f`.

Floating-point registers are loaded and stored from memory using `lwc1` and `swc1`. These instructions move 32 bits, so two are necessary to handle a double-precision number.

6.8 REAL-WORLD PERSPECTIVE: IA-32 ARCHITECTURE*

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

The IA-32 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 IA-32 or x86 processors. The Pentium, Core, and Athlon processors are well known IA-32 processors. Section 7.9 describes the evolution of IA-32 microprocessors in more detail.

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 MIPS. As Patterson and Hennessy explain, "this checkered ancestry has led to an architecture that is difficult to explain and impossible to love." However, software compatibility is far more important than technical elegance, so IA-32 has been the *de facto* PC standard for more than two decades. More than 100 million IA-32 processors are sold every year. This huge market justifies more than $5 billion of research and development annually to continue improving the processors.

IA-32 is an example of a *Complex Instruction Set Computer* (*CISC*) architecture. In contrast to RISC architectures such as MIPS, 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.

Table 6.9 Major differences between MIPS and IA-32

Feature	MIPS	IA-32
# of registers	32 general purpose	8, some restrictions on purpose
# of operands	3 (2 source, 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 codes	no	yes
instruction types	simple	simple and complicated
instruction encoding	fixed, 4 bytes	variable, 1–15 bytes

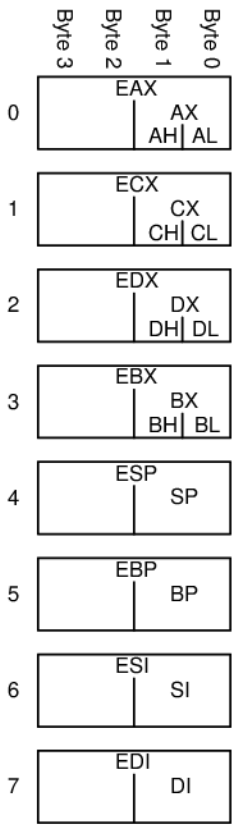


Figure 6.36 IA-32 registers

This section introduces the IA-32 architecture. The goal is not to make you into an IA-32 assembly language programmer, but rather to illustrate some of the similarities and differences between IA-32 and MIPS. We think it is interesting to see how IA-32 works. However, none of the material in this section is needed to understand the rest of the book. Major differences between IA-32 and MIPS are summarized in Table 6.9.

6.8.1 IA-32 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 6.36.

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 MIPS, `ESP` is normally reserved for the stack pointer.

The IA-32 program counter is called `EIP` (the *extended instruction pointer*). Like the MIPS `PC`, it advances from one instruction to the next or can be changed with branch, jump, and subroutine call instructions.

6.8.2 IA-32 Operands

MIPS instructions always act on registers or immediates. Explicit load and store instructions are needed to move data between memory and the registers. In contrast, IA-32 instructions may operate on registers, immediates, or memory. This partially compensates for the small set of registers.

MIPS instructions generally specify three operands: two sources and one destination. IA-32 instructions specify only two operands. The first is a source. The second is both a source and the destination. Hence, IA-32 instructions always overwrite one of their sources with the result. Table 6.10 lists the combinations of operand locations in IA-32. All of the combinations are possible except memory to memory.

Like MIPS, IA-32 has a 32-bit memory space that is byte-addressable. However, IA-32 also supports a much wider variety of memory *addressing modes*. The memory location is specified with any combination of a *base register*, *displacement*, and a *scaled index register*. Table 6.11 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 MIPS base addressing mode for loads and stores. 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.

Table 6.10 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 6.11 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 6.12 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

While MIPS always acts on 32-bit words, IA-32 instructions can operate on 8-, 16-, or 32-bit data. Table 6.12 illustrates these variations.

6.8.3 Status Flags

IA-32, like many CISC architectures, uses *status flags* (also called *condition codes*) to make decisions about branches and to keep track of carries and arithmetic overflow. IA-32 uses a 32-bit register, called `EFLAGS`, that stores the status flags. Some of the bits of the `EFLAGS` register are given in Table 6.13. Other bits are used by the operating system.

The architectural state of an IA-32 processor includes `EFLAGS` as well as the eight registers and the `EIP`.

6.8.4 IA-32 Instructions

IA-32 has a larger set of instructions than MIPS. Table 6.14 describes some of the general purpose instructions. IA-32 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

Table 6.13 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.

Table 6.14 Selected IA-32 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	procedure call	ESP = ESP − 4; MEM[ESP] = EIP; EIP = S
RET	procedure return	EIP = MEM[ESP]; ESP = ESP + 4

Table 6.15 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	

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. The jumps usually follow an instruction, such as the compare instruction (CMP), that sets the flags. Table 6.15 lists some of the conditional jumps and how they depend on the flags set by a prior compare operation.

6.8.5 IA-32 Instruction Encoding

The IA-32 instruction encodings are truly messy, a legacy of decades of piecemeal changes. Unlike MIPS, whose instructions are uniformly 32 bits, IA-32 instructions vary from 1 to 15 bytes, as shown in Figure 6.37.[2] 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

[2] It is possible to construct 17-byte instructions if all the optional fields are used. However, IA-32 places a 15-byte limit on the length of legal instructions.

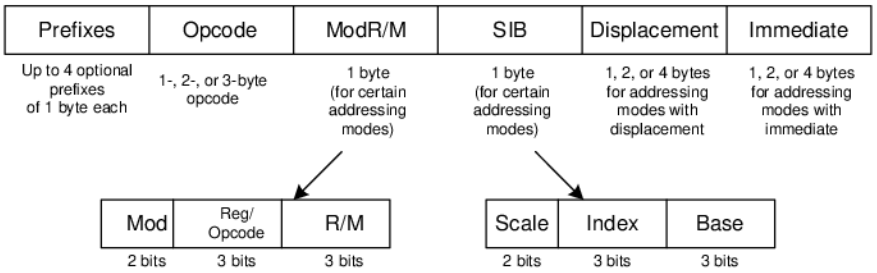


Figure 6.37 IA-32 instruction encodings

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.

MIPS fully specifies the instruction in the `opcode` and `funct` fields of the instruction. IA-32 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 IA-32 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.

IA-32 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.

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 IA-32, 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 IA-32.

The bane of any architecture is to run out of memory capacity. With 32-bit addresses, IA-32 can access 4 GB of memory. This was far more than the largest computers had in 1985, but by the early 2000s 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 IA-32 architecture, the IA-32 Intel Architecture Software Developer's Manual, is freely available on Intel's Web site.

6.8.7 The Big Picture

This section has given a taste of some of the differences between the MIPS RISC architecture and the IA-32 CISC architecture. IA-32 tends to have shorter programs, because a complex instruction is equivalent to a series of simple MIPS instructions and because the instructions are encoded to minimize memory use. However, the IA-32 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, IA-32 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 IA-32 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?

MIPS is a 32-bit architecture because it operates on 32-bit data. The MIPS architecture has 32 general-purpose registers. In principle, almost any register can be used for any purpose. However, by convention, certain registers are reserved for certain purposes, for ease of programming and so that procedures written by different programmers can communicate easily. For example, register 0, `$0`, always holds the constant 0, `$ra` holds the return address after a `jal` instruction, and `$a0-$a3` and `$v0 - $v1` hold the arguments and return value of a procedure. MIPS has a byte-addressable memory system with 32-bit addresses. The memory map was described in Section 6.6.1. Instructions are 32 bits long and must be word aligned. This chapter discussed the most commonly used MIPS 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 Core 2 Duo or AMD Athlon processors in 2006.

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 MIPS architecture of each of the architecture design principles: (1) simplicity favors regularity; (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 MIPS architecture has a register set that consists of 32 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 MIPS 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. Your drawing should be similar to Figure 6.4. Clearly label the byte address corresponding to each data byte value.

Exercise 6.4 Explain how the following program can be used to determine whether a computer is big-endian or little-endian:

```
li $t0, 0xABCD9876
sw $t0, 100($0)
lb $s5, 101($0)
```

Exercise 6.5 Write the following strings using ASCII encoding. Write your final answers in hexadecimal.

(a) SOS

(b) Cool!

(c) (your own name)

Exercise 6.6 Show how the strings in Exercise 6.5 are stored in a byte-addressable memory on (a) a big-endian machine and (b) a little-endian machine starting at memory address 0x1000100C. Use a memory diagram similar to Figure 6.4. Clearly indicate the memory address of each byte on each machine.

Exercise 6.7 Convert the following MIPS assembly code into machine language. Write the instructions in hexadecimal.

```
add  $t0, $s0, $s1
lw   $t0, 0x20($t7)
addi $s0, $0, -10
```

Exercise 6.8 Repeat Exercise 6.7 for the following MIPS assembly code:

```
addi $s0, $0, 73
sw   $t1, -7($t2)
sub  $t1, $s7, $s2
```

Exercise 6.9 Consider I-type instructions.

(a) Which instructions from Exercise 6.8 are I-type instructions?

(b) Sign-extend the 16-bit immediate of each instruction from part (a) so that it becomes a 32-bit number.

Exercise 6.10 Convert the following program from machine language into MIPS assembly language. The numbers on the left are the instruction address 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. `$a0` is the input, and it initially contains a positive number, `n`. `$v0` is the output.

```
0x00400000   0x20080000
0x00400004   0x20090001
0x00400008   0x0089502a
0x0040000c   0x15400003
0x00400010   0x01094020
0x00400014   0x21290002
0x00400018   0x08100002
0x0040001c   0x01001020
```

Exercise 6.11 The `nori` instruction is not part of the MIPS instruction set, because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: `$t0 = $t1 NOR 0xF234`. Use as few instructions as possible.

Exercise 6.12 Implement the following high-level code segments using the `slt` instruction. Assume the integer variables `g` and `h` are in registers `$s0` and `$s1`, respectively.

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

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

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

Exercise 6.13 Write a procedure in a high-level language for `int find42(int array[], int size)`. `size` specifies the number of elements in the array. `array` specifies the base address of the array. The procedure 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 6.14 The high-level procedure `strcpy` copies the character string `x` to the character string `y`.

```
// high-level code
void strcpy(char x[], char y[]) {
  int i = 0;

  while (x[i] != 0) {
    y[i] = x[i];
    i = i + 1;
  }
}
```

(a) Implement the `strcpy` procedure in MIPS assembly code. Use `$s0` for `i`.

(b) Draw a picture of the stack before, during, and after the `strcpy` procedure call. Assume `$sp` = 0x7FFFFF00 just before `strcpy` is called.

Exercise 6.15 Convert the high-level procedure from Exercise 6.13 into MIPS assembly code.

This simple string copy program has a serious flaw: it has no way of knowing that `y` has enough space to receive `x`. If a malicious programmer were able to execute `strcpy` with a long string `x`, 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 6.16 Each number in the *Fibonacci series* is the sum of the previous two numbers. Table 6.16 lists the first few numbers in the series, *fib(n)*.

Table 6.16 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	. . .

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

(b) Write a procedure 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 procedure of part (b) into MIPS assembly code. Add comments after every line of code that explain clearly what it does. Use the SPIM simulator to test your code on fib(9).

Exercise 6.17 Consider the MIPS assembly code below. `proc1`, `proc2`, and `proc3` are non-leaf procedures. `proc4` is a leaf procedure. The code is not shown for each procedure, but the comments indicate which registers are used within each procedure.

```
0x00401000    proc1:   . . .          # proc1 uses $s0 and $s1
0x00401020             jal proc2
 . . .
0x00401100    proc2:   . . .          # proc2 uses $s2 - $s7
0x0040117C             jal proc3
 . . .
0x00401400    proc3:   . . .          # proc3 uses $s1 - $s3
0x00401704             jal proc4
 . . .
0x00403008    proc4:   . . .          # proc4 uses no preserved
                                      # registers
0x00403118             jr $ra
```

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

(b) Sketch the stack after `proc4` is called. Clearly indicate which registers are stored where on the stack. Give values where possible.

Exercise 6.18 Ben Bitdiddle is trying to compute the function $f(a, b) = 2a + 3b$ for nonnegative b. He goes overboard in the use of procedure calls and recursion and produces the following high-level code for procedures `f` and `f2`.

```
// high-level code for procedures f and f2
int f(int a, int b) {
  int j;
  j = a;
  return j + a + f2(b);
}

int f2(int x)
{
  int k;
  k = 3;
  if (x == 0) return 0;
  else return k + f2(x-1);
}
```

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

```
# MIPS assembly code
# f: $a0 = a, $a1 = b, $s0 = j f2: $a0 = x, $s0 = k

0x00400000  test:addi  $a0, $0, 5     # $a0 = 5 (a = 5)
0x00400004       addi  $a1, $0, 3     # $a1 = 3 (b = 3)
0x00400008       jal   f              # call f(5,3)
0x0040000c  loop:j     loop           # and loop forever

0x00400010  f:   addi  $sp, $sp, -16  # make room on the stack
                                      # for $s0, $a0, $a1, and $ra
0x00400014       sw    $a1, 12($sp)   # save $a1 (b)
0x00400018       sw    $a0, 8($sp)    # save $a0 (a)
0x0040001c       sw    $ra, 4($sp)    # save $ra
0x00400020       sw    $s0, 0($sp)    # save $s0
0x00400024       add   $s0, $a0, $0   # $s0 = $a0 (j = a)
0x00400028       add   $a0, $a1, $0   # place b as argument for f2
0x0040002c       jal   f2             # call f2(b)
0x00400030       lw    $a0, 8($sp)    # restore $a0 (a) after call
0x00400034       lw    $a1, 12($sp)   # restore $a1 (b) after call
0x00400038       add   $v0, $v0, $s0  # $v0 = f2(b) + j
0x0040003c       add   $v0, $v0, $a0  # $v0 = (f2(b) + j) + a
0x00400040       lw    $s0, 0($sp)    # restore $s0
0x00400044       lw    $ra, 4($sp)    # restore $ra
0x00400048       addi  $sp, $sp, 16   # restore $sp (stack pointer)
0x0040004c       jr    $ra            # return to point of call

0x00400050  f2:  addi  $sp, $sp, -12  # make room on the stack for
                                      # $s0, $a0, and $ra
0x00400054       sw    $a0, 8($sp)    # save $a0 (x)
0x00400058       sw    $ra, 4($sp)    # save return address
0x0040005c       sw    $s0, 0($sp)    # save $s0
0x00400060       addi  $s0, $0, 3     # k = 3
```

```
0x00400064        bne   $a0, $0, else  # x = 0?
0x00400068        addi  $v0, $0, 0     # yes: return value should be 0
0x0040006c        j     done           # and clean up
0x00400070  else: addi  $a0, $a0, -1   # no: $a0 = $a0 - 1 (x = x - 1)
0x00400074        jal   f2             # call f2(x - 1)
0x00400078        lw    $a0, 8($sp)    # restore $a0 (x)
0x0040007c        add   $v0, $v0, $s0  # $v0 = f2(x - 1) + k
0x00400080  done: lw    $s0, 0($sp)    # restore $s0
0x00400084        lw    $ra, 4($sp)    # restore $ra
0x00400088        addi  $sp, $sp, 12   # restore $sp
0x0040008c        jr    $ra            # return to point of call
```

You will probably find it useful to make drawings of the stack similar to the one in Figure 6.26 to help you answer the following questions.

(a) If the code runs starting at `test`, what value is in `$v0` when the program gets to `loop`? Does his program correctly compute $2a + 3b$?

(b) Suppose Ben deletes the instructions at addresses 0x0040001C and 0x00400040 that save and restore `$ra`. Will the program (1) enter an infinite loop but not crash; (2) crash (cause the stack to grow beyond the dynamic data segment or the PC to jump to a location outside the program); (3) produce an incorrect value in `$v0` when the program returns to loop (if so, what value?), or (4) run correctly despite the deleted lines?

(c) Repeat part (b) when the instructions at the following instruction addresses are deleted:

(i) 0x00400018 and 0x00400030 (instructions that save and restore `$a0`)

(ii) 0x00400014 and 0x00400034 (instructions that save and restore `$a1`)

(iii) 0x00400020 and 0x00400040 (instructions that save and restore `$s0`)

(iv) 0x00400050 and 0x00400088 (instructions that save and restore `$sp`)

(v) 0x0040005C and 0x00400080 (instructions that save and restore `$s0`)

(vi) 0x00400058 and 0x00400084 (instructions that save and restore `$ra`)

(vii) 0x00400054 and 0x00400078 (instructions that save and restore `$a0`)

Exercise 6.19 Convert the following `beq`, `j`, and `jal` assembly instructions into machine code. Instruction addresses are given to the left of each instruction.

(a)

```
0x00401000            beq $t0, $s1, Loop
0x00401004             . . .
0x00401008             . . .
0x0040100C     Loop:   . . .
```

(b)

```
0x00401000            beq $t7, $s4, done
 . . .                . . .
0x00402040     done:  . . .
```

(c)

```
0x0040310C     back:  . . .
 . . .                . . .
0x00405000            beq $t9, $s7, back
```

(d)

```
0x00403000            jal proc
 . . .                . . .
0x0041147C     proc:  . . .
```

(e)

```
0x00403004     back:  . . .
 . . .                . . .
0x0040400C     j      back
```

Exercise 6.20 Consider the following MIPS assembly language snippet. The numbers to the left of each instruction indicate the instruction address.

```
0x00400028        add  $a0, $a1, $0
0x0040002c        jal  f2
0x00400030 f1:    jr   $ra
0x00400034 f2:    sw   $s0, 0($s2)
0x00400038        bne  $a0, $0, else
0x0040003c        j    f1
0x00400040 else:  addi $a0, $a0, -1
0x00400044        j    f2
```

(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 6.21 Consider the following C code snippet.

```
// C code
   void set_array(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 MIPS assembly language. Use `$s0` to hold the variable `i`. Be sure to handle the stack pointer appropriately. The array is stored on the stack of the `set_array` procedure (see Section 6.4.6).

(b) Assume `set_array` is the first procedure called. Draw the status of the stack before calling `set_array` and during each procedure call. Indicate the names of registers and variables stored on the stack and mark the location of `$sp`.

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

Exercise 6.22 Consider the following high-level procedure.

```
// high-level 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 procedure `f` into MIPS assembly language. Pay particular attention to properly saving and restoring registers across procedure calls and using the MIPS preserved register conventions. Clearly comment your code. You can use the MIPS `mult`, `mfhi`, and `mflo` instructions. The procedure starts at instruction address 0x00400100. Keep local variable `b` in `$s0`.

(b) Step through your program from part (a) by hand for the case of `f(2, 4)`. Draw a picture of the stack similar to the one in Figure 6.26(c). Write the register name and data value stored at each location in the stack and keep track of the stack pointer value (`$sp`). You might also find it useful to keep track of the values in `$a0`, `$a1`, `$v0`, and `$s0` throughout execution. Assume that when `f` is called, `$s0` = 0xABCD and `$ra` = 0x400004. What is the final value of `$v0`?

Exercise 6.23 What is the range of instruction addresses to which conditional branches, such as `beq` and `bne`, can branch in MIPS? Give your answer in number of instructions relative to the conditional branch instruction.

Exercise 6.24 The following questions examine the limitations of the jump instruction, `j`. Give your answer in number of instructions relative to the jump instruction.

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

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

(c) In the worst case, how far can the jump instruction (`j`) jump backward (to lower addresses)? Explain.

(d) In the best case, how far can the jump instruction (`j`) jump backward? Explain.

Exercise 6.25 Explain why it is advantageous to have a large address field, `addr`, in the machine format for the jump instructions, `j` and `jal`.

Exercise 6.26 Write assembly code that jumps to the instruction 64 Minstructions from the first instruction. Recall that 1 Minstruction = 2^{20} instructions = 1,048,576 instructions. Assume that your code begins at address 0x00400000. Use a minimum number of instructions.

Exercise 6.27 Write a procedure in high-level code that takes a ten-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 MIPS assembly code. Comment all your code and use a minimum number of instructions.

Exercise 6.28 Consider two strings: `string1` and `string2`.

(a) Write high-level code for a procedure called `concat` that concatenates (joins together) the two strings: `void concat(char[] string1, char[] string2, char[] stringconcat)`. The procedure 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 procedure from part (a) into MIPS assembly language.

Exercise 6.29 Write a MIPS assembly program that adds two positive single-precision floating point numbers held in `$s0` and `$s1`. Do not use any of the MIPS floating-point instructions. You need not worry about any of the encodings that are reserved for special purposes (e.g., 0, NANs, INF) or numbers that overflow or underflow. Use the SPIM simulator to test your code. You will need to manually set the values of `$s0` and `$s1` to test your code. Demonstrate that your code functions reliably.

Exercise 6.30 Show how the following MIPS program would be loaded into memory and executed.

```
# MIPS assembly code
main:
  lw $a0, x
  lw $a1, y
  jal diff
  jr $ra
diff:
  sub $v0, $a0, $a1
  jr $ra
```

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

(b) Draw the symbol table showing the labels and their addresses.

(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.

Exercise 6.31 Show the MIPS instructions that implement the following pseudoinstructions. You may use the assembler register, `$at`, but you may not corrupt (overwrite) any other registers.

(a) `beq $t1,` $\text{imm}_{31:0}$`, L`

(b) `ble $t3, $t5, L`

(c) `bgt $t3, $t5, L`

(d) `bge $t3, $t5, L`

(e) `addi $t0, $2,` $\text{imm}_{31:0}$

(f) `lw $t5,` $\text{imm}_{31:0}$`($s0)`

(g) `rol $t0, $t1, 5` (rotate `$t1` left by 5 and put the result in `$t0`)

(h) `ror $s4, $t6, 31` (rotate `$t6` right by 31 and put the result in `$s4`)

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 MIPS assembly code for swapping the contents of two registers, `$t0` and `$t1`. You may not use any other registers.

Question 6.2 Suppose you are given an array of both positive and negative integers. Write MIPS 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 `$a0` and `$a1`, respectively. Your code should place the resulting subset of the array starting at base address `$a2`. Write code that runs as fast as possible.

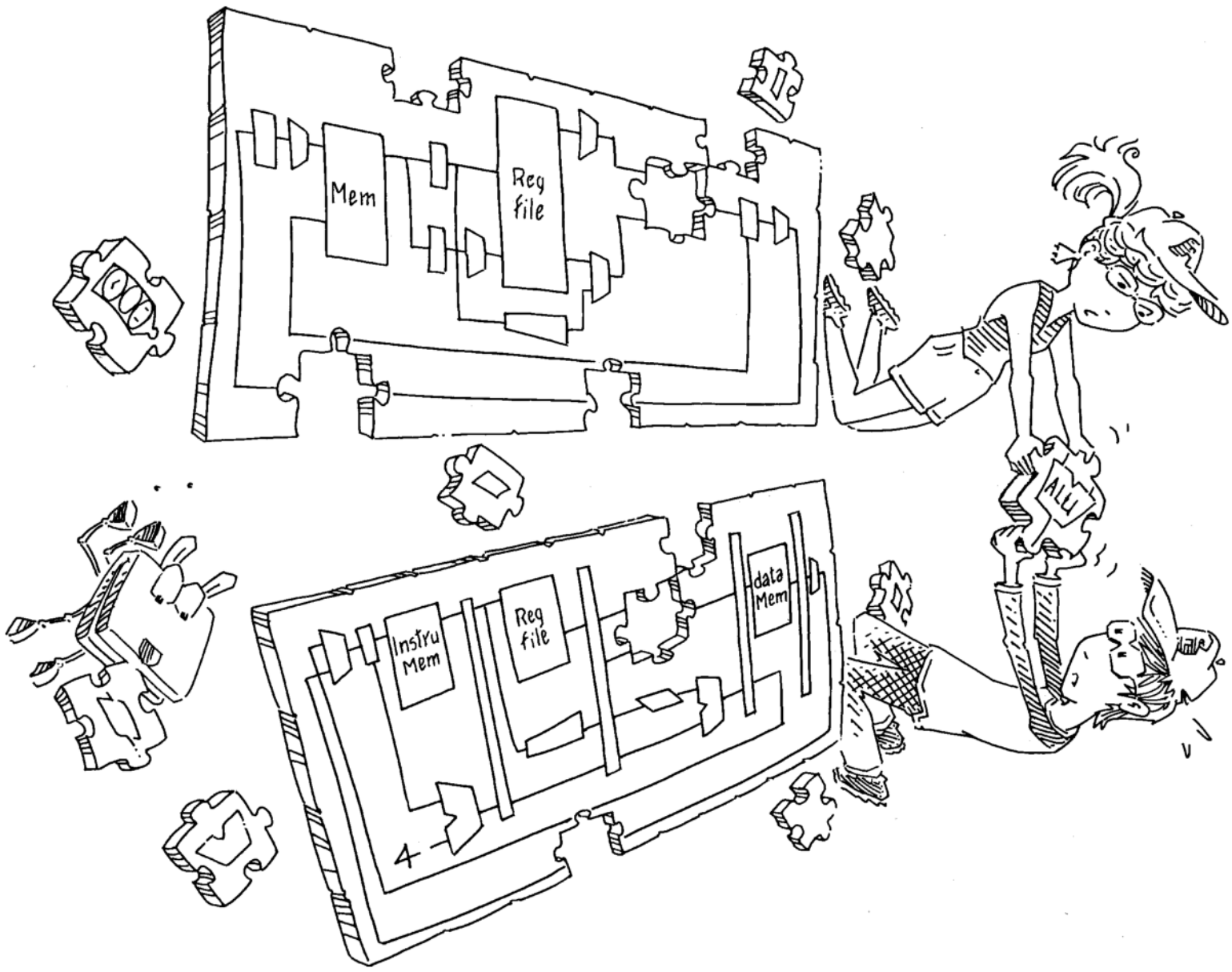
Question 6.3 You are given an array that holds a C string. The string forms a sentence. Design an algorithm for reversing the words in the sentence and storing the new sentence back in the array. Implement your algorithm using MIPS assembly code.

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

Question 6.5 Write MIPS assembly code to reverse the bits in a register. Use as few instructions as possible. Assume the register of interest is `$t3`.

Question 6.6 Write MIPS assembly code to test whether overflow occurs when `$t2` and `$t3` 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 MIPS assembly code.


Mem
Reg file
ALU
Instru Mem
Reg file
data Mem
4

Microarchitecture

7

7.1 INTRODUCTION

In this chapter, you will learn how to piece together a MIPS 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 MIPS architecture, which specifies the programmer's view of the MIPS 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 MIPS, 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 will design three different microarchitectures in this chapter to illustrate the trade-offs.

This chapter draws heavily on David Patterson and John Hennessy's classic MIPS designs in their text *Computer Organization and Design*. They have generously shared their elegant designs, which have the virtue of illustrating a real commercial architecture while being relatively simple and easy to understand.

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 MIPS processor consists

David Patterson was the first in his family to graduate from college (UCLA, 1969). He has been a professor of computer science at UC Berkeley since 1977, where he coinvented RISC, the Reduced Instruction Set Computer. In 1984, he developed the SPARC architecture used by Sun Microsystems. He is also the father of *RAID* (*Redundant Array of Inexpensive Disks*) and *NOW* (*Network of Workstations*).

John Hennessy is president of Stanford University and has been a professor of electrical engineering and computer science there since 1977. He coinvented RISC. He developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems. As of 2004, more than 300 million MIPS microprocessors have been sold.

In their copious free time, these two modern paragons write textbooks for recreation and relaxation.

of the program counter and the 32 registers. Any MIPS 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 additional *nonarchitectural state* to either simplify the logic or improve performance; we will point this out as it arises.

To keep the microarchitectures easy to understand, we consider only a subset of the MIPS instruction set. Specifically, we handle the following instructions:

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

After building the microarchitectures with these instructions, we extend them to handle `addi` and `j`. 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 will divide our microarchitectures into two interacting parts: the *datapath* and the *control*. The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers. MIPS is a 32-bit architecture, so we will use a 32-bit datapath. The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

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

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 5-bit address busses on the register file. Narrow blue lines are used to indicate

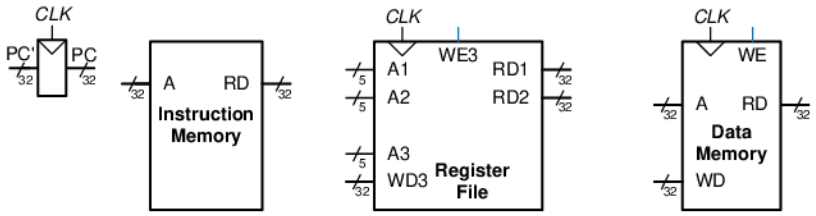


Figure 7.1 State elements of MIPS processor

control signals, such as the register file write enable. We will 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.

The *program counter* is an ordinary 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 32-element × 32-bit *register file* has two read ports and one write port. The read ports take 5-bit address inputs, *A1* and *A2*, each specifying one of $2^5 = 32$ registers as source operands. They read the 32-bit register values onto read data outputs *RD1* and *RD2*, respectively. The write port takes a 5-bit address input, *A3*; a 32-bit write data input, *WD*; a write enable input, *WE3*; and a clock. If the write enable is 1, the register file writes the data into the specified register on the rising edge of the clock.

The *data memory* has a single read/write port. If the write enable, *WE*, is 1, it writes data *WD* into address *A* on the rising edge of the clock. If the write enable is 0, it reads address *A* onto *RD*.

The instruction memory, register file, and data memory are all read *combinationally*. In other words, if the address changes, 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 sometime 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

Resetting the PC

At the very least, the program counter must have a reset signal to initialize its value when the processor turns on. MIPS processors initialize the PC to `0xBFC00000` on reset and begin executing code to start up the operating system (OS). The OS then loads an application program at `0x00400000` and begins executing it. For simplicity in this chapter, we will reset the PC to `0x00000000` and place our programs there instead.

[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.

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.

7.1.3 MIPS Microarchitectures

In this chapter, we develop three microarchitectures for the MIPS processor 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.

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 several 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 *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. 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 get 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. Each year, CMOS processes can pack more transistors on a chip for the same amount of money, and processors take advantage

of these additional transistors to deliver more performance. 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, Intel and Advanced Micro Devices (AMD) both sell compatible microprocessors conforming to the IA-32 architecture. Intel Pentium III and Pentium 4 microprocessors were largely advertised according to clock frequency in the late 1990s and early 2000s, because Intel offered higher clock frequencies than its competitors. However, Intel's main competitor, AMD, sold Athlon microprocessors that executed programs faster than Intel's chips at the same clock frequency. What is a consumer 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 haven't written your program yet or if somebody else who doesn't 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.

The execution time of a program, measured in seconds, is given by Equation 7.1.

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

The number of instructions in a program depends on the processor architecture. Some architectures have complicated instructions that do more work per instruction, thus reducing the number of instructions in a program. However, these complicated instructions are often slower to execute in hardware. The number of instructions also depends enormously on the cleverness of the programmer. For the purposes of this chapter, we will assume that we are executing known programs on a MIPS processor, so the number of instructions for each program is constant, independent of the microarchitecture.

The number of cycles per instruction, often called *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 will 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 much 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.

There are many other factors that 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 doesn't 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 MIPS 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 carried out by the datapath at any given time. The controller 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), while the hardware that has already been studied is shown in gray.

The program counter (PC) register 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*.

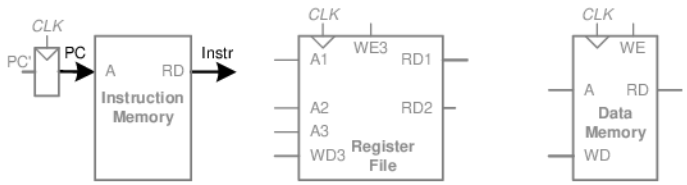


Figure 7.2 Fetch instruction from memory

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

For a `lw` instruction, the next step is to read the source register containing the base address. This register is specified in the `rs` field of the instruction, $Instr_{25:21}$. These bits of the instruction are connected to the address input of one of the register file read ports, *A1*, as shown in Figure 7.3. The register file reads the register value onto *RD1*.

The `lw` instruction also requires an offset. The offset is stored in the immediate field of the instruction, $Instr_{15:0}$. Because the 16-bit immediate might be either positive or negative, it must be sign-extended to 32 bits, as shown in Figure 7.4. The 32-bit sign-extended value is called *SignImm*. Recall from Section 1.4.6 that sign extension simply copies the sign bit (most significant bit) of a short input into all of the upper bits of the longer output. Specifically, $SignImm_{15:0} = Instr_{15:0}$ and $SignImm_{31:16} = Instr_{15}$.

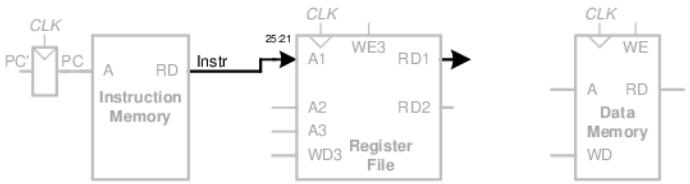


Figure 7.3 Read source operand from register file

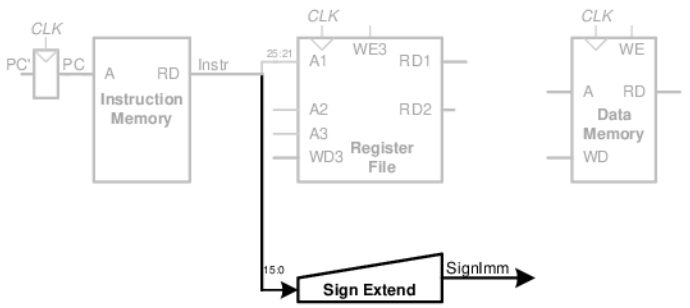


Figure 7.4 Sign-extend the immediate

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 sign-extended immediate. The ALU can perform many operations, as was described in Section 5.2.4. The 3-bit *ALUControl* signal specifies the operation. The ALU generates a 32-bit *ALUResult* and a *Zero* flag, that indicates whether *ALUResult* == 0. For a `lw` instruction, the *ALUControl* signal should be set to 010 to add the base address and offset. *ALUResult* is sent to the data memory as the address for the load instruction, as shown in Figure 7.5.

The data is read from the data memory onto the *ReadData* bus, then written back to the destination register in the register file at the end of the cycle, as shown in Figure 7.6. Port 3 of the register file is the

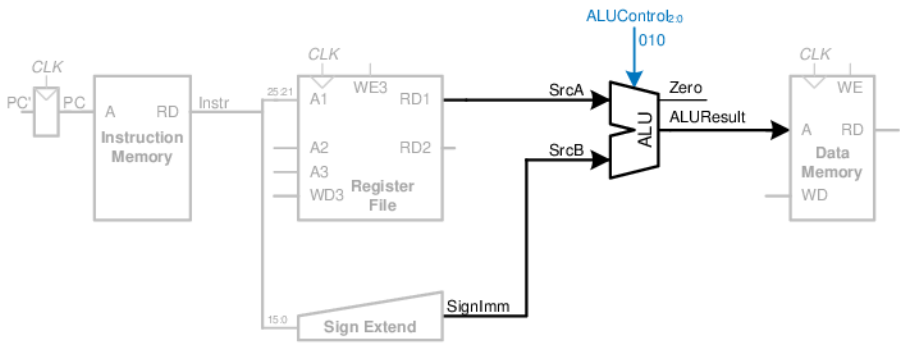


Figure 7.5 Compute memory address

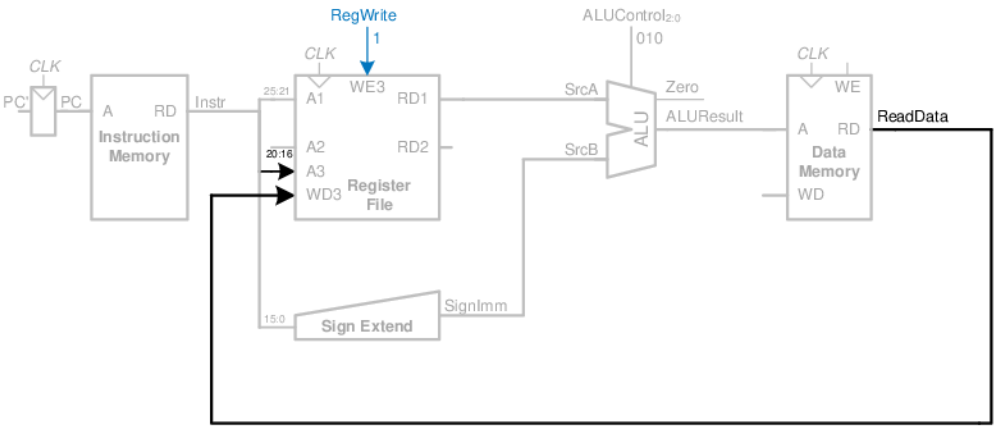


Figure 7.6 Write data back to register file

write port. The destination register for the `lw` instruction is specified in the `rt` field, $Instr_{20:16}$, 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 a `lw` instruction so that the data value is written into the register file. The write takes place on the rising edge of the clock at the end of the cycle.

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 another adder to 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 `lw` instruction.

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

The `sw` instruction also reads a second register from the register file and writes it to the data memory. Figure 7.8 shows the new connections for this function. The register is specified in the `rt` field, $Instr_{20:16}$. These bits of the instruction are connected to the second register file read port, *A2*. The register value is read onto the *RD2* port. It is connected to the write data port of the data memory. The write enable port of the data memory, *WE*, is controlled by *MemWrite*. For a `sw` instruction, *MemWrite* = 1, to write the data to memory; *ALUControl* = 010, to add the base address

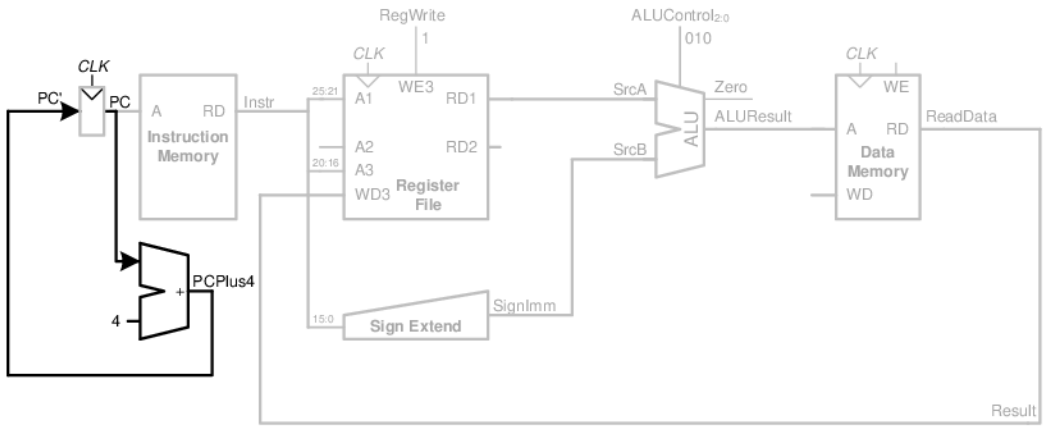


Figure 7.7 Determine address of next instruction for PC

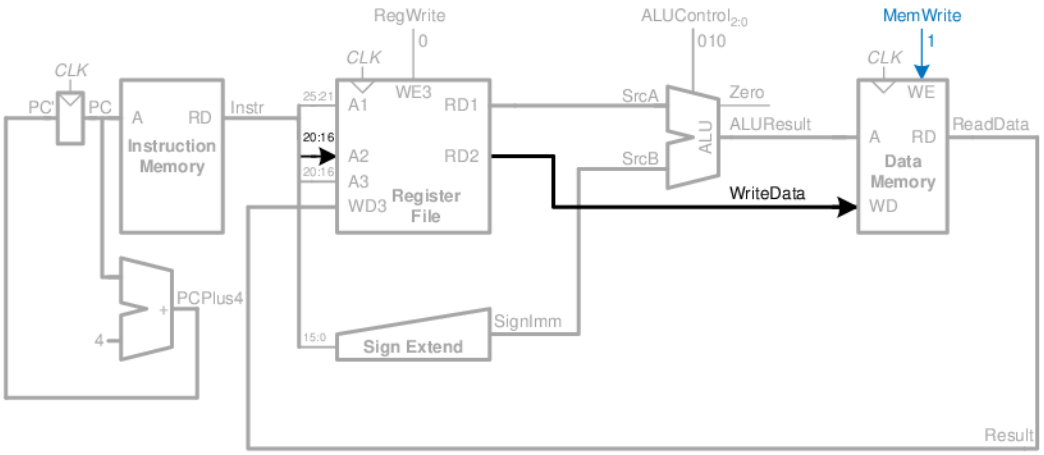


Figure 7.8 **Write data to memory for `sw` instruction**

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.

Next, consider extending the datapath to handle the R-type instructions `add`, `sub`, `and`, `or`, and `slt`. All of these instructions read two registers from the register file, perform some ALU operation on them, and write the result back to a third register file. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware, using different *ALUControl* signals.

Figure 7.9 shows the enhanced datapath handling R-type instructions. The register file reads two registers. The ALU performs an operation on these two registers. In Figure 7.8, the ALU always received its *SrcB* operand from the sign-extended immediate (*SignImm*). Now, we add a multiplexer to choose *SrcB* from either the register file *RD2* port or *SignImm*.

The multiplexer is controlled by a new signal, *ALUSrc*. *ALUSrc* is 0 for R-type instructions to choose *SrcB* from the register file; it is 1 for `lw` and `sw` to choose *SignImm*. This principle of enhancing the datapath's capabilities by adding a multiplexer to choose inputs from several possibilities is extremely useful. Indeed, we will apply it twice more to complete the handling of R-type instructions.

In Figure 7.8, the register file always got its write data from the data memory. However, R-type instructions write the *ALUResult* to the register file. Therefore, we add another multiplexer to choose between *ReadData* and *ALUResult*. We call its output *Result*. This multiplexer is controlled by another new signal, *MemtoReg*. *MemtoReg* is 0

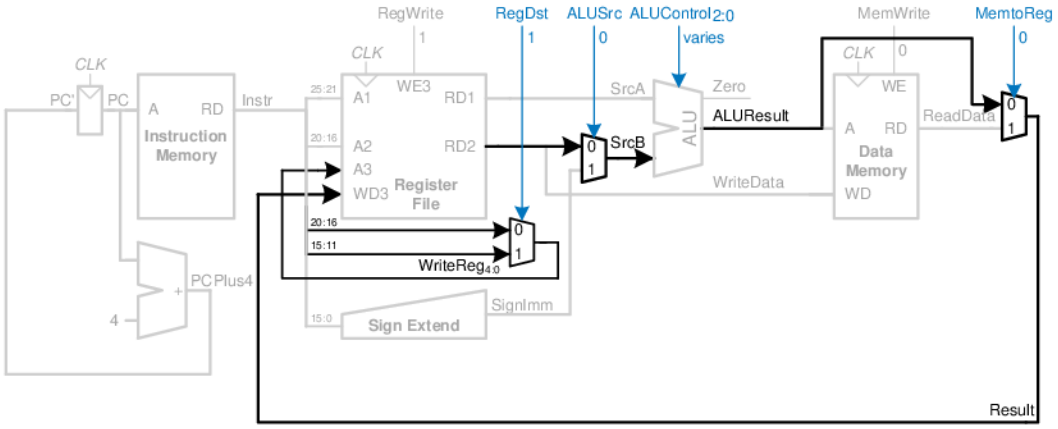


Figure 7.9 Datapath enhancements for R-type instruction

for R-type instructions to choose *Result* from the *ALUResult*; it is 1 for `lw` to choose *ReadData*. We don't care about the value of *MemtoReg* for `sw`, because `sw` does not write to the register file.

Similarly, in Figure 7.8, the register to write was specified by the `rt` field of the instruction, $Instr_{20:16}$. However, for R-type instructions, the register is specified by the `rd` field, $Instr_{15:11}$. Thus, we add a third multiplexer to choose *WriteReg* from the appropriate field of the instruction. The multiplexer is controlled by *RegDst*. *RegDst* is 1 for R-type instructions to choose *WriteReg* from the `rd` field, $Instr_{15:11}$; it is 0 for `lw` to choose the `rt` field, $Instr_{20:16}$. We don't care about the value of *RegDst* for `sw`, because `sw` does not write to the register file.

Finally, let us extend the datapath to handle `beq`. `beq` compares two registers. If they are equal, it takes the branch by adding the branch offset to the program counter. Recall that the offset is a positive or negative number, stored in the `imm` field of the instruction, $Instr_{31:26}$. The offset indicates the number of instructions to branch past. Hence, the immediate must be sign-extended and multiplied by 4 to get the new program counter value: $PC' = PC + 4 + SignImm \times 4$.

Figure 7.10 shows the datapath modifications. The next *PC* value for a taken branch, *PCBranch*, is computed by shifting *SignImm* left by 2 bits, then adding it to *PCPlus4*. The left shift by 2 is an easy way to multiply by 4, because a shift by a constant amount involves just wires. The two registers are compared by computing $SrcA - SrcB$ using the ALU. If *ALUResult* is 0, as indicated by the *Zero* flag from the ALU, the registers are equal. We add a multiplexer to choose PC' from either *PCPlus4* or *PCBranch*. *PCBranch* is selected if the instruction is

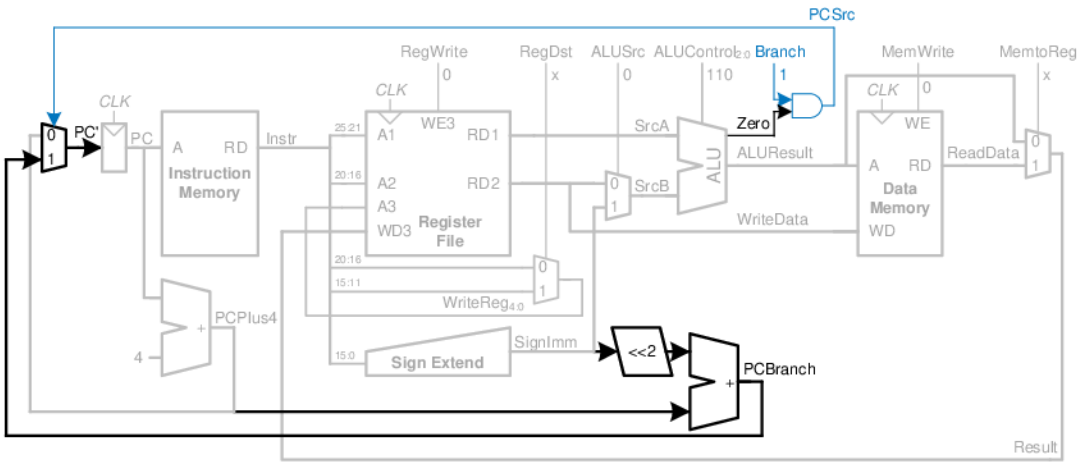


Figure 7.10 Datapath enhancements for `beq` instruction

a branch and the *Zero* flag is asserted. Hence, *Branch* is 1 for `beq` and 0 for other instructions. For `beq`, *ALUControl* = 110, so the ALU performs a subtraction. *ALUSrc* = 0 to choose *SrcB* from the register file. *RegWrite* and *MemWrite* are 0, because a branch does not write to the register file or memory. We don't care about the values of *RegDst* and *MemtoReg*, because the register file is not written.

This completes the design of the single-cycle MIPS 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

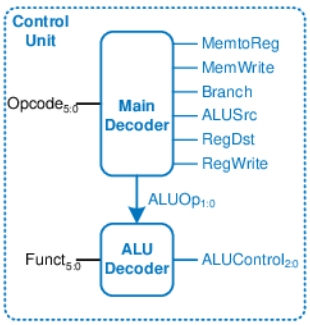


Figure 7.12 Control unit internal structure

The control unit computes the control signals based on the `opcode` and `funct` fields of the instruction, $Instr_{31:26}$ and $Instr_{5:0}$. Figure 7.11 shows the entire single-cycle MIPS processor with the control unit attached to the datapath.

Most of the control information comes from the `opcode`, but R-type instructions also use the `funct` field to determine the ALU operation. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic, as shown in Figure 7.12. The *main decoder* computes most of the outputs from the `opcode`. It also determines a 2-bit *ALUOp* signal. The ALU decoder uses this *ALUOp* signal in conjunction with the `funct` field to compute *ALUControl*. The meaning of the *ALUOp* signal is given in Table 7.1.

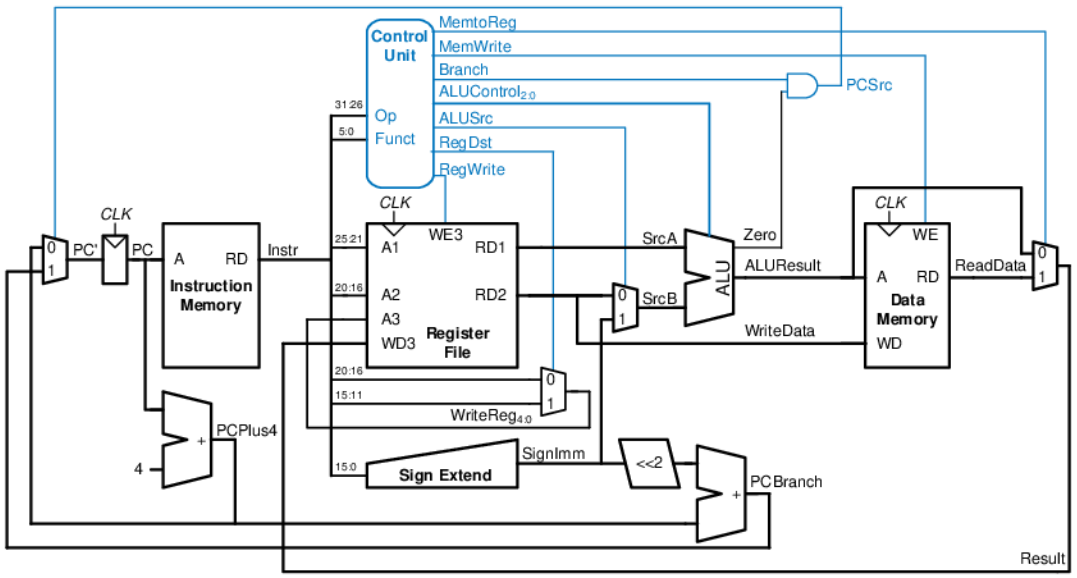


Figure 7.11 Complete single-cycle MIPS processor

Table 7.1 *ALUOp* encoding

ALUOp	Meaning
00	add
01	subtract
10	look at `funct` field
11	n/a

Table 7.2 is a truth table for the ALU decoder. Recall that the meanings of the three *ALUControl* signals were given in Table 5.1. Because *ALUOp* is never 11, the truth table can use don't care's X1 and 1X instead of 01 and 10 to simplify the logic. When *ALUOp* is 00 or 01, the ALU should add or subtract, respectively. When *ALUOp* is 10, the decoder examines the `funct` field to determine the *ALUControl*. Note that, for the R-type instructions we implement, the first two bits of the `funct` field are always 10, so we may ignore them to simplify the decoder.

The control signals for each instruction were described as we built the datapath. Table 7.3 is a truth table for the main decoder that summarizes the control signals as a function of the opcode. All R-type instructions use the same main decoder values; they differ only in the

Table 7.2 ALU decoder truth table

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

Table 7.3 Main decoder truth table

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

ALU decoder output. Recall that, for instructions that do not write to the register file (e.g., sw and beq), the *RegDst* and *MemtoReg* control signals are don't cares (X); the address and data to the register write port do not matter because *RegWrite* is not asserted. The logic for the decoder can be designed using your favorite techniques for combinational logic design.

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 or instruction.

Solution: Figure 7.13 illustrates the control signals and flow of data during execution of the or instruction. The PC points to the memory location holding the instruction, and the instruction memory fetches this instruction.

The main flow of data through the register file and ALU is represented with a dashed blue line. The register file reads the two source operands specified by $Instr_{25:21}$ and $Instr_{20:16}$. *SrcB* should come from the second port of the register

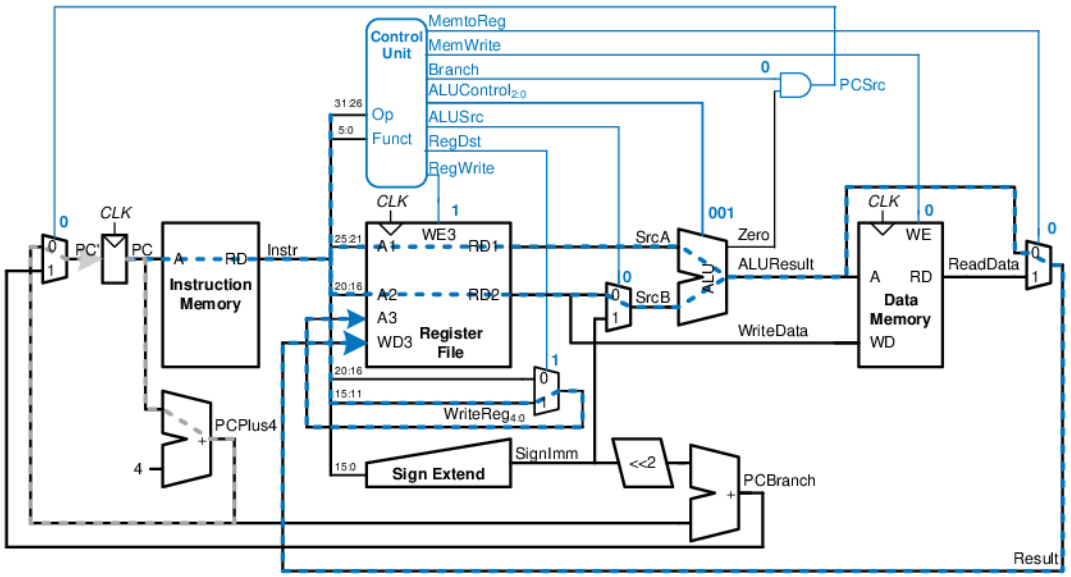


Figure 7.13 Control signals and data flow while executing `or` instruction

file (not *SignImm*), so *ALUSrc* must be 0. `or` is an R-type instruction, so *ALUOp* is 10, indicating that *ALUControl* should be determined from the `funct` field to be 001. *Result* is taken 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 selection of the destination register is also shown with a dashed blue line. The destination register is specified in the `rd` field, $Instr_{15:11}$, so *RegDst* = 1.

The updating of the PC is shown with the dashed gray line. The instruction is not a branch, so *Branch* = 0 and, hence, *PCSrc* is also 0. The PC gets its next value from *PCPlus4*.

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 sign-extended and data is read from memory, but these values do not influence the next state of the system.

7.3.3 More Instructions

We have considered a limited subset of the full MIPS instruction set. Adding support for the `addi` and `j` instructions illustrates the principle of how to handle new instructions and also gives us a sufficiently rich instruction set to write many interesting programs. We will see that

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

Example 7.2 `addi` INSTRUCTION

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

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

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

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

Example 7.3 `j` INSTRUCTION

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

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

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

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

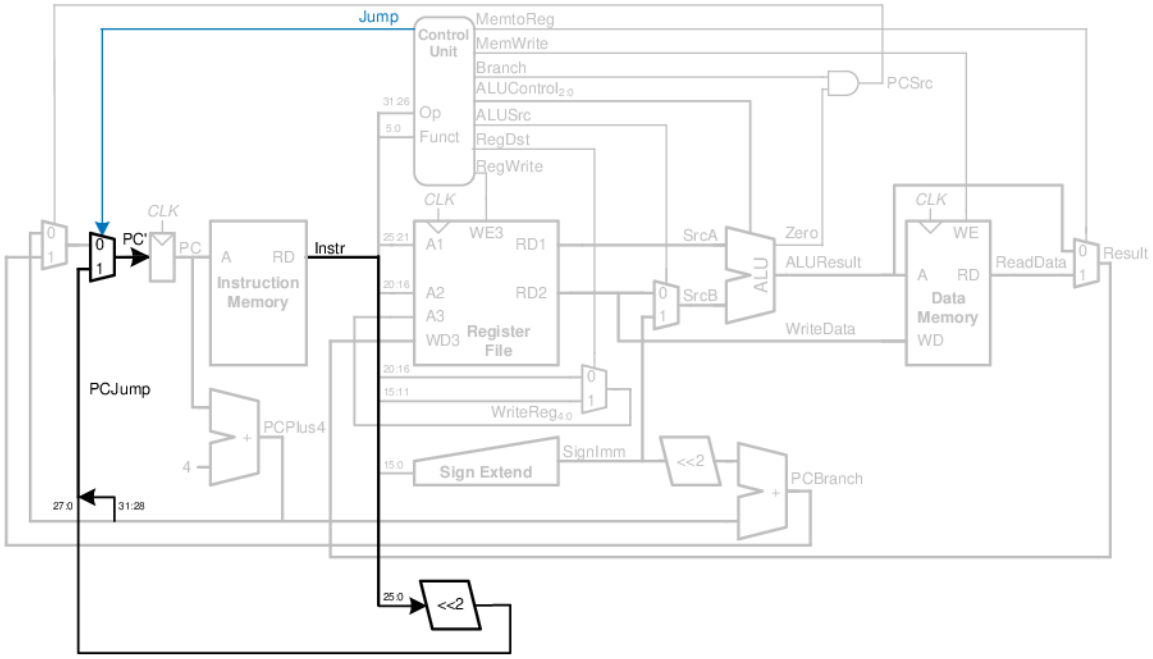


Figure 7.14 Single-cycle MIPS datapath enhanced to support the `j` instruction

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

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

7.3.4 Performance Analysis

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

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

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

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

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

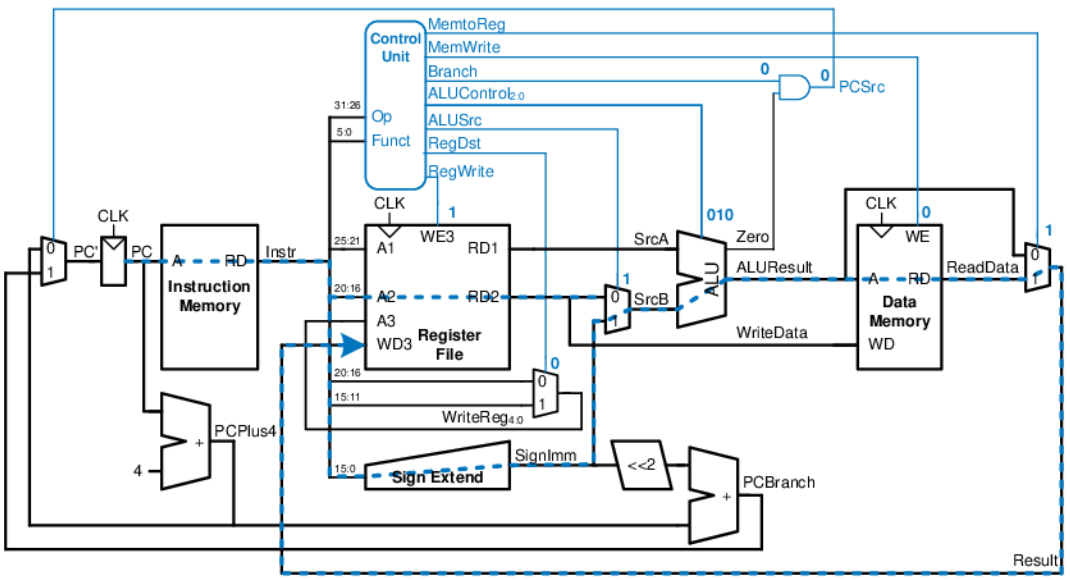


Figure 7.15 Critical path for `lw` instruction

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

Example 7.4 SINGLE-CYCLE PROCESSOR PERFORMANCE

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

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

Table 7.6 Delays of circuit elements

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

7.4 MULTICYCLE PROCESSOR

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

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

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

7.4.1 Multicycle Datapath

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

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

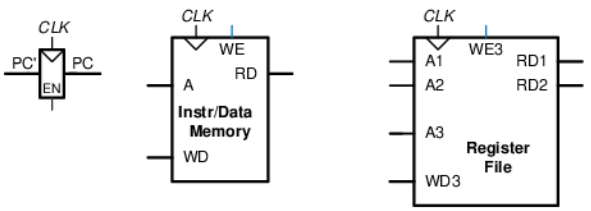


Figure 7.16 State elements with unified instruction/data memory

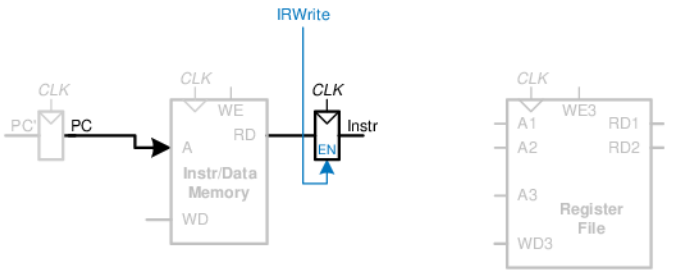


Figure 7.17 Fetch instruction from memory

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

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

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

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

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

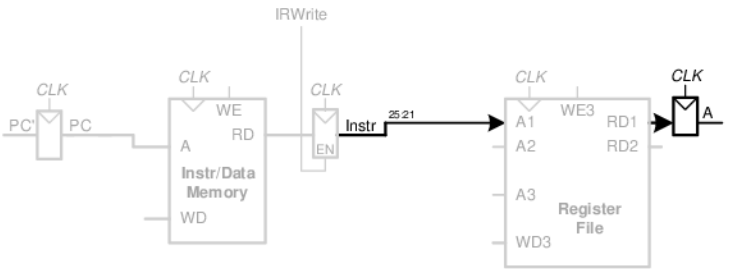


Figure 7.18 Read source operand from register file

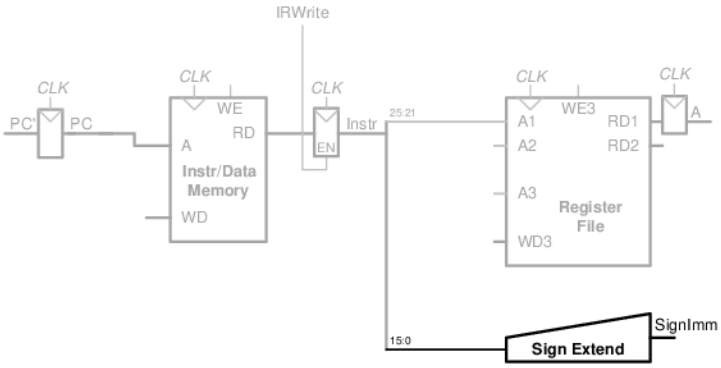


Figure 7.19 Sign-extend the immediate

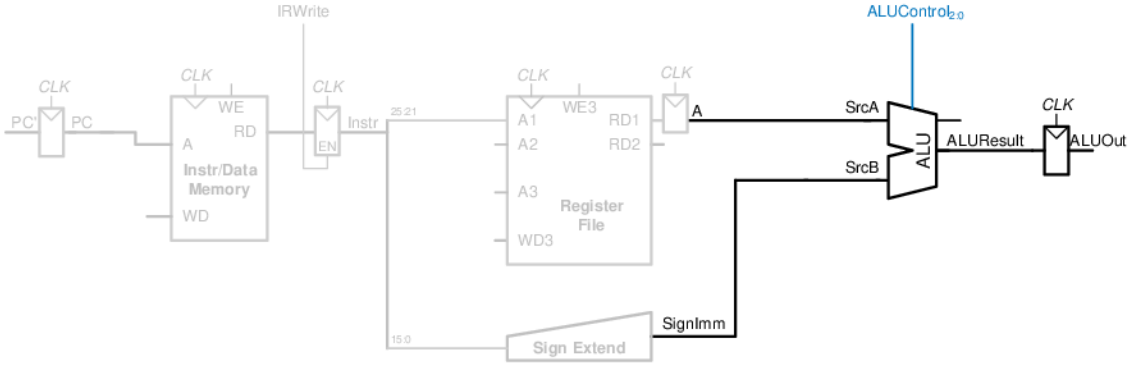


Figure 7.20 Add base address to offset

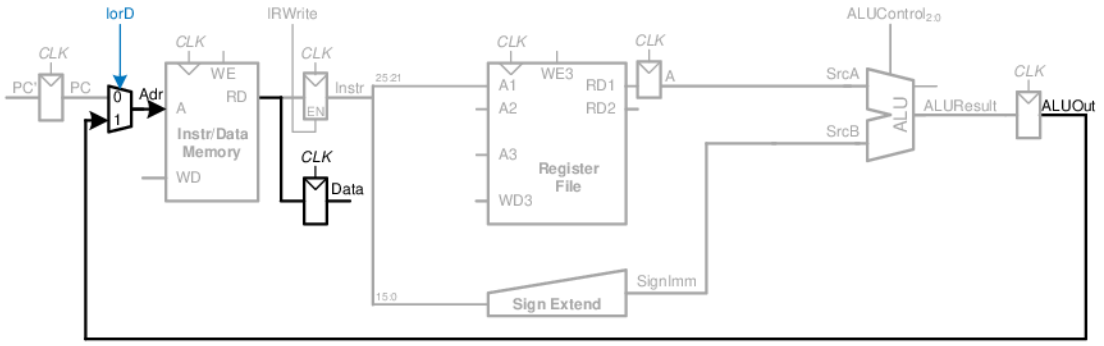


Figure 7.21 Load data from memory

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

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

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

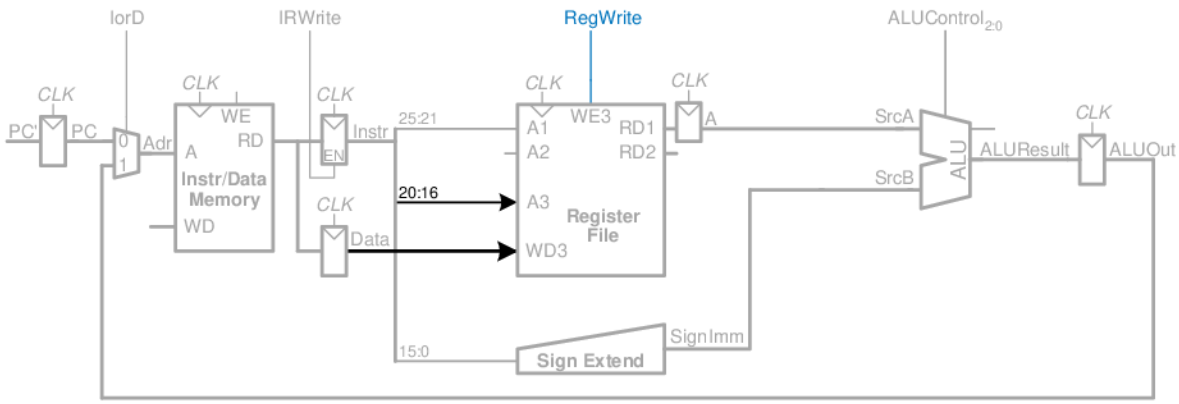


Figure 7.22 Write data back to register file

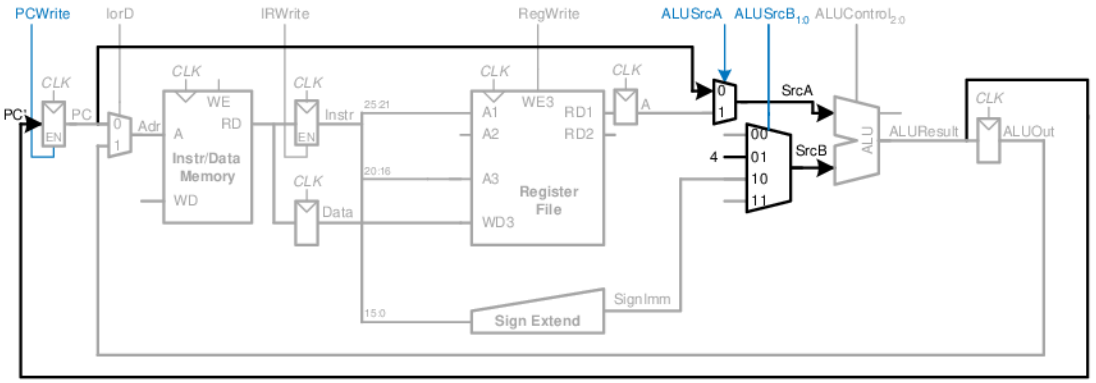


Figure 7.23 Increment PC by 4

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

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

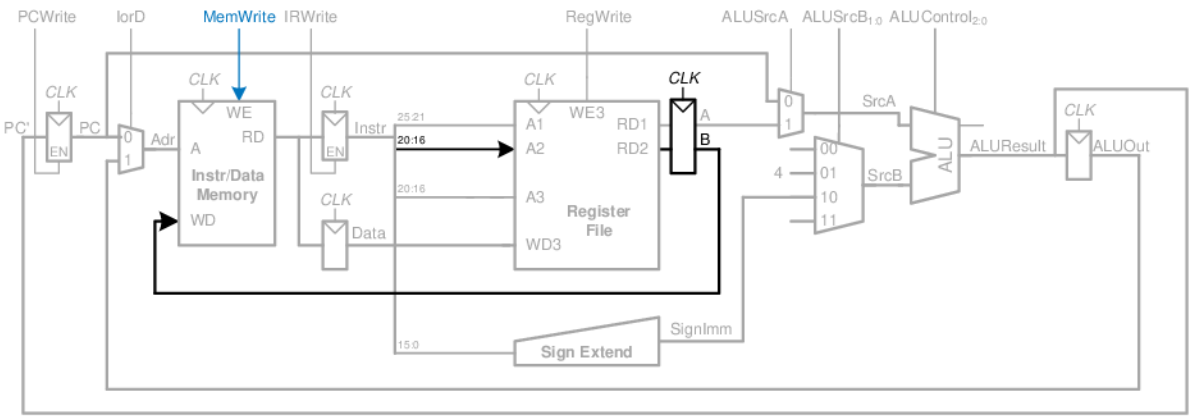


Figure 7.24 Enhanced datapath for `sw` instruction

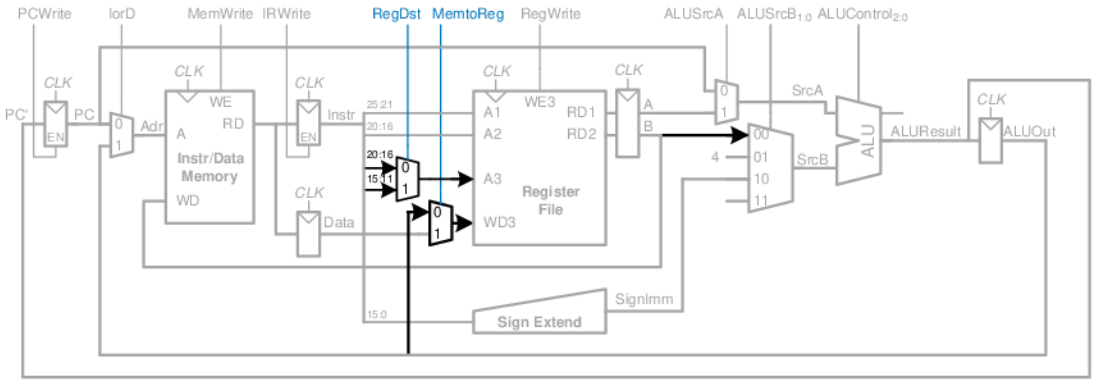


Figure 7.25 Enhanced datapath for R-type instructions

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

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

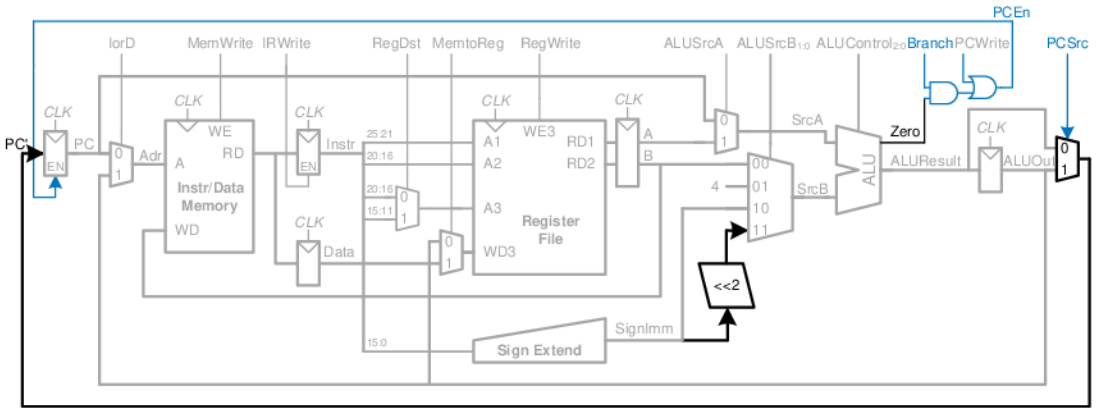


Figure 7.26 Enhanced datapath for `beq` instruction

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

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

7.4.2 Multicycle Control

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

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

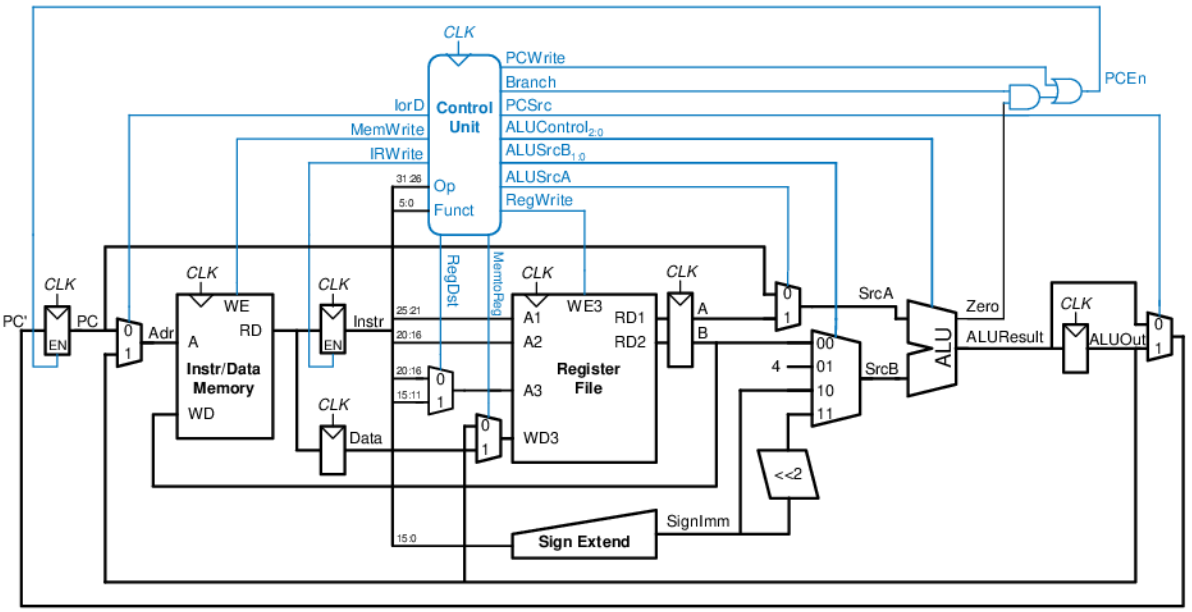


Figure 7.27 Complete multicycle MIPS processor

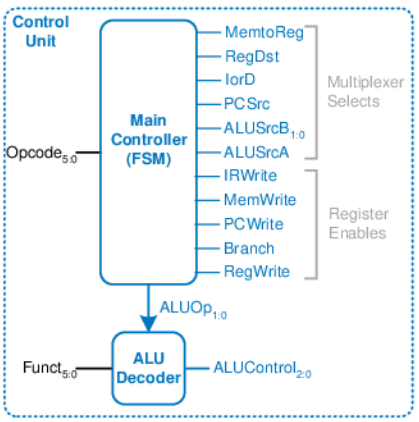


Figure 7.28 Control unit internal structure

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

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

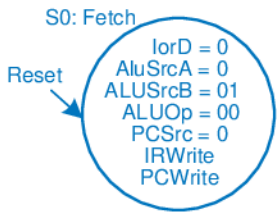


Figure 7.29 Fetch

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

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

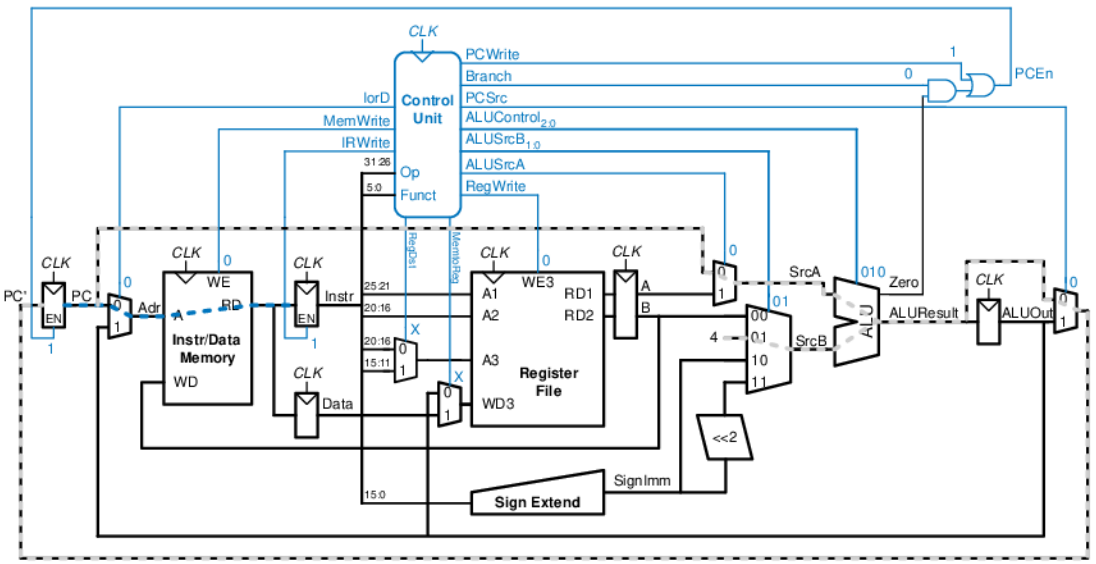


Figure 7.30 Data flow during the fetch step

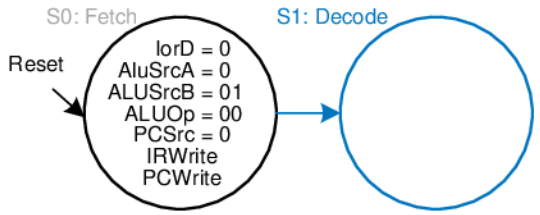


Figure 7.31 Decode

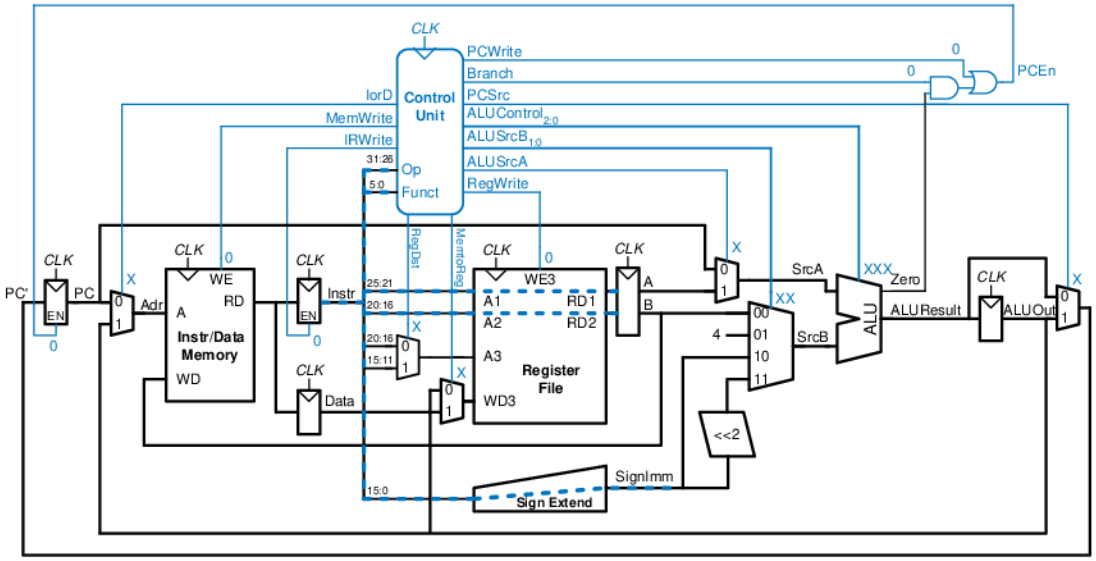


Figure 7.32 Data flow during the decode step

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

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

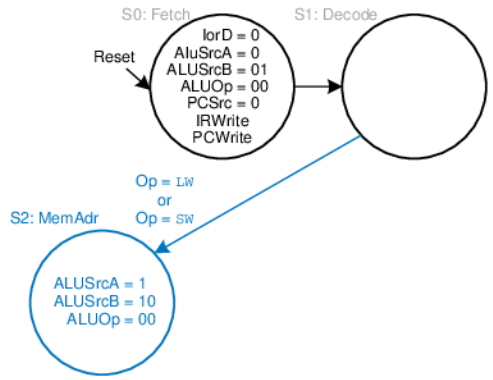


Figure 7.33 Memory address computation

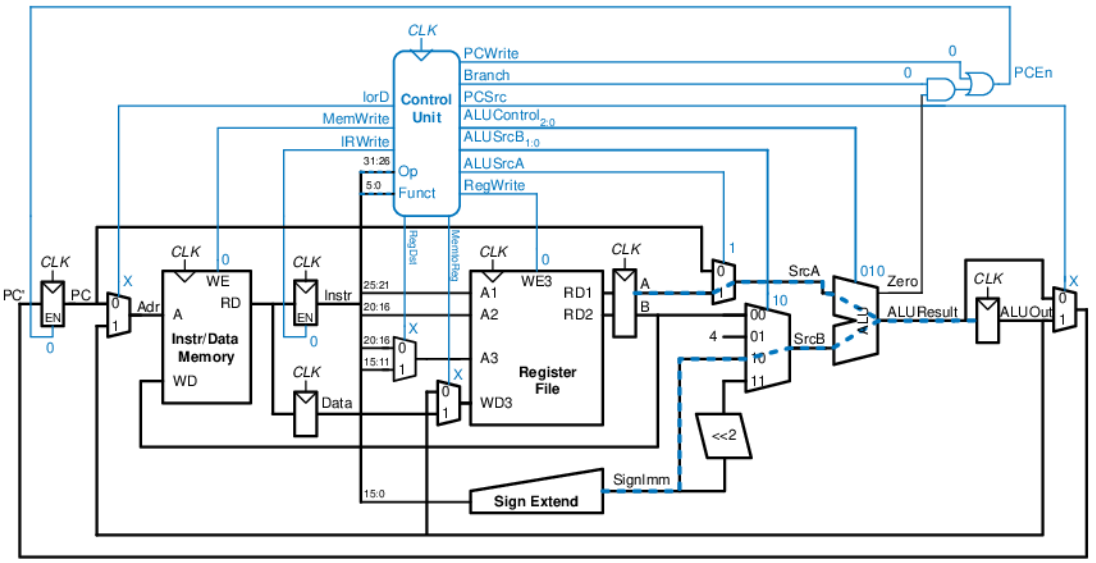


Figure 7.34 Data flow during memory address computation

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

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

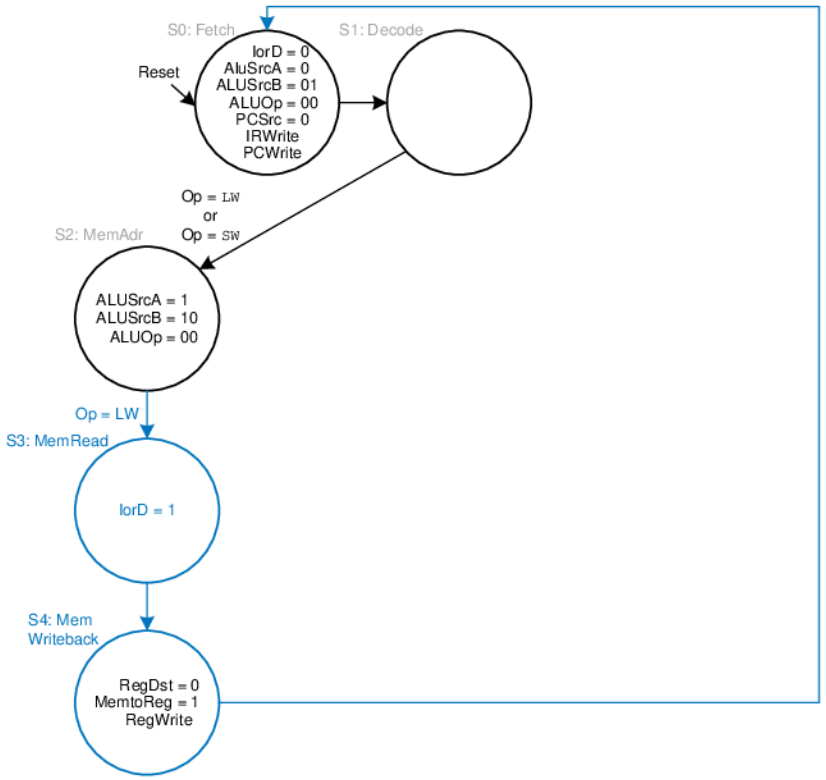


Figure 7.35 Memory read

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

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

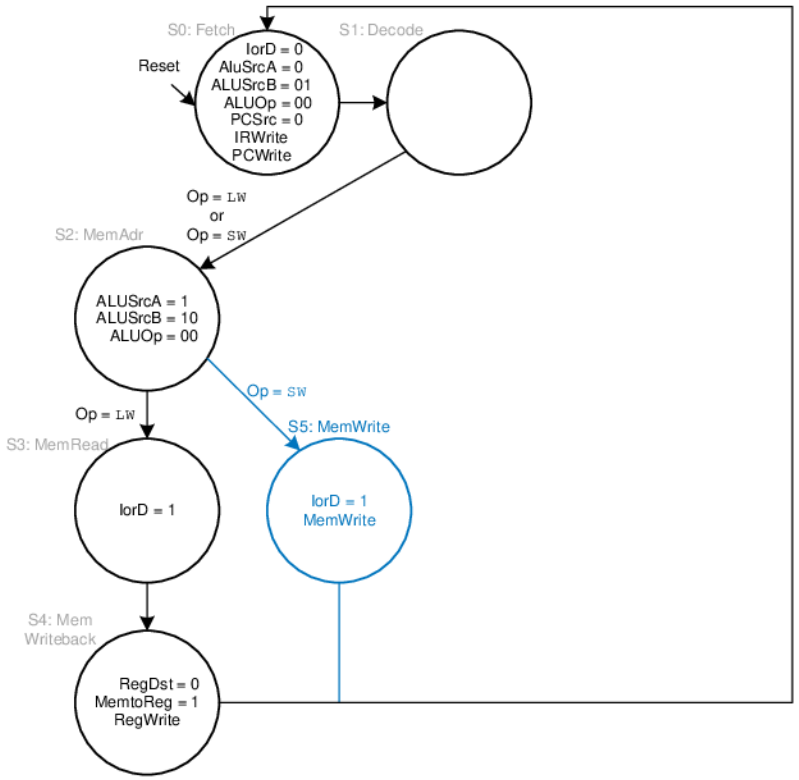


Figure 7.36 Memory write

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

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

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

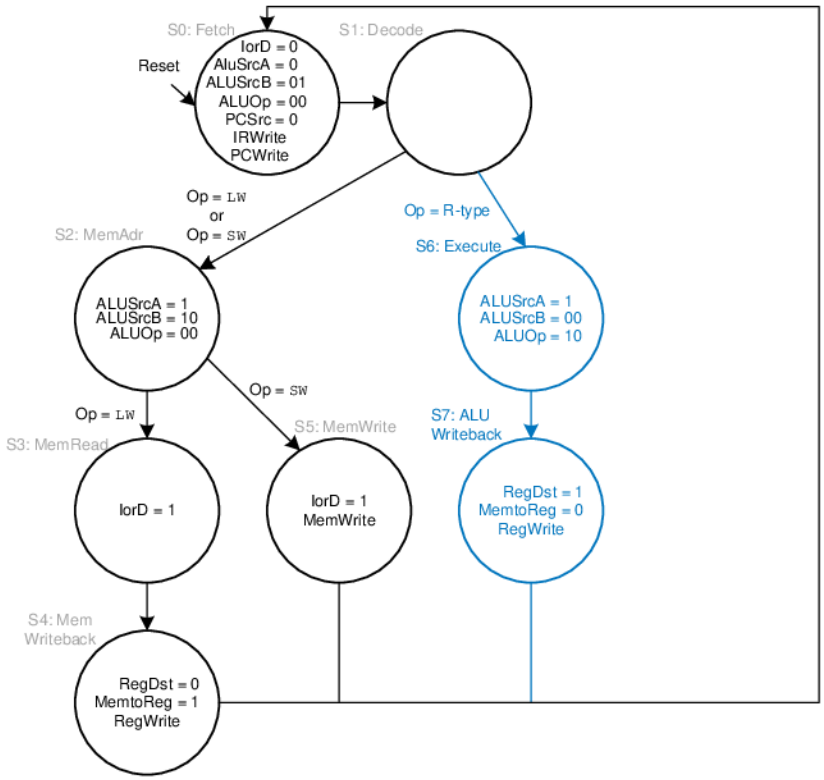


Figure 7.37 Execute R-type operation

7.4.3 More Instructions

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

Example 7.5 `addi` INSTRUCTION

Modify the multicycle processor to support `addi`.

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

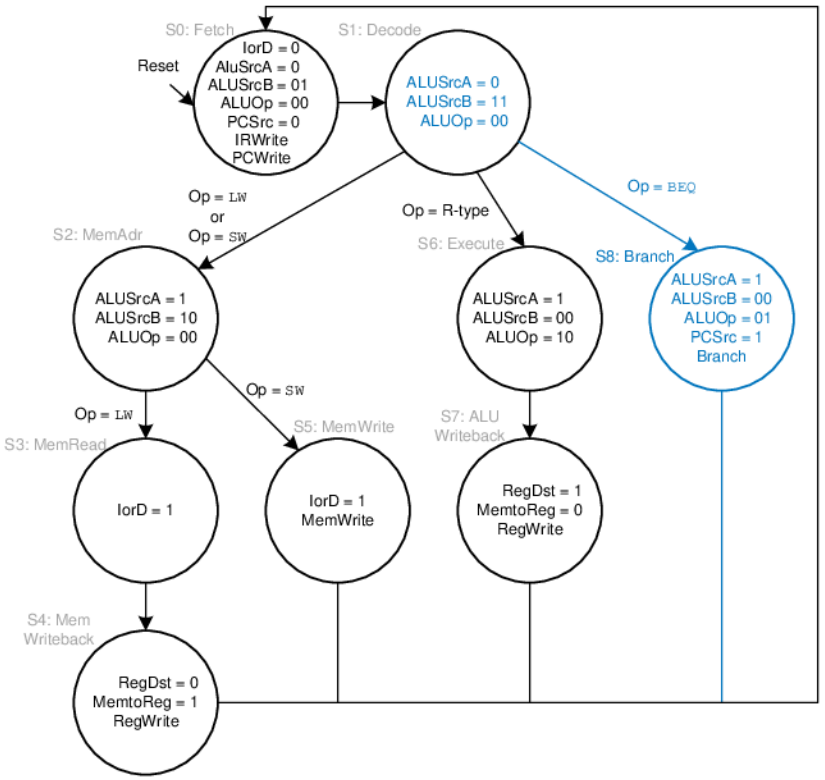


Figure 7.38 Branch

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

Example 7.6 `j` INSTRUCTION

Modify the multicycle processor to support `j`.

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

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

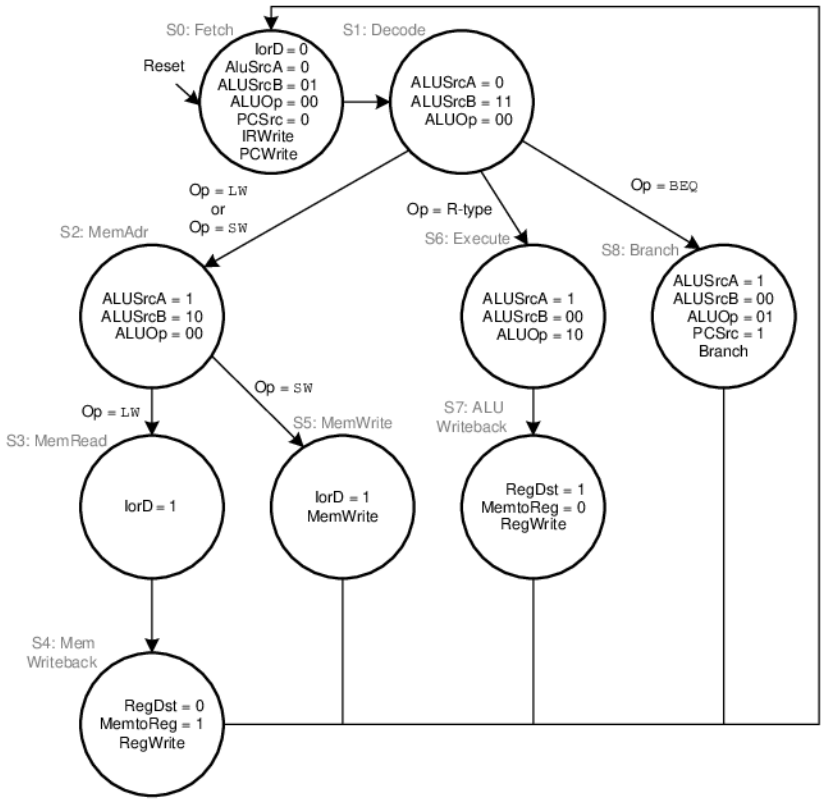


Figure 7.39 Complete multicycle control FSM

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

7.4.4 Performance Analysis

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

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

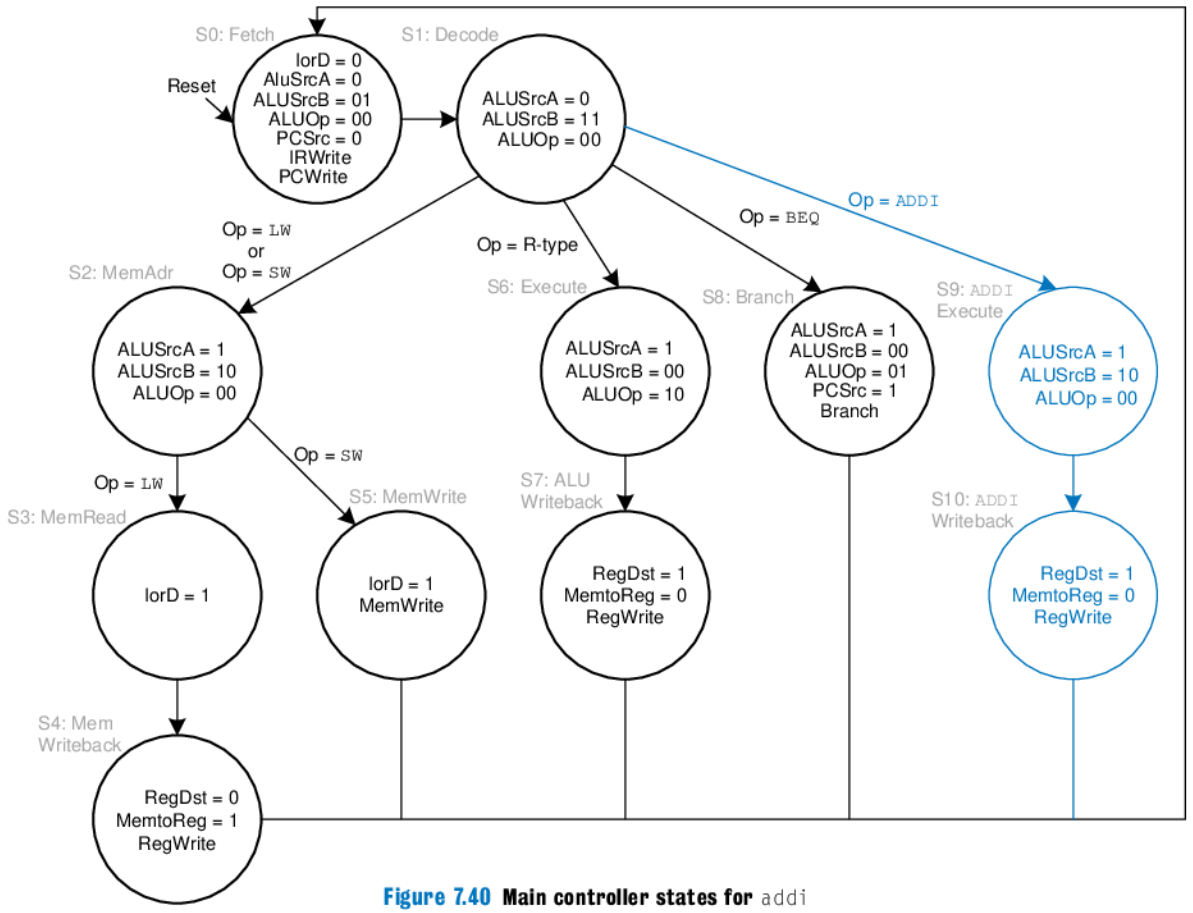


Figure 7.40 Main controller states for `addi`

Example 7.7 MULTICYCLE PROCESSOR CPI

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

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

---

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

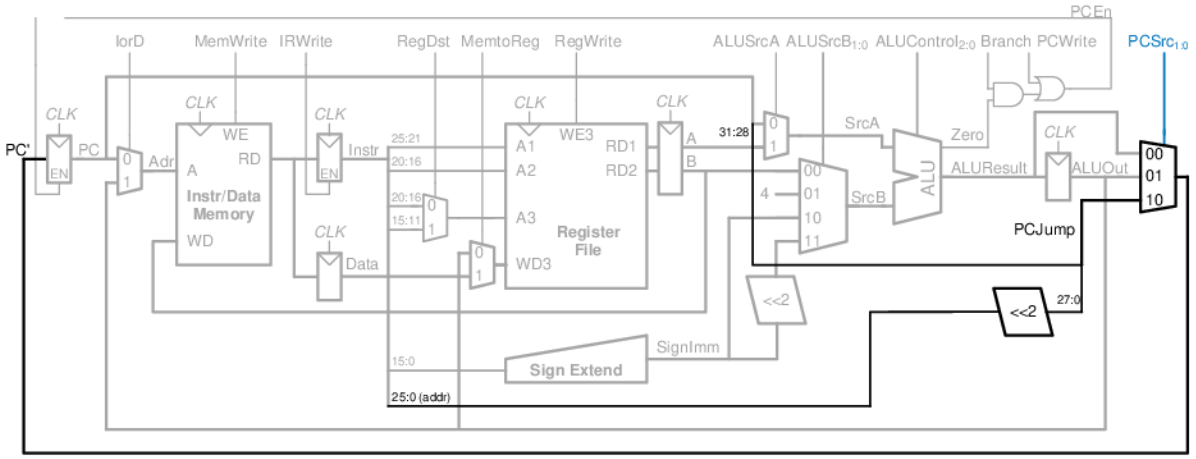


Figure 7.41 **Multicycle MIPS datapath enhanced to support the `j` instruction**

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

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

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

Example 7.8 PROCESSOR PERFORMANCE COMPARISON

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

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

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

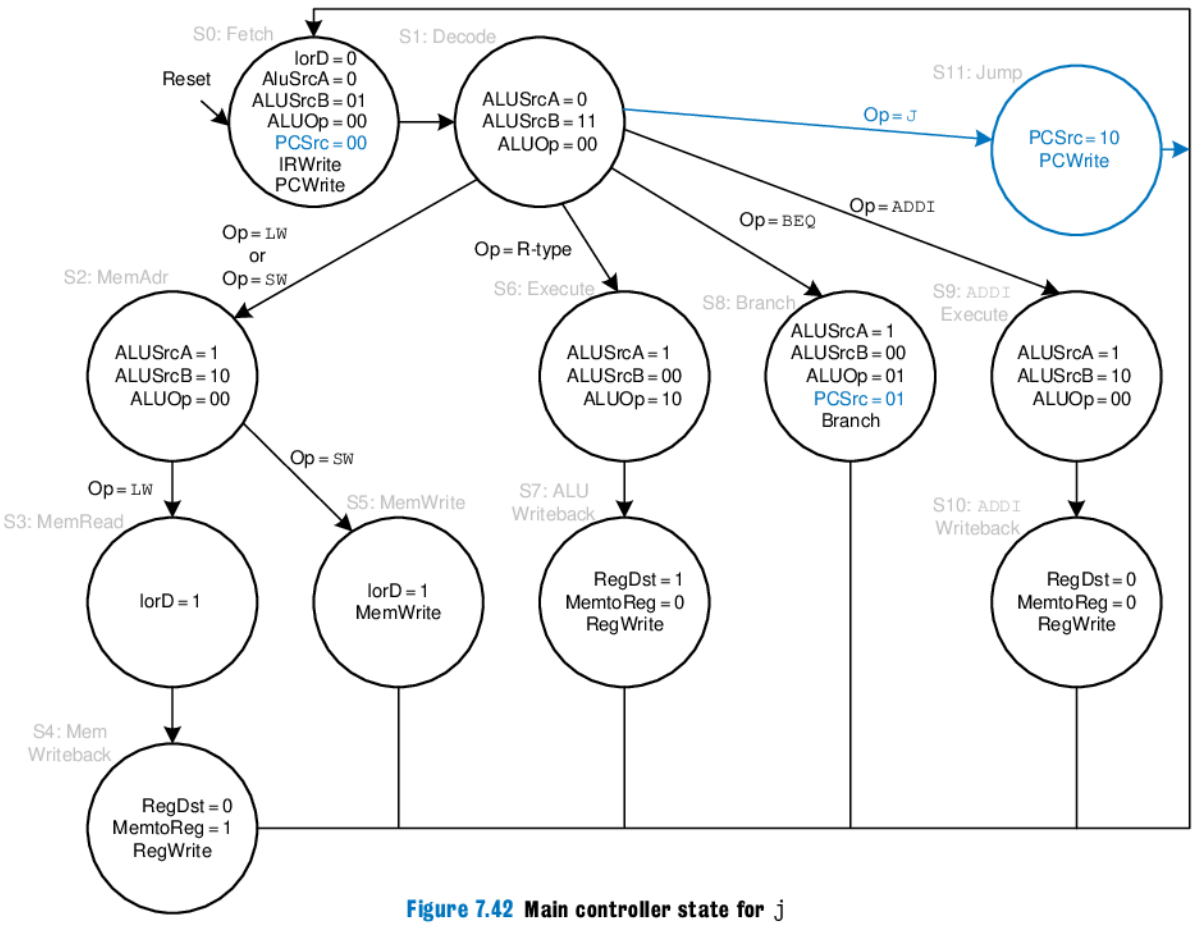


Figure 7.42 **Main controller state for `j`**

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

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

7.5 PIPELINED PROCESSOR

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

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

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

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

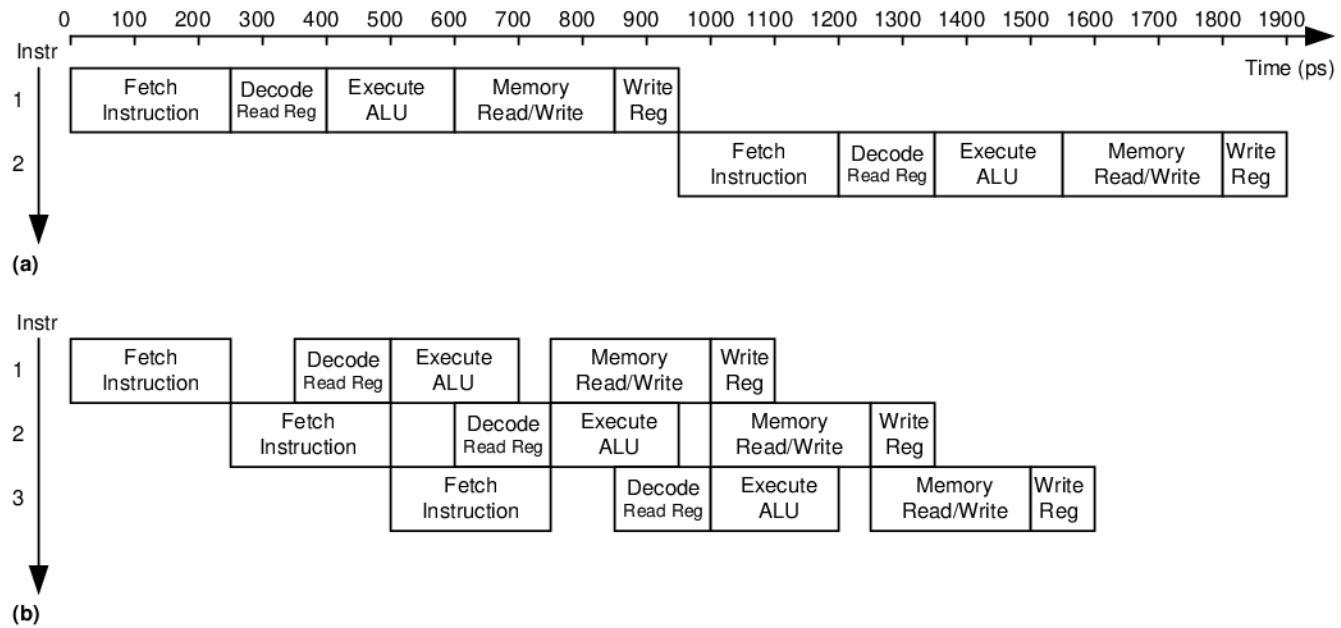


Figure 7.43 Timing diagrams: (a) single-cycle processor, (b) pipelined processor

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

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

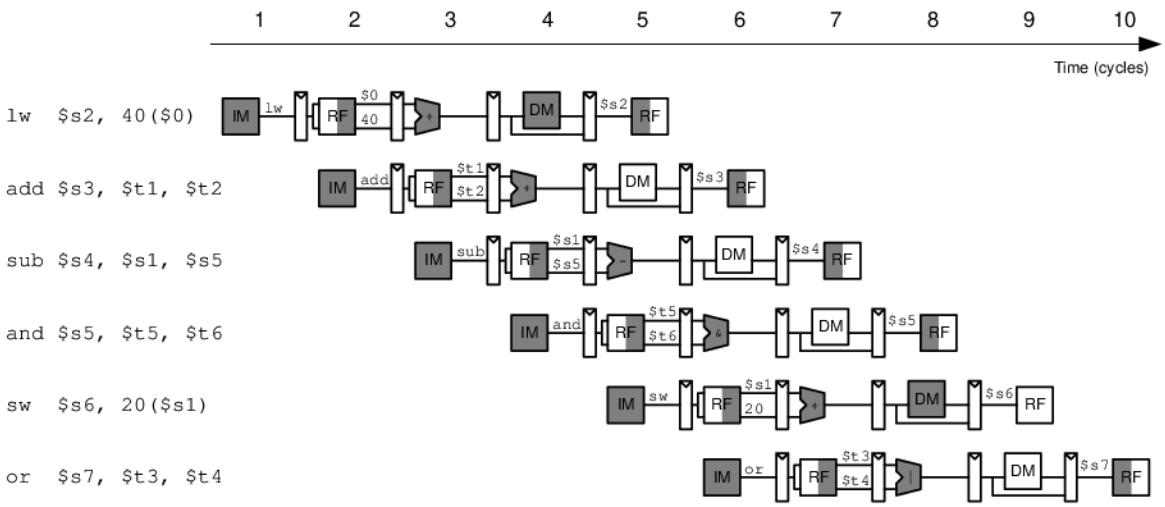


Figure 7.44 Abstract view of pipeline in operation

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

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

7.5.1 Pipelined Datapath

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

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

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

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

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

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

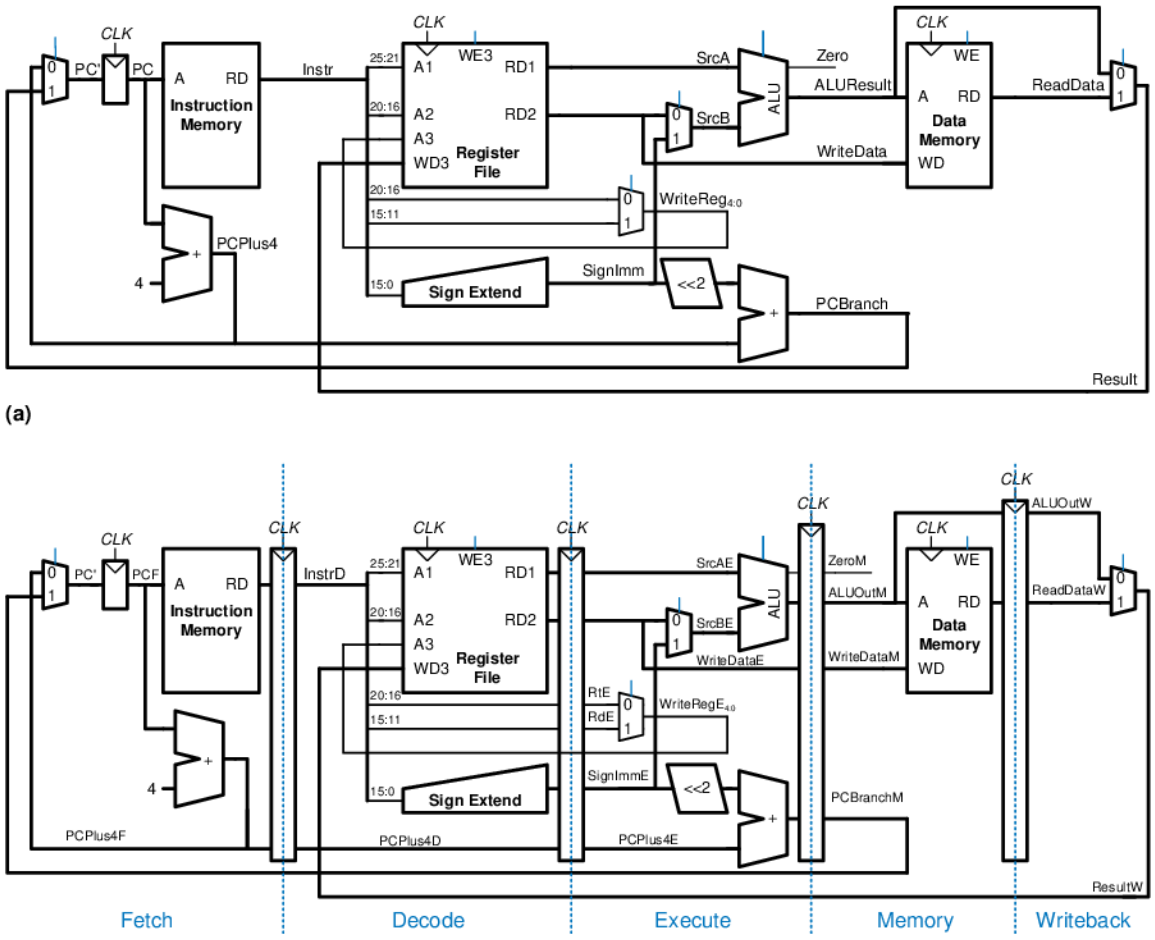


Figure 7.45 Single-cycle and pipelined datapaths

7.5.2 Pipelined Control

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

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

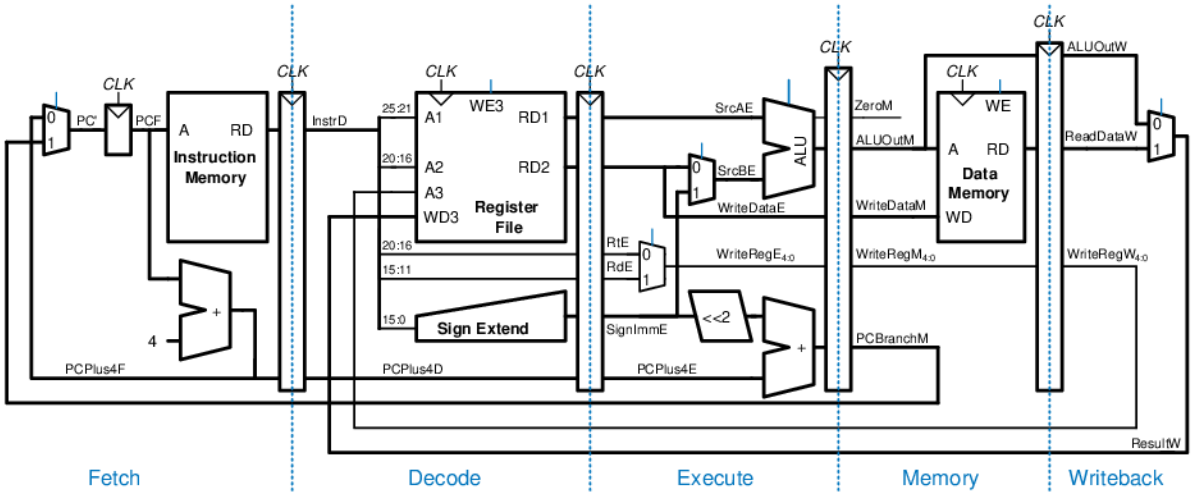


Figure 7.46 Corrected pipelined datapath

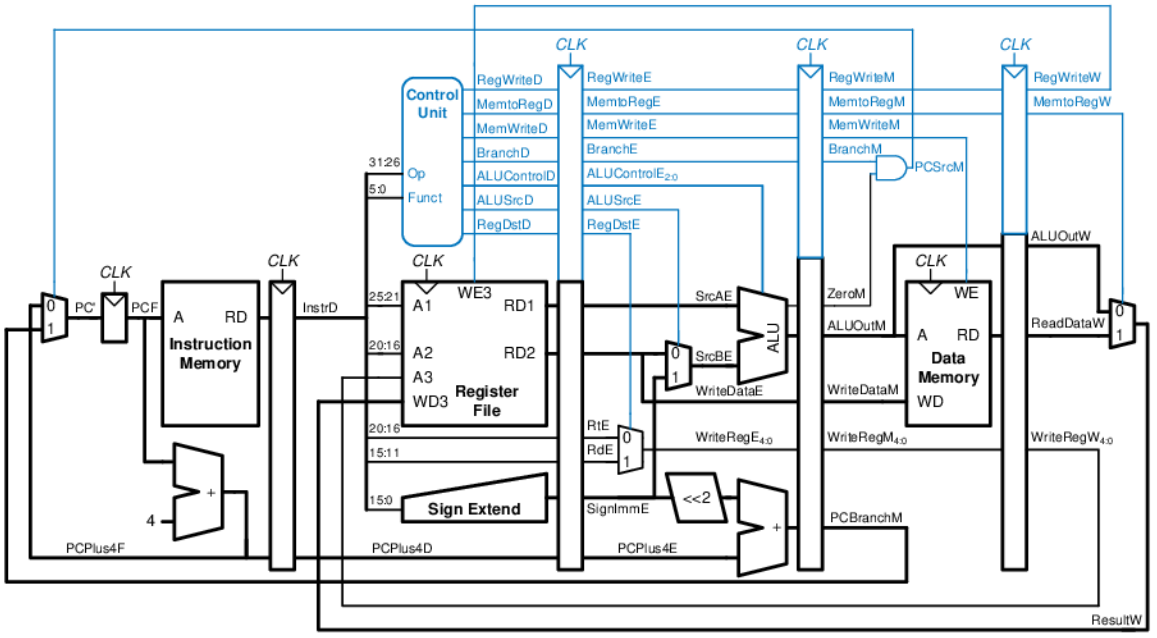


Figure 7.47 Pipelined processor with control

7.5.3 Hazards

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

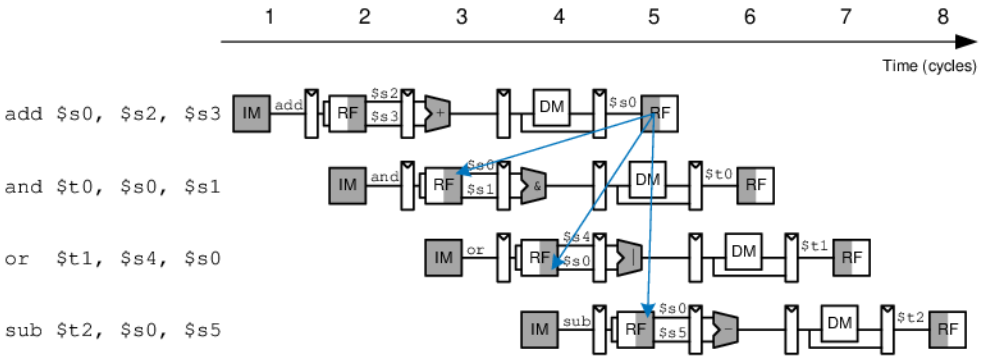


Figure 7.48 Abstract pipeline diagram illustrating hazards

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

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

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

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

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

Solving Data Hazards with Forwarding

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

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

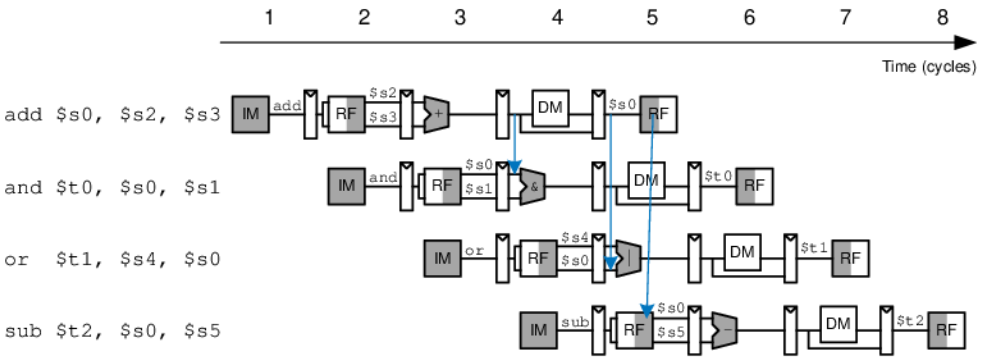


Figure 7.49 Abstract pipeline diagram illustrating forwarding

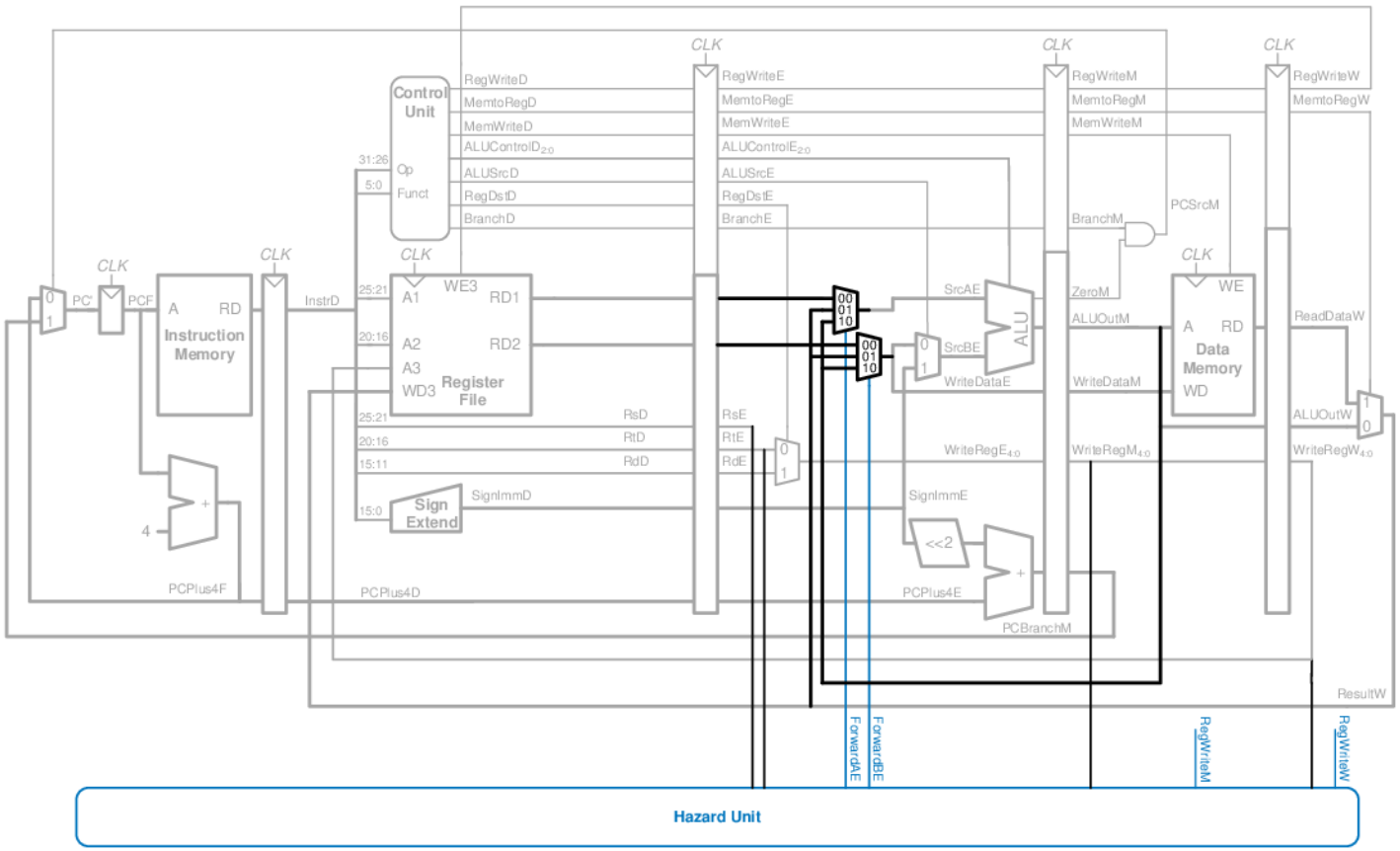


Figure 7.50 Pipelined processor with forwarding to solve hazards

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

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

Solving Data Hazards with Stalls

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

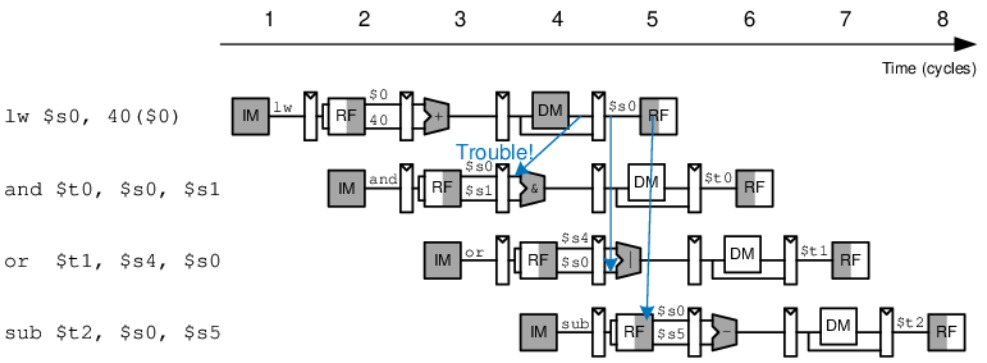


Figure 7.51 Abstract pipeline diagram illustrating trouble forwarding from `lw`

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

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

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

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

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

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

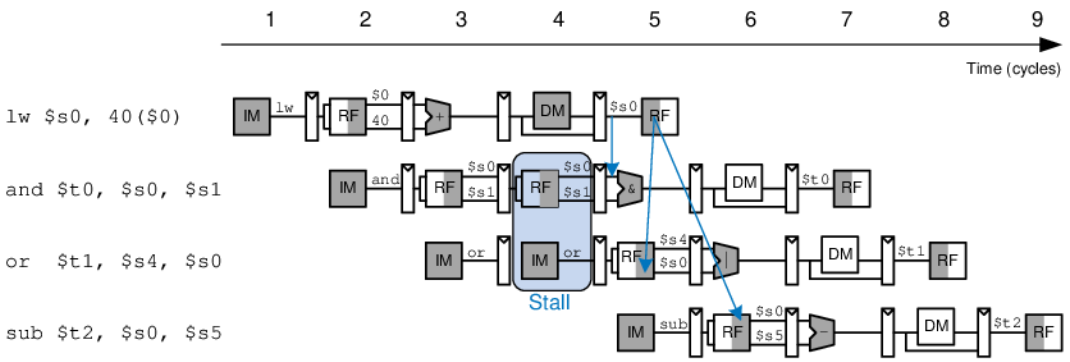


Figure 7.52 Abstract pipeline diagram illustrating stall to solve hazards

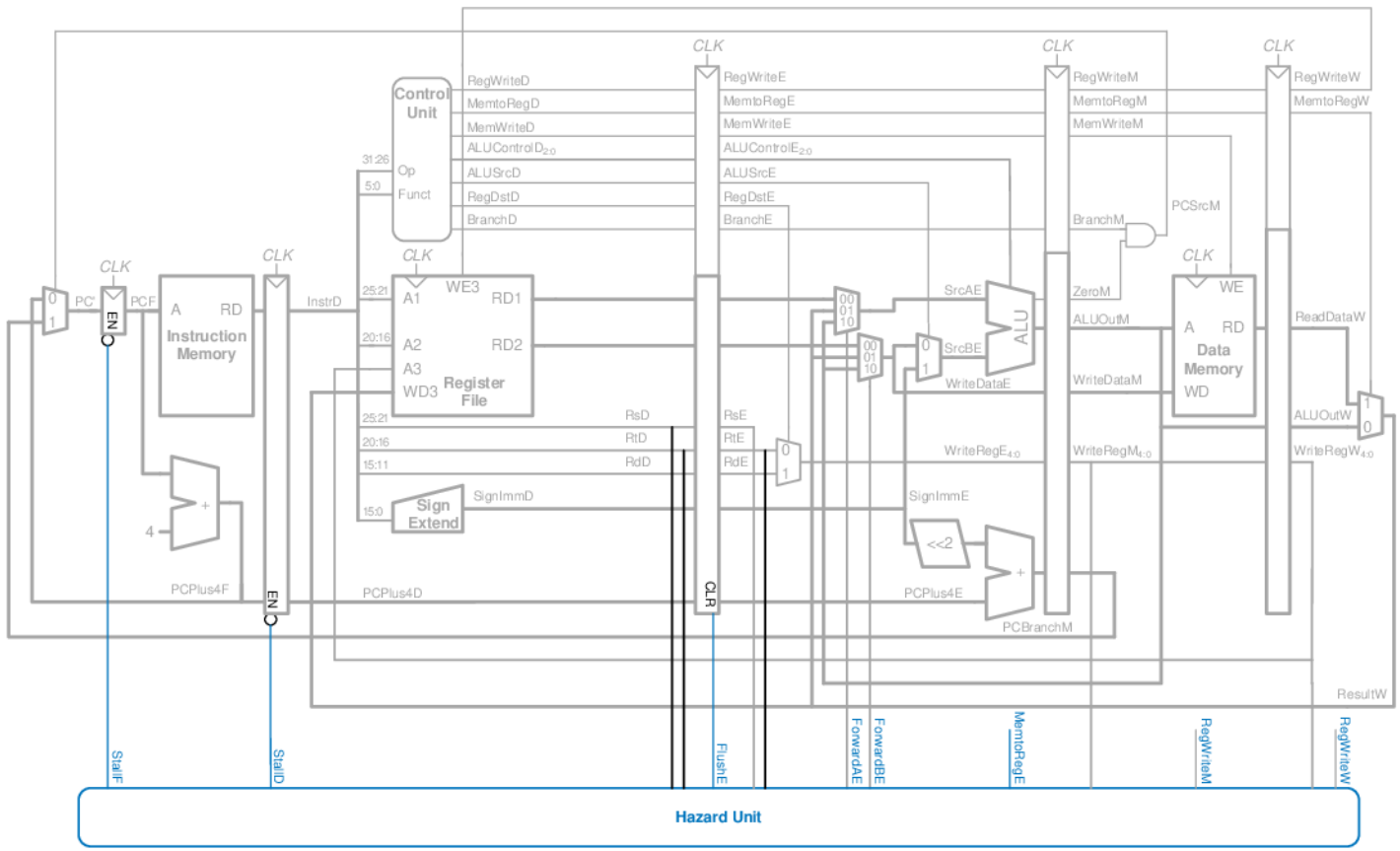


Figure 7.53 **Pipelined processor with stalls to solve `lw` data hazard**

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

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

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

Solving Control Hazards

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

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

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

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

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

---

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

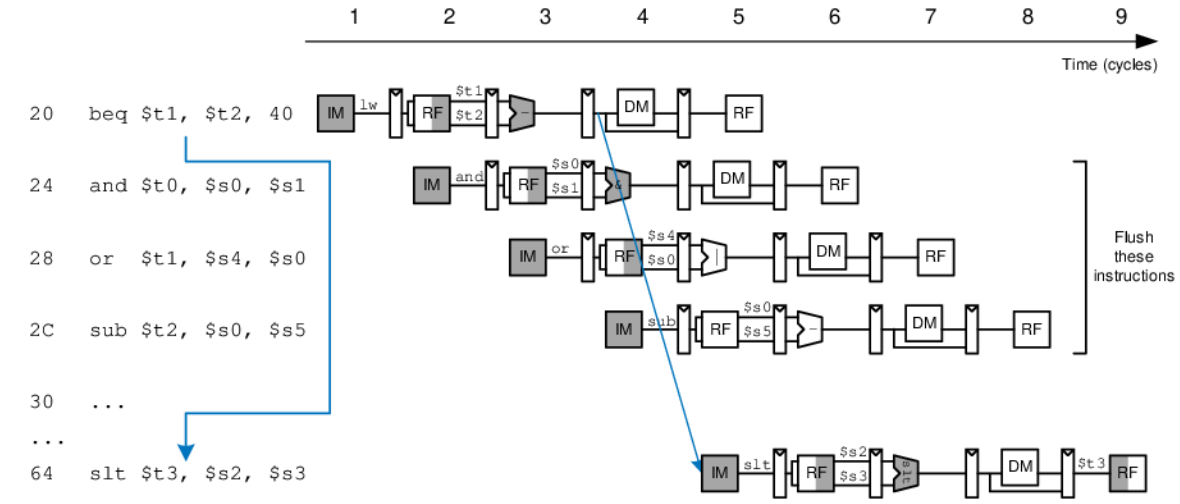


Figure 7.54 Abstract pipeline diagram illustrating flushing when a branch is taken

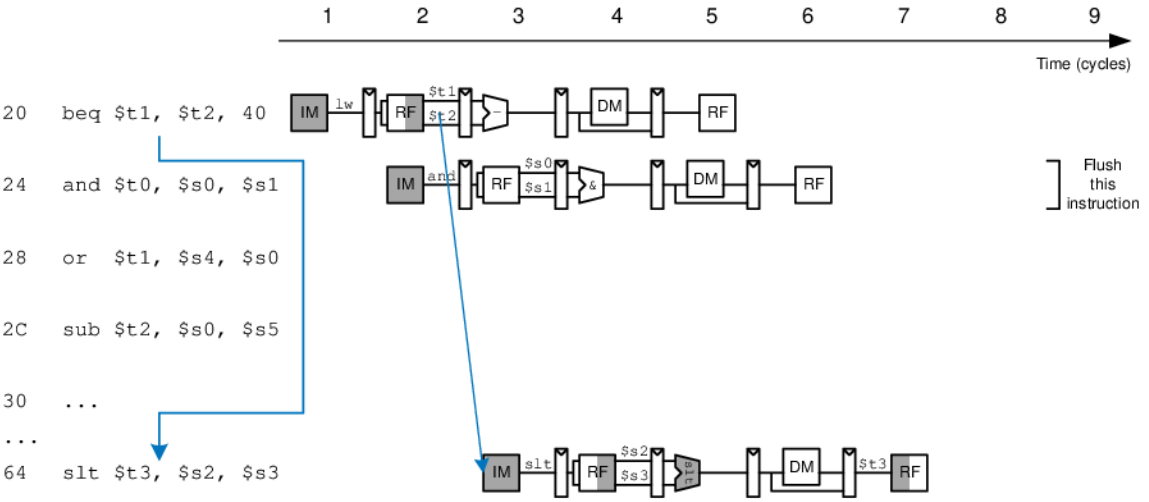


Figure 7.55 Abstract pipeline diagram illustrating earlier branch decision

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

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

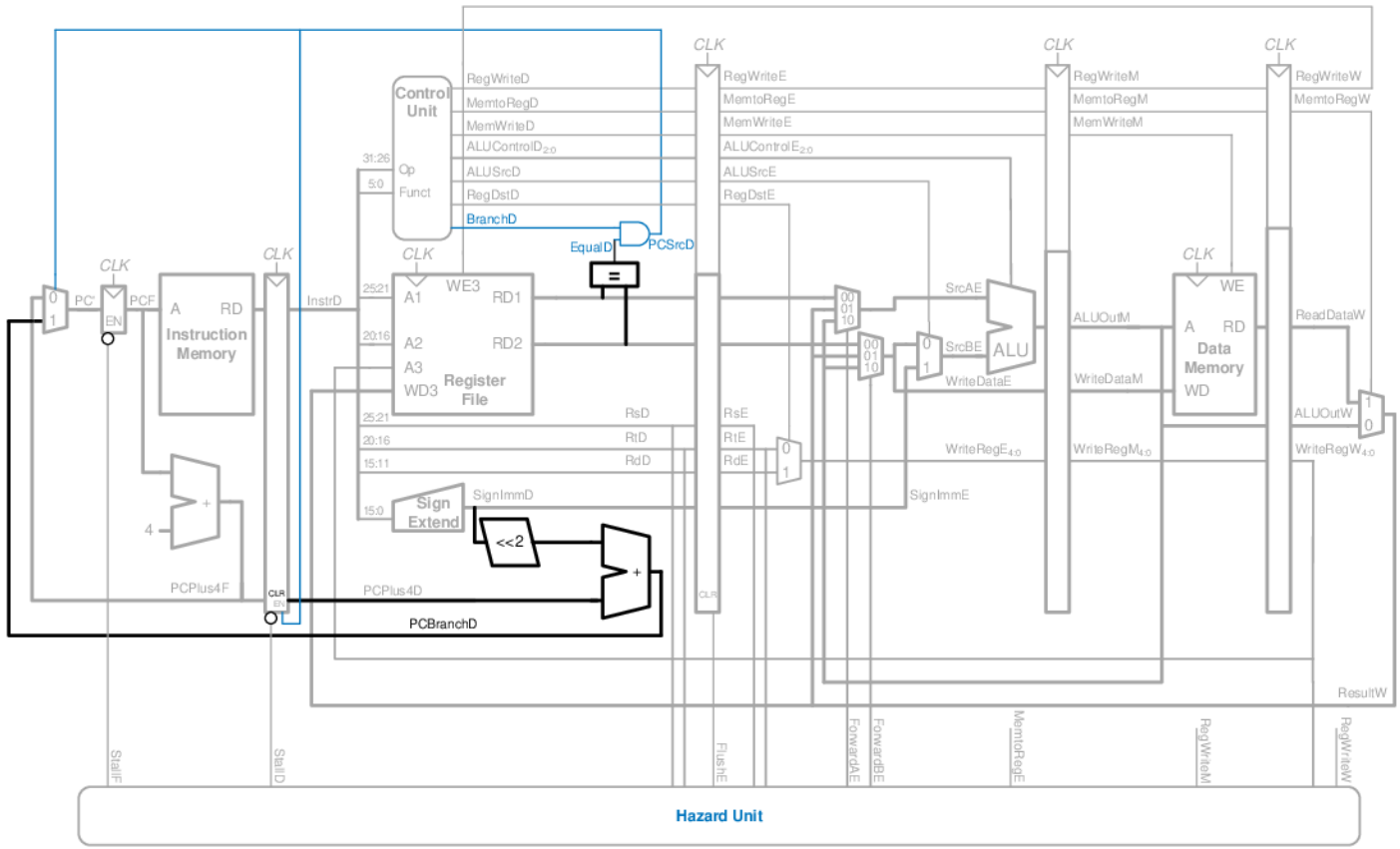


Figure 7.56 Pipelined processor handling branch control hazard

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

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

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

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

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

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

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

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

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

Hazard Summary

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

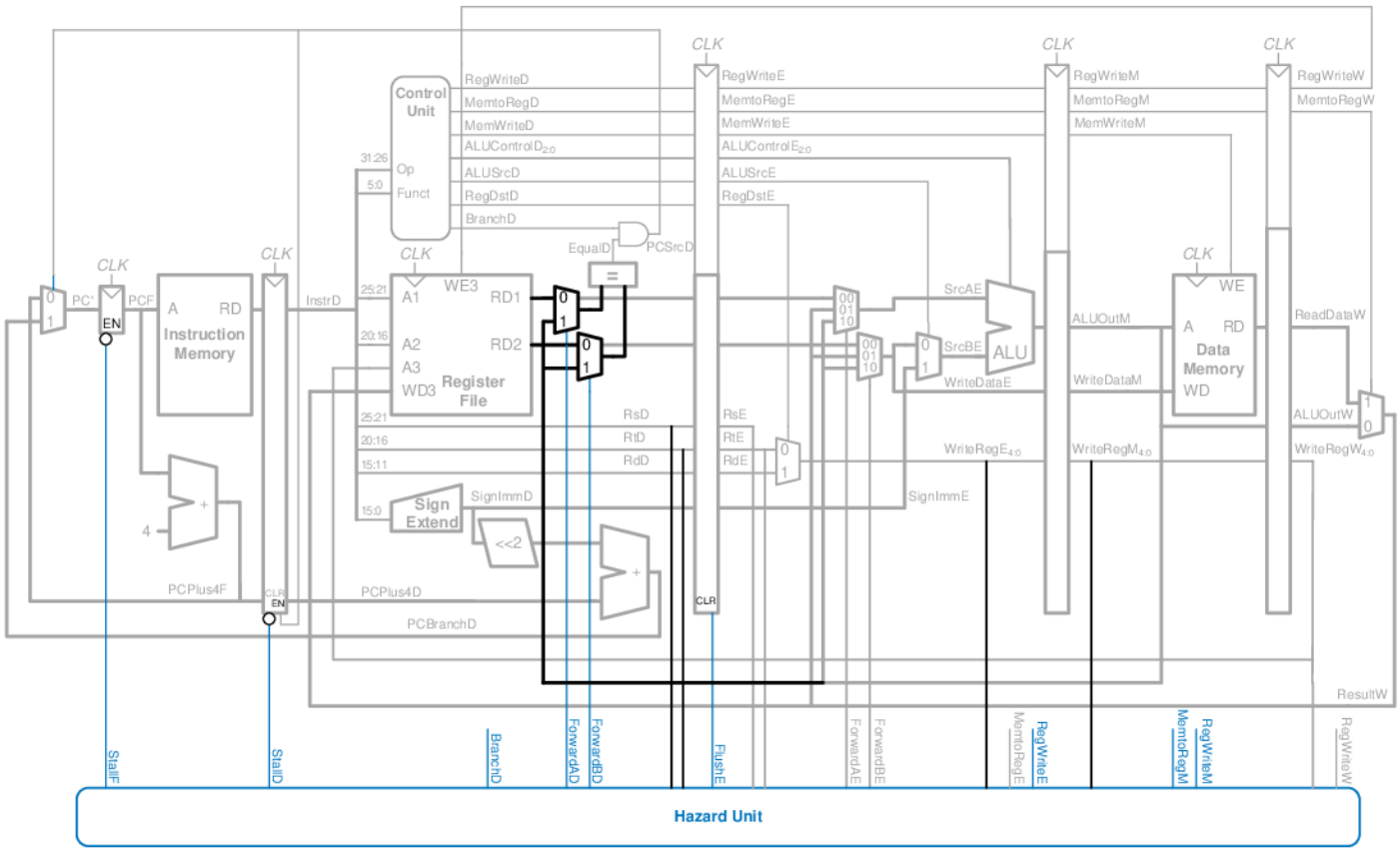


Figure 7.57 Pipelined processor handling data dependencies for branch instructions

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

7.5.4 More Instructions

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

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

7.5.5 Performance Analysis

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

Example 7.9 PIPELINED PROCESSOR CPI

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

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

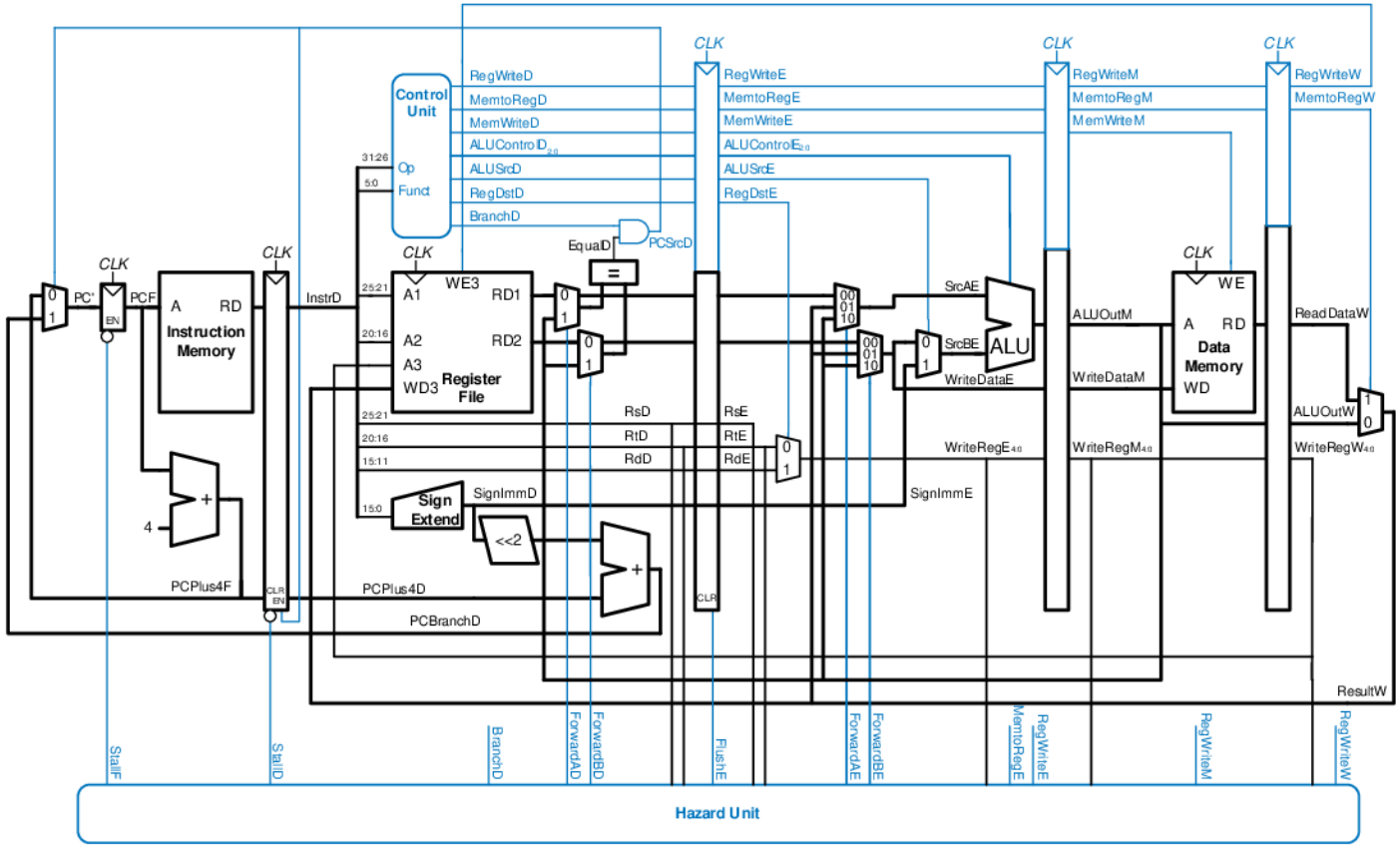


Figure 7.58 Pipelined processor with full hazard handling

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

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

Example 7.10 PROCESSOR PERFORMANCE COMPARISON

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

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

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

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

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

7.6 HDL REPRESENTATION*

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

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

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

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

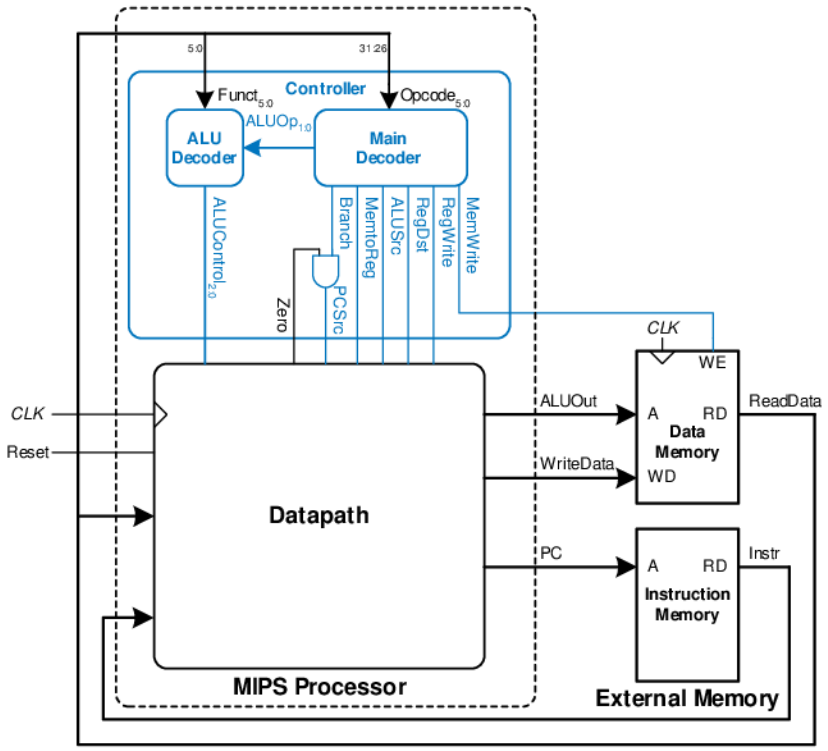


Figure 7.59 MIPS single-cycle processor interfaced to external memory

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

7.6.1 Single-Cycle Processor

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

HDL Example 7.1 SINGLE-CYCLE MIPS PROCESSOR

Verilog

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

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

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

VHDL

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

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

HDL Example 7.2 CONTROLLER

Verilog

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

  wire [1:0] aluop;
  wire       branch;

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

  assign pcsrc = branch & zero;
endmodule
```

VHDL

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

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

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

HDL Example 7.3 MAIN DECODER

Verilog

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

 reg [8:0] controls;

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

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

VHDL

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

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

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

HDL Example 7.4 ALU DECODER

Verilog

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

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

VHDL

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

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

HDL Example 7.5 DATAPATH

Verilog

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

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

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

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

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

VHDL

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

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

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

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

7.6.2 Generic Building Blocks

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

HDL Example 7.6 REGISTER FILE

Verilog

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

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

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

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

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

VHDL

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

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

HDL Example 7.7 ADDER

Verilog

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

  assign y = a + b;
endmodule
```

VHDL

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

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

HDL Example 7.8 LEFT SHIFT (MULTIPLY BY 4)

Verilog

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

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

VHDL

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

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

HDL Example 7.9 SIGN EXTENSION

Verilog

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

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

VHDL

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

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

HDL Example 7.10 RESETTABLE FLIP-FLOP

Verilog

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

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

VHDL

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

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