2

Instructions

Language of the Computer

Abstract

This chapter describes instructions, the language of the computer. It explains the two principles of the stored-program computer: the use of instructions that are indistinguishable from numbers and the use of alterable memory for programs. The "instruction set architecture" (ISA) is an abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly. Above this machine level is assembly language, a language that humans can read. The assembler translates the language into the binary numbers that machines can understand, and it even "extends" the instruction set by creating symbolic instructions that aren't in the hardware. Each category of RISC-V instructions is associated with constructs that appear in programming languages. The popularity of a few instructions dominates the many. The varying popularity of these instructions plays an important role in the chapters about datapath, control, and pipelining.

Keywords

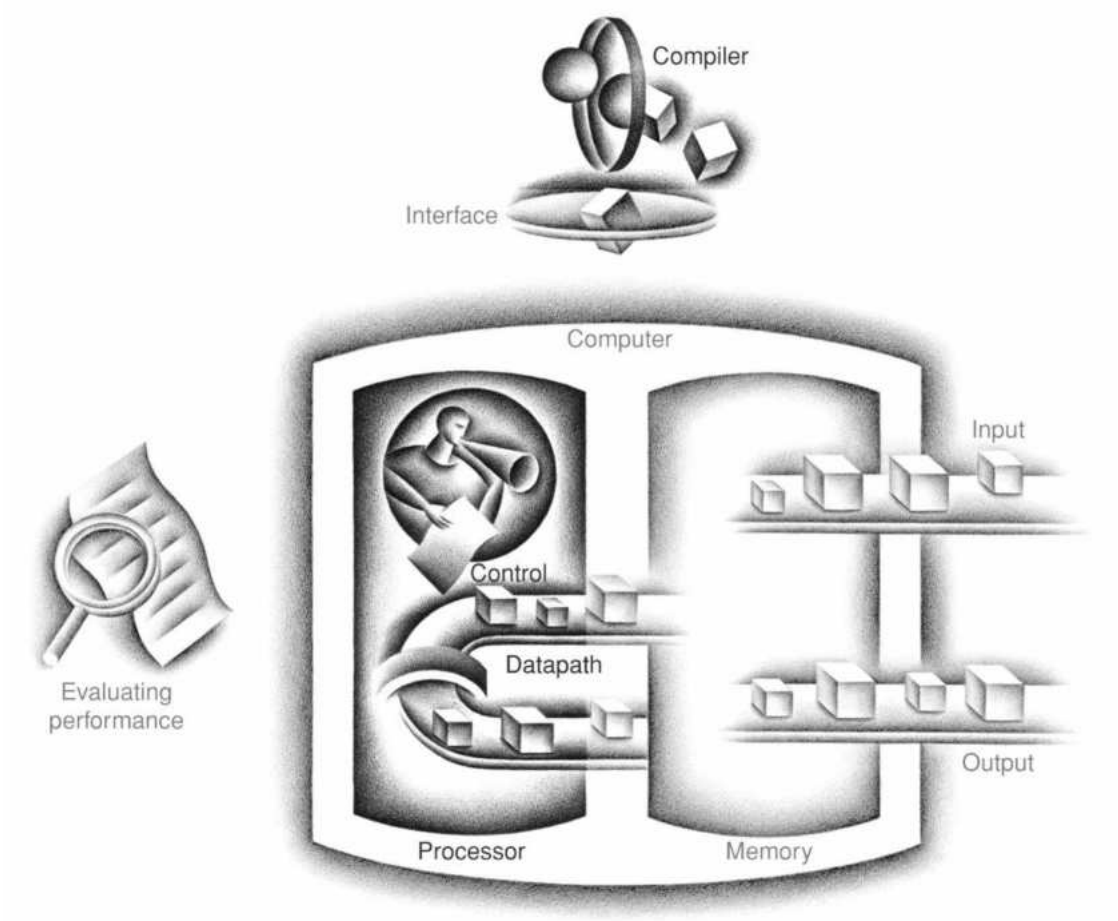
Operand; signed number; unsigned number; instructions; logical operations; procedures; synchronization; C Sort; arrays; pointers; compiling C; C; interpreting Java; Java; ARM instructions; RISC-V; MIPS; x86 instructions; instruction set; stored-program concept; word; doubleword; data transfer instruction; address; alignment restriction; binary digit; binary bit; least significant bit; most significant bit; one's complement; biased notation; instruction format; machine language; hexadecimal; opcode; AND; OR; NOT; XOR; EOR; ORR; conditional branch; basic block; branch address table; branch table; procedure; branch-and-link instruction;

return address; caller; callee; program counter; PC; stack; stack pointer; push; pop; global pointer; procedure frame; activation record; frame pointer; text segment; PC-relative addressing; addressing mode; data race; assembly language; pseudoinstruction; symbol table; linker; link editor; executable file; loader; dynamically linked libraries; DLL; Java bytecode; Java Virtual Machine; JVM; Just In Time compiler; JIT; object oriented language; general-purpose register; GPR

I speak Spanish to God, Italian to women, French to men, and German to my horse.

Charles V, Holy Roman Emperor (1500–1558)

OUTLINE



The Five Classic Components of a Computer

2.1 Introduction

To command a computer's hardware, you must speak its language. The words of a computer's language are called *instructions*, and its vocabulary is called an **instruction set**. In this chapter, you will see the instruction set of a real computer, both in the form written by people and in the form read by the computer. We introduce instructions in a top-down fashion. Starting from a notation that looks like a restricted programming language, we refine it step-by-step until you see the actual language of a real computer. Chapter 3 continues our downward descent, unveiling the hardware for arithmetic and the representation of floating-point numbers.

instruction set

The vocabulary of commands understood by a given architecture.

You might think that the languages of computers would be as diverse as those of people, but in reality, computer languages are quite similar, more like regional dialects than independent languages. Hence, once you learn one, it is easy to pick up others.

The chosen instruction set is RISC-V, which was originally developed at UC Berkeley starting in 2010.

To demonstrate how easy it is to pick up other instruction sets, we will also take a quick look at two other popular instruction sets.
1. MIPS is an elegant example of the instruction sets designed since the 1980s. In several respects, RISC-V follows a similar design.
2. The Intel x86 originated in the 1970s, but still today powers both the PC and the Cloud of the post-PC era.

This similarity of instruction sets occurs because all computers are constructed from hardware technologies based on similar underlying principles and because there are a few basic operations that all computers must provide. Moreover, computer designers have a common goal: to find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and energy. This goal is time-honored; the following quote was written before you could buy a computer, and it is as true today as it was in 1947:

> *It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations…. The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.*
>
> *Burks, Goldstine, and von Neumann, 1947*

The "simplicity of the equipment" is as valuable a consideration for today's computers as it was for those of the 1950s. The goal of this chapter is to teach an instruction set that follows this advice, showing both how it is represented in hardware and the

relationship between high-level programming languages and this more primitive one. Our examples are in the C programming language;  **Section 2.15** shows how these would change for an object-oriented language like Java.

By learning how to represent instructions, you will also discover the secret of computing: the **stored-program concept**. Moreover, you will exercise your "foreign language" skills by writing programs in the language of the computer and running them on the simulator that comes with this book. You will also see the impact of programming languages and compiler optimization on performance. We conclude with a look at the historical evolution of instruction sets and an overview of other computer dialects.

stored-program concept

The idea that instructions and data of many types can be stored in memory as numbers and thus be easy to change, leading to the stored-program computer.

We reveal our first instruction set a piece at a time, giving the rationale along with the computer structures. This top-down, step-by-step tutorial weaves the components with their explanations, making the computer's language more palatable. Figure 2.1 gives a sneak preview of the instruction set covered in this chapter.

RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory words	Memory[0], Memory[8], ..., Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands; add
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands; subtract
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
Data transfer	Load doubleword	ld x5, 40(x6)	x5 = Memory[x6 + 40]	Doubleword from memory to register
	Store doubleword	sd x5, 40(x6)	Memory[x6 + 40] = x5	Doubleword from register to memory
	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate
Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

FIGURE 2.1 RISC-V assembly language revealed in this chapter.
This information is also found in Column 1 of the RISC-V Reference Data Card at the front of this book.

2.2 Operations of the Computer

Hardware

There must certainly be instructions for performing the fundamental arithmetic operations.

Burks, Goldstine, and von Neumann, 1947

Every computer must be able to perform arithmetic. The RISC-V assembly language notation

```
add a, b, c
```

instructs a computer to add the two variables `b` and `c` and to put their sum in `a`.

This notation is rigid in that each RISC-V arithmetic instruction performs only one operation and must always have exactly three variables. For example, suppose we want to place the sum of four variables `b`, `c`, `d`, and `e` into variable `a`. (In this section, we are being deliberately vague about what a "variable" is; in the next section, we'll explain in detail.)

The following sequence of instructions adds the four variables:

```
add a, b, c  // The sum of b and c is placed in a
add a, a, d  // The sum of b, c, and d is now in a
add a, a, e  // The sum of b, c, d, and e is now in a
```

Thus, it takes three instructions to sum the four variables.

The words to the right of the double slashes (`//`) on each line above are *comments* for the human reader, so the computer ignores them. Note that unlike other programming languages, each line of this language can contain at most one instruction. Another difference from C is that comments always terminate at the end of a line.

The natural number of operands for an operation like addition is three: the two numbers being added together and a place to put the sum. Requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple: hardware for a variable number of operands is more complicated than hardware for a fixed number. This situation illustrates the first of three underlying principles of hardware design:

Design Principle 1: Simplicity favors regularity.

We can now show, in the two examples that follow, the relationship of programs written in higher-level programming languages to programs in this more primitive notation.

Compiling Two C Assignment Statements into RISC-V

Example

This segment of a C program contains the five variables `a`, `b`, `c`, `d`, and `e`. Since Java evolved from C, this example and the next few work for either high-level programming language:

```
a = b + c;
d = a -e;
```

The *compiler* translates from C to RISC-V assembly language instructions. Show the RISC-V code produced by a compiler.

Answer

A RISC-V instruction operates on two source operands and places the result in one destination operand. Hence, the two simple statements above compile directly into these two RISC-V assembly language instructions:

```
add a, b, c
sub d, a, e
```

Compiling a Complex C Assignment into RISC-V

Example

A somewhat complicated statement contains the five variables `f`, `g`, `h`, `i`, and `j`:

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

What might a C compiler produce?

Answer

The compiler must break this statement into several assembly instructions, since only one operation is performed per RISC-V instruction. The first RISC-V instruction calculates the sum of `g` and `h`. We must place the result somewhere, so the compiler creates a temporary variable, called `t0`:

```
add t0, g, h // temporary variable t0 contains g + h
```

Although the next operation is subtract, we need to calculate the sum of `i` and `j` before we can subtract. Thus, the second instruction places the sum of `i` and `j` in another temporary variable created by the compiler, called `t1`:

```
add t1, i, j // temporary variable t1 contains i + j
```

Finally, the subtract instruction subtracts the second sum from the first and places the difference in the variable `f`, completing the compiled code:

```
sub f, t0, t1 // f gets t0 –t1, which is (g + h) –(i + j)
```

Check Yourself

For a given function, which programming language likely takes the most lines of code? Put the three representations below in order.

1. Java
2. C
3. RISC-V assembly language

Elaboration

To increase portability, Java was originally envisioned as relying on a software interpreter. The instruction set of this interpreter is called *Java bytecodes* (see  **Section 2.15**), which is quite different from the RISC-V instruction set. To get performance close to the equivalent C program, Java systems today typically compile Java bytecodes into the native instruction sets like RISC-V. Because this compilation is normally done much later than for C programs, such Java compilers are often called *Just In Time* (JIT) compilers. Section 2.12 shows how JITs are used later than C compilers in the start-up process, and Section 2.13 shows the performance consequences of compiling versus interpreting Java programs.

2.3 Operands of the Computer Hardware

Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited

number of special locations built directly in hardware called *registers*. Registers are primitives used in hardware design that are also visible to the programmer when the computer is completed, so you can think of registers as the bricks of computer construction. The size of a register in the RISC-V architecture is 64 bits; groups of 64 bits occur so frequently that they are given the name **doubleword** in the RISC-V architecture. (Another popular size is a group of 32 bits, called a **word** in the RISC-V architecture.)

doubleword

Another natural unit of access in a computer, usually a group of 64 bits; corresponds to the size of a register in the RISC-V architecture.

word

A natural unit of access in a computer, usually a group of 32 bits.

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers, like RISC-V. (See  **Section 2.21** for the history of the number of registers.) Thus, continuing in our top-down, stepwise evolution of the symbolic representation of the RISC-V language, in this section we have added the restriction that the three operands of RISC-V arithmetic instructions must each be chosen from one of the 32 64-bit registers.

The reason for the limit of 32 registers may be found in the second of our three underlying design principles of hardware technology:

Design Principle 2: Smaller is faster.

A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.

Guidelines such as "smaller is faster" are not absolutes; 31 registers may not be faster than 32. Even so, the truth behind such observations causes computer designers to take them seriously. In this case, the designer must balance the craving of programs for

more registers with the designer's desire to keep the clock cycle fast. Another reason for not using more than 32 is the number of bits it would take in the instruction format, as Section 2.5 demonstrates.

Chapter 4 shows the central role that registers play in hardware construction; as we shall see in that chapter, effective use of registers is critical to program performance.

Although we could simply write instructions using numbers for registers, from 0 to 31, the RISC-V convention is `x` followed by the number of the register, except for a few register names that we will cover later.

Compiling a C Assignment Using Registers

Example

It is the compiler's job to associate program variables with registers. Take, for instance, the assignment statement from our earlier example:

```
f = (g + h) −(i + j);
```

The variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `x19`, `x20`, `x21`, `x22`, and `x23`, respectively. What is the compiled RISC-V code?

Answer

The compiled program is very similar to the prior example, except we replace the variables with the register names mentioned above plus two temporary registers, `x5` and `x6`, which correspond to the temporary variables above:

```
add x5, x20, x21// register x5 contains g + h
add x6, x22, x23// register x6 contains i + j
sub x19, x5, x6// f gets x5 −x6, which is (g + h)−(i + j)
```

Memory Operands

Programming languages have simple variables that contain single data elements, as in these examples, but they also have more complex data structures—arrays and structures. These composite data structures can contain many more data elements than there are registers in a computer. How can a computer represent and access

such large structures?

Recall the five components of a computer introduced in Chapter 1 and repeated on page 61. The processor can keep only a small amount of data in registers, but computer memory contains billions of data elements. Hence, data structures (arrays and structures) are kept in memory.

As explained above, arithmetic operations occur only on registers in RISC-V instructions; thus, RISC-V must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**. To access a word or doubleword in memory, the instruction must supply the memory **address**. Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0. For example, in Figure 2.2, the address of the third data element is 2, and the value of memory [2] is 10.

data transfer instruction

A command that moves data between memory and registers.

address

A value used to delineate the location of a specific data element within a memory array.

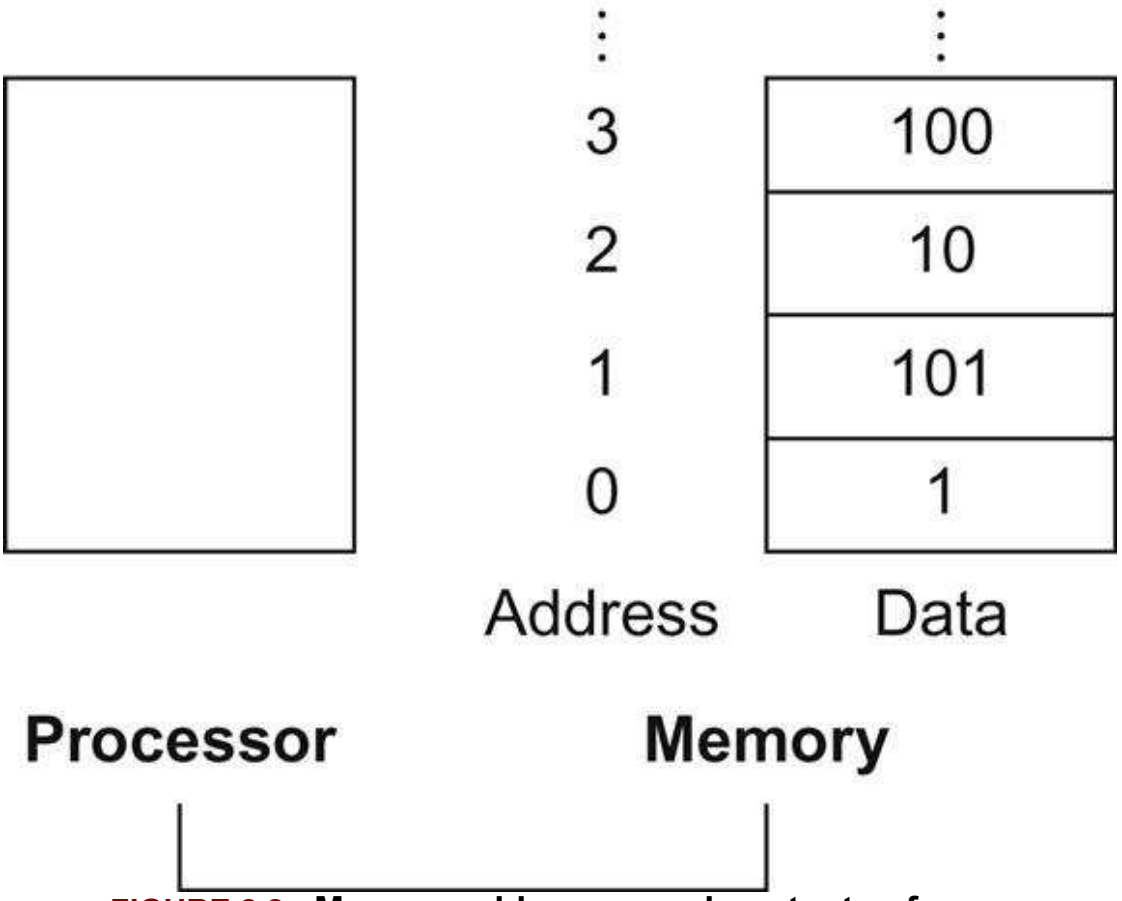


FIGURE 2.2 Memory addresses and contents of memory at those locations.
If these elements were doublewords, these addresses would be incorrect, since RISC-V actually uses byte addressing, with each doubleword representing 8 bytes. Figure 2.3 shows the correct memory addressing for sequential doubleword addresses.

The data transfer instruction that copies data from memory to a register is traditionally called *load*. The format of the load instruction is the name of the operation followed by the register to be loaded, then register and a constant used to access memory. The sum of the constant portion of the instruction and the contents of the second register forms the memory address. The real RISC-V name for this instruction is `ld`, standing for *load doubleword*.

Compiling an Assignment When an Operand Is in Memory

Example

Let's assume that `A` is an array of 100 doublewords and that the compiler has associated the variables `g` and `h` with the registers `x20` and `x21` as before. Let's also assume that the starting address, or *base address*, of the array is in `x22`. Compile this C assignment statement:

```
g = h + A[8];
```

Answer

Although there is a single operation in this assignment statement, one of the operands is in memory, so we must first transfer `A[8]` to a register. The address of this array element is the sum of the base of the array `A`, found in register `x22`, plus the number to select element 8. The data should be placed in a temporary register for use in the next instruction. Based on Figure 2.2, the first compiled instruction is

```
ld  x9, 8(x22) // Temporary reg x9 gets A[8]
```

(We'll be making a slight adjustment to this instruction, but we'll use this simplified version for now.) The following instruction can operate on the value in `x9` (which equals `A[8]`) since it is in a register. The instruction must add `h` (contained in `x21`) to `A[8]` (contained in `x9`) and put the sum in the register corresponding to `g` (associated with `x20`):

```
add  x20, x21, x9 // g = h + A[8]
```

The register added to form the address (`x22`) is called the *base register*, and the constant in a data transfer instruction (8) is called the *offset*.

Hardware/Software Interface

In addition to associating variables with registers, the compiler allocates data structures like arrays and structures to locations in memory. The compiler can then place the proper starting address into the data transfer instructions.

Since 8-bit *bytes* are useful in many programs, virtually all architectures today address individual bytes. Therefore, the address of a doubleword matches the address of one of the 8 bytes within the doubleword, and addresses of sequential doublewords differ by 8. For example, Figure 2.3 shows the actual RISC-V addresses for the doublewords in Figure 2.2; the byte address of

the third doubleword is 16.

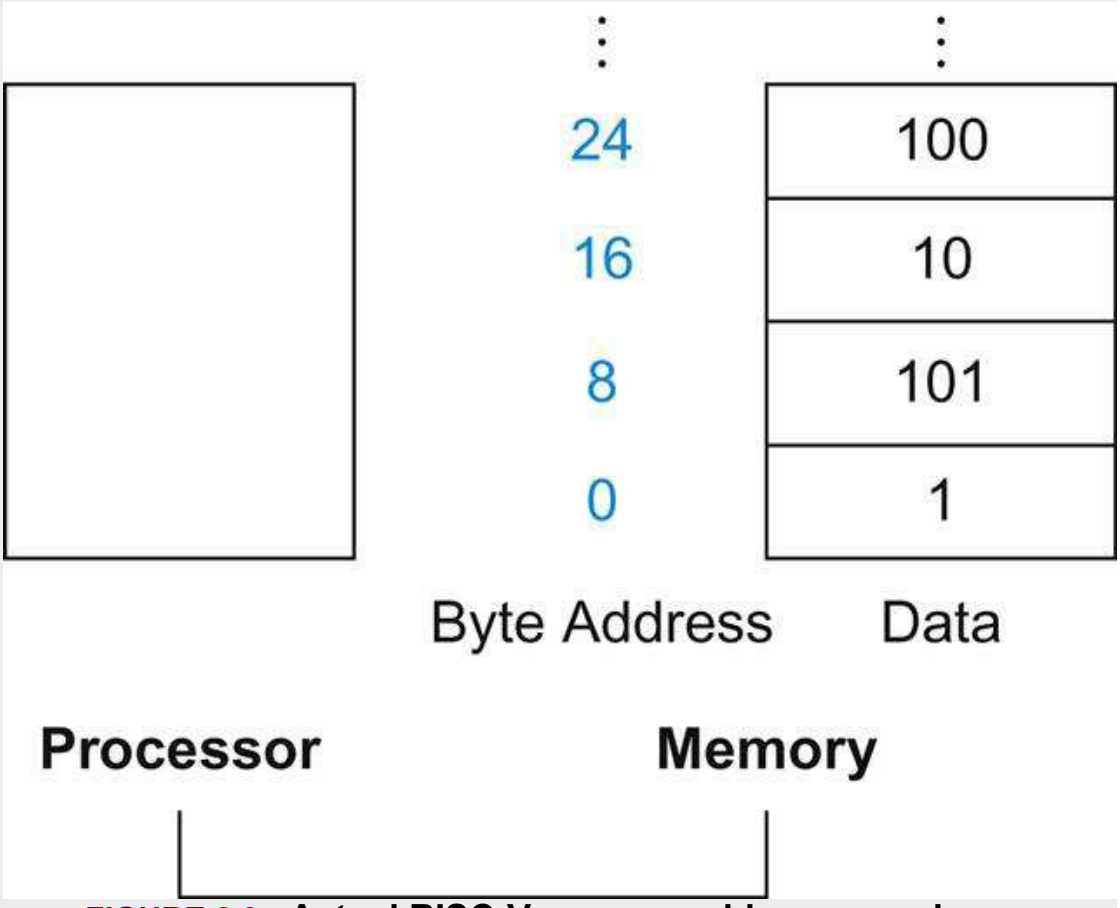


FIGURE 2.3 Actual RISC-V memory addresses and contents of memory for those doublewords. The changed addresses are highlighted to contrast with Figure 2.2. Since RISC-V addresses each byte, doubleword addresses are multiples of 8: there are 8 bytes in a doubleword.

Computers divide into those that use the address of the leftmost or "big end" byte as the doubleword address versus those that use the rightmost or "little end" byte. RISC-V belongs to the latter camp, referred to as *little-endian*. Since the order matters only if you access the identical data both as a doubleword and as eight individual bytes, few need to be aware of the "endianness."

Byte addressing also affects the array index. To get the proper byte address in the code above, *the offset to be added to the base register* `x22` *must be 8×8, or 64*, so that the load address will select `A[8]` and not `A[8/8]`. (See the related *Pitfall* on page 159 of Section 2.19.)

The instruction complementary to load is traditionally called *store;* it copies data from a register to memory. The format of a store is similar to that of a load: the name of the operation, followed by the register to be stored, then the base register, and finally the offset to select the array element. Once again, the RISC-V address is specified in part by a constant and in part by the contents of a register. The actual RISC-V name is `sd`, standing for *store doubleword*.

Elaboration

In many architectures, words must start at addresses that are multiples of 4 and doublewords must start at addresses that are multiples of 8. This requirement is called an **alignment restriction**. (Chapter 4 suggests why alignment leads to faster data transfers.) RISC-V and Intel x86 do *not* have alignment restrictions, but MIPS does.

alignment restriction

A requirement that data be aligned in memory on natural boundaries.

Hardware/Software Interface

As the addresses in loads and stores are binary numbers, we can see why the DRAM for main memory comes in binary sizes rather than in decimal sizes. That is, in gibibytes (2^{30}) or tebibytes (2^{40}), not in gigabytes (10^{9}) or terabytes (10^{12}); see Figure 1.1.

Compiling Using Load and Store

Example

Assume variable `h` is associated with register `x21` and the base address of the array `A` is in `x22`. What is the RISC-V assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

Answer

Although there is a single operation in the C statement, now two of

the operands are in memory, so we need even more RISC-V instructions. The first two instructions are the same as in the prior example, except this time we use the proper offset for byte addressing in the load register instruction to select `A[8]`, and the `add` instruction places the sum in `x9`:

```
ld x9, 64(x22) // Temporary reg x9 gets A[8]
add x9, x21, x9 // Temporary reg x9 gets h + A[8]
```

The final instruction stores the sum into `A[12]`, using 96 (8×12) as the offset and register `x22` as the base register.

```
sdx9, 96(x22)// Stores h + A[8] back into A[12]
```

Load doubleword and store doubleword are the instructions that copy doublewords between memory and registers in the RISC-V architecture. Some brands of computers use other instructions along with load and store to transfer data. An architecture with such alternatives is the Intel x86, described in Section 2.17.

Hardware/Software Interface

Many programs have more variables than computers have registers. Consequently, the compiler tries to keep the most frequently used variables in registers and places the rest in memory, using loads and stores to move variables between registers and memory. The process of putting less frequently used variables (or those needed later) into memory is called *spilling* registers.

The hardware principle relating size and speed suggests that memory must be slower than registers, since there are fewer registers. This suggestion is indeed the case; data accesses are faster if data are in registers instead of memory.

Moreover, data are more useful when in a register. A RISC-V arithmetic instruction can read two registers, operate on them, and write the result. A RISC-V data transfer instruction only reads one operand or writes one operand, without operating on it.

Thus, registers take less time to access *and* have higher throughput than memory, making data in registers both considerably faster to access and simpler to use. Accessing registers also uses much less energy than accessing memory. To achieve the highest performance and conserve energy, an instruction set architecture must have enough registers, and compilers must use

registers efficiently.

Elaboration

Let's put the energy and performance of registers versus memory into perspective. Assuming 64-bit data, registers are roughly 200 times faster (0.25 vs. 50 nanoseconds) and are 10,000 times more energy efficient (0.1 vs. 1000 picoJoules) than DRAM in 2015. These large differences led to caches, which reduce the performance and energy penalties of going to memory (see Chapter 5).

Constant or Immediate Operands

Many times a program will use a constant in an operation—for example, incrementing an index to point to the next element of an array. In fact, more than half of the RISC-V arithmetic instructions have a constant as an operand when running the SPEC CPU2006 benchmarks.

Using only the instructions we have seen so far, we would have to load a constant from memory to use one. (The constants would have been placed in memory when the program was loaded.) For example, to add the constant 4 to register `x22`, we could use the code

```
ld  x9, AddrConstant4(x3)  // x9 = constant 4
add x22, x22, x9           // x22 = x22 + x9 (where x9 == 4)
```

assuming that `x3 + AddrConstant4` is the memory address of the constant 4.

An alternative that avoids the load instruction is to offer versions of the arithmetic instructions in which one operand is a constant. This quick add instruction with one constant operand is called *add immediate* or `addi`. To add 4 to register `x22`, we just write

```
addi  x22, x22, 4   // x22 = x22 + 4
```

Constant operands occur frequently; indeed, `addi` is the most popular instruction in most RISC-V programs. By including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory.

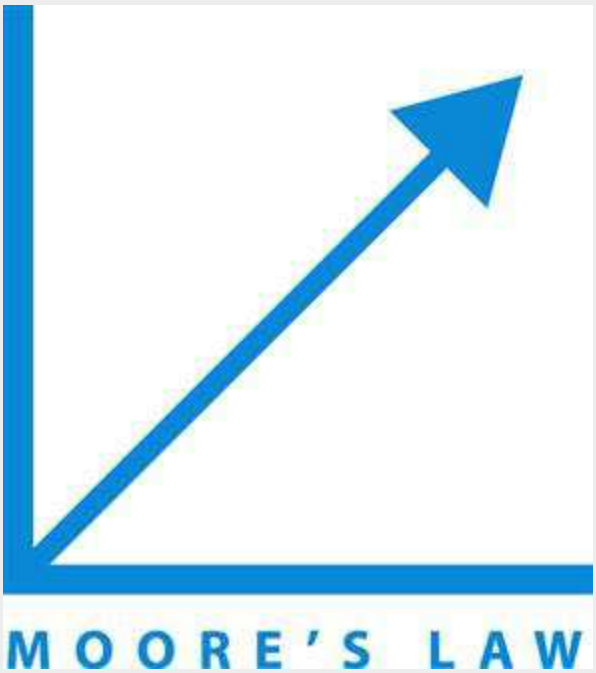
The constant zero has another role, which is to simplify the instruction set by offering useful variations. For example, you can negate the value in a register by using the `sub` instruction with zero

for the first operand. Hence, RISC-V dedicates register `x0` to be hard-wired to the value zero. Using frequency to justify the inclusions of constants is another example of the great idea from Chapter 1 of making the **common case fast**.

Check Yourself

Given the importance of registers, what is the rate of increase in the number of registers in a chip over time?

1. Very fast: They increase as fast as **Moore's Law**, which predicts doubling the number of transistors on a chip every 18 months.
2. Very slow: Since programs are usually distributed in the language of the computer, there is inertia in instruction set architecture, and so the number of registers increases only as fast as new instruction sets become viable.



Elaboration

Although the RISC-V registers in this book are 64 bits wide, the RISC-V architects conceived multiple variants of the ISA. In addition to this variant, known as RV64, a variant named RV32 has 32-bit registers, whose reduced cost make RV32 better suited to very low-cost processors.

Elaboration

The RISC-V offset plus base register addressing is an excellent match to structures as well as arrays, since the register can point to the beginning of the structure and the offset can select the desired element. We'll see such an example in Section 2.13.

Elaboration

The register in the data transfer instructions was originally invented to hold an index of an array with the offset used for the starting address of an array. Thus, the base register is also called the *index register*. Today's memories are much larger, and the software model of data allocation is more sophisticated, so the base address of the array is normally passed in a register since it won't fit in the offset, as we shall see.

Elaboration

The migration from 32-bit address computers to 64-bit address computers left compiler writers a choice of the size of data types in C. Clearly, pointers should be 64 bits, but what about integers? Moreover, C has the data types `int`, `long int`, and `long long int`. The problems come from converting from one data type to another and having an unexpected overflow in C code that is not fully standard compliant, which unfortunately is not rare code. The table below shows the two popular options:

Operating System	pointers	int	long int	long long int
Microsoft Windows	64 bits	32 bits	32 bits	64 bits
Linux, Most Unix	64 bits	32 bits	64 bits	64 bits

While each compiler could have different choices, generally the compilers associated with each operating system make the same decision. To keep the examples simple, in this book we'll assume pointers are all 64 bits and declare all C integers as `long long int` to keep them the same size. We also follow C99 standard and declare variables used as indexes to arrays to be `size_t`, which guarantees they are the right size no matter how big the array. They are typically declared the same as `long int`.

2.4 Signed and Unsigned Numbers

First, let's quickly review how a computer represents numbers. Humans are taught to think in base 10, but numbers may be represented in any base. For example, 123 base 10=1111011 base 2.

Numbers are kept in computer hardware as a series of high and low electronic signals, and so they are considered base 2 numbers. (Just as base 10 numbers are called *decimal* numbers, base 2 numbers are called *binary* numbers.)

A single digit of a binary number is thus the "atom" of computing, since all information is composed of **binary digits** or *bits*. This fundamental building block can be one of two values, which can be thought of as several alternatives: high or low, on or

off, true or false, or 1 or 0.

binary digit

Also called bit. One of the two numbers in base 2, 0 or 1, that are the components of information.

Generalizing the point, in any number base, the value of *i*th digit *d* is

$$d \times \text{Base}^{i}$$

where *i* starts at 0 and increases from right to left. This representation leads to an obvious way to number the bits in the doubleword: simply use the power of the base for that bit. We subscript decimal numbers with *ten* and binary numbers with *two*. For example,

1011_{two}

represents

$$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{ten}$$
$$= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{ten}$$
$$= 8+0+2+1_{ten}$$
$$= 11_{ten}$$

We number the bits 0, 1, 2, 3, … from *right to left* in a doubleword. The drawing below shows the numbering of bits within a RISC-V doubleword and the placement of the number 1011_{two}, (which we must unfortunately split in half to fit on the page of the book):

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	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
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(64 bits wide, split into two 32-bit rows)

Since doublewords are drawn vertically as well as horizontally, leftmost and rightmost may be unclear. Hence, the phrase **least significant bit** is used to refer to the rightmost bit (bit 0 above) and **most significant bit** to the leftmost bit (bit 63).

least significant bit

The rightmost bit in an RISC-V doubleword.

most significant bit

The leftmost bit in an RISC-V doubleword.

The RISC-V doubleword is 64 bits long, so we can represent 2^{64} different 64-bit patterns. It is natural to let these combinations represent the numbers from 0 to $2^{64} - 1$ ($18,446,774,073,709,551,615_{ten}$):

```
  00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000two= 0ten
  00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000001two= 1ten
  00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000010two= 2ten

  . . .          . . .
  11111111 11111111 11111111 11111111 11111111 11111111
11111111 11111101two= 18,446,774,073,709,551,613ten
  11111111 11111111 11111111 11111111 11111111 11111111
11111111 11111110two= 18,446,744,073,709,551,614ten
  11111111 11111111 11111111 11111111 11111111 11111111
11111111 11111111two= 18,446,744,073,709,551,615ten
```

That is, 64-bit binary numbers can be represented in terms of the bit value times a power of 2 (here x_i means the ith bit of x):

$$(x_{63} \times 2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

For reasons we will shortly see, these positive numbers are called unsigned numbers.

Hardware/Software Interface

Base 2 is not natural to human beings; we have 10 fingers and so find base 10 natural. Why didn't computers use decimal? In fact, the first commercial computer *did* offer decimal arithmetic. The problem was that the computer still used on and off signals, so a decimal digit was simply represented by several binary digits. Decimal proved so inefficient that subsequent computers reverted

to all binary, converting to base 10 only for the relatively infrequent input/output events.

Keep in mind that the binary bit patterns above are simply *representatives* of numbers. Numbers really have an infinite number of digits, with almost all being 0 except for a few of the rightmost digits. We just don't normally show leading 0s.

Hardware can be designed to add, subtract, multiply, and divide these binary bit patterns. If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, *overflow* is said to have occurred. It's up to the programming language, the operating system, and the program to determine what to do if overflow occurs.

Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative. The most obvious solution is to add a separate sign, which conveniently can be represented in a single bit; the name for this representation is *sign and magnitude*.

Alas, sign and magnitude representation has several shortcomings. First, it's not obvious where to put the sign bit. To the right? To the left? Early computers tried both. Second, adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be. Finally, a separate sign bit means that sign and magnitude has both a positive and a negative zero, which can lead to problems for inattentive programmers. Because of these shortcomings, sign and magnitude representation was soon abandoned.

In the search for a more attractive alternative, the question arose as to what would be the result for unsigned numbers if we tried to subtract a large number from a small one. The answer is that it would try to borrow from a string of leading 0s, so the result would have a string of leading 1s.

Given that there was no obvious better alternative, the final solution was to pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative. This convention for representing signed binary numbers is called *two's complement* representation:

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000$

$00000001_{two} = 1_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$

. . . . . .

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}805_{ten}$

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}806_{ten}$

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = -9{,}223{,}372{,}036{,}854{,}775{,}808_{ten}$

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = -9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = -9{,}223{,}372{,}036{,}854{,}775{,}806_{ten}$

… . . .

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = -3_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = -2_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = -1_{ten}$

The positive half of the numbers, from 0 to $9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$ ($2^{63}-1$), use the same representation as before. The following bit pattern ($1000\ \ldots\ 0000_{two}$) represents the most negative number $-9{,}223{,}372{,}036{,}854{,}775{,}808_{ten}$ (-2^{63}). It is followed by a declining set of negative numbers: $-9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$ ($1000\ \ldots\ 0001_{two}$) down to -1_{ten} ($1111\ \ldots\ 1111_{two}$).

Two's complement does have one negative number that has no corresponding positive number: $-9{,}223{,}372{,}036{,}854{,}775{,}808_{ten}$. Such imbalance was also a worry to the inattentive programmer, but sign and magnitude had problems for both the programmer *and* the hardware designer. Consequently, every computer today uses two's complement binary representations for signed numbers.

Two's complement representation has the advantage that all negative numbers have a 1 in the most significant bit. Thus, hardware needs to test only this bit to see if a number is positive or negative (with the number 0 is considered positive). This bit is often

called the *sign bit*. By recognizing the role of the sign bit, we can represent positive and negative 64-bit numbers in terms of the bit value times a power of 2:

$$(x_{63} \times -2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

The sign bit is multiplied by -2^{63}, and the rest of the bits are then multiplied by positive versions of their respective base values.

Binary to Decimal Conversion

Example

What is the decimal value of this 64-bit two's complement number?

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111100_{two}$$

Answer

Substituting the number's bit values into the formula above:

$$\begin{aligned}
&(1 \times -2^{63}) + (1 \times 2^{62}) + (1 \times 2^{61}) + \cdots + (1 \times 2^1) + (0 \times 2^1) + (0 \times 2^0) \\
&= -2^{63} + 2^{62} + 2^{61} + \cdots + 2^2 + 0 + 0 \\
&= -9,223,372,036,854,775,808_{ten} + 9,223,372,036,854,775,804_{ten} \\
&= -4_{ten}
\end{aligned}$$

We'll see a shortcut to simplify conversion from negative to positive soon.

Just as an operation on unsigned numbers can overflow the capacity of hardware to represent the result, so can an operation on two's complement numbers. Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect): a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive.

Hardware/Software Interface

Signed versus unsigned applies to loads as well as to arithmetic.

The *function* of a signed load is to copy the sign repeatedly to fill the rest of the register—called *sign extension*—but its *purpose* is to place a correct representation of the number within that register. Unsigned loads simply fill with 0s to the left of the data, since the number represented by the bit pattern is unsigned.

When loading a 64-bit doubleword into a 64-bit register, the point is moot; signed and unsigned loads are identical. RISC-V does offer two flavors of byte loads: *load byte unsigned* (`lbu`) treats the byte as an unsigned number and thus zero-extends to fill the leftmost bits of the register, while *load byte* (`lb`) works with signed integers. Since C programs almost always use bytes to represent characters rather than consider bytes as very short signed integers, `lbu` is used practically exclusively for byte loads.

Hardware/Software Interface

Unlike the signed numbers discussed above, memory addresses naturally start at 0 and continue to the largest address. Put another way, negative addresses make no sense. Thus, programs want to deal sometimes with numbers that can be positive or negative and sometimes with numbers that can be only positive. Some programming languages reflect this distinction. C, for example, names the former *integers* (declared as `long long int` in the program) and the latter *unsigned integers* (`unsigned long long int`). Some C style guides even recommend declaring the former as `signed long long int` to keep the distinction clear.

Let's examine two useful shortcuts when working with two's complement numbers. The first shortcut is a quick way to negate a two's complement binary number. Simply invert every 0 to 1 and every 1 to 0, then add one to the result. This shortcut is based on the observation that the sum of a number and its inverted representation must be 111 … 111_{two}, which represents −1. Since $x + \bar{x} = -1$, therefore $x + \bar{x} + 1 = 0$ or $\bar{x} + 1 = -x$. (We use the notation $\bar{x}$ to mean invert every bit in x from 0 to 1 and vice versa.)

Negation Shortcut

Example

Negate 2_{ten}, and then check the result by negating -2_{ten}.

Answer

2_{ten}=00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000010$_{two}$

Negating this number by inverting the bits and adding one,

$$\begin{array}{l} \quad 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} \\ + \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} \\ = -2_{ten} \end{array}$$

Going the other direction,

11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111110$_{two}$

is first inverted and then incremented:

$$\begin{array}{l} \quad 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} \\ + \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} \\ = 2_{ten} \end{array}$$

Our next shortcut tells us how to convert a binary number represented in *n* bits to a number represented with more than *n* bits. The shortcut is to take the most significant bit from the smaller quantity—the sign bit—and replicate it to fill the new bits of the larger quantity. The old nonsign bits are simply copied into the right portion of the new doubleword. This shortcut is commonly called *sign extension*.

Sign Extension Shortcut

Example

Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 64-bit binary numbers.

Answer

The 16-bit binary version of the number 2 is

$$00000000\ 00000010_{two} = 2_{ten}$$

It is converted to a 64-bit number by making 48 copies of the value in the most significant bit (0) and placing that in the left of the doubleword. The right part gets the old value:

$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$$

Let's negate the 16-bit version of 2 using the earlier shortcut. Thus,

$$0000\ 0000\ 0000\ 0010_{two}$$

becomes

$$\begin{array}{r} 1111\ 1111\ 1111\ 1101_{two} \\ + \qquad\qquad\qquad\qquad 1_{two} \\ \hline \\ = \ 1111\ 1111\ 1111\ 1110_{two} \end{array}$$

Creating a 64-bit version of the negative number means copying the sign bit 48 times and placing it on the left:

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = -2_{ten}$$

This trick works because positive two's complement numbers really have an infinite number of 0s on the left and negative two's complement numbers have an infinite number of 1s. The binary bit pattern representing a number hides leading bits to fit the width of the hardware; sign extension simply restores some of them.

Summary

The main point of this section is that we need to represent both positive and negative integers within a computer, and although there are pros and cons to any option, the unanimous choice since

1965 has been two's complement.

Elaboration

For signed decimal numbers, we used "–" to represent negative because there are no limits to the size of a decimal number. Given a fixed data size, binary and hexadecimal (see Figure 2.4) bit strings can encode the sign; therefore, we do not normally use "+" or "–" with binary or hexadecimal notation.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

FIGURE 2.4 The hexadecimal–binary conversion table.
Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

Check Yourself

What is the decimal value of this 64-bit two's complement number?

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111000_{two}$

1) -4_{ten}
2) -8_{ten}
3) -16_{ten}
4) $18,446,744,073,709,551,609_{ten}$

Elaboration

Two's complement gets its name from the rule that the unsigned sum of an n-bit number and its n-bit negative is 2^n; hence, the negation or complement of a number x is $2^n - x$, or its "two's complement."

A third alternative representation to two's complement and sign and magnitude is called **one's complement**. The negative of a one's complement is found by inverting each bit, from 0 to 1 and from 1 to 0, or $\bar{x}$. This relation helps explain its name since the complement of x is $2^n - x - 1$. It was also an attempt to be a better solution than sign and magnitude, and several early scientific computers did use the notation. This representation is similar to two's complement except that it also has two 0s: $00 \ldots 00_{two}$ is positive 0 and $11 \ldots 11_{two}$ is negative 0. The most negative number, $10 \ldots 000_{two}$, represents $-2{,}147{,}483{,}647_{ten}$, and so the positives and negatives are balanced. One's complement adders did need an extra step to subtract a number, and hence two's complement dominates today.

one's complement

A notation that represents the most negative value by $10 \ldots 000_{two}$ and the most positive value by $01 \ldots 11_{two}$, leaving an equal number of negatives and positives but ending up with two zeros, one positive ($00 \ldots 00_{two}$) and one negative ($11 \ldots 11_{two}$). The term is also used to mean the inversion of every bit in a pattern: 0 to 1 and 1 to 0.

A final notation, which we will look at when we discuss floating point in Chapter 3, is to represent the most negative value by $00 \ldots 000_{two}$ and the most positive value by $11 \ldots 11_{two}$, with 0 typically having the value $10 \ldots 00_{two}$. This representation is called a **biased notation**, since it biases the number such that the number plus the bias has a non-negative representation.

biased notation

A notation that represents the most negative value by $00 \ldots 000_{two}$ and the most positive value by $11 \ldots 11_{two}$, with 0 typically having the value $10 \ldots 00_{two}$, thereby biasing the number such that the number plus the bias has a non-negative representation.

2.5 Representing Instructions in the Computer

We are now ready to explain the difference between the way humans instruct computers and the way computers see instructions.

Instructions are kept in the computer as a series of high and low electronic signals and may be represented as numbers. In fact, each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction. The 32 registers of RISC-V are just referred to by their number, from 0 to 31.

Translating a RISC-V Assembly Instruction into a Machine Instruction

Example

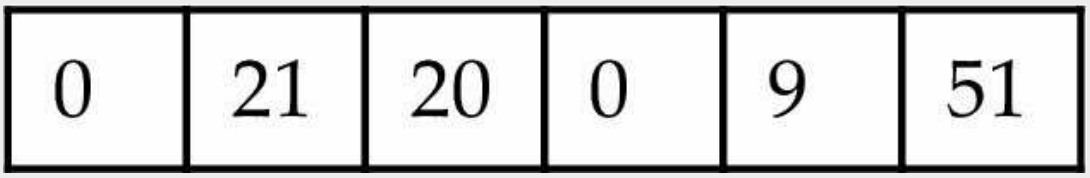
Let's do the next step in the refinement of the RISC-V language as an example. We'll show the real RISC-V language version of the instruction represented symbolically as

```
add x9, x20, x21
```

first as a combination of decimal numbers and then of binary numbers.

Answer

The decimal representation is



Each of these segments of an instruction is called a *field*. The first, fourth, and sixth fields (containing 0, 0, and 51 in this case) collectively tell the RISC-V computer that this instruction performs addition. The second field gives the number of the register that is the second source operand of the addition operation (21 for `x21`), and the third field gives the other source operand for the addition (20 for `x20`). The fifth field contains the number of the register that

is to receive the sum (9 for `x9`). Thus, this instruction adds register `x20` to register `x21` and places the sum in register `x9`.

This instruction can also be represented as fields of binary numbers instead of decimal:

0000000	10101	10100	000	01001	0110011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

This layout of the instruction is called the **instruction format**. As you can see from counting the number of bits, this RISC-V instruction takes exactly 32 bits—a word, or one half of a doubleword. In keeping with our design principle that simplicity favors regularity, RISC-V instructions are all 32 bits long.

instruction format

A form of representation of an instruction composed of fields of binary numbers.

To distinguish it from assembly language, we call the numeric version of instructions **machine language** and a sequence of such instructions *machine code*.

machine language

Binary representation used for communication within a computer system.

It would appear that you would now be reading and writing long, tiresome strings of binary numbers. We avoid that tedium by using a higher base than binary that converts easily into binary. Since almost all computer data sizes are multiples of 4, **hexadecimal** (base 16) numbers are popular. As base 16 is a power of 2, we can trivially convert by replacing each group of four binary digits by a single hexadecimal digit, and vice versa. Figure 2.4 converts between hexadecimal and binary.

hexadecimal

Numbers in base 16.

Because we frequently deal with different number bases, to avoid confusion, we will subscript decimal numbers with *ten*, binary numbers with *two*, and hexadecimal numbers with *hex*. (If there is no subscript, the default is base 10.) By the way, C and Java use the notation 0x*nnnn* for hexadecimal numbers.

Binary to Hexadecimal and Back

Example

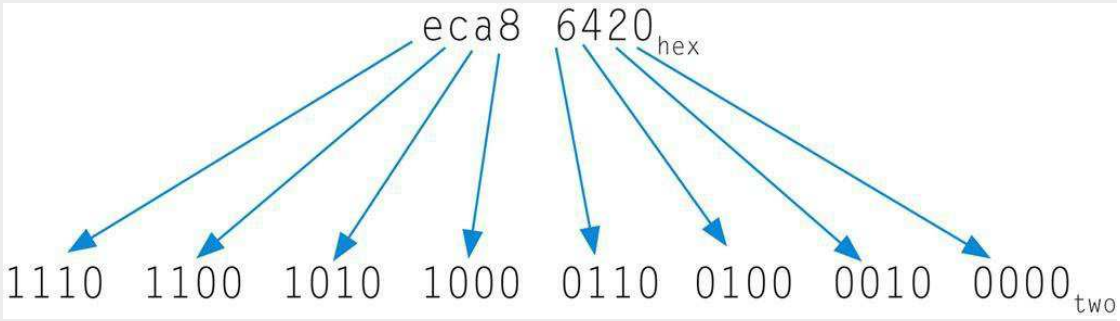
Convert the following 8-digit hexadecimal and 32-bit binary numbers into the other base:

$$\texttt{eca8 6420}_{hex}$$

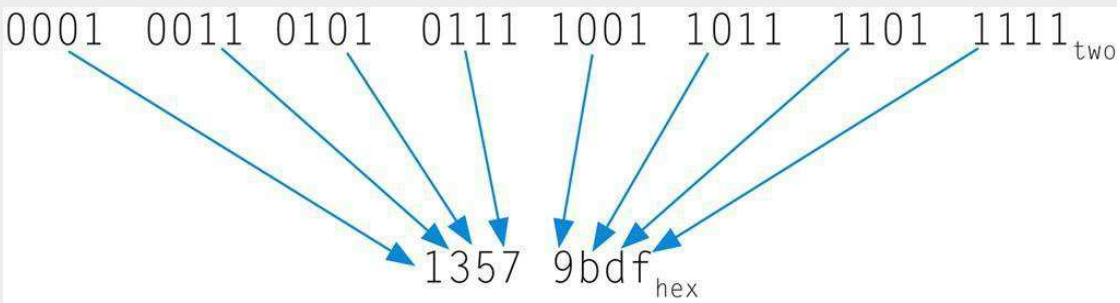
$$\texttt{0001 0011 0101 0111 1001 1011 1101 1111}_{two}$$

Answer

Using Figure 2.4, the answer is just a table lookup one way:



And then the other direction:



RISC-V Fields

RISC-V fields are given names to make them easier to discuss:

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Here is the meaning of each name of the fields in RISC-V instructions:

- *opcode:* Basic operation of the instruction, and this abbreviation is its traditional name.
- *rd:* The register destination operand. It gets the result of the operation.
- *funct3:* An additional opcode field.
- *rs1:* The first register source operand.
- *rs2:* The second register source operand.
- *funct7:* An additional opcode field.

opcode

The field that denotes the operation and format of an instruction.

A problem occurs when an instruction needs longer fields than those shown above. For example, the load register instruction must specify two registers and a constant. If the address were to use one of the 5-bit fields in the format above, the largest constant within the load register instruction would be limited to only 2^5-1 or 31. This constant is used to select elements from arrays or data structures, and it often needs to be much larger than 31. This 5-bit field is too small to be useful.

Hence, we have a conflict between the desire to keep all instructions the same length and the desire to have a single instruction format. This conflict leads us to the final hardware design principle:

Design Principle 3: Good design demands good compromises.

The compromise chosen by the RISC-V designers is to keep all instructions the same length, thereby requiring distinct instruction formats for different kinds of instructions. For example, the format above is called *R-type* (for register). A second type of instruction format is *I-type* and is used by arithmetic operands with one constant operand, including `addi`, and by load instructions. The fields of the I-type format are

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

The 12-bit immediate is interpreted as a two's complement value, so it can represent integers from -2^{11} to $2^{11}-1$. When the I-type format is used for load instructions, the immediate represents a byte offset, so the load doubleword instruction can refer to any doubleword within a region of $\pm 2^{11}$ or 2048 bytes ($\pm 2^8$ or 256 doublewords) of the base address in the base register rd. We see that more than 32 registers would be difficult in this format, as the rd and rs1 fields would each need another bit, making it harder to fit everything in one word.

Let's look at the load register instruction from page 71:

```
ld x9, 64(x22) // Temporary reg x9 gets A[8]
```

Here, 22 (for `x22`) is placed in the rs1 field, 64 is placed in the immediate field, and 9 (for `x9`) is placed in the rd field. We also need a format for the store doubleword instruction, `sd`, which needs two source registers (for the base address and the store data) and an immediate for the address offset. The fields of the S-type format are

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

The 12-bit immediate in the S-type format is split into two fields, which supply the lower 5 bits and upper 7 bits. The RISC-V architects chose this design because it keeps the rs1 and rs2 fields in the same place in all instruction formats. Keeping the instruction formats as similar as possible reduces hardware complexity.

Similarly, the opcode and funct3 fields are the same size in all locations, and they are always in the same place.

In case you were wondering, the formats are distinguished by the values in the opcode field: each format is assigned a distinct set of opcode values in the first field (opcode) so that the hardware knows how to treat the rest of the instruction. Figure 2.5 shows the numbers used in each field for the RISC-V instructions covered so far.

Translating RISC-V Assembly Language into Machine Language

Example

We can now take an example all the way from what the programmer writes to what the computer executes. If `x10` has the base of the array `A` and `x21` corresponds to `h`, the assignment statement

```
A[30] = h + A[30] + 1;
```

is compiled into

```
ld x9, 240(x10) // Temporary reg x9 gets A[30]
add x9, x21, x9 // Temporary reg x9 gets h+A[30]
addi x9, x9, 1 // Temporary reg x9 gets h+A[30]+1
sd x9, 240(x10) // Stores h+A[30]+1 back into A[30]
```

What is the RISC-V machine language code for these three instructions?

Answer

For convenience, let's first represent the machine language instructions using decimal numbers. From Figure 2.5, we can determine the three machine language instructions:

immediate	rs1	funct3	rd	opcode
240	10	3	9	3

funct7	rs2	rs1	funct3	rd	opcode
0	9	21	0	9	51

immediate	rs1	funct3	rd	opcode
1	9	0	9	19

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7	9	10	3	16	35

The `ld` instruction is identified by 3 (see Figure 2.5) in the opcode field and 3 in the funct3 field. The base register 10 is specified in the rs1 field, and the destination register 9 is specified in the rd field. The offset to select `A[30]` (240=30×8) is found in the immediate field.

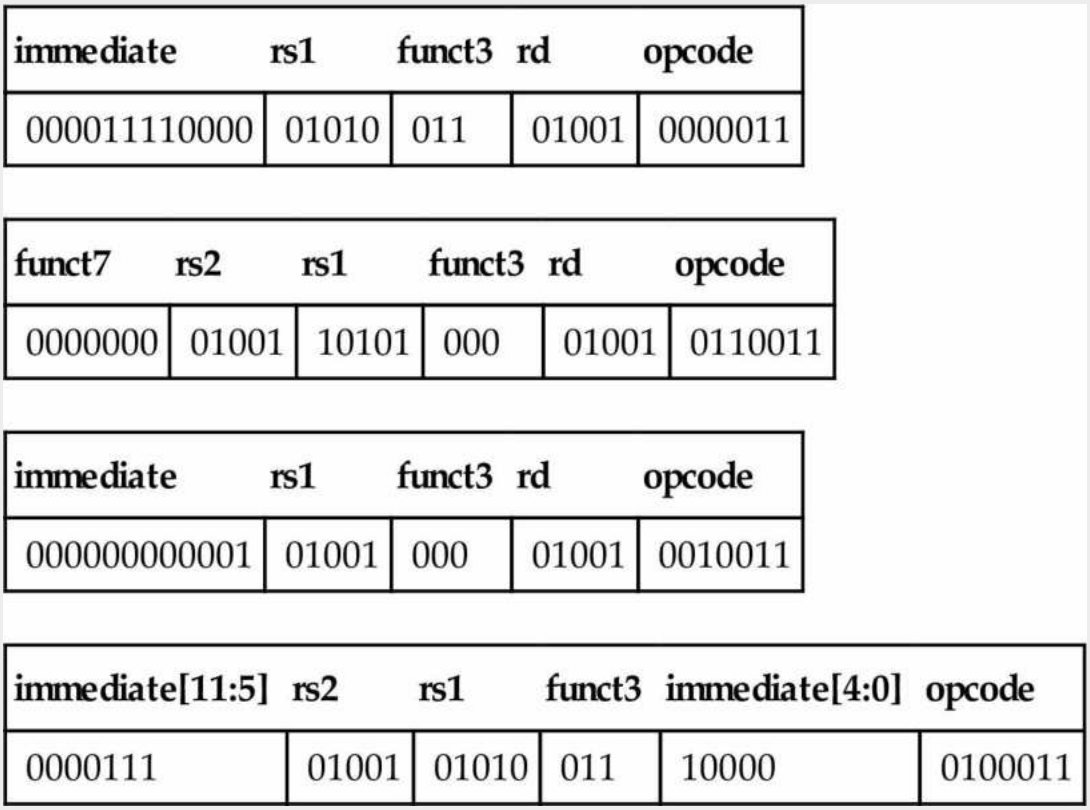
The `add` instruction that follows is specified with 51 in the opcode field, 0 in the funct3 field, and 0 in the funct7 field. The three register operands (9, 21, and 9) are found in the rd, rs1, and rs2 fields.

The subsequent `addi` instruction is specified with 19 in the opcode field and 0 in the funct3 field. The register operands (9 and 9) are found in the rd and rs1 fields, and the constant addend 1 is found in the immediate field.

The `sd` instruction is identified with 35 in the opcode field and 3 in the funct3 field. The register operands (9 and 10) are found in the rs2 and rs1 fields, respectively. The address offset 240 is split across the two immediate fields. Since the upper part of the immediate holds bits 5 and above, we can decompose the offset 240 by dividing by 2^5. The upper part of the immediate holds the

quotient, 7, and the lower part holds the remainder, 16.

Since 240_{ten} = 0000 1111 0000_{two}, the binary equivalent to the decimal form is:



immediate	rs1	funct3	rd	opcode
000011110000	01010	011	01001	0000011

funct7	rs2	rs1	funct3	rd	opcode
0000000	01001	10101	000	01001	0110011

immediate	rs1	funct3	rd	opcode
000000000001	01001	000	01001	0010011

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
0000111	01001	01010	011	10000	0100011

Elaboration

RISC-V assembly language programmers aren't forced to use `addi` when working with constants. The programmer simply writes `add`, and the assembler generates the proper opcode and the proper instruction format depending on whether the operands are all registers (R-type) or if one is a constant (I-type). We use the explicit names in RISC-V for the different opcodes and formats as we think it is less confusing when introducing assembly language versus machine language.

Elaboration

Although RISC-V has both `add` and `sub` instructions, it does not have a `subi` counterpart to `addi`. This is because the immediate field represents a two's complement integer, so `addi` can be used to subtract constants.

Hardware/Software Interface

The desire to keep all instructions the same size conflicts with the desire to have as many registers as possible. Any increase in the number of registers uses up at least one more bit in every register field of the instruction format. Given these constraints and the design principle that smaller is faster, most instruction sets today have 16 or 32 general-purpose registers.

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011
Instruction	**Format**	**immediate**		**rs1**	**funct3**	**rd**	**opcode**
addi (add immediate)	I	constant		reg	000	reg	0010011
ld (load doubleword)	I	address		reg	011	reg	0000011
Instruction	**Format**	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**
sd (store doubleword)	S	address	reg	reg	011	address	0100011

FIGURE 2.5 RISC-V instruction encoding.
In the table above, "reg" means a register number between 0 and 31 and "address" means a 12-bit address or constant. The funct3 and funct7 fields act as additional opcode fields.

Figure 2.6 summarizes the portions of RISC-V machine language described in this section. As we shall see in Chapter 4, the similarity of the binary representations of related instructions simplifies hardware design. These similarities are another example of regularity in the RISC-V architecture.

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (add immediate)	001111101000		00010	000	00001	0010011	addi x1, x2, 1000
ld (load doubleword)	001111101000		00010	011	00001	0000011	ld x1, 1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sd (store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1, 1000(x2)

FIGURE 2.6 RISC-V architecture revealed through Section 2.5.
The three RISC-V instruction formats so far are R, I, and S. The R-type format has two source register operand and one destination register operand. The I-type format replaces one source register operand with a 12-bit *immediate* field. The S-type format has two source operands and a 12-bit immediate field, but no destination register operand. The S-type immediate field is split into two parts, with bits 11—5 in the leftmost field and bits 4—0 in the second-rightmost field.

The BIG Picture

Today's computers are built on two key principles:

1. Instructions are represented as numbers.
2. Programs are stored in memory to be read or written, just like data.

These principles lead to the *stored-program* concept; its invention let the computing genie out of its bottle. Figure 2.7 shows the power of the concept; specifically, memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code.

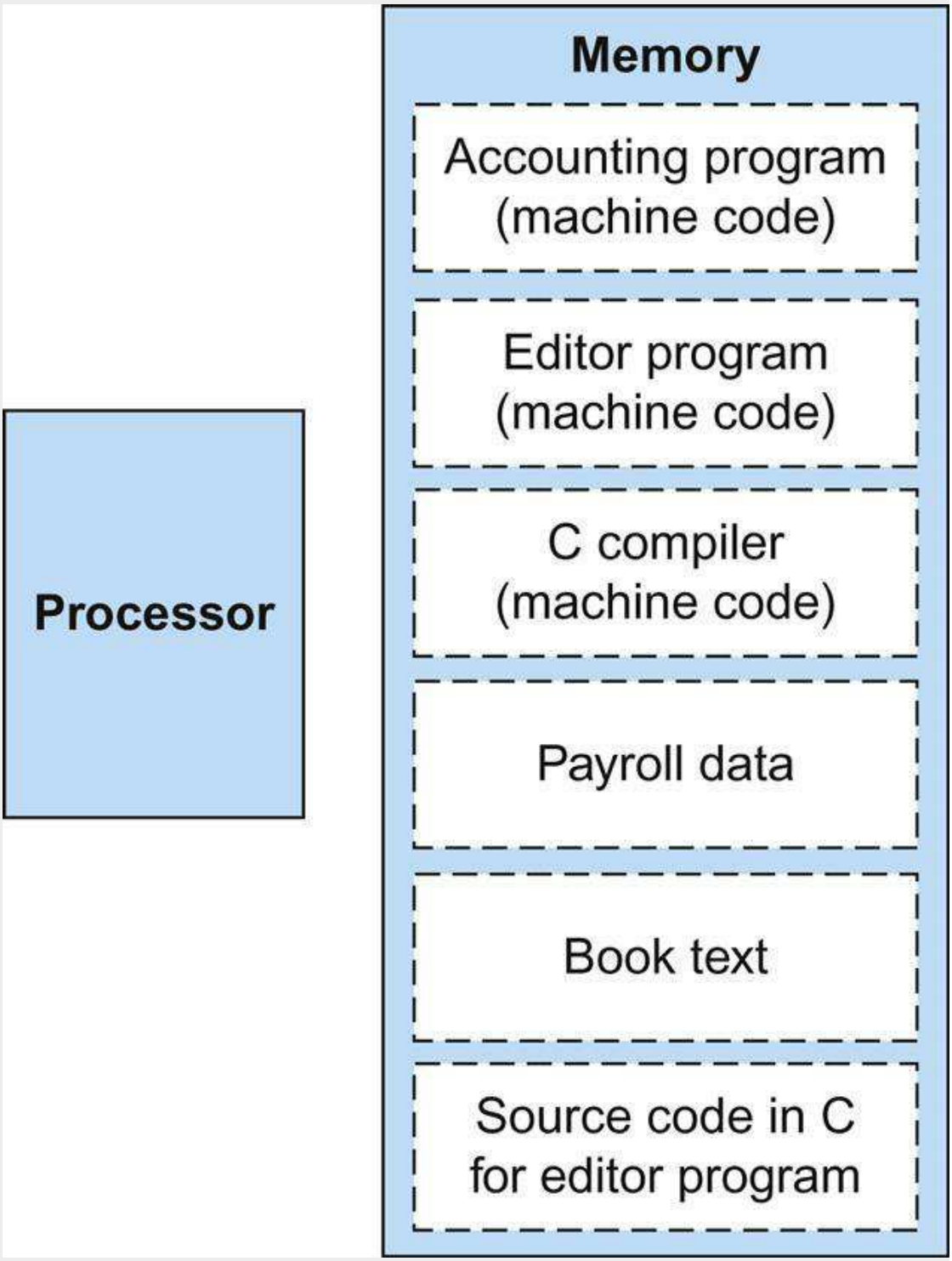


FIGURE 2.7 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed

for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.

One consequence of instructions as numbers is that programs are often shipped as files of binary numbers. The commercial implication is that computers can inherit ready-made software provided they are compatible with an existing instruction set. Such "binary compatibility" often leads industry to align around a small number of instruction set architectures.

Check Yourself

What RISC-V instruction does this represent? Choose from one of the four options below.

funct7	rs2	rs1	funct3	rd	opcode
32	9	10	000	11	51

1. `subx9,x10,x11`
2. `addx11,x9,x10`
3. `subx11,x10,x9`
4. `subx11,x9,x10`

2.6 Logical Operations

Although the first computers operated on full words, it soon became clear that it was useful to operate on fields of bits within a word or even on individual bits. Examining characters within a word, each of which is stored as 8 bits, is one example of such an operation (see Section 2.9). It follows that operations were added to programming languages and instruction set architectures to simplify, among other things, the packing and unpacking of bits into words. These instructions are called *logical operations*. Figure 2.8 shows logical operations in C, Java, and RISC-V.

"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

Lewis Carroll, Alice's Adventures in Wonderland, 1865

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

FIGURE 2.8 C and Java logical operators and their corresponding RISC-V instructions.
One way to implement NOT is to use XOR with one operand being all ones (FFFF FFFF FFFF $FFFF_{hex}$).

The first class of such operations is called *shifts*. They move all the bits in a doubleword to the left or right, filling the emptied bits with 0s. For example, if register `x19` contained

`00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001001`$_{two}$ = 9_{ten}

and the instruction to shift left by 4 was executed, the new value would be:

`00000000 00000000 00000000 00000000 00000000 00000000 00000000 10010000`$_{two}$ = 144_{ten}

The dual of a shift left is a shift right. The actual names of the two RISC-V shift instructions are *shift left logical immediate* (`slli`) and *shift right logical immediate* (`srli`). The following instruction performs the operation above, if the original value was in register `x19` and the result should go in register `x11`:

```
slli x11, x19, 4 // reg x11 = reg x19 << 4 bits
```

These shift instructions use the I-type format. Since it isn't useful to shift a 64-bit register by more than 63 bits, only the lower 6 bits of the I-type format's 12-bit immediate are actually used. The remaining 6 bits are repurposed as an additional opcode field, funct6.

funct6	immediate	rs1	funct3	rd	opcode
0	4	19	1	11	19

The encoding of `slli` is 19 in the opcode field, rd contains 11, funct3 contains 1, rs1 contains 19, immediate contains 4, and funct6 contains 0.

Shift left logical provides a bonus benefit. Shifting left by *i* bits gives the identical result as multiplying by 2^i, just as shifting a decimal number by *i* digits is equivalent to multiplying by 10*i*. For example, the above `slli` shifts by 4, which gives the same result as multiplying by 2^4 or 16. The first bit pattern above represents 9, and 9×16=144, the value of the second bit pattern. RISC-V provides a third type of shift, *shift right arithmetic* (`srai`). This variant is similar to `srli`, except rather than filling the vacated bits on the left with zeros, it fills them with copies of the old sign bit. It also provides variants of all three shifts that take the shift amount from a register, rather than from an immediate: `sll`, `srl`, and `sra`.

Another useful operation that isolates fields is **AND**. (We capitalize the word to avoid confusion between the operation and the English conjunction.) AND is a bit-by-bit operation that leaves a 1 in the result only if both bits of the operands are 1. For example, if register `x11` contains

AND

A logical bit-by-bit operation with two operands that calculates a 1 only if there is a 1 in *both* operands.

```
  00000000 00000000 00000000 00000000 00000000 00000000
00001101 11000000two
```

and register `x10` contains

```
  00000000 00000000 00000000 00000000 00000000 00000000
00111100 00000000two
```

then, after executing the RISC-V instruction

```
and x9, x10, x11  // reg x9 = reg x10 & reg x11
```

the value of register `x9` would be

```
00000000 00000000 00000000 00000000 00000000 00000000
```

$00001100\ 00000000_{two}$

As you can see, AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a *mask*, since the mask "conceals" some bits.

To place a value into one of these seas of 0s, there is the dual to AND, called **OR**. It is a bit-by-bit operation that places a 1 in the result if *either* operand bit is a 1. To elaborate, if the registers `x10` and `x11` are unchanged from the preceding example, the result of the RISC-V instruction

OR

A logical bit-by-bit operation with two operands that calculates a 1 if there is a 1 in *either* operand.

```
or x9, x10, x11 // reg x9 = reg x10 | reg x11
```

is this value in register `x9`:

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00111101\ 11000000_{two}$

The final logical operation is a contrarian. **NOT** takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. Using our prior notation, it calculates $\bar{x}$.

NOT

A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

In keeping with the three-operand format, the designers of RISC-V decided to include the instruction **XOR** (exclusive OR) instead of NOT. Since exclusive OR creates a 0 when bits are the same and a 1 if they are different, the equivalent to NOT is an `xor` 111…111.

XOR

A logical bit-by-bit operation with two operands that calculates the exclusive OR of the two operands. That is, it calculates a 1 only if the values are different in the two operands.

If the register `x10` is unchanged from the preceding example and

register `x12` has the value 0, the result of the RISC-V instruction

```
xor x9, x10, x12 // reg x9 = reg x10 ^ reg x12
```

is this value in register `x9`:

```
00000000 00000000 00000000 00000000 00000000 00000000
00110001 11000000two
```

Figure 2.8 above shows the relationship between the C and Java operators and the RISC-V instructions. Constants are useful in logical operations as well as in arithmetic operations, so RISC-V also provides the instructions *and immediate* (`andi`), *or immediate* (`ori`), and *exclusive or immediate* (`xori`).

Elaboration

C allows *bit fields* or *fields* to be defined within doublewords, both allowing objects to be packed within a doubleword and to match an externally enforced interface such as an I/O device. All fields must fit within a single doubleword. Fields are unsigned integers that can be as short as 1 bit. C compilers insert and extract fields using logical instructions in RISC-V: `andi`, `ori`, `slli`, and `srli`.

Check Yourself

Which operations can isolate a field in a doubleword?
1. AND
2. A shift left followed by a shift right

> *The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent upon the results of the computation.… This choice can be made to depend upon the sign of a number (zero being reckoned as plus for machine purposes). Consequently, we introduce an [instruction] (the conditional transfer [instruction]) which will, depending on the sign of a given number, cause the proper one of two routines to be executed.*
>
> *Burks, Goldstine, and von Neumann, 1947*

2.7 Instructions for Making Decisions

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels. RISC-V assembly language includes two decision-making instructions, similar to an *if* statement with a *go to*. The first instruction is

```
beq rs1, rs2, L1
```

This instruction means go to the statement labeled `L1` if the value in register `rs1` equals the value in register rs2. The mnemonic `beq` stands for *branch if equal*. The second instruction is

```
bne rs1, rs2, L1
```

It means go to the statement labeled `L1` if the value in register rs1 does *not* equal the value in register rs2. The mnemonic `bne` stands for *branch if not equal*. These two instructions are traditionally called **conditional branches**.

conditional branch

An instruction that tests a value and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the test.

Compiling *if-then-else* into Conditional Branches

Example

In the following code segment, `f`, `g`, `h`, `i`, and `j` are variables. If the five variables `f` through `j` correspond to the five registers `x19` through `x23`, what is the compiled RISC-V code for this C *if* statement?

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

Answer

Figure 2.9 shows a flowchart of what the RISC-V code should do. The first expression compares for equality between two variables in registers. It would seem that we would want to branch if I and j are equal (`beq`). In general, the code will be more efficient if we test

for the opposite condition to branch over the code that branches if the values are *not* equal (`bne`). Here is the code:

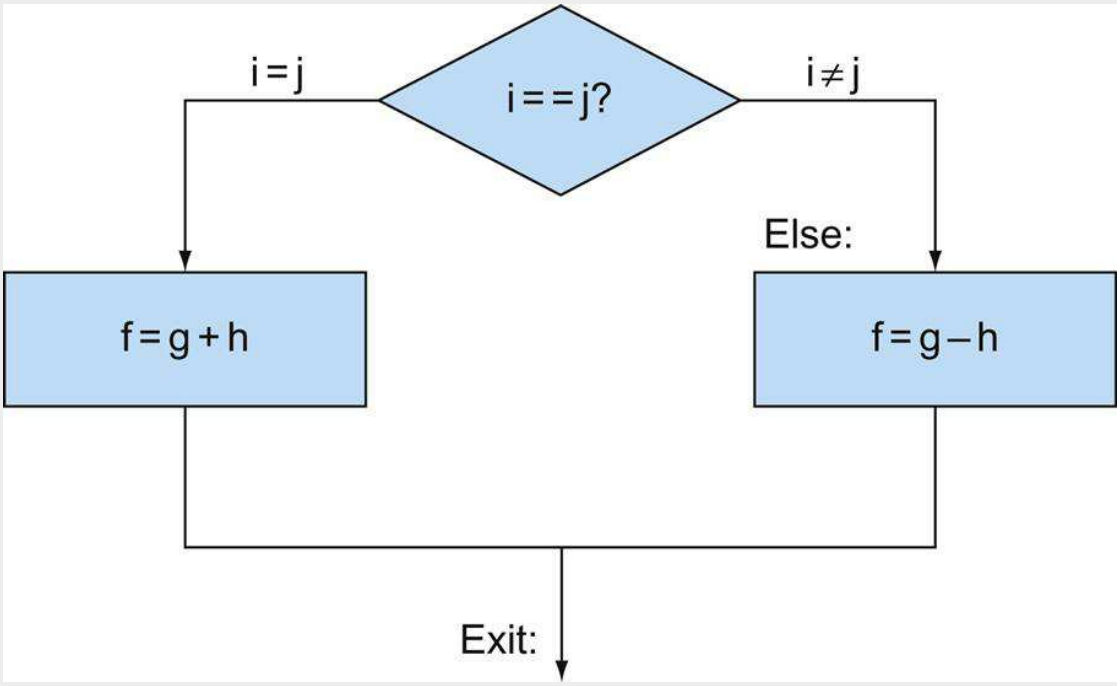


FIGURE 2.9 Illustration of the options in the *if* statement above.
The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

```
bne x22, x23, Else // go to Else if i ≠ j
```

The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

```
add x19, x20, x21  // f = g + h (skipped if i ≠ j)
```

We now need to go to the end of the *if* statement. This example introduces another kind of branch, often called an *unconditional branch*. This instruction says that the processor always follows the branch. One way to express an unconditional branch in RISC-V is to use a conditional branch whose condition is always true:

```
beq x0, x0, Exit  // if 0 == 0, go to Exit
```

The assignment statement in the *else* portion of the *if* statement can again be compiled into a single instruction. We just need to append the label `Else` to this instruction. We also show the label `Exit` that is after this instruction, showing the end of the *if-then-else*

compiled code:

```
Else:sub x19, x20, x21 // f = g −h (skipped if i = j)
Exit:
```

Notice that the assembler relieves the compiler and the assembly language programmer from the tedium of calculating addresses for branches, just as it does for calculating data addresses for loads and stores (see Section 2.12).

Hardware/Software Interface

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level.

Loops

Decisions are important both for choosing between two alternatives —found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.

Compiling a *while* Loop in C

Example

Here is a traditional loop in C:

```
while (save[i] == k)
  i += 1;
```

Assume that `i` and `k` correspond to registers `x22` and `x24` and the base of the array `save` is in `x25`. What is the RISC-V assembly code corresponding to this C code?

Answer

The first step is to load `save[i]` into a temporary register. Before we can load `save[i]` into a temporary register, we need to have its address. Before we can add `i` to the base of array `save` to form the address, we must multiply the index `i` by 8 due to the byte

addressing issue. Fortunately, we can use shift left, since shifting left by 3 bits multiplies by 2^3 or 8 (see page 90 in the prior section). We need to add the label `Loop` to it so that we can branch back to that instruction at the end of the loop:

```
Loop: slli x10, x22, 3  // Temp reg x10 = i * 8
```

To get the address of `save[i]`, we need to add `x10` and the base of save in `x25`:

```
add x10, x10, x25  // x10 = address of save[i]
```

Now we can use that address to load `save[i]` into a temporary register:

```
ld x9, 0(x10)  // Temp reg x9 = save[i]
```

The next instruction performs the loop test, exiting if `save[i]` ≠ `k`:

```
bne x9, x24, Exit  // go to Exit if save[i] ≠ k
```

The following instruction adds 1 to `i`:

```
addi x22, x22, 1  // i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
beq x0, x0, Loop  // go to Loop
Exit:
```

(See the exercises for an optimization of this sequence.)

Hardware/Software Interface

Such sequences of instructions that end in a branch are so fundamental to compiling that they are given their own buzzword: a **basic block** is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning. One of the first early phases of compilation is breaking the program into basic blocks.

basic block

A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

The test for equality or inequality is probably the most popular test, but there are many other relationships between two numbers.

For example, a *for* loop may want to test to see if the index variable is less than 0. The full set of comparisons is less than (<), less than or equal (≤), greater than (>), greater than or equal (≥), equal (=), and not equal (≠).

Comparison of bit patterns must also deal with the dichotomy between signed and unsigned numbers. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0. (We'll soon take advantage of this dual meaning of the most significant bit to reduce the cost of the array bounds checking.) RISC-V provides instructions that handle both cases. These instructions have the same form as `beq` and `bne`, but perform different comparisons. The branch if less than (`blt`) instruction compares the values in registers rs1 and rs2 and takes the branch if the value in rs1 is smaller, when they are treated as two's complement numbers. Branch if greater than or equal (`bge`) takes the branch in the opposite case, that is, if the value in rs1 is at least the value in rs2. Branch if less than, unsigned (`bltu`) takes the branch if the value in rs1 is smaller than the value in rs2 when the values are treated as unsigned numbers. Finally, branch if greater than or equal, unsigned (`bgeu`) takes the branch in the opposite case.

An alternative to providing these additional branch instructions is to set a register based upon the result of the comparison, then branch on the value in that temporary register with the `beq` or `bne` instructions. This approach, used by the MIPS instruction set, can make the processor datapath slightly simpler, but it takes more instructions to express a program.

Yet another alternative, used by ARM's instruction sets, is to keep extra bits that record what occurred during an instruction. These additional bits, called *condition codes* or *flags*, indicate, for example, if the result of an arithmetic operation was negative, or zero, or resulted in overflow.

Conditional branches then use combinations of these condition codes to perform the desired test.

One downside to condition codes is that if many instructions always set them, it will create dependencies that will make it difficult for pipelined execution (see Chapter 4).

Bounds Check Shortcut

Treating signed numbers as if they were unsigned gives us a low-cost way of checking if $0 \leq x < y$, which matches the index out-of-bounds check for arrays. The key is that negative integers in two's complement notation look like large numbers in unsigned notation; that is, the most significant bit is a sign bit in the former notation but a large part of the number in the latter. Thus, an unsigned comparison of $x<y$ checks if x is negative as well as if x is less than y.

Example

Use this shortcut to reduce an index-out-of-bounds check: branch to `IndexOutOfBounds` if `x20` ≥ `x11` or if `x20` is negative.

Answer

The checking code just uses unsigned greater than or equal to do both checks:

```
  bgeu x20, x11, IndexOutOfBounds // if x20 >= x11 or x20 < 0,
goto IndexOutOfBounds
```

Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **branch address table** or **branch table**, and the program needs only to index into the table and then branch to the appropriate sequence. The branch table is therefore just an array of double-words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the branch table into a register. It then needs to branch using the address in the register. To support such situations, computers like RISC-V include an *indirect jump* instruction, which performs an unconditional branch to the address specified in a register. In RISC-V, the jump-and-link register instruction (`jalr`) serves this purpose. We'll see an even

more popular use of this versatile instruction in the next section.

branch address table

Also called **branch table**. A table of addresses of alternative instruction sequences.

Hardware/Software Interface

Although there are many statements for decisions and loops in programming languages like C and Java, the bedrock statement that implements them at the instruction set level is the conditional branch.

Check Yourself

I. C has many statements for decisions and loops, while RISC-V has few. Which of the following does or does not explain this imbalance? Why?

1. More decision statements make code easier to read and understand.
2. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.
3. More decision statements mean fewer lines of code, which generally reduces coding time.
4. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.

II. Why does C provide two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||), while RISC-V doesn't?

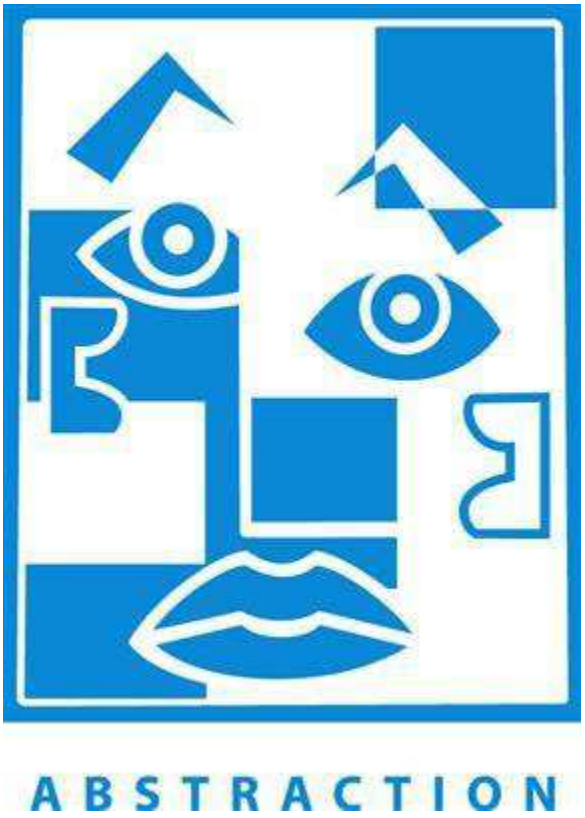
1. Logical operations AND and ORR implement & and |, while conditional branches implement && and ||.
2. The previous statement has it backwards: && and || correspond to logical operations, while & and | map to conditional branches.
3. They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.

2.8 Supporting Procedures in Computer Hardware

A **procedure** or function is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused. Procedures allow the programmer to concentrate on just one portion of the task at a time; parameters act as an interface between the procedure and the rest of the program and data, since they can pass values and return results. We describe the equivalent to procedures in Java in  **Section 2.15**, but Java needs everything from a computer that C needs. Procedures are one way to implement **abstraction** in software.

procedure

A stored subroutine that performs a specific task based on the parameters with which it is provided.



You can think of a procedure like a spy who leaves with a secret

plan, acquires resources, performs the task, covers his or her tracks, and then returns to the point of origin with the desired result. Nothing else should be perturbed once the mission is complete. Moreover, a spy operates on only a "need to know" basis, so the spy can't make assumptions about the spymaster.

Similarly, in the execution of a procedure, the program must follow these six steps:

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. RISC-V software follows the following convention for procedure calling in allocating its 32 registers:

- `x10-x17`: eight parameter registers in which to pass parameters or return values.
- `x1`: one return address register to return to the point of origin.

In addition to allocating these registers, RISC-V assembly language includes an instruction just for the procedures: it branches to an address and simultaneously saves the address of the following instruction to the destination register rd. The **jump-and-link instruction** (`jal`) is written

```
  jal x1, ProcedureAddress    // jump to ProcedureAddress and
write return address to x1
```

jump-and-link instruction

An instruction that branches to an address and simultaneously saves the address of the following instruction in a register (usually `x1` in RISC-V).

The *link* portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This "link," stored in register `x1`, is called the **return address**. The return address is needed because

the same procedure could be called from several parts of the program.

return address

A link to the calling site that allows a procedure to return to the proper address; in RISC-V it is stored in register `x1`.

To support the return from a procedure, computers like RISC-V use an indirect jump, like the jump-and-link instruction (`jalr`) introduced above to help with case statements:

```
jalr x0, 0(x1)
```

The jump-and-link register instruction branches to the address stored in register `x1`—which is just what we want. Thus, the calling program, or **caller**, puts the parameter values in `x10-x17` and uses `jal x1, X` to branch to procedure `X` (sometimes named the **callee**). The callee then performs the calculations, places the results in the same parameter registers, and returns control to the caller using `jalr x0, 0(x1)`.

caller

The program that instigates a procedure and provides the necessary parameter values.

callee

A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed. For historical reasons, this register is almost always called the **program counter**, abbreviated *PC* in the RISC-V architecture, although a more sensible name would have been *instruction address register*. The `jal` instruction actually saves PC+4 in its designation register (usually `x1`) to link to the byte address of the following instruction to set up the procedure return.

program counter (PC)

The register containing the address of the instruction in the program being executed.

Elaboration

The jump-and-link instruction can also be used to perform an unconditional branch within a procedure by using x0 as the destination register. Since x0 is hard-wired to zero, the effect is to discard the return address:

```
jal x0, Label // unconditionally branch to Label
```

Using More Registers

Suppose a compiler needs more registers for a procedure than the eight argument registers. Since we must cover our tracks after our mission is complete, any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked. This situation is an example in which we need to spill registers to memory, as mentioned in the *Hardware/Software Interface* section on page 69.

The ideal data structure for spilling registers is a **stack**—a last-in-first-out queue. A stack needs a pointer to the most recently allocated address in the stack to show where the next procedure should place the registers to be spilled or where old register values are found. In RISC-V, the **stack pointer** is register `x2`, also known by the name `sp`. The stack pointer is adjusted by one doubleword for each register that is saved or restored. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: placing data onto the stack is called a **push**, and removing data from the stack is called a **pop**.

stack

A data structure for spilling registers organized as a last-in-first-out queue.

stack pointer

A value denoting the most recently allocated address in a stack that

shows where registers should be spilled or where old register values can be found. In RISC-V, it is register `sp`, or `x2`.

push

Add element to stack.

pop

Remove element from stack.

By historical precedent, stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

Compiling a C Procedure That Doesn't Call Another Procedure

Example

Let's turn the example on page 66 from Section 2.2 into a C procedure:

```
long long int leaf_example (long long int g, long long int
h, long long int i, long long int j)
{
  long long int f;
  f = (g + h) -(i + j);
  return f;
}
```

What is the compiled RISC-V assembly code?

Answer

The parameter variables `g`, `h`, `i`, and `j` correspond to the argument registers `x10`, `x11`, `x12`, and `x13`, and `f` corresponds to `x20`. The compiled program starts with the label of the procedure:

```
leaf_example:
```

The next step is to save the registers used by the procedure. The C assignment statement in the procedure body is identical to the example on page 67, which uses two temporary registers (`x5` and `x6`). Thus, we need to save three registers: `x5`, `x6`, and `x20`. We

"push" the old values onto the stack by creating space for three doublewords (24 bytes) on the stack and then store them:

```
addi sp,  sp, -24  // adjust stack to make room for 3 items
sdx5, 16(sp)  // save register x5 for use afterwards
sdx6, 8(sp)   // save register x6 for use afterwards
sdx20, 0(sp)  // save register x20 for use afterwards
```

Figure 2.10 shows the stack before, during, and after the procedure call.

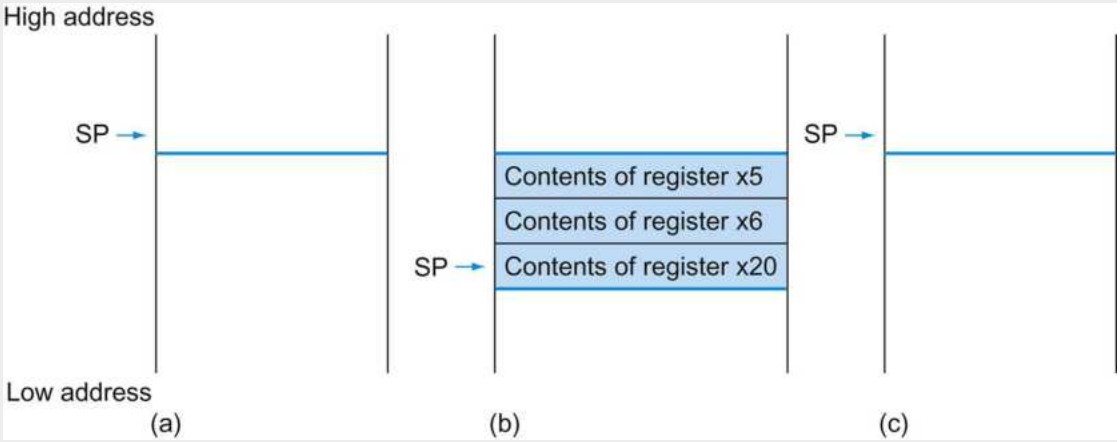


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call.
The stack pointer always points to the "top" of the stack, or the last doubleword in the stack in this drawing.

The next three statements correspond to the body of the procedure, which follows the example on page 67:

```
add x5, x10, x11// register x5 contains g + h
add x6, x12, x13// register x6 contains i + j
sub x20, x5, x6// f = x5 - x6, which is (g + h)−(i + j)
```

To return the value of `f`, we copy it into a parameter register:

```
addi x10, x20, 0 // returns f (x10 = x20 + 0)
```

Before returning, we restore the three old values of the registers we saved by "popping" them from the stack:

```
ld x20, 0(sp) // restore register x20 for caller
ld x6, 8(sp) // restore register x6 for caller
ld x5, 16(sp) // restore register x5 for caller
addi sp, sp, 24  // adjust stack to delete 3 items
```

The procedure ends with a branch register using the return

```
address:
  jalr x0, 0(x1)   // branch back to calling routine
```

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, RISC-V software separates 19 of the registers into two groups:

- `x5–x7` and `x28–x31`: temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- `x8–x9` and `x18–x27`: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

This simple convention reduces register spilling. In the example above, since the caller does not expect registers `x5` and `x6` to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore `x20`, since the callee must assume that the caller needs its value.

Nested Procedures

Procedures that do not call others are called *leaf* procedures. Life would be simple if all procedures were leaf procedures, but they aren't. Just as a spy might employ other spies as part of a mission, who in turn might use even more spies, so do procedures invoke other procedures. Moreover, recursive procedures even invoke "clones" of themselves. Just as we need to be careful when using registers in procedures, attention must be paid when invoking nonleaf procedures.

For example, suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register `x10` and then using `jal x1, A`. Then suppose that procedure A calls procedure B via `jal x1, B` with an argument of 7, also placed in `x10`. Since A hasn't finished its task yet, there is a conflict over the use of register `x10`. Similarly, there is a conflict over the return address in register `x1`, since it now has the return address for B. Unless we take steps to prevent the problem, this conflict will eliminate procedure A's ability to return to its caller.

One solution is to push all the other registers that must be preserved on the stack, just as we did with the saved registers. The

caller pushes any argument registers (x10–x17) or temporary registers (x5–x7 and x28–x31) that are needed after the call. The callee pushes the return address register x1 and any saved registers (x8–x9 and x18–x27) used by the callee. The stack pointer sp is adjusted to account for the number of registers placed on the stack. Upon the return, the registers are restored from memory, and the stack pointer is readjusted.

Compiling a Recursive C Procedure, Showing Nested Procedure Linking

Example

Let's tackle a recursive procedure that calculates factorial:

```
long long int fact (long long int n)
{
   if (n < 1) return (1);
      else return (n * fact(n −1));
}
```

What is the RISC-V assembly code?

Answer

The parameter variable n corresponds to the argument register x10. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and x10:

```
fact:
  addi  sp, sp, -16// adjust stack for 2 items
    sd x1, 8(sp)// save the return address
    sd x10, 0(sp)// save the argument n
```

The first time fact is called, sd saves an address in the program that called fact. The next two instructions test whether n is less than 1, going to L1 if n ≥ 1.

```
addi  x5, x10, -1  // x5 = n - 1
bge  x5, x0, L1  // if (n - 1) >= 0, go to L1
```

If n is less than 1, fact returns 1 by putting 1 into a value register: it adds 1 to 0 and places that sum in x10. It then pops the two saved values off the stack and branches to the return address:

```
addi  x10, x0, 1  // return 1
addi  sp, sp, 16  // pop 2 items off stack
jalr    // return to caller
```

Before popping two items off the stack, we could have loaded `x1` and `x10`. Since `x1` and `x10` don't change when `n` is less than 1, we skip those instructions.

If `n` is not less than 1, the argument `n` is decremented and then `fact` is called again with the decremented value:

```
L1: addi x10, x10, -1  // n >= 1: argument gets (n –1)
  jalx1, fact  // call fact with (n –1)
```

The next instruction is where `fact` returns; its result is in `x10`. Now the old return address and old argument are restored, along with the stack pointer:

```
addi  x6, x10, 0  // return from jal: move result of fact(n - 1) to x6:
ld  x10, 0(sp)  // restore argument n
ld  x1, 8(sp)  // restore the return address
addi  sp, sp, 16  // adjust stack pointer to pop 2 items
```

Next, argument register `x10` gets the product of the old argument and the result of fact(n - 1), now in `x6`. We assume a multiply instruction is available, even though it is not covered until Chapter 3:

```
mul  x10, x10, x6  // return n * fact (n –1)
```

Finally, `fact` branches again to the return address:

```
jalr  x0, 0(x1)  // return to the caller
```

Hardware/Software Interface

A C variable is generally a location in storage, and its interpretation depends both on its *type* and *storage class*. Example types include integers and characters (see Section 2.9). C has two storage classes: *automatic* and *static*. Automatic variables are local to a procedure and are discarded when the procedure exits. Static variables exist across exits from and entries to procedures. C variables declared outside all procedures are considered static, as are any variables declared using the keyword *static*. The rest are automatic. To simplify access to static data, some RISC-V compilers reserve a register `x3` for use as the **global pointer**, or `gp`.

global pointer

The register that is reserved to point to the static area.

Figure 2.11 summarizes what is preserved across a procedure call. Note that several schemes preserve the stack, guaranteeing that the caller will get the same data back on a load from the stack as it stored onto the stack. The stack above `sp` is preserved simply by making sure the callee does not write above `sp`; `sp` is itself preserved by the callee adding exactly the same amount that was subtracted from it; and the other registers are preserved by saving them on the stack (if they are used) and restoring them from there.

Preserved	Not preserved
Saved registers: `x8-x9, x18-x27`	Temporary registers: `x5-x7, x28-x31`
Stack pointer register: `x2(sp)`	Argument/result registers: `x10-x17`
Frame pointer: `x8(fp)`	
Return address: `x1(ra)`	
Stack above the stack pointer	Stack below the stack pointer

FIGURE 2.11 What is and what is not preserved across a procedure call.
If the software relies on the global pointer register, discussed in the following subsections, it is also preserved.

Allocating Space for New Data on the Stack

The final complexity is that the stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures. The segment of the stack containing a procedure's saved registers and local variables is called a **procedure frame** or **activation record**. Figure 2.12 shows the state of the stack before, during, and after the procedure call.

procedure frame
Also called **activation record**. The segment of the stack containing a procedure's saved registers and local variables.

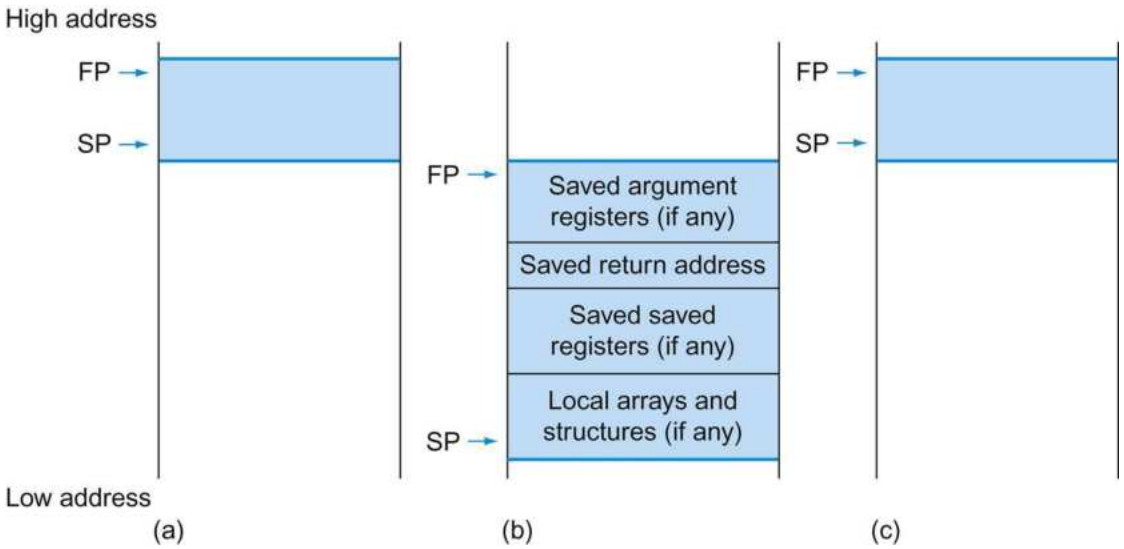


FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer (`fp` or `x8`) points to the first doubleword of the frame, often a saved argument register, and the stack pointer (`sp`) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it's easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by *not* setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in `sp` on a call, and `sp` is restored using `fp`. This information is also found in Column 4 of the RISC-V Reference Data Card at the front of this book.

Some RISC-V compilers use a **frame pointer** `fp`, or register `x8` to point to the first doubleword of the frame of a procedure. A stack pointer might change during the procedure, and so references to a local variable in memory might have different offsets depending on where they are in the procedure, making the procedure harder to understand. Alternatively, a frame pointer offers a stable base register within a procedure for local memory-references. Note that an activation record appears on the stack whether or not an explicit frame pointer is used. We've been avoiding using `fp` by avoiding changes to `sp` within a procedure: in our examples, the stack is

adjusted only on entry to and exit from the procedure.

frame pointer
A value denoting the location of the saved registers and local variables for a given procedure.

Allocating Space for New Data on the Heap

In addition to automatic variables that are local to procedures, C programmers need space in memory for static variables and for dynamic data structures. Figure 2.13 shows the RISC-V convention for allocation of memory when running the Linux operating system. The stack starts in the high end of the user addresses space (see Chapter 5) and grows down. The first part of the low end of memory is reserved, followed by the home of the RISC-V machine code, traditionally called the **text segment**. Above the code is the *static data segment*, which is the place for constants and other static variables. Although arrays tend to be a fixed length and thus are a good match to the static data segment, data structures like linked lists tend to grow and shrink during their lifetimes. The segment for such data structures is traditionally called the *heap*, and it is placed next in memory. Note that this allocation allows the stack and heap to grow toward each other, thereby allowing the efficient use of memory as the two segments wax and wane.

text segment
The segment of a UNIX object file that contains the machine language code for routines in the source file.

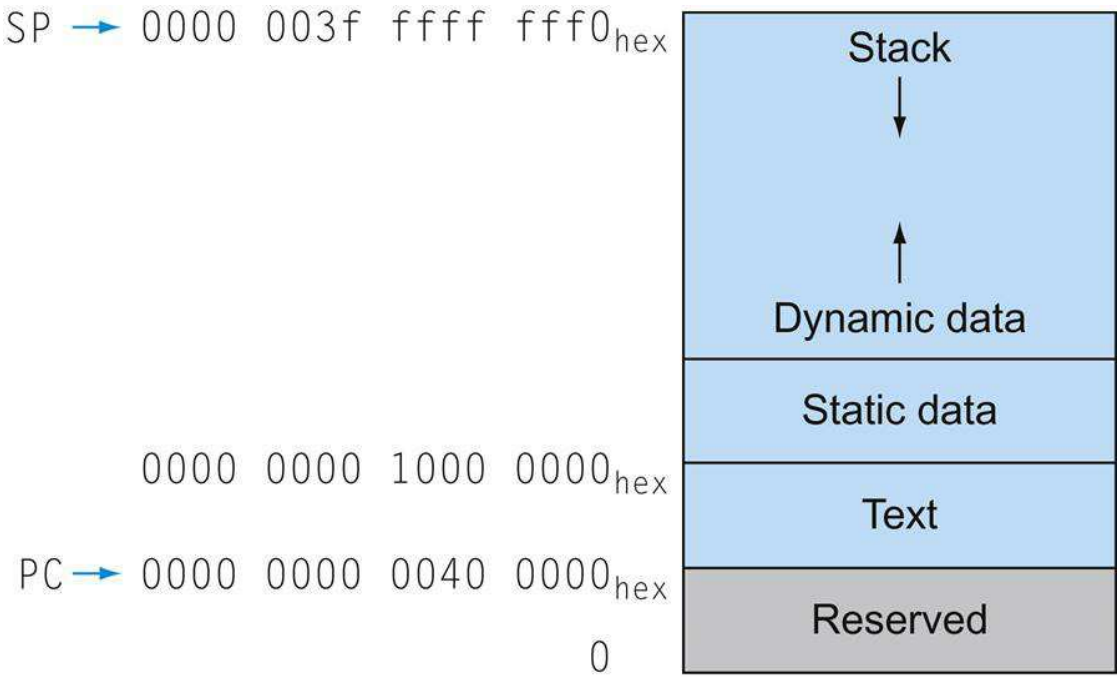


FIGURE 2.13 The RISC-V memory allocation for program and data.
These addresses are only a software convention, and not part of the RISC-V architecture. The user address space is set to 2^{38} of the potential 2^{64} total address space given a 64-bit architecture (see Chapter 5). The stack pointer is initialized to `0000 003f ffff fff0`$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at `0000 0000 0040 0000`$_{hex}$. The static data starts immediately after the end of the text segment; in this example, we assume that address is `0000 0000 1000 0000`$_{hex}$. Dynamic data, allocated by `malloc` in C and by `new` in Java, is next. It grows up toward the stack in an area called the *heap*. This information is also found in Column 4 of the RISC-V Reference Data Card at the front of this book.

C allocates and frees space on the heap with explicit functions. `malloc()` allocates space on the heap and returns a pointer to it, and `free()` releases space on the heap to which the pointer points. C

programs control memory allocation, which is the source of many common and difficult bugs. Forgetting to free space leads to a "memory leak," which ultimately uses up so much memory that the operating system may crash. Freeing space too early leads to "dangling pointers," which can cause pointers to point to things that the program never intended. Java uses automatic memory allocation and garbage collection just to avoid such bugs.

Figure 2.14 summarizes the register conventions for the RISC-V assembly language. This convention is another example of making the **common case fast**: most procedures can be satisfied with up to eight argument registers, twelve saved registers, and seven temporary registers without ever going to memory.

Elaboration

What if there are more than eight parameters? The RISC-V convention is to place the extra parameters on the stack just above the frame pointer. The procedure then expects the first eight parameters to be in registers `x10` through `x17` and the rest in memory, addressable via the frame pointer.

As mentioned in the caption of Figure 2.12, the frame pointer is convenient because all references to variables in the stack within a procedure will have the same offset. The frame pointer is not necessary, however. The RISC-V C compiler only uses a frame pointer in procedures that change the stack pointer in the body of the procedure.

Elaboration

Some recursive procedures can be implemented iteratively without using recursion. Iteration can significantly improve performance by removing the overhead associated with recursive procedure calls. For example, consider a procedure used to accumulate a sum:

```
long long int sum (long long int n, long long int acc) {
  if (n > 0)
    return sum(n - 1, acc + n);
  else
    return acc;
}
```

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

FIGURE 2.14 RISC-V register conventions.
This information is also found in Column 2 of the RISC-V Reference Data Card at the front of this book.

Consider the procedure call `sum(3,0)`. This will result in recursive calls to `sum(2,3)`, `sum(1,5)`, and `sum(0,6)`, and then the result 6 will be returned four times. This recursive call of sum is referred to as a *tail call*, and this example use of tail recursion can be implemented very efficiently (assume `x10 = n, x11 = acc`, and the result goes into `x12`):

```
sum: ble x10, x0, sum_exit  // go to sum_exit if n <= 0
 add x11, x11, x10  // add n to acc
 addi x10, x10, -1   // subtract 1 from n
 jal x0, sum    // jump to sum
```

```
sum_exit:
 addi x12, x11, 0   // return value acc
 jalr x0, 0(x1)   // return to caller
```

Check Yourself

Which of the following statements about C and Java is generally true?

1. C programmers manage data explicitly, while it's automatic in Java.
2. C leads to more pointer bugs and memory leak bugs than does Java.

!(@ |=>(wow open tab at bar is great)

Fourth line of the keyboard poem "Hatless Atlas," 1991 (some give names to ASCII characters: "!" is "wow," "(" is open, "|" is bar, and so on).

2.9 Communicating with People

Computers were invented to crunch numbers, but as soon as they became commercially viable they were used to process text. Most computers today offer 8-bit bytes to represent characters, with the *American Standard Code for Information Interchange* (ASCII) being the representation that nearly everyone follows. Figure 2.15 summarizes ASCII.

ASCII versus Binary Numbers

Example

We could represent numbers as strings of ASCII digits instead of as integers. How much does storage increase if the number 1 billion is represented in ASCII versus a 32-bit integer?

Answer

One billion is 1,000,000,000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be (10×8)/32 or 2.5. Beyond the expansion in storage, the hardware to add, subtract, multiply, and divide such decimal numbers is difficult and would

consume more energy. Such difficulties explain why computing professionals are raised to believe that binary is natural and that the occasional decimal computer is bizarre.

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string.

A series of instructions can extract a byte from a doubleword, so load register and store register are sufficient for transferring bytes as well as words. Because of the popularity of text in some programs, however, RISC-V provides instructions to move bytes. *Load byte* unsigned (`lbu`) loads a byte from memory, placing it in the rightmost 8 bits of a register. *Store byte* (`sb`) takes a byte from the rightmost 8 bits of a register and writes it to memory. Thus, we copy a byte with the sequence

```
lbu x12, 0(x10) // Read byte from source
sbx12, 0(x11) // Write byte to destination
```

Characters are normally combined into strings, which have a variable number of characters. There are three choices for representing a string: (1) the first position of the string is reserved to give the length of a string, (2) an accompanying variable has the

length of the string (as in a structure), or (3) the last position of a string is indicated by a character used to mark the end of a string. C uses the third choice, terminating a string with a byte whose value is 0 (named null in ASCII). Thus, the string "Cal" is represented in C by the following 4 bytes, shown as decimal numbers: 67, 97, 108, and 0. (As we shall see, Java uses the first option.)

Compiling a String Copy Procedure, Showing How to Use C Strings

Example

The procedure `strcpy` copies string `y` to string `x` using the null byte termination convention of C:

```
void strcpy (char x[], char y[])
{
 size_t i;
 i = 0;
 while ((x[i] = y[i]) != '\0') /* copy & test byte */
 i += 1;
}
```

What is the RISC-V assembly code?

Answer

Below is the basic RISC-V assembly code segment. Assume that base addresses for arrays `x` and `y` are found in `x10` and `x11`, while `i` is in `x19`. `strcpy` adjusts the stack pointer and then saves the saved register `x19` on the stack:

```
strcpy:
 addi sp, sp, -8 // adjust stack for 1 more item
 sd x19, 0(sp) // save x19
```

To initialize `i` to 0, the next instruction sets `x19` to 0 by adding 0 to 0 and placing that sum in `x19`:

```
add x19, x0, x0// i = 0+0
```

This is the beginning of the loop. The address of `y[i]` is first formed by adding `i` to `y[]`:

```
L1: add x5, x19, x11 // address of y[i] in x5
```

Note that we don't have to multiply `i` by 8 since `y` is an array of *bytes* and not of doublewords, as in prior examples.

To load the character in `y[i]`, we use load byte unsigned, which

puts the character into x6:

```
lbu x6, 0(x5) // x6 = y[i]
```

A similar address calculation puts the address of x[i] in x7, and then the character in x6 is stored at that address.

```
add x7, x19, x10 // address of x[i] in x7
sb x6, 0(x7) // x[i] = y[i]
```

Next, we exit the loop if the character was 0. That is, we exit if it is the last character of the string:

```
beq x6, x0, L2
```

If not, we increment i and loop back:

```
addi x19, x19, 1 // i = i + 1
jal x0, L1 // go to L1
```

If we don't loop back, it was the last character of the string; we restore x19 and the stack pointer, and then return.

```
L2: ld x19, 0(sp) // restore old x19
  addi sp, sp, 8 // pop 1 doubleword off stack
  jalr x0, 0(x1) // return
```

String copies usually use pointers instead of arrays in C to avoid the operations on i in the code above. See Section 2.14 for an explanation of arrays versus pointers.

Since the procedure strcpy above is a leaf procedure, the compiler could allocate i to a temporary register and avoid saving and restoring x19. Hence, instead of thinking of these registers as being just for temporaries, we can think of them as registers that the callee should use whenever convenient. When a compiler finds a leaf procedure, it exhausts all temporary registers before using registers it must save.

Characters and Strings in Java

Unicode is a universal encoding of the alphabets of most human languages. Figure 2.16 gives a list of Unicode alphabets; there are almost as many *alphabets* in Unicode as there are useful *symbols* in ASCII. To be more inclusive, Java uses Unicode for characters. By default, it uses 16 bits to represent a character.

Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes

FIGURE 2.16 Example alphabets in Unicode. Unicode version 4.0 has more than 160 "blocks," which is their name for a collection of symbols. Each block is a multiple of 16. For example, Greek starts at 0370_{hex}, and Cyrillic at 0400_{hex}. The first three columns show 48 blocks that correspond to human languages in roughly Unicode numerical order. The last column has 16 blocks that are multilingual and are not in order. A 16-bit encoding, called UTF-16, is the default. A variable-length encoding, called UTF-8, keeps the ASCII subset as eight bits and uses 16 or 32 bits for the other characters. UTF-32 uses 32 bits per character. To learn more, see www.unicode.org.

The RISC-V instruction set has explicit instructions to load and store such 16-bit quantities, called *halfwords*. *Load half unsigned* loads a halfword from memory, placing it in the rightmost 16 bits of a register, filling the leftmost 48 bits with zeros. Like load byte, *load half* (`lh`) treats the halfword as a signed number and thus sign-extends to fill the 48 leftmost bits of the register. *Store half* (`sh`) takes a halfword from the rightmost 16 bits of a register and writes it to memory. We copy a halfword with the sequence

```
lhu x19, 0(x10) // Read halfword (16 bits) from source
shx19, 0(x11) // Write halfword (16 bits) to dest
```

Strings are a standard Java class with special built-in support and predefined methods for concatenation, comparison, and conversion. Unlike C, Java includes a word that gives the length of the string, similar to Java arrays.

Elaboration

RISC-V software is required to keep the stack aligned to "quadword" (16 byte) addresses to get better performance. This convention means that a `char` variable allocated on the stack may occupy as much as 16 bytes, even though it needs less. However, a C string variable or an array of bytes *will* pack 16 bytes per quadword, and a Java string variable or array of shorts packs 8 halfwords per quadword.

Elaboration

Reflecting the international nature of the web, most web pages today use Unicode instead of ASCII. Hence, Unicode may be even more popular than ASCII today.

Elaboration

RISC-V also includes instructions to move 32-bit values to and from memory. *Load word unsigned* (`lwu`) loads a 32-bit word from memory into the rightmost 32 bits of a register, filling the leftmost 32 bits with zeros. *Load word* (`lw`) instead fills the leftmost 32 bits with copies of bit 31. *Store word* (`sw`) takes a word from the rightmost 32 bits of a register and stores it to memory.

Check Yourself

I. Which of the following statements about characters and strings in C and Java is true?

1. A string in C takes about half the memory as the same string in Java.
2. Strings are just an informal name for single-dimension arrays of characters in C and Java.
3. Strings in C and Java use null (0) to mark the end of a string.
4. Operations on strings, like length, are faster in C than in Java.

II. Which type of variable that can contain $1{,}000{,}000{,}000_{ten}$ takes the most memory space?

1. `long long int` in C
2. `string` in C
3. `string` in Java

2.10 RISC-V Addressing for Wide Immediates and Addresses

Although keeping all RISC-V instructions 32 bits long simplifies the hardware, there are times where it would be convenient to have 32-bit or larger constants or addresses. This section starts with the general solution for large constants, and then shows the optimizations for instruction addresses used in branches.

Wide Immediate Operands

Although constants are frequently short and fit into the 12-bit fields, sometimes they are bigger.

The RISC-V instruction set includes the instruction *Load upper immediate* (`lui`) to load a 20-bit constant into bits 12 through 31 of a register. The leftmost 32 bits are filled with copies of bit 31, and the rightmost 12 bits are filled with zeros. This instruction allows, for example, a 32-bit constant to be created with two instructions. `lui` uses a new instruction format, U-type, as the other formats cannot accommodate such a large constant.

Loading a 32-Bit Constant

Example

What is the RISC-V assembly code to load this 64-bit constant into register `x19`?

```
  00000000 00000000 00000000 00000000 00000000 00111101
00000101 00000000
```

Answer

First, we would load bits 12 through 31 with that bit pattern, which is 976 in decimal, using `lui`:

```
  lui  x19, 976 // 976decimal = 0000 0000 0011 1101 0000
```

The value of register `x19` afterward is:

```
  0000000 00000000 00000000 00000000 00000000 00111101
00000000 00000000
```

The next step is to add in the lowest 12 bits, whose decimal value is 1280:

```
  addi  x19, x19, 1280// 1280decimal = 00000101 00000000
```

The final value in register `x19` is the desired value:

```
00000000 00000000 00000000 00000000 00000000 00111101
00000101 00000000
```

Elaboration

In the previous example, bit 11 of the constant was 0. If bit 11 had been set, there would have been an additional complication: the 12-bit immediate is sign-extended, so the addend would have been negative. This means that in addition to adding in the rightmost 11 bits of the constant, we would have also subtracted 2^{12}. To compensate for this error, it suffices to add 1 to the constant loaded with lui, since the lui constant is scaled by 2^{12}.

Hardware/Software Interface

Either the compiler or the assembler must break large constants into pieces and then reassemble them into a register. As you might expect, the immediate field's size restriction may be a problem for memory addresses in loads and stores as well as for constants in immediate instructions.

Hence, the symbolic representation of the RISC-V machine language is no longer limited by the hardware, but by whatever the creator of an assembler chooses to include (see Section 2.12). We stick close to the hardware to explain the architecture of the computer, noting when we use the enhanced language of the assembler that is not found in the processor.

Addressing in Branches

The RISC-V branch instructions use the RISC-V instruction format called SB-type. This format can represent branch addresses from −4096 to 4094, in multiples of 2. For reasons revealed shortly, it is only possible to branch to even addresses. The SB-type format consists of a 7-bit opcode, a 3-bit function code, two 5-bit register operands (rs1 and rs2), and a 12-bit address immediate. The address uses an unusual encoding, which simplifies datapath design but complicates assembly. The instruction

```
bne x10, x11, 2000 // if x10 != x11, go to location 2000ten =
0111 1101 0000
```

could be assembled into this format (it's actually a bit more

complicated, as we will see):

0	111110	01011	01010	001	1000	0	1100111
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

where the opcode for conditional branches is 1100111_{two} and `bne`'s funct3 code is 001_{two}.

The unconditional jump-and-link instruction (`jal`) is the only instruction that uses the *UJ-type* format. This instruction consists of a 7-bit opcode, a 5-bit destination register operand (rd), and a 20-bit address immediate. The link address, which is the address of the instruction following the `jal`, is written to rd.

Like the SB-type format, the UJ-type format's address operand uses an unusual immediate encoding, and it cannot encode odd addresses. So,

`jal x0, 2000 // go to location` 2000_{ten} `= 0111 1101 0000`

is assembled into this format:

0	1111101000	0	00000000	00000	1101111
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

If addresses of the program had to fit in this 20-bit field, it would mean that no program could be bigger than 2^{20}, which is far too small to be a realistic option today. An alternative would be to specify a register that would always be added to the branch offset, so that a branch instruction would calculate the following:

Program counter = Register + Branch offset

This sum allows the program to be as large as 2^{64} and still be able to use conditional branches, solving the branch address size problem. Then the question is, which register?

The answer comes from seeing how conditional branches are used. Conditional branches are found in loops and in *if* statements, so they tend to branch to a nearby instruction. For example, about half of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away. Since the *program counter* (PC)

contains the address of the current instruction, we can branch within $\pm 2^{10}$ words of the current instruction, or jump within $\pm 2^{18}$ words of the current instruction, if we use the PC as the register to be added to the address. Almost all loops and *if* statements are smaller than 2^{10} words, so the PC is the ideal choice. This form of branch addressing is called **PC-relative addressing**.

PC-relative addressing

An addressing regime in which the address is the sum of the *program counter* (PC) and a constant in the instruction.

Like most recent computers, RISC-V uses PC-relative addressing for both conditional branches and unconditional jumps, because the destination of these instructions is likely to be close to the branch. On the other hand, procedure calls may require jumping more than 2^{18} words away, since there is no guarantee that the callee is close to the caller. Hence, RISC-V allows very long jumps to any 32-bit address with a two-instruction sequence: `lui` writes bits 12 through 31 of the address to a temporary register, and `jalr` adds the lower 12 bits of the address to the temporary register and jumps to the sum.

Since RISC-V instructions are 4 bytes long, the RISC-V branch instructions could have been designed to stretch their reach by having the PC-relative address refer to the number of *words* between the branch and the target instruction, rather than the number of bytes. However, the RISC-V architects wanted to support the possibility of instructions that are only 2 bytes long, so the branch instructions represent the number of *halfwords* between the branch and the branch target. Thus, the 20-bit address field in the `jal` instruction can encode a distance of $\pm 2^{19}$ halfwords, or ±1 MiB from the current PC. Similarly, the 12-bit field in the conditional branch instructions is also a halfword address, meaning that it represents a 13-bit byte address.

Showing Branch Offset in Machine Language

Example

The *while* loop on page 94 was compiled into this RISC-V assembler code:

```
Loop:slli x10, x22, 3 // Temp reg x10 = i * 8
 add x10, x10, x25 // x10 = address of save[i]
 ld x9, 0(x10) // Temp reg x9 = save[i]
 bne x9, x24, Exit // go to Exit if save[i] != k
 addi x22, x22, 1 // i = i + 1
 beq x0, x0, Loop // go to Loop
Exit:
```

Answer

If we assume we place the loop starting at location 80000 in memory, what is the RISC-V machine code for this loop?

The assembled instructions and their addresses are:

Address	Instruction					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

Remember that RISC-V instructions have byte addresses, so addresses of sequential words differ by 4. The `bne` instruction on the fourth line adds 3 words or 12 bytes to the address of the instruction, specifying the branch destination relative to the branch instruction (12+80012) and not using the full destination address (80024). The branch instruction on the last line does a similar calculation for a backwards branch (−20+80020), corresponding to the label `Loop`.

Hardware/Software Interface

Most conditional branches are to a nearby location, but

occasionally they branch far away, farther than can be represented in the 12-bit address in the conditional branch instruction. The assembler comes to the rescue just as it did with large addresses or constants: it inserts an unconditional branch to the branch target, and inverts the condition so that the conditional branch decides whether to skip the unconditional branch.

Branching Far Away

Example

Given a branch on register `x10` being equal to zero,

```
beq x10, x0, L1
```

replace it by a pair of instructions that offers a much greater branching distance.

Answer

These instructions replace the short-address conditional branch:

```
 bne x10, x0, L2
 jal x0, L1
L2:
```

RISC-V Addressing Mode Summary

Multiple forms of addressing are generically called **addressing modes**. Figure 2.17 shows how operands are identified for each addressing mode. The addressing modes of the RISC-V instructions are the following:

1. *Immediate addressing*, where the operand is a constant within the instruction itself.
2. *Register addressing*, where the operand is a register.
3. *Base* or *displacement addressing*, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction.
4. *PC-relative addressing*, where the branch address is the sum of the PC and a constant in the instruction.

addressing mode

One of several addressing regimes delimited by their varied use of

operands and/or addresses.

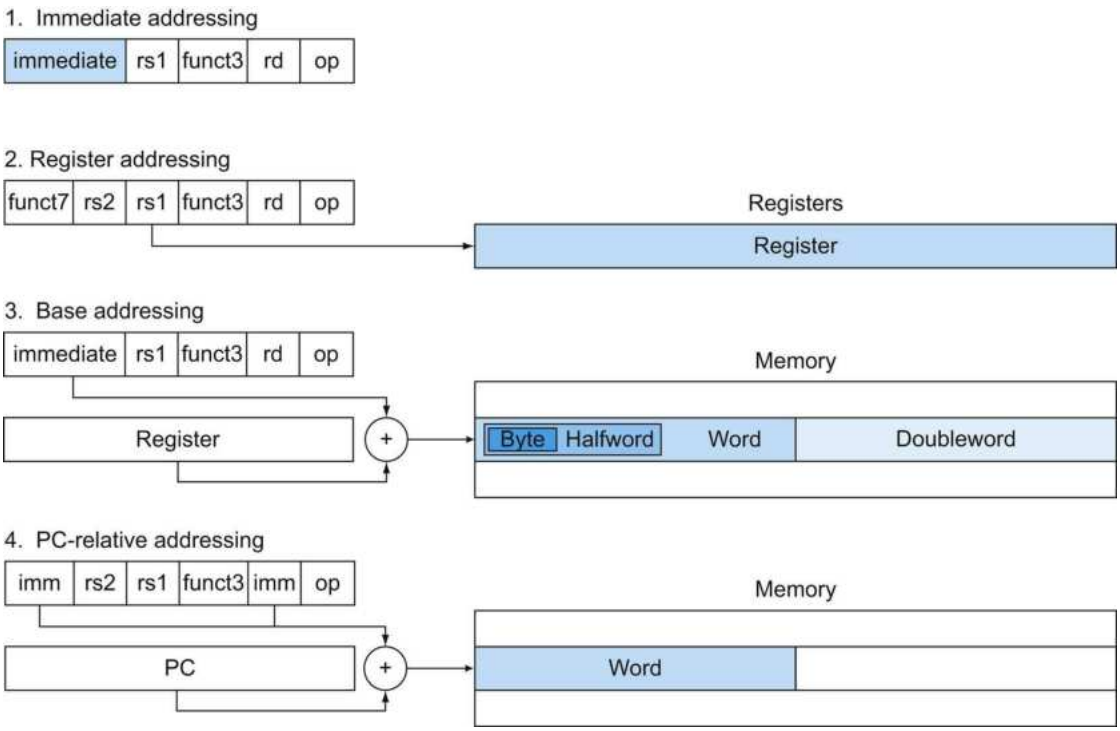


FIGURE 2.17 Illustration of four RISC-V addressing modes.
The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, words, or doublewords. For mode 1, the operand is part of the instruction itself. Mode 4 addresses instructions in memory, with mode 4 adding a long address to the PC. Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (`addi`) and register (`add`) addressing.

Decoding Machine Language

Sometimes you are forced to reverse-engineer machine language to create the original assembly language. One example is when looking at "core dump." Figure 2.18 shows the RISC-V encoding of the opcodes for the RISC-V machine language. This figure helps when translating by hand between assembly language and machine language.

Decoding Machine Code

Example

What is the assembly language statement corresponding to this machine instruction?

00578833_{hex}

Answer

The first step is converting hexadecimal to binary:

```
0000 0000 0101 0111 1000 1000 0011 0011
```

To know how to interpret the bits, we need to determine the instruction format, and to do that we first need to determine the `opcode`. The opcode is the rightmost 7 bits, or 0110011. Searching Figure 2.20 for this value, we see that the opcode corresponds to the R-type arithmetic instructions. Thus, we can parse the binary format into fields listed in Figure 2.21:

funct7	rs2	rs1	funct3	rd	opcode
0000000	00101	01111	000	10000	0110011

We decode the rest of the instruction by looking at the field values. The funct7 and funct3 fields are both zero, indicating the instruction is `add`. The decimal values for the register operands are 5 for the rs2 field, 15 for rs1, and 16 for rd. These numbers represent registers `x5`, `x15`, and `x16`. Now we can reveal the assembly instruction:

```
add x16, x15, x5
```

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	iw	0000011	010	n.a.
	id	0000011	011	n.a.
	ibu	0000011	100	n.a.
	ihu	0000011	101	n.a.
	iwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

FIGURE 2.18 **RISC-V instruction encoding.**

All instructions have an opcode field, and all formats except U-type and UJ-type use the funct3 field. R-type instructions use the funct7 field, and immediate shifts (`slli`, `srli`, `srai`) use the funct6 field.

Figure 2.19 shows all the RISC-V instruction formats. Figure 2.1 on pages 64–65 shows the RISC-V assembly language revealed in this chapter. The next chapter covers RISC-V instructions for multiply, divide, and arithmetic for real numbers.

Check Yourself

I. What is the range of byte addresses for conditional branches in RISC-V (K =1024)?

1. Addresses between 0 and 4K −1
2. Addresses between 0 and 8K −1
3. Addresses up to about 2K before the branch to about 2K after
4. Addresses up to about 4K before the branch to about 4K after

II. What is the range of byte addresses for the jump-and-link instruction in RISC-V (M =1024K)?

1. Addresses between 0 and 512K −1
2. Addresses between 0 and 1M −1
3. Addresses up to about 512K before the branch to about 512K after
4. Addresses up to about 1M before the branch to about 1M after

Name	Field						Comments
(Field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

FIGURE 2.19 RISC-V instruction formats.

2.11 Parallelism and Instructions: Synchronization

Parallel execution is easier when tasks are independent, but often

they need to cooperate. Cooperation usually means some tasks are writing new values that others must read. To know when a task is finished writing so that it is safe for another to read, the tasks need to synchronize. If they don't synchronize, there is a danger of a **data race**, where the results of the program can change depending on how events happen to occur.

data race

Two memory accesses form a data race if they are from different threads to the same location, at least one is a write, and they occur one after another.

For example, recall the analogy of the eight reporters writing a story on pages 44–45 of Chapter 1. Suppose one reporter needs to read all the prior sections before writing a conclusion. Hence, he or she must know when the other reporters have finished their sections, so that there is no danger of sections being changed afterwards. That is, they had better synchronize the writing and reading of each section so that the conclusion will be consistent with what is printed in the prior sections.

In computing, synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied

synchronization instructions. In this section, we focus on the implementation of *lock* and *unlock* synchronization operations. Lock and unlock can be used straightforwardly to create regions where only a single processor can operate, called a *mutual exclusion*, as well as to implement more complex synchronization mechanisms.

The critical ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to *atomically* read and modify a memory location. That is, nothing else can interpose itself between the read and the write of the memory location. Without such a capability, the cost of building basic synchronization primitives will be high and will increase unreasonably as the processor count increases.

There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically. In general, architects do not expect users to employ the basic hardware primitives, but instead expect system programmers will use the primitives to build a synchronization library, a process that is often complex and tricky.

Let's start with one such hardware primitive and show how it can be used to build a basic synchronization primitive. One typical operation for building synchronization operations is the *atomic exchange* or *atomic swap*, which inter-changes a value in a register for a value in memory.

To see how to use this to build a basic synchronization primitive, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock. The value returned from the exchange instruction is 1 if some other processor had already claimed access, and 0 otherwise. In the latter case, the value is also changed to 1, preventing any competing exchange in another processor from also retrieving a 0.

For example, consider two processors that each try to do the exchange simultane-ously: this race is prevented, since exactly one of the processors will perform the exchange first, returning 0, and the second processor will return 1 when it does the exchange. The

key to using the exchange primitive to implement synchronization is that the operation is atomic: the exchange is indivisible, and two simultaneous exchanges will be ordered by the hardware. It is impossible for two processors trying to set the synchronization variable in this manner to both think they have simultaneously set the variable.

Implementing a single atomic memory operation introduces some challenges in the design of the processor, since it requires both a memory read and a write in a single, uninterruptible instruction.

An alternative is to have a pair of instructions in which the second instruction returns a value showing whether the pair of instructions was executed as if the pair was atomic. The pair of instructions is effectively atomic if it appears as if all other operations executed by any processor occurred before or after the pair. Thus, when an instruction pair is effectively atomic, no other processor can change the value between the pair of instructions.

In RISC-V this pair of instructions includes a special load called a *load-reserved doubleword* (`lr.d`) and a special store called a *store-conditional doubleword* (`sc.d`). These instructions are used in sequence: if the contents of the memory location specified by the load-reserved are changed before the store-conditional to the same address occurs, then the store-conditional fails and does not write the value to memory. The store-conditional is defined to both store the value of a (presumably different) register in memory *and* to change the value of another register to a 0 if it succeeds and to a nonzero value if it fails. Thus, `sc.d` specifies three registers: one to hold the address, one to indicate whether the atomic operation failed or succeeded, and one to hold the value to be stored in memory if it succeeded. Since the load-reserved returns the initial value, and the store-conditional returns 0 only if it succeeds, the following sequence implements an atomic exchange on the memory location specified by the contents of `x20`:

```
again:lr.d x10, (x20)   // load-reserved
 sc.d x11, x23, (x20)   // store-conditional
 bnex11, x0, again   // branch if store fails
 addi x23, x10, 0   // put loaded value in x23
```

Any time a processor intervenes and modifies the value in memory between the `lr.d` and `sc.d` instructions, the `sc.d` writes a

nonzero value into `x11`, causing the code sequence to try again. At the end of this sequence, the contents of `x23` and the memory location specified by `x20` have been atomically exchanged.

Elaboration

Although it was presented for multiprocessor synchronization, atomic exchange is also useful for the operating system in dealing with multiple processes in a single processor. To make sure nothing interferes in a single processor, the store-conditional also fails if the processor does a context switch between the two instructions (see Chapter 5).

Elaboration

An advantage of the load-reserved/store-conditional mechanism is that it can be used to build other synchronization primitives, such as *atomic compare and swap* or *atomic fetch-and-increment*, which are used in some parallel programming models. These involve more instructions between the `lr.d` and the `sc.d`, but not too many.

Since the store-conditional will fail after either another attempted store to the load reservation address or any exception, care must be taken in choosing which instructions are inserted between the two instructions. In particular, only integer arithmetic, forward branches, and backward branches out of the load-reserved/store-conditional block can safely be permitted; otherwise, it is possible to create deadlock situations where the processor can never complete the `sc.d` because of repeated page faults. In addition, the number of instructions between the load-reserved and the store-conditional should be small to minimize the probability that either an unrelated event or a competing processor causes the store-conditional to fail frequently.

Elaboration

While the code above implemented an atomic exchange, the following code would more efficiently acquire a lock at the location in register `x20`, where the value of 0 means the lock was free and 1 to mean lock was acquired:

```
addi x12, x0, 1   // copy locked value
```

```
again: lr.d x10, (x20)   // load-reserved to read lock
   bnex10, x0, again   // check if it is 0 yet
   sc.d x11, x12, (x20)   // attempt to store new value
   bnex11, x0, again   // branch if store fails
```

We release the lock just using a regular store to write 0 into the location:

```
sd x0, 0(x20)   // free lock by writing 0
```

Check Yourself

When do you use primitives like load-reserved and store-conditional?

1. When cooperating threads of a parallel program need to synchronize to get proper behavior for reading and writing shared data.
2. When cooperating processes on a uniprocessor need to synchronize for reading and writing shared data.

2.12 Translating and Starting a Program

This section describes the four steps in transforming a C program in a file from storage (disk or flash memory) into a program running on a computer. Figure 2.20 shows the translation hierarchy. Some systems combine these steps to reduce translation time, but programs go through these four logical phases. This section follows this translation hierarchy.

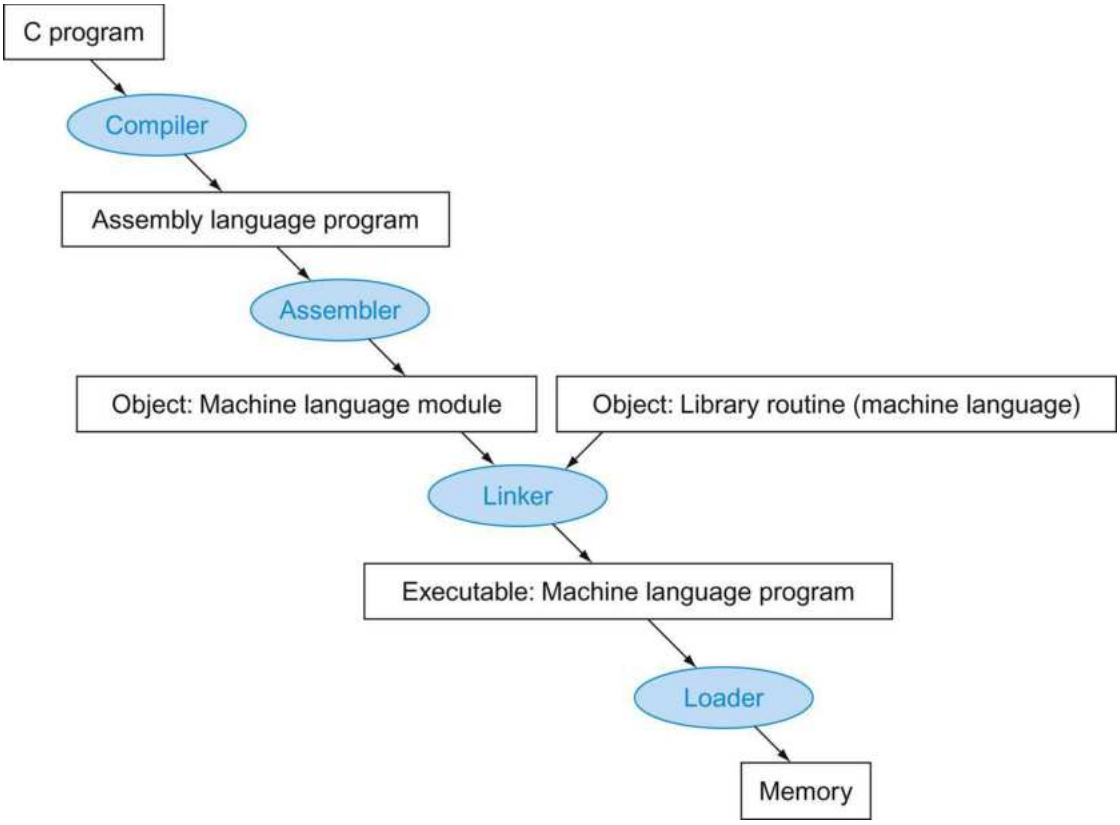


FIGURE 2.20 A translation hierarchy for C.
A high-level language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, UNIX follows a suffix convention for files: C source files are named `x.c`, assembly files are `x.s`, object files are named `x.o`, statically linked library routines are `x.a`, dynamically linked library routes are `x.so`, and executable files by default are called `a.out`. MS-DOS uses the suffixes `.C`, `.ASM`, `.OBJ`, `.LIB`, `.DLL`, and `.EXE` to the same effect.

Compiler

The compiler transforms the C program into an *assembly language program*, a symbolic form of what the machine understands. High-level language programs take many fewer lines of code than assembly language, so programmer productivity is much higher.

In 1975, many operating systems and assemblers were written in **assembly language** because memories were small and compilers were inefficient. The million-fold increase in memory capacity per single DRAM chip has reduced program size concerns, and optimizing compilers today can produce assembly language programs nearly as well as an assembly language expert, and sometimes even better for large programs.

assembly language

A symbolic language that can be translated into binary machine language.

Assembler

Since assembly language is an interface to higher-level software, the assembler can also treat common variations of machine language instructions as if they were instructions in their own right. The hardware need not implement these instructions; however, their appearance in assembly language simplifies translation and programming. Such instructions are called **pseudoinstructions**.

pseudoinstruction

A common variation of assembly language instructions often treated as if it were an instruction in its own right.

As mentioned above, the RISC-V hardware makes sure that register `x0` always has the value 0. That is, whenever register `x0` is used, it supplies a 0, and if the programmer attempts to change the value in `x0`, the new value is simply discarded. Register `x0` is used to create the assembly language instruction that copies the contents of one register to another. Thus, the RISC-V assembler accepts the following instruction even though it is not found in the RISC-V machine language:

```
li x9, 123  // load immediate value 123 into register x9
```

The assembler converts this assembly language instruction into the machine language equivalent of the following instruction:

```
addi x9, x0, 123 // register x9 gets register x0 + 123
```

The RISC-V assembler also converts `mv` (move) into an `addi` instruction. Thus

```
mv x10, x11  // register x10 gets register x11
```

becomes

```
addi x10, x11, 0  // register x10 gets register x11 + 0
```

The assembler also accepts `j` Label to unconditionally branch to a label, as a stand-in for `jal x0, Label`. It also converts branches to faraway locations into a branch and a jump. As mentioned above, the RISC-V assembler allows large constants to be loaded into a register despite the limited size of the immediate instructions. Thus, the *load immediate* (`li`) pseudoinstruction introduced above can create constants larger than `addi`'s immediate field can contain; the *load address* (`la`) macro works similarly for symbolic addresses. Finally, it can simplify the instruction set by determining which variation of an instruction the programmer wants. For example, the RISC-V assembler does not require the programmer to specify the immediate version of the instruction when using a constant for arithmetic and logical instructions; it just generates the proper opcode. Thus

```
and x9, x10, 15 // register x9 gets x10 AND 15
```

becomes

```
andi x9, x10, 15 // register x9 gets x10 AND 15
```

We include the "`i`" on the instructions to remind the reader that `andi` produces a different opcode in a different instruction format than the `and` instruction with no immediate operands.

In summary, pseudoinstructions give RISC-V a richer set of assembly language instructions than those implemented by the hardware. If you are going to write assembly programs, use pseudoinstructions to simplify your task. To understand the RISC-V architecture and be sure to get best performance, however, study the real RISC-V instructions found in Figures 2.1 and 2.18.

Assemblers will also accept numbers in a variety of bases. In addition to binary and decimal, they usually accept a base that is more succinct than binary yet converts easily to a bit pattern. RISC-V assemblers use hexadecimal and octal.

Such features are convenient, but the primary task of an

assembler is assembly into machine code. The assembler turns the assembly language program into an *object file*, which is a combination of machine language instructions, data, and information needed to place instructions properly in memory.

To produce the binary version of each instruction in the assembly language program, the assembler must determine the addresses corresponding to all labels. Assemblers keep track of labels used in branches and data transfer instructions in a **symbol table**. As you might expect, the table contains pairs of symbols and addresses.

symbol table

A table that matches names of labels to the addresses of the memory words that instructions occupy.

The object file for UNIX systems typically contains six distinct pieces:

- The *object file header* describes the size and position of the other pieces of the object file.
- The *text segment* contains the machine language code.
- The *static data segment* contains data allocated for the life of the program. (UNIX allows programs to use both *static data*, which is allocated throughout the program, and *dynamic data*, which can grow or shrink as needed by the program. See Figure 2.13.)
- The *relocation information* identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.
- The *symbol table* contains the remaining labels that are not defined, such as external references.
- The *debugging information* contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable.

The next subsection shows how to attach such routines that have already been assembled, such as library routines.

Linker

What we have presented so far suggests that a single change to one line of one procedure requires compiling and assembling the whole

program. Complete retranslation is a terrible waste of computing resources. This repetition is particularly wasteful for standard library routines, because programmers would be compiling and assembling routines that by definition almost never change. An alternative is to compile and assemble each procedure independently, so that a change to one line would require compiling and assembling only one procedure. This alternative requires a new systems program, called a **link editor** or **linker**, which takes all the independently assembled machine language programs and "stitches" them together. The reason a linker is useful is that it is much faster to patch code than it is to recompile and reassemble.

linker

Also called **link editor**. A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.

There are three steps for the linker:
1. Place code and data modules symbolically in memory.
2. Determine the addresses of data and instruction labels.
3. Patch both the internal and external references.

The linker uses the relocation information and symbol table in each object module to resolve all undefined labels. Such references occur in branch instructions and data addresses, so the job of this program is much like that of an editor: it finds the old addresses and replaces them with the new addresses. Editing is the origin of the name "link editor," or linker for short.

If all external references are resolved, the linker next determines the memory locations each module will occupy. Recall that Figure 2.13 on page 106 shows the RISC-V convention for allocation of program and data to memory. Since the files were assembled in isolation, the assembler could not know where a module's instructions and data would be placed relative to other modules. When the linker places a module in memory, all *absolute* references, that is, memory addresses that are not relative to a register, must be *relocated* to reflect its true location.

The linker produces an **executable file** that can be run on a

computer. Typically, this file has the same format as an object file, except that it contains no unresolved references. It is possible to have partially linked files, such as library routines, that still have unresolved addresses and hence result in object files.

executable file

A functional program in the format of an object file that contains no unresolved references. It can contain symbol tables and debugging information. A "stripped executable" does not contain that information. Relocation information may be included for the loader.

Linking Object Files

Example

Link the two object files below. Show updated addresses of the first few instructions of the completed executable file. We show the instructions in assembly language just to make the example understandable; in reality, the instructions would be numbers.

Note that in the object files we have highlighted the addresses and symbols that must be updated in the link process: the instructions that refer to the addresses of procedures `A` and `B` and the instructions that refer to the addresses of data doublewords `X` and `Y`.

Object file header			
	Name	Procedure `A`	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	`0`	`ld x10, 0(x3)`	
	`4`	`jal x1, 0`	
	. . .	. . .	
Data segment	0	(X)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	`0`	`ld`	`X`
	`4`	`jal`	`B`
Symbol table	Label	Address	
	`X`	–	
	`B`	–	
	Name	Procedure `B`	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	`0`	`sd x11, 0(x3)`	
	`4`	`jal x1, 0`	
	. . .	. . .	
Data segment	0	(Y)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	`0`	`sd`	`Y`
	`4`	`jal`	`A`
Symbol table	Label	Address	
	`Y`	–	
	`A`	–	

Answer

Procedure A needs to find the address for the variable labeled X to put in the load instruction and to find the address of procedure B to place in the jal instruction. Procedure B needs the address of the variable labeled Y for the store instruction and the address of procedure A for its jal instruction.

From Figure 2.14 on page 107, we know that the text segment starts at address $0000\ 0000\ 0040\ 0000_{hex}$ and the data segment at $0000\ 0000\ 1000\ 0000_{hex}$. The text of procedure A is placed at the first address and its data at the second. The object file header for procedure A says that its text is 100_{hex} bytes and its data is 20_{hex} bytes, so the starting address for procedure B text is $40\ 0100_{hex}$, and its data starts at $1000\ 0020_{hex}$.

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0000\ 0000\ 0040\ 0000_{hex}$	ld x10, 0(x3)
	$0000\ 0000\ 0040\ 0004_{hex}$	jal x1, 252_{ten}
	...	...
	$0000\ 0000\ 0040\ 0100_{hex}$	sd x11, 32(x3)
	$0000\ 0000\ 0040\ 0104_{hex}$	jal x1, -260_{ten}
	...	...
Data segment	Address	
	$0000\ 0000\ 1000\ 0000_{hex}$	(X)
	...	...
	$0000\ 0000\ 1000\ 0020_{hex}$	(Y)
	...	...

Now the linker updates the address fields of the instructions. It uses the instruction type field to know the format of the address to be edited. We have three types here:

1. The jump and link instructions use PC-relative addressing. Thus, for the jal at address $40\ 0004_{hex}$ to go to $40\ 0100_{hex}$ (the address of procedure B), it must put ($40\ 0100_{hex}$ – $40\ 0004_{hex}$) or 252_{ten} in its address field. Similarly, since $40\ 0000_{hex}$ is the address of procedure A, the jal at $40\ 0104_{hex}$ gets the negative number -260_{ten} ($40\ 0000_{hex}$ – $40\ 0104_{hex}$) in its address field.
2. The load addresses are harder because they are relative to a base

register. This example uses `x3` as the base register, assuming it is initialized to $0000\ 0000\ 1000\ 0000_{hex}$. To get the address $0000\ 0000\ 1000\ 0000_{hex}$ (the address of doubleword `X`), we place 0_{ten} in the address field of `ld` at address $40\ 0000_{hex}$. Similarly, we place 20_{hex} in the address field of `sd` at address $40\ 0100_{hex}$ to get the address $0000\ 0000\ 1000\ 0020_{hex}$ (the address of doubleword `Y`).

3. Store addresses are handled just like load addresses, except that their S-type instruction format represents immediates differently than loads' I-type format. We place 32_{ten} in the address field of `sd` at address $40\ 0100_{hex}$ to get the address $0000\ 0000\ 1000\ 0020_{hex}$ (the address of doubleword `Y`).

Loader

Now that the executable file is on disk, the operating system reads it to memory and starts it. The **loader** follows these steps in UNIX systems:

loader

A systems program that places an object program in main memory so that it is ready to execute.

1. Reads the executable file header to determine size of the text and data segments.
2. Creates an address space large enough for the text and data.
3. Copies the instructions and data from the executable file into memory.
4. Copies the parameters (if any) to the main program onto the stack.
5. Initializes the processor registers and sets the stack pointer to the first free location.
6. Branches to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an `exit` system call.

Virtually every problem in computer science can be solved by another

level of indirection.

David Wheeler

Dynamically Linked Libraries

The first part of this section describes the traditional approach to linking libraries before the program is run. Although this static approach is the fastest way to call library routines, it has a few disadvantages:

- The library routines become part of the executable code. If a new version of the library is released that fixes bugs or supports new hardware devices, the statically linked program keeps using the old version.
- It loads all routines in the library that are called anywhere in the executable, even if those calls are not executed. The library can be large relative to the program; for example, the standard C library on a RISC-V system running the Linux operating system is 1.5 MiB.

These disadvantages lead to **dynamically linked libraries (DLLs)**, where the library routines are not linked and loaded until the program is run. Both the program and library routines keep extra information on the location of nonlocal procedures and their names. In the original version of DLLs, the loader ran a dynamic linker, using the extra information in the file to find the appropriate libraries and to update all external references.

dynamically linked libraries (DLLs)
Library routines that are linked to a program during execution.

The downside of the initial version of DLLs was that it still linked all routines of the library that might be called, versus just those that are called during the running of the program. This observation led to the lazy procedure linkage version of DLLs, where each routine is linked only *after* it is called.

Like many innovations in our field, this trick relies on a level of indirection. Figure 2.21 shows the technique. It starts with the nonlocal routines calling a set of dummy routines at the end of the program, with one entry per nonlocal routine. These dummy

entries each contain an indirect branch.

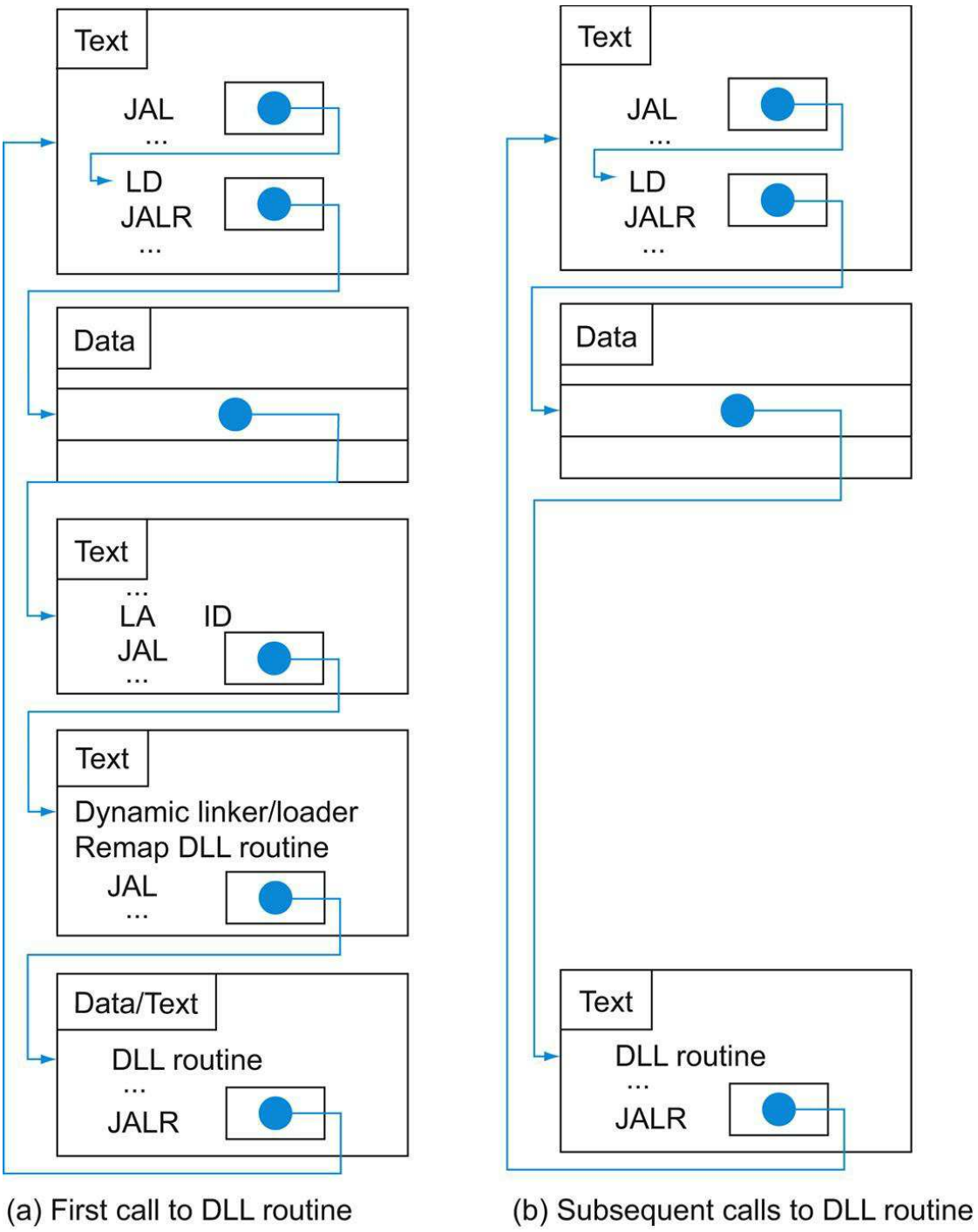


FIGURE 2.21 Dynamically linked library via lazy procedure linkage.
(a) Steps for the first time a call is made to the DLL routine. (b) The steps to find the routine, remap it, and link it are skipped on subsequent calls. As we will see in Chapter 5, the operating system may avoid copying the desired routine by remapping it using virtual memory management.

The first time the library routine is called, the program calls the

dummy entry and follows the indirect branch. It points to code that puts a number in a register to identify the desired library routine and then branches to the dynamic linker/loader. The linker/loader finds the wanted routine, remaps it, and changes the address in the indirect branch location to point to that routine. It then branches to it. When the routine completes, it returns to the original calling site. Thereafter, the call to the library routine branches indirectly to the routine without the extra hops.

In summary, DLLs require additional space for the information needed for dynamic linking, but do not require that whole libraries be copied or linked. They pay a good deal of overhead the first time a routine is called, but only a single indirect branch thereafter. Note that the return from the library pays no extra overhead. Microsoft's Windows relies extensively on dynamically linked libraries, and it is also the default when executing programs on UNIX systems today.

Starting a Java Program

The discussion above captures the traditional model of executing a program, where the emphasis is on fast execution time for a program targeted to a specific instruction set architecture, or even a particular implementation of that architecture. Indeed, it is possible to execute Java programs just like C. Java was invented with a different set of goals, however. One was to run safely on any computer, even if it might slow execution time.

Figure 2.22 shows the typical translation and execution steps for Java. Rather than compile to the assembly language of a target computer, Java is compiled first to instructions that are easy to interpret: the **Java bytecode** instruction set (see  **Section 2.15**). This instruction set is designed to be close to the Java language so that this compilation step is trivial. Virtually no optimizations are performed. Like the C compiler, the Java compiler checks the types of data and produces the proper operation for each type. Java programs are distributed in the binary version of these bytecodes.

Java bytecode

Instruction from an instruction set designed to interpret Java

programs.

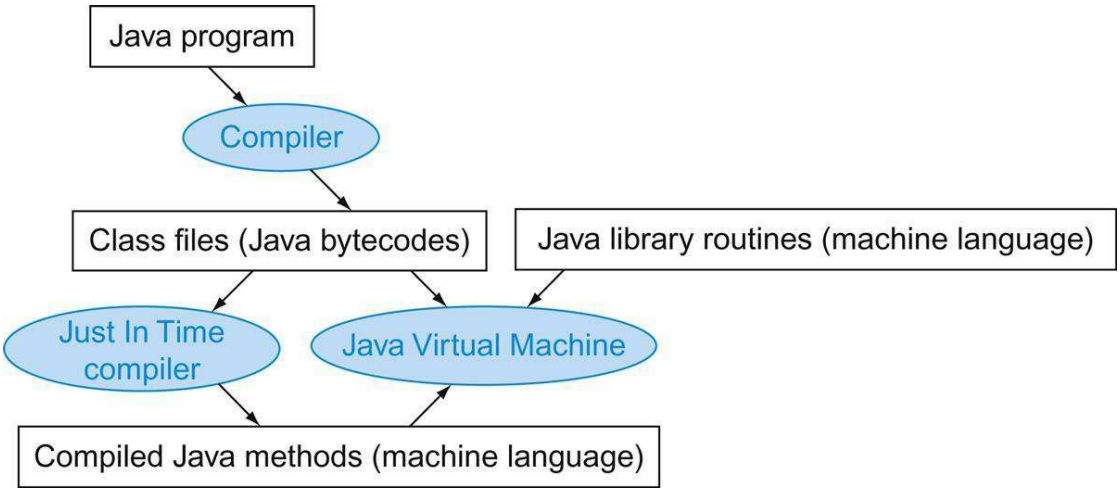


FIGURE 2.22 A translation hierarchy for Java. A Java program is first compiled into a binary version of Java bytecodes, with all addresses defined by the compiler. The Java program is now ready to run on the interpreter, called the *Java Virtual Machine* (JVM). The JVM links to desired methods in the Java library while the program is running. To achieve greater performance, the JVM can invoke the JIT compiler, which selectively compiles methods into the native machine language of the machine on which it is running.

A software interpreter, called a **Java Virtual Machine (JVM)**, can execute Java bytecodes. An interpreter is a program that simulates an instruction set architecture. For example, the RISC-V simulator used with this book is an interpreter. There is no need for a separate assembly step since either the translation is so simple that the compiler fills in the addresses or JVM finds them at runtime.

Java Virtual Machine (JVM)

The program that interprets Java bytecodes.

The upside of interpretation is portability. The availability of software Java virtual machines meant that most people could write and run Java programs shortly after Java was announced. Today, Java virtual machines are found in billions of devices, in everything

from cell phones to Internet browsers.

The downside of interpretation is lower performance. The incredible advances in performance of the 1980s and 1990s made interpretation viable for many important applications, but the factor of 10 slowdown when compared to traditionally compiled C programs made Java unattractive for some applications.

To preserve portability and improve execution speed, the next phase of Java's development was compilers that translated *while* the program was running. Such **Just In Time compilers (JIT)** typically profile the running program to find where the "hot" methods are and then compile them into the native instruction set on which the virtual machine is running. The compiled portion is saved for the next time the program is run, so that it can run faster each time it is run. This balance of interpretation and compilation evolves over time, so that frequently run Java programs suffer little of the overhead of interpretation.

Just In Time compiler (JIT)

The name commonly given to a compiler that operates at runtime, translating the interpreted code segments into the native code of the computer.

As computers get faster so that compilers can do more, and as researchers invent betters ways to compile Java on the fly, the performance gap between Java and C or C++ is closing.  **Section 2.15** goes into much greater depth on the implementation of Java, Java bytecodes, JVM, and JIT compilers.

Check Yourself

Which of the advantages of an interpreter over a translator was the most important for the designers of Java?

1. Ease of writing an interpreter
2. Better error messages
3. Smaller object code
4. Machine independence

2.13 A C Sort Example to Put it All Together

One danger of showing assembly language code in snippets is that you will have no idea what a full assembly language program looks like. In this section, we derive the RISC-V code from two procedures written in C: one to swap array elements and one to sort them.

The Procedure `swap`

Let's start with the code for the procedure `swap` in Figure 2.23. This procedure simply swaps two locations in memory. When translating from C to assembly language by hand, we follow these general steps:

1. Allocate registers to program variables.
2. Produce code for the body of the procedure.
3. Preserve registers across the procedure invocation.

```
void swap(long long int v[], size_t k)
{
    long long int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

FIGURE 2.23 A C procedure that swaps two locations in memory.
This subsection uses this procedure in a sorting example.

This section describes the `swap` procedure in these three pieces, concluding by putting all the pieces together.

Register Allocation for `swap`

As mentioned on page 98, the RISC-V convention on parameter

passing is to use registers x10 to x17. Since swap has just two parameters, v and k, they will be found in registers x10 and x11. The only other variable is temp, which we associate with register x5 since swap is a leaf procedure (see page 102). This register allocation corresponds to the variable declarations in the first part of the swap procedure in Figure 2.23.

Code for the Body of the Procedure swap

The remaining lines of C code in swap are

```
temp= v[k];
v[k]= v[k+1];
v[k+1] = temp;
```

Recall that the memory address for RISC-V refers to the *byte* address, and so doublewords are really 8 bytes apart. Hence, we need to multiply the index k by 8 before adding it to the address. *Forgetting that sequential doubleword addresses differ by 8 instead of by 1 is a common mistake in assembly language programming*. Hence, the first step is to get the address of v[k] by multiplying k by 8 via a shift left by 3:

```
slli  x6, x11, 3  // reg x6 = k * 8
add  x6, x10, x6  // reg x6 = v + (k * 8)
```

Now we load v[k] using x6, and then v[k+1] by adding 8 to x6:

```
ld  x5, 0(x6)  // reg x5 (temp) = v[k]
ld  x7, 8(x6)  // reg x7 = v[k + 1]
          // refers to next element of v
```

Next we store x9 and x11 to the swapped addresses:

```
sd  x7, 0(x6)  // v[k] = reg x7
sd  x5, 8(x6)  // v[k+1] = reg x5 (temp)
```

Now we have allocated registers and written the code to perform the operations of the procedure. What is missing is the code for preserving the saved registers used within swap. Since we are not using saved registers in this leaf procedure, there is nothing to preserve.

The Full swap Procedure

We are now ready for the whole routine. All that remains is to add the procedure label and the return branch.

```
swap:
sllix6, x11, 3  // reg x6 = k * 8
```

```
addx6, x10, x6  // reg x6 = v + (k * 8)
ldx5, 0(x6)  // reg x5 (temp) = v[k]
ldx7, 8(x6)  // reg x7 = v[k + 1]
sdx7, 0(x6)  // v[k] = reg x7
sdx5, 8(x6)  // v[k+1] = reg x5 (temp)
jalrx0, 0(x1)  // return to calling routine
```

The Procedure `sort`

To ensure that you appreciate the rigor of programming in assembly language, we'll try a second, longer example. In this case, we'll build a routine that calls the swap procedure. This program sorts an array of integers, using bubble or exchange sort, which is one of the simplest if not the fastest sorts. Figure 2.24 shows the C version of the program. Once again, we present this procedure in several steps, concluding with the full procedure.

```
void sort (long long int v[], size_t int n)
{
    size_t i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v,j);
        }
    }
}
```

FIGURE 2.24 A C procedure that performs a sort on the array `v`.

Register Allocation for `sort`

The two parameters of the procedure `sort`, `v` and `n`, are in the parameter registers `x10` and `x11`, and we assign register `x19` to `i` and register `x20` to `j`.

Code for the Body of the Procedure `sort`

The procedure body consists of two nested *for* loops and a call to `swap` that includes parameters. Let's unwrap the code from the outside to the middle.

The first translation step is the first *for* loop:

```
for (i = 0; i < n; i += 1) {
```

Recall that the C *for* statement has three parts: initialization, loop test, and iteration increment. It takes just one instruction to initialize `i` to 0, the first part of the *for* statement:

```
li  x19,  0
```

(Remember that `li` is a pseudoinstruction provided by the assembler for the convenience of the assembly language programmer; see page 125.) It also takes just one instruction to increment `i`, the last part of the *for* statement:

```
addi  x19, x19, 1  // i += 1
```

The loop should be exited if `i < n` is *not* true or, said another way, should be exited if `i ≥ n`. This test takes just one instruction:

```
for1tst: bge x19, x11, exit1// go to exit1 if x19 ≥ x1 (i≥n)
```

The bottom of the loop just branches back to the loop test:

```
  j for1tst  // branch to test of outer loop
exit1:
```

The skeleton code of the first *for* loop is then

```
  li x19, 0  // i = 0
for1tst:
  bge x19, x11, exit1   // go to exit1 if x19 ≥ x1 (i≥n)
  ...
  (body of first for loop)
  ...
  addi x19, x19, 1  // i += 1
  j  for1tst  // branch to test of outer loop
exit1:
```

Voila! (The exercises explore writing faster code for similar loops.)

The second *for* loop looks like this in C:

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

The initialization portion of this loop is again one instruction:

```
addi    x20, x19, -1    // j = i - 1
```

The decrement of `j` at the end of the loop is also one instruction:

```
addi  x20, x20, -1 j -= 1
```

The loop test has two parts. We exit the loop if either condition fails, so the first test must exit the loop if it fails ($j < 0$):

```
for2tst:
  blt x20, x0, exit2    // go to exit2 if x20 < 0 (j < 0)
```

This branch will skip over the second condition test. If it doesn't skip, then `j ≥ 0`.

The second test exits if `v[j] > v[j + 1]` is *not* true, or exits if `v[j] ≤ v[j + 1]`. First we create the address by multiplying `j` by 8 (since we need a byte address) and add it to the base address of `v`:

```
slli  x5, x20, 3  // reg x5 = j * 8
add  x5, x10, x5  // reg x5 = v + (j * 8)
```

Now we load `v[j]`:

```
ld  x6, 0(x5)  // reg x6 = v[j]
```

Since we know that the second element is just the following doubleword, we add 8 to the address in register `x5` to get `v[j + 1]`:

```
ld  x7, 8(x5)  // reg x7 = v[j + 1]
```

We test `v[j] ≤ v[j + 1]` to exit the loop:

```
ble  x6, x7, exit2   // go to exit2 if x6 ≤ x7
```

The bottom of the loop branches back to the inner loop test:

```
jfor2tst  // branch to test of inner loop
```

Combining the pieces, the skeleton of the second *for* loop looks

like this:

```
  addi x20, x19, -1 // j = i - 1
  for2tst: blt x20, x0, exit2  // go to exit2 if x20 < 0 (j <
0)
   sllix5, x20, 3 // reg x5 = j * 8
   addx5, x10, x5 // reg x5 = v + (j * 8)
   ldx6, 0(x5) // reg x6 = v[j]
   ldx7, 8(x5) // reg x7 = v[j + 1]
   ble x6, x7, exit2 // go to exit2 if x6 ≤ x7
   . . .
   (body of second for loop)
   . . .
   addi x20, x20, -1 // j -= 1
   j for2tst // branch to test of inner loop
  exit2:
```

The Procedure Call in `sort`

The next step is the body of the second *for* loop:

```
swap(v,j);
```

Calling `swap` is easy enough:

```
jal x1, swap
```

Passing Parameters in `sort`

The problem comes when we want to pass parameters because the `sort` procedure needs the values in registers `x10` and `x11`, yet the swap procedure needs to have its parameters placed in those same registers. One solution is to copy the parameters for `sort` into other registers earlier in the procedure, making registers `x10` and `x11` available for the call of `swap`. (This copy is faster than saving and restoring on the stack.) We first copy `x10` and `x11` into `x21` and `x22` during the procedure:

```
mv x21, x10 // copy parameter x10 into x21
mv x22, x11 // copy parameter x11 into x22
```

Then we pass the parameters to `swap` with these two instructions:

```
mv x10, x21 // first swap parameter is v
mv x11, x20 // second swap parameter is j
```

Preserving Registers in `sort`

The only remaining code is the saving and restoring of registers. Clearly, we must save the return address in register `x1`, since `sort` is a procedure and is itself called. The `sort` procedure also uses the callee-saved registers `x19`, `x20`, `x21`, and `x22`, so they must be saved. The prologue of the `sort` procedure is then

```
addi sp, sp, -40 // make room on stack for 5 regs
sd x1, 32(sp) // save x1 on stack
sd x22, 24(sp) // save x22 on stack
sd x21, 16(sp) // save x21 on stack
sd x20, 8(sp) // save x20 on stack
sd x19, 0(sp) // save x19 on stack
```

The tail of the procedure simply reverses all these instructions, and then adds a `jalr` to return.

The Full Procedure `sort`

Now we put all the pieces together in Figure 2.25, being careful to replace references to registers `x10` and `x11` in the *for* loops with references to registers `x21` and `x22`. Once again, to make the code easier to follow, we identify each block of code with its purpose in the procedure. In this example, nine lines of the `sort` procedure in C became 34 lines in the RISC-V assembly language.

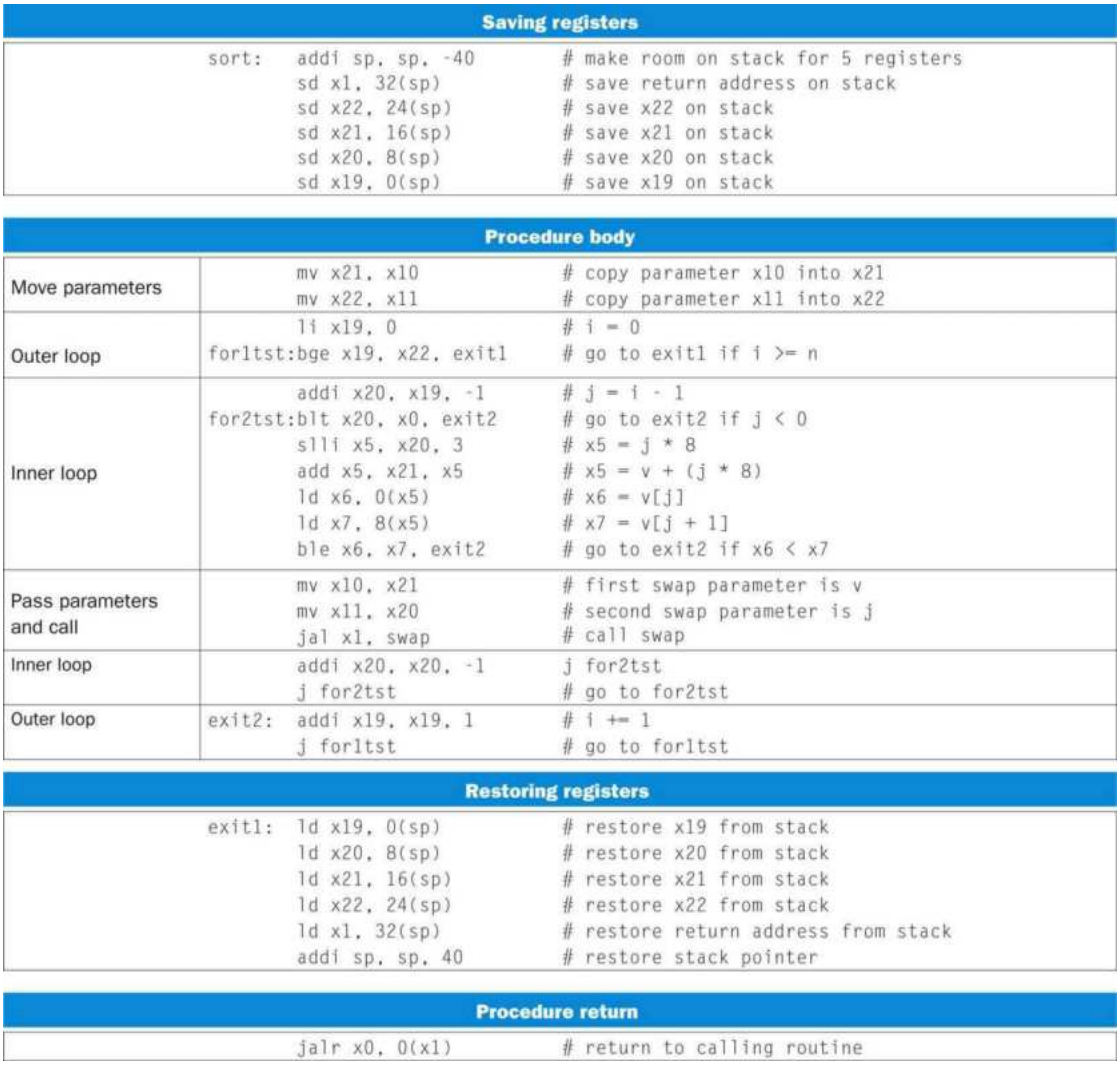


FIGURE 2.25 RISC-V assembly version of procedure `sort` in Figure 2.27.

Elaboration

One optimization that works with this example is *procedure inlining*. Instead of passing arguments in parameters and invoking the code with a `jal` instruction, the compiler would copy the code from the body of the `swap` procedure where the call to `swap` appears in the code. Inlining would avoid four instructions in this example. The downside of the inlining optimization is that the compiled code would be bigger if the inlined procedure is called from several locations. Such a code expansion might turn into *lower* performance if it increased the cache miss rate; see Chapter 5.

Figure 2.26 shows the impact of compiler optimization on sort program performance, compile time, clock cycles, instruction count, and CPI. Note that unoptimized code has the best CPI, and O1 optimization has the lowest instruction count, but O3 is the fastest, reminding us that time is the only accurate measure of program performance.

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (procedure integration)	2.41	65,747	44,993	1.46

FIGURE 2.26 Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort.
The programs sorted 100,000 32-bit words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

Figure 2.27 compares the impact of programming languages, compilation versus interpretation, and algorithms on performance of sorts. The fourth column shows that the unoptimized C program is 8.3 times faster than the interpreted Java code for Bubble Sort. Using the JIT compiler makes Java 2.1 times *faster* than the unoptimized C and within a factor of 1.13 of the highest optimized C code. ( **Section 2.15** gives more details on interpretation versus compilation of Java and the Java and jalr code for Bubble Sort.) The ratios aren't as close for Quicksort in Column 5, presumably because it is harder to amortize the cost of runtime compilation over the shorter execution time. The last column demonstrates the impact of a better algorithm, offering three orders of magnitude a performance increase by when sorting 100,000 items. Even comparing interpreted Java in Column 5 to the C compiler at highest optimization in Column 4, Quicksort beats

Bubble Sort by a factor of 50 (0.05×2468, or 123 times faster than the unoptimized C code versus 2.41 times faster).

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	Compiler	None	1.00	1.00	2468
	Compiler	01	2.37	1.50	1562
	Compiler	02	2.38	1.50	1555
	Compiler	03	2.41	1.91	1955
Java	Interpreter	–	0.12	0.05	1050
	JIT compiler	–	2.13	0.29	338

FIGURE 2.27 Performance of two sort algorithms in C and Java using interpretation and optimizing compilers relative to unoptimized C version. The last column shows the advantage in performance of Quicksort over Bubble Sort for each language and execution option. These programs were run on the same system as in Figure 2.29. The JVM is Sun version 1.3.1, and the JIT is Sun Hotspot version 1.3.1.

2.14 Arrays versus Pointers

A challenge for any new C programmer is understanding pointers. Comparing assembly code that uses arrays and array indices to the assembly code that uses pointers offers insights about pointers. This section shows C and RISC-V assembly versions of two procedures to clear a sequence of doublewords in memory: one using array indices and one with pointers. Figure 2.28 shows the two C procedures.

```
clear1(long long int array[], size_t int size)
{
    size_t i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}
clear2(long long int *array, size_t int size)
{
    long long int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```

FIGURE 2.28 Two C procedures for setting an array to all zeros.

`clear1` uses indices, while `clear2` uses pointers. The second procedure needs some explanation for those unfamiliar with C. The address of a variable is indicated by &, and the object pointed to by a pointer is indicated by *. The declarations declare that `array` and `p` are pointers to integers. The first part of the *for* loop in `clear2` assigns the address of the first element of `array` to the pointer `p`. The second part of the *for* loop tests to see if the pointer is pointing beyond the last element of `array`. Incrementing a pointer by one, in the bottom part of the *for* loop, means moving the pointer to the next sequential object of its declared size. Since `p` is a pointer to integers, the compiler will generate RISC-V instructions to increment `p` by eight, the number of bytes in an RISC-V integer. The assignment in the loop places 0 in the object pointed to by `p`.

The purpose of this section is to show how pointers map into RISC-V instructions, and not to endorse a dated programming style. We'll see the impact of modern compiler optimization on these two procedures at the end of the section.

Array Version of Clear

Let's start with the array version, `clear1`, focusing on the body of the loop and ignoring the procedure linkage code. We assume that the two parameters `array` and `size` are found in the registers `x10` and `x11`, and that `i` is allocated to register `x5`.

The initialization of `i`, the first part of the *for* loop, is straightforward:

```
li  x5, 0  // i = 0 (register x5 = 0)
```

To set `array[i]` to 0 we must first get its address. Start by multiplying `i` by 8 to get the byte address:

```
loop1: slli  x6, x5, 3  // x6 = i * 8
```

Since the starting address of the array is in a register, we must add it to the index to get the address of `array[i]` using an add instruction:

```
add x7, x10, x6  // x7 = address of array[i]
```

Finally, we can store 0 in that address:

```
sd  x0, 0(x7)  // array[i] = 0
```

This instruction is the end of the body of the loop, so the next step is to increment `i`:

```
addi x5, x5, 1  // i = i + 1
```

The loop test checks if `i` is less than size:

```
blt x5, x11, loop1 // if (i < size) go to loop1
```

We have now seen all the pieces of the procedure. Here is the RISC-V code for clearing an array using indices:

```
 li x5, 0 // i = 0
loop1: slli x6, x5, 3 // x6 = i * 8
 add x7, x10, x6 // x7 = address of array[i]
 sd x0, 0(x7) // array[i] = 0
 addi x5, x5, 1 // i = i + 1
 blt x5, x11, loop1 // if (i < size) go to loop1
```

(This code works as long as `size` is greater than 0; ANSI C requires a test of size before the loop, but we'll skip that legality here.)

Pointer Version of Clear

The second procedure that uses pointers allocates the two parameters `array` and `size` to the registers `x10` and `x11` and allocates `p` to register `x5`. The code for the second procedure starts with assigning the pointer `p` to the address of the first element of the array:

```
mv x5, x10  // p = address of array[0]
```

The next code is the body of the *for* loop, which simply stores 0 into `p`:

```
loop2: sd  x0, 0(x5)  // Memory[p] = 0
```

This instruction implements the body of the loop, so the next code is the iteration increment, which changes `p` to point to the next doubleword:

```
addi  x5, x5, 8  // p = p + 8
```

Incrementing a pointer by 1 means moving the pointer to the next sequential object in C. Since `p` is a pointer to integers declared as `long long int`, each of which uses 8 bytes, the compiler increments `p` by 8.

The loop test is next. The first step is calculating the address of the last element of `array`. Start with multiplying `size` by 8 to get its byte address:

```
slli  x6, x11, 3  // x6 = size * 8
```

and then we add the product to the starting address of the array to get the address of the first doubleword *after* the array:

```
add  x7, x10, x6  // x7 = address of array[size]
```

The loop test is simply to see if `p` is less than the last element of `array`:

```
bltu x5, x7, loop2  // if (p<&array[size]) go to loop2
```

With all the pieces completed, we can show a pointer version of the code to zero an array:

```
mvx5, x10 // p = address of array[0]
loop2: sdx0, 0(x5) // Memory[p] = 0
addi  x5, x5, 8 // p = p + 8
slli x6, x11, 3 // x6 = size * 8
add x7, x10, x6 // x7 = address of array[size]
bltu x5, x7, loop2 // if (p<&array[size]) go to loop2
```

As in the first example, this code assumes `size` is greater than 0.

Note that this program calculates the address of the end of the array in every iteration of the loop, even though it does not change. A faster version of the code moves this calculation outside the loop:

```
mv x5, x10 // p = address of array[0]
slli x6, x11, 3 // x6 = size * 8
add x7, x10, x6 // x7 = address of array[size]
loop2: sdx0, 0(x5) // Memory[p] = 0
addi x5, x5, 8 // p = p + 8
bltu x5, x7, loop2 // if (p < &array[size]) go to loop2
```

Comparing the Two Versions of Clear

Comparing the two code sequences side by side illustrates the difference between array indices and pointers (the changes introduced by the pointer version are highlighted):

```
        li   x5, 0          // i = 0
loop1: slli x6, x5, 3       // x6 = i * 8
        add  x7, x10, x6    // x7 = address of array[i]
        sd   x0, 0(x7)      // array[i] = 0
        addi x5, x5, 1      // i = i + 1
        blt  x5, x11, loop1 // if (i < size) go to loop1
```

```
        mv   x5, x10        // p = address of array[0]
        slli x6, x11, 3     // x6 = size * 8
        add  x7, x10, x6    // x7 = address of array[size]
loop2: sd   x0, 0(x5)       // Memory[p] = 0
        addi x5, x5, 8      // p = p + 8
        bltu x5, x7, loop2  // if (p < &array[size]) go to loop2
```

The version on the left must have the "multiply" and add inside the loop because `i` is incremented and each address must be recalculated from the new index. The memory pointer version on the right increments the pointer `p` directly. The pointer version moves the scaling shift and the array bound addition outside the loop, thereby reducing the instructions executed per iteration from five to three. This manual optimization corresponds to the compiler optimization of strength reduction (shift instead of multiply) and induction variable elimination (eliminating array address calculations within loops).  **Section 2.15** describes these two and many other optimizations.

Elaboration

As mentioned earlier, a C compiler would add a test to be sure that `size` is greater than 0. One way would be to branch to the instruction after the loop with `blt x0, x11, afterLoop`.

Understanding Program Performance

People were once taught to use pointers in C to get greater efficiency than that available with arrays: "Use pointers, even if you can't understand the code." Modern optimizing compilers can produce code for the array version that is just as good. Most programmers today prefer that the compiler do the heavy lifting.

Advanced Material: Compiling C and Interpreting Java

This section gives a brief overview of how the C compiler works and how Java is executed. Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to understanding performance. Keep in mind that the subject of compiler construction is usually taught in a one- or two-semester course, so our introduction will necessarily only touch on the basics.

The second part of this section is for readers interested in seeing how an **object-oriented language** like Java executes on an RISC-V architecture. It shows the Java byte-codes used for interpretation and the RISC-V code for the Java version of some of the C segments in prior sections, including Bubble Sort. It covers both the Java Virtual Machine and JIT compilers.

object-oriented language

A programming language that is oriented around objects rather than actions, or data versus logic.

The rest of  **Section 2.15** can be found online.

2.15 Advanced Material: Compiling C and Interpreting Java

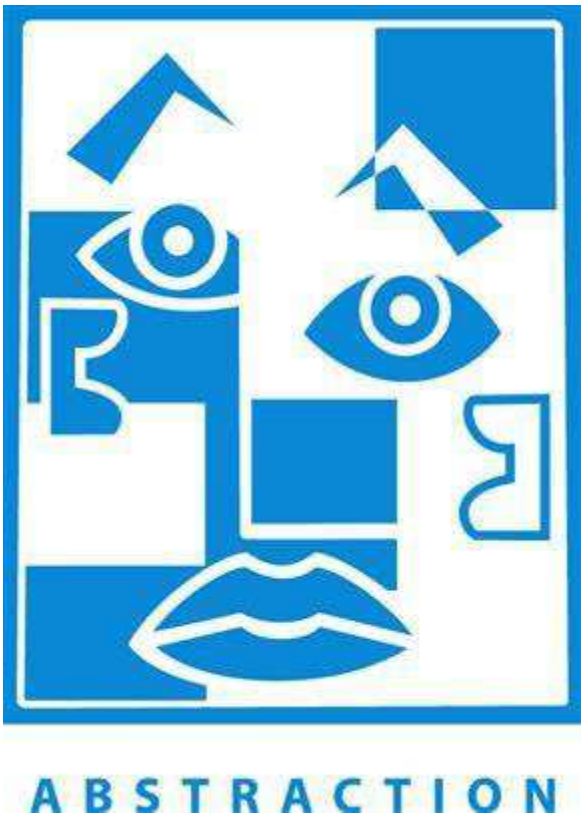
This section gives a brief overview of how the C compiler works and how Java is executed. Because the compiler will significantly affect the performance of a computer, understanding compiler

technology today is critical to understanding performance. Keep in mind that the subject of compiler construction is usually taught in a one- or two-semester course, so our introduction will necessarily only touch on the basics.

The second part of this section, starting on page 150.e15, is for readers interested in seeing how an objected-oriented language like Java executes on the RISC-V architecture. It shows the Java bytecodes used for interpretation and the RISC-V code for the Java version of some of the C segments in prior sections, including Bubble Sort. It covers both the Java virtual machine and just-in-time (JIT) compilers.

Compiling C

This first part of the section introduces the internal **anatomy** of a compiler. To start, Figure e2.15.1 shows the structure of recent compilers, and we describe the optimizations in the order of the passes of that structure.



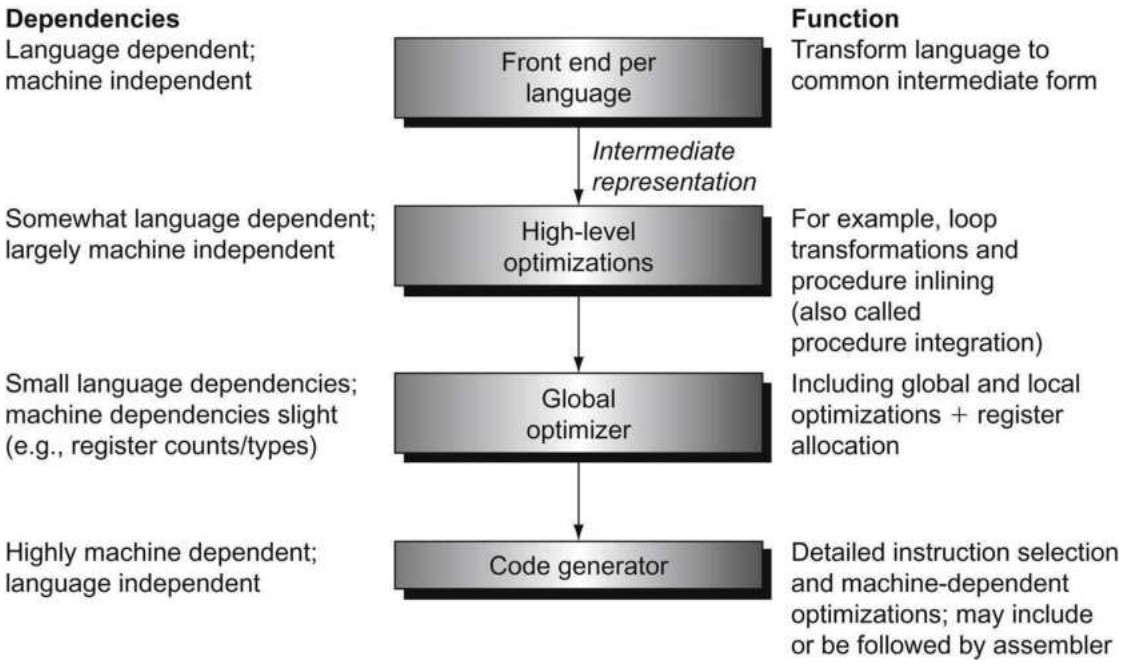


FIGURE E2.15.1 The structure of a modern optimizing compiler consists of a number of passes or phases.
Logically, each pass can be thought of as running to completion before the next occurs. In practice, some passes may handle one procedure at a time, essentially interleaving with another pass.

To illustrate the concepts in this part of this section, we will use the C version of a *while* loop from page 95:

```
while (save[i] == k)
 i += 1;
```

The Front End

The function of the front end is to read in a source program; check the syntax and semantics; and translate the source program to an intermediate form that interprets most of the language-specific operation of the program. As we will see, intermediate forms are usually simple, and some are, in fact, similar to the Java bytecodes (see Figure e2.15.8).

The front end is typically broken into four separate functions:
1. *Scanning* reads in individual characters and creates a string of tokens. Examples of *tokens* are reserved words, names, operators, and punctuation symbols. In the above example, the token sequence is `while, (, save, [, i, ], ==, k, ), i, +=, 1`. A word like `while` is recognized as a reserved word in C, but `save`, `i`, and `j`

are recognized as names, and `1` is recognized as a number.
2. *Parsing* takes the token stream, ensures the syntax is correct, and produces an *abstract syntax tree*, which is a representation of the syntactic structure of the program. Figure e2.15.2 shows what the abstract syntax tree might look like for this program fragment.
3. *Semantic analysis* takes the abstract syntax tree and checks the program for semantic correctness. Semantic checks normally ensure that variables and types are properly declared and that the types of operators and objects match, a step called *type checking*. During this process, a symbol table representing all the named objects—classes, variables, and functions—is usually created and used to type-check the program.
4. *Generation of the intermediate representation* (IR) takes the symbol table and the abstract syntax tree and generates the intermediate representation that is the output of the front end. Intermediate representations usually use simple operations on a small set of primitive types, such as integers, characters, and reals. Java bytecodes represent one type of intermediate form. In modern compilers, the most common intermediate form looks much like the RISC-V instruction set but with an infinite number of virtual registers; later, we describe how to map these virtual registers to a finite set of real registers. Figure e2.15.3 shows how our example might be represented in such an intermediate form.

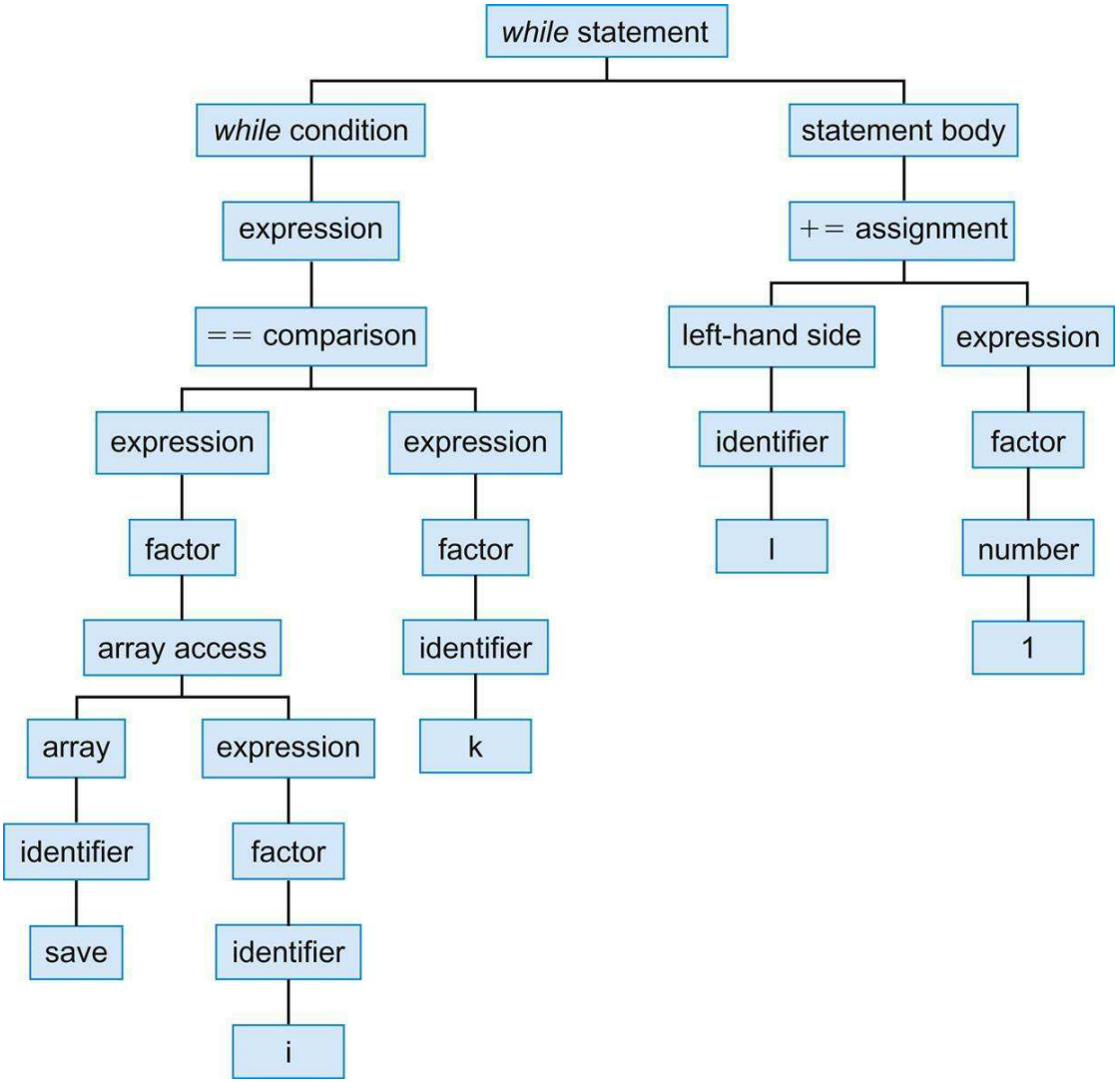


FIGURE E2.15.2 An abstract syntax tree for the *while* example.
The roots of the tree consist of the informational tokens such as numbers and names. Long chains of straight-line descendents are often omitted in constructing the tree.

```
loop:
  # comments are written like this--source code often included
  # while (save[i] == k)
  la   r100, save          # r100 = &save[0]
  ld   r101, i
  li   r102, 8
  mul  r103, r101, r102
  add  r104, r103, r100
  ld   r105, 0(r104)       # r105 = save[i]
  ld   r106, k
  bne  r105, r106, exit
  # i += 1
  ld   r106, i
  addi r107, r106, i       # increment
  sd   r107, i
  j    loop                # next iteration
exit:
```

FIGURE E2.15.3 The *while* loop example is shown using a typical intermediate representation. In practice, the names `save`, `i`, and `k` would be replaced by some sort of address, such as a reference to either the local stack pointer or a global pointer, and an offset, similar to the way `save[i]` is accessed. Note that the format of the RISC-V instructions is different from the rest of the chapter, because they represent intermediate representations here using `rXX` notation for virtual registers.

The intermediate form specifies the functionality of the program in a manner independent of the original source. After this front end has created the intermediate form, the remaining passes are largely language independent.

High-Level Optimizations

High-level optimizations are transformations that are done at something close to the source level.

The most common high-level transformation is probably *procedure inlining*, which replaces a call to a function by the body of the function, substituting the caller's arguments for the procedure's parameters. Other high-level optimizations involve loop transformations that can reduce loop overhead, improve memory access, and exploit the hardware more effectively. For example, in loops that execute many iterations, such as those traditionally

controlled by a *for* statement, the optimization of **loop-unrolling** is often useful. Loop-unrolling involves taking a loop, replicating the body multiple times, and executing the transformed loop fewer times. Loop-unrolling reduces the loop overhead and provides opportunities for many other optimizations. Other types of high-level transformations include sophisticated loop transformations such as interchanging nested loops and blocking loops to obtain better memory behavior; see Chapter 5 for examples.

loop-unrolling

A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

Local and Global Optimizations

Within the pass dedicated to local and global optimization, three classes of optimization are performed:

1. *Local optimization* works within a single basic block. A local optimization pass is often run as a precursor and successor to global optimization to "clean up" the code before and after global optimization.
2. *Global optimization* works across multiple basic blocks; we will see an example of this shortly.
3. Global *register allocation* allocates variables to registers for regions of the code. Register allocation is crucial to getting good performance in modern processors.

Several optimizations are performed both locally and globally, including common subexpression elimination, constant propagation, copy propagation, dead store elimination, and strength reduction. Let's look at some simple examples of these optimizations.

Common subexpression elimination finds multiple instances of the same expression and replaces the second one by a reference to the first. Consider, for example, a code segment to add 4 to an array element:

```
x[i] = x[i] + 4
```

The address calculation for `x[i]` occurs twice and is identical since neither the starting address of `x` nor the value of `i` changes.

Thus, the calculation can be reused. Let's look at the intermediate code for this fragment, since it allows several other optimizations to be performed. The unoptimized intermediate code is on the left. On the right is the optimized code, using common subexpression elimination to replace the second address calculation with the first. Note that the register allocation has not yet occurred, so the compiler is using virtual register numbers like `r100` here.

```
// x[i] + 4 // x[i] + 4
la r100,x la r100,x
ld r101,i ld r101,i
mul r102,r101,8 slli r102,r101,3
add r103,r100,r102 add r103,r100,r102
ld r104, 0(r103) ld r104, 0(r103)
// // value of x[i] is in r104
addi r105, r104,4 addi r105, r104,4
la r106,x sd r105, 0(r103)
ld r107,i
mul r108,r107,8
add r109,r106,r107
sd r105,0(r109)
```

If the same optimization were possible across two basic blocks, it would then be an instance of *global common subexpression elimination.*

Let's consider some of the other optimizations:

- *Strength reduction* replaces complex operations by simpler ones and can be applied to this code segment, replacing the mul by a shift left.
- *Constant propagation* and its sibling *constant folding* find constants in code and propagate them, collapsing constant values whenever possible.
- *Copy propagation* propagates values that are simple copies, eliminating the need to reload values and possibly enabling other optimizations, such as common subexpression elimination.
- *Dead store elimination* finds stores to values that are not used again and eliminates the store; its "cousin" is *dead code elimination*, which finds unused code—code that cannot affect the result of the program—and eliminates it. With the heavy use of macros, templates, and the similar techniques designed to reuse code in high-level languages, dead code occurs surprisingly often. Compilers must be *conservative*. The first task of a compiler is to

produce correct code; its second task is usually to produce fast code, although other factors, such as code size, may sometimes be important as well. Code that is fast but incorrect—for any possible combination of inputs—is simply wrong. Thus, when we say a compiler is "conservative," we mean that it performs an optimization only if it knows with 100% certainty that, no matter what the inputs, the code will perform as the user wrote it. Since most compilers translate and optimize one function or procedure at a time, most compilers, especially at lower optimization levels, assume the worst about function calls and about their own parameters.

Understanding Program Performance

Programmers concerned about the performance of critical loops, especially in real-time or embedded applications, can find themselves staring at the assembly language produced by a compiler and wondering why the compiler failed to perform some global optimization or to allocate a variable to a register throughout a loop. The answer often lies in the dictate that the compiler be conservative. The opportunity for improving the code may seem obvious to the programmer, but then the programmer often has knowledge that the compiler does not have, such as the absence of aliasing between two pointers or the absence of side effects by a function call. The compiler may indeed be able to perform the transformation with a little help, which could eliminate the worst-case behavior that it must assume. This insight also illustrates an important observation: programmers who use pointers to try to improve performance in accessing variables, especially pointers to values on the stack that also have names as variables or as elements of arrays, are likely to disable many compiler optimizations. The result is that the lower-level pointer code may run no better, or perhaps even worse, than the higher-level code optimized by the compiler.

Global Code Optimizations

Many global code optimizations have the same aims as those used in the local case, including common subexpression elimination, constant propagation, copy propagation, and dead store and dead

code elimination.

There are two other important global optimizations: code motion and induction variable elimination. Both are loop optimizations; that is, they are aimed at code in loops. *Code motion* finds code that is loop invariant: a particular piece of code computes the same value on every iteration of the loop and, hence, may be computed once outside the loop. *Induction variable elimination* is a combination of transformations that reduce overhead on indexing arrays, essentially replacing array indexing with pointer accesses. Rather than examine induction variable elimination in depth, we point the reader to Section 2.14, which compares the use of array indexing and pointers; for most loops, a modern optimizing compiler can perform the transformation from the more obvious array code to the faster pointer code.

Implementing Local Optimizations

Local optimizations are implemented on basic blocks by scanning the basic block in instruction execution order, looking for optimization opportunities. In the assignment statement example on page 150.e6, the duplication of the entire address calculation is recognized by a series of sequential passes over the code. Here is how the process might proceed, including a description of the checks that are needed:

1. Determine that the two `LDA` operations return the same result by observing that the operand `x` is the same and that the value of its address has not been changed between the two `LDA` operations.
2. Replace all uses of `R106` in the basic block by `R101`.
3. Observe that `i` cannot change between the two `LDURs` that reference it. So replace all uses of `R107` with `R101`.
4. Observe that the `MUL` instructions now have the same input operands, so that `R108` may be replaced by `R102`.
5. Observe that now the two `ADD` instructions have identical input operands (`R100` and `R102`), so replace the `R109` with `R103`.
6. Use dead store code elimination to delete the second set of `LDA`,`LDUR`, `MUL`, and `ADD` instructions since their results are unused.

Throughout this process, we need to know when two instances of an operand have the same value. This is easy to determine when they refer to virtual registers, since our intermediate representation

uses such registers only once, but the problem can be trickier when the operands are variables in memory, even though we are only considering references within a basic block.

It is reasonably easy for the compiler to make the common subexpression elimination determination in a conservative fashion in this case; as we will see in the next subsection, this is more difficult when branches intervene.

Implementing Global Optimizations

To understand the challenge of implementing global optimizations, let's consider a few examples:

- Consider the case of an opportunity for common subexpression elimination, say, of an IR statement like `ADD Rx, R20, R50`. To determine whether two such statements compute the same value, we must determine whether the values of `R20` and `R50` are identical in the two statements. In practice, this means that the values of `R20` and `R50` have not changed between the first statement and the second. For a single basic block, this is easy to decide; it is more difficult for a more complex program structure involving multiple basic blocks and branches.
- Consider the second `LDUR` of `i` into `R107` within the earlier example: how do we know whether its value is used again? If we consider only a single basic block, and we know that all uses of `R107` are within that block, it is easy to see. As optimization proceeds, however, common subexpression elimination and copy propagation may create other uses of a value. Determining that a value is unused and the code is dead is more difficult in the case of multiple basic blocks.
- Finally, consider the load of `k` in our loop, which is a candidate for code motion. In this simple example, we might argue that it is easy to see that `k` is not changed in the loop and is, hence, loop invariant. Imagine, however, a more complex loop with multiple nestings and *if* statements within the body. Determining that the load of `k` is loop invariant is harder in such a case.

The information we need to perform these global optimizations is similar: we need to know where each operand in an IR statement could have been changed or *defined* (use-definition information). The dual of this information is also needed: that is, finding all the

uses of that changed operand (definition-use information). *Data flow analysis* obtains both types of information.

Global optimizations and data flow analysis operate on a *control flow graph*, where the nodes represent basic blocks and the arcs represent control flow between basic blocks. Figure e2.15.4 shows the control flow graph for our simple loop example, with one important transformation introduced. We describe the transformation in the caption, but see if you can discover it, and why it was done, on your own!

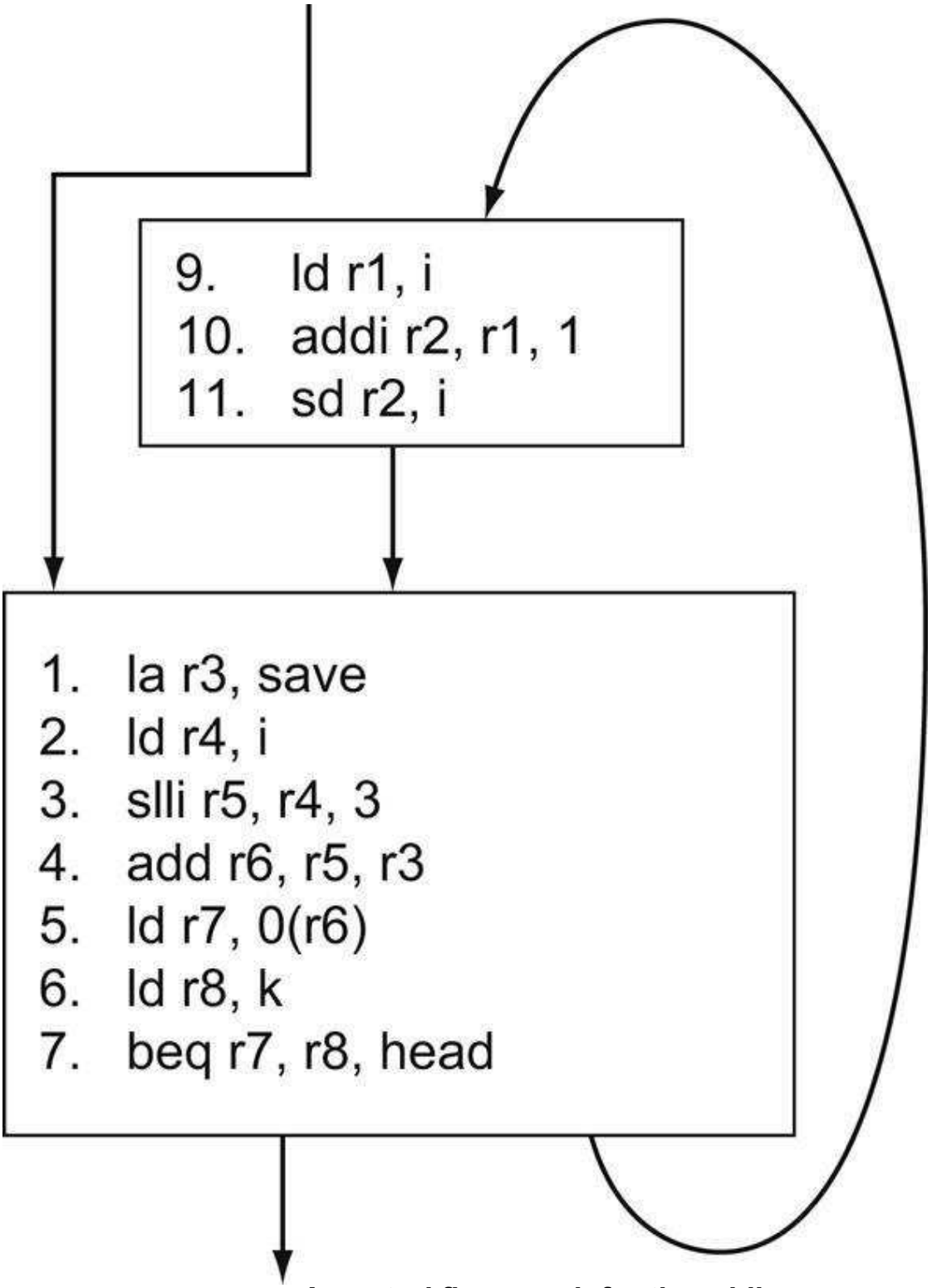


FIGURE E2.15.4 A control flow graph for the *while* loop example.
Each node represents a basic block, which terminates with a branch or by sequential fall-through into another basic block that is also the target of a branch. The IR statements have been numbered for ease in referring to them. The important transformation performed was to move the *while* test and conditional branch to the

end. This eliminates the unconditional branch that was formerly inside the loop and places it before the loop. This transformation is so important that many compilers do it during the generation of the IR. The `mul` was also replaced with ("strength-reduced to") an `slli`.

Suppose we have computed the use-definition information for the control flow graph in Figure e2.15.4. How does this information allow us to perform code motion? Consider IR statements number 1 and 6: in both cases, the use-definition information tells us that there are no definitions (changes) of the operands of these statements within the loop. Thus, these IR statements can be moved outside the loop. Notice that if the `LDA` of `save` and the `LDUR` of `k` are executed once, just prior to the loop entrance, the computational effect is the same, but the program now runs faster since these two statements are outside the loop. In contrast, consider IR statement 2, which loads the value of `i`. The definitions of `i` that affect this statement are both outside the loop, where `i` is initially defined, and inside the loop in statement 10 where it is stored. Hence, this statement is not loop invariant.

Figure e2.15.5 shows the code after performing both code motion and induction variable elimination, which simplifies the address calculation. The variable `i` can still be register allocated, eliminating the need to load and store it every time, and we will see how this is done in the next subsection.

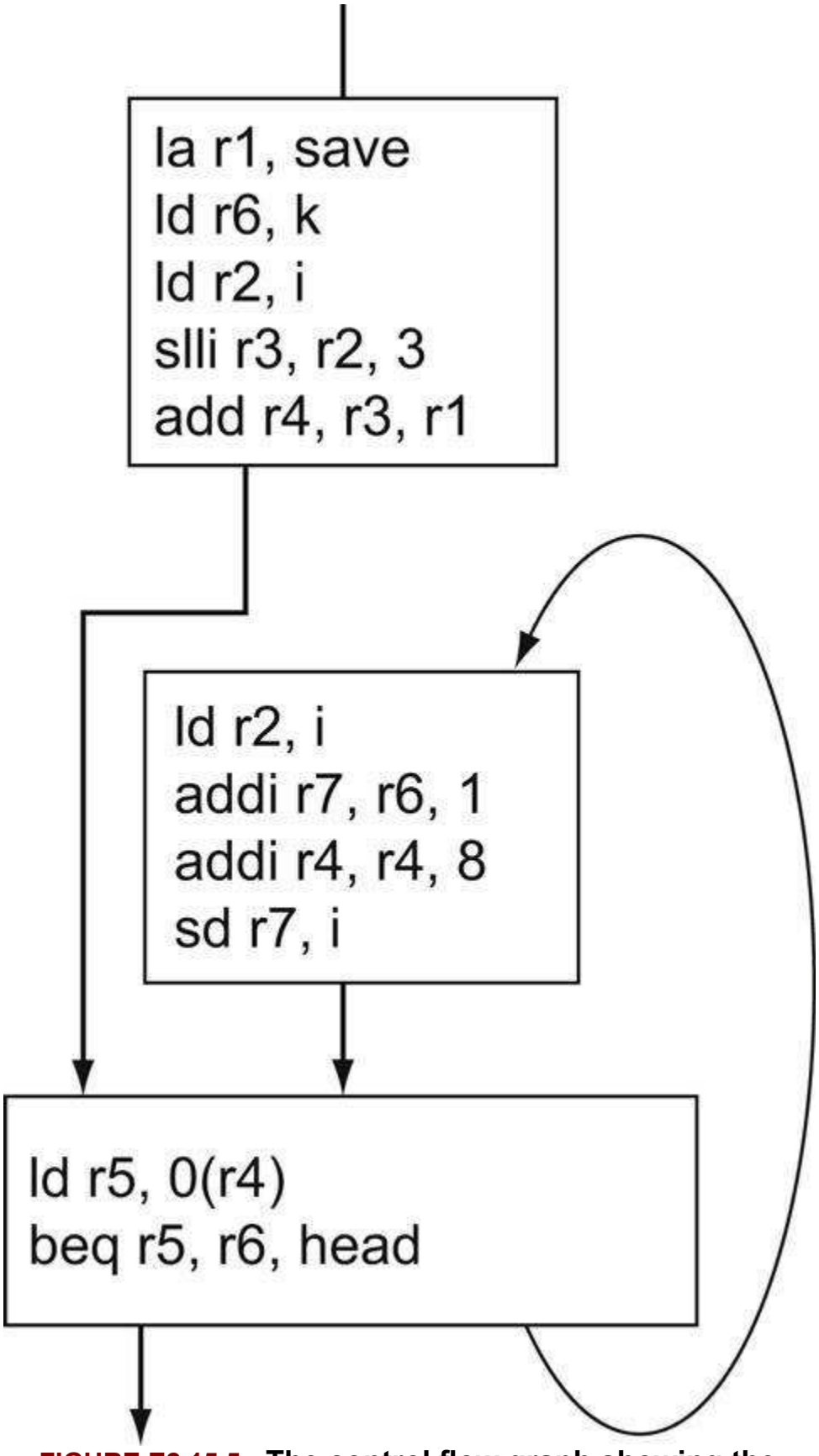


FIGURE E2.15.5 The control flow graph showing the representation of the *while* loop example after code motion and induction variable elimination. The number of instructions in the inner loop has been

reduced from 10 to 6.

Before we turn to register allocation, we need to mention a caveat that also illustrates the complexity and difficulty of optimizers. Remember that the compiler must be cautious. To be conservative, a compiler must consider the following question: Is there *any way* that the variable `k` could possibly ever change in this loop? Unfortunately, there is one way. Suppose that the variable `k` and the variable `i` actually refer to the same memory location, which could happen if they were accessed by pointers or reference parameters.

I am sure that many readers are saying, "Well, that would certainly be a stupid piece of code!" Alas, this response is not open to the compiler, which must translate the code as it is written. Recall too that the aliasing information must also be conservative; thus, compilers often find themselves negating optimization opportunities because of a possible alias that exists in one place in the code or because of incomplete information about aliasing.

Register Allocation

Register allocation is perhaps the most important optimization for modern load-store architectures. Eliminating a load or a store gets rid of an instruction. Furthermore, register allocation enhances the value of other optimizations, such as common subexpression elimination. Fortunately, the trend toward larger register counts in modern architectures has made register allocation simpler and more effective. Register allocation is done on both a local basis and a global basis, that is, across multiple basic blocks but within a single function. Local register allocation is usually done late in compilation, as the final code is generated. Our focus here is on the more challenging and more opportunistic global register allocation.

Modern global register allocation uses a region-based approach, where a region (sometimes called a *live range*) represents a section of code during which a particular variable could be allocated to a particular register. How is a region selected? The process is iterative:

1. Choose a definition (change) of a variable in a given basic block; add that block to the region.
2. Find any uses of that definition, which is a data flow analysis

problem; add any basic blocks that contain such uses, as well as any basic block that the value passes through to reach a use, to the region.
3. Find any other definitions that also can affect a use found in the previous step and add the basic blocks containing those definitions, as well as the blocks the definitions pass through to reach a use, to the region.
4. Repeat steps 2 and 3 using the definitions discovered in step 3 until convergence.

The set of basic blocks found by this technique has a special property: if the designated variable is allocated to a register in all these basic blocks, then there is no need for loading and storing the variable.

Modern global register allocators start by constructing the regions for every virtual register in a function. Once the regions are constructed, the key question is how to allocate a register to each region: the challenge is that certain regions overlap and may not use the same register. Regions that do not overlap (i.e., share no common basic blocks) can share the same register. One way to record the interference among regions is with an *interference graph*, where each node represents a region, and the arcs between nodes represent that the regions have some basic blocks in common.

Once an interference graph has been constructed, the problem of allocating registers is equivalent to a famous problem called *graph coloring:* find a color for each node in a graph such that no two adjacent nodes have the same color. If the number of colors equals the number of registers, then coloring an interference graph is equivalent to allocating a register for each region! This insight was the initial motivation for the allocation method now known as region-based allocation, but originally called the graph-coloring approach. Figure e2.15.6 shows the flow graph representation of the *while* loop example after register allocation.

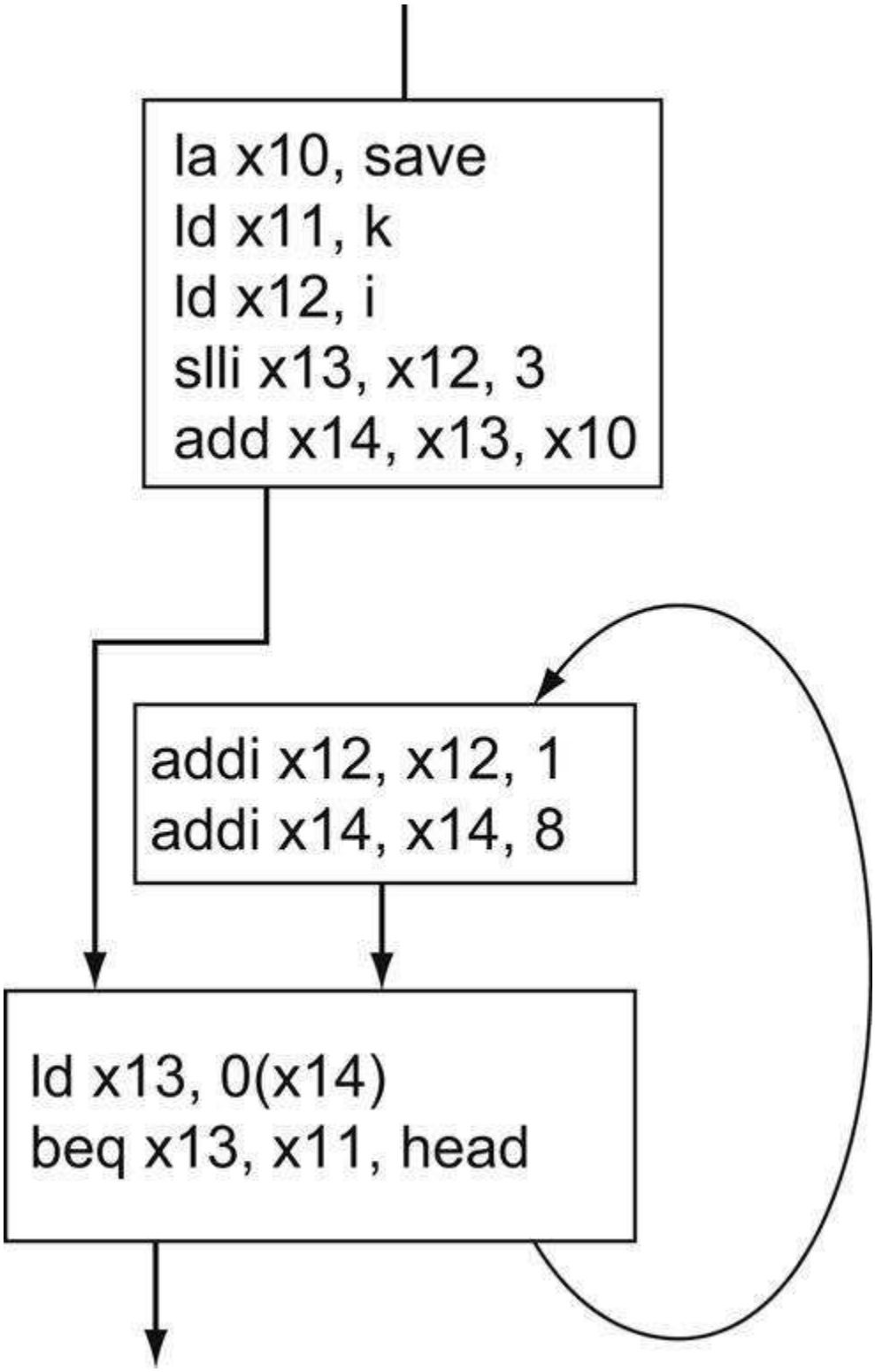


FIGURE E2.15.6 The control flow graph showing the representation of the *while* loop example after code motion and induction variable elimination and register allocation, using the RISC-V register names.

The number of IR statements in the inner loop has now dropped to only four from six before register allocation and 10 before any global optimizations. The value of `i`

resides in `x12` at the end of the loop and may need to be stored eventually to maintain the program semantics. If `i` were unused after the loop, not only could the store be avoided, but also the increment inside the loop could be eliminated!

What happens if the graph cannot be colored using the number of registers available? The allocator must spill registers until it can complete the coloring. By doing the coloring based on a priority function that takes into account the number of memory references saved and the cost of tying up the register, the allocator attempts to avoid spilling for the most important candidates.

Spilling is equivalent to splitting up a region (or live range); if the region is split, fewer other regions will interfere with the two separate nodes representing the original region. A process of splitting regions and successive coloring is used to allow the allocation process to complete, at which point all candidates will have been allocated a register. Of course, whenever a region is split, loads and stores must be introduced to get the value from memory or to store it there. The location chosen to split a region must balance the cost of the loads and stores that must be introduced against the advantage of freeing up a register and reducing the number of interferences.

Modern register allocators are incredibly effective in using the large register counts available in modern processors. In many programs, the effectiveness of register allocation is limited not by the availability of registers but by the possibilities of aliasing that cause the compiler to be conservative in its choice of candidates.

Code Generation

The final steps of the compiler are code generation and assembly. Most compilers do not use a stand-alone assembler that accepts assembly language source code; to save time, they instead perform most of the same functions: filling in symbolic values and generating the binary code as the last stage of code generation.

In modern processors, code generation is reasonably straightforward, since the simple architectures make the choice of instruction relatively obvious. Code generation is more complex for the more complicated architectures, such as the x86, since multiple

IR instructions may collapse into a single machine instruction. In modern compilers, this compilation process uses pattern matching with either a tree-based pattern matcher or a pattern matcher driven by a parser.

During code generation, the final stages of machine-dependent optimization are also performed. These include some constant folding optimizations, as well as localized instruction scheduling (see Chapter 4).

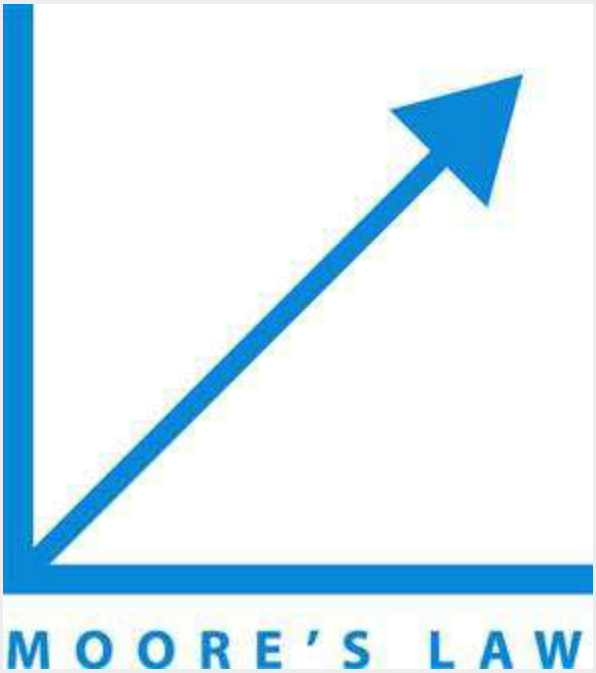
Optimization Summary

Figure e2.15.7 gives examples of typical optimizations, and the last column indicates where the optimization is performed in the gcc compiler. It is sometimes difficult to separate some of the simpler optimizations—local and processor-dependent optimizations—from transformations done in the code generator, and some optimizations are done multiple times, especially local optimizations, which may be performed before and after global optimization as well as during code generation.

Hardware/Software Interface

Today, essentially all programming for desktop and server applications is done in high-level languages, as is most programming for embedded applications. This development means that since most instructions executed are the output of a compiler, an instruction set architecture is mainly a compiler target. With **Moore's Law** comes the temptation of adding sophisticated operations in an instruction set. The challenge is that they may not exactly match what the compiler needs to produce or may be so general that they aren't fast. For example, consider special loop instructions found in some computers. Suppose that instead of decrementing by one, the compiler wanted to increment by four, or instead of branching on not equal zero, the compiler wanted to branch if the index was less than or equal to the limit. The loop instruction may be a mismatch. When faced with such objections, the instruction set designer might next generalize the operation, adding another operand to specify the increment and perhaps an option on which branch condition to use. Then the danger is that a common case, say, incrementing by one, will be

slower than a sequence of simple operations.



Elaboration

Some more sophisticated compilers, and many research compilers, use an analysis technique called *interprocedural analysis* to obtain more information about functions and how they are called. Interprocedural analysis attempts to discover what properties remain true across a function call. For example, we might discover that a function call can never change any global variables, which might be useful in optimizing a loop that calls such a function. Such information is called *may-information* or *flow-insensitive information* and can be obtained reasonably efficiently, although analyzing a call to a function F requires analyzing all the functions that F calls, which makes the process somewhat time consuming for large programs. A more costly property to discover is that a function *must* always change some variable; such information is called *must-information* or *flow-sensitive information*. Recall the dictate to be conservative: may-information can never be used as must-information—just because a function *may* change a variable does not mean that it *must* change it. It is conservative, however, to use the negation of may-information, so the compiler can rely on

the fact that a function *will* never change a variable in optimizations around the call site of that function.

Optimization name	Explanation	gcc level
High level	*At or near the source level; processor independent*	
Procedure integration	Replace procedure call by procedure body	03
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	01
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	01
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	01
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	02
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A* = *X*) with *X*	02
Code motion	Remove code from a loop that computes the same value each iteration of the loop	02
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	02
Processor dependent	*Depends on processor knowledge*	
Strength reduction	Many examples; replace multiply by a constant with shifts	01
Pipeline scheduling	Reorder instructions to improve pipeline performance	01
Branch offset optimization	Choose the shortest branch displacement that reaches target	01

FIGURE E2.15.7 Major types of optimizations and explanation of each class.
The third column shows when these occur at different levels of optimization in gcc. The GNU organization calls the three optimization levels medium (O1), full (O2), and full with integration of small procedures (O3).

One of the most important uses of interprocedural analysis is to obtain so-called alias information. An *alias* occurs when two names may designate the same variable. For example, it is quite helpful to know that two pointers passed to a function may never designate the same variable. Alias information is usually flow-insensitive and must be used conservatively.

Interpreting Java

This second part of the section is for readers interested in seeing how an **object-oriented language** like Java executes on an RISC-V architecture. It shows the Java bytecodes used for interpretation and the RISC-V code for the Java version of some of the C segments in prior sections, including Bubble Sort.

object-oriented language
A programming language that is oriented around objects rather

than actions, or data versus logic.

Let's quickly review the Java lingo to make sure we are all on the same page. The big idea of object-oriented programming is for programmers to think in terms of abstract objects, and operations are associated with each *type* of object. New types can often be thought of as refinements to existing types, and so the new types use some operations for the existing types without change. The hope is that the programmer thinks at a higher level, and that code can be reused more readily if the programmer implements the common operations on many different types.

This different perspective led to a different set of terms. The type of an object is a *class*, which is the definition of a new data type together with the operations that are defined to work on that data type. A particular object is then an *instance* of a class, and creating an object from a class is called *instantiation*. The operations in a class are called *methods*, which are similar to C procedures. Rather than call a procedure as in C, you *invoke* a method in Java. The other members of a class are *fields*, which correspond to variables in C. Variables inside objects are called *instance fields*. Rather than access a structure with a pointer, Java uses an *object reference* to access an object. The syntax for method invocation is `x.y`, where `x` is an object reference and `y` is the method name.

The parent–child relationship between older and newer classes is captured by the verb "extends": a child class *extends* (or subclasses) a parent class. The child class typically will redefine some of the methods found in the parent to match the new data type. Some methods work fine, and the child class *inherits* those methods.

To reduce the number of errors associated with pointers and explicit memory deallocation, Java automatically frees unused storage, using a separate garbage collector that frees memory when it is full. Hence, `new` creates a new instance of a dynamic object on the heap, but there is no `free` in Java. Java also requires array bounds to be checked at runtime to catch another class of errors that can occur in C programs.

Interpretation

As mentioned before, Java programs are distributed as Java

bytecodes, and the Java Virtual Machine (JVM) executes Java byte codes. The JVM understands a binary format called the *class file* format. A class file is a stream of bytes for a single class, containing a table of valid methods with their bytecodes, a pool of constants that acts in part as a symbol table, and other information such as the parent class of this class.

When the JVM is first started, it looks for the class method `main`. To start any Java class, the JVM dynamically loads, links, and initializes a class. The JVM loads a class by first finding the binary representation of the proper class (class file) and then creating a class from that binary representation. Linking combines the class into the runtime state of the JVM so that it can be executed. Finally, it executes the class initialization method that is included in every class.

Figure e2.15.8 shows Java bytecodes and their corresponding RISC-V instructions, illustrating five major differences between the two:

1. To simplify compilation, Java uses a stack instead of registers for operands. Operands are pushed on the stack, operated on, and then popped off the stack.
2. The designers of the JVM were concerned about code size, so bytecodes vary in length between one and five bytes, versus the four-byte, fixed-size RISC-V instructions. To save space, the JVM even has redundant instructions of varying lengths whose only difference is size of the immediate. This decision illustrates a code size variation of our third design principle: make the common case *small*.
3. The JVM has safety features embedded in the architecture. For example, array data transfer instructions check to be sure that the first operand is a reference and that the second index operand is within bounds.
4. To allow garbage collectors to find all live pointers, the JVM uses different instructions to operate on addresses versus integers so that the JVM can know what operands contain addresses. RISC-V generally lumps integers and addresses together.
5. Finally, unlike RISC-V, Java bytecodes include Java-specific instructions that perform complex operations, like allocating an array on the heap or invoking a method.

Compiling a *while* Loop in Java Using Bytecodes

Example

Compile the *while* loop from page 95, this time using Java bytecodes:

```
while (save[i] == k)
 i += 1;
```

Assume that `i`, `k`, and `save` are the first three local variables. Show the addresses of the bytecodes. The RISC-V version of the C loop in Figure e2.15.3 took six instructions and 24 bytes. How big is the bytecode version?

Answer

The first step is to put the array reference in `save` on the stack:

```
0 aload_3 // Push local variable 3 (save[]) onto stack
```

This 1-byte instruction informs the JVM that an address in local variable 3 is being put on the stack. The 0 on the left of this instruction is the byte address of this first instruction; bytecodes for each method start at 0. The next step is to put the index on the stack:

```
1 iload_1 // Push local variable 1 (i) onto stack
```

Like the prior instruction, this 1-byte instruction is a short version of a more general instruction that takes 2 bytes to load a local variable onto the stack. The next instruction is to get the value from the array element:

```
2 iaload // Put array element (save[i]) onto stack
```

This 1-byte instruction checks the prior two operands, pops them off the stack, and then puts the value of the desired array element onto the new top of the stack. Next, we place `k` on the stack:

```
3 iload_2 // Push local variable 2 (k) onto stack
```

We are now ready for the *while* test:

```
4 if_icompne, Exit // Compare and exit if not equal
```

This 3-byte instruction compares the top two elements of the stack, pops them off the stack, and branches if they are not equal. We are finally prepared for the body of the loop:

```
7 iinc, 1, 1 // Increment local variable 1 by 1 (i+=1)
```

This unusual 3-byte instruction increments a local variable by 1 without using the operand stack, an optimization that again saves space. Finally, we return to the top of the loop with a 3-byte

branch:

```
10 go to 0 // Go to top of Loop (byte address 0)
```

Thus, the bytecode version takes seven instructions and 13 bytes, just over half the size of the RISC-V C code. (As before, we can optimize this code to branch less.)

Category	Operation	Java bytecode	Size (bits)	RISC-V instr.	Meaning
Arithmetic	add	iadd	8	add	NOS=TOS+NOS; pop
	subtract	isub	8	sub	NOS=TOS-NOS; pop
	increment	iinc I8a I8b	8	addi	Frame[I8a]= Frame[I8a] + I8b
Data transfer	load local integer/address	iload I8/aload I 8	16	ld	TOS=Frame[I8]
	load local integer/addres s	iload_/aload_{0,1,2,3 }	8	ld	TOS=Frame[{0,1,2,3}]
	store local integer/addres s	istore I8/astore I 8	16	sd	Frame[I8]=TOS; pop
	load integer/address from a rray	iaload/aaloa d	8	ld	NOS=*NOS[TOS]; pop
	store integer/address into a rray	iastore/aastor e	8	sd	*NNOS[NOS]=TOS; pop2
	load half from array	saload	8	lh	NOS=*NOS[TOS]; pop
	store half into array	sastore	8	sh	*NNOS[NOS]=TOS; pop2
	load byte from array	baload	8	lb	NOS=*NOS[TOS]; pop
	store byte into array	bastore	8	sb	*NNOS[NOS]=TOS; pop2
	load immediat e	bipush I8 , sipush I1 6	16, 24	addi	push; TOS=I8 or I16
	load immediat e	iconst_{-1,0,1,2,3,4,5 }	8	addi	push; TOS={-1,0,1,2,3,4,5}
Logical	and	iand	8	and	NOS=TOS&NOS; pop
	or	ior	8	or	NOS=TOS|NOS; pop
	shift left	ishl	8	sll	NOS=NOS << TOS; pop
	shift right	iushr	8	srl	NOS=NOS >> TOS; pop
Conditional branch	branch on equa l	if_icompeq I1 6	24	beq	if TOS == NOS , go to I16; pop2
	branch on not equa l	if_icompne I1 6	24	bne	if TOS != NOS , go to I16; pop2
	compare	if_icomp{lt,le,gt,ge} I16	24	blt/bge	if TOS {<,<=,>,>=} NOS, go to I16; pop2
Unconditional jump	jump	goto I16	24	jal	go to I16
	return	ret, ireturn	8	jalr	
	jump to subroutine	jsr I16	24	jal	go to I16; push; TOS=PC+3
Stack management	remove from stack	pop, pop2	8		pop, pop2
	duplicate on stack	dup	8		push; TOS=NOS
	swap top 2 positions on stack	swap	8		T=NOS; NOS=TOS; TOS=T
Safety check	check for null reference	fnull I16, ifnotnull I16	24		if TOS {==,!=} null, go to I16
	get length of array	arraylength	8		push; TOS = length of array
	check if object a type	instanceof I16	24		TOS = 1 if TOS matches type of Const[I16]; TOS = 0 otherwise
Invocation	invoke metho d	invokevirtual I16	24		Invoke method in Const[I16] , dispatching on type
Allocatio n	create new class instanc e	new I16	24		Allocate object type Const[I16] on heap
	create new array	newarray I16	24		Allocate array type Const[I16] on heap

FIGURE E2.15.8 Java bytecode architecture versus RISC-V.

Although many bytecodes are simple, those in the last half-dozen rows above are complex and specific to Java. Bytecodes are one to five bytes in length, hence their name. The Java mnemonics uses the prefix `i` for 32-bit integer, `a` for reference (address), `s` for 16-bit integers (short), and `b` for 8-bit bytes. We use `I8` for an 8-bit constant and `I16` for a 16-bit constant. RISC-V uses registers for operands, but the JVM uses a stack. The compiler knows the maximum size of the operand stack for each method and simply allocates space for it in the current frame. Here is the notation in the

Meaning column: `TOS`: top of stack; `NOS`: next position below `TOS`; `NNOS`: next position below `NOS`; `pop`: remove `TOS`; `pop2`: remove `TOS` and `NOS`; and `push`: add a position to the stack. `*NOS` and `*NNOS` mean access the memory location pointed to by the address in the stack at those positions. `Const[]` refers to the runtime constant pool of a class created by the JVM, and `Frame[]` refers to the variables of the local method frame. The missing Java bytecodes from Figure e2.1 are a few arithmetic and logical operators, some tricky stack management, compares to 0 and branch, support for branch tables, type conversions, more variations of the complex, Java-specific instructions plus operations on floating-point data, 64-bit integers (longs), and 16-bit characters.

Compiling for Java

Since Java is derived from C and Java has the same built-in types as C, the assignment statement examples in Sections 2.2 to 2.6 are the same in Java as they are in C. The same is true for the *if* statement example in Section 2.7.

The Java version of the *while* loop is different, however. The designers of C leave it up to the programmers to be sure that their code does not exceed the array bounds. The designers of Java wanted to catch array bound bugs, and thus require the compiler to check for such violations. To check bounds, the compiler needs to know what they are. Java includes an extra doubleword in every array that holds the upper bound. The lower bound is defined as 0.

Compiling a *while* Loop in Java

Example

Modify the RISC-V code for the *while* loop on page 95 to include the array bounds checks that are required by Java. Assume that the length of the array is located just before the first element of the array.

Answer

Let's assume that Java arrays reserved the first two doublewords of arrays before the data start. We'll see the use of the first doubleword soon, but the second doubleword has the array length. Before we enter the loop, let's load the length of the array into a temporary register:

```
ld x5, 8(x25) // Temp reg x5 = length of array save
```

Before we multiply `i` by 8, we must test to see if it's less than 0 or greater than the last element of the array. The first step is to check if `i` is less than 0:

```
Loop: blt x22, x0, IndexOutOfBounds // if i<0, goto Error
```

Since the array starts at 0, the index of the last array element is one less than the length of the array. Thus, the test of the upper array bound is to be sure that `i` is less than the length of the array. Thus, the second step is to branch to an error if it's greater than or equal to `length`.

```
bge x22, x5, IndexOutOfBounds //if i>=length, goto Error
```

The next two lines of the RISC-V *while* loop are unchanged from the C version:

```
slli x10, x22, 3 // Temp reg x10 = i * 8
add x10, x10, x25 // x10 = address of save[i]
```

We need to account for the first 16 bytes of an array that are reserved in Java. We do that by changing the address field of the load from 0 to 16:

```
ld x9, 16(x10) // Temp reg x9 = save[i]
```

The rest of the RISC-V code from the C *while* loop is fine as is:

```
bne x9, x24, Exit // go to Exit if save[i] ≠ k
addi x22, x22, 1 // i = i + 1
beq x0, x0, Loop // go to Loop
Exit:
```

(See the exercises for an optimization of this sequence.)

Invoking Methods in Java

The compiler picks the appropriate method depending on the type of object. In a few cases, it is unambiguous, and the method can be invoked with no more overhead than a C procedure. In general, however, the compiler knows only that a given variable contains a pointer to an object that belongs to some subtype of a general class. Since it doesn't know at compile time which subclass the object is,

and thus which method should be invoked, the compiler will generate code that first tests to be sure the pointer isn't null and then uses the code to load a pointer to a table with all the legal methods for that type. The first doubleword of the object has the method table address, which is why Java arrays reserve two doublewords. Let's say it's using the fifth method that was declared for that class. (The method order is the same for all subclasses.) The compiler then takes the fifth address from that table and invokes the method at that address.

The cost of object orientation in general is that method invocation takes five steps:

1. A conditional branch to be sure that the pointer to the object is valid;
2. A load to get the address of the table of available methods;
3. Another load to get the address of the proper method;
4. Placing a return address into the return register; and finally
5. A branch register to invoke the method.

A Sort Example in Java

Figure e2.15.9 shows the Java version of exchange sort. A simple difference is that there is no need to pass the length of the array as a separate parameter, since Java arrays include their length: `v.length` denotes the length of `v`.

```
public class sort {
   public static void sort (int[] v) {
      for (int i = 0; i < v.length; i += 1) {
         for (int j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
               swap(v, j);
         }
   }

   protected static void swap(int[] v, int k) {
      int temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
   }}
```

FIGURE E2.15.9 An initial Java procedure that

performs a sort on the array v.
Changes from Figures e2.24 and e2.26 are highlighted.

A more significant difference is that Java methods are prepended with keywords not found in the C procedures. The `sort` method is declared `public static` while `swap` is declared `protected static`. **Public** means that `sort` can be invoked from any other method, while **protected** means `swap` can only be called by other methods within the same **package** and from methods within derived classes. A **static method** is another name for a class method—methods that perform class-wide operations and do not apply to an individual object. Static methods are essentially the same as C procedures.

public

A Java keyword that allows a method to be invoked by any other method.

protected

A Java key word that restricts invocation of a method to other methods in that package.

package

Basically a directory that contains a group of related classes.

static method

A method that applies to the whole class rather than to an individual object. It is unrelated to static in C.

This straightforward translation from C into static methods means there is no ambiguity on method invocation, and so it can be just as efficient as C. It also is limited to sorting integers, which means a different sort has to be written for each data type.

To demonstrate the object orientation of Java, Figure e2.15.10 shows the new version with the changes highlighted. First, we declare `v` to be of the type `Comparable` and replace `v[j] > v[j + 1]` with an invocation of `compareTo`. By changing `v` to this new class, we

can use this code to `sort` many data types.

```
public class sort {
   public static void sort (Comparable[] v) {
      for (int i = 0; i < v.length; i += 1) {
        for (int j = i - 1; j >= 0 && v[j].compareTo(v[j + 1]);j -= 1) {

                    swap(v, j);
            }
   }

   protected static void swap(Comparable[] v, int k) {
      Comparable temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
   }}
public class Comparable {
      public int(compareTo (int x)
      { return value - x; }
      public int value;
}
```

FIGURE E2.15.10 A revised Java procedure that sorts on the array v that can take on more types. Changes from Figure e2.15.9 are highlighted.

The method `compareTo` compares two elements and returns a value greater than 0 if the parameter is larger than the object, 0 if it is equal, and a negative number if it is smaller than the object. These two changes generalize the code so it can sort integers, characters, strings, and so on, if there are subclasses of `Comparable` with each of these types and if there is a version of `compareTo` for each type. For pedagogic purposes, we redefine the class `Comparable` and the method `compareTo` here to compare integers. The actual definition of `Comparable` in the Java library is considerably different.

Starting from the RISC-V code that we generated for C, we show what changes we made to create the RISC-V code for Java.

For `swap`, the only significant differences are that we must check to be sure the object reference is not null and that each array reference is within bounds. The first test checks that the address in the first parameter is not zero:

```
swap: beq x10, x0, Error x10, NullPointer // if X0==0,goto
Error
```

Next, we load the length of v into a register and check that index

k is OK.

```
ld x5, 8(x10) // Temp reg x5 = length of array v
blt x11, x0, IndexOutOfBounds // if k < 0, goto Error
bge x11, x5, IndexOutOfBounds // if k >= length, goto Error
```

This check is followed by a check that k+1 is within bounds.

```
addi x6,x11,1 // Temp reg x6 = k+1
blt x6, x0, IndexOutOfBounds // if k+1 < 0, goto Error
bge x6, x5, IndexOutOfBounds // if k+1 >= length, goto Error
```

Figure e2.15.11 highlights the extra RISC-V instructions in swap that a Java compiler might produce. We again must adjust the offset in the load and store to account for two doublewords reserved for the method table and length.

Bounds check

```
swap: beq x10, x0, NullPointer          # If x10==0, goto Error
      ld x5, 8(x10)                     # Temp reg x5 = length of array v
      blt x11, x0, IndexOutOfBounds     # If k < 0, goto Error
      bge x11, x5, IndexOutOfBounds     # If k >= length, goto Error
      addi x6, x11, 1                   # Temp reg x6 = k+1
      blt x6, x0, IndexOutOfBounds      # If k+1 < 0, goto Error
      bge x6, x5, IndexOutOfBOunds      # If k+1 >= length, goto Error
```

Method body

```
      slli x11, x11, 3                  # reg X11 = k * 8
      add x11, x11, x10                 # reg X11 = v + (k * 8)

      ld x12, 0(x11)                    # reg x12 = v[k]
      ld x13, 8(x11)                    # reg x13 = v[k+1]

      ld x13, 0(x11)                    # v[k] = reg x13
      ld x12, 8(x11)                    # v[k+1] = reg x12
```

Method return

```
      jalr x0,0(x1)                     # return to calling routine
```

FIGURE E2.15.11 RISC-V assembly code of the procedure swap in Figure e2.24.

Figure e2.15.12 shows the method body for those new instructions for sort. (We can take the saving, restoring, and return from Figure e2.28.)

Method body			
Move parameters		mv x21, x10	# Copy parameter x10 into x21
Test ptr null		beq x10, x0, NullPointer	# If x10==0, goto Error
Get array length		ld x22, 8(x10)	# x22 = length of array v
Outer loop head	for1tst:	li x19, 0 bge x19, x22, exit1	# i = 0 # If i >= length, go to exit1
Inner loop head	for2tst:	addi x20, x19, -1 blt x20, x0, exit1	# j = i - 1 # If j < 0, goto exit2
Test if j too big		bge x20, x22, IndexOutOfBounds	# If j >= length, goto error

Get v[j]		slli x5, x20, 3
		add x5, x21, x5
		ld x6, 0(x5)
			# x5 = j * 8
			# x5 = v + (j * 8)
			# x6 = v[j]

Test if j+1 < 0 or too big		addi x7, x20, 1 blt x7, x0, IndexOutOfBounds bge x7, x22, IndexOutOfBounds	# x7 = j + 1 # If j + 1 < 0, goto Error # If j + 1 >= length, goto Error
Get v[j+1]		ld x7, 8(x5)	# x7 = v[j+1]
Load method table		ld x28, 0(x10)	# x28 = address of method table
Get method address		ld x28, 16(x28)	# x28 = address of third method
Pass parameters	for2tst:	mv x10, x6 mv x11, x7	# 1st parameter is v[j] # 2nd parameter is v[j+1]
Call method indirectly		jalr x1, 0(x28)	# Call compareTo
Test if should skip swap		ble x10, x0, exit2	# If result <= 0, skip swap
Pass parameters and call swap		mv x10, x21 mv x11, x20 jal x1, swap	# 1st parameter is v # 2nd parameter is j # Invoke swap routine (Figure 2.34)
Inner loop end		addi x20,x20,-1 j for2tst	# j -= 1 # Go to for2tst
Outer loop end	exit2:	addi x19,x19,1 j for1tst	# i += 1 # Go to for1tst

FIGURE E2.15.12 RISC-V assembly version of the method body of the Java version of `sort`. The new code is highlighted in this figure. We must still add the code to save and restore registers and the return from the RISC-V code found in Figure e2.27. To keep the code similar to that figure, we load `v.length` into `x22` instead of into a temporary register. To reduce the number of lines of code, we make the simplifying assumption that `compareTo` is a leaf procedure and we do not need to push registers to be saved on the stack.

The first test is again to make sure the pointer to `v` is not null:

```
beq x10, x0, Error // if x10==0,goto Error
```

Next, we load the length of the array (we use register `x22` to keep it similar to the code for the C version of sort):

```
ld x22, 8(x10) // x22 = length of array v
```

Now we must ensure that the index is within bounds. Since the first test of the inner loop is to test if `j` is negative, we can skip that initial bound test. That leaves the test for too big:

```
bge x20, x22, IndexOutOfBounds // if j > = length, goto Error
```

The code for testing `j + 1` is quite similar to the code for checking `k + 1` in `swap`, so we skip it here.

The key difference is the invocation of `compareTo`. We first load the address of the table of legal methods, which we assume is two

doublewords before the beginning of the array:

```
ld x28, 0(x10) // x28 = address of method table
```

Given the address of the method table for this object, we then get the desired method. Let's assume `compareTo` is the third method in the `Comparable` class. To pick the address of the third method, we load that address into a temporary register:

```
ld x28, 16(x28) // x28 = address of third method
```

We are now ready to call `compareTo`. The next step is to save the necessary registers on the stack. Fortunately, we don't need the temporary registers or argument registers after the method invocation, so there is nothing to save. Thus, we simply pass the parameters for `compareTo`:

```
mv x10, x6 // 1st parameter of compareTo is v[j]
mv x11, x7 // 2nd parameter of compareTo is v[j+1]
```

Then, we use the jump-and-link register to invoke compareTo:

```
jalr x1, 0(x28) // invoke compareTo, and save return address
in x1
```

The method returns, with `x10` determining which of the two elements is larger. If `x10 > 0`, then `v[j] >v[j+1]`, and we need to `swap`. Thus, to skip the `swap`, we need to test if `x10` ≤ `0`:

```
ble x10, x0, exit2 // go to exit2 if v[j] ≤ v[j+1]
```

The RISC-V code for `compareTo` is left as an exercise.

Hardware/Software Interface

The main changes for the Java versions of `sort` and `swap` are testing for null object references and index out-of-bounds errors, and the extra method invocation to give a more general compare. This method invocation is more expensive than a C procedure call, since it requires, a conditional branch, a pair of chained loads, and an indirect branch. As we see in Chapter 4, dependent loads and indirect branches can be relatively slow on modern processors. The increasing popularity of Java suggests that many programmers today are willing to leverage the high performance of modern processors to pay for error checking and code reuse.

Elaboration

Although we test each reference to j and $j + 1$ to be sure that these

indices are within bounds, an assembly language programmer might look at the code and reason as follows:

1. The inner *for* loop is only executed if $j \leq 0$ and since $j + 1 > j$, there is no need to test $j + 1$ to see if it is less than 0.
2. Since i takes on the values, 0, 1, 2, …, (data.length −1) and since j takes on the values i −1, i −2, …, 2, 1, 0, there is no need to test if $j \leq$ data.length since the largest value j can be is data.length −2.
3. Following the same reasoning, there is no need to test whether $j + 1 \leq$data.length since the largest value of j+1 is data.length −1.

There are coding tricks in Chapter 2 and superscalar execution in Chapter 4 that lower the effective cost of such bounds checking, but only high optimizing compilers can reason this way. Note that if the compiler inlined the swap method into sort, many checks would be unnecessary.

Elaboration

Look carefully at the code for swap in Figure e2.15.11. See anything wrong in the code, or at least in the explanation of how the code works? It implicitly assumes that each `Comparable` element in `v` is 8 bytes long. Surely, you need much more than 8 bytes for a complex subclass of `Comparable`, which could contain any number of fields. Surprisingly, this code does work, because an important property of Java's semantics forces the use of the same, small representation for all variables, fields, and array elements that belong to `Comparable` or its subclasses.

Java types are divided into *primitive types*—the predefined types for numbers, characters, and Booleans—and *reference types*—the built-in classes like String, user-defined classes, and arrays. Values of reference types are pointers (also called *references*) to anonymous objects that are themselves allocated in the heap. For the programmer, this means that assigning one variable to another does not create a new object, but instead makes both variables refer to the same object. Because these objects are anonymous, and programs therefore have no way to refer to them directly, a program must use indirection through a variable to read or write any objects' fields (variables). Thus, because the data structure allocated for the array `v` consists entirely of pointers, it is safe to assume they are all the same size, and the same swapping code

works for all of `Comparable`'s subtypes.

To write sorting and swapping functions for arrays of primitive types requires that we write new versions of the functions, one for each type. This replication is for two reasons. First, primitive type values do not include the references to dispatching tables that we used on `Comparables` to determine at runtime how to compare values. Second, primitive values come in different sizes: 1, 2, 4, or 8 bytes.

The pervasive use of pointers in Java is elegant in its consistency, with the penalty being a level of indirection and a requirement that objects be allocated on the heap. Furthermore, in any language where the lifetimes of the heap-allocated anonymous objects are independent of the lifetimes of the named variables, fields, and array elements that reference them, programmers must deal with the problem of deciding when it is safe to deallocate heap-allocated storage. Java's designers chose to use garbage collection. Of course, use of garbage collection rather than explicit user memory management also improves program safety.

C++ provides an interesting contrast. Although programmers can write essentially the same pointer-manipulating solution in C++, there is another option. In C++, programmers can elect to forgo the level of indirection and directly manipulate an array of objects, rather than an array of pointers to those objects. To do so, C++ programmers would typically use the template capability, which allows a class or function to be parameterized by the *type* of data on which it acts. Templates, however, are compiled using the equivalent of macro expansion. That is, if we declared an instance of sort capable of sorting types X and Y, C++ would create two copies of the code for the class: one for sort<X> and one for sort<Y>, each specialized accordingly. This solution increases code size in exchange for making comparison faster (since the function calls would not be indirect, and might even be subject to inline expansion). Of course, the speed advantage would be canceled if swapping the objects required moving large amounts of data instead of just single pointers. As always, the best design depends on the details of the problem.

2.16 Real Stuff: MIPS Instructions

The instruction set most similar to RISC-V, MIPS, also originated in academia, but is now owned by Imagination Technologies. MIPS and RISC-V share the same design philosophy, despite MIPS being 25 years more senior than RISC-V. The good news is that if you know RISC-V, it will be very easy to pick up MIPS. To show their similarity, Figure 2.29 compares instruction formats for RISC-V and MIPS.

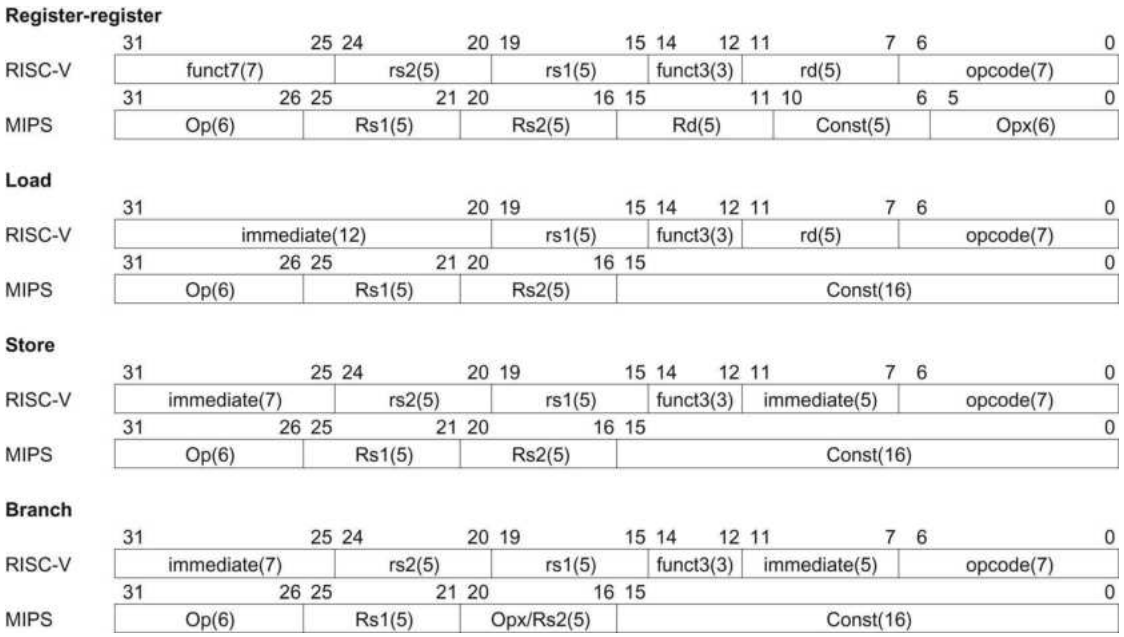


FIGURE 2.29 Instruction formats of RISC-V and MIPS.
The similarities result in part from both instruction sets having 32 registers.

The MIPS ISA has both 32-bit address and 64-bit address versions, sensibly called MIPS-32 and MIPS-64. These instruction sets are virtually identical except for the larger address size needing 64-bit registers instead of 32-bit registers. Here are the common features between RISC-V and MIPS:

- All instructions are 32 bits wide for both architectures.
- Both have 32 general-purpose registers, with one register being hardwired to 0.
- The only way to access memory is via load and store instructions on both architectures.
- Unlike some architectures, there are no instructions that can load or store many registers in MIPS or RISC-V.
- Both have instructions that branch if a register is equal to zero and branch if a register is not equal to zero.
- Both sets of addressing modes work for all word sizes.

One of the main differences between RISC-V and MIPS is for conditional branches other than equal or not equal. Whereas RISC-V simply provides branch instructions to compare two registers, MIPS relies on a comparison instruction that sets a register to 0 or 1 depending on whether the comparison is true. Programmers then

follow that comparison instruction with a branch on equal to or not equal to zero depending on the desired outcome of the comparison. Keeping with its minimalist philosophy, MIPS only performs less than comparisons, leaving it up to the programmer to switch order of operands or to switch the condition being tested by the branch to get all the desired outcomes. MIPS has both signed and unsigned versions of the set on less than instructions: `slt` and `sltu`.

When we look beyond the core instructions that are most commonly used, the other main difference is that the full MIPS is a much larger instruction set than RISC-V, as we shall see in Section 2.18.

2.17 Real Stuff: x86 Instructions

Beauty is altogether in the eye of the beholder.

Margaret Wolfe Hungerford, Molly Bawn, 1877

Designers of instruction sets sometimes provide more powerful operations than those found in RISC-V and MIPS. The goal is generally to reduce the number of instructions executed by a program. The danger is that this reduction can occur at the cost of simplicity, increasing the time a program takes to execute because the instructions are slower. This slowness may be the result of a slower clock cycle time or of requiring more clock cycles than a simpler sequence.

The path toward operation complexity is thus fraught with peril. Section 2.19 demonstrates the pitfalls of complexity.

Evolution of the Intel x86

RISC-V and MIPS were the vision of single groups working at the same time; the pieces of these architectures fit nicely together. Such is not the case for the x86; it is the product of several independent groups who evolved the architecture over almost 40 years, adding new features to the original instruction set as someone might add clothing to a packed bag. Here are important x86 milestones.

general-purpose register (GPR)

A register that can be used for addresses or for data with virtually any instruction.

- **1978**: The Intel 8086 architecture was announced as an assembly language-compatible extension of the then-successful Intel 8080, an 8-bit microprocessor. The 8086 is a 16-bit architecture, with all internal registers 16 bits wide. Unlike RISC-V, the registers have dedicated uses, and hence the 8086 is not considered a **general-purpose register (GPR)** architecture.
- **1980**: The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Instead of using registers, it relies on a stack (see  **Section 2.21** and Section 3.7).
- **1982**: The 80286 extended the 8086 architecture by increasing the address space to 24 bits, by creating an elaborate memory-mapping and protection model (see Chapter 5), and by adding a few instructions to round out the instruction set and to manipulate the protection model.
- **1985**: The 80386 extended the 80286 architecture to 32 bits. In addition to a 32-bit architecture with 32-bit registers and a 32-bit address space, the 80386 added new addressing modes and additional operations. The expanded instructions make the 80386 nearly a general-purpose register machine. The 80386 also added paging support in addition to segmented addressing (see Chapter 5). Like the 80286, the 80386 has a mode to execute 8086 programs without change.
- **1989–95**: The subsequent 80486 in 1989, Pentium in 1992, and Pentium Pro in 1995 were aimed at higher performance, with only four instructions added to the user-visible instruction set: three to help with multiprocessing (see Chapter 6) and a conditional move instruction.
- **1997**: After the Pentium and Pentium Pro were shipping, Intel announced that it would expand the Pentium and the Pentium Pro architectures with MMX (*Multi Media Extensions*). This new set of 57 instructions uses the floating-point stack to accelerate multimedia and communication applications. MMX instructions typically operate on multiple short data elements at a time, in the

tradition of *single instruction, multiple data* (SIMD) architectures (see Chapter 6). Pentium II did not introduce any new instructions.

- **1999**: Intel added another 70 instructions, labeled SSE (*Streaming SIMD Extensions*) as part of Pentium III. The primary changes were to add eight separate registers, double their width to 128 bits, and add a single precision floating-point data type. Hence, four 32-bit floating-point operations can be performed in parallel. To improve memory performance, SSE includes cache prefetch instructions plus streaming store instructions that bypass the caches and write directly to memory.
- **2001**: Intel added yet another 144 instructions, this time labeled SSE2. The new data type is double precision arithmetic, which allows pairs of 64-bit floating-point operations in parallel. Almost all of these 144 instructions are versions of existing MMX and SSE instructions that operate on 64 bits of data in parallel. Not only does this change enable more multimedia operations; it gives the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE registers as floating-point registers like those found in other computers. This change boosted the floating-point performance of the Pentium 4, the first microprocessor to include SSE2 instructions.
- **2003**: A company other than Intel enhanced the x86 architecture this time. AMD announced a set of architectural extensions to increase the address space from 32 to 64 bits. Similar to the transition from a 16- to 32-bit address space in 1985 with the 80386, AMD64 widens all registers to 64 bits. It also increases the number of registers to 16 and increases the number of 128-bit SSE registers to 16. The primary ISA change comes from adding a new mode called *long mode* that redefines the execution of all x86 instructions with 64-bit addresses and data. To address the larger number of registers, it adds a new prefix to instructions. Depending how you count, long mode also adds four to 10 new instructions and drops 27 old ones. PC-relative data addressing is another extension. AMD64 still has a mode that is identical to x86 (*legacy mode*) plus a mode that restricts user programs to x86 but allows operating systems to use AMD64 (*compatibility mode*). These modes allow a more graceful transition to 64-bit

addressing than the HP/Intel IA-64 architecture.

- **2004**: Intel capitulates and embraces AMD64, relabeling it *Extended Memory 64 Technology* (EM64T). The major difference is that Intel added a 128-bit atomic compare and swap instruction, which probably should have been included in AMD64. At the same time, Intel announced another generation of media extensions. SSE3 adds 13 instructions to support complex arithmetic, graphics operations on arrays of structures, video encoding, floating-point conversion, and thread synchronization (see Section 2.11). AMD added SSE3 in subsequent chips and the missing atomic swap instruction to AMD64 to maintain binary compatibility with Intel.
- **2006**: Intel announces 54 new instructions as part of the SSE4 instruction set extensions. These extensions perform tweaks like sum of absolute differences, dot products for arrays of structures, sign or zero extension of narrow data to wider sizes, population count, and so on. They also added support for virtual machines (see Chapter 5).
- **2007**: AMD announces 170 instructions as part of SSE5, including 46 instructions of the base instruction set that adds three operand instructions like RISC-V.
- **2011**: Intel ships the Advanced Vector Extension that expands the SSE register width from 128 to 256 bits, thereby redefining about 250 instructions and adding 128 new instructions.

This history illustrates the impact of the "golden handcuffs" of compatibility on the x86, as the existing software base at each step was too important to jeopardize with significant architectural changes.

Whatever the artistic failures of the x86, keep in mind that this instruction set largely drove the PC generation of computers and still dominates the Cloud portion of the post-PC era. Manufacturing 350M x86 chips per year may seem small compared to 14 billion ARM chips, but many companies would love to control such a market. Nevertheless, this checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

Brace yourself for what you are about to see! Do *not* try to read this section with the care you would need to write x86 programs; the goal instead is to give you familiarity with the strengths and weaknesses of the world's most popular desktop architecture.

Rather than show the entire 16-bit, 32-bit, and 64-bit instruction set, in this section we concentrate on the 32-bit subset that originated with the 80386. We start our explanation with the registers and addressing modes, move on to the integer operations, and conclude with an examination of instruction encoding.

x86 Registers and Data Addressing Modes

The registers of the 80386 show the evolution of the instruction set (Figure 2.30). The 80386 extended all 16-bit registers (except the segment registers) to 32 bits, prefixing an *E* to their name to indicate the 32-bit version. We'll refer to them generically as GPRs (*general-purpose registers*). The 80386 contains only eight GPRs. This means RISC-V and MIPS programs can use four times as many.

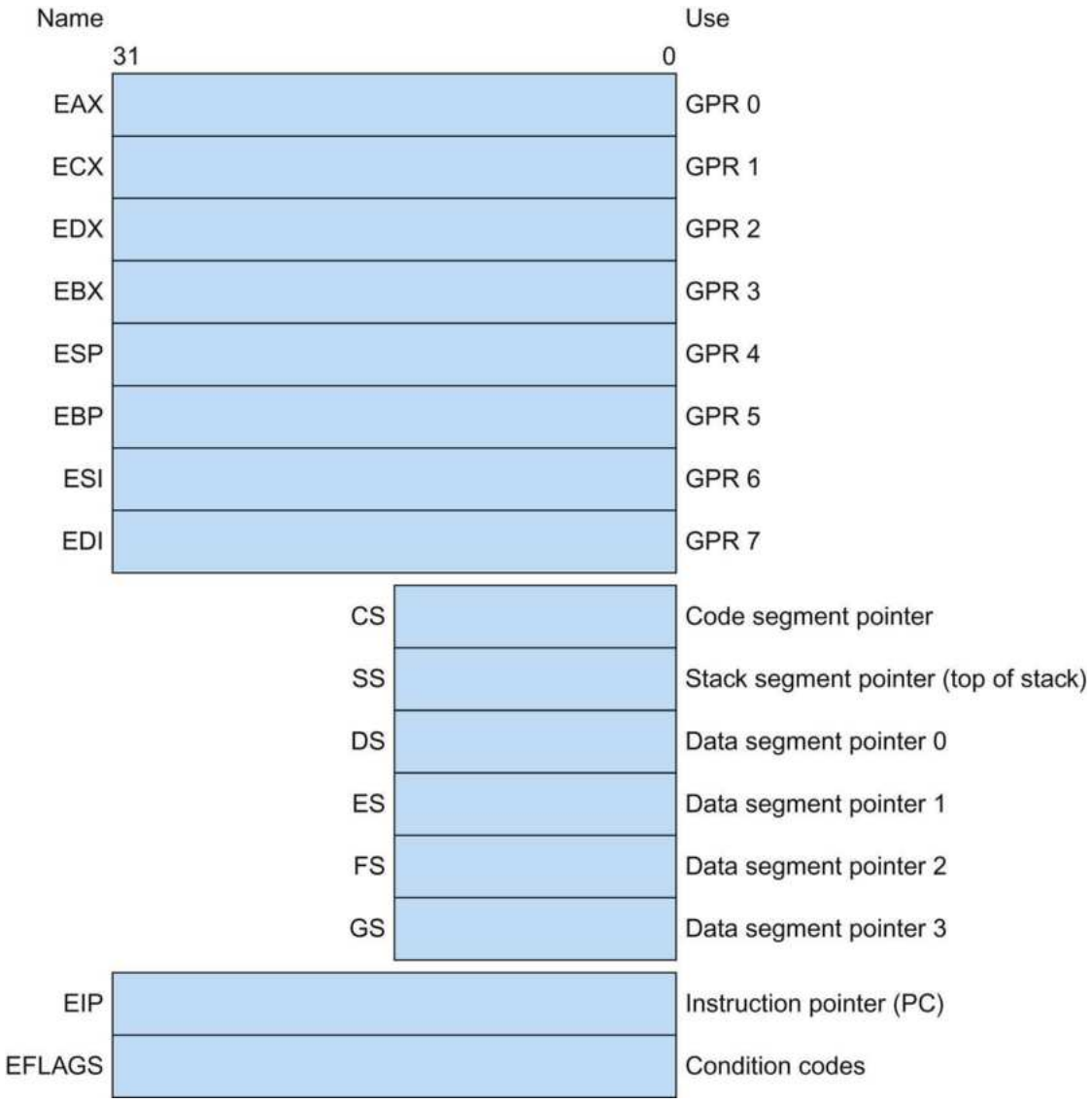


FIGURE 2.30 The 80386 register set.
Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers.

Figure 2.31 shows the arithmetic, logical, and data transfer instructions are two-operand instructions. There are two important differences here. The x86 arithmetic and logical instructions must have one operand act as both a source and a destination; RISC-V and MIPS allow separate registers for source and destination. This restriction puts more pressure on the limited registers, since one source register must be modified. The second important difference is that one of the operands can be in memory. Thus, virtually any instruction may have one operand in memory, unlike RISC-V and MIPS.

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

FIGURE 2.31 Instruction types for the arithmetic, logical, and data transfer instructions.
The x86 allows the combinations shown. The only restriction is the absence of a memory-memory mode. Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure 2.33 (not EIP or EFLAGS).

Data memory-addressing modes, described in detail below, offer two sizes of addresses within the instruction. These so-called *displacements* can be 8 bits or 32 bits.

Although a memory operand can use any addressing mode, there are restrictions on which *registers* can be used in a mode. Figure 2.32 shows the x86 addressing modes and which GPRs cannot be used with each mode, as well as how to get the same effect using RISC-V instructions.

Mode	Description	Register restrictions	RISC-V equivalent
Register indirect	Address is in a register.	Not ESP or EBP	`ld x10, 0(x11)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	Not ESP	`ld x10, 40(x11)`

Base plus scaled index	The address is Base + (2^{Scale} × Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`slli x12, x12, 3`
			`add x11, x11, x12`
			`ld x10, 0(x11)`

Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} × Index) + Displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`slli x12, x12, 3`
			`add x11, x11, x12`
			`ld x10, 40(x11)`

FIGURE 2.32 x86 32-bit addressing modes with register restrictions and the equivalent RISC-V code.
The Base plus Scaled Index addressing mode, not found in RISC-V or MIPS, is included to avoid the multiplies by 8 (scale factor of 3) to turn an index in a register into a byte address (see Figures 2.26 and 2.28). A scale factor of 1 is used for 16-bit data, and a scale factor of 2 for 32-bit data. A scale factor of 0 means the address is not scaled. If the displacement is

longer than 12 bits in the second or fourth modes, then the RISC-V equivalent mode would need more instructions, usually a `lui` to load bits 12 through 31 of the displacement, followed by an `add` to sum these bits with the base register. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

x86 Integer Operations

The 8086 provides support for both 8-bit (*byte*) and 16-bit (*word*) data types. The 80386 adds 32-bit addresses and data (*doublewords*) in the x86. (AMD64 adds 64-bit addresses and data, called *quad words*; we'll stick to the 80386 in this section.) The data type distinctions apply to register operations as well as memory accesses.

Almost every operation works on both 8-bit data and on one longer data size. That size is determined by the mode and is either 16 bits or 32 bits.

Clearly, some programs want to operate on data of all three sizes, so the 80386 architects provided a convenient way to specify each version without expanding code size significantly. They decided that either 16-bit or 32-bit data dominate most programs, and so it made sense to be able to set a default large size. This default data size is set by a bit in the code segment register. To override the default data size, an 8-bit *prefix* is attached to the instruction to tell the machine to use the other large size for this instruction.

The prefix solution was borrowed from the 8086, which allows multiple prefixes to modify instruction behavior. The three original prefixes override the default segment register, lock the bus to support synchronization (see Section 2.11), or repeat the following instruction until the register ECX counts down to 0. This last prefix was intended to be paired with a byte move instruction to move a variable number of bytes. The 80386 also added a prefix to override the default address size.

The x86 integer operations can be divided into four major classes:
1. Data movement instructions, including move, push, and pop.
2. Arithmetic and logic instructions, including test, integer, and

decimal arithmetic operations.
3. Control flow, including conditional branches, unconditional branches, calls, and returns.
4. String instructions, including string move and string compare.

The first two categories are unremarkable, except that the arithmetic and logic instruction operations allow the destination to be either a register or a memory location. Figure 2.33 shows some typical x86 instructions and their functions.

Instruction	Function
je name	if equal(condition code) {EIP=name}; EIP-128 <= name < EIP+128
jmp name	EIP=name
call name	SP=SP-4; M[SP]=EIP+5; EIP=name;
movw EBX,[EDI+45]	EBX=M[EDI+45]
push ESI	SP=SP-4; M[SP]=ESI
pop EDI	EDI=M[SP]; SP=SP+4
add EAX,#6765	EAX= EAX+6765
test EDX,#42	Set condition code (flags) with EDX and 42
movsl	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

FIGURE 2.33 Some typical x86 instructions and their functions.
A list of frequent operations appears in Figure 2.37. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

Conditional branches on the x86 are based on *condition codes* or *flags*. Condition codes are set as a side effect of an operation; most are used to compare the value of a result to 0. Branches then test the condition codes. PC-relative branch addresses must be specified in the number of bytes, since unlike RISC-V and MIPS, 80386 instructions have no alignment restriction.

String instructions are part of the 8080 ancestry of the x86 and are not commonly executed in most programs. They are often slower than equivalent software routines (see the *Fallacy* on page 157).

Figure 2.34 lists some of the integer x86 instructions. Many of the instructions are available in both byte and word formats.

Instruction	Meaning
Control	**Conditional and unconditional branches**
jnz, jz	Jump if condition to EIP + 8-bit offset; JNE (for JNZ), JE (for JZ) are alternative names
jmp	Unconditional jump—8-bit or 16-bit offset
call	Subroutine call—16-bit offset; return address pushed onto stack
ret	Pops return address from stack and jumps to it
loop	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX≠0
Data transfer	**Move data between registers or between register and memory**
move	Move between two registers or between register and memory
push, pop	Push source operand on stack; pop operand from stack top to a register
les	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
add, sub	Add source to destination; subtract source from destination; register-memory format
cmp	Compare source and destination; register-memory format
shl, shr, rcr	Shift left; shift logical right; rotate right with carry condition code as fill
cbw	Convert byte in eight rightmost bits of EAX to 16-bit word in right of EAX
test	Logical AND of source and destination sets condition codes
inc, dec	Increment destination, decrement destination
or, xor	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
movs	Copies from string source to destination by incrementing ESI and EDI; may be repeated
lods	Loads a byte, word, or doubleword of a string into the EAX register

FIGURE 2.34 Some typical operations on the x86. Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

x86 Instruction Encoding

Saving the worst for last, the encoding of instructions in the 80386 is complex, with many different instruction formats. Instructions for the 80386 may vary from 1 byte, when there is only one operand, up to 15 bytes.

Figure 2.35 shows the instruction format for several of the example instructions in Figure 2.33. The opcode byte usually contains a bit saying whether the operand is 8 bits or 32 bits. For some instructions, the opcode may include the addressing mode and the register; this is true in many instructions that have the form "register=register op immediate." Other instructions use a "postbyte" or extra opcode byte, labeled "mod, reg, r/m," which

contains the addressing mode information. This postbyte is used for many of the instructions that address memory. The base plus scaled index mode uses a second postbyte, labeled “sc, index, base.”

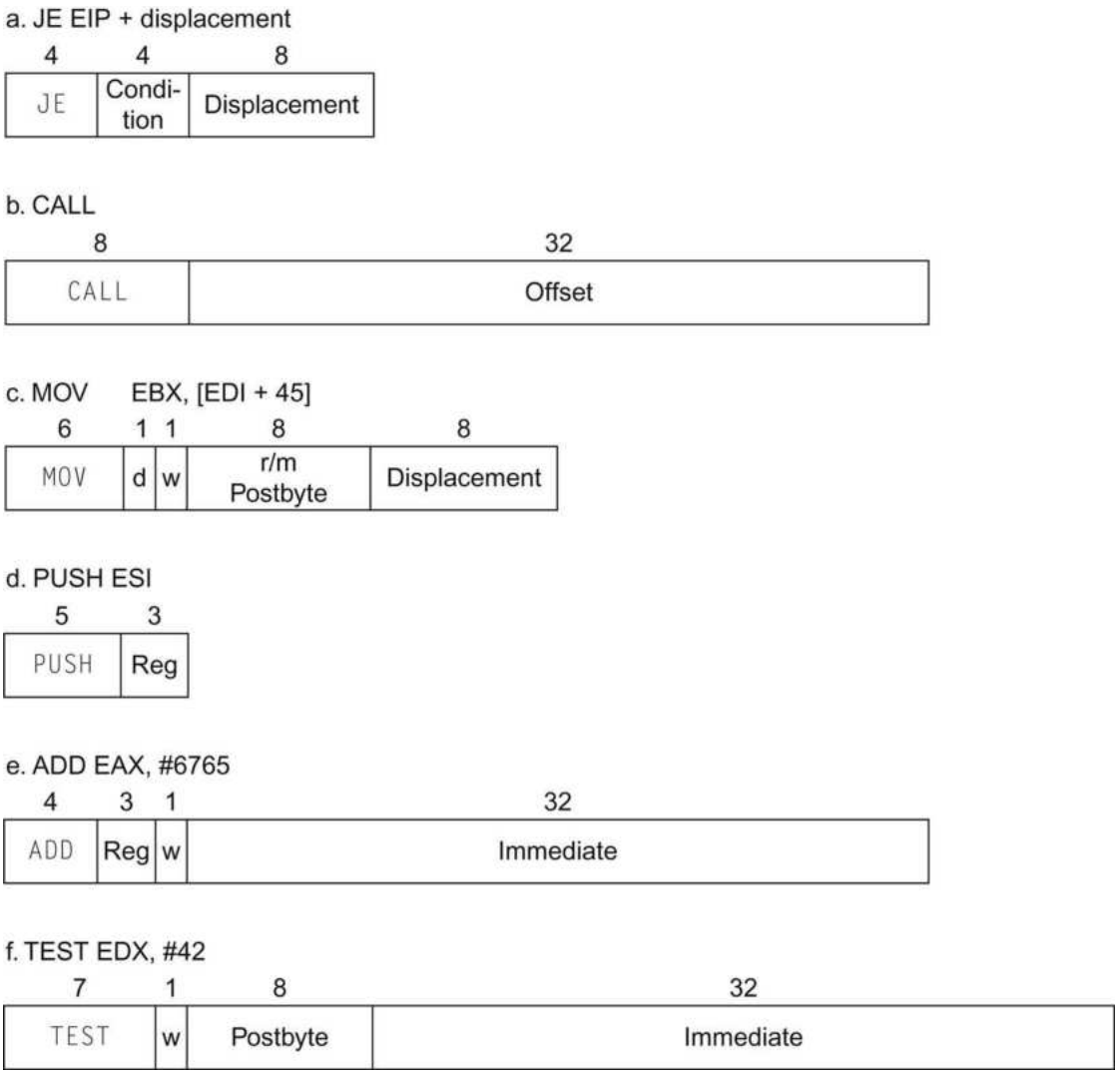


FIGURE 2.35 Typical x86 instruction formats.

Figure 2.39 shows the encoding of the postbyte. Many instructions contain the 1-bit field *w*, which says whether the operation is a byte or a doubleword. The *d* field in `MOV` is used in instructions that may move to or from memory and shows the direction of the move. The `ADD` instruction requires 32 bits for the immediate field, because in 32-bit mode, the immediates are either 8 bits or 32 bits. The immediate field in the `TEST` is 32 bits long because there is no 8-bit immediate for test in 32-bit mode. Overall, instructions may vary from 1 to 15 bytes in length. The long length comes from extra 1-byte prefixes, having both a 4-byte immediate and a 4-byte displacement address, using an opcode of 2 bytes, and using the scaled index mode specifier, which adds another byte.

Figure 2.36 shows the encoding of the two postbyte address specifiers for both 16-bit and 32-bit modes. Unfortunately, to understand fully which registers and which addressing modes are available, you need to see the encoding of all addressing modes and sometimes even the encoding of the instructions.

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

FIGURE 2.36 The encoding of the first address specifier of the x86: *mod*, reg, *r/m*. The first four columns show the encoding of the 3-bit reg field, which depends on the *w* bit from the opcode and whether the machine is in 16-bit mode (8086) or 32-bit mode (80386). The remaining columns explain the *mod* and *r/m* fields. The meaning of the 3-bit *r/m* field depends on the value in the 2-bit *mod* field and the address size. Basically, the registers used in the address calculation are listed in the sixth and seventh columns, under *mod* =0, with *mod* =1 adding an 8-bit displacement and *mod* =2 adding a 16-bit or 32-bit displacement, depending on the address mode. The exceptions are 1) *r/m* =6 when *mod* =1 or *mod* =2 in 16-bit mode selects BP plus the displacement; 2) *r/m* =5 when *mod* =1 or *mod* =2 in 32-bit mode selects EBP plus displacement; and 3) *r/m* =4 in 32-bit mode when *mod* does not equal 3, where (sib) means use the scaled index mode shown in Figure 2.35. When *mod* =3, the *r/m* field indicates a register, using the same encoding as the reg field combined with the *w* bit.

x86 Conclusion

Intel had a 16-bit microprocessor two years before its competitors' more elegant architectures, such as the Motorola 68000, and this head start led to the selection of the 8086 as the CPU for the IBM

PC. Intel engineers generally acknowledge that the x86 is more difficult to build than computers like RISC-V and MIPS, but the large market meant in the PC era that AMD and Intel could afford more resources to help overcome the added complexity. What the x86 lacks in style, it rectifies with market size, making it beautiful from the right perspective.

Its saving grace is that the most frequently used x86 architectural components are not too difficult to implement, as AMD and Intel have demonstrated by rapidly improving performance of integer programs since 1978. To get that performance, compilers must avoid the portions of the architecture that are hard to implement fast.

In the post-PC era, however, despite considerable architectural and manufacturing expertise, x86 has not yet been competitive in the personal mobile device.

2.18 Real Stuff: The Rest of the RISC-V Instruction Set

With the goal of making an instruction set architecture suitable for a wide variety of computers, the RISC-V architects partitioned the instruction set into a *base architecture* and several *extensions*. Each is named with a letter of the alphabet, and the base architecture is named `I` for *integer*. The base architecture has few instructions relative to other popular instruction sets today; indeed, this chapter has already covered nearly all of them. This section rounds out the base architecture, then describes the five standard extensions.

Figure 2.37 lists the remaining instructions in the base RISC-V architecture. The first instruction, `auipc`, is used for PC-relative memory addressing. Like the `lui` instruction, it holds a 20-bit constant that corresponds to bits 12 through 31 of an integer. `auipc`'s effect is to add this number to the PC and write the sum to a register. Combined with an instruction like `addi`, it is possible to address any byte of memory within 4 GiB of the PC. This feature is useful for *position-independent code*, which can execute correctly no matter where in memory it is loaded. It is most frequently used in dynamically linked libraries.

Additional Instructions in RISC-V Base Architecture

Instruction	Name	Format	Description
Add upper immediate to PC	`auipc`	U	Add 20-bit upper immediate to PC; write sum to register
Set if less than	`slt`	R	Compare registers; write Boolean result to register
Set if less than, unsigned	`sltu`	R	Compare registers; write Boolean result to register
Set if less than, immediate	`slti`	I	Compare registers; write Boolean result to register
Set if less than immediate, unsigned	`sltiu`	I	Compare registers; write Boolean result to register
Add word	`addw`	R	Add 32-bit numbers
Subtract word	`subw`	R	Subtract 32-bit numbers
Add word immediate	`addiw`	I	Add constant to 32-bit number
Shift left logical word	`sllw`	R	Shift 32-bit number left by register
Shift right logical word	`srlw`	R	Shift 32-bit number right by register
Shift right arithmetic word	`sraw`	R	Shift 32-bit number right arithmetically by register
Shift left logical word immedate	`slliw`	I	Shift 32-bit number left by immediate
Shift right logical word immediate	`srliw`	I	Shift 32-bit number right by immediate
Shift right arithmetic word immediate	`sraiw`	I	Shift 32-bit number right arithmetically by immediate

FIGURE 2.37 The remaining 14 instructions in the base RISC-V instruction set architecture.

The next four instructions compare two integers, then write the Boolean result of the comparison to a register. `slt` and `sltu` compare two registers as signed and unsigned numbers, respectively, then write 1 to a register if the first value is less than the second value, or 0 otherwise. `slti` and `sltiu` perform the same comparisons, but with an immediate for the second operand.

The remaining instructions should all look familiar, as their names are the same as other instructions discussed in this chapter, but with the letter `w`, short for *word*, appended. These instructions perform the same operation as the similarly named ones we've discussed, except these only operate on the lower 32 bits of their operands, ignoring bits 32 through 63. Additionally, they produce sign-extended 32-bit results: that is, bits 32 through 63 are all the same as bit 31. The RISC-V architects included these `w` instructions because operations on 32-bit numbers remain very common on computers with 64-bit addresses. The main reason is that the popular data type `int` remains 32 bits in Java and in most implementations of the C language.

That's it for the base architecture! Figure 2.38 lists the five standard extensions. The first, M, adds instructions to multiply and divide integers. Chapter 3 will introduce several instructions in the M extension.

RISC-V Base and Extensions

Mnemonic	Description	Insn. Count
I	Base architecture	51
M	Integer multiply/divide	13
A	Atomic operations	22
F	Single-precision floating point	30
D	Double-precision floating point	32
C	Compressed instructions	36

FIGURE 2.38 The RISC-V instruction set architecture is divided into the base ISA, named I, and five standard extensions, M, A, F, D, and C.

The second extension, A, supports atomic memory operations for multiprocessor synchronization. The load-reserved (`lr.d`) and store-conditional (`sc.d`) instructions introduced in Section 2.11 are members of the A extension. Also included are versions that operate on 32-bit words (`lr.w` and `sc.w`). The remaining 18 instructions are optimizations of common synchronization patterns, like atomic exchange and atomic addition, but do not add any additional functionality over load-reserved and store-conditional.

The third and fourth extensions, F and D, provide operations on floating-point numbers, which are described in Chapter 3.

The last extension, C, provides no new functionality at all. Rather, it takes the most popular RISC-V instructions, like `addi`, and provides equivalent instructions that are only 16 bits in length, rather than 32. It thereby allows programs to be expressed in fewer bytes, which can reduce cost and, as we will see in Chapter 5, can improve performance. To fit in 16 bits, the new instructions have restrictions on their operands: for example, some instructions can only access some of the 32 registers, and the immediate fields are narrower.

Taken together, the RISC-V base and extensions have 184 instructions, plus 13 system instructions that will be introduced at the end of Chapter 5.

2.19 Fallacies and Pitfalls

Fallacy: More powerful instructions mean higher performance.

Part of the power of the Intel x86 is the prefixes that can modify the execution of the following instruction. One prefix can repeat the subsequent instruction until a counter steps down to 0. Thus, to move data in memory, it would seem that the natural instruction sequence is to use move with the repeat prefix to perform 32-bit memory-to-memory moves.

An alternative method, which uses the standard instructions found in all computers, is to load the data into the registers and then store the registers back to memory. This second version of this program, with the code replicated to reduce loop overhead, copies at about 1.5 times as fast. A third version, which uses the larger floating-point registers instead of the integer registers of the x86, copies at about 2.0 times as fast as the complex move instruction.

Fallacy: Write in assembly language to obtain the highest performance.

At one time compilers for programming languages produced naïve instruction sequences; the increasing sophistication of compilers means the gap between compiled code and code produced by hand is closing fast. In fact, to compete with current compilers, the assembly language programmer needs to understand the concepts in Chapters 4 and 5 thoroughly (processor pipelining and memory hierarchy).

This battle between compilers and assembly language coders is another situation in which humans are losing ground. For example, C offers the programmer a chance to give a hint to the compiler about which variables to keep in registers versus spilled to memory. When compilers were poor at register allocation, such hints were vital to performance. In fact, some old C textbooks spent a fair amount of time giving examples that effectively use register hints. Today's C compilers generally ignore these hints, because the compiler does a better job at allocation than the programmer does.

Even *if* writing by hand resulted in faster code, the dangers of writing in assembly language are the protracted time spent coding and debugging, the loss in portability, and the difficulty of maintaining such code. One of the few widely accepted axioms of

software engineering is that coding takes longer if you write more lines, and it clearly takes many more lines to write a program in assembly language than in C or Java. Moreover, once it is coded, the next danger is that it will become a popular program. Such programs always live longer than expected, meaning that someone will have to update the code over several years and make it work with new releases of operating systems and recent computers. Writing in higher-level language instead of assembly language not only allows future compilers to tailor the code to forthcoming machines; it also makes the software easier to maintain and allows the program to run on more brands of computers.

Fallacy: The importance of commercial binary compatibility means successful instruction sets don't change.

While backwards binary compatibility is sacrosanct, Figure 2.39 shows that the x86 architecture has grown dramatically. The average is more than one instruction per month over its 35-year lifetime!

Pitfall: Forgetting that sequential word or doubleword addresses in machines with byte addressing do not differ by one.

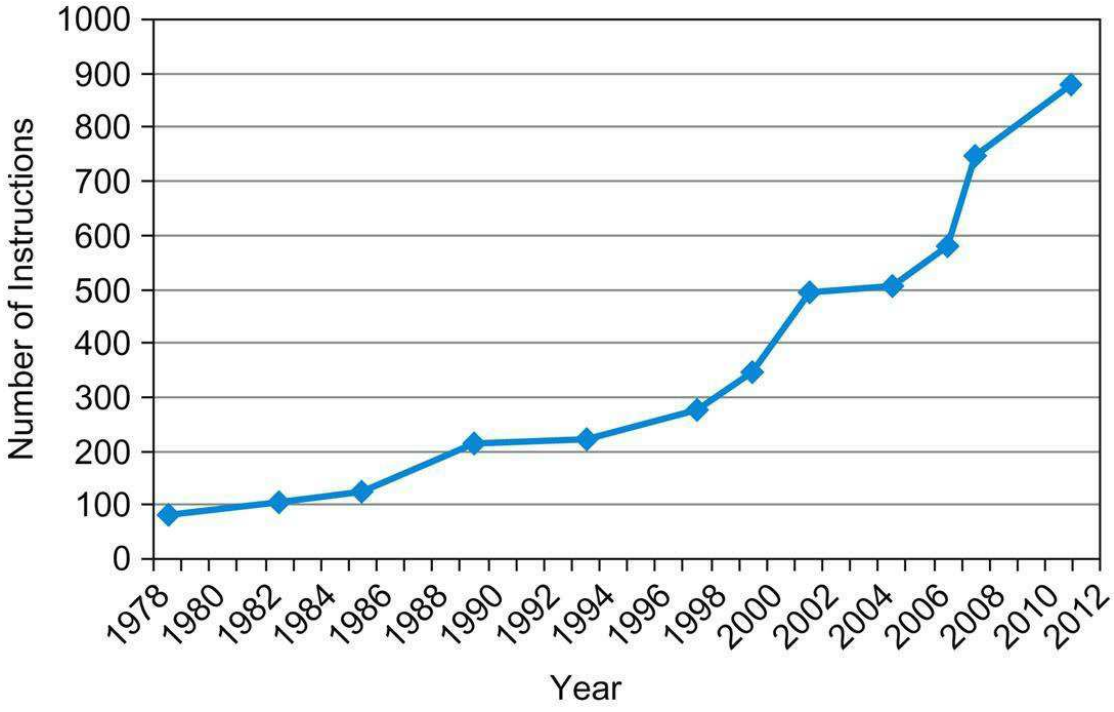


FIGURE 2.39 Growth of x86 instruction set over time.
While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.

Many an assembly language programmer has toiled over errors made by assuming that the address of the next word or doubleword can be found by incrementing the address in a register by one instead of by the word or doubleword size in bytes. Forewarned is forearmed!

Pitfall: Using a pointer to an automatic variable outside its defining procedure.

A common mistake in dealing with pointers is to pass a result from a procedure that includes a pointer to an array that is local to that procedure. Following the stack discipline in Figure 2.12, the memory that contains the local array will be reused as soon as the procedure returns. Pointers to automatic variables can lead to chaos.

2.20 Concluding Remarks

Less is more.

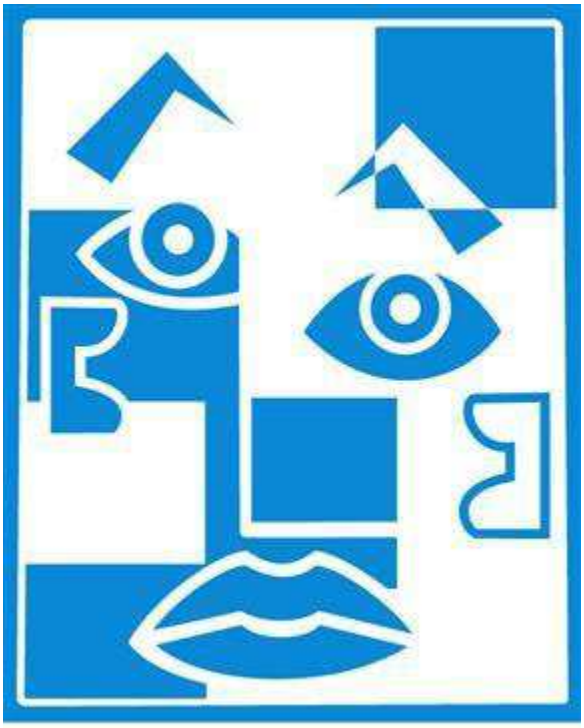
Robert Browning, Andrea del Sarto, 1855

The two principles of the *stored-program* computer are the use of instructions that are indistinguishable from numbers and the use of alterable memory for programs. These principles allow a single machine to aid cancer researchers, financial advisers, and novelists in their specialties. The selection of a set of instructions that the machine can understand demands a delicate balance among the number of instructions needed to execute a program, the number of clock cycles needed by an instruction, and the speed of the clock. As illustrated in this chapter, three design principles guide the authors of instruction sets in making that tricky tradeoff:
1. *Simplicity favors regularity.* Regularity motivates many features of

the RISC-V instruction set: keeping all instructions a single size, always requiring register operands in arithmetic instructions, and keeping the register fields in the same place in all instruction formats.
2. *Smaller is faster.* The desire for speed is the reason that RISC-V has 32 registers rather than many more.
3. *Good design demands good compromises.* One RISC-V example is the compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length.

We also saw the great idea from Chapter 1 of making the **common cast fast** applied to instruction sets as well as computer architecture. Examples of making the common RISC-V case fast include PC-relative addressing for conditional branches and immediate addressing for larger constant operands.

Above this machine level is assembly language, a language that humans can read. The assembler translates it into the binary numbers that machines can understand, and it even "extends" the instruction set by creating symbolic instructions that aren't in the hardware. For instance, constants or addresses that are too big are broken into properly sized pieces, common variations of instructions are given their own name, and so on. Figure 2.40 lists the RISC-V instructions we have covered so far, both real and pseudoinstructions. Hiding details from the higher level is another example of the great idea of **abstraction**.



ABSTRACTION

RISC-V Instructions	Name	Format	Pseudo RISC-V	Name	Real Instruction
Add	add	R	Move	mv	addi
Subtract	sub	R	Load immediate	li	addi
Add immediate	addi	I	Jump	j	jal
Load doubleword	ld	I	Load address	la	lui+addi
Store doubleword	sd	S			
Load word	lw	I			
Load word, unsigned	lwu	I			
Store word	sw	S			
Load halfword	lh	I			
Load halfword, unsigned	lhu	I			
Store halfword	sh	S			
Load byte	lb	I			
Load byte, unsigned	lbu	I			
Store byte	sb	S			
Load reserved	lr.d	R			
Store conditional	sc.d	R			
Load upper immediate	lui	U			
And	and	R			
Inclusive or	or	R			
Exclusive or	xor	R			
And immediate	andi	I			
Inclusive or immediate	ori	I			
Exclusive or immediate	xori	I			
Shift left logical	sll	R			
Shift right logical	srl	R			
Shift right arithmetic	sra	R			
Shift left logical immediate	slli	I			
Shift right logical immediate	srli	I			
Shift right arithmetic immediate	srai	I			
Branch if equal	beq	SB			
Branch if not equal	bne	SB			
Branch if less than	blt	SB			
Branch if greater or equal	bge	SB			
Branch if less, unsigned	bltu	SB			
Branch if greatr/eq, unsigned	bgeu	SB			
Jump and link	jal	UJ			
Jump and link register	jalr	I			

FIGURE 2.40 The RISC-V instruction set covered so far, with the real RISC-V instructions on the left and the pseudoinstructions on the right. Figure 2.1 shows more details of the RISC-V architecture revealed in this chapter. The information given here is also found in Columns 1 and 2 of the RISC-V Reference Data Card at the front of the book.

Each category of RISC-V instructions is associated with constructs that appear in programming languages:

- Arithmetic instructions correspond to the operations found in assignment statements.
- Transfer instructions are most likely to occur when dealing with

data structures like arrays or structures.

- Conditional branches are used in *if* statements and in loops.
- Unconditional branches are used in procedure calls and returns and for *case*/*switch* statements.

These instructions are not born equal; the popularity of the few dominates the many. For example, Figure 2.41 shows the popularity of each class of instructions for SPEC CPU2006. The varying popularity of instructions plays an important role in the chapters about datapath, control, and pipelining.

Instruction class	RISC-V examples	HLL correspondence	Frequency	
			Integer	Fl. Pt.
Arithmetic	`add, sub, addi`	Operations in assignment statements	16%	48%
Data transfer	`ld, sd, lw, sw, lh, sh, lb, sb, lui`	References to data structures in memory	35%	36%
Logical	`and, or, xor, sll, srl, sra`	Operations in assignment statements	12%	4%
Branch	`beq, bne, blt, bge, bltu, bgeu`	*If* statements; loops	34%	8%
Jump	`jal, jalr`	Procedure calls & returns; *switch* statements	2%	0%

FIGURE 2.41 RISC-V instruction classes, examples, correspondence to high-level program language constructs, and percentage of RISC-V instructions executed by category for the average integer and floating point SPEC CPU2006 benchmarks. Figure 3.24 in Chapter 3 shows average percentage of the individual RISC-V instructions executed.

After we explain computer arithmetic in Chapter 3, we reveal more of the RISC-V instruction set architecture.

Historical Perspective and Further Reading

This section surveys the history of *instruction set architectures* (ISAs) over time, and we give a short history of programming languages and compilers. ISAs include accumulator architectures, general-purpose register architectures, stack architectures, and a brief history of the x86 and ARM's 32-bit architecture, ARMv7. We also review the controversial subjects of high-level-language computer architectures and reduced instruction set computer architectures. The history of programming languages includes Fortran, Lisp, Algol, C, Cobol, Pascal, Simula, Smalltalk, C++, and Java, and the history of compilers includes the key milestones and the pioneers who achieved them. The rest of  **Section 2.21** is found online.

2.22 Historical Perspective and Further Reading

This section surveys the history of instruction set architectures over time, and we give a short history of programming languages and compilers. ISAs include accumulator architectures, general-purpose register architectures, stack architectures, and a brief history of ARMv7 and the x86. We also review the controversial subjects of high-level-language computer architectures and reduced instruction set computer architectures. The history of programming languages includes Fortran, Lisp, Algol, C, Cobol, Pascal, Simula, Smalltalk, C++, and Java, and the history of compilers includes the key milestones and the pioneers who achieved them.

Accumulator Architectures

Hardware was precious in the earliest stored-program computers. Consequently, computer pioneers could not afford the number of registers found in today's architectures. In fact, these architectures had a single register for arithmetic instructions. Since all operations would accumulate in one register, it was called the **accumulator**, and this style of instruction set is given the same name. For example, EDSAC in 1949 had a single accumulator.

accumulator

> *Archaic term for register. On-line use of it as a synonym for "register" is a fairly reliable indication that the user has been around quite a while.*
>
> Eric Raymond, The New Hacker's Dictionary, 1991

The three-operand format of RISC-V suggests that a single register is at least two registers shy of our needs. Having the accumulator as both a source operand *and* the destination of the operation fills part of the shortfall, but it still leaves us one operand short. That final operand is found in memory. Accumulator architectures have the memory-based operand-addressing mode suggested earlier. It follows that the add instruction of an accumulator instruction set would look like this:

```
ADD 200
```

This instruction means add the accumulator to the word in memory at address 200 and place the sum back into the accumulator. No registers are specified because the accumulator is known to be both a source and a destination of the operation.

The next step in the evolution of instruction sets was the addition of registers dedicated to specific operations. Hence, registers might be included to act as indices for array references in data transfer instructions, to act as separate accumulators for multiply or divide instructions, and to serve as the top-of-stack pointer. Perhaps the best-known example of this style of instruction set is found in the Intel 80x86. This style of instruction set is labeled *extended accumulator*, *dedicated register*, or *special-purpose register*. Like the single-register accumulator architectures, one operand may be in memory for arithmetic instructions. Like the RISC-V architecture, however, there are also instructions where all the operands are registers.

General-Purpose Register Architectures

The generalization of the dedicated-register architecture allows all the registers to be used for any purpose, hence the name *general-purpose register*. RISC-V is an example of a general-purpose register architecture. This style of instruction set may be further divided into those that allow one operand to be in memory (as found in accumulator architectures), called a *register-memory* architecture,

and those that demand that operands always be in registers, called either a **load-store** or a **register-register** architecture. Figure e2.22.1 shows a history of the number of registers in some popular computers.

load-store architecture

Also called **register-register** architecture. An instruction set architecture in which all operations are between registers and data memory may only be accessed via loads or stores.

Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-store	1963
IBM 360	16	Register-memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	8	Register-memory	1985
ARM	16	Load-store	1985
MIPS	32	Load-store	1985
HP PA-RISC	32	Load-store	1986
SPARC	32	Load-store	1987
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992
HP/Intel IA-64	128	Load-store	2001
AMD64 (EMT64)	16	Register-memory	2003
RISC-V	32	Load-store	2010

FIGURE E2.22.1 The number of general-purpose registers in popular architectures over the years.

The first load-store architecture was the CDC 6600 in 1963, considered by many to be the first supercomputer. RISC-V, ARMv7, ARMv8, and MIPS are more recent examples of a load-store architecture.

The 80386 was Intel's attempt to transform the 8086 into a general-purpose register-memory instruction set. Perhaps the best-known register-memory instruction set is the IBM 360 architecture, first announced in 1964. This instruction set is still at the core of IBM's mainframe computers—responsible for a large part of the business of the largest computer company in the world. Register-memory architectures were the most popular in the 1960s and the first half of the 1970s.

Digital Equipment Corporation's VAX architecture took memory operands one step further in 1977. It allowed an instruction to use any combination of registers and memory operands. A style of architecture in which all operands can be in memory is called *memory-memory*. (In truth the VAX instruction set, like almost all

other instruction sets since the IBM 360, is a hybrid, since it also has general-purpose registers.)

The Intel x86 has many versions of a 64-bit add to specify whether an operand is in memory or is in a register. In addition, the memory operand can be accessed with more than seven addressing modes. This combination of address modes and register-memory operands means that there are dozens of variants of an x86 add instruction. Clearly, this variability makes x86 implementations more challenging.

Compact Code and Stack Architectures

When memory is scarce, it is also important to keep programs small, so architectures like the Intel x86, IBM 360, and VAX had variable-length instructions, both to match the varying operand specifications and to minimize code size. Intel x86 instructions are from 1 to 15 bytes long; IBM 360 instructions are 2, 4, or 6 bytes long; and VAX instruction lengths are anywhere from 1 to 54 bytes.

One place where code size is still important is embedded applications. In recognition of this need, ARM, MIPS, and RISC-V all made versions of their instructions sets that offer both 16-bit instruction formats and 32-bit instruction formats: Thumb and Thumb-2 for ARM, MIPS-16, and RISC-V Compressed. Despite being limited to just two sizes, Thumb, Thumb-2, MIPS-16, and RISC-V Compressed programs are about 25% to 30% smaller, which makes their code sizes smaller than those of the 80x86. Smaller code sizes have the added benefit of improving instruction cache hit rates (see Chapter 5).

In the 1960s, a few companies followed a radical approach to instruction sets. In the belief that it was too hard for compilers to utilize registers effectively, these companies abandoned registers altogether! Instruction sets were based on a *stack model* of execution, like that found in the older Hewlett-Packard handheld calculators. Operands are pushed on the stack from memory or popped off the stack into memory. Operations take their operands from the stack and then place the result back onto the stack. In addition to simplifying compilers by eliminating register allocation, stack architectures lent themselves to compact instruction encoding, thereby removing memory size as an excuse not to program in

high-level languages.

Memory space was perceived to be precious again for Java, both because memory space is limited to keep costs low in embedded applications and because programs may be downloaded over the Internet or phone lines as Java applets, and smaller programs take less time to transmit. Hence, compact instruction encoding was desirable for Java bytecodes.

High-Level-Language Computer Architectures

In the 1960s, systems software was rarely written in high-level languages. For example, virtually every commercial operating system before UNIX was programmed in assembly language, and more recently even OS/2 was originally programmed at that same low level. Some people blamed the code density of the instruction sets, rather than the programming languages and the compiler technology.

Hence, an architecture design philosophy called *high-level-language computer architecture* was advocated, with the goal of making the hardware more like the programming languages. More efficient programming languages and compilers, plus expanding memory, doomed this movement to a historical footnote. The Burroughs B5000 was the commercial fountainhead of this philosophy, but today there is no significant commercial descendant of this 1960s radical.

Reduced Instruction Set Computer Architectures

This language-oriented design philosophy was replaced in the 1980s by *RISC (reduced instruction set computer)*. Improvements in programming languages, compiler technology, and memory cost meant that less programming was being done at the assembly level, so instruction sets could be measured by how well compilers used them, in contrast to how skillfully assembly language programmers used them.

Virtually all new instruction sets since 1982 have followed this RISC philosophy of fixed instruction lengths, load-store instruction

sets, limited addressing modes, and limited operations. ARMv7, ARMv8 Hitachi SH, IBM PowerPC, MIPS, Sun SPARC, and, of course, RISC-V, are all examples of RISC architectures.

A Brief History of the ARMv7

ARM started as the processor for the Acorn computer, hence its original name of Acorn RISC Machine. The Berkeley RISC papers influenced its architecture.

One of the most important early applications was emulation of the AM 6502, a 16-bit microprocessor. This emulation was to provide most of the software for the Acorn computer. As the 6502 had a variable-length instruction set that was a multiple of bytes, 6502 emulation helps explain the emphasis on shifting and masking in the ARMv7 instruction set.

Its popularity as a low-power embedded computer began with its selection as the processor for the ill-fated Apple Newton personal digital assistant. Although the Newton was not as popular as Apple hoped, Apple's blessing gave visibility to the earlier ARM instruction sets, and they subsequently caught on in several markets, including cell phones. Unlike the Newton experience, the extraordinary success of cell phones explains why 12 billion ARM processors were shipped in 2014.

One of the major events in ARM's history is the 64-bit address extension called version 8. ARM took the opportunity to redesign the instruction set to make it look much more like MIPS than like earlier ARM versions.

A Brief History of the x86

The ancestors of the x86 were the first microprocessors, produced starting in 1972. The Intel 4004 and 8008 were extremely simple 4-bit and 8-bit accumulator-style architectures. Morse et al. [1980] describe the evolution of the 8086 from the 8080 in the late 1970s as an attempt to provide a 16-bit architecture with better throughput. At that time, almost all programming for microprocessors was done in assembly language—both memory and compilers were in short supply. Intel wanted to keep its base of 8080 users, so the 8086 was designed to be "compatible" with the 8080. The 8086 was *never*

object-code compatible with the 8080, but the architectures were close enough that translation of assembly language programs could be done automatically.

In early 1980, IBM selected a version of the 8086 with an 8-bit external bus, called the 8088, for use in the IBM PC. They chose the 8-bit version to reduce the cost of the architecture. This choice, together with the tremendous success of the IBM PC, has made the 8086 architecture ubiquitous. The success of the IBM PC was due in part because IBM opened the architecture of the PC and enabled the PC-clone industry to flourish. As discussed in Section 2.18, the 80286, 80386, 80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, and AMD64 have extended the architecture and provided a series of performance enhancements.

Although the 68000 was chosen for the Macintosh, the Mac was never as pervasive as the PC, partly because Apple did not allow Mac clones based on the 68000, and the 68000 did not acquire the same software following that which the 8086 enjoys. The Motorola 68000 may have been more significant *technically* than the 8086, but the impact of IBM's selection and open architecture strategy dominated the technical advantages of the 68000 in the market.

Some argue that the inelegance of the x86 instruction set is unavoidable, the price that must be paid for rampant success by any architecture. We reject that notion. Obviously, no successful architecture can jettison features that were added in previous implementations, and over time, some features may be seen as undesirable. The awkwardness of the x86 begins at its core with the 8086 instruction set and was exacerbated by the architecturally inconsistent expansions found in the 8087, 80286, 80386, MMX, SSE, SSE2, SSE3, SSE4, AMD64 (EM64T), and AVX.

A counterexample is the IBM 360/370 architecture, which is much older than the x86. It dominated the mainframe market just as the x86 dominated the PC market. Due undoubtedly to a better base and more compatible enhancements, this instruction set makes much more sense than the x86 50 years after its first implementation.

Extending the x86 to 64-bit addressing means the architecture may last for several more decades. Instruction set anthropologists of the future will peel off layer after layer from such architectures until they uncover artifacts from the first microprocessor. Given

such a find, how will they judge today's computer architecture?

A Brief History of Programming Languages

In 1954, John Backus led a team at IBM to create a more natural notation for scientific programming. The goal of Fortran, for "FORmula TRANslator," was to reduce the time to develop programs. Fortran included many ideas found in programming languages today, including assignment statements, expressions, typed variables, loops, and arrays. The development of the language and the compiler went hand in hand. This language became a standard that has evolved over time to improve programmer productivity and program portability. The evolutionary steps are Fortran I, II, IV, 77, and 90.

Fortran was developed for IBM's second commercial computer, the 704, which was also the cradle of another important programming language: Lisp. John McCarthy invented the "LISt Processing" language in 1958. Its mantra is that programming can be considered as manipulating lists, so the language contains operations to follow links and to compose new lists from old ones. This list notation is used for the code as well as the data, so modifying or composing Lisp programs is common. The big contribution was dynamic data structures and, hence, pointers. Given that its inventor was a pioneer in artificial intelligence, Lisp became popular in the AI community. Lisp has no type declarations, and Lisp traditionally reclaims storage automatically via built-in garbage collection. Lisp was originally interpreted, although compilers were later developed for it.

Fortran inspired the international community to invent a programming language that was more natural to express algorithms than Fortran, with less emphasis on coding. This language became Algol, for "ALGOrithmic Language." Like Fortran, it included type declarations, but it added recursive procedure calls, nested *if-then-else* statements, *while* loops, *begin-end* statements to structure code, and call-by-name. Algol-60 became the classic language for academics to teach programming in the 1960s.

Although engineers, AI researchers, and computer scientists had their own programming languages, the same could not be said for

business data processing. Cobol, for "COmmon Business-Oriented Language," was developed as a standard for this purpose contemporary with Algol-60. Cobol was created to be easy to read, so it follows English vocabulary and punctuation. It added records to programming languages, and separated description of data from description of code.

Niklaus Wirth was a member of the Algol-68 committee, which was supposed to update Algol-60. He was bothered by the complexity of the result, and so he wrote a minority report to show that a programming language could combine the algorithmic power of Algol-60 with the record structure from Cobol and be simple to understand, easy to implement, yet still powerful. This minority report became Pascal. It was first implemented with an interpreter and a set of Pascal bytecodes. The ease of implementation led to its being widely deployed, much more than Algol-68, and it soon replaced Algol-60 as the most popular language for academics to teach programming.

In the same period, Dennis Ritchie invented the C programming language to use in building UNIX. Its inventors say it is not a "very high level" programming language or a big one, and it is not aimed at a particular application. Given its birthplace, it was very good at systems programming, and the UNIX operating system and C compiler were written in C. UNIX's popularity helped spur C's popularity.

The concept of object orientation is first captured in Simula-67, a simulation language successor to Algol-60. Invented by Ole-Johan Dahl and Kristen Nygaard at the University of Oslo in 1967, it introduced objects, classes, and inheritance.

Object orientation proved to be a powerful idea. It led Alan Kay and others at Xerox Palo Alto Research Center to invent Smalltalk in the 1970s. Smalltalk-80 married the typeless variables and garbage collection from Lisp and the object orientation of Simula-67. It relied on interpretation that was defined by a Smalltalk virtual machine with a Smalltalk bytecode instruction set. Kay and his colleagues argued that processors were getting faster, and that we must eventually be willing to sacrifice some performance to improve program development. Another example was CLU, which demonstrated that an object-oriented language could be defined that allowed compile-time type checking. Simula-67 also inspired

Bjarne Stroustrup of Bell Labs to develop an object-oriented version of C called C++ in the 1980s. C++ became widely used in industry.

Dissatisfied with C++, a group at Sun led by James Gosling invented Oak in the early 1990s. It was invented as an object-oriented C dialect for embedded devices as part of a major Sun project. To make it portable, it was interpreted and had its own virtual machine and bytecode instruction set. Since it was a new language, it had a more elegant object-oriented design than C++ and was much easier to learn and compile than Smalltalk-80. Since Sun's embedded project failed, we might never have heard of it had someone not made the connection between Oak and programmable browsers for the World Wide Web. It was rechristened Java, and in 1995, Netscape announced that it would be shipping with its browser. It soon became extraordinarily popular. Java had the rare distinction of becoming the standard language for new business data processing applications *and* the favored language for academics to teach programming. Java and languages like it encourage reuse of code, and hence programmers make heavy use of libraries, whereas in the past they were more likely to write everything from scratch.

A Brief History of Compilers

Backus and his group were very concerned that Fortran would be unsuccessful if skeptics found examples where the Fortran version ran at half the speed of the equivalent assembly language program. Their success with one of the first compilers created a beachhead that many others followed.

Early compilers were ad hoc programs that performed the steps described in Section 2.15 online. These ad hoc approaches were replaced with a solid theoretical foundation for each of these steps. Each time the theory was established, a tool was built based on that theory that automated the creation of that step.

The theoretical roots underlying scanning and parsing derive from automata theory, and the relationship between languages and automata was known early. The scanning task corresponds to recognition of a language accepted by a finite-state automata, and parsing corresponds to recognition of a language by a push-down automata (basically an automata with a stack). Languages are

described by grammars, which are a set of rules that tell how any legal program can be generated.

The scanning pass of a compiler was well understood early, but parsing is harder. The earliest parsers use precedence techniques, which derived from the structure of arithmetic statements, and were then generalized. The great breakthrough in modern parsing was made by Donald Knuth in the invention of LR-parsing, which codified the two key steps in the parsing technique, pushing a token on the stack or reducing a set of tokens on the stack using a grammar rule. The strong theory formulation for scanning and parsing led to the development of automated tools for compiler constructions, such as `lex` and `yacc`, the tools developed as part of UNIX.

Optimizations occurred in many compilers, and it is harder to determine the first examples in most cases. However, Victor Vyssotsky did the first papers on data flow analysis in 1963, and William McKeeman is generally credited with the first peephole optimizer in 1965. The group at IBM, including John Cocke and Fran Allan, developed many of the early optimization concepts, as well as defining and extending the concepts of flow analysis. Important contributions were also made by Al Aho and Jeff Ullman.

One of the biggest challenges for optimization was register allocation. It was so difficult that some architects used stack architectures just to avoid the problem. The breakthrough came when researchers working on compilers for the 801, an early RISC architecture, recognized that coloring a graph with a minimum number of colors was equivalent to allocating a fixed number of registers to the unlimited number of virtual registers used in intermediate forms.

Compilers also played an important role in the open-source movement. Richard Stallman's self-appointed mission was to make a public domain version of UNIX. He built the GNU C Compiler (gcc) as an open-source compiler in 1987. It soon was ported to many architectures, and is used in many systems today.

Further Reading

Bayko, J. [1996]. "Great microprocessors of the past and present,"

search for it on the *http://www.cpushack.com/CPU/cpu.html*.

A personal view of the history of both representative and unusual microprocessors, from the Intel 4004 to the Patriot Scientific ShBoom!

Kane, G. and J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ.

This book describes the MIPS architecture in greater detail than Appendix A.

Levy, H. and R. Eckhouse [1989]. *Computer Programming and Architecture*, The VAX, Digital Press, Boston.

This book concentrates on the VAX, but also includes descriptions of the Intel 8086, IBM 360, and CDC 6600.

Morse, S., B. Ravenal, S. Mazor, and W. Pohlman [1980]. "Intel microprocessors—8080 to 8086", *Computer* 13 10 (October).

The architecture history of the Intel from the 4004 to the 8086, according to the people who participated in the designs.

Wakerly, J. [1989]. Microcomputer Architecture and Programming, Wiley, New York.

The Motorola 6800 is the main focus of the book, but it covers the Intel 8086, Motorola 6809, TI 9900, and Zilog Z8000.

2.22 Exercises

2.1 [5] <§2.2> For the following C statement, write the corresponding RISC-V assembly code. Assume that the C variables `f`, `g`, and `h`, have already been placed in registers `x5`, `x6`, and `x7` respectively. Use a minimal number of RISC-V assembly instructions.

```
f = g + (h - 5);
```

2.2 [5] <§2.2> Write a single C statement that corresponds to the two RISC-V assembly instructions below.

```
add f, g, h
add f, i, f
```

2.3 [5] <§§2.2, 2.3> For the following C statement, write the corresponding RISC-V assembly code. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `x5, x6, x7, x28`, and `x29`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `x10` and `x11`, respectively.

```
B[8] = A[i-j];
```

2.4 [10] <§§2.2, 2.3> For the RISC-V assembly instructions below,