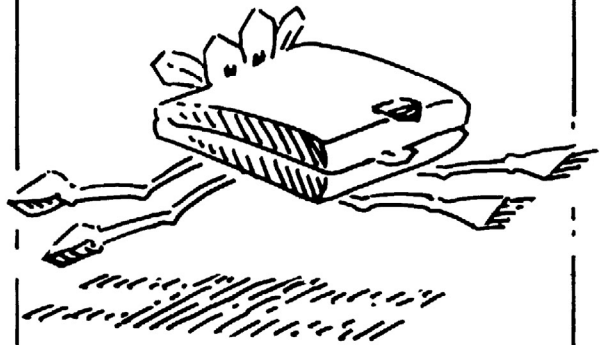


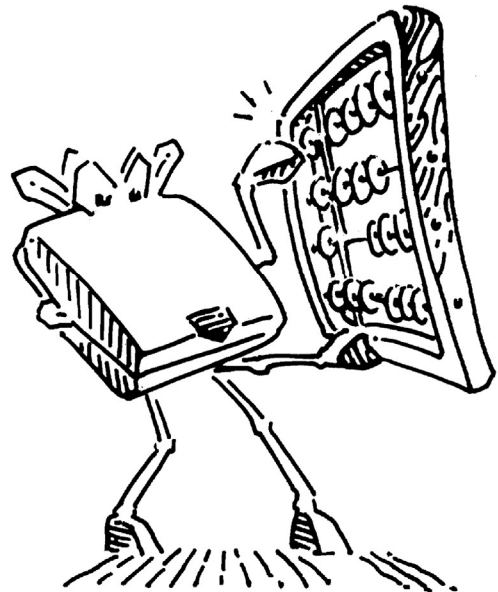
JUMP



FETCH



ADD



Architecture

6

6.1 INTRODUCTION

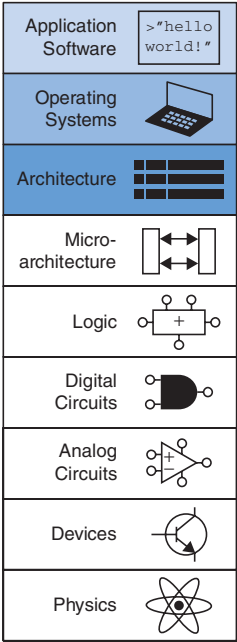
The previous chapters introduced digital design principles and building blocks. In this chapter, we jump up a few levels of abstraction to define the architecture of a computer. The *architecture* is the programmer's view of a computer. It is defined by the instruction set (language) and operand locations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

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

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

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

- 6.1 Introduction
- 6.2 Assembly Language
- 6.3 Programming
- 6.4 Machine Language
- 6.5 Lights, Camera, Action: Compiling, Assembling, and Loading*
- 6.6 Odds and Ends*
- 6.7 Evolution of the RISC-V Architecture
- 6.8 Another Perspective: x86 Architecture
- 6.9 Summary
- Exercises
- Interview Questions





Krste Asanović started RISC-V as a summer project. He is a professor of computer science at the University of California, Berkeley and the Chairman of the Board for RISC-V International, formerly known as the RISC-V Foundation. He is also the cofounder of SiFive, a company that develops and commercializes RISC-V chips, boards, and supporting tools. (Photo printed with permission.)



Andrew Waterman designs microprocessors at SiFive, a company he cofounded with Krste Asanović in 2015 to provide low-cost RISC-V cores and custom chips. He earned his PhD in computer science from UC Berkeley in 2016, where, weary of the vagaries of existing instruction-set architectures, he co-designed the RISC-V ISA and the first RISC-V cores. (Photo printed with permission.)

A computer architecture does not define the underlying hardware implementation. Often, many different hardware implementations of a single architecture exist. For example, Intel and Advanced Micro Devices (AMD) both sell various microprocessors belonging to the same x86 architecture. These microprocessors can all run the same programs, but they use different underlying hardware. Therefore, they 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 gadgets or laptop computers. The specific arrangement of registers, memories, arithmetic/logical units (ALUs), and other building blocks to form a microprocessor is called the *microarchitecture* and will be the subject of [Chapter 7](#).

In this text, we introduce the RISC-V (pronounced “risk five”) architecture, the first open-source instruction set architecture with broad commercial support. We describe the RISC-V 32-bit integer instruction set (RV32I) version 2.2, which forms the core of RISC-V’s instruction set. [Sections 6.6](#) and [6.7](#) summarize features of other versions of the architecture. The *RISC-V Instruction Set Manual*, available online, is the authoritative definition of the architecture.

The RISC-V architecture was initially defined in 2010 at the University of California, Berkeley by Krste Asanović, Andrew Waterman, David Patterson, and others. Since its inception, many people have contributed to its development. RISC-V is unusual in that its open-source nature makes it free to use, and it is comparable in capabilities to commercial architectures such as ARM and x86. So far, only a few companies have built commercial chips, including SiFive and Western Digital, but adoption is rapidly increasing. We start our journey into understanding the RISC-V architecture by introducing assembly language instructions, operand locations, and common programming constructs, such as branches, loops, array manipulations, and function calls. We then describe how the assembly language translates into machine language and show how a program is loaded into memory and executed.

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

6.2 ASSEMBLY LANGUAGE

Assembly language is the human-readable representation of the computer’s native language. Each assembly language instruction specifies both the operation to perform and the operands on which to operate. We introduce simple arithmetic instructions and show how these operations

are written in assembly language. We then define the RISC-V instruction operands: registers, memory, and constants.

This chapter assumes that you already have some familiarity with a high-level programming language such as C, C++, or Java. (These languages are practically identical for most of the examples in this chapter, but where they differ, we will use C.) [Appendix C](#) provides an introduction to C for those with little or no prior programming experience.

6.2.1 Instructions

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

Code Example 6.1 ADDITION

High-Level Code	RISC-V Assembly Code
<code>a = b + c;</code>	<code>add a, b, c</code>

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*.

[Code Example 6.2](#) shows that subtraction is similar to addition. The `sub` instruction format is the same as the `add` instruction: destination operand, followed by two sources. This consistent instruction format is an example of the first design principle:

Design Principle 1: Regularity supports simplicity.

Code Example 6.2 SUBTRACTION

High-Level Code	RISC-V Assembly Code
<code>a = b - c;</code>	<code>sub a, b, c</code>

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 RISC-V instructions, as shown in [Code Example 6.3](#).



David Patterson has been a professor of computer science at the University of California, Berkeley since 1976, and he coined the term *reduced instruction set computing* with John Hennessy in 1984. It was later commercialized as the SPARC architecture. He helped develop the RISC-V architecture and continues to play an integral role in its development. (Photo printed with permission.)

Mnemonic (pronounced nuh-maa-nik) comes from the Greek word *μνησkesthai*, *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.

RISC-V is called “five” because it is the fifth RISC architecture developed at Berkeley.

In the Preface, we describe several simulators and tools for compiling and simulating C and RISC-V assembly code. We also provide labs (available on this textbook’s companion site, see Preface) that show how to use these tools.



John Hennessy is a professor of electrical engineering and computer science at Stanford University and served as the president of Stanford from 2000 to 2016. He coined *reduced instruction set computing* with David Patterson. He also developed the MIPS computer architecture and cofounded MIPS Computer Systems in 1984. The MIPS processor was used in many commercial systems, including products from Silicon Graphics, Nintendo, and Cisco. John Hennessy and David Patterson were given the Turing Award in 2017 for pioneering a quantitative approach to the design and evaluation of computer architectures. (Photo printed with permission.)

Code Example 6.3 MORE COMPLEX CODE

High-Level Code	RISC-V Assembly Code
<pre>a = b + c - d; // single-line comment /* multiple-line comment */</pre>	<pre>add t, b, c # t = b + c sub a, t, d # a = t - d</pre>

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 RISC-V assembly language, only single-line comments are used. They begin with a hash symbol (`#`) and continue until the end of the line. The assembly language program in [Code Example 6.3](#) stores the intermediate result (`b + c`) in a temporary variable `t`. Using multiple assembly language instructions to perform more complex operations is an example of the second design principle of computer architecture:

Design Principle 2: Make the common case fast.

The RISC-V 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, RISC-V is a *reduced instruction set computer* (RISC) architecture. Architectures with many complex instructions, such as Intel’s x86 architecture, are *complex instruction set computers* (CISC). For example, x86 defines a “string move” instruction that copies a string (a series of characters) from one part of memory to another. Such an operation requires many, possibly even hundreds, of simple instructions in a RISC machine. However, the cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions.

A RISC architecture, such as RISC-V, 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, whereas an instruction set with 256 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 6.2](#), the variables `a`, `b`, and `c` are all operands. But computers operate on 1’s and 0’s, not variable names. The instructions need a physical location from which to

retrieve the binary data. Operands can be stored in registers or memory, or they may be constants stored in the instruction itself. Computers use various locations to hold operands in order to optimize for speed and data capacity. Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data. Additional data must be accessed from memory, which is large but slow. RISC-V is called a 32-bit architecture because it operates on 32-bit data.

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 RISC-V architecture has 32 registers, called the *register set*, stored in a small multiported memory called a *register file*. The fewer the registers, the faster they can be accessed. This leads to the third design principle:

Design Principle 3: Smaller is faster.

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

[Code Example 6.4](#) shows the `add` instruction with register operands. The variables `a`, `b`, and `c` are arbitrarily placed in `s0`, `s1`, and `s2`. The name `s1` is pronounced “register `s1`” or simply “`s1`.” The instruction adds the 32-bit values contained in `s1` (`b`) and `s2` (`c`) and writes the 32-bit result to `s0` (`a`). [Code Example 6.5](#) shows RISC-V assembly code using a register, `t0`, to store the intermediate calculation of `b + c`.

64- and 128-bit versions of the RISC-V architecture also exist, but we will focus on the 32-bit version in this book. The wider versions (RV64I and RV128I) are nearly identical to the 32-bit version (RV32I) except for the width of the registers and memory addresses. The main other additions are instructions that operate on only the lower half of a word and memory operations that transfer wider words.

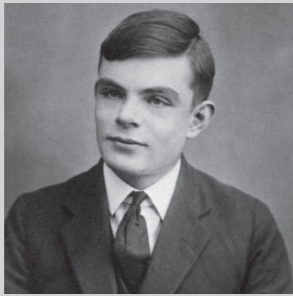
[Appendix B](#), which is located on the inside covers of the textbook, gives a handy summary of the entire RISC-V instruction set.

Code Example 6.4 REGISTER OPERANDS

High-Level Code	RISC-V Assembly Code
<code>a = b + c;</code>	<code># s0 = a, s1 = b, s2 = c</code> <code>add s0, s1, s2 # a = b + c</code>

Code Example 6.5 TEMPORARY REGISTERS

High-Level Code	RISC-V Assembly Code
<code>a = b + c - d;</code>	<code># s0 = a, s1 = b, s2 = c, s3 = d, t0 = t</code> <code>add t0, s1, s2 # t = b + c</code> <code>sub s0, t0, s3 # a = t - d</code>



Alan Turing, 1912–1954

A British mathematician and computer scientist who is considered the founder of theoretical computer science and artificial intelligence. He invented the Turing machine, a mathematical model of computation that represents an abstract processor. He also developed an electromechanical machine to decipher encrypted messages during World War II, which shortened the war and saved millions of lives. In 1952, Turing was prosecuted for homosexual acts and was sentenced to a chemical castration treatment in lieu of prison. Two years later, he died of cyanide poisoning. The Turing Award, which is the highest honor in computing, was named in his honor and has been awarded annually since 1966. It currently includes an accompanying cash prize of \$1 million.

Immediates can be written in decimal, hexadecimal, or binary. For example, the following instructions all put the decimal value 109 into s5:

```
addi s5,x0,0b1101101
addi s5,x0,0x6D
addi s5,x0,109
```

Example 6.1 TRANSLATING HIGH-LEVEL CODE TO ASSEMBLY LANGUAGE

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

```
// high-level code
a = b - c;
f = (g + h) - (i + j);
```

Solution The program uses four assembly language instructions.

```
# RISC-V 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

Table 6.1 lists the name and use for each of the 32 RISC-V registers. Registers are numbered 0 to 31 and are given a special name to indicate a register’s conventional purpose. Assembly instructions typically use the special name—for example, s1—for clarity, but they may also use the register number (e.g., x9 for register number 9). The zero register always holds the constant 0; values written to it are discarded. Registers s0 to s11 (registers 8–9 and 18–27) and t0 to t6 (registers 5–7 and 28–31) are used for storing variables; ra and a0 to a7 have special uses during function calls, as discussed in Section 6.3.7. Registers 2 to 4 are also called sp, gp, and tp and will be described later.

Constants/Immediates

In addition to register operations, RISC-V instructions can use constant or *immediate* operands. These constants are called *immediates* because their values are immediately available from the instruction and do not require a register or memory access. Code Example 6.6 shows the add immediate instruction, addi, that adds an immediate to a register. In assembly code, the immediate can be written in decimal, hexadecimal, or binary. Hexadecimal constants in RISC-V assembly language start with 0x and binary constants start with 0b, as they do in C. Immediates are 12-bit two’s complement numbers, so they are sign-extended to 32 bits. The addi instruction is a useful way to initialize register values with small constants. Code Example 6.7 initializes the variables i, x, and y to 0, 2032, –78, respectively.

Table 6.1 RISC-V register set

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary registers
s0/fp	x8	Saved register/Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments/Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporary registers

Code Example 6.6 IMMEDIATE OPERANDS

High-Level Code	RISC-V Assembly Code
<pre>a = a + 4; b = a - 12;</pre>	<pre># s0 = a, s1 = b addi s0, s0, 4 # a = a + 4 addi s1, s0, -12 # b = a - 12</pre>

Code Example 6.7 INITIALIZING VALUES USING IMMEDIATES

High-Level Code	RISC-V Assembly Code
<pre>i = 0; x = 2032; y = -78;</pre>	<pre># s4 = i, s5 = x, s6 = y addi s4, zero, 0 # i = 0 addi s5, zero, 2032 # x = 2032 addi s6, zero, -78 # y = -78</pre>

To create larger constants, use a *load upper immediate* instruction (`lui`) followed by an add immediate instruction (`addi`), as shown in [Code Example 6.8](#). The `lui` instruction loads a 20-bit immediate into the most significant 20 bits of the instruction and places zeros in the least significant bits.

The `int` data type in C represents a *signed* number, that is, a two’s complement integer. The C specification requires that `int` be *at least* 16 bits wide but does not require a particular size. Most modern compilers (including those for RV32I) use 32 bits, so an `int` represents a number in the range $[-2^{31}, 2^{31}-1]$. C also defines `int32_t` as a 32-bit two’s complement integer, but this is more cumbersome to type.

Code Example 6.8 32-BIT CONSTANT EXAMPLE

High-Level Code	RISC-V Assembly Code
<pre>int a = 0xABCDE123;</pre>	<pre>lui s2, 0xABCDE # s2 = 0xABCDE000 addi s2, s2, 0x123 # s2 = 0xABCDE123</pre>

When creating large immediates, if the 12-bit immediate in `addi` is negative (i.e., bit 11 is 1), the upper immediate in the `lui` must be incremented by one. Remember that `addi` *sign*-extends the 12-bit immediate, so a negative immediate will have all 1’s in its upper 20 bits. Because all 1’s is -1 in two’s complement, adding all 1’s to the upper immediate results in subtracting 1 from the upper immediate. [Code Example 6.9](#) shows such a case where the desired immediate is 0xFEEDA987. `lui s2, 0xFEEDB` puts 0xFEEDB000 into `s2`. The desired 20-bit upper immediate, 0xFEEDA, is incremented by 1. 0x987 is the 12-bit representation of -1657 , so `addi s2, s2, -1657` adds `s2` and the sign-extended 12-bit immediate ($0xFEEDB000 + 0xFFFFF987 = 0xFEEDA987$) and places the result in `s2`, as desired.

Code Example 6.9 32-BIT CONSTANT WITH A ONE IN BIT 11

High-Level Code	RISC-V Assembly Code
<pre>int a = 0xFEEDA987;</pre>	<pre>lui s2, 0xFEEDB # s2 = 0xFEEDB000 addi s2, s2, -1657 # s2 = 0xFEEDA987</pre>

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. Whereas the register file is small and fast, memory is larger and slower. For this reason, frequently used variables are kept in registers. In the RISC-V architecture, instructions operate exclusively on registers, so data stored in memory must be moved to a register before it can be processed. By using a combination of memory and registers, a program can access a large amount of data fairly quickly. Recall from [Section 5.5](#) that memories are organized as an array of data words. The RV32I RISC-V architecture uses 32-bit memory addresses and 32-bit data words.

RISC-V uses a *byte-addressable* memory. That is, each byte in memory has a unique address, as shown in [Figure 6.1\(a\)](#). A 32-bit word consists of four 8-bit bytes, so each word address is a multiple of 4.

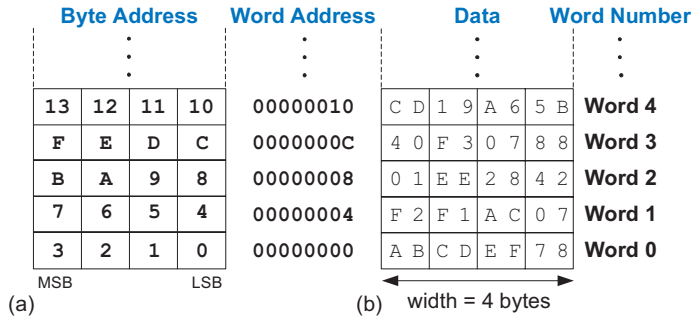


Figure 6.1 RISC-V byte-addressable memory showing: (a) byte address and (b) data

The most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. The order of bytes within a word is discussed further in [Section 6.6.1](#). Both the 32-bit word address and the data value in [Figure 6.1\(b\)](#) are given in hexadecimal. For example, data word 0xF2F1AC07 is stored at memory address 4. By convention, memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top.

The *load word* instruction, `lw`, reads a data word from memory into a register. [Code Example 6.10](#) loads memory word 2, located at address 8, into `a` (`s7`). In C, the number inside the brackets is the *index* or word number, which we discuss further in [Section 6.3.6](#). The `lw` instruction specifies the memory address using an *offset* added to a *base register*. Recall that each data word is 4 bytes, so the word address is four times the word number. Word number 0 is at address 0, word 1 is at address 4, word 2 at address 8, and so on. In this example, the base register, zero, is added to the offset, 8, to get address 8, or word 2. After the load word instruction (`lw`) is executed in [Code Example 6.10](#), `s7` holds the value 0x01EE2842, which is the data value stored at memory address 8 in [Figure 6.1](#).

Code Example 6.10 READING MEMORY

High-Level Code

```
a = mem[2];
```

RISC-V Assembly Code

```
# s7 = a
lw s7, 8(zero) # s7 = data at memory address (zero + 8)
```

The *store word* instruction, `sw`, writes a data word from a register into memory. [Code Example 6.11](#) writes the value 42 from register `t3` into memory word 5, located at address 20.

Many RISC-V implementations require *word-aligned addresses*—that is, a word address that is divisible by four—for `lw` and `sw`. Some architectures, such as x86, allow non-word-aligned data reads and writes, but others require strict alignment for simplicity. In this textbook, we will assume strict alignment. Of course, byte addresses for load byte and store byte, `lb` and `sb` (discussed in [Section 6.3.6](#)), need not be word aligned.

Code Example 6.11 WRITING MEMORY

High-Level Code	RISC-V Assembly Code
<pre>mem[5] = 42;</pre>	<pre>addi t3, zero, 42 # t3 = 42 sw t3, 20(zero) # data value at memory address 20 = 42</pre>



Katherine Johnson, 1918–2020
Creola Katherine Johnson was a mathematician and computer scientist and one of the first African American women to work at NASA. At 18 years old, she graduated summa cum laude with a bachelor’s degrees in mathematics and French from West Virginia University. When she joined NASA, she worked as a “computer,” a group of people, mostly women, who performed precise calculations. In 1961, Johnson calculated the trajectory of Alan Shepard, the first American in space. Early in NASA’s history, women were discouraged from having their names on reports, even when they did most of the work. Her NASA colleagues trusted her calculations, so Johnson helped facilitate the adoption of electronic computers by verifying their calculations. President Barack Obama awarded her the Presidential Medal of Freedom in 2015.

6.3 PROGRAMMING

Software languages such as C or Java are called *high-level programming languages* because they are written at a more abstract level than assembly language. Many high-level languages use common software constructs, such as arithmetic and logical operations, if/else statements, for and while loops, array indexing, and function calls. See [Appendix C](#) for more examples of these constructs in C. In this section, we begin by discussing program flow and instructions that support these high-level constructs. Then, we explore how to translate the high-level constructs into RISC-V assembly code.

6.3.1 Program Flow

Like data, instructions are stored in memory. Each instruction is 32 bits (4 bytes) long, as we will discuss in [Section 6.4](#), so consecutive instruction addresses increase by four. For example, in the code snippet below, the `addi` instruction is located in memory at address 0x538 and the next instruction, `lw`, is at address 0x53C.

Memory address	Instruction
0x538	<code>addi s1, s2, s3</code>
0x53C	<code>lw t2, 8(s1)</code>
0x540	<code>sw s3, 3(t6)</code>

The *program counter*—also called the PC—keeps track of the current instruction. The PC holds the memory address of the current instruction and increments by four after each instruction completes so that the processor can read or *fetch* the next instruction from memory. For example, when `addi` is executing, PC is 0x538. After `addi` completes, PC increments by four to 0x53C and the processor fetches the `lw` instruction at that address.

6.3.2 Logical, Shift, and Multiply Instructions

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

Logical Instructions

RISC-V *logical operations* include `and`, `or`, and `xor`. These each operate bitwise on two source registers and write the result to a destination register, as shown in Figure 6.2. Immediate versions of these logical operations, `andi`, `ori`, and `xori`, use one source register and a 12-bit sign-extended immediate.¹

		Source registers			
s1		0100 0110	1010 0001	1111 0001	1011 0111
s2		1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
<code>and s3, s1, s2</code>	s3	0100 0110	1010 0001	0000 0000	0000 0000
<code>or s4, s1, s2</code>	s4	1111 1111	1111 1111	1111 0001	1011 0111
<code>xor s5, s1, s2</code>	s5	1011 1001	0101 1110	1111 0001	1011 0111

Figure 6.2 Logical operations

The `and` instruction is useful for *clearing* or *masking* bits (i.e., forcing bits to 0). For example, the `and` instruction in Figure 6.2 clears the bits in `s1` that are low in `s2`. In this case, the bottom two bytes of `s1` are cleared. The unmasked top two bytes of `s1`, `0x46A1`, are placed in `s3`. Any subset of register bits can be cleared. For example, to clear bit 3 of `s0` and place the result in `s6`, use `andi s6, s0, 0xFF7`.

The `or` instruction is useful for combining bitfields from two registers. For example, `0x347A0000 OR 0x000072FC = 0x347A72FC`. It can also be used to *set* bits in a register (i.e., force a bit to 1). For example, to set bit 5 of `s0` and place the result in `s7`, use `ori s7, s0, 0x020`.

A logical NOT operation can be performed with `xori s8, s1, -1`. Remember that `-1` (`0xFFFF`) is sign-extended to `0xFFFFFFFF` (all 1's). XOR with all 1's inverts all the bits, so `s8` will get the one's complement of `s1`.

Shift Instructions

Shift instructions shift the value in a register left or right, dropping bits off the end. RISC-V 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 zeros. However, right shifts can be either *logical* (zeros shift into the most significant bits) or *arithmetic* (the sign bit shifts into the most significant bits). These shifts specify the shift amount in the second source register. Immediate versions of each instruction are also available (`slli`, `srl`, and `srai`), where the amount to shift is specified by a 5-bit unsigned immediate.

The RISC-V base instruction set does not currently include bit manipulations beyond shifts. Some instruction set architectures also include rotate instructions, as well as other bit manipulation instructions such as bit clear, bit set, etc. As of 2021, the “B” Standard Extension of RISC-V for Bit Manipulations is planned but not completed.

¹Sign-extended logical immediates are somewhat unusual. Many other architectures, such as MIPS and ARM, zero-extend the immediate for logical operations.

Figure 6.3 shows the assembly code and resulting register values for `slli`, `srl`, and `srai` when shifting by an immediate value. `s5` is shifted by the immediate amount, and the result is placed in the destination register.

Figure 6.3 Shift instructions with immediate shift amounts

		Source register			
s5		1111 1111	0001 1100	0001 0000	1110 0111
Assembly code		Result			
slli t0, s5, 7	t0	1000 1110	0000 1000	0111 0011	1000 0000
srl s1, s5, 17	s1	0000 0000	0000 0000	0111 1111	1000 1110
srai t2, s5, 3	t2	1111 1111	1110 0011	1000 0010	0001 1100

As discussed in Section 5.2.5, shifting a value left by N is equivalent to multiplying it by 2^N . For example, `slli s0, s0, 3` multiplies `s0` by 8 (i.e., 2^3). Likewise, shifting a value right by N is equivalent to dividing it by 2^N . Arithmetic right shifts divide two’s complement numbers, while logical right shifts divide unsigned numbers.

Logical shifts are also used with `and` and `or` instructions to extract or assemble bitfields. For example, the following code extracts bits 15:8 from `s7` and places them in the lower byte of `s6`. If `s7` is 0x1234ABCD, then `s6` will be 0xAB after this code completes.

```
srl s6, s7, 8
andi s6, s6, 0xFF
```

Multiply Instructions*

Multiplication is somewhat different from other arithmetic operations because multiplying two N -bit numbers produces a $2N$ -bit product. The RISC-V architecture provides various *multiply instructions* that result in 32- or 64-bit products. These instructions are not part of RV32I but are included in the RVM (RISC-V multiply/divide) extension.

The *multiply* instruction (`mul`) multiplies two 32-bit numbers and produces a 32-bit product. `mul s1, s2, s3` multiplies the values in `s2` and `s3` and places the least significant 32 bits of the product in `s1`; the most significant 32 bits of the product are discarded. This instruction is useful for multiplying small numbers whose product fits in 32 bits. The bottom 32 bits of the product are the same whether the operands are viewed as signed or unsigned.

Three versions of the “multiply high” operation exist: `mulh`, `mulhsu`, and `mulhu`. These instructions put the high 32 bits of the multiplication result in the destination register. `mulh` (*multiply high signed signed*)

treats both operands as signed. `mulhsu` (*multiply high signed unsigned*) treats the first operand as signed and the second as unsigned, and `mulhu` (*multiply high unsigned unsigned*) treats both operands as unsigned. For example, `mulhsu t1, t2, t3` treats `t2` as a 32-bit signed (two's complement) number and `t3` as a 32-bit unsigned number, multiplies these two source operands, and puts the upper 32 bits of the result in `t1`. Using a series of two instructions—one of the “multiply high” instructions followed by the `mul` instruction—will place the entire 64-bit result of the 32-bit multiplication in the two registers designated by the user. For example, the following code multiplies 32-bit signed numbers in `s3` and `s5` and places the 64-bit product in `t1` and `t2`. That is, $\{t1, t2\} = s3 \times s5$.

```
mulh t1, s3, s5
mul  t2, s3, s5
```

6.3.3 Branching

Programs would be boring if they could only run in the same order every time, independent of the input. An advantage of a computer over a calculator is its ability to make decisions. A computer performs different tasks depending on the input. For example, *if/else* statements, *switch/case* statements, *while* loops, and *for* loops all conditionally execute code depending on some test. *Branch* instructions modify the flow of the program so that the processor can fetch instructions that are not in sequential order in memory. They modify the PC to skip over sections of code or 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 RISC-V instruction set has six conditional branch instructions, each of which take two source registers and a label indicating where to go. `beq` (*branch if equal*) branches when the values in the two source registers are equal. `bne` (*branch if not equal*) branches when they are unequal. `blt` (*branch if less than*) branches when the value in the first source register is less than the value in the second, and `bge` (*branch if greater than or equal to*) branches when the first is greater than or equal to the second. `blt` and `bge` treat the operands as signed numbers, while `bltu` and `bgeu` treat the operands as unsigned.

[Code Example 6.12](#) illustrates the use of `beq`. When the program reaches the branch if equal instruction (`beq`), the value in `s0` is equal to the value in `s1`, so the branch is taken. Thus, the next instruction executed is the `add` instruction just after the label called `target`. The `addi` and `sub` instructions between the branch and the label are not executed.

There is no need for `bgt` or `ble` because these can be obtained by switching the order of the source registers of `blt` and `bge`. However, these are available as pseudoinstructions (see [Section 6.3.8](#)).

Code Example 6.12 CONDITIONAL BRANCHING USING `beq`**RISC-V Assembly Code**

```

addi s0, zero, 4      # s0 = 0 + 4 = 4
addi s1, zero, 1      # s1 = 0 + 1 = 1
slli 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:                # label
add  s1, s1, s0         # s1 = 4 + 4 = 8

```

Assembly code uses *labels* to indicate instruction locations in the program. A label refers to the instruction just after the label. When the assembly code is translated into machine code, these labels correspond to instruction addresses (as will be discussed in [Sections 6.4.3 and 6.4.4](#)). RISC-V assembly labels are followed by a colon (:). Most programmers indent their instructions but not the labels to help make labels stand out.

In [Code Example 6.13](#), the branch is not taken because `s0` is equal to `s1`, and the code continues to execute directly after the `bne` (branch if not equal) instruction. All instructions in this code snippet are executed.

Code Example 6.13 CONDITIONAL BRANCHING USING `bne`**RISC-V Assembly Code**

```

addi s0, zero, 4      # s0 = 0 + 4 = 4
addi s1, zero, 1      # s1 = 0 + 1 = 1
slli s1, s1, 2         # s1 = 1 << 2 = 4
bne  s0, s1, target    # branch not taken
addi s1, s1, 1         # s1 = 4 + 1 = 5
sub  s1, s1, s0         # s1 = 5 - 4 = 1
target:                # label
add  s1, s1, s0         # s1 = 1 + 4 = 5

```

Jumps

A program can jump—that is, unconditionally branch—using one of three instructions: *jump* (`j`), *jump and link* (`jal`), or *jump register* (`jr`). `j` jumps directly to the instruction at the specified label. [Code Example 6.14](#) shows the use of the `j` (jump) instruction to skip over three instructions and continue at the `add` instruction after the label `target`. After the `j` instruction executes, this program unconditionally continues executing the `add` instruction at the label `target`. All of the instructions between the jump and the label are skipped. We will discuss `jal` and `jr` instructions in [Section 6.3.7](#), where they are used for function calls.

Code Example 6.14 UNCONDITIONAL BRANCHING USING `j`**RISC-V Assembly Code**

```

j    target      # jump to target
srai s1, s1, 2    # not executed
addi s1, s1, 1    # not executed
sub  s1, s1, s0   # not executed
target:
add  s1, s1, s0   # s1 = s1 + s0

```

6.3.4 Conditional Statements

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

If Statements

An *if* statement executes a block of code, the *if block*, only when a condition is met. [Code Example 6.15](#) shows how to translate an if statement into RISC-V assembly code. The assembly code for the if statement tests the opposite condition of the one in the high-level code. In [Code Example 6.15](#), the high-level code tests for `apples == oranges`. The assembly code tests for `apples != oranges` using `bne` to skip the if block if the condition is not satisfied. Otherwise (i.e., when `apples == oranges`), the branch is not taken, and the if block is executed.

In C and many other high-level programming languages, the double equals sign, `==`, is an equality comparison, returning TRUE if both sides are equal. `!=` is an inequality comparison.

Code Example 6.15 IF STATEMENT**High-Level Code**

```

if (apples == oranges)
    f = g + h;
apples = oranges - h;

```

RISC-V Assembly Code

```

# s0 = apples, s1 = oranges
# s2 = f, s3 = g, s4 = h
bne s0, s1, L1 # skip if (apples != oranges)
add s2, s3, s4 # f = g + h
L1: sub s0, s1, s4 # apples = oranges - h

```

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 6.16](#) shows an example if/else statement.

Like if statements, if/else assembly code tests the opposite condition of the one in the high-level code. In [Code Example 6.16](#), the high-level code tests for `(apples == oranges)` and the assembly code tests for

Code Example 6.16 IF/ELSE STATEMENT

High-Level Code	RISC-V Assembly Code
<pre> if (apples == oranges) f = g + h; else apples = oranges - h; </pre>	<pre> # s0 = apples, s1 = oranges # s2 = f, s3 = g, s4 = h bne s0, s1, L1 # skip if (apples != oranges) add s2, s3, s4 # f = g + h j L2 L1: sub s0, s1, s4 # apples = oranges - h L2: </pre>

(apples != oranges). If that opposite condition is TRUE, bne skips the if block and executes the else block. Otherwise, the if block executes and finishes with a jump (j) past the else block.

Switch/case Statements*

Switch/case statements, also called simply *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 6.17](#) shows two high-level code snippets with the same functionality: they calculate whether to dispense \$20, \$50, or \$100 from an ATM (automatic

Code Example 6.17 SWITCH/CASE STATEMENTS

High-Level Code	RISC-V Assembly Code
<pre> switch (button) { case 1: amt = 20; break; case 2: amt = 50; break; case 3: amt = 100; break; default: amt = 0; } // equivalent function using // if/else statements if (button == 1) amt = 20; else if (button == 2) amt = 50; else if (button == 3) amt = 100; else amt = 0; </pre>	<pre> # s0 = button, s1 = amt case1: addi t0, zero, 1 # t0 = 1 bne s0, t0, case2 # button == 1? addi s1, zero, 20 # if yes, amt = 20 j done # break out of case case2: addi t0, zero, 2 # t0 = 2 bne s0, t0, case3 # button == 2? addi s1, zero, 50 # if yes, amt = 50 j done # break out of case case3: addi t0, zero, 3 # t0 = 3 bne s0, t0, default # button == 3? addi s1, zero, 100 # if yes, amt = 100 j done # break out of case default: add s1, zero, zero # amt=0 done: </pre>

teller machine) depending on the button pressed. The RISC-V assembly implementation is the same for both high-level code snippets.

6.3.5 Getting Loopy

Loops repeatedly execute a block of code, depending on a condition. While loops and for loops are commonly used in high-level languages. This section shows how to translate them into RISC-V assembly language, taking advantage of conditional branching.

While Loops

While loops repeatedly execute a block of code while a condition is met—that is, *until* a condition is *not* met. The while loop in [Code Example 6.18](#) determines the value of x such that $2^x = 128$. It executes seven times, until $\text{pow} = 128$.

Code Example 6.18 WHILE LOOP

High-Level Code	RISC-V Assembly Code
<pre>// determines the power // of x such that 2^x=128 int pow = 1; int x = 0; while (pow != 128) { pow = pow * 2; x = x + 1; }</pre>	<pre># s0 = pow, s1 = x addi s0, zero, 1 # pow = 1 add s1, zero, zero # x = 0 addi t0, zero, 128 # t0 = 128 while: beq s0, t0, done # pow = 128? slli s0, s0, 1 # pow = pow * 2 addi s1, s1, 1 # x = x + 1 j while # repeat loop done:</pre>

Like *if/else* statements, the assembly code for while loops tests the opposite condition of the one in the high-level code. If that opposite condition is TRUE (in this case, $\text{s0} == 128$), the while loop is finished. Otherwise, the branch isn't taken and the loop body executes. [Code Example 6.18](#) initializes pow to 1 and x to 0 before the while loop. The while loop compares pow to 128 and exits the loop if it is equal. Otherwise, it doubles pow (using a left shift), increments x , and branches back to the start of the while loop.

Do/while loops are similar to while loops, but they execute the body of the loop once before checking the condition. [Code Example 6.19](#) illustrates such a loop. Notice that, unlike previous examples, the branch checks the same condition as in the high-level code.

For Loops

It is very common to initialize a variable before a while loop, check that variable in the loop condition, and change that variable each time

Code Example 6.19 DO/WHILE LOOP

High-Level Code	RISC-V Assembly Code
<pre>// determines the power // of x such that 2^x = 128 int pow = 1; int x = 0; do { pow = pow * 2; x = x + 1; } while (pow != 128);</pre>	<pre># s0 = pow, s1 = x addi s0, zero, 1 # pow = 1 add s1, zero, zero # x = 0 addi t0, zero, 128 # t0 = 128 while: slli s0, s0, 1 # pow = pow * 2 addi s1, s1, 1 # x = x + 1 bne s0, t0, while # pow = 128? done:</pre>

through the while loop. *For* loops are a convenient shorthand that combines the initialization, condition check, and variable change in one place. The high-level code format of the for loop is:

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

The initialization code executes *before* the for loop begins. The condition is tested at the *beginning of each* loop iteration. If the condition is not met, the loop exits. If the condition is met, the statement (or statements) in the loop body are executed. The loop operation executes at the *end* of each loop iteration.

Code Example 6.20 adds the numbers from 0 to 9. The loop variable, in this case *i*, is initialized to 0 and is incremented at the end of each loop iteration. The for loop executes as long as *i* is less than 10. Note that this example also illustrates relative comparisons. The loop checks the < condition to continue, so the assembly code checks the opposite condition, >=, to exit the loop.

For loops are especially useful for accessing large amounts of similar data stored in memory arrays, which are discussed next.

Code Example 6.20 FOR LOOP

High-Level Code	RISC-V Assembly Code
<pre>// add the numbers from 0 to 9 int sum = 0; int i; for (i = 0; i < 10; i = i + 1) { sum = sum + i; }</pre>	<pre># s0 = i, s1 = sum addi s1, zero, 0 # sum = 0 addi s0, zero, 0 # i = 0 addi t0, zero, 10 # t0 = 10 for: bge s0, t0, done # i >= 10? add s1, s1, s0 # sum = sum + i addi s0, s0, 1 # i = i + 1 j for # repeat loop done:</pre>

6.3.6 Arrays

For ease of storage and access, similar data can be grouped together into an *array*. An array stores its contents at sequential addresses in memory. Each array element is identified by a number called its *index*. The number of elements in the array is called the *length* of the array. Figure 6.4 shows a 200-element array of integer scores stored in memory. Each consecutive element address increases by 4, the number of bytes in an integer. The address of the 0th element of an array is called the array's *base address*.

Code Example 6.21 is a grade inflation algorithm that adds 10 points to each of the scores. The code for initializing the `scores` array is not shown. Assume that `s0` is initially `0x174300A0`, the base address of the array. The index into the array is a variable (`i`) that increments by 1 for each array element, so we multiply it by 4 before adding it to the base address.

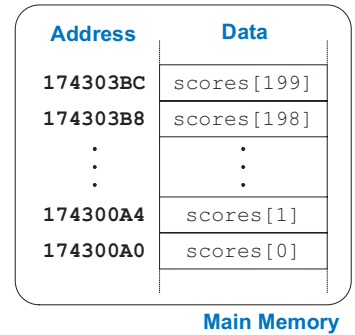


Figure 6.4 Memory holding `scores[200]` starting at base address `0x174300A0`

Code Example 6.21 USING A FOR LOOP TO ACCESS AN ARRAY

High-Level Code

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

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

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

RISC-V Assembly Code

```
# s0 = scores base address, s1 = i

addi s1, zero, 0    # i = 0
addi t2, zero, 200  # t2 = 200

for:
    bge s1, t2, done # if i >= 200 then done
    slli t0, s1, 2    # t0 = i * 4
    add t0, t0, s0     # address of scores[i]
    lw t1, 0(t0)      # t1 = scores[i]
    addi t1, t1, 10    # t1 = scores[i] + 10
    sw t1, 0(t0)      # scores[i] = t1
    addi s1, s1, 1    # i = i + 1
    j for             # repeat
done:
```

Bytes and Characters

Numbers in the range $[-128, 127]$ can be stored in a single byte rather than an entire word. Because the English language keyboard has fewer than 256 characters, English characters are often represented using 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

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

Table 6.2 ASCII encodings

#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`
21	!	31	1	41	A	51	Q	61	a
22	"	32	2	42	B	52	R	62	b
23	#	33	3	43	C	53	S	63	c
24	\$	34	4	44	D	54	T	64	d
25	%	35	5	45	E	55	U	65	e
26	&	36	6	46	F	56	V	66	f
27	'	37	7	47	G	57	W	67	g
28	(38	8	48	H	58	X	68	h
29)	39	9	49	I	59	Y	69	i
2A	*	3A	:	4A	J	5A	Z	6A	j
2B	+	3B	;	4B	K	5B	[6B	k
2C	,	3C	<	4C	L	5C	\	6C	l
2D	-	3D	=	4D	M	5D]	6D	m
2E	.	3E	>	4E	N	5E	^	6E	n
2F	/	3F	?	4F	O	5F	_	6F	o

encodings for printable characters. The ASCII values are given in hexadecimal. Lowercase and uppercase letters differ by 0x20 (32).

The *load byte* (lb), *load byte unsigned* (lbu), and *store byte* (sb) instructions access individual bytes in memory. lb sign-extends the byte, whereas lbu zero-extends the byte to fill the entire 32-bit register. sb stores the least significant byte of the 32-bit register into the specified byte address in memory. All three instructions are illustrated in Figure 6.5, with the base address, s4, being 0xD0. lbu s1, 2(s4) loads the byte at memory address 0xD2 into the least significant byte of s1 and fills the remaining register bits with 0. lb s2, 3(s4) loads the byte at memory address 0xD3 into the least significant byte of s2 and sign-extends the byte into the upper 24 bits of the register. sb s3, 1(s4) stores the least significant byte of s3 (0x9B) into memory byte address 0xD1; it replaces 0x42 with 0x9B. No other memory bytes are changed, and the more significant bytes of s3 are ignored.

RISC-V also defines lh, lhu, and sh *half-word* loads and stores that operate on 16-bit data. Memory addresses for these instructions must be half-word aligned.

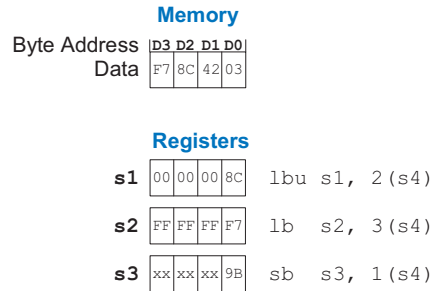


Figure 6.5 Instructions for loading and storing bytes

A series of characters, such as a word or sentence, 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 6.6 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).

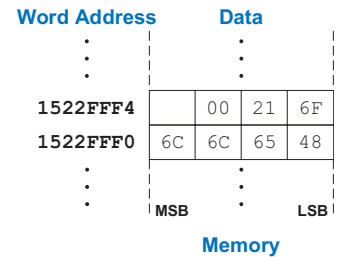


Figure 6.6 The string “Hello!” stored in memory

Example 6.2 USING lb AND sb TO ACCESS A CHARACTER ARRAY

The following high-level code converts a 10-entry array of characters from lowercase to uppercase by subtracting 32 from each array entry. Translate it into RISC-V assembly language. Remember that array elements are now 1 byte, not 4 bytes, so consecutive elements are in consecutive addresses. Assume that s0 already holds the base address of chararray.

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

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

Solution

```
# RISC-V assembly code
# s0 = base address of chararray (initialized earlier), s1 = i
addi s1, zero, 0           # i = 0
addi t3, zero, 10          # t3 = 10
for: bge s1, t3, done       # i >= 10 ?
    add t4, s0, s1          # t4 = address of chararray[i]
    lb t5, 0(t4)            # t5 = chararray[i]
    addi t5, t5, -32        # t5 = chararray[i] - 32
    sb t5, 0(t4)            # chararray[i] = t5
    addi s1, s1, 1          # i = i + 1
    j for                  # repeat loop
done:
```

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 keyboard characters, but 8-bit encoding was. Thus, as electronic communication became prevalent, 8-bit ASCII encoding emerged as the standard.

6.3.7 Function Calls

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

When one function calls another, the calling function, the *caller*, and the called function, the *callee*, must agree on where to put the arguments and the return value. In RISC-V programs, the caller conventionally places up to eight arguments in registers a0 to a7 before making the function call, and the callee places the return value in register a0 before finishing. By following this convention, both functions know where to find the arguments and return value, even if the caller and callee were written by different people.

The callee must not interfere with the behavior of the caller. This means that the callee must know where to return to after it completes and it must not trample on any registers or memory needed by the caller. The caller stores the return address in the *return address register* 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-s11), the return address (ra), and the *stack*, a portion of memory used for temporary variables, unmodified.

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

Function Calls and Returns

RISC-V uses the *jump and link* instruction (jal) to call a function and the *jump register* instruction (jr) to return from a function. [Code Example 6.22](#) shows the main function calling the simple function. main is the caller and simple is the callee. The simple function

RISC-V actually provides two registers for the return value, a0 and a1. This allows for 64-bit return values, such as int64_t.

Code Example 6.22 simple FUNCTION CALL

High-Level Code	RISC-V Assembly Code
<pre>int main() { simple(); ... } // void means the function // returns no value void simple() { return; }</pre>	<pre>0x00000300 main: jal simple # call function 0x00000304 0x0000051c simple: jr ra # return</pre>

is called with no input arguments and generates no return value; it just returns to the caller. In [Code Example 6.22](#), example instruction addresses are given to the left of each RISC-V instruction.

`jal` and `jr ra` are the two essential instructions needed for a function call and return. In [Code Example 6.22](#), the `main` function calls the `simple` function by executing `jal simple`, which performs two tasks: it jumps to the target instruction located at `simple` (`0x0000051C`) and it stores the *return address*, the address of the instruction after `jal` (in this case, `0x00000304`) in the return address register (`ra`). The programmer can specify which register gets written with the return address, but the default is `ra`. So, `jal simple` is equivalent to `jal ra, simple` and is the preferred style. The `simple` function returns immediately by executing the instruction `jr ra`, which jumps to the instruction address held in `ra`. The `main` function then continues executing at this address (`0x00000304`).

The instruction address of the currently executing instruction is held in `PC`, the program counter. So, the following instruction address is referred to as `PC+4`.

Input Arguments and Return Values

The `simple` function in [Code Example 6.22](#) is not very useful because it receives no input from the calling function (`main`) and returns no output. By RISC-V convention, functions use `a0` to `a7` for input arguments and `a0` for the return value. In [Code Example 6.23](#), the function `diffofsums` is called with four arguments and returns one result. `result` is a local variable, which we choose to keep in `s3`. (Saving and restoring registers will be discussed soon.)

According to RISC-V convention, the calling function, `main`, places the function arguments from left to right into the input registers, `a0` to `a7`, before calling the function. The called function, `diffofsums`, stores

`j` and `jr` are *pseudoinstructions*. They are not part of the instruction set but are convenient for programmers to use. The RISC-V assembler replaces them with actual RISC-V instructions. The assembler replaces `j target` with `jal zero, target`, which jumps and discards the return address by writing it to the zero register; and the assembler replaces `jr ra` with `jalr zero, ra, 0`.

The jump and link register instruction (`jalr`) is like `jal`, but it takes the destination address from a register, optionally added to a 12-bit signed immediate. For example, `jalr ra, s1, 0x4C` jumps to address `s1 + 0x4C` and puts `PC+4` in `ra`.

Code Example 6.23 FUNCTION CALL WITH ARGUMENTS AND RETURN VALUES

High-Level Code	RISC-V Assembly Code
<pre>int main(){ int y; ... y = diffofsums(2, 3, 4, 5); ... }</pre> <pre>int diffofsums(int f, int g, int h, int i){ int result; result = (f + g) - (h + i); return result; }</pre>	<pre># s7 = y main: ... addi a0, zero, 2 # argument 0 = 2 addi a1, zero, 3 # argument 1 = 3 addi a2, zero, 4 # argument 2 = 4 addi a3, zero, 5 # argument 3 = 5 jal diffofsums # call function add s7, a0, zero # y = returned value ... # s3 = result diffofsums: add t0, a0, a1 # t0 = f+g add t1, a2, a3 # t1 = h+i sub s3, t0, t1 # result = (f+g)-(h+i) add a0, s3, zero # put return value in a0 jr ra # return to caller</pre>

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

the return value in the return register, *a0*. When a function with more than eight arguments is called, the additional input arguments are placed on the stack, which we discuss next.

The Stack

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

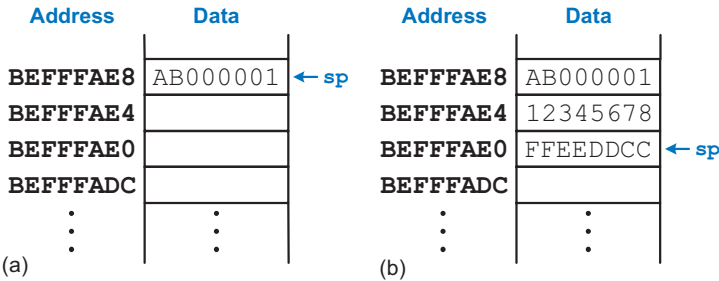
The stack is a last-in-first-out (LIFO) queue. Like a stack of dishes, the last item *pushed* (or placed) onto the stack (the top dish) is the first one that can be *popped* off (removed). Each function may allocate stack space to store local variables and to use as scratch space, but the function must 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 RISC-V stack grows down in memory. That is, the stack expands to lower memory addresses when a program needs more scratch space.

Figure 6.7 shows a picture of the stack. The stack pointer, *sp* (register 2), is an ordinary RISC-V register that, by convention, *points* to the *top of the stack*. A pointer is a fancy name for a memory address. *sp* points to (gives the address of) data. For example, in Figure 6.7(a), the stack pointer, *sp*, holds the address value 0xBEFFFAE8 and points to the data value 0xAB000001.

The stack pointer (*sp*) starts at a high memory address and decrements to expand as needed. Figure 6.7(b) shows the stack expanding to allow two more data words of temporary storage. To do so, *sp* decrements by eight to become 0xBEFFFAE0. Two additional data words, 0x12345678 and 0xFFEEDDCC, are temporarily stored on the stack.

One of the important uses of the stack is to save and restore registers that are used by a function. Recall that a function should calculate a return value but have no other unintended side effects. In particular, a

Figure 6.7 The stack (a) before expansion and (b) after two-word expansion



function should not modify any registers besides `a0`, the one containing the return value. The `diffofsums` function in [Code Example 6.23](#) violates this rule because it modifies `t0`, `t1`, and `s3`. If `main` had been using these registers before the call to `diffofsums`, their contents would have been corrupted by the function call.

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

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

[Code Example 6.24](#) shows an improved version of `diffofsums` that saves and restores `t0`, `t1`, and `s3`. [Figure 6.8](#) shows the stack before, during,

Code Example 6.24 FUNCTION THAT SAVES REGISTERS ON THE STACK

High-Level Code

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

    result = (f + g) - (h + i);

    return result;
}
```

RISC-V Assembly Code

```
# s3 = result
diffofsums:
    addi sp, sp, -12    # make space on stack to
                        # store three registers
    sw    s3, 8(sp)     # save s3 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   s3, t0, t1    # result = (f + g) - (h + i)
    add   a0, s3, zero   # put return value in a0
    lw    s3, 8(sp)     # restore s3 from stack
    lw    t0, 4(sp)     # restore t0 from stack
    lw    t1, 0(sp)     # restore t1 from stack
    addi  sp, sp, 12    # deallocate stack space
    jr    ra            # return to caller
```

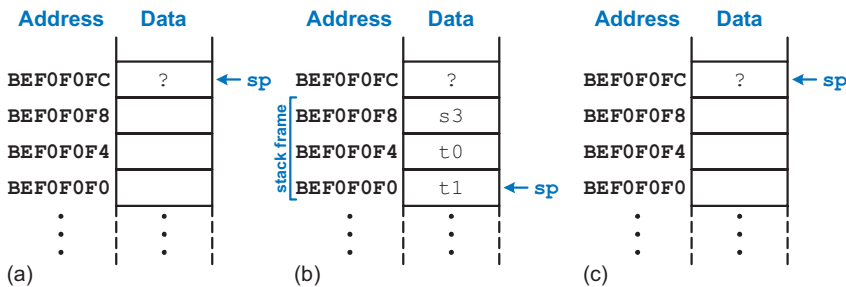


Figure 6.8 The stack: (a) before, (b) during, and (c) after the `diffofsums` function call

Saving a register value on the stack is called *pushing* a register onto the stack. Restoring the register value from the stack is called *poping* a register off of the stack.

and after a call to the `diffofsums` function from [Code Example 6.24](#). The stack starts at `0xBEF0F0FC`. `diffofsums` makes room for three words on the stack by decrementing the stack pointer `sp` by 12. It then stores the current values held in `t0`, `t1`, and `s3` in the newly allocated space. It executes the rest of the function, changing the values in these three registers. At the end of the function, `diffofsums` restores the values of these registers from the stack, deallocates its stack space, and returns. When the function returns, `a0` holds the result, but there are no other side effects: `t0`, `t1`, `s3`, and `sp` have the same values as they did before the function call.

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

Preserved Registers

[Code Example 6.24](#) assumes that all of the used registers (`t0`, `t1`, and `s3`) must be saved and restored. If the calling function does not use those registers, the effort to save and restore them is wasted. To avoid this waste, RISC-V divides registers into *preserved* and *nonpreserved* categories. Preserved registers must contain the same values at the beginning and end of a called function because the caller expects preserved register values to be the same after the call.

The preserved registers are `s0` to `s11` (hence their name, *saved*), `sp`, and `ra`. The nonpreserved registers, also called *scratch* registers, are `t0` to `t6` (hence their name, *temporary*) and `a0` to `a7`, the argument registers. A function can change the nonpreserved registers freely but must save and restore any of the preserved registers that it uses.

[Code Example 6.25](#) shows a further improved version of `diffofsums` that saves only `s3` on the stack. `t0` and `t1` are nonpreserved registers, so they need not be saved.

Code Example 6.25 FUNCTION THAT SAVES PRESERVED REGISTERS ON THE STACK

RISC-V Assembly Code

```
# s3 = result
diffofsums:
    addi sp, sp, -4      # make space on stack to store one register
    sw   s3, 0(sp)      # save s3 on stack
    add  t0, a0, a1      # t0 = f + g
    add  t1, a2, a3      # t1 = h + i
    sub  s3, t0, t1      # result = (f + g) - (h + i)
    add  a0, s3, zero    # put return value in a0
    lw   s3, 0(sp)      # restore s3 from stack
    addi sp, sp, 4      # deallocate stack space
    jr   ra             # return to caller
```

Table 6.3 Preserved and nonpreserved registers and memory

Preserved (<i>callee-saved</i>)	Nonpreserved (<i>caller-saved</i>)
Saved registers: <code>s0-s11</code>	Temporary registers: <code>t0-t6</code>
Return address: <code>ra</code>	Argument registers: <code>a0-a7</code>
Stack pointer: <code>sp</code>	
Stack above the stack pointer	Stack below the stack pointer

Because a callee function may freely change any nonpreserved registers, the caller must save any nonpreserved registers containing essential information before making a function call and then restore these registers afterward. For these reasons, preserved registers are also called *callee-saved* and nonpreserved registers are called *caller-saved*.

Table 6.3 summarizes which registers are preserved. The convention of which registers are preserved or not preserved is part of the standard calling convention² for the RISC-V Architecture, instead of being part of the architecture itself.

`s0` to `s11` are generally used to hold local variables within a function, so they must be saved. `ra` must also be saved so that the callee knows where to return. `t0` to `t6` are used to hold temporary results. These calculations typically complete before a function call is made, so they are not preserved across a function call, and it is rare that the caller needs to save them. `a0` to `a7` are often overwritten in the process of calling a function. Hence, they must be saved by the caller if the caller depends on any of its own arguments after a called function returns.

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

The astute reader or an optimizing compiler may notice that `diffofsums`' local variable, `result`, is immediately returned without being used for anything else. Hence, we can eliminate the variable and simply store the calculation directly in the return register `a0`, eliminating the need to both allocate space on the stack frame and move the result from `s3` to `a0`. Code Example 6.26 shows this even further optimized `diffofsums` function.

²From the RISC-V *Instruction Set Manual*, Volume I, version 2.2 © 2017.

Code Example 6.26 OPTIMIZED `diffofsums` FUNCTION**RISC-V Assembly Code**

```

# a0 = result
diffofsums:
    add t0, a0, a1 # t0 = f + g
    add t1, a2, a3 # t1 = h + i
    sub a0, t0, t1 # result = (f + g) - (h + i)
    jr  ra         # return to caller

```

Nonleaf Function Calls

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

Caller save rule: Before a function call, the caller must save any nonpreserved registers (`t0-t6` and `a0-a7`) that it needs after the call. After the call, it must restore these registers before using them.

Callee save rule: Before a callee disturbs any of the preserved registers (`s0-s11` and `ra`), it must save the registers. Before it returns, it must restore these registers.

[Code Example 6.27](#) shows a nonleaf function `f1` and a leaf function `f2`, including all of the necessary saving and preserving of registers. `f1` keeps `i` in `s4` and `x` in `s5`; `f2` keeps `r` in `s4`. `f1` uses preserved registers `s4`, `s5`, and `ra`, so it initially pushes them onto the stack according to the callee save rule. It uses `t3` to hold the intermediate result $(a-b)$ so that it does not need to preserve another register for this calculation. Before calling `f2`, `f1` saves `a0` and `a1` onto the stack according to the caller save rule because these are nonpreserved registers that `f2` might change and that `f1` will still need after the call. `ra` changes because it is overwritten by the call to `f2`. Although `t3` is also a nonpreserved register that `f2` could overwrite, `f1` no longer needs `t3` and does not have to save it. `f1` then passes the argument to `f2` in `a0`, makes the function call, and gets the result in `a0`. `f1` then restores `a0` and `a1` because it still needs them. When `f1` is done, it puts the return value in `a0`, restores registers `s4`, `s5`, `ra`, and `sp`, and returns. `f2` saves and restores `s4` (and `sp`) according to the callee save rule.

On careful inspection, one might note that `f2` does not modify `a1`, so `f1` did not need to save and restore it. However, a compiler cannot always easily ascertain which nonpreserved registers may be disturbed during a function call. Hence, a simple compiler will always make the caller save and restore any nonpreserved registers that it needs after the call. An

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

Code Example 6.27 NONLEAF FUNCTION CALL

High-Level Code	RISC-V Assembly Code
<pre> int f1(int a, int b) { int i, x; x = (a + b) * (a - b); for (i = 0; i < a; i++) x = x + f2(b + i); return x; } </pre>	<pre> # a0 = a, a1 = b, s4 = i, s5 = x f1: addi sp, sp, -12 # make room on stack for 3 registers sw ra, 8(sp) # save preserved registers used by f1 sw s4, -4(sp) sw s5, 0(sp) add s5, a0, a1 # x = (a + b) sub t3, a0, a1 # temp = (a - b) mul s5, s5, t3 # x = x * temp = (a + b) * (a - b) addi s4, zero, 0 # i = 0 for: bge s4, a0, return # if i >= a, exit loop addi sp, sp, -8 # make room on stack for 2 registers sw a0, 4(sp) # save nonpreserved regs. on stack sw a1, 0(sp) add a0, a1, s4 # argument is b + i jal f2 # call f2(b + i) add s5, s5, a0 # x = x + f2(b + i) lw a0, 4(sp) # restore nonpreserved registers lw a1, 0(sp) addi sp, sp, 8 addi s4, s4, 1 # i++ j for # continue for loop return: add a0, zero, s5 # return value is x lw ra, 8(sp) # restore preserved registers lw s4, 4(sp) lw s5, 0(sp) addi sp, sp, 12 # restore sp jr ra # return from f1 # a0 = p, s4 = r f2: addi sp, sp, -4 # save preserved regs. used by f2 sw s4, 0(sp) addi s4, a0, 5 # r = p + 5 add a0, s4, a0 # return value is r + p lw s4, 0(sp) # restore preserved registers addi sp, sp, 4 # restore sp jr ra # return from f2 </pre>

optimizing compiler could observe that `f2` is a leaf procedure and could allocate `r` to a nonpreserved register, avoiding the need to save and restore `s4`. [Figure 6.9](#) shows the stack during execution of the functions. For this example, the stack pointer originally starts at `0xBEF7FF0C`.

Recursive Function Calls

A *recursive function* is a nonleaf function that calls itself. Recursive functions behave as both caller and callee and must save both preserved and nonpreserved registers. For example, the factorial function can be written as a recursive function. Recall that $factorial(n) = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$. The factorial function can be written recursively as $factorial(n) = n \times factorial(n - 1)$, as shown in [Code Example 6.28](#). The

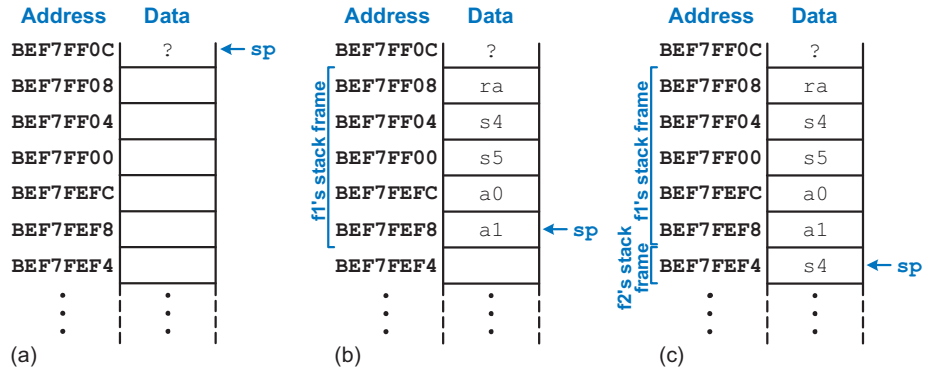


Figure 6.9 The stack: (a) before function calls, (b) during f_1 , and (c) during f_2

Code Example 6.28 factorial RECURSIVE FUNCTION CALL

High-Level Code	RISC-V Assembly Code	
<pre> int factorial(int n) { if (n <= 1) return 1; else return (n * factorial(n - 1)); } </pre>	<pre> 0x8500 factorial: addi sp, sp, -8 # make room for a0, ra 0x8504 sw a0, 4(sp) 0x8508 sw ra, 0(sp) 0x850C addi t0, zero, 1 # temporary = 1 0x8510 bgt a0, t0, else # if n > 1, go to else 0x8514 addi a0, zero, 1 # otherwise, return 1 0x8518 addi sp, sp, 8 # restore sp 0x851C jr ra # return 0x8520 else: addi a0, a0, -1 # n = n - 1 0x8524 jal factorial # recursive call 0x8528 lw t1, 4(sp) # restore n into t1 0x852C lw ra, 0(sp) # restore ra 0x8530 addi sp, sp, 8 # restore sp 0x8534 mul a0, t1, a0 # a0 = n * factorial(n - 1) 0x8538 jr ra # return </pre>	

factorial of 1 is simply 1. To conveniently refer to program addresses, we show the program starting at address 0x8500. According to the callee save rule, `factorial` is a nonleaf function and must save `ra`. According to the caller save rule, `factorial` will need `n` after calling itself, so it must save `a0`. Hence, it pushes both registers onto the stack at the start. It then checks whether $n \leq 1$. If so, it puts the return value of 1 in `a0`, restores the stack pointer, and returns to the caller. It does not have to restore `ra` in this case, because it was never modified. If $n > 1$, the function recursively calls `factorial(n-1)`. It then restores the value of `n` and the return address register (`ra`) from the stack, performs the multiplication, and returns this result. Notice that the function cleverly restores `n` into `t1` so as not to overwrite the returned value. The multiply instruction (`mul a0, t1, a0`) multiplies `n` (`t1`) and the returned value (`a0`) and puts the result in `a0`, the return register.

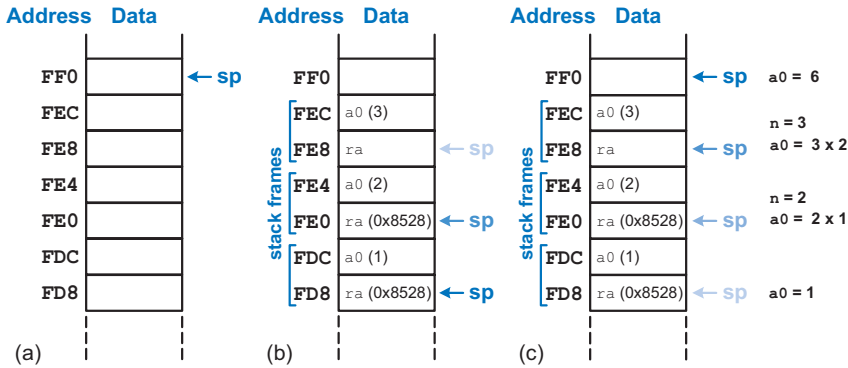


Figure 6.10 Stack: (a) before, (b) during, and (c) after factorial function call with $n = 3$

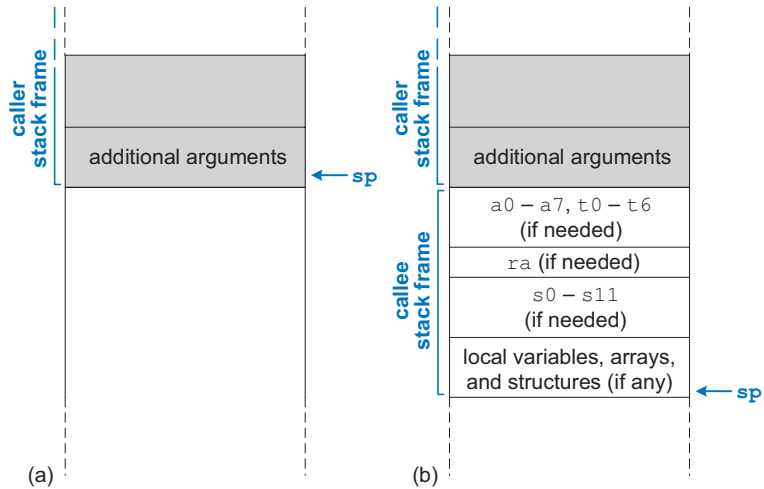
For clarity, we save registers at the start of a function call. An optimizing compiler might observe that there is no need to save $a0$ and ra when $n \leq 1$ and, thus, save registers on the stack only in the else portion of the function.

Figure 6.10 shows the stack when executing `factorial(3)`. For illustration, we show sp initially pointing to $0xFF0$ (the upper address bits are 0), as shown in Figure 6.10(a). The function creates a two-word stack frame to hold n ($a0$) and ra . On the first invocation, `factorial` saves $a0$ (holding $n = 3$) at $0xFEC$ and ra at $0xFE8$, as shown in Figure 6.10(b). The function then changes n to 2 and recursively calls `factorial(2)`, making ra hold $0x8528$. On the second invocation, it saves $a0$ (holding $n = 2$) at $0xFE4$ and ra at $0xFE0$. This time, we know that ra contains $0x8528$. The function then changes n to 1 and recursively calls `factorial(1)`. On the third invocation, it saves $a0$ (holding $n = 1$) at $0xFDC$ and ra at $0xFD8$. This time, ra again contains $0x8528$. The third invocation of `factorial` returns the value 1 in $a0$ and deallocates the stack frame before returning to the second invocation. The second invocation restores n (into $t1$) to 2, restores ra to $0x8528$ (it happened to already have this value), deallocates the stack frame, and returns $a0 = 2 \times 1 = 2$ to the first invocation. The first invocation restores n (into $t1$) to 3, restores ra , the return address of the caller, deallocates the stack frame, and returns $a0 = 3 \times 2 = 6$. Figure 6.10(c) shows the stack as the recursively called functions return. When `factorial` returns to the caller, the stack pointer is in its original position ($0xFF0$), none of the contents of the stack above the pointer have changed, and all of the preserved registers hold their original values. $a0$ holds the return value, 6.

Additional Arguments and Local Variables*

Functions may have more than eight input arguments and may have too many local variables to keep in preserved registers. The stack is used to

Figure 6.11 Expanded stack frame with additional arguments
(a) before call, (b) after call



store this information. By RISC-V convention, if a function has more than eight arguments, the first eight are passed in the argument registers ($a0 - a7$) 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 6.11(a) shows the caller's stack for calling a function with more than eight arguments.

A function can also declare local variables or arrays. Local variables are declared within a function and can be accessed only within that function. Local variables are stored in $s0$ to $s11$; if a function has too many local variables, they can also be stored in the function's stack frame. Local arrays and structures are also stored on the stack.

Figure 6.11(b) shows the organization of a callee's stack frame. The stack frame holds the temporary, argument, and return address registers (if they need to be saved because of a subsequent function call), and any of the saved registers that the function will modify. It also holds local arrays and any excess local variables. If the callee has more than eight arguments, it finds them in the caller's stack frame. Accessing additional input arguments is the one exception in which a function can access stack data not in its own stack frame.

Some functions also include a *frame pointer* that points to the *bottom* of the active stack frame – the stack frame of the executing function. By convention, this address is held in the fp register ($x8$), which is also a preserved register.

6.3.8 Pseudoinstructions

Before we show how to convert assembly code into machine code, 1's and 0's, let us revisit pseudoinstructions. Remember that RISC-V is a reduced instruction set computer (RISC), so the instruction size and hardware complexity are minimized by keeping the number of instructions small. However, RISC-V defines pseudoinstructions that are not actually part of the RISC-V instruction set but that are commonly used by programmers and compilers. When converted to machine code, pseudoinstructions are

Table 6.4 Pseudoinstructions

Pseudoinstruction	RISC-V Instructions	Description	Operation
j label	jal zero, label	jump	PC = label
jr ra	jalr zero, ra, 0	jump register	PC = ra
mv t5, s3	addi t5, s3, 0	move	t5 = t3
not s7, t2	xori s7, t2, -1	one's complement	s7 = ~t2
nop	addi zero, zero, 0	no operation	
li s8, 0x7EF	addi s8, zero, 0x7EF	load 12-bit immediate	s8 = 0x7EF
li s8, 0x56789DEF	lui s8, 0x5678A addi s8, s8, 0xDEF	load 32-bit immediate	s8 = 0x56789DEF
bgt s1, t3, L3	blt t3, s1, L3	branch if >	if (s1 > t3), PC = L3
bgez t2, L7	bge t2, zero, L7	branch if ≥ 0	if (t2 ≥ 0), PC = L7
call L1	jal L1	call nearby function	PC = L1, ra = PC + 4
call L5	auipc ra, imm _{31:12} jalr ra, ra, imm _{11:0}	call far away function	PC = L5, ra = PC + 4
ret	jalr zero, ra, 0	return from function	PC = ra

translated into one or more RISC-V instructions. For example, we have already discussed the jump (j) pseudoinstruction that is converted to the jump and link (jal) instruction with x0 as the destination address—that is, no return address is written. We also noted that logical NOT can be performed by XORing the source operand with all 1's.

Table 6.4 gives examples of pseudoinstructions and the RISC-V instructions used to implement them. For example, the move instruction (mv) copies the contents of one register to another register. The load immediate pseudoinstruction (li) loads a 32-bit constant using a combination of the lui and addi instructions. If the constant can be represented in 12 bits, li is translated into an addi instruction. The no operation pseudoinstruction (nop, pronounced “no op”) performs no operation. The PC is incremented by 4 upon its execution, but no other registers or memory values are altered. The call pseudoinstruction makes a procedure call. If the call is to a nearby function, call is translated into a jalr instruction. However, if the function is far away, call is translated into two RISC-V instructions: auipc and jalr. For example, auipc s1, 0xABCDE adds 0xABCDE000 to the PC and puts the result in s1. So, if PC is 0x02000000, then s1 now holds 0xADCDE000. jalr ra, s1, 0x730 then jumps to address s1 + 0x730 (0xADCDE730) and puts PC+4 in ra. The ret pseudoinstruction returns from a function.

Nops are commonly used to generate precise delays in a program.

It translates into `jalr x0, ra, 0`. Table B.7 in Appendix B lists the most common RV32I pseudoinstructions. Appendix B is printed on the inside covers of this textbook.

6.4 MACHINE LANGUAGE

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

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

Design Principle 4: Good design demands good compromises.

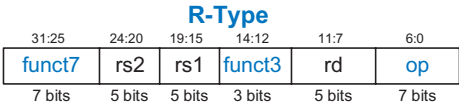
RISC-V makes the compromise of defining four main instruction formats: *R-type*, *I-type*, *S/B-type*, and *U/J-type*. This small number of formats allows for some regularity among instructions and, thus, simpler decoder hardware, while also accommodating different instruction needs. *R-type* (*register*) instructions, such as `add s0, s1, s2`, operate on three registers. *I-type* (*immediate*) instructions, such as `addi s3, s4, 42`, and *S/B-type* (*store/branch*) instructions, such as `sw a0, 4(sp)` or `beq a0, a1, L1`, operate on two registers and a 12- or 13-bit signed immediate. *U/J-type* (*upper immediate/jump*) instructions, such as `jal ra, factorial`, operate on one register and a 20- or 21-bit immediate. This section discusses these RISC-V machine instruction formats and shows how they are encoded into binary. Appendix B provides a quick reference for all RV32I instructions.

S/B-type is notably not called B/S-type.

6.4.1 R-Type Instructions

R-type (*register-type*) instructions use three registers as operands: two as sources and one as a destination. Figure 6.12 shows the R-type machine

Figure 6.12 R-type instruction format



Assembly	Field Values						Machine Code						
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op	
add s2, s3, s4	0	20	19	0	18	51	0000,000	10100	1001,1	000,	10010	011,0011	(0x01498933)
add x18, x19, x20													
sub t0, t1, t2	32	7	6	0	5	51	0100,000	00111	0011,0	000,	00101	011,0011	(0x407302B3)
sub x5, x6, x7													
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Figure 6.13 Machine code for R-type instructions

instruction format. The 32-bit instruction has six fields: **funct7**, **rs2**, **rs1**, **funct3**, **rd**, and **op**. Each field is three to seven bits, as indicated.

The operation the instruction performs is encoded in the three fields highlighted in blue: 7-bit **op** (also called opcode or operation code) and 7- and 3-bit **funct7** and **funct3** (also called the function fields). The specific R-type operation is determined by the opcode and the function fields. These bits together are called the *control bits* because they control what operation to perform. For example, the opcode and function fields for the `add` instruction are **op** = 51 (0110011₂), **funct7** = 0 (0000000₂) and **funct3** = 0 (000₂). Similarly, the `sub` instruction has **op** = 51, **funct7** = 32 (0100000₂), and **funct3** = 0 (000₂). Figure 6.13 shows the machine code for two R-type instructions, `add` and `sub`. The two source registers and the destination register are encoded in the three fields: **rs1**, **rs2**, and **rd**. The fields contain the register numbers that were given in Table 6.1. For example, `s0` is register 8 (x8). Notice that the registers are in the opposite order in the assembly and machine language instructions. For example, the assembly instruction `add s2, s3, s4` has **rd** = `s2` (18), **rs1** = `s3` (19), and **rs2** = `s4` (20). These registers are listed left to right in the assembly instruction but right to left in the machine instruction.

Table B.1 in Appendix B lists the opcode and the function fields (**funct3** and **funct7**) for RV32I instructions. The easiest way to translate from assembly to machine code (as shown in Figure 6.13) is to write out the values of each field and convert these values to binary. Then, group the bits into blocks of four to convert to hexadecimal and make the machine language representation more compact.

Other R-type instructions include shifts (`sll`, `srl`, and `sra`) and logical operations (`and`, `or`, and `xor`). Shift instructions with an immediate shift amount (`slli`, `srl`, and `srai`) are I-type instructions, which are discussed next in Section 6.4.2.

Figure 6.14 shows the machine code for shift left logical (`sll`) and `xor`. The opcode is 51 (0110011₂) for all R-type operations. Shift instructions with a register shift amount (`sll`, `srl`, and `sra`), shift **rs1** by the unsigned 5-bit value in bits 4:0 of register **rs2**, and place the result in **rd**. For all shift instructions, **funct7** and **funct3** encode the type of shift or logical operation to perform, as given in Table B.1. For `sll`, **funct7** = 0 and **funct3** = 1; `xor` uses **funct7** = 0 and **funct3** = 4.

Appendix B is located on the inside covers of this book.

Assembly	Field Values						Machine Code					
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op
sll s7, t0, s1	0	9	5	1	23	51	0000 000	01001	00101	001	10111	011 0011
sll x23, x5, x9	0	26	25	4	24	51	0000 000	11010	11001	100	11000	011 0011
xor s8, s9, s10												
xor x24, x25, x26												
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Figure 6.14 More machine code for R-type instructions

R-type instructions have 17 bits of **op** and **funct** codes, enough to represent $2^{17} = 131,072$ different instructions. This seems grossly excessive, considering we have defined less than a dozen R-type instructions so far. However, only 15 other bits are needed to encode the source and destination registers. This large instruction set space makes RISC-V highly extensible. For example, the *RISC-V F Extension* adds floating-point instructions, described further in Section 6.6.4 and Appendix B.

Example 6.3 TRANSLATING R-TYPE ASSEMBLY INSTRUCTIONS INTO MACHINE CODE

Translate the following RISC-V assembly instruction into machine language:

```
add t3, s4, s5
```

Solution According to Table 6.1, t3, s4, and s5 are registers 28, 20, and 21. According to Table B.1, add has an opcode of 51 (0110011₂) and function codes of funct7 = 0 and funct3 = 0. Thus, the fields and machine code are given in Figure 6.15. 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 by blue underbars). Hence, the machine language instruction is 0x015A0E33.

Assembly	Field Values						Machine Code					
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op
add t3, s4, s5	0	21	20	0	28	51	0000 000	10101	10100	000	11100	011 0011
add x28, x20, x21												
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

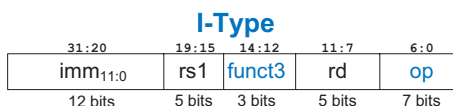
Figure 6.15 Machine code for the R-type instruction of Example 6.3

6.4.2 I-Type Instructions

I-type (*immediate*) instructions use two register operands and one immediate operand. I-type instructions include `addi`, `andi`, `ori`, and `xori`, loads (`lw`, `lh`, `lb`, `lhu`, and `lbu`), and register jumps (`jalr`). Figure 6.16 shows the I-type machine instruction format. It is similar to R-type but includes a 12-bit immediate field **imm** instead of the **funct7** and **rs2** fields. **rs1** and **imm** are the source operands, and **rd** is the destination register.

Figure 6.17 shows several examples of encoding I-type instructions. The immediate field represents a 12-bit signed (two's complement)

Figure 6.16 I-type instruction format



Assembly	Field Values					Machine Code					
	imm _{11:0}	rs1	funct3	rd	op	imm _{11:0}	rs1	funct3	rd	op	
addi s0, s1, 12	12	9	0	8	19	0000 0000 1100	01001	000	01000	001 0011	(0x00C48413)
addi x8, x9, 12											
addi s2, t1, -14	-14	6	0	18	19	1111 1111 0010	00110	000	10010	001 0011	(0xFF230913)
addi x18, x6, -14											
lw t2, -6(s3)	-6	19	2	7	3	1111 1111 1010	10011	010	00111	000 0011	(0xFFA9A383)
lw x7, -6(x19)											
lb s4, 0x1F(s4)	0x1F	20	0	20	3	0000 0001 1111	10100	000	10100	000 0011	(0x01FA0A03)
lb x20, 0x1F(x20)											
slli s2, s7, 5	5	23	1	18	19	0000 0000 0101	10111	001	10010	001 0011	(0x005B9913)
slli x18, x23, 5											
srai t1, t2, 29	(upper 7 bits = 32) 29	7	5	6	19	0100 0001 1101	00111	101	00110	001 0011	(0x41D3D313)
srai x6, x7, 29											
	12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Figure 6.17 Machine code for I-type instructions

number for all I-type instructions except immediate shift instructions (`slli`, `srli`, and `srai`). For these shift instructions, `imm4:0` is the 5-bit unsigned shift amount; the upper seven `imm` bits are 0 for `srli` and `slli`, but `srai` puts a 1 in `imm10` (i.e., instruction bit 30), as shown in Figure 6.17. As with R-type instructions, the order of the operands in the I-type assembly instructions differs from that of the machine instruction.

Example 6.4 TRANSLATING I-TYPE ASSEMBLY INSTRUCTIONS INTO MACHINE CODE

Translate the following assembly instruction into machine language.

```
lw t3, -36(s4)
```

Solution According to Table 6.1, `t3`, and `s4` are registers 28 and 20. `rs1` (`s4` = `x20`) specifies the base address, and `rd` (`t3` = `x28`) specifies the destination. The immediate, `imm`, encodes the 12-bit offset (`-36`). Table B.1 indicates that `lw` has an `op` of 3 (`00000112`) and `funct3` of 2 (`0102`). The fields and machine code are given in Figure 6.18.

I-type instructions have a 12-bit immediate field, but the immediates are used in 32-bit operations. For example, `lw` adds a 12-bit offset to a 32-bit base register. What should go in the upper 20 bits of the 32 bits? For positive immediates, the upper bits should be all 0's, but for negative

Assembly	Field Values					Machine Code					
	imm _{11:0}	rs1	funct3	rd	op	imm _{11:0}	rs1	funct3	rd	op	
lw t3, -36(s4)	-36	20	2	28	3	1111 1101 1100	10100	010	11100	000 0011	(0xFDCA2E03)
lw x28, -36(x20)											
	12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Figure 6.18 Machine code for the I-type instruction of Example 6.4

Remember that an M -bit two's complement number is sign-extended to an N -bit number ($N > M$) by copying the sign bit (most significant bit) of the M -bit number into all of the upper bits of the N -bit number. Sign-extending a two's complement number does not change its value. For example, 1101_2 is a 4-bit two's complement number representing -3_{10} . When sign-extended to 8 bits, it becomes 11111101_2 and still represents -3_{10} .

immediates, the upper bits should be all 1's. Recall from [Section 1.4.6](#) that this is called *sign extension*.

6.4.3 S/B-Type Instructions

Like I-type instructions, S/B-type (*store/branch*) instructions use two register operands and one immediate operand. However, both of the register operands are source registers (**rs1** and **rs2**) in S/B-type, whereas I-type instructions use one source register (**rs1**) and one destination register (**rd**). [Figure 6.19](#) shows the S/B-type machine instruction format. The instruction replaces the **funct7** and **rd** fields of R-type instructions with the 12-bit immediate **imm**. Thus, this immediate field is split across two bit ranges, bits 31:25 and bits 11:7, of the instruction.

Store instructions use S-type and branch instructions use B-type. S- and B-type formats differ only in how the immediate is encoded. S-type instructions encode a 12-bit signed (two's complement) immediate, with the top seven bits (**imm**_{11:5}) in bits 31:25 of the instruction and the lower five bits (**imm**_{4:0}) in bits 11:7 of the instruction.

B-type instructions encode a 13-bit signed immediate representing the *branch offset*, but only 12 of the bits are encoded in the instruction. The least significant bit is always 0, because branch amounts are always an even number of bytes, as will be explained later. The immediate of the B-type instruction is a somewhat strange bit swizzling of the immediate. **imm**₁₂ is in **instr**₃₁; **imm**₁₁ is in **instr**₇; **imm**_{10:5} is in **instr**_{30:25}; **imm**_{4:1} is in **instr**_{11:8}; and **imm**₀ is always 0 and, hence, isn't part of the instruction. This bit mashup is done so that immediate bits occupy the same instruction bit across instruction formats as much as possible and so that the sign bit is always in **instr**₃₁, as will be described in [Section 6.4.5](#).

[Figure 6.20](#) shows several examples of encoding store instructions using the S-type format. **rs1** is the base address, **imm** is the offset, and **rs2** is the value to be stored to memory. Recall that negative immediate values are represented using 12-bit two's complement notation. For example, in `sw x7, -6(x19)`, register **x19** is the base address (**rs1**), **x7** is the second source (**rs2**), the value to be stored to memory, and **-6** is the offset. For all S-type instructions, **op** is 35 (0100011₂) and **funct3** distinguishes between **sb** (0), **sh** (1), and **sw** (2).

In the `sw` assembly instruction, **rs2** is the leftmost register, that is: `sw rs2, offset(rs1)`

Figure 6.19 S- and B-type instruction formats

31:25	24:20	19:15	14:12	11:7	6:0	
imm _{11:5}	rs2	rs1	funct3	imm _{4:0}	op	S-Type
imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op	B-Type
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Assembly	Field Values						Machine Code					
	imm _{11:5}	rs2	rs1	funct3	imm _{4:0}	op	imm _{11:5}	rs2	rs1	funct3	imm _{4:0}	op
sw t2, -6(s3)	1111 111	7	19	2	11010	35	1111 111	00111	10011	010	11010	010 0011
sw x7, -6(x19)	0000 000	20	5	1	10111	35	0000 000	10100	00101	001	10111	010 0011
sh s4, 23(t0)	0000 001	30	0	0	01101	35	0000 001	11110	00000	000	01101	010 0011
sh x20, 23(x5)												
sb t5, 0x2D(zero)												
sb x30, 0x2D(x0)												
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Figure 6.20 Machine code for S-type instructions

#Address	# RISC-V Assembly
0x70	beq s0, t5, L1
0x74	add s1, s2, s3
0x78	sub s5, s6, s7
0x7C	lw t0, 0(s1)
0x80	L1: addi s1, s1, -15

L1 is 4 instructions (i.e., 16 bytes) past beq

imm _{12:0} = 16	0	0	0	0	0	0	0	0	1	0	0	0	0
bit number	12	11	10	9	8	7	6	5	4	3	2	1	0

Assembly	Field Values						Machine Code					
	imm _{12:0}	rs2	rs1	funct3	imm _{4:1,11}	op	imm _{12:0}	rs2	rs1	funct3	imm _{4:1,11}	op
beq s0, t5, L1	0000 000	30	8	0	1000 0	99	0000 000	11110	01000	000	1000 0	110 0011
beq x8, x30, 16												
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Figure 6.21 B-type instruction format and calculations for beq

Branches (beq, bne, blt, bge, bltu, and bgeu) use the B-type instruction format. Figure 6.21 shows some example code with the branch if equal instruction, beq. Instruction addresses are given to the left of each instruction. The *branch target address (BTA)* is the destination of the branch. The beq instruction in Figure 6.21 has a BTA of 0x80, the instruction address of the L1 label. The *branch offset* is sign-extended and added to the address of the branch instruction to obtain the branch target address.

For B-type instructions, rs1 and rs2 are the two source registers, and the 13-bit immediate branch offset, imm_{12:0}, gives the *number of bytes* between the branch instruction and the BTA. In this case, the BTA is four instructions after the beq instruction, that is, $4 \times 4 = 16$ bytes past beq. Thus, the branch offset is 16. Only bits 12:1 are encoded in the instruction because bit 0 of the branch offset is always 0.

For 32-bit instructions, bits 1:0 of the 13-bit branch offset (imm_{12:0}) are always zero because 32-bit instructions occupy 4 bytes of memory. Thus, instruction addresses are always divisible by four and neither bits 1 nor 0 of the branch offset need to be encoded in the instruction. However, RV32I only omits bit 0. This enables compatibility with 16-bit (2-byte) RISC-V compressed instructions (see Section 6.6.5). Compilers can then mix 16-bit and 32-bit instructions if the processor hardware supports both instruction sizes.

Example 6.5 ENCODING B-TYPE ASSEMBLY INSTRUCTIONS INTO MACHINE CODE

Consider the following RISC-V assembly code snippet. The instruction address is written to the left of each instruction. Translate the branch if not equal (bne) instruction into machine code.

Address	Instruction
0x354	L1: addi s1, s1, 1
0x358	sub t0, t1, s7
...	...
0xEB0	bne s8, s9, L1

Solution According to Table 6.1, s8 and s9 are registers 24 and 25. So, rs1 is 24 and rs2 is 25. The label L1 is $0xEB0 - 0x354 = 0xB5C$ (2908) bytes *before* the bne instruction. So, the 13-bit immediate is -2908 (1010010100100_2). From Appendix B, the op is 99 (1100011_2) and funct3 is 1 (001_2). Thus, the machine code is given in Figure 6.22. Notice that branch instructions can branch forward (to higher addresses) or, as in this case, backward (to lower addresses).

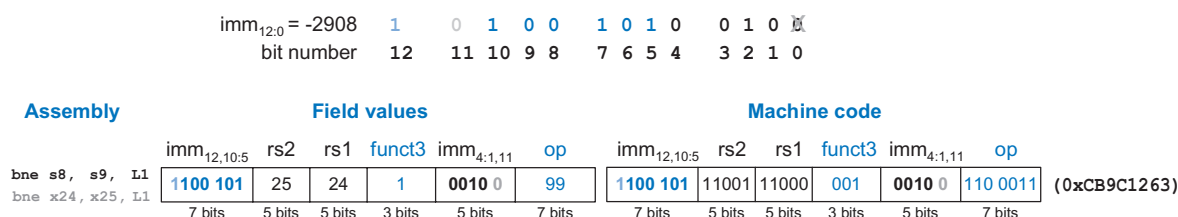


Figure 6.22 Machine code for the B-type instruction of Example 6.5

6.4.4 U/J-Type Instructions

U/J-type (*upper immediate/jump*) instructions have one destination register operand rd and a 20-bit immediate field, as shown in Figure 6.23. Like other formats, U/J-type instructions have a 7-bit opcode. In U-type instructions, the remaining bits specify the most significant 20 bits of a 32-bit immediate. In J-type instructions, the remaining 20 bits specify the most significant 20 bits of a 21-bit immediate jump offset. As with B-type instructions, the least significant bit of the immediate is always 0 and is not encoded in the J-type instruction.

As with B-type instructions, the J-type immediate bits are oddly scrambled. Computers don't care, but this is annoying to humans.

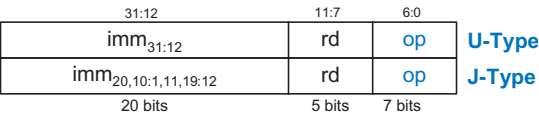


Figure 6.23 U- and J-type instruction formats

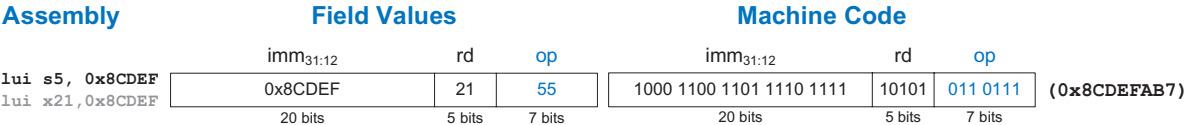


Figure 6.24 Machine code for U-type instruction lui

Figure 6.24 shows the load upper immediate instruction, `lui`, translated into machine code. The 32-bit immediate consists of the upper 20 bits encoded in the instruction and 0’s in the lower bits. So, in this case, after the instruction executes, register `s5` (`rd`) holds the value `0x8CDEF000`.

Figure 6.25 shows some example code using the jump and link instruction, `jal`. The instruction address is written to the left of each instruction. Like branch instructions, J-type instructions jump to an instruction address that is relative to the current PC, that is, the instruction address of the `jal` instruction. In Figure 6.25, the jump target address (JTA) is `0xABC04`, which is `0xA67F8` bytes past the `jal` instruction at address `0x540C` because `0xABC04 – 0x540C = 0xA67F8` bytes. Like branch instructions, the least significant bit is not encoded in

#	Address	RISC-V Assembly
0x0000540C		jal ra, func1
0x00005410		add s1, s2, s3
...		...
0x000ABC04	func1:	add s4, s5, s8
...		...

func1 is 0xA67F8 bytes past jal

imm = 0xA67F8	0	1	0	1	0	0	1	1	0	0	1	1	1	1	1	1	1	0	0	0	
bit number	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

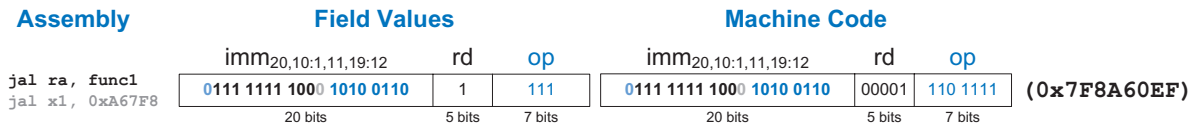


Figure 6.25 Machine code for J-type instruction jal

`jalr` is an I-type (*not* J-type!) instruction. `jal` is the only J-type instruction.

the instruction because it is always 0. The remaining bits are swizzled into the 20-bit immediate field, as shown in Figure 6.25. If a destination register, `rd`, is not specified by a `jal` assembly instruction, that field defaults to `ra` (`x1`). For example, the instruction `jal L1` is equivalent to `jal ra, L1` and has `rd = 1`. Ordinary jump (`j`) is encoded as `jal` with `rd = 0`.

6.4.5 Immediate Encodings

RISC-V uses 32-bit signed immediates. Only 12 to 21 bits of the immediate are encoded in the instruction. Figure 6.26 shows how immediates are formed for each instruction type. I- and S-type instructions encode 12-bit signed immediates. J- and B-type instructions use 21- and 13-bit signed immediates, where the least significant bit is always 0 (see Sections 6.4.3 and 6.4.4). U-type instructions encode the top 20 bits of a 32-bit immediate.

Figure 6.26 RISC-V immediates

imm ₁₁												imm _{11:1}				imm ₀	I, S B U J														
imm ₁₂												imm _{11:1}				0															
imm _{31:21}						imm _{20:12}						0																			
imm ₂₀						imm _{20:12}						imm _{11:1}				0															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Across instruction formats, RV32I attempts to keep immediate bits in the same instruction bits with the aim of simplifying hardware design (and at the cost of complicating instruction encodings). Figure 6.27 highlights this consistency by showing instruction fields for all formats. (The opcode is bits 6:0 for all instructions and is not shown.) `instr31` always holds the sign bit of the immediate. `instr30:20` holds `imm30:20` for U-type instructions. Otherwise, `instr30:25` holds `imm10:5` `instr19:12` holds `imm19:12` for U/J-type instructions. `imm4:1` occupies either `instr24:21` or `instr11:8`. Immediate bit 11 (when it is not the sign bit) and bit 0 are roving bits that are in bit 0 or 20 of the instruction.

Figure 6.27 RISC-V immediate encodings in machine instructions

11	10	9	8	7	6	5	4	3	2	1	0	rs1				funct3				rd				I	
11	10	9	8	7	6	5	rs2					rs1				funct3				4	3	2	1	0	S
12	10	9	8	7	6	5	rs2					rs1				funct3				4	3	2	1	11	B
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	rd				U	
20	10	9	8	7	6	5	4	3	2	1	11	19	18	17	16	15	14	13	12	rd				J	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	

Keeping immediate bit locations consistent across instruction formats is another example of regularity simplifying the design—specifically, it minimizes the number of wires and multiplexers needed to extract and sign-extend the immediate. Exercises 6.47 and 6.48 explore the hardware implications of this design decision further.

6.4.6 Addressing Modes

An *addressing mode* defines how an instruction specifies its operands. This section summarizes the modes used for addressing instruction operands. RISC-V uses four main modes: *register*, *immediate*, *base*, and *PC-relative* addressing. Most other architectures provide similar addressing modes, so understanding these modes helps you learn other assembly languages. The first three modes (register, immediate, and base addressing) define modes of reading and writing operands. The last mode (PC-relative addressing) defines a mode of writing the program counter (PC).

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 an immediate, along with registers, as operands. Some I-type instructions, such as add immediate (`addi`) and `xori`, use immediate addressing with a 12-bit signed immediate. Shift instructions with an immediate shift amount (`slli`, `srl`, and `srai`) are I-type instructions that encode the 5-bit unsigned immediate shift amount in `imm4:0`. Load instructions (`lb`, `lh`, and `lw`) use the I-type instruction format but use base addressing, which is discussed next.

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 calculated by adding the base address in register `rs1` to the sign-extended 12-bit offset found in the immediate field. Loads are I-type instructions and stores are S-type instructions.

PC-Relative Addressing

Branch and jump and link (`jal`) instructions use *PC-relative addressing* to specify the new value of the PC. The signed offset encoded in the immediate field is added to the PC to obtain the target address, the new PC; hence, the target address is said to be relative to the current PC. Branches and `jal` use a 13- and 21-bit signed immediate, respectively,

The jump and link register (`jlr`) instruction uses base addressing, not PC-relative addressing. It can jump to any instruction address in the 32-bit address space because its target address is formed by adding `rs1` to the 12-bit signed immediate. The return address, `PC+4`, is written to `rd`.

The sequence of instructions below allows this program to jump to any address. Instruction addresses are listed to the left of each instruction. In this case, the program jumps to address `0x12345678` and writes `0x0100FE7C` (i.e., `PC+4`) to `t1`.

# Address	RISC-V assembly
0x0100FE74	<code>lui s1, 0x12345</code>
0x0100FE78	<code>jlr t1, s1, 0x678</code>
...	...
0x12345678	...

for the offset. The most significant bits of the offset are encoded in the 12- and 20-bit immediate fields of the B- and J-type instructions. The offset's least significant bit is always 0, so it is not encoded in the instruction. The `auipc` (add upper immediate to PC) instruction also uses PC-relative addressing. For example, the instruction `auipc s3, 0xABCDE` places `PC + 0xABCDE000` in `s3`.

6.4.7 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 share a 7-bit opcode field. Thus, the best place to begin is to look at the opcode to determine if it is an R-, I-, S/B-, or U/J-type instruction.

Example 6.6 TRANSLATING MACHINE LANGUAGE TO ASSEMBLY LANGUAGE

Translate the following machine language code into assembly language.

```
0x41FE83B3
0xFDA48293
```

Solution First, we represent each instruction in binary and look at the seven least significant bits to find the opcode for each instruction.

```
0100 0001 1111 1110 1000 0011 1011 0011 (0x41FE83B3)
1111 1101 1010 0100 1000 0010 1001 0011 (0xFDA48293)
```

The opcode determines how to interpret the rest of the bits. The first instruction's opcode is `01100112`; so, according to Table B.1 in Appendix B, it is an R-type instruction and we can divide the rest of the bits into the R-type fields, as shown at the top of Figure 6.28. The second instruction's opcode is `00100112`, which means it is an I-type instruction. We group the remaining bits into the I-type format, as seen in Figure 6.28, which shows the assembly code equivalent of the two machine instructions.

	Machine Code						Field Values						Assembly
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op	
(0x41FE83B3)	0100 000	11111	11101	000	00111	011 0011	32	31	29	0	7	51	<code>sub x7, x29, x31</code> <code>sub t2, t4, t6</code>
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
	imm11:0	rs1	funct3	rd	op		imm11:0	rs1	funct3	rd	op		
(0xFDA48293)	1111 1101 1010	01001	000	00101	001 0011		-38	9	0	5	19		<code>addi x5, x9, -38</code> <code>addi t0, s1, -38</code>
	12 bits	5 bits	3 bits	5 bits	7 bits		12 bits	5 bits	3 bits	5 bits	7 bits		

Figure 6.28 Machine code to assembly code translation

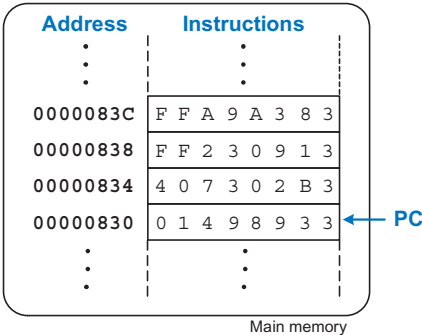
6.4.8 The Power of the Stored Program


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 a new program to memory. In contrast to dedicated hardware, the stored program offers general-purpose computing. In this way, a computer can execute applications ranging from a calculator to a word processor to a video player simply by changing the stored program.

Instructions in a stored program are retrieved, or *fetch*ed, from memory and executed by the processor. Even large, complex programs are simply a series of memory accesses and instruction executions. Figure 6.29 shows how machine instructions are stored in memory. In RISC-V programs, the instructions are normally stored starting at low addresses, but this may differ for each implementation. Figure 6.29 shows the code stored between addresses 0x00000830 and 0x0000083C. Remember that RISC-V memory is byte-addressable, so instruction addresses advance by 4, not 1.

To run or execute the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in the 32-bit program counter (PC) register.

Assembly code	Machine code
add s2, s3, s4	0x01498933
sub t0, t1, t2	0x407302B3
addi s2, t1, -14	0xFF230913
lw t2, -6(s3)	0xFFA9A383





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

Figure 6.29 Stored program

To execute the code in Figure 6.29, the PC is initialized to address 0x00000830. The processor fetches the instruction at that memory address and executes the instruction, 0x01498933 (`add s2, s3, s4`). The processor then increments the PC by 4 to 0x00000834, fetches and executes that instruction, and repeats.

The *architectural state* of a microprocessor holds everything necessary to determine what a program will do. For RISC-V, the architectural state includes the memory, register file, and PC. If the operating system (OS) saves the architectural state at some point in the program, it can interrupt the program, do something else, and then restore the state such that the program continues properly, unaware that it was ever interrupted. The architectural state is also of great importance when we build a microprocessor in Chapter 7.

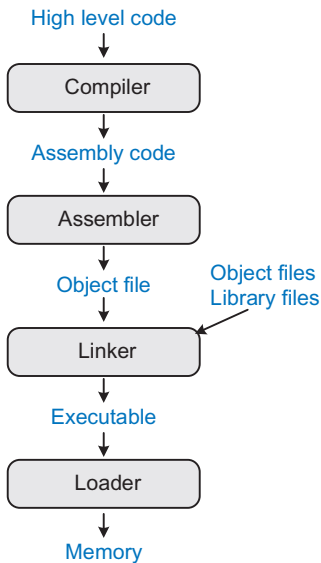


Figure 6.30 Steps for translating and starting a program

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

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

Figure 6.30 shows the steps required to translate a program from a high-level language into machine language and to start executing that program. First, a *compiler* translates the high-level code into assembly code. The *assembler* translates the assembly code into machine code and puts it in an object file. The *linker* combines the machine code with code from libraries and other files and determines the proper branch addresses and variable locations to produce an entire executable program. In practice, most compilers perform all three steps of compiling, assembling, and linking. Finally, the *loader* loads the program into memory and starts execution. The remainder of this section walks through these steps for a simple program.

6.5.1 The Memory Map

With 32-bit addresses, the RISC-V address space spans 2^{32} bytes (4 GB). Word addresses are multiples of 4 and range from 0 to 0xFFFFFFF. Figure 6.31 shows an example memory map. Our memory map divides the address space into five parts or *segments*: the text, global data, and dynamic data segments, and segments for exception handlers and the operating system (OS), which includes memory dedicated to input/output (I/O). The following sections describe each segment. We present an

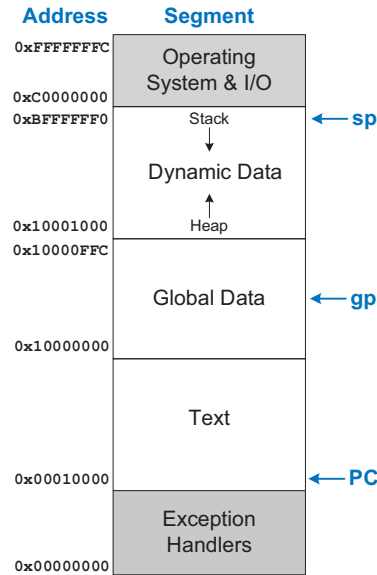


Figure 6.31 Example RISC-V memory map

example RISC-V memory map here; however, RISC-V does not define a specific memory map. While the exception handler is typically located at either low or high addresses, the user can define where the text (code and constant data), memory-mapped I/O, stack, and global data are placed. This allows for flexibility, especially with smaller systems, such as handheld devices, where only part of the memory range is used and, thus, populated with physical memory.

The Text Segment

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

The Global Data Segment

The *global data segment* stores global variables that, in contrast to local variables, can be accessed by all functions in a program. Local variables are defined within a function and can only be accessed by that function; they are typically located in registers or on the stack. Global variables are allocated in memory before the program begins executing, and they are typically accessed using the *global pointer* register *gp* (register *x3*) that points to the middle of the global data segment. In this case, *gp* is 0x10000800. Using the 12-bit signed offset, programmers can use *gp* to access the entire global data segment.

RISC-V requires that `sp` maintain 16-byte alignment to enable compatibility with the quad-precision RISC-V base instruction set, RV128I, which operates on 128-bit (i.e., 16-byte) data. So, `sp` decrements by multiples of 16 to make room on the stack, even if smaller amounts of stack space are needed. We glossed over this requirement in [Section 6.3.7](#) to highlight functionality above convention.

The Dynamic Data Segment

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

Upon start-up, the operating system sets up the stack pointer (`sp`, register `x2`) to point to the top of the stack, in this case `0xBFFFFFF0`. The stack typically grows downward, as shown here. The stack includes temporary storage and local variables, such as arrays, that do not fit in the registers. As discussed in [Section 6.3.7](#), functions also use the stack to save and restore registers. Each stack frame is accessed in last-in-first-out order.

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

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

The Exception Handler, OS, and I/O Segments

The lowest part of the example RISC-V memory map is reserved for the exception handlers (see [Section 6.6.2](#)) and boot code that is run at start-up. The highest part of the memory map is reserved for the operating system and memory-mapped I/O (see [Section 9.2](#)).

6.5.2 Assembler Directives

Assembler directives guide the assembler in allocating and initializing global variables, defining constants, and differentiating between code and data. [Table 6.5](#) lists common RISC-V assembler directives, and [Code Example 6.29](#) shows how to use them.

The `.data`, `.text`, `.bss`, and `.section .rodata` assembler directives tell the assembler to place the proceeding data or code in the global data, text (code), BSS, or read-only data (`.rodata`) segments of memory, respectively. The BSS segment is located in the global data segment but is initialized to zero. The read-only data segment is constant data that is placed in the text segment (i.e., in the *program memory*).

The program in [Code Example 6.29](#) begins by making the `main` label global (`.globl main`) so that the `main` function can be called from outside this file, typically by the OS or bootloader. The value `N` is then set to 5 (`.equ N, 5`). The assembler replaces `N` with 5 before translating

BSS stands for *block started symbol* and it was initially a keyword to allocate a block of uninitialized data. Now, most operating systems initialize data in the BSS segment to zero.

Table 6.5 RISC-V assembler directives

Assembler Directive	Description
<code>.text</code>	Text section
<code>.data</code>	Global data section
<code>.bss</code>	Global data initialized to 0
<code>.section .foo</code>	Section named <code>.foo</code>
<code>.align N</code>	Align next data/instruction on 2^N -byte boundary
<code>.balign N</code>	Align next data/instruction on N -byte boundary
<code>.globl sym</code>	Label <code>sym</code> is global
<code>.string "str"</code>	Store string <code>"str"</code> in memory
<code>.word w1, w2, ..., wN</code>	Store N 32-bit values in successive memory words
<code>.byte b1, b2, ..., bN</code>	Store N 8-bit values in successive memory bytes
<code>.space N</code>	Reserve N bytes to store variable
<code>.equ name, constant</code>	Define symbol <code>name</code> with value <code>constant</code>
<code>.end</code>	End of assembly code

assembly instructions into machine code. For example, the instruction `lw t5, N*4(t0)` is translated into `lw t5, 20(t0)` and then converted into machine code (0x0142AF03). Next, the program allocates the following global variables, as shown in Figure 6.32: `A` (a 7-element array of 32-byte values), `str1` (a null-terminated string), `B` and `C` (4 bytes each), and `D` (1 byte). `A`, `B`, and `str1` are initialized, respectively, to {5, 42, -88, 2, -5033, 720, 314}, 0x32A, and "RISC-V" (i.e., {52, 49, 53, 43, 2D, 56, 00} – see Table 6.2). Remember that, in the C programming language, strings are terminated with the null character (0x00). The variables `C` and `D` are uninitialized by the user and are located in the BSS segment. This assembler includes 16 bytes of unallocated memory between the data and BSS segments, as indicated by the gray boxes in Figure 6.32.

The `.align 2` assembler directive aligns the proceeding data or code on a $2^2 = 4$ -byte boundary. The `.balign 4` (byte align 4) assembler directive is equivalent. These assembler directives help maintain alignment for data and instructions. For example, if the `.align 2` were removed before `B` is allocated (i.e., before `B: .word 0x32A`), `B` would have been allocated directly after the `str1` variable, in bytes 0x2157 – 0x215A (instead of 0x2158 – 0x215B).

The program from Code Example 6.29 was run on Western Digital's open-source commercial SweRV EH1 RISC-V core. Other processors use different memory maps, so would place variables and code at different addresses. The free RVfpga (RISC-V FPGA) course from Imagination Technologies shows how to use the SweRV EH1 core targeted to an FPGA to run C and assembly programs and to explore, expand, and modify that RISC-V processor and system. See <https://university.imgtec.com/rvfpga/>.

Notice that `str2` is located in the code segment (*not* the data segment) at address 0x140, near the user code (`main`) which starts at address 0x88. By placing code and data together, the program minimizes the needed memory and the number of instructions to access data, which are both critical in handheld and embedded systems.

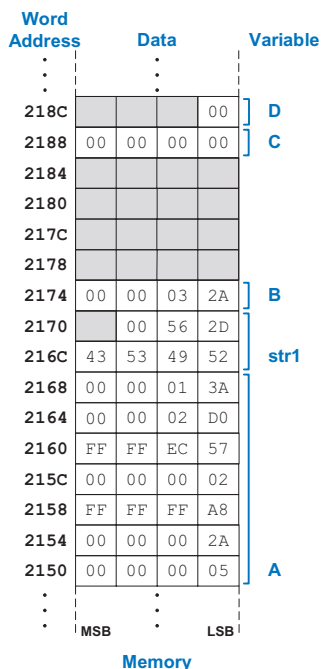


Figure 6.32 Memory allocation of global variables in Code Example 6.29

Code Example 6.29 USING ASSEMBLER DIRECTIVES

```
.globl main          # make the main label global
.equ N, 5            # N = 5

.data                # global data segment
A: .word 5, 42, -88, 2, -5033, 720, 314
str1: .string "RISC-V"
.align 2             # align next data on 2^2-byte boundary
B: .word 0x32A

.bss                 # bss segment - variables initialized to 0
C: .space 4
D: .space 1

.balign 4            # align next instruction on 4-byte boundary
.text                # text segment (code)
main:
    la t0, A          # t0 = address of A                = 0x2150
    la t1, str1        # t1 = address of str1            = 0x216C
    la t2, B           # t2 = address of B                = 0x2174
    la t3, C           # t3 = address of C                = 0x2188
    la t4, D           # t4 = address of D                = 0x218C
    lw t5, N*4(t0)     # t5 = A[N] = A[5] = 720          = 0x2D0
    lw t6, 0(t2)       # t6 = B = 810                    = 0x32A
    add t5, t5, t6      # t5 = A[N] + C = 720 + 810 = 1530 = 0x5FA
    sw t5, 0(t3)       # C = 1530                        = 0x5FA
    lb t5, N-1(t1)     # t5 = str1[N-1] = str1[4] = '-' = 0x2D
    sb t5, 0(t4)       # D = str1[N-1]                  = 0x2D
    la t5, str2        # t5 = address of str2            = 0x140
    lb t6, 8(t5)       # t6 = str2[8] = 'r'              = 0x72
    sb t6, 0(t1)       # str1[0] = 'r'                  = 0x72
    jr ra              # return

.section .rodata
str2: .string "Hello world!"
.end                  # end of assembly file
```

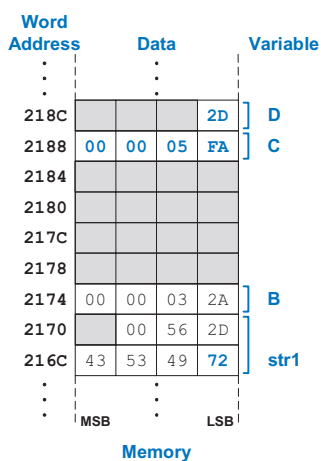


Figure 6.33 Final values of global variables C, D, and str1

The main function begins by loading the addresses of the global variables into t0–t4 using the load address (la) pseudoinstruction (see Table B.7, located on the inside covers of the textbook). The program retrieves A[5] and C from memory, adds them together, and places the result (0x5FA) in D. Then it loads the value of str1[4] (which is ‘-’ = ASCII code 0x2D) using instruction lb t5, N-1(t1) and places that value in global variable B. At the end, the program reads str2[8], which is the character ‘r’, and places that value in str1[0]. The main function finishes by returning to the operating system or boot code using jr ra. Figure 6.33 shows the final values of C, D, and str1. The .end assembly directive indicates the end of the assembly file.

6.5.3 Compiling

A compiler translates high-level code into assembly language, and an assembler then translates that assembly code into machine code. The examples in this section are based on GCC, a popular and widely used

free compiler. GCC is part of a *toolchain* that includes other capabilities, some of which will be discussed in this section. [Code Example 6.30](#) shows a simple high-level program with three global variables and two functions, along with the assembly code produced by GCC from SiFive's Freedom E SDK toolchain. See the Preface for instructions about using RISC-V compilers.

In [Code Example 6.30](#), the main function starts by storing *ra* on the stack. It makes room for four words (16 bytes) but only uses one of the stack locations. Recall that *sp* must maintain 16-byte alignment for

Code Example 6.30 COMPILING A HIGH-LEVEL PROGRAM

High-Level Code

```
int f, g, y;

int func(int a, int b) {
    if (b < 0)
        return (a + b);
    else
        return(a + func(a, b - 1));
}

void main() {
    f=2;
    g=3;
    y=func(f,g);

    return;
}
```

RISC-V Assembly Code

```
.text
.globl func
.type func,@function

func:
    addi sp,sp,-16
    sw   ra,12(sp)
    sw   s0,8(sp)
    mv   s0,a0
    add  a0,a1,a0
    bge  a1,zero,.L5

.L1:
    lw   ra,12(sp)
    lw   s0,8(sp)
    addi sp,sp,16
    jr   ra

.L5:
    addi a1,a1,-1
    mv   a0,s0
    call func
    add  a0,a0,s0
    j    .L1

.globl main
.type main,@function

main:
    addi sp,sp,-16
    sw   ra,12(sp)
    lui  a5,%hi(f)
    li   a4,2
    sw   a4,%lo(f)(a5)
    lui  a5,%hi(g)
    li   a4,3
    sw   a4,%lo(g)(a5)
    li   a1,3
    li   a0,2
    call func
    lui  a5,%hi(y)
    sw   a0,%lo(y)(a5)
    lw   ra,12(sp)
    addi sp,sp,16
    jr   ra
    .comm y,4,4
    .comm g,4,4
    .comm f,4,4
```



Grace Hopper, 1906–1902

Graduated from Yale University with a Ph.D. in mathematics. Developed the first compiler while working for the Remington Rand Corporation and was instrumental in developing the COBOL programming language. As a naval officer, she received many awards, including a World War II Victory Medal and the National Defense Service Medal. She has also documented the first computer “bug,” which, in this case was an actual insect that interfered with a punchcard.

compatibility with RV128I. `main` then writes the value 2 to global variable `f` and 3 to global variable `g`. The global variables are not yet placed in memory—they will be later, by the assembler. So, for now, the assembly code uses two instructions (`lui` followed by `sw`) instead of just one (`sw`) to store to each global variable in case it needs to specify a 32-bit address.

The program then puts `f` and `g` (i.e., 2 and 3) into the argument registers, `a0` and `a1`, and calls `func` by using the pseudocode `call func`. The function (`func`) stores `ra` and `s0` on the stack. It then places `a0` (`a`) in `s0` (because it will be needed after the recursive call to `func`) and calculates `a0 = a0 + a1` (the return value = `a + b`). `func` then branches if `a1` (`b`) is greater than or equal to zero. Otherwise, it restores `ra`, `s0`, and `sp` and returns using `jr ra`. If the branch is taken ($b \geq 0$), `func` decrements `a1` (`b`), and recursively calls `func`. After it returns from the recursive call, it adds the return value (`a0`) and `s0` (`a`) and jumps to label `.L1`, where `ra`, `s0`, and `sp` are restored and the function returns. The `main` function then stores the returned result from `func` (`a0`) into global variable `y`, restores `ra` and `sp`, and returns `y`. At the bottom of the assembly code, the program indicates that it has three 4-byte-wide global variables `f`, `g`, and `y`, using `.comm g, 4, 4`, etc. The first 4 indicates 4-byte alignment and the second 4 indicates the size of the variable (4 bytes).

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

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

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

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

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

The output, `prog.s`, is rather verbose, but the interesting parts are shown in [Code Example 6.30](#).

6.5.4 Assembling

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

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

or

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

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

We can *disassemble* the object file using the `objdump` command to see the assembly language code beside the machine language code.

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

The following shows the disassembly of the `.text` section. If the code was originally compiled with `-g`, the disassembler also shows the corresponding lines of C code, as shown below, interspersed with the assembly code. Notice that the `call` pseudoinstruction was translated into two RISC-V instructions: `auipc ra,0x0` and `jalr ra` in case the function is far away, that is, farther away from the current PC than `jal`'s signed 21-bit offset can reach. The instructions for storing to the global variables are also just placeholders until the global variables are placed in memory. For example, the three instructions at addresses `0x48` to `0x50` are for storing the value 2 in global variable `f`. Once `f` is placed in memory in the linking stage, the instructions will be updated.

```
00000000 <func>:
int f, g, y;
int func(int a, int b) {
    0: ff010113          addi    sp,sp,-16
    4: 00112623          sw      ra,12(sp)
    8: 00812423          sw      s0,8(sp)
    c: 00050413          mv      s0,a0
    if (b<0) return (a+b);
   10: 00a58533          add     a0,a1,a0
   14: 0005da63          bgez   a1,28 <.L5>
00000018 <.L1>:
    else return(a + func(a, b-1));
}
   18: 00c12083          lw      ra,12(sp)
   1c: 00812403          lw      s0,8(sp)
   20: 01010113          addi    sp,sp,16
   24: 00008067          ret
00000028 <.L5>:
    else return(a + func(a, b-1));
   28: fff58593          addi    a1,a1,-1
   2c: 00040513          mv      a0,s0
```

```

30: 00000097          auipc ra,0x0
34: 000080e7          jalr  ra # 30 <.LVL5+0x4>
38: 00850533          add   a0,a0,s0
3c: fddff06f          j     18 <.L1>

00000040 <main>:
void main() {
  40: ff010113          addi  sp,sp,-16
  44: 00112623          sw    ra,12(sp)
  f=2;
  48: 000007b7          lui   a5,0x0
  4c: 00200713          li    a4,2
  50: 00e7a023          sw    a4,0(a5) # 0 <func>
  g=3;
  54: 000007b7          lui   a5,0x0
  58: 00300713          li    a4,3
  5c: 00e7a023          sw    a4,0(a5) # 0 <func>
  y=func(f,g);
  60: 00300593          li    a1,3
  64: 00200513          li    a0,2
  68: 00000097          auipc ra,0x0
  6c: 000080e7          jalr  ra # 68 <main+0x28>
  70: 000007b7          lui   a5,0x0
  74: 00a7a023          sw    a0,0(a5) # 0 <func>
  return;
}
  78: 00c12083          lw     ra,12(sp)
  7c: 01010113          addi  sp,sp,16
  80: 00008067          ret

```

We can view the symbol table from the object file using `objdump` with the `-t` flag. The interesting parts are shown below. We added labels for the three columns of interest: the symbol's memory address, size, and name. Because the program has not yet been placed in memory (it has not been linked), the addresses are only placeholders for now. The `.text` indicates the code (text) segment and `.data` the data (global data) segment. The size of those two symbols is currently 0 because the program has not yet been linked. The size of the two functions, `func` and `main`, are listed: `func` is 0x40 (64) bytes = 16 instructions, and `main` is 0x44 (68) bytes = 17 instructions, as shown in the code above. The global variable symbols `f`, `g`, and `y` are listed and are 4 bytes each, but their addresses are listed as a placeholder value, 0x00000004, because they have not yet been assigned addresses.

```
objdump -t prog.o
```

SYMBOL TABLE:

Address		Size	Symbol Name
00000000	l d .text	00000000	.text
00000000	l d .data	00000000	.data
00000000	g F .text	00000040	func
00000040	g F .text	00000044	main
00000004	0 *COM*	00000004	f
00000004	0 *COM*	00000004	g
00000004	0 *COM*	00000004	y

We will largely ignore the unlabeled columns in this symbol table. They show the flags associated with the symbols (l for local or g for global, and d for debug, F for function, or 0 for object) and the section in which the symbol is located (.text, .data, or *COM* (common) when it is not located in a section).

6.5.5 Linking

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

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

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

We can again disassemble the executable using:

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

The start-up code is too lengthy to show, but the updated symbol table and program code disassembled from the executable are shown below. We have again added labels to the columns of interest. The functions and global variables are now relocated to actual addresses. According to the symbol table, the overall text and data segments (which include start-up code and system data) begin at 0x10074 and 0x115e0, respectively. `func` starts at address 0x10144 and is 0x3c bytes (15 instructions). `main` starts at 0x10180 and is 0x34 bytes (13 instructions). The global variables are each 4 bytes; `f` is located at memory address 0x11a30, `g` at 0x11a34, and `y` at 0x11a38.

Notice that now that the global variables, `f`, `g`, and `y`, are allocated memory addresses, they are listed as global symbols (as indicated by the `g` flag) and are located in the `.bss` segment, where uninitialized global variables are placed.

SYMBOL TABLE:

Address		Size	Symbol Name
00010074	l d .text	00000000	.text
000115e0	l d .data	00000000	.data
00010144	g F .text	0000003c	func
00010180	g F .text	00000034	main
00011a30	g 0 .bss	00000004	f
00011a34	g 0 .bss	00000004	g
00011a38	g 0 .bss	00000004	y

Notice that the size of `func`, as shown below, is now only 15 instructions instead of 16. The call to `func` was nearby; so, only one instruction, `jalr`, was needed to make the call. Likewise, the `main` code reduced from 17 to 13 instructions because of near calls and stores near to the global pointer, `gp`. The program stores to `f` using a single instruction: `sw a4, -944(gp)`. From this instruction, we can also determine the value of the global pointer, `gp`, that was initialized by the start-up code. We know that `f` is at address `0x11a30`; so, `gp` is `0x11a30 + 944 = 0x11DE0`.

```
00010144 <func>:
```

```
int f, g, y;
```

```
int func(int a, int b) {
    10144: ff010113          addi  sp,sp,-16
    10148: 00112623          sw    ra,12(sp)
    1014c: 00812423          sw    s0,8(sp)
    10150: 00050413          mv    s0,a0
    if (b<0) return (a+b);
    10154: 00a58533          add   a0,a1,a0
    10158: 0005da63          bgez  a1,1016c <func+0x28>
    else return(a + func(a, b-1));
}
    1015c: 00c12083          lw    ra,12(sp)
    10160: 00812403          lw    s0,8(sp)
    10164: 01010113          addi  sp,sp,16
    10168: 00008067          ret
    else return(a + func(a, b-1));
    1016c: fff58593          addi  a1,a1,-1
    10170: 00040513          mv    a0,s0
    10174: fd1ff0ef          jal   ra,10144 <func>
    10178: 00850533          add   a0,a0,s0
    1017c: fe1ff06f          j     1015c <func+0x18>
```

```
00010180 <main>:
```

```

void main() {
    10180: ff010113      addi  sp,sp,-16
    10184: 00112623      sw   ra,12(sp)
    f=2;
    10188: 00200713      li   a4,2
    1018c: c4e1a823      sw   a4,-944(gp) # 11a30 <f>
    g=3;
    10190: 00300713      li   a4,3
    10194: c4e1aa23      sw   a4,-940(gp) # 11a34 <g>
    y=func(f,g);
    10198: 00300593      li   a1,3
    1019c: 00200513      li   a0,2
    101a0: fa5ff0ef      jal  ra,10144 <func>
    101a4: c4a1ac23      sw   a0,-936(gp) # 11a38 <y>

    return;
}
    101a8: 00c12083      lw   ra,12(sp)
    101ac: 01010113      addi  sp,sp,16
    101b0: 00008067      ret

```

6.5.6 Loading

The operating system loads a program by reading the text segment of the executable file from a storage device (usually the hard disk or flash storage) into the text segment of memory. The operating system jumps to the beginning of the program to begin executing. Figure 6.34 shows the memory map at the beginning of program execution.

6.6 ODDS AND ENDS*

This section covers a few optional topics that do not fit naturally elsewhere in the chapter. These topics include endianness, exceptions, signed and unsigned arithmetic instructions, floating-point instructions, and compressed (16-bit) instructions.

6.6.1 Endianness

Byte-addressable memories are organized in a big-endian or little-endian fashion, as shown in Figure 6.35. In both formats, a 32-bit word's most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. Word addresses are the same in both formats and refer to the same four bytes. Only the addresses of bytes within a word differ (see Figure 6.35). In *big-endian* machines, bytes are numbered starting with 0 at the big (most significant) end. In *little-endian* machines, bytes are numbered starting with 0 at the little (least significant) end.

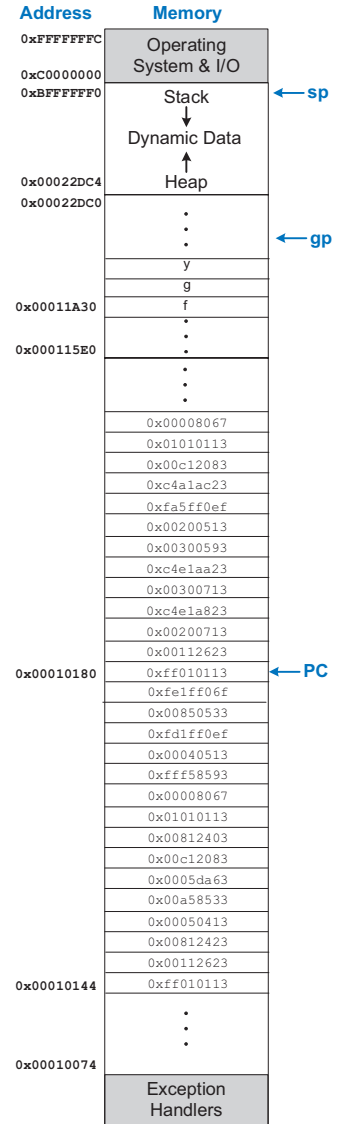
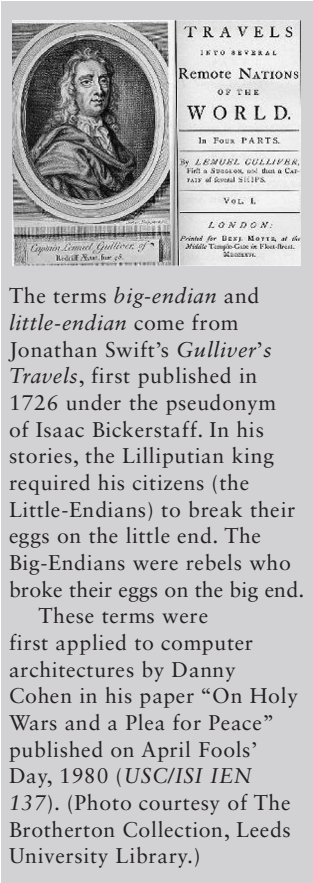


Figure 6.34 prog loaded into memory



A fourth privilege level exists called *hypervisor mode* (*H-mode*) that supports *machine virtualization*, that is, the appearance of multiple machines (potentially with multiple operating systems) running on a single physical machine. H-mode has higher privilege than S-mode but not as much privilege as M-mode.

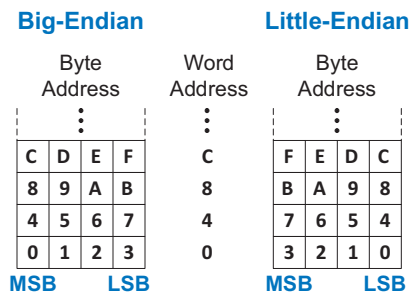


Figure 6.35 Big- and little-endian memory addressing

RISC-V is typically little-endian, although a big-endian variant has been defined. IBM's PowerPC (formerly found in Macintosh computers) uses big-endian addressing. Intel's x86 architecture (found in PCs) uses little-endian addressing. The choice of endianness is completely arbitrary but leads to hassles when sharing data between big- and little-endian computers. In examples in this text, we use little-endian format whenever byte ordering matters.

6.6.2 Exceptions

An *exception* is like an unscheduled function call caused by an event in hardware or software. For example, the processor may receive notification that the user pressed a key on a keyboard. The processor may stop what it is doing, determine which key was pressed, save it for future reference, and then resume the program that was running. Such a hardware exception triggered by an input/output (I/O) device such as a keyboard is often called an *interrupt*. Alternatively, the program may encounter an error condition caused by the software, such as an undefined instruction. Software exceptions are sometimes called *traps*. Other causes of exceptions include reset and attempts to read nonexistent memory. Like any other function call, an exception must save the return address, jump to some address, do its work, clean up after itself, and return to the program where it left off.

Execution Modes and Privilege Levels

A RISC-V processor can operate in one of several execution modes with different privilege levels. Privilege levels dictate what instructions can be executed and what memory can be accessed. The three main RISC-V privilege levels are *user mode*, *supervisor mode*, and *machine mode*, in order of increasing privilege. Machine mode (M-mode) is the highest privilege level; a program running in this mode can access all registers and memory locations. M-mode is the only required privilege mode and the only mode used

in processors without an operating system (OS), including many embedded systems. User applications that run on top of an OS typically run in user mode (U-mode) and the OS runs in supervisor mode (S-mode); user programs do not have access to privileged registers or memory locations reserved for the OS. The different modes keep the key state from being corrupted. We discuss exceptions when running in M-mode. Exceptions that occur at other levels are similar but use registers associated with that mode.

Exception Handlers

Exception handlers use four special-purpose registers, called *control and status registers* (CSRs), to handle an exception: `mtvec`, `mcause`, `mepc`, and `mscratch`. The machine trap-vector base-address register, `mtvec`, holds the address of the exception handler code. When an exception occurs, the processor records the cause of an exception in `mcause` (see Table 6.6), stores the PC of the excepting instruction in `mepc`, the machine exception PC register, and jumps to the exception handler at the address preconfigured in `mtvec`.

After jumping to the address in `mtvec`, the exception handler reads the `mcause` register to examine what caused the exception and responds appropriately (e.g., by reading the keyboard on a hardware interrupt).

Table 6.6 Common exception cause encodings

Interrupt	Exception Code	Description
1	3	Machine software interrupt
1	7	Machine timer interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-Mode
0	9	Environment call from S-Mode
0	11	Environment call from M-Mode

RISC-V defines a whole slew of CSRs, all of which must be initialized at start-up.

The value of `mcause` can be classified as either an interrupt or an exception, as indicated by the left-most column in Table 6.6, which is bit 31 of `mcause`. Bits [30:0] of `mcause` hold the *exception code*, that indicates the cause of the interrupt or exception.

Exceptions can use one of two exception handling modes: direct or vectored. RISC-V typically uses the direct mode described here, where all exceptions branch to same address, that is, the base address encoded in bits 31:2 of `mtvec`. In vectored mode, exceptions branch to an *offset* from the base address, depending on the cause of the exception. Each offset is separated by a small number of addresses—for example, 32 bytes—so the exception handler code may need to jump to a larger exception handler to deal with the exception. The exception mode is encoded in bits 1:0 of `mtvec`; `002` is for direct mode and `012` for vectored.

Exception-related registers are specific to the operating mode. M-mode registers are `mtvec`, `mepc`, `mcause`, and `mscratch`, and S-mode registers are `sepc`, `scause`, and `sscratch`. H-mode also has its own registers. Separate exception registers dedicated to each mode provide hardware support for multiple privilege levels.

`csrrw` is an actual RISC-V instruction (see Table B.8 in Appendix B), but `csrr` and `csrw` are pseudoinstructions. `csrr` is implemented as `csrrs rd, CSR, x0` and `csrw` as `csrrw x0, CSR, rs1`.

It then either aborts the program or returns to the program by executing the `mret`, machine exception return instruction, that jumps to the address in `mepc`. Holding the PC of the excepting instruction in `mepc` is analogous to using `ra` to store the return address during a `jal` instruction. Exception handlers must use program registers (`x1–x31`) to handle exceptions, so they use the memory pointed to by `mscratch` to store and restore these registers.

Exception-Related Instructions

Exception handlers use special instructions to deal with exceptions. These instructions are called *privileged instructions* because they access CSRs. They are part of the base RV32I instruction set (see Appendix B, Table B.8). The `mepc` and `mcause` registers are not part of the RISC-V program registers (`x1–x31`), so the exception handler must move these special-purpose (CSR) registers into the program registers to read and operate on them. RISC-V uses three instructions to read, write, or both read and write CSRs: `csrr` (read CSR), `csrw` (write CSR), and `csrrw` (read/write CSR). For example, `csrr t1, mcause` reads the value in `mcause` into `t1`; `csrw mepc, t2` writes the value in `t2` into `mepc`; and `csrrw t1, mscratch, t0` simultaneously reads the value in `mscratch` into `t1` and writes the value in `t0` into `mscratch`.

Exception Handling Summary

In summary, when a processor detects an exception, it:

1. Jumps to the exception handler address held in `mtvec`.
2. The exception handler saves registers on a small stack pointed to by `mscratch` and then uses `csrr` (read CSR) to look at the cause of the exception (encoded in `mcause`) and respond accordingly.
3. When the handler is finished, it optionally increments `mepc` by 4, restores registers from memory and either aborts the program or returns to the user code using the `mret` instruction, which jumps to the address held in `mepc`.

Example 6.7 EXCEPTION HANDLER CODE

Write an exception handler for dealing with the following two exceptions: illegal instruction (`mcause = 2`) and load address misaligned (`mcause = 4`). If an illegal instruction occurs, the program should simply continue executing after the illegal instruction. Upon a load address misaligned exception, the program should abort. If any other exception occurs, the program should attempt to re-execute the instruction.

Solution The exception handler begins by preserving program registers that will be overwritten. It then checks for each exception cause and (1) continues executing just past the excepting instruction (i.e., at `mepc + 4`) upon an illegal instruction exception, (2) aborts upon a misaligned load address, or (3) attempts to re-execute the excepting instruction (i.e., returns to `mepc`) upon any other exception. Before returning to the program, the handler restores any registers that were overwritten. To abort the program, the handler jumps to exit code located at the `exit` label (not shown). For programs running on top of an OS, the `j exit` instruction may be replaced by an environment call (`ecall`) with the return code stored in a program register such as `a0`.

```
# save registers that will be overwritten
csrrw t0, mscratch, t0    # swap t0 and mscratch
sw     t1, 0(t0)          # save t1 on mscratch stack
sw     t2, 4(t0)          # save t2 on mscratch stack

# check cause of exception
csrr   t1, mcause         # t1 = mcause
addi   t2, x0, 2          # t2 = 2 (illegal instruction exception code)

illegalinstr:
bne    t1, t2, checkother # branch if not an illegal instruction
csrr   t2, mepc           # t2 = exception PC
addi   t2, t2, 4          # increment exception PC by 4
csrrw  mepc, t2           # mepc = mepc + 4
j      done              # restore registers and return

checkother:
addi   t2, x0, 4          # t2 = 4 (misaligned load exception code)
bne    t1, t2, done       # branch if not a misaligned load
j      exit              # exit program

# restore registers and return from the exception
done:
lw     t1, 0(t0)          # restore t1 from mscratch stack
lw     t2, 4(t0)          # restore t2 from mscratch stack
csrrw  t0, mscratch, t0   # swap t0 and mscratch
mret                   # return to program (PC = mepc)
...
exit:
...
```

At start-up, the processor jumps to the *reset exception vector*, a hardwired memory address—for example, `0x200`—which is the starting address of the *boot loader* code, also called *boot code*. Although reset is not a typical exception that occurs during program execution, it is called such because reset is an exceptional state of the processor. The boot code configures the memory system, initializes the CSRs and stack pointer, and reads part of the OS from disk. Then, it begins a much longer boot process in the OS. The OS eventually will load a program, change to unprivileged user mode, and jump to the start of the program. In *bare metal* systems—that is, systems with no OS—user code (potentially with a lightweight boot code for setting up the stack pointer, etc.) is typically placed directly at the reset vector.

A particularly important exception is a *system call*, also called an *environment call*. Programs use these to call a function in the OS, which runs at a higher privilege level than user code. This exception is initiated by the user program executing the `ecall` instruction. Like a function call, the program may set up argument registers before making the system call.

Unlike other architectures, such as MIPS and ARM, RISC-V does not include instructions (or exceptions) for detecting overflow because it can be detected using a series of existing instructions. For example, the following code detects unsigned overflow when adding `t1` and `t2`.

```
add t0, t1, t2
bltu t0, t1, overflow
```

In other words, if the result (`t0`) is less than either of the operands (in this case, `t1`), overflow occurred.

The following code detects overflow when adding two signed numbers, `t1` and `t2`:

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

In equation form, overflow =
 $(t2 < 0) \ \&\ (t0 \geq t1) \mid$
 $(t2 \geq 0) \ \&\ (t0 < t1)$

In words, overflow occurs when one operand is negative (`t3 = 1`) and the result is not less than the other operand (`t4 = 0`), or when one operand is greater than or equal to 0 (`t3 = 0`), and the result is less than the other operand (`t4 = 1`).

6.6.3 Signed and Unsigned Instructions

Recall that a binary number may be signed or unsigned. As with most architectures, RISC-V uses two's complement representation for signed numbers. RISC-V has certain instructions that come in signed and unsigned flavors, including multiplication, division, set less than, branches, and partial word loads.

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, $0xFFFFFFFF \times 0xFFFFFFFF$ would equal $0xFFFFFFFFE00000001$ if the numbers were unsigned but $0x00000000000000001$ if the numbers were signed. (Notice that the lower 32 bits are the same for both signed and unsigned multiplication.) Therefore, multiplication and division come in both signed and unsigned flavors. `mulh` and `div` treat the operands as signed numbers. `mulhu` and `divu` treat the operands as unsigned numbers. `mulhsu` treats the first operand as signed and the second operand as unsigned. All multiply high instructions (`mulh`, `mulhu`, and `mulhsu`) put the most significant 32 bits in the destination register `rd`. The lower 32 bits of the result are the same for unsigned or signed multiplication, so `mul` puts the lower 32 bits of the multiplication result in `rd` for both unsigned and signed multiplication.

Set Less Than

Set less than instructions 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 12-bit immediate before treating it as an unsigned number. For example, `sltiu s0, s1, -1273` compares `s1` to `0xFFFFFB07`, treating the immediate as a large positive number.

Branches

The *branch if less than* and *branch if greater than or equal to* instructions also come in signed (`blt` and `bge`) and unsigned (`bltu` and `bgeu`) versions. The signed versions treat the two source operands as two's complement numbers and the unsigned versions treat the source operands as unsigned numbers.

Loads

As described in [Section 6.3.6](#), 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, signed and unsigned half-word loads (`lh` and `lhu`) load two bytes into the lower half and sign- or zero-extend the upper half of the word.

6.6.4 Floating-Point Instructions

The RISC-V architecture defines optional floating-point extensions called RVE, RVD, and RVQ for operating on single-, double-, and quad-precision floating-point numbers, respectively. RVE/D/Q define 32 floating-point registers, `f0` to `f31`, with a width of 32, 64, or 128 bits, respectively. When a processor implements multiple floating-point extensions, it uses the lower part of the floating-point register for lower-precision instructions. `f0` to `f31` are separate from the program (also called *integer*) registers, `x0` to `x31`. As with program registers, floating-point registers are reserved for certain purposes by convention, as given in [Table 6.7](#).

[Table B.3](#) in [Appendix B](#) lists all of the floating-point instructions. Computation and comparison instructions use the same mnemonics for all precisions, with `.s`, `.d`, or `.q` appended at the end to indicate precision. For example, `fadd.s`, `fadd.d`, and `fadd.q` perform single-, double-, and quad-precision addition, respectively. Other floating-point instructions include `fsub`, `fmul`, `fdiv`, `fsqrt`, `fmadd` (multiply-add), and `fmin`. Memory accesses use separate instructions for each precision. Loads are `flw`, `fld`, and `flq`, and stores are `fsw`, `fsd`, and `fsq`.

Floating-point instructions use R-, I-, and S-type formats, as well as a new format, the R4-type instruction format (see [Figure B.1](#) in [Appendix B](#)).

Table 6.7 RISC-V floating-point register set

Name	Register Number	Use
<code>ft0–7</code>	<code>f0–7</code>	Temporary variables
<code>fs0–1</code>	<code>f8–9</code>	Saved variables
<code>fa0–1</code>	<code>f10–11</code>	Function arguments/Return values
<code>fa2–7</code>	<code>f12–17</code>	Function arguments
<code>fs2–11</code>	<code>f18–27</code>	Saved variables
<code>ft8–11</code>	<code>f28–31</code>	Temporary variables

Code Example 6.31 USING A FOR LOOP TO ACCESS AN ARRAY OF FLOATS

High-Level Code	RISC-V Assembly Code
<pre>int i; float scores[200]; for (i = 0; i < 200; i = i + 1) scores[i] = scores[i] + 10;</pre>	<pre># s0 = scores base address, s1 = i addi s1, zero, 0 # i = 0 addi t2, zero, 200 # t2 = 200 addi t3, zero, 10 # t3 = 10 fcvt.s.w ft0, t3 # ft0 = 10.0 for: bge s1, t2, done # if i >= 200 then done slli t3, s1, 2 # t3 = i * 4 add t3, t3, s0 # address of scores[i] flw ft1, 0(t3) # ft1 = scores[i] fadd.s ft1, ft1, ft0 # ft1 = scores[i] + 10 fsw ft1, 0(t3) # scores[i] = t1 addi s1, s1, 1 # i = i + 1 j for # repeat done:</pre>

This format is needed for multiply-add instructions, which use four register operands. [Code Example 6.31](#) modifies [Code Example 6.21](#) to operate on an array of single-precision floating-point scores. The changes are in bold.

6.6.5 Compressed Instructions

RISC-V's compressed instruction extension (RVC) reduces the size of common integer and floating-point instructions to 16 bits by reducing the sizes of the control, immediate, and register fields and by taking advantage of redundant or implied registers. This reduced instruction size decreases cost, power, and required memory—all of which can be crucial for handheld and mobile applications. According to the *RISC-V Instruction Set Manual*, typically 50% to 60% of a program's instructions can be replaced with RVC instructions. 16-bit instructions still operate on the base data size (32, 64, or 128 bits), as determined by the base instruction set. Assembly programs may use a mix of compressed and 32-bit instructions if the processor can handle both.

Most RV32I instructions have a compressed counterpart starting with **c.**, as listed in [Table B.6](#) of [Appendix B](#). To reduce their size, most compressed instructions only specify two registers; the first source is also the destination. Most use 3-bit register codes to specify one of 8 registers from **x8** to **x15**. **x8** is encoded as **000₂**, **x9** as **001₂**, and so on. immediates are also shorter (6–11 bits), and fewer bits are available for opcodes. [Figure B.2](#), located on the inside back cover of this book, shows the compressed instruction formats.

Code Example 6.32 modifies Code Example 6.21 to use compressed instructions. Notice that the constant 200 is too large to fit in a compressed immediate, so `s0` is initialized with an uncompressed `addi`. There is no compressed `c.bge`, so we also use a noncompressed `bge`. We also increment `s0` as a pointer to `scores[i]` because shifting and adding is cumbersome with 2-operand compressed instructions. The program is shrunk from 40 to 22 bytes.

Many RISC-V assemblers generate code using a mix of compressed and uncompressed instructions, using compressed instructions wherever possible to minimize code size.

Code Example 6.32 USING COMPRESSED INSTRUCTIONS

High-Level Code	RISC-V Assembly Code
<pre>int i; int scores[200]; for (i = 0; i < 200; i = i + 1) scores[i] = scores[i] + 10;</pre>	<pre># s0 = scores base address, s1 = i c.li s1, 0 # i = 0 addi t2, zero, 200 # t2 = 200 for: bge s1, t2, done # if i >= 200 then done c.lw a3, 0(s0) # a3 = scores[i] c.addi a3, 10 # a3 = scores[i] + 10 c.sw a3, 0(s0) # scores[i] = a3 c.addi s0, 4 # next element of # scores c.addi s1, 1 # i = i + 1 c.j for # repeat done:</pre>

6.7 EVOLUTION OF THE RISC-V ARCHITECTURE

RISC-V was designed to be a commercially viable, open-source computer architecture, that is robust, efficient, and flexible. RISC-V differentiates itself from other architectures because it is open source, uses base instruction sets to ease compatibility, supports the full range of micro-architectures, from embedded systems to high-performance computers, offers both defined and customizable extensions, and provides features, such as compressed instructions and RV128I, that optimize hardware and support both existing and future designs, ensuring the architecture's longevity.

RISC-V has also created a community of both industry and academic partners by forming RISC-V International (see riscv.org), thereby accelerating innovation, commercialization, and collaboration. This consortium of developers also helps design and ratify the RISC-V architecture. RISC-V International has grown to have more than 500 industrial and academic members as of 2021, including Western Digital, NVIDIA, Microchip, and Samsung.

The RISC-V architecture is described in *The RISC-V Instruction Set Manual* (riscv.org/specifications). Early versions of the manual, through version 2.2, are a gem of a specification, succinct, readable, and interspersed with the rationale behind the design decisions embodied in the architecture.

6.7.1 RISC-V Base Instruction Sets and Extensions

RISC-V includes various base instruction sets and extensions so that it can support a broad range of processors—from small, inexpensive embedded processors, such as those in handheld devices, to high-performance, multicore, multithreaded systems. RISC-V has 32-, 64-, and 128-bit base instruction sets: RV32I/E, RV64I, and RV128I. The 32-bit base instruction set comes in the standard version RV32I discussed in this chapter and in an embedded version RV32E with only 16 registers, intended for very low-cost processors. As of 2021, only the RV32I and RV64I instruction sets are frozen; RV32E and RV128I are still being defined. Along with these base architectures, RISC-V also defines the extensions listed in Table 6.8. The most commonly used extensions—those for floating-point operations (RVF/D/Q), compressed instructions (RVC), and atomic instructions (RVA)—are fully specified and frozen to enable development and commercialization. The remaining extensions are still being developed.

All RISC-V processors must support one of the base architectures—RV32/64/128I or RV32E—and may optionally support extensions, such as the compressed or floating-point extensions. By using extensions, instead of new architecture versions, RISC-V alleviates the burden of backward or forward compatibility between microarchitectures. All

Table 6.8 RISC-V extensions

Extension	Description	Status
M	Integer multiplication and division	Frozen
F	Single-precision floating-point	Frozen
D	Double-precision floating-point	Frozen
Q	Quad-precision floating-point	Frozen
C	Compressed instructions	Frozen
A	Atomic instructions	Frozen
B	Bit manipulations	Open
L	Decimal floating-point	Open
J	Dynamically translated languages	Open
T	Transactional memory	Open
P	Packed-SIMD instructions	Open
V	Vector operations	Open

processors must support at least the base architecture. However, a processor need not support all (or even any) of the extensions.

To understand the evolution of the RISC-V architecture, it is important to understand other architectures that preceded RISC-V and, notably, the MIPS architecture. RISC-V follows many principles from the MIPS architecture but also benefits from the perspective of modern architectures and applications to include features that support embedded, multicore, and multithreaded systems and extensibility, among other features. The next section compares the RISC-V and MIPS architectures.

6.7.2 Comparison of RISC-V and MIPS Architectures

The RISC-V architecture has many similarities to the MIPS architecture developed by John Hennessy in the 1980's, but it eliminates some unnecessary complexity—and introduces some, in the case of immediates! Similarities include assembly and machine code formats, instruction mnemonics, register naming, and stack and calling conventions. Differences include RISC-V's immediate sizes and encodings, branching relative to PC (instead of PC+4), both branches and jumps being PC-relative, removal of the branch delay slot from MIPS, a strict definition of source and destination register instruction fields, different numbers of temporary, saved, and argument registers, and more extensibility by including more control bits in the instruction. By keeping **rs1**, **rs2**, and **rd** in the same bitfields of every instruction type that uses them, RISC-V simplifies the decoder hardware relative to MIPS. Similarly, RISC-V's immediate encodings simplify the immediate extension hardware.

6.7.3 Comparison of RISC-V and ARM Architectures

ARM is a RISC architecture that was developed around the same time as the MIPS architecture in the 1980's. Over the past decade or more, ARM processors have dominated the mobile devices arena and are also found in other applications, such as robots, pinball machines, and servers. ARM's similarities to RISC-V include its small number of machine code formats and assembly instructions and similar stack and calling conventions. ARM differs from RISC-V by including conditional execution, complex indexing modes for accessing memory, its ability to push and pop multiple registers onto/off the stack using a single instruction, optionally shifted source registers, and unconventional immediate encodings. Immediates are encoded as an 8-bit value and a 4-bit rotation, and they encode only positive immediates (subtraction is determined by the control bits). These features—particularly, conditional execution, shifted registers, and indexing modes—are typically only found in CISC architectures, but ARM includes them to reduce program

and, thus, memory size, which is critical for embedded and handheld devices. However, these design decisions also result in more complex hardware.

6.8 ANOTHER PERSPECTIVE: x86 ARCHITECTURE

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

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

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

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

This section introduces the x86 architecture. The goal is not to make you into an x86 assembly language programmer but rather to illustrate some of the similarities and differences between x86 and RISC-V. We think it is interesting to see how x86 works. However, none of the material in this section is needed to understand the rest of the book. Major differences between x86 and RISC-V (RV32I) are summarized in [Table 6.9](#).

6.8.1 x86 Registers

The 8086 microprocessor provided eight 16-bit registers. It could separately access the upper and lower eight bits of some of these registers. When the 32-bit 80386 was introduced, the registers were extended to

Table 6.9 Major differences between RISC-V (RV32I) and x86

Feature	RISC-V	x86
# of registers	32 general-purpose	8, some restrictions on purpose
# of operands	3 (2 sources, 1 destination)	2 (1 source, 1 source/destination)
Operand locations	Registers or immediates	Registers, immediates, or memory
Operand size	32 bits	8, 16, or 32 bits
Condition flags	No	Yes
Instruction types	Simple	Simple and complicated
Instruction encoding	Fixed: 4 bytes	Variable: 1–15 bytes

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 RISC-V, ESP is normally reserved for the stack pointer.

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

6.8.2 x86 Operands

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

RISC-V instructions generally specify three operands: two sources and one destination. x86 instructions specify only two operands. The first is a source. The second is both a source and the destination. Hence, x86 instructions always overwrite one of their sources with the result. [Table 6.10](#) lists the combinations of operand locations in x86. All combinations are possible except memory to memory.

Like RISC-V (RV32I), x86 has a 32-bit memory space that is byte-addressable. However, unlike RISC-V, x86 supports a wider variety of memory indexing modes. Memory locations are specified with any combination of a *base register*, *displacement*, and a *scaled index register*. [Table 6.11](#) illustrates these combinations. The displacement can be an

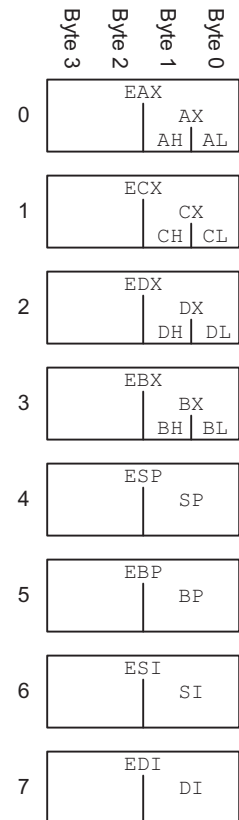
**Figure 6.36** x86 registers

Table 6.10 Operand locations

Source/ Destination	Source	Example	Meaning
register	register	add EAX, EBX	$EAX \leftarrow EAX + EBX$
register	immediate	add EAX, 42	$EAX \leftarrow EAX + 42$
register	memory	add EAX, [20]	$EAX \leftarrow EAX + \text{Mem}[20]$
memory	register	add [20], EAX	$\text{Mem}[20] \leftarrow \text{Mem}[20] + EAX$
memory	immediate	add [20], 42	$\text{Mem}[20] \leftarrow \text{Mem}[20] + 42$

Table 6.11 Memory addressing modes

Example	Meaning	Comment
add EAX, [20]	$EAX \leftarrow EAX + \text{Mem}[20]$	displacement
add EAX, [ESP]	$EAX \leftarrow EAX + \text{Mem}[\text{ESP}]$	base addressing
add EAX, [EDX+40]	$EAX \leftarrow EAX + \text{Mem}[\text{EDX}+40]$	base + displacement
add EAX, [60+EDI*4]	$EAX \leftarrow EAX + \text{Mem}[60+\text{EDI}*4]$	displacement + scaled index
add EAX, [EDX+80+EDI*2]	$EAX \leftarrow EAX + \text{Mem}[\text{EDX}+80+\text{EDI}*2]$	base + displacement + scaled index

Table 6.12 Instructions acting on 8-, 16-, or 32-bit data

Example	Meaning	Data Size
add AH, BL	$AH \leftarrow AH + BL$	8-bit
add AX, -1	$AX \leftarrow AX + 0xFFFF$	16-bit
add EAX, EDX	$EAX \leftarrow EAX + EDX$	32-bit

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 RISC-V base addressing mode for loads and stores, but RISC-V instructions do not allow for scaling. x86 also provides a scaled index. In x86, the scaled index provides an easy way to access arrays or structures of 2-, 4-, or 8-byte elements without having to issue a sequence of instructions to generate the address. While RISC-V always acts on 32-bit words, x86 instructions can operate on 8-, 16-, or 32-bit data. [Table 6.12](#) illustrates these variations.

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

6.8.3 Status Flags

x86, like many CISC architectures, uses condition flags (also called *status flags*) to make decisions about branches and to keep track of carries and arithmetic overflow. x86 uses a 32-bit register, called EFLAGS, that stores the status flags. Some of the bits of the EFLAGS register are given in Table 6.13. Other bits are used by the operating system. The architectural state of an x86 processor includes EFLAGS as well as the eight registers and the EIP.

6.8.4 x86 Instructions

x86 has a larger set of instructions than RISC-V. Table 6.14 describes some of the general-purpose instructions. x86 also has instructions for floating-point arithmetic and for arithmetic on multiple short data elements packed into a longer word. D indicates the destination (a register or memory location), and S indicates the source (a register, memory location, or immediate).

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

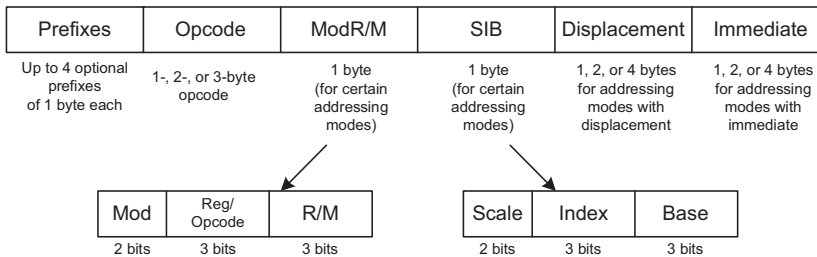
Conditional jumps check the flags and branch if the appropriate condition is met. They come in many flavors. For example, JZ jumps if the zero flag (ZF) is 1. JNZ jumps if the zero flag is 0. The jumps usually follow an instruction, such as the compare instruction (CMP), that sets the flags. Table 6.15 lists some of the conditional jumps and how they depend on the flags set by a prior compare operation. Unlike RISC-V, conditional jumps (called conditional branches in RISC-V) usually require two instructions instead of one.

Table 6.14 Selected x86 instructions

Instruction	Meaning	Function
ADD/SUB	add/subtract	$D = D + S$ / $D = D - S$
ADDC	add with carry	$D = D + S + CF$
INC/DEC	increment/decrement	$D = D + 1$ / $D = D - 1$
CMP	compare	set flags based on $D - S$
NEG	negate	$D = -D$
AND/OR/XOR	logical AND/OR/XOR	$D = D \text{ op } S$
NOT	logical NOT	$D = \bar{D}$
IMUL/MUL	signed/unsigned multiply	$EDX:EAX = EAX \times D$
IDIV/DIV	signed/unsigned divide	$EDX:EAX/D$ $EAX = \text{quotient}$; $EDX = \text{remainder}$
SAR/SHR	arithmetic/logical shift right	$D = D \ggg S$ / $D = D \gg S$
SAL/SHL	left shift	$D = D \ll S$
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 $D \text{ AND } S$
MOV	move	$D = S$
PUSH	push onto stack	$ESP = ESP - 4$; $\text{Mem}[ESP] = S$
POP	pop off stack	$D = \text{MEM}[ESP]$; $ESP = ESP + 4$
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 ($ECX \neq 0$) $EIP = EIP + \text{imm}$
CALL	function call	$ESP = ESP - 4$; $\text{MEM}[ESP] = EIP$; $EIP = S$
RET	function return	$EIP = \text{MEM}[ESP]$; $ESP = ESP + 4$

Table 6.15 Selected branch conditions

Instruction	Meaning	Function after <code>CMP D, S</code>
JZ/JE	jump if $ZF = 1$	jump if $D = S$
JNZ/JNE	jump if $ZF = 0$	jump if $D \neq S$
JGE	jump if $SF = 0F$	jump if $D \geq S$
JG	jump if $SF = 0F$ and $ZF = 0$	jump if $D > S$
JLE	jump if $SF \neq 0F$ or $ZF = 1$	jump if $D \leq S$
JL	jump if $SF \neq 0F$	jump if $D < S$
JC/JB	jump if $CF = 1$	
JNC	jump if $CF = 0$	
JO	jump if $OF = 1$	
JNO	jump if $OF = 0$	
JS	jump if $SF = 1$	
JNS	jump if $SF = 0$	

**Figure 6.37** x86 instruction encodings

6.8.5 x86 Instruction Encoding

The x86 instruction encodings are truly messy, a legacy of decades of piecemeal changes. Unlike RISC-V, whose instructions are uniformly 32 bits (or 16 bits for compressed instructions), x86 instructions vary from 1 to 15 bytes, as shown in Figure 6.37.³ The *opcode* may be 1, 2, or 3 bytes. It is followed by four optional fields: *ModR/M*, *SIB*, *Displacement*, and *Immediate*. *ModR/M* specifies an addressing mode. *SIB* specifies the scale, index, and base registers in certain addressing

³It is possible to construct 17-byte instructions if all of the optional fields are used. However, x86 places a 15-byte limit on the length of legal instructions.

modes. *Displacement* indicates a 1-, 2-, or 4-byte displacement in certain addressing modes. *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.

RISC-V fully specifies the instruction in the **op**, **funct3**, and **funct7** fields of the instruction. x86 uses a variable number of bits to specify different instructions. It uses fewer bits to specify more common instructions, decreasing the average length of the instructions. Some instructions even have multiple opcodes. For example, `ADD AL, imm8` performs an 8-bit add of an immediate to AL. It is represented with the 1-byte opcode, 0x04, followed by a 1-byte immediate. The A register (AL, AX, or EAX) is called the *accumulator*. On the other hand, `ADD D, imm8` performs an 8-bit add of an immediate to an arbitrary destination, *D* (memory or a register). It is represented with the 1-byte opcode 0x80 followed by one or more bytes specifying *D*, followed by a 1-byte immediate. Many instructions have shortened encodings when the destination is the accumulator.

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

6.8.6 Other x86 Peculiarities

The 80286 introduced *segmentation* to divide memory into segments of up to 64 KB in length. When the OS enables segmentation, addresses are

computed relative to the beginning of the segment. The processor checks for addresses that go beyond the end of the segment and indicates an error, thus preventing programs from accessing memory outside their own segment. Segmentation proved to be a hassle for programmers and is not used in modern versions of the Windows operating system.

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

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

Intel and Hewlett-Packard jointly developed a new 64-bit architecture called IA-64 in the mid-1990's. It was designed from a clean slate, bypassing the convoluted history of x86, taking advantage of 20 years of new research in computer architecture, and providing a 64-bit address space. However, the first IA-64 chip was too late to market and never became a commercial success. Most computers needing the large address space now use the 64-bit extensions of x86.

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

For those interested in examining x86 architecture in more detail, the *x86 Intel Architecture Software Developer's Manual* is freely available on Intel's website.

6.8.7 The Big Picture

This section has given a taste of some of the differences between the RISC-V architecture and the x86 CISC architecture. x86 tends to have shorter programs because a complex instruction is equivalent to a series of simple RISC-V instructions and because the instructions are encoded to minimize memory usage. However, the x86 architecture is a hodgepodge of features accumulated over the years, some of which are no longer useful but must be kept for compatibility with old programs.

It has too few registers, and the instructions are difficult to decode. Merely explaining the instruction set is difficult. Despite all these failings, x86 is firmly entrenched as the dominant computer architecture for PCs because the value of software compatibility is so great and because the huge market justifies the effort required to build fast x86 microprocessors.

6.9 SUMMARY

To command a computer, you must speak its language. A computer architecture defines how to command a processor. Many different computer architectures are in widespread commercial use today, but once you understand one, learning others is much easier. The key questions to ask when approaching a new architecture are:

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

RISC-V (RV32I) is a 32-bit architecture because it operates on 32-bit data. The RISC-V 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 functions written by different programmers can communicate easily. For example, register 0 (zero) always holds the constant 0, *ra* holds the return address after a *jal* instruction, and *a0* to *a7* hold the arguments of a function. *a0* to *a1* hold a function's return value. RISC-V has a byte-addressable memory system with 32-bit addresses. Instructions are 32 bits long and are word-aligned for efficient access. This chapter discussed the most commonly used RISC-V 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 i9 or AMD Ryzen processors in 2021.

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. We believe it is one of the most exciting topics in all of engineering: You will learn to build your own microprocessor!