



# 2

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

**Charles V, Holy Roman Emperor**  
(1500–1558)

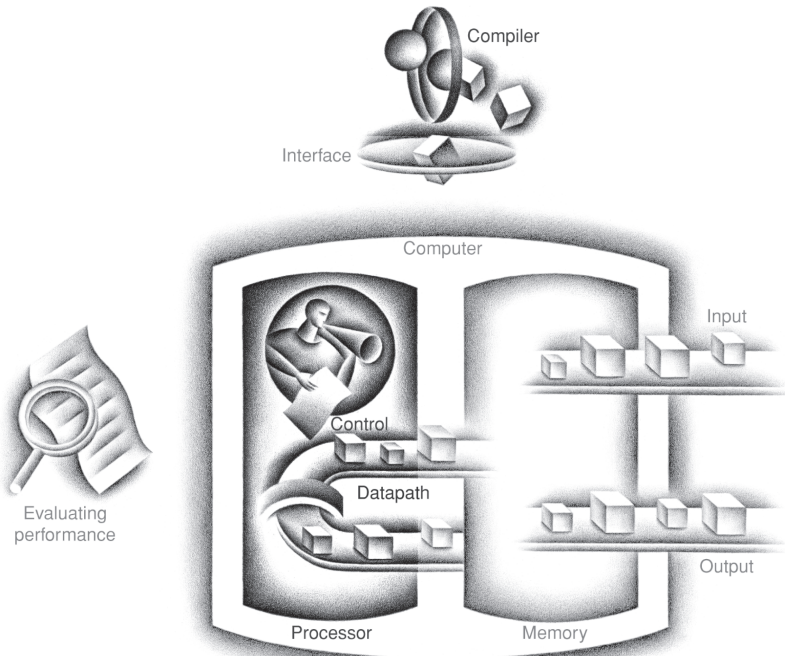
## Instructions: Language of the Computer

- 2.1 Introduction** 62
- 2.2 Operations of the Computer Hardware** 63
- 2.3 Operands of the Computer Hardware** 66
- 2.4 Signed and Unsigned Numbers** 73
- 2.5 Representing Instructions in the  
Computer** 80
- 2.6 Logical Operations** 87
- 2.7 Instructions for Making Decisions** 90

<b>2.8</b>	<b>Supporting Procedures in Computer Hardware</b>	96
<b>2.9</b>	<b>Communicating with People</b>	106
<b>2.10</b>	<b>MIPS Addressing for 32-Bit Immediates and Addresses</b>	111
<b>2.11</b>	<b>Parallelism and Instructions: Synchronization</b>	121
<b>2.12</b>	<b>Translating and Starting a Program</b>	123
<b>2.13</b>	<b>A C Sort Example to Put It All Together</b>	132
<b>2.14</b>	<b>Arrays versus Pointers</b>	141
	<b>2.15 Advanced Material: Compiling C and Interpreting Java</b>	145
<b>2.16</b>	<b>Real Stuff: ARMv7 (32-bit) Instructions</b>	145
<b>2.17</b>	<b>Real Stuff: x86 Instructions</b>	149
<b>2.18</b>	<b>Real Stuff: ARMv8 (64-bit) Instructions</b>	158
<b>2.19</b>	<b>Fallacies and Pitfalls</b>	159
<b>2.20</b>	<b>Concluding Remarks</b>	161
	<b>2.21 Historical Perspective and Further Reading</b>	163
<b>2.22</b>	<b>Exercises</b>	164

---

## The Five Classic Components of a Computer



## 2.1 Introduction

**instruction set** The vocabulary of commands understood by a given architecture.

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 real language of a real computer. Chapter 3 continues our downward descent, unveiling the hardware for arithmetic and the representation of floating-point numbers.

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 like independent languages. Hence, once you learn one, it is easy to pick up others.


The chosen instruction set comes from MIPS Technologies, and is an elegant example of the instruction sets designed since the 1980s. To demonstrate how easy it is to pick up other instruction sets, we will take a quick look at three other popular instruction sets.

1. ARMv7 is similar to MIPS. More than 9 billion chips with ARM processors were manufactured in 2011, making it the most popular instruction set in the world.
2. The second example is the Intel x86, which powers both the PC and the cloud of the PostPC Era.
3. The third example is ARMv8, which extends the address size of the ARMv7 from 32 bits to 64 bits. Ironically, as we shall see, this 2013 instruction set is closer to MIPS than it is to ARMv7.

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.

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.

### stored-program concept

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

## 2.2

## Operations of the Computer Hardware

Every computer must be able to perform arithmetic. The MIPS 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 MIPS 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 sharp symbol (*#*) 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

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

Burks, Goldstine, and von Neumann, 1947

## MIPS operands

Name	Example	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register \$zero always equals 0, and register \$at is reserved by the assembler to handle large constants.
2 <sup>30</sup> memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

## MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three register operands
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Three register operands
	add immediate	addi \$s1,\$s2,20	\$s1 = \$s2 + 20	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word from memory to register
	store word	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Word from register to memory
	load half	lh \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	store half	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	Memory[\$s2+20]=\$s1; \$s1=0 or 1	Store word as 2nd half of atomic swap
Logical	load upper immed.	lui \$s1,20	\$s1 = 20 * 2 <sup>16</sup>	Loads constant in upper 16 bits
	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2   \$s3	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	\$s1 = ~ (\$s2   \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	\$s1 = \$s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	\$s1 = \$s2   20	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Shift left by constant
Conditional branch	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Shift right by constant
	branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if (\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant
Unconditional jump	set less than immediate unsigned	sltiu \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant unsigned
	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call

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

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 MIPS

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 translation from C to MIPS assembly language instructions is performed by the *compiler*. Show the MIPS code produced by a compiler.

A MIPS 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 MIPS assembly language instructions:

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

**EXAMPLE****ANSWER**

### Compiling a Complex C Assignment into MIPS

A somewhat complex statement contains the five variables *f*, *g*, *h*, *i*, and *j*:

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

What might a C compiler produce?

**EXAMPLE**

**ANSWER**

The compiler must break this statement into several assembly instructions, since only one operation is performed per MIPS instruction. The first MIPS 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. MIPS 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 MIPS instruction set. To get performance close to the equivalent C program, Java systems today typically compile Java bytecodes into the native instruction sets like MIPS. 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 MIPS architecture is 32 bits; groups of 32 bits occur so frequently that they are given the name **word** in the MIPS architecture.

**word** The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers, like MIPS. (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 MIPS language, in this section we have added the restriction that the three operands of MIPS arithmetic instructions must each be chosen from one of the 32 32-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. Yet, 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 this 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 MIPS convention is to use two-character names following a dollar sign to represent a register. Section 2.8 will explain the reasons behind these names. For now, we will use `$s0`, `$s1`, ... for registers that correspond to variables in C and Java programs and `$t0`, `$t1`, ... for temporary registers needed to compile the program into MIPS instructions.

### 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 `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. What is the compiled MIPS 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, \$t0 and \$t1, which correspond to the temporary variables above:

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```

Memory Operands

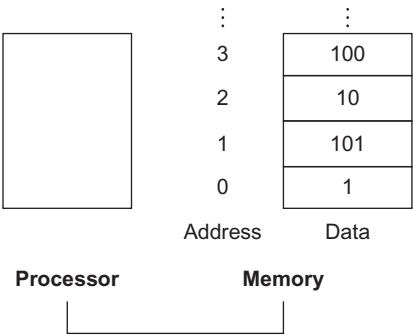
Programming languages have simple variables that contain single data elements, as in these examples, but they also have more complex data structures—arrays and structures. These complex data structures can contain many more data elements than there are registers in a computer. How can a computer represent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

**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.

As explained above, arithmetic operations occur only on registers in MIPS instructions; thus, MIPS must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**. To access a word 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.



**FIGURE 2.2 Memory addresses and contents of memory at those locations.** If these elements were words, these addresses would be incorrect, since MIPS actually uses byte addressing, with each word representing four bytes. Figure 2.3 shows the memory addressing for sequential word 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 a constant and register 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 actual MIPS name for this instruction is *lw*, standing for *load word*.

### Compiling an Assignment When an Operand Is in Memory

Let's assume that *A* is an array of 100 words and that the compiler has associated the variables *g* and *h* with the registers *\$s1* and *\$s2* as before. Let's also assume that the starting address, or *base address*, of the array is in *\$s3*. Compile this C assignment statement:

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

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 *\$s3*, 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

```
lw    $t0,8($s3) # Temporary reg $t0 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 *\$t0* (which equals *A[8]*) since it is in a register. The instruction must add *h* (contained in *\$s2*) to *A[8]* (contained in *\$t0*) and put the sum in the register corresponding to *g* (associated with *\$s1*):

```
add    $s1,$s2,$t0 # g = h + A[8]
```

The constant in a data transfer instruction (8) is called the *offset*, and the register added to form the address (*\$s3*) is called the *base register*.

### EXAMPLE

### ANSWER

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.

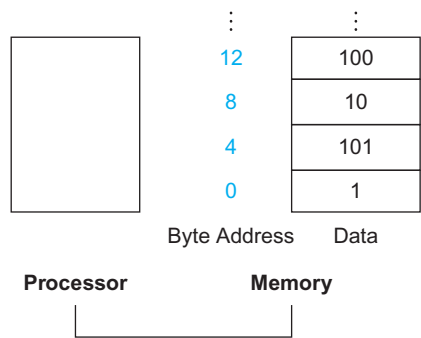
Since 8-bit *bytes* are useful in many programs, virtually all architectures today address individual bytes. Therefore, the address of a word matches the address of one of the 4 bytes within the word, and addresses of sequential words differ by 4. For example, Figure 2.3 shows the actual MIPS addresses for the words in Figure 2.2; the byte address of the third word is 8.

In MIPS, words must start at addresses that are multiples of 4. This requirement is called an **alignment restriction**, and many architectures have it. (Chapter 4 suggests why alignment leads to faster data transfers.)

### Hardware/ Software Interface

#### alignment restriction

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



**FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words.** The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

Computers divide into those that use the address of the leftmost or “big end” byte as the word address versus those that use the rightmost or “little end” byte. MIPS is in the *big-endian* camp. Since the order matters only if you access the identical data both as a word and as four bytes, few need to be aware of the endianness. (Appendix A shows the two options to number bytes in a word.)

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 \$s3 must be  $4 \times 8$ , or 32*, so that the load address will select `A[8]` and not `A[8/4]`. (See the related pitfall on page 160 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 offset to select the array element, and finally the base register. Once again, the MIPS address is specified in part by a constant and in part by the contents of a register. The actual MIPS name is `sw`, standing for *store word*.

**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

Assume variable `h` is associated with register `$s2` and the base address of the array `A` is in `$s3`. What is the MIPS assembly code for the C assignment statement below?

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

Although there is a single operation in the C statement, now two of the operands are in memory, so we need even more MIPS 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 word instruction to select `A[8]`, and the add instruction places the sum in `$t0`:

```
lw    $t0,32($s3) # Temporary reg $t0 gets A[8]
add   $t0,$s2,$t0 # Temporary reg $t0 gets h + A[8]
```

The final instruction stores the sum into `A[12]`, using 48 ( $4 \times 12$ ) as the offset and register `$s3` as the base register.

```
sw    $t0,48($s3) # Stores h + A[8] back into A[12]
```

Load word and store word are the instructions that copy words between memory and registers in the MIPS architecture. Other 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.

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 commonly 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 is indeed the case; data accesses are faster if data is in registers instead of memory.

Moreover, data is more useful when in a register. A MIPS arithmetic instruction can read two registers, operate on them, and write the result. A MIPS 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 faster to access and simpler to use. Accessing registers also uses less energy than accessing memory. To achieve highest performance and conserve energy, an instruction set architecture must have a sufficient number of registers, and compilers must use registers efficiently.

### EXAMPLE

### ANSWER

## Hardware/ Software Interface

## 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 MIPS 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 `$s3`, we could use the code

```
lw $t0, AddrConstant4($s1)    # $t0 = constant 4
add $s3,$s3,$t0                # $s3 = $s3 + $t0 ($t0 == 4)
```

assuming that `$s1 + 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 `$s3`, we just write

```
addi    $s3,$s3,4              # $s3 = $s3 + 4
```

Constant operands occur frequently, and 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, the move operation is just an add instruction where one operand is zero. Hence, MIPS dedicates a register `$zero` to be hard-wired to the value zero. (As you might expect, it is register number 0.) Using frequency to justify the inclusions of constants is another example of the great idea of making the **common case fast**.




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 MIPS registers in this book are 32 bits wide, there is a 64-bit version of the MIPS instruction set with 32 64-bit registers. To keep them straight, they are officially called MIPS-32 and MIPS-64. In this chapter, we use a subset of MIPS-32.  **Appendix E** shows the differences between MIPS-32 and MIPS-64. Sections 2.16 and 2.18 show the much more dramatic difference between the 32-bit address ARMv7 and its 64-bit successor, ARMv8.

**Elaboration:** The MIPS 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:** Since MIPS supports negative constants, there is no need for subtract immediate in MIPS.

## 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.

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 word: simply use the power of the base for that bit. We subscript decimal numbers with *ten* and binary numbers with *two*. For example,

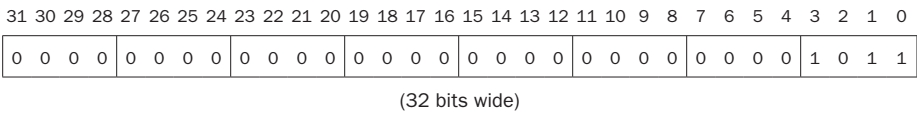
$$1011_{\text{two}}$$

represents

$$\begin{aligned} & (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{\text{ten}} \\ = & (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{\text{ten}} \\ = & 8 + 0 + 2 + 1_{\text{ten}} \\ = & 11_{\text{ten}} \end{aligned}$$

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

We number the bits 0, 1, 2, 3, . . . from *right to left* in a word. The drawing below shows the numbering of bits within a MIPS word and the placement of the number 1011<sub>two</sub>:



**least significant bit** The rightmost bit in a MIPS word.

**most significant bit** The leftmost bit in a MIPS word.

Since words 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 31).

The MIPS word is 32 bits long, so we can represent  $2^{32}$  different 32-bit patterns. It is natural to let these combinations represent the numbers from 0 to  $2^{32} - 1$  (4,294,967,295<sub>ten</sub>):

0000	0000	0000	0000	0000	0000	0000	0000	<sub>two</sub>	=	0 <sub>ten</sub>
0000	0000	0000	0000	0000	0000	0000	0001	<sub>two</sub>	=	1 <sub>ten</sub>
0000	0000	0000	0000	0000	0000	0000	0010	<sub>two</sub>	=	2 <sub>ten</sub>
...										
1111	1111	1111	1111	1111	1111	1111	1101	<sub>two</sub>	=	4,294,967,293 <sub>ten</sub>
1111	1111	1111	1111	1111	1111	1111	1110	<sub>two</sub>	=	4,294,967,294 <sub>ten</sub>
1111	1111	1111	1111	1111	1111	1111	1111	<sub>two</sub>	=	4,294,967,295 <sub>ten</sub>

That is, 32-bit binary numbers can be represented in terms of the bit value times a power of 2 (here *xi* means the *i*th bit of *x*):

$$(x_{31} \times 2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

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. As a result 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:

0000 0000 0000 0000 0000 0000 0000 0000	$= 0_{\text{ten}}$
0000 0000 0000 0000 0000 0000 0000 0001	$= 1_{\text{ten}}$
0000 0000 0000 0000 0000 0000 0000 0010	$= 2_{\text{ten}}$
...	...
0111 1111 1111 1111 1111 1111 1111 1101	$= 2,147,483,645_{\text{ten}}$
0111 1111 1111 1111 1111 1111 1111 1110	$= 2,147,483,646_{\text{ten}}$
0111 1111 1111 1111 1111 1111 1111 1111	$= 2,147,483,647_{\text{ten}}$
1000 0000 0000 0000 0000 0000 0000 0000	$= -2,147,483,648_{\text{ten}}$
1000 0000 0000 0000 0000 0000 0000 0001	$= -2,147,483,647_{\text{ten}}$
1000 0000 0000 0000 0000 0000 0000 0010	$= -2,147,483,646_{\text{ten}}$
...	...
1111 1111 1111 1111 1111 1111 1111 1101	$= -3_{\text{ten}}$
1111 1111 1111 1111 1111 1111 1111 1110	$= -2_{\text{ten}}$
1111 1111 1111 1111 1111 1111 1111 1111	$= -1_{\text{ten}}$

The positive half of the numbers, from 0 to  $2,147,483,647_{\text{ten}}$  ( $2^{31} - 1$ ), use the same representation as before. The following bit pattern ( $1000 \dots 0000_{\text{two}}$ ) represents the most negative number  $-2,147,483,648_{\text{ten}}$  ( $-2^{31}$ ). It is followed by a declining set of negative numbers:  $-2,147,483,647_{\text{ten}}$  ( $1000 \dots 0001_{\text{two}}$ ) down to  $-1_{\text{ten}}$  ( $1111 \dots 1111_{\text{two}}$ ).

Two's complement does have one negative number,  $-2,147,483,648_{\text{ten}}$ , that has no corresponding positive number. 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. Consequently, hardware needs to test only this bit to see if a number is positive or negative (with the number 0 considered positive). This bit is often called the *sign bit*. By recognizing the role of the sign bit, we can represent positive and negative 32-bit numbers in terms of the bit value times a power of 2:

$$(x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

The sign bit is multiplied by  $-2^{31}$ , and the rest of the bits are then multiplied by positive versions of their respective base values.

## EXAMPLE

### Binary to Decimal Conversion

What is the decimal value of this 32-bit two's complement number?

1111 1111 1111 1111 1111 1111 1111 1100<sub>two</sub>

## ANSWER

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

$$\begin{aligned} & (1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \dots + (1 \times 2^1) + (0 \times 2^1) + (0 \times 2^0) \\ &= -2^{31} + 2^{30} + 2^{29} + \dots + 2^2 + 0 + 0 \\ &= -2,147,483,648_{\text{ten}} + 2,147,483,644_{\text{ten}} \\ &= -4_{\text{ten}} \end{aligned}$$

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 32-bit word into a 32-bit register, the point is moot; signed and unsigned loads are identical. MIPS does offer two flavors of byte loads: *load byte* (`lb`) treats the byte as a signed number and thus sign-extends to fill the 24 left-most bits of the register, while *load byte unsigned* (`lbu`) works with unsigned 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.

Unlike the 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 `int` in the program) and the latter *unsigned integers* (`unsigned int`). Some C style guides even recommend declaring the former as `signed int` to keep the distinction clear.

## Hardware/ Software Interface

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  $111 \dots 111_{\text{two}}$ , 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

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

$$2_{\text{ten}} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}}$$

Negating this number by inverting the bits and adding one,

$$\begin{array}{r} 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{\text{two}} \\ + \phantom{1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ } 1_{\text{two}} \\ \hline = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} \\ = -2_{\text{ten}} \end{array}$$

Going the other direction,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}}$$

is first inverted and then incremented:

$$\begin{array}{r} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} \\ + \phantom{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ } 1_{\text{two}} \\ \hline = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} \\ = 2_{\text{ten}} \end{array}$$

**EXAMPLE**

**ANSWER**

Our next shortcut tells us how to convert a binary number represented in  $n$  bits to a number represented with more than  $n$  bits. For example, the immediate field in the load, store, branch, add, and set on less than instructions contains a two's complement 16-bit number, representing  $-32,768_{\text{ten}} (-2^{15})$  to  $32,767_{\text{ten}} (2^{15} - 1)$ . To add the immediate field to a 32-bit register, the computer must convert that 16-bit number to its 32-bit equivalent. 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 word. This shortcut is commonly called *sign extension*.

## EXAMPLE

## ANSWER

### Sign Extension Shortcut

Convert 16-bit binary versions of  $2_{\text{ten}}$  and  $-2_{\text{ten}}$  to 32-bit binary numbers.

The 16-bit binary version of the number 2 is

$$0000\ 0000\ 0000\ 0010_{\text{two}} = 2_{\text{ten}}$$

It is converted to a 32-bit number by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word. The right half gets the old value:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = 2_{\text{ten}}$$

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

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

becomes

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

Creating a 32-bit version of the negative number means copying the sign bit 16 times and placing it on the left:

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2_{\text{ten}}$$

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 word, 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 word size, binary and hexadecimal (see Figure 2.4) bit strings can encode the sign; hence we do not normally use “+” or “−” with binary or hexadecimal notation.

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

1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1000<sub>two</sub>

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

**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:  $00 \dots 00_{\text{two}}$  is positive 0 and  $11 \dots 11_{\text{two}}$  is negative 0. The most negative number,  $10 \dots 000_{\text{two}}$ , represents  $-2,147,483,647_{\text{ten}}$ , 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.

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

## Check Yourself

### one's complement

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

### biased notation

A notation that represents the most negative value by  $00 \dots 000_{\text{two}}$  and the most positive value by  $11 \dots 11_{\text{two}}$ , with 0 typically having the value  $10 \dots 00_{\text{two}}$ , 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.

Since registers are referred to in instructions, there must be a convention to map register names into numbers. In MIPS assembly language, registers `$s0` to `$s7` map onto registers 16 to 23, and registers `$t0` to `$t7` map onto registers 8 to 15. Hence, `$s0` means register 16, `$s1` means register 17, `$s2` means register 18, . . . , `$t0` means register 8, `$t1` means register 9, and so on. We'll describe the convention for the rest of the 32 registers in the following sections.

EXAMPLE

Translating a MIPS Assembly Instruction into a Machine Instruction

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

```
add $t0,$s1,$s2
```

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

ANSWER

The decimal representation is

0	17	18	8	0	32
---	----	----	---	---	----

Each of these segments of an instruction is called a *field*. The first and last fields (containing 0 and 32 in this case) in combination tell the MIPS computer that this instruction performs addition. The second field gives the number of the register that is the first source operand of the addition operation (17 = `$s1`), and the third field gives the other source operand for the addition (18 = `$s2`). The fourth field contains the number of the register that is to receive the sum (8 = `$t0`). The fifth field is unused in this instruction, so it is set to 0. Thus, this instruction adds register `$s1` to register `$s2` and places the sum in register `$t0`.

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

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

This layout of the instruction is called the **instruction format**. As you can see from counting the number of bits, this MIPS instruction takes exactly 32 bits—the same size as a data word. In keeping with our design principle that simplicity favors regularity, all MIPS instructions are 32 bits long.

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

It would appear that you would now be reading and writing long, tedious 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. Since 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.

**instruction format**

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

**machine language**

Binary representation used for communication within a computer system.

**hexadecimal**

Numbers in base 16.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0 <sub>hex</sub>	0000 <sub>two</sub>	4 <sub>hex</sub>	0100 <sub>two</sub>	8 <sub>hex</sub>	1000 <sub>two</sub>	c <sub>hex</sub>	1100 <sub>two</sub>
1 <sub>hex</sub>	0001 <sub>two</sub>	5 <sub>hex</sub>	0101 <sub>two</sub>	9 <sub>hex</sub>	1001 <sub>two</sub>	d <sub>hex</sub>	1101 <sub>two</sub>
2 <sub>hex</sub>	0010 <sub>two</sub>	6 <sub>hex</sub>	0110 <sub>two</sub>	a <sub>hex</sub>	1010 <sub>two</sub>	e <sub>hex</sub>	1110 <sub>two</sub>
3 <sub>hex</sub>	0011 <sub>two</sub>	7 <sub>hex</sub>	0111 <sub>two</sub>	b <sub>hex</sub>	1011 <sub>two</sub>	f <sub>hex</sub>	1111 <sub>two</sub>

**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.

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 0xnnnn for hexadecimal numbers.

**Binary to Hexadecimal and Back**

Convert the following hexadecimal and binary numbers into the other base:

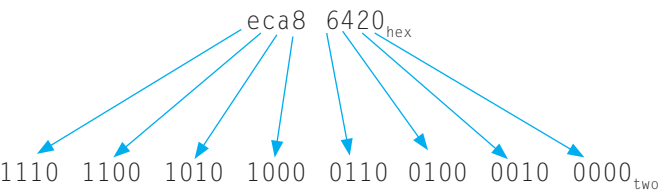
eca8    6420<sub>hex</sub>

0001   0011   0101   0111   1001   1011   1101   1111<sub>two</sub>

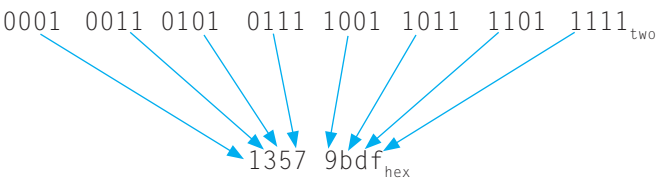
**EXAMPLE**

ANSWER

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



And then the other direction:



MIPS Fields

MIPS fields are given names to make them easier to discuss:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Here is the meaning of each name of the fields in MIPS instructions:

- *op*: Basic operation of the instruction, traditionally called the **opcode**.
- *rs*: The first register source operand.
- *rt*: The second register source operand.
- *rd*: The register destination operand. It gets the result of the operation.
- *shamt*: Shift amount. (Section 2.6 explains shift instructions and this term; it will not be used until then, and hence the field contains zero in this section.)
- *funct*: Function. This field, often called the *function code*, selects the specific variant of the operation in the *op* 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 word 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 constant within the load word instruction would be limited to only 2<sup>5</sup> or 32. This constant is used to select elements from arrays or data structures, and it often needs to be much larger than 32. 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 leads us to the final hardware design principle:

*Design Principle 3:* Good design demands good compromises.

The compromise chosen by the MIPS designers is to keep all instructions the same length, thereby requiring different kinds of instruction formats for different kinds of instructions. For example, the format above is called *R-type* (for register) or *R-format*. A second type of instruction format is called *I-type* (for immediate) or *I-format* and is used by the immediate and data transfer instructions. The fields of I-format are

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

The 16-bit address means a load word instruction can load any word within a region of  $\pm 2^{15}$  or 32,768 bytes ( $\pm 2^{13}$  or 8192 words) of the address in the base register rs. Similarly, add immediate is limited to constants no larger than  $\pm 2^{15}$ . We see that more than 32 registers would be difficult in this format, as the rs and rt fields would each need another bit, making it harder to fit everything in one word.

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

```
lw    $t0,32($s3)    # Temporary reg $t0 gets A[8]
```

Here, 19 (for \$s3) is placed in the rs field, 8 (for \$t0) is placed in the rt field, and 32 is placed in the address field. Note that the meaning of the rt field has changed for this instruction: in a load word instruction, the rt field specifies the *destination* register, which receives the result of the load.

Although multiple formats complicate the hardware, we can reduce the complexity by keeping the formats similar. For example, the first three fields of the R-type and I-type formats are the same size and have the same names; the length of the fourth field in I-type is equal to the sum of the lengths of the last three fields of R-type.

In case you were wondering, the formats are distinguished by the values in the first field: each format is assigned a distinct set of values in the first field (op) so that the hardware knows whether to treat the last half of the instruction as three fields (R-type) or as a single field (I-type). Figure 2.5 shows the numbers used in each field for the MIPS instructions covered so far.

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32 <sub>ten</sub>	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 <sub>ten</sub>	n.a.
add immediate	I	8 <sub>ten</sub>	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35 <sub>ten</sub>	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43 <sub>ten</sub>	reg	reg	n.a.	n.a.	n.a.	address

**FIGURE 2.5 MIPS instruction encoding.** In the table above, “reg” means a register number between 0 and 31, “address” means a 16-bit address, and “n.a.” (not applicable) means this field does not appear in this format. Note that add and sub instructions have the same value in the op field; the hardware uses the funct field to decide the variant of the operation: add (32) or subtract (34).



## EXAMPLE

## Translating MIPS Assembly Language into Machine Language

We can now take an example all the way from what the programmer writes to what the computer executes. If `$t1` has the base of the array `A` and `$s2` corresponds to `h`, the assignment statement

$$A[300] = h + A[300];$$

is compiled into

```
lw    $t0,1200($t1) # Temporary reg $t0 gets A[300]
add   $t0,$s2,$t0    # Temporary reg $t0 gets h + A[300]
sw    $t0,1200($t1) # Stores h + A[300] back into A[300]
```

What is the MIPS 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:

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

The `lw` instruction is identified by 35 (see [Figure 2.5](#)) in the first field (op). The base register 9 (`$t1`) is specified in the second field (rs), and the destination register 8 (`$t0`) is specified in the third field (rt). The offset to select `A[300]` ( $1200 = 300 \times 4$ ) is found in the final field (address).

The `add` instruction that follows is specified with 0 in the first field (op) and 32 in the last field (funct). The three register operands (18, 8, and 8) are found in the second, third, and fourth fields and correspond to `$s2`, `$t0`, and `$t0`.

The `sw` instruction is identified with 43 in the first field. The rest of this final instruction is identical to the `lw` instruction.

Since  $1200_{\text{ten}} = 0000\ 0100\ 1011\ 0000_{\text{two}}$ , the binary equivalent to the decimal form is:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Note the similarity of the binary representations of the first and last instructions. The only difference is in the third bit from the left, which is highlighted here.

The desire to keep all instructions the same size is in conflict 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.

## Hardware/ Software Interface

Figure 2.6 summarizes the portions of MIPS 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 MIPS architecture.

**MIPS machine language**

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
addi	I	8	18	17	100			addi \$s1,\$s2,100
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

**FIGURE 2.6 MIPS architecture revealed through Section 2.5.** The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; the *shamt* field, which Section 2.6 explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single *address* field.

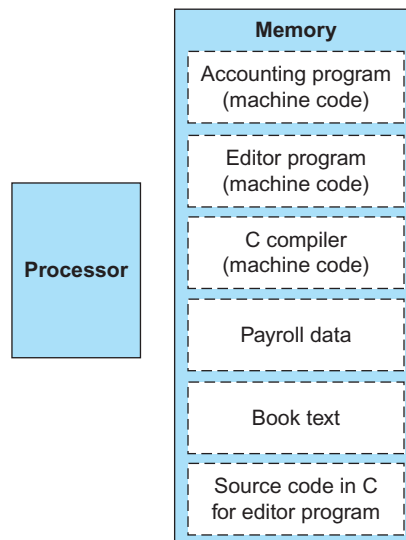
## The BIG Picture

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.

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.



**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.

What MIPS instruction does this represent? Choose from one of the four options below.

Check Yourself

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

- 1. `sub $t0, $t1, $t2`
- 2. `add $t2, $t0, $t1`
- 3. `sub $t2, $t1, $t0`
- 4. `sub $t2, $t0, $t1`

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 MIPS.

“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	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

**FIGURE 2.8** C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

The first class of such operations is called *shifts*. They move all the bits in a word to the left or right, filling the emptied bits with 0s. For example, if register `$s0` contained

0000 0000 0000 0000 0000 0000 0000 1001<sub>two</sub> = 9<sub>ten</sub>

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

0000 0000 0000 0000 0000 0000 1001 0000<sub>two</sub> = 144<sub>ten</sub>

The dual of a shift left is a shift right. The actual name of the two MIPS shift instructions are called *shift left logical* (`sll`) and *shift right logical* (`srl`). The following instruction performs the operation above, assuming that the original value was in register `$s0` and the result should go in register `$t2`:

```
sll $t2,$s0,4 # reg $t2 = reg $s0 << 4 bits
```

We delayed explaining the *shamt* field in the R-format. Used in shift instructions, it stands for *shift amount*. Hence, the machine language version of the instruction above is

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

The encoding of `sll` is 0 in both the `op` and `funct` fields, `rd` contains 10 (register `$t2`), `rt` contains 16 (register `$s0`), and `shamt` contains 4. The `rs` field is unused and thus is set to 0.

Shift left logical provides a bonus benefit. Shifting left by  $i$  bits gives the same 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 `sll` shifts by 4, which gives the same result as multiplying by  $2^4$  or 16. The first bit pattern above represents 9, and  $9 \times 16 = 144$ , the value of the second bit pattern.

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

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 `$t2` contains

```
0000 0000 0000 0000 0000 1101 1100 0000two
```

and register `$t1` contains

```
0000 0000 0000 0000 0011 1100 0000 0000two
```

then, after executing the MIPS instruction

```
and $t0,$t1,$t2 # reg $t0 = reg $t1 & reg $t2
```

the value of register `$t0` would be

```
0000 0000 0000 0000 0000 1100 0000 0000two
```

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 \$t1 and \$t2 are unchanged from the preceding example, the result of the MIPS instruction

```
or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2
```

is this value in register \$t0:

```
0000 0000 0000 0000 0011 1101 1100 0000two
```

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  $\bar{x}$ .

In keeping with the three-operand format, the designers of MIPS decided to include the instruction **NOR** (NOT OR) instead of NOT. If one operand is zero, then it is equivalent to NOT: A NOR 0 = NOT (A OR 0) = NOT (A).

If the register \$t1 is unchanged from the preceding example and register \$t3 has the value 0, the result of the MIPS instruction

```
nor $t0,$t1,$t3 # reg $t0 = ~ (reg $t1 | reg $t3)
```

is this value in register \$t0:

```
1111 1111 1111 1111 1100 0011 1111 1111two
```

Figure 2.8 above shows the relationship between the C and Java operators and the MIPS instructions. Constants are useful in AND and OR logical operations as well as in arithmetic operations, so MIPS also provides the instructions *and immediate* (*andi*) and *or immediate* (*ori*). Constants are rare for NOR, since its main use is to invert the bits of a single operand; thus, the MIPS instruction set architecture has no immediate version of NOR.

**Elaboration:** The full MIPS instruction set also includes exclusive or (XOR), which sets the bit to 1 when two corresponding bits differ, and to 0 when they are the same. C allows *bit fields* or *fields* to be defined within words, both allowing objects to be packed within a word and to match an externally enforced interface such as an I/O device. All fields must fit within a single word. Fields are unsigned integers that can be as short as 1 bit. C compilers insert and extract fields using logical instructions in MIPS: *and*, *or*, *sll*, and *srl*.

**Elaboration:** Logical AND immediate and logical OR immediate put 0s into the upper 16 bits to form a 32-bit constant, unlike *add immediate*, which does sign extension.

Which operations can isolate a field in a word?

1. AND
2. A shift left followed by a shift right

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

**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.

**NOR** A logical bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands. That is, it calculates a 1 only if there is a 0 in *both* operands.

**Check Yourself**

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

## EXAMPLE

## ANSWER

## 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. MIPS assembly language includes two decision-making instructions, similar to an *if* statement with a *go to*. The first instruction is

```
beq register1, register2, L1
```

This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic beq stands for *branch if equal*. The second instruction is

```
bne register1, register2, L1
```

It means go to the statement labeled L1 if the value in register1 does *not* equal the value in register2. The mnemonic bne stands for *branch if not equal*. These two instructions are traditionally called **conditional branches**.

### Compiling *if-then-else* into Conditional Branches

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 \$s0 through \$s4, what is the compiled MIPS code for this C *if* statement?

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

Figure 2.9 shows a flowchart of what the MIPS code should do. The first expression compares for equality, so it would seem that we would want the branch if registers are equal instruction (beq). In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the *if* (the label Else is defined below) and so we use the branch if registers are *not* equal instruction (bne):

```
bne $s3,$s4,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 $s0,$s1,$s2    # 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. To distinguish between conditional and unconditional branches, the MIPS name for this type of instruction is *jump*, abbreviated as *j* (the label `Exit` is defined below).

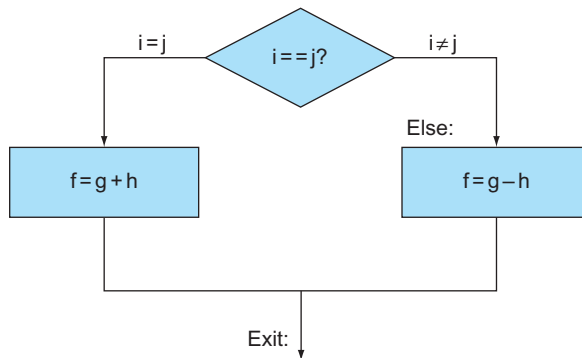
```
j Exit    # go to Exit
```

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 $s0,$s1,$s2 # 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).

**conditional branch** An instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.



**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.



## 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.

#### EXAMPLE

##### Compiling a *while* Loop in C

Here is a traditional loop in C:

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

Assume that *i* and *k* correspond to registers *\$s3* and *\$s5* and the base of the array *save* is in *\$s6*. What is the MIPS assembly code corresponding to this C segment?

#### 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 4 due to the byte addressing problem. Fortunately, we can use shift left logical, since shifting left by 2 bits multiplies by  $2^2$  or 4 (see page 88 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: sll  $t1,$s3,2    # Temp reg $t1 = i * 4
```

To get the address of *save[i]*, we need to add *\$t1* and the base of *save* in *\$s6*:

```
add $t1,$t1,$s6    # $t1 = address of save[i]
```

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

```
lw $t0,0($t1)      # Temp reg $t0 = save[i]
```

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

```
bne $t0,$s5, Exit  # go to Exit if save[i] ≠ k
```

The next instruction adds 1 to `i`:

```
addi $s3,$s3,1      # 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:

```
      j      Loop      # go to Loop
Exit:
```

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

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.

## Hardware/ Software Interface

**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 sometimes it is useful to see if a variable is less than another variable. For example, a *for* loop may want to test to see if the index variable is less than 0. Such comparisons are accomplished in MIPS assembly language with an instruction that compares two registers and sets a third register to 1 if the first is less than the second; otherwise, it is set to 0. The MIPS instruction is called *set on less than*, or `slt`. For example,

```
slt    $t0, $s3, $s4    # $t0 = 1 if $s3 < $s4
```

means that register `$t0` is set to 1 if the value in register `$s3` is less than the value in register `$s4`; otherwise, register `$t0` is set to 0.

Constant operands are popular in comparisons, so there is an immediate version of the set on less than instruction. To test if register `$s2` is less than the constant 10, we can just write

```
slti    $t0,$s2,10      # $t0 = 1 if $s2 < 10
```

MIPS compilers use the `slt`, `slti`, `beq`, `bne`, and the fixed value of 0 (always available by reading register `$zero`) to create all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal.

## Hardware/ Software Interface

Heeding von Neumann's warning about the simplicity of the "equipment," the MIPS architecture doesn't include branch on less than because it is too complicated; either it would stretch the clock cycle time or it would take extra clock cycles per instruction. Two faster instructions are more useful.

## Hardware/ Software Interface

Comparison instructions must 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.)

MIPS offers two versions of the set on less than comparison to handle these alternatives. *Set on less than* (`slt`) and *set on less than immediate* (`slti`) work with signed integers. Unsigned integers are compared using *set on less than unsigned* (`sltu`) and *set on less than immediate unsigned* (`sltiu`).

## EXAMPLE

### Signed versus Unsigned Comparison

Suppose register `$s0` has the binary number

```
1111 1111 1111 1111 1111 1111 1111 1111two
```

and that register `$s1` has the binary number

```
0000 0000 0000 0000 0000 0000 0000 0001two
```

What are the values of registers `$t0` and `$t1` after these two instructions?

```
slt      $t0, $s0, $s1 # signed comparison
sltu     $t1, $s0, $s1 # unsigned comparison
```

## ANSWER

The value in register `$s0` represents  $-1_{\text{ten}}$  if it is an integer and  $4,294,967,295_{\text{ten}}$  if it is an unsigned integer. The value in register `$s1` represents  $1_{\text{ten}}$  in either case. Then register `$t0` has the value 1, since  $-1_{\text{ten}} < 1_{\text{ten}}$ , and register `$t1` has the value 0, since  $4,294,967,295_{\text{ten}} > 1_{\text{ten}}$ .

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  $x < y$  also checks if  $x$  is negative as well as if  $x$  is less than  $y$ .

### Bounds Check Shortcut

Use this shortcut to reduce an index-out-of-bounds check: jump to `IndexOutOfBounds` if `$s1 ≥ $t2` or if `$s1` is negative.

The checking code just uses `u` to do both checks:

```
sltu $t0,$s1,$t2 # $t0=0 if $s1>=length or $s1<0
beq  $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

### EXAMPLE

### ANSWER

## 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 **jump address table** or **jump table**, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the jump table into a register. It then needs to jump using the address in the register. To support such situations, computers like MIPS include a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction. We'll see an even more popular use of `jr` in the next section.

**jump address table** Also called **jump 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

**Elaboration:** If you have heard about *delayed branches*, covered in Chapter 4, don't worry: the MIPS assembler makes them invisible to the assembly language programmer.

- I. C has many statements for decisions and loops, while MIPS has few. Which of the following do or do 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 MIPS doesn't?
  1. Logical operations AND and OR 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.



ABSTRACTION

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

## 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.

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 his employer.

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. MIPS software follows the following convention for procedure calling in allocating its 32 registers:

- `$a0-$a3`: four argument registers in which to pass parameters
- `$v0-$v1`: two value registers in which to return values
- `$ra`: one return address register to return to the point of origin

In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures: it jumps to an address and simultaneously saves the address of the following instruction in register `$ra`. The **jump-and-link instruction** (`jal`) is simply written

```
jal ProcedureAddress
```

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 `$ra` (register 31), is called the **return address**. The return address is needed because the same procedure could be called from several parts of the program.

To support such situations, computers like MIPS use *jump register* instruction (`jr`), introduced above to help with case statements, meaning an unconditional jump to the address specified in a register:

```
jr    $ra
```

#### jump-and-link instruction

An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register (`$ra` in MIPS).

**return address** A link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in register `$ra`.

**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.

**program counter (PC)** The register containing the address of the instruction in the program being executed.

**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 MIPS, it is register `$sp`.

**push** Add element to stack.

**pop** Remove element from stack.

The jump register instruction jumps to the address stored in register `$ra`—which is just what we want. Thus, the calling program, or **caller**, puts the parameter values in `$a0–$a3` and uses `jal X` to jump to procedure `X` (sometimes named the **callee**). The callee then performs the calculations, places the results in `$v0` and `$v1`, and returns control to the caller using `jr $ra`.

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 MIPS architecture, although a more sensible name would have been *instruction address register*. The `jal` instruction actually saves `PC + 4` in register `$ra` to link to the following instruction to set up the procedure return.

## Using More Registers

Suppose a compiler needs more registers for a procedure than the four argument and two return value 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 above.

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. The **stack pointer** is adjusted by one word for each register that is saved or restored. MIPS software reserves register 29 for the stack pointer, giving it the obvious name `$sp`. 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**.

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.

## EXAMPLE

### Compiling a C Procedure That Doesn't Call Another Procedure

Let's turn the example on page 65 from Section 2.2 into a C procedure:

```
int leaf_example (int g, int h, int i, int j)
{
    int f;

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

What is the compiled MIPS assembly code?

**ANSWER**

The parameter variables *g*, *h*, *i*, and *j* correspond to the argument registers \$a0, \$a1, \$a2, and \$a3, and *f* corresponds to \$s0. 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 68, which uses two temporary registers. Thus, we need to save three registers: \$s0, \$t0, and \$t1. We “push” the old values onto the stack by creating space for three words (12 bytes) on the stack and then store them:

```
addi $sp, $sp, -12 # adjust stack to make room for 3 items
sw  $t1, 8($sp)   # save register $t1 for use afterwards
sw  $t0, 4($sp)   # save register $t0 for use afterwards
sw  $s0, 0($sp)   # save register $s0 for use afterwards
```

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

The next three statements correspond to the body of the procedure, which follows the example on page 68:

```
add $t0, $a0, $a1 # register $t0 contains g + h
add $t1, $a2, $a3 # register $t1 contains i + j
sub $s0, $t0, $t1 # f = $t0 - $t1, which is (g + h) - (i + j)
```

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

```
add $v0, $s0, $zero # returns f ($v0 = $s0 + 0)
```

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

```
lw $s0, 0($sp) # restore register $s0 for caller
lw $t0, 4($sp) # restore register $t0 for caller
lw $t1, 8($sp) # restore register $t1 for caller
addi $sp, $sp, 12 # adjust stack to delete 3 items
```

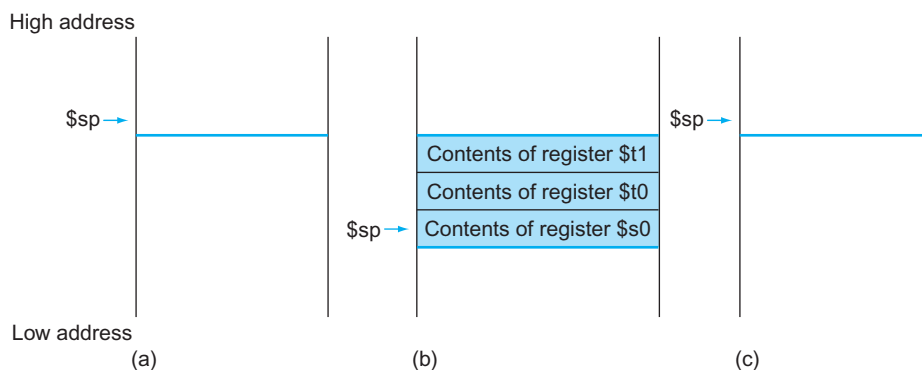
The procedure ends with a jump register using the return address:

```
jr  $ra # jump 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, MIPS software separates 18 of the registers into two groups:

- \$t0–\$t9: temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- \$s0–\$s7: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)





**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 word in the stack in this drawing.

This simple convention reduces register spilling. In the example above, since the caller does not expect registers `$t0` and `$t1` to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore `$s0`, 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, more care must also be taken 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 `$a0` and then using `jal A`. Then suppose that procedure A calls procedure B via `jal B` with an argument of 7, also placed in `$a0`. Since A hasn’t finished its task yet, there is a conflict over the use of register `$a0`. Similarly, there is a conflict over the return address in register `$ra`, 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 onto the stack, just as we did with the saved registers. The caller pushes any argument registers (`$a0–$a3`) or temporary registers (`$t0–$t9`) that are needed after the call. The callee pushes the return address register `$ra` and any saved registers (`$s0–$s7`) used by the callee. The stack pointer `$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:

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

What is the MIPS assembly code?

The parameter variable `n` corresponds to the argument register `$a0`. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and `$a0`:

#### ANSWER

```
fact:
    addi $sp, $sp, -8 # adjust stack for 2 items
    sw   $ra, 4($sp) # save the return address
    sw   $a0, 0($sp) # save the argument n
```

The first time `fact` is called, `sw` 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`.

```
slti $t0,$a0,1    # test for n < 1
beq  $t0,$zero,L1 # if n >= 1, 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 `$v0`. It then pops the two saved values off the stack and jumps to the return address:

```
addi $v0,$zero,1 # return 1
addi $sp,$sp,8   # pop 2 items off stack
jr   $ra         # return to caller
```

Before popping two items off the stack, we could have loaded `$a0` and `$ra`. Since `$a0` and `$ra` 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 $a0,$a0,-1 # n >= 1: argument gets (n - 1)
    jal fact        # call fact with (n - 1)
```

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

```
lw    $a0, 0($sp) # return from jal: restore argument n
lw    $ra, 4($sp) # restore the return address
addi  $sp, $sp, 8  # adjust stack pointer to pop 2 items
```

Next, the value register `$v0` gets the product of old argument `$a0` and the current value of the value register. We assume a multiply instruction is available, even though it is not covered until Chapter 3:

```
mul   $v0,$a0,$v0  # return n * fact (n - 1)
```

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

```
jr    $ra           # return to the caller
```

Hardware/  
Software  
Interface

**global pointer** The register that is reserved to point to the static area.

A C variable is generally a location in storage, and its interpretation depends both on its *type* and *storage class*. Examples 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, MIPS software reserves another register, called the **global pointer**, or `$gp`.

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

Preserved	Not preserved
Saved registers: \$s0-\$s7	Temporary registers: \$t0-\$t9
Stack pointer register: \$sp	Argument registers: \$a0-\$a3
Return address register: \$ra	Return value registers: \$v0-\$v1
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 frame pointer register or on the global pointer register, discussed in the following subsections, they are also preserved.

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.

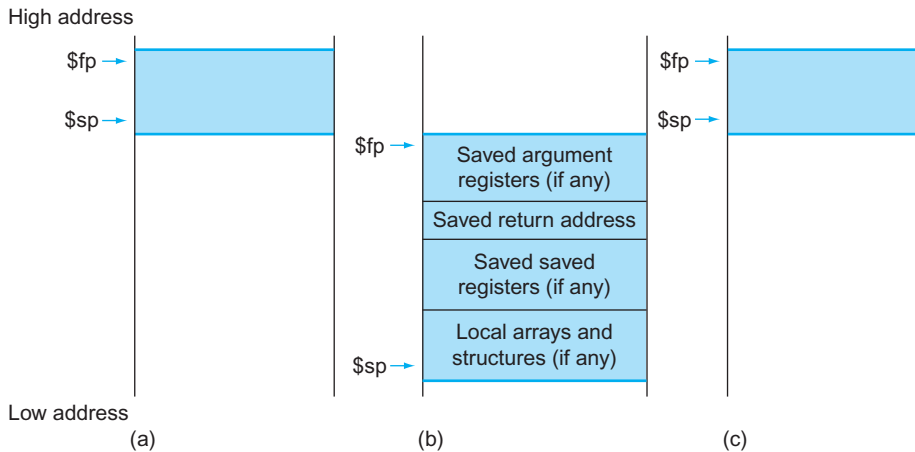
## Allocating Space for New Data on the Stack

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.

Some MIPS software uses a **frame pointer** (\$fp) to point to the first word 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 and exit of the procedure.

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

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

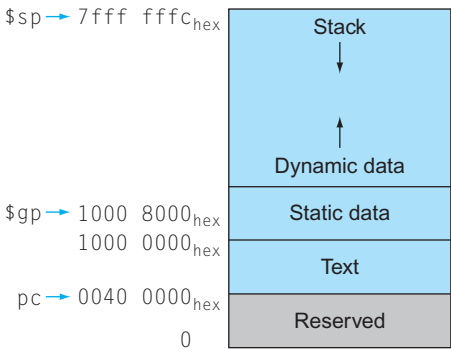


**FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call.** The frame pointer (\$fp) points to the first word of the frame, often a saved argument register, and the stack pointer (\$sp) points to the top of the stack. The stack is adjusted to make room 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 MIPS Reference Data Card at the front of this book.

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 MIPS convention for allocation of memory. The stack starts in the high end of memory and grows down. The first part of the low end of memory is reserved, followed by the home of the MIPS 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 MIPS memory allocation for program and data.** These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to  $7fff\ fffc_{hex}$  and grows down toward the data segment. At the other end, the program code (“text”) starts at  $0040\ 0000_{hex}$ . The static data starts at  $1000\ 0000_{hex}$ . 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. The global pointer,  $\$gp$ , is set to an address to make it easy to access data. It is initialized to  $1000\ 8000_{hex}$  so that it can access from  $1000\ 0000_{hex}$  to  $1000\ ffff_{hex}$  using the positive and negative 16-bit offsets from  $\$gp$ . This information is also found in Column 4 of the MIPS 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. Memory allocation is controlled by programs in C, and it is the source of many common and difficult bugs. Forgetting to free space leads to a “memory leak,” which eventually 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 MIPS assembly language. This convention is another example of making the **common case fast**: most procedures can be satisfied with up to 4 arguments, 2 registers for a return value, 8 saved registers, and 10 temporary registers without ever going to memory.



COMMON CASE FAST

Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0-\$v1	2-3	Values for results and expression evaluation	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes

**FIGURE 2.14 MIPS register conventions.** Register 1, called \$at, is reserved for the assembler (see Section 2.12), and registers 26-27, called \$k0-\$k1, are reserved for the operating system. This information is also found in Column 2 of the MIPS Reference Data Card at the front of this book.

**Elaboration:** What if there are more than four parameters? The MIPS convention is to place the extra parameters on the stack just above the frame pointer. The procedure then expects the first four parameters to be in registers \$a0 through \$a3 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 GNU MIPS C compiler uses a frame pointer, but the C compiler from MIPS does not; it treats register 30 as another save register (\$s8).

**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:

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

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 \$a0 = n and \$a1 = acc):

```
sum: slti $t0, $a0, 1           # test if n <= 0
     bne $t0, $zero, sum_exit   # go to sum_exit if n <= 0
     add$al, $al, $a0           # add n to acc
     addi$a0, $a0, -1           # subtract 1 from n
     j sum                      # go to sum
sum_exit:
     add$v0, $al, $zero         # return value acc
     jr $ra                    # return to caller
```

Check Yourself

Which of the following statements about C and Java are 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 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. This information is also found in Column 3 of the MIPS Reference Data Card at the front of this book.

### ASCII versus Binary Numbers

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?

One billion is 1,000,000,000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be  $(10 \times 8)/32$  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.

A series of instructions can extract a byte from a word, so load word and store word are sufficient for transferring bytes as well as words. Because of the popularity of text in some programs, however, MIPS provides instructions to move bytes. *Load byte* (`lb`) 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

```
lb $t0,0($sp)      # Read byte from source
sb $t0,0($gp)      # 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, 0. (As we shall see, Java uses the first option.)

### EXAMPLE

### ANSWER



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

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

```
void strcpy (char x[], char y[])
{
    int i;

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

What is the MIPS assembly code?

**ANSWER**

Below is the basic MIPS assembly code segment. Assume that base addresses for arrays `x` and `y` are found in `$a0` and `$a1`, while `i` is in `$s0`. `strcpy` adjusts the stack pointer and then saves the saved register `$s0` on the stack:

```
strcpy:
    addi    $sp,$sp,-4    # adjust stack for 1 more item
    sw      $s0, 0($sp)  # save $s0
```

To initialize `i` to 0, the next instruction sets `$s0` to 0 by adding 0 to 0 and placing that sum in `$s0`:

```
add    $s0,$zero,$zero # 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    $t1,$s0,$a1 # address of y[i] in $t1
```

Note that we don't have to multiply `i` by 4 since `y` is an array of *bytes* and not of words, as in prior examples.

To load the character in `y[i]`, we use load byte unsigned, which puts the character into `$t2`:

```
lbu    $t2, 0($t1) # $t2 = y[i]
```

A similar address calculation puts the address of `x[i]` in `$t3`, and then the character in `$t2` is stored at that address.

```

add    $t3,$s0,$a0 # address of x[i] in $t3
sb     $t2, 0($t3) # 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     $t2,$zero,L2 # if y[i] == 0, go to L2

```

If not, we increment *i* and loop back:

```

addi    $s0, $s0, 1 # i = i + 1
j       L1          # go to L1

```

If we don't loop back, it was the last character of the string; we restore *\$s0* and the stack pointer, and then return.

```

L2: lw    $s0, 0($sp) # y[i] == 0: end of string.
          # Restore old $s0
addi    $sp,$sp,4     # pop 1 word off stack
jr      $ra           # 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 *\$s0*. Hence, instead of thinking of the *\$t* 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<sub>hex</sub>, and Cyrillic at 0400<sub>hex</sub>. 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](http://www.unicode.org).

The MIPS instruction set has explicit instructions to load and store such 16-bit quantities, called *halfwords*. *Load half* (`lh`) loads a halfword from memory, placing it in the rightmost 16 bits of a register. Like load byte, *load half* (`lh`) treats the halfword as a signed number and thus sign-extends to fill the 16 leftmost bits of the register, while *load halfword unsigned* (`lhu`) works with unsigned integers. Thus, `lhu` is the more popular of the two. *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 $t0,0($sp) # Read halfword (16 bits) from source
sh $t0,0($gp)  # Write halfword (16 bits) to destination
```

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:** MIPS software tries to keep the stack aligned to word addresses, allowing the program to always use `lw` and `sw` (which must be aligned) to access the stack. This convention means that a `char` variable allocated on the stack occupies 4 bytes, even though it needs less. However, a C string variable or an array of bytes *will* pack 4 bytes per word, and a Java string variable or array of shorts packs 2 halfwords per word.

**Elaboration:** Reflecting the international nature of the web, most web pages today use Unicode instead of ASCII.

- I. Which of the following statements about characters and strings in C and Java are 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  $1,000,000,000_{\text{ten}}$  takes the most memory space?
  1. `int` in C
  2. `string` in C
  3. `string` in Java

**Check  
Yourself**

## 2.10

### MIPS Addressing for 32-bit Immediates and Addresses

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

32-Bit Immediate Operands

Although constants are frequently short and fit into the 16-bit field, sometimes they are bigger. The MIPS instruction set includes the instruction *load upper immediate* (`lui`) specifically to set the upper 16 bits of a constant in a register, allowing a subsequent instruction to specify the lower 16 bits of the constant. Figure 2.17 shows the operation of `lui`.

EXAMPLE

ANSWER

Loading a 32-Bit Constant

What is the MIPS assembly code to load this 32-bit constant into register `$s0`?

0000 0000 0011 1101 0000 1001 0000 0000

First, we would load the upper 16 bits, which is 61 in decimal, using `lui`:

`lui $s0, 61` # 61 decimal = 0000 0000 0011 1101 binary

The value of register `$s0` afterward is

0000 0000 0011 1101 0000 0000 0000 0000

The next step is to insert the lower 16 bits, whose decimal value is 2304:

`ori $s0, $s0, 2304` # 2304 decimal = 0000 1001 0000 0000

The final value in register `$s0` is the desired value:

0000 0000 0011 1101 0000 1001 0000 0000

The machine language version of `lui $t0, 255` # `$t0` is register 8:

001111	00000	01000	0000 0000 1111 1111
--------	-------	-------	---------------------

Contents of register `$t0` after executing `lui $t0, 255`:

0000 0000 1111 1111	0000 0000 0000 0000
---------------------	---------------------

**FIGURE 2.17 The effect of the `lui` instruction.** The instruction `lui` transfers the 16-bit immediate constant field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s.

## 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. If this job falls to the assembler, as it does for MIPS software, then the assembler must have a temporary register available in which to create the long values. This need is a reason for the register `$at` (assembler temporary), which is reserved for the assembler.

Hence, the symbolic representation of the MIPS 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.

**Elaboration:** Creating 32-bit constants needs care. The instruction `addi` copies the left-most bit of the 16-bit immediate field of the instruction into the upper 16 bits of a word. *Logical or immediate* from Section 2.6 loads 0s into the upper 16 bits and hence is used by the assembler in conjunction with `lui` to create 32-bit constants.

### Addressing in Branches and Jumps

The MIPS jump instructions have the simplest addressing. They use the final MIPS instruction format, called the *J-type*, which consists of 6 bits for the operation field and the rest of the bits for the address field. Thus,

```
j 10000    # go to location 10000
```

could be assembled into this format (it's actually a bit more complicated, as we will see):

2	10000
6 bits	26 bits

where the value of the jump opcode is 2 and the jump address is 10000.

Unlike the jump instruction, the conditional branch instruction must specify two operands in addition to the branch address. Thus,

```
bne $s0,$s1,Exit # go to Exit if $s0 ≠ $s1
```

is assembled into this instruction, leaving only 16 bits for the branch address:

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

If addresses of the program had to fit in this 16-bit field, it would mean that no program could be bigger than  $2^{16}$ , 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 address, so that a branch instruction would calculate the following:

$$\text{Program counter} = \text{Register} + \text{Branch address}$$

This sum allows the program to be as large as  $2^{32}$  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  $\pm 2^{15}$  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 much smaller than  $2^{16}$  words, so the PC is the ideal choice.

This form of branch addressing is called **PC-relative addressing**. As we shall see in Chapter 4, it is convenient for the hardware to increment the PC early to point to the next instruction. Hence, the MIPS address is actually relative to the address of the following instruction (PC + 4) as opposed to the current instruction (PC). It is yet another example of making the **common case fast**, which in this case is addressing nearby instructions.

Like most recent computers, MIPS uses PC-relative addressing for all conditional branches, because the destination of these instructions is likely to be close to the branch. On the other hand, jump-and-link instructions invoke procedures that have no reason to be near the call, so they normally use other forms of addressing. Hence, the MIPS architecture offers long addresses for procedure calls by using the J-type format for both jump and jump-and-link instructions.

Since all MIPS instructions are 4 bytes long, MIPS stretches the distance of the branch by having PC-relative addressing refer to the number of *words* to the next instruction instead of the number of bytes. Thus, the 16-bit field can branch four times as far by interpreting the field as a relative word address rather than as a relative byte address. Similarly, the 26-bit field in jump instructions is also a word address, meaning that it represents a 28-bit byte address.

**Elaboration:** Since the PC is 32 bits, 4 bits must come from somewhere else for jumps. The MIPS jump instruction replaces only the lower 28 bits of the PC, leaving the upper 4 bits of the PC unchanged. The loader and linker (Section 2.12) must be careful to avoid placing a program across an address boundary of 256 MB (64 million instructions); otherwise, a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register.

### PC-relative

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



COMMON CASE FAST

**Showing Branch Offset in Machine Language****EXAMPLE**

The *while* loop on pages 92–93 was compiled into this MIPS assembler code:

```

Loop: sll $t1,$s3,2      # Temp reg $t1 = 4 * i
      add $t1,$t1,$s6    # $t1 = address of save[i]
      lw  $t0,0($t1)     # Temp reg $t0 = save[i]
      bne $t0,$s5, Exit  # go to Exit if save[i] ≠ k
      addi $s3,$s3,1     # i = i + 1
      j   Loop          # go to Loop
Exit:

```

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

The assembled instructions and their addresses are:

**ANSWER**

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

Remember that MIPS instructions have byte addresses, so addresses of sequential words differ by 4, the number of bytes in a word. The *bne* instruction on the fourth line adds 2 words or 8 bytes to the address of the *following* instruction (80016), specifying the branch destination relative to that following instruction ( $8 + 80016$ ) instead of relative to the branch instruction ( $12 + 80012$ ) or using the full destination address (80024). The jump instruction on the last line does use the full address ( $20000 \times 4 = 80000$ ), 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 16 bits of the conditional branch instruction. The assembler comes to the rescue just as it did with large addresses or constants: it inserts an unconditional jump to the branch target, and inverts the condition so that the branch decides whether to skip the jump.

### EXAMPLE

#### Branching Far Away

Given a branch on register \$s0 being equal to register \$s1,

```
beq    $s0, $s1, L1
```

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

### ANSWER

These instructions replace the short-address conditional branch:

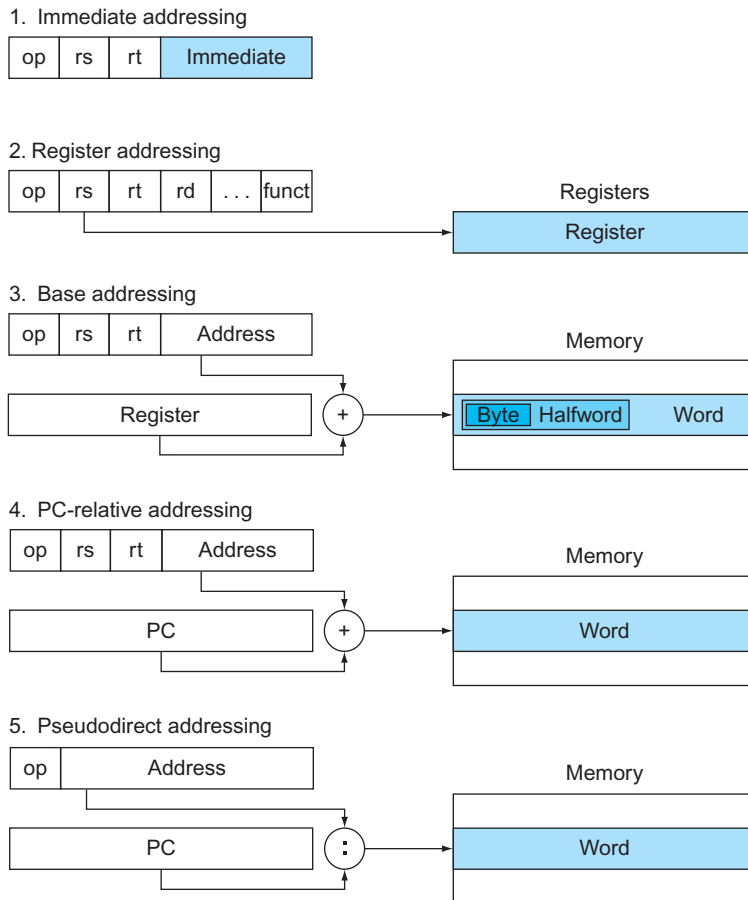
```
       bne    $s0, $s1, L2
       j      L1
L2:
```

**addressing mode** One of several addressing regimes delimited by their varied use of operands and/or addresses.

## MIPS Addressing Mode Summary

Multiple forms of addressing are generically called **addressing modes**. Figure 2.18 shows how operands are identified for each addressing mode. The MIPS addressing modes 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
5. *Pseudodirect addressing*, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC



**FIGURE 2.18 Illustration of the five MIPS 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, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of 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.

Although we show MIPS as having 32-bit addresses, nearly all microprocessors (including MIPS) have 64-bit address extensions (see [Appendix E](#) and [Section 2.18](#)). These extensions were in response to the needs of software for larger programs. The process of instruction set extension allows architectures to expand in such a way that is able to move software compatibly upward to the next generation of architecture.

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 2.19](#) shows the MIPS encoding of the fields for the MIPS machine language. This figure helps when translating by hand between assembly language and machine language.

EXAMPLE

Decoding Machine Code

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

00af8020hex

ANSWER

The first step in converting hexadecimal to binary is to find the op fields:

(Bits: 31 28 26 5 2 0)  
0000 0000 1010 1111 1000 0000 0010 0000

We look at the op field to determine the operation. Referring to [Figure 2.19](#), when bits 31–29 are 000 and bits 28–26 are 000, it is an R-format instruction. Let’s reformat the binary instruction into R-format fields, listed in [Figure 2.20](#):

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

The bottom portion of [Figure 2.19](#) determines the operation of an R-format instruction. In this case, bits 5–3 are 100 and bits 2–0 are 000, which means this binary pattern represents an add instruction.

We decode the rest of the instruction by looking at the field values. The decimal values are 5 for the rs field, 15 for rt, and 16 for rd (shamt is unused). [Figure 2.14](#) shows that these numbers represent registers \$a1, \$t7, and \$s0. Now we can reveal the assembly instruction:

add \$s0,\$a1,\$t7

op(31:26)								
28–26	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
31–29								
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						
op(31:26)=010000 (TLB), rs(25:21)								
23–21	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
25–24								
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								
op(31:26)=000000 (R-format), funct(5:0)								
2–0	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
5–3								
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jralr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

**FIGURE 2.19 MIPS instruction encoding.** This notation gives the value of a field by row and by column. For example, the top portion of the figure shows **load word** in row number 4 (100<sub>two</sub> for bits 31–29 of the instruction) and column number 3 (011<sub>two</sub> for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011<sub>two</sub>. Underscore means the field is used elsewhere. For example, R-format in row 0 and column 0 (op = 000000<sub>two</sub>) is defined in the bottom part of the figure. Hence, **subtract** in row 4 and column 2 of the bottom section means that the funct field (bits 5–0) of the instruction is 100010<sub>two</sub> and the op field (bits 31–26) is 000000<sub>two</sub>. The floating point value in row 2, column 1 is defined in Figure 3.18 in Chapter 3. Bltz/gez is the opcode for four instructions found in Appendix A: bltz, bgez, bltzal, and bgezal. This chapter describes instructions given in full name using color, while Chapter 3 describes instructions given in mnemonics using color. Appendix A covers all instructions.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

**FIGURE 2.20 MIPS instruction formats.**

Figure 2.20 shows all the MIPS instruction formats. Figure 2.1 on page 64 shows the MIPS assembly language revealed in this chapter. The remaining hidden portion of MIPS instructions deals mainly with arithmetic and real numbers, which are covered in the next chapter.

### Check Yourself

- I. What is the range of addresses for conditional branches in MIPS ( $K = 1024$ )?
  1. Addresses between 0 and  $64K - 1$
  2. Addresses between 0 and  $256K - 1$
  3. Addresses up to about 32K before the branch to about 32K after
  4. Addresses up to about 128K before the branch to about 128K after
- II. What is the range of addresses for jump and jump and link in MIPS ( $M = 1024K$ )?
  1. Addresses between 0 and  $64M - 1$
  2. Addresses between 0 and  $256M - 1$
  3. Addresses up to about 32M before the branch to about 32M after
  4. Addresses up to about 128M before the branch to about 128M after
  5. Anywhere within a block of 64M addresses where the PC supplies the upper 6 bits
  6. Anywhere within a block of 256M addresses where the PC supplies the upper 4 bits
- III. What is the MIPS assembly language instruction corresponding to the machine instruction with the value  $0000\ 0000_{\text{hex}}?$ 
  1. j
  2. R-format
  3. addi
  4. sll
  5. mfc0
  6. Undefined opcode: there is no legal instruction that corresponds to 0

## 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.

For example, recall the analogy of the eight reporters writing a story on page 44 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 that the primitives will be used by system programmers 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 interchanges 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



### PARALLELISM

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

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 simultaneously: this race is broken, 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 were 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 instruction pair.

In MIPS this pair of instructions includes a special load called a *load linked* and a special store called a *store conditional*. These instructions are used in sequence: if the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails. The store conditional is defined to both store the value of a (presumably different) register in memory *and* to change the value of that register to a 1 if it succeeds and to a 0 if it fails. Since the load linked returns the initial value, and the store conditional returns 1 only if it succeeds, the following sequence implements an atomic exchange on the memory location specified by the contents of \$s1:

```
again: addi $t0,$zero,1      ;copy locked value
      ll     $t1,0($s1)      ;load linked
      sc     $t0,0($s1)      ;store conditional
      beq    $t0,$zero,again ;branch if store fails
      add    $s4,$zero,$t1   ;put load value in $s4
```

Any time a processor intervenes and modifies the value in memory between the `ll` and `sc` instructions, the `sc` returns 0 in `$t0`, causing the code sequence to try again. At the end of this sequence the contents of `$s4` and the memory location specified by `$s1` 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).

An advantage of the load linked/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 `ll` and the `sc`, but not too many.

Since the store conditional will fail after either another attempted store to the load linked address or any exception, care must be taken in choosing which instructions are inserted between the two instructions. In particular, only register-register instructions can safely be permitted; otherwise, it is possible to create deadlock situations where the processor can never complete the `sc` because of repeated page faults. In addition, the number of instructions between the load linked 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.

When do you use primitives like load linked 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

**Check  
Yourself**

## 2.12 Translating and Starting a Program

This section describes the four steps in transforming a C program in a file on disk into a program running on a computer. [Figure 2.21](#) shows the translation hierarchy. Some systems combine these steps to reduce translation time, but these are the logical four phases that programs go through. This section follows this translation hierarchy.

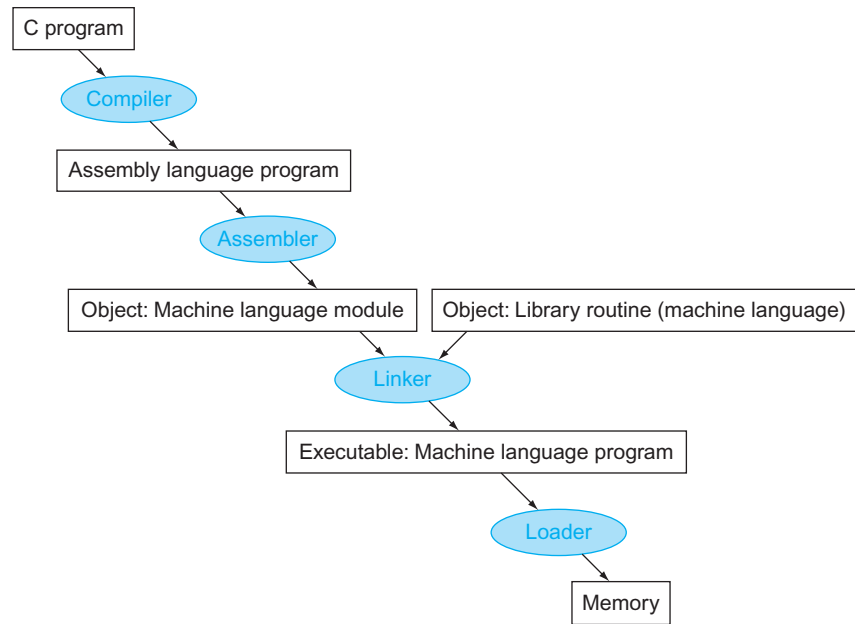
### Compiler

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.





**FIGURE 2.21 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 named `x.s`, object files are named `x.o`, statically linked library routines are `x.a`, dynamically linked library routines 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.

## 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**.

As mentioned above, the MIPS hardware makes sure that register `$zero` always has the value 0. That is, whenever register `$zero` is used, it supplies a 0, and the programmer cannot change the value of register `$zero`. Register `$zero` is used to create the assembly language instruction that copies the contents of one register to another. Thus the MIPS assembler accepts this instruction even though it is not found in the MIPS architecture:

```
move $t0,$t1      # register $t0 gets register $t1
```

### pseudoinstruction

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

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

```
add $t0,$zero,$t1 # register $t0 gets 0 + register $t1
```

The MIPS assembler also converts `blt` (branch on less than) into the two instructions `slt` and `bne` mentioned in the example on page 95. Other examples include `bgt`, `bge`, and `ble`. It also converts branches to faraway locations into a branch and jump. As mentioned above, the MIPS assembler allows 32-bit constants to be loaded into a register despite the 16-bit limit of the immediate instructions.

In summary, pseudoinstructions give MIPS a richer set of assembly language instructions than those implemented by the hardware. The only cost is reserving one register, `$at`, for use by the assembler. If you are going to write assembly programs, use pseudoinstructions to simplify your task. To understand the MIPS architecture and be sure to get best performance, however, study the real MIPS instructions found in [Figures 2.1 and 2.19](#).

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. MIPS assemblers use hexadecimal.

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.

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.

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

- 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.

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, jump 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. The reason a linker is useful is that it is much faster to patch code than it is to recompile and reassemble.

If all external references are resolved, the linker next determines the memory locations each module will occupy. Recall that [Figure 2.13](#) on page 104 shows the MIPS 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.

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

**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 words X and Y.

Object file header			
	Name	Procedure A	
	Text size	100 <sub>hex</sub>	
	Data size	20 <sub>hex</sub>	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	–	
	B	–	
Object file header			
	Name	Procedure B	
	Text size	200 <sub>hex</sub>	
	Data size	30 <sub>hex</sub>	
Text segment	Address	Instruction	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	sw	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 2.13 on page 104, we know that the text segment starts at address  $40\ 0000_{\text{hex}}$  and the data segment at  $1000\ 0000_{\text{hex}}$ . 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  $100_{\text{hex}}$  bytes and its data is  $20_{\text{hex}}$  bytes, so the starting address for procedure B text is  $40\ 0100_{\text{hex}}$ , and its data starts at  $1000\ 0020_{\text{hex}}$ .

Executable file header		
	Text size	$300_{\text{hex}}$
	Data size	$50_{\text{hex}}$
Text segment	Address	Instruction
	$0040\ 0000_{\text{hex}}$	<code>lw \$a0, <math>8000_{\text{hex}}</math>(\$gp)</code>
	$0040\ 0004_{\text{hex}}$	<code>jal <math>40\ 0100_{\text{hex}}</math></code>
	...	...
	$0040\ 0100_{\text{hex}}$	<code>sw \$a1, <math>8020_{\text{hex}}</math>(\$gp)</code>
	$0040\ 0104_{\text{hex}}$	<code>jal <math>40\ 0000_{\text{hex}}</math></code>
	...	...
Data segment	Address	
	$1000\ 0000_{\text{hex}}$	(X)
	...	...
	$1000\ 0020_{\text{hex}}$	(Y)
	...	...

Figure 2.13 also shows that the text segment starts at address  $40\ 0000_{\text{hex}}$  and the data segment at  $1000\ 0000_{\text{hex}}$ . 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  $100_{\text{hex}}$  bytes and its data is  $20_{\text{hex}}$  bytes, so the starting address for procedure B text is  $40\ 0100_{\text{hex}}$ , and its data starts at  $1000\ 0020_{\text{hex}}$ .

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 two types here:

1. The `lws` are easy because they use pseudodirect addressing. The `jal` at address  $40\ 0004_{\text{hex}}$  gets  $40\ 0100_{\text{hex}}$  (the address of procedure B) in its address field, and the `jal` at  $40\ 0104_{\text{hex}}$  gets  $40\ 0000_{\text{hex}}$  (the address of procedure A) in its address field.
2. The load and store addresses are harder because they are relative to a base register. This example uses the global pointer as the base register. Figure 2.13 shows that `$gp` is initialized to  $1000\ 8000_{\text{hex}}$ . To get the address  $1000\ 0000_{\text{hex}}$  (the address of word X), we place  $8000_{\text{hex}}$  in the address field of `lw` at address  $40\ 0000_{\text{hex}}$ . Similarly, we place  $8020_{\text{hex}}$  in the address field of `sw` at address  $40\ 0100_{\text{hex}}$  to get the address  $1000\ 0020_{\text{hex}}$  (the address of word Y).

**Elaboration:** Recall that MIPS instructions are word aligned, so `jal` drops the right two bits to increase the instruction's address range. Thus, it uses 26 bits to create a 28-bit byte address. Hence, the actual address in the lower 26 bits of the `jal` instruction in this example is `10 0040hex`, rather than `40 0100hex`.

## 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:

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 machine registers and sets the stack pointer to the first free location.
6. Jumps 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.

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

Sections A.3 and A.4 in Appendix A describe linkers and loaders in more detail.

## 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 is 2.5 MB.

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 initial 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.

*Virtually every problem in computer science can be solved by another level of indirection.*

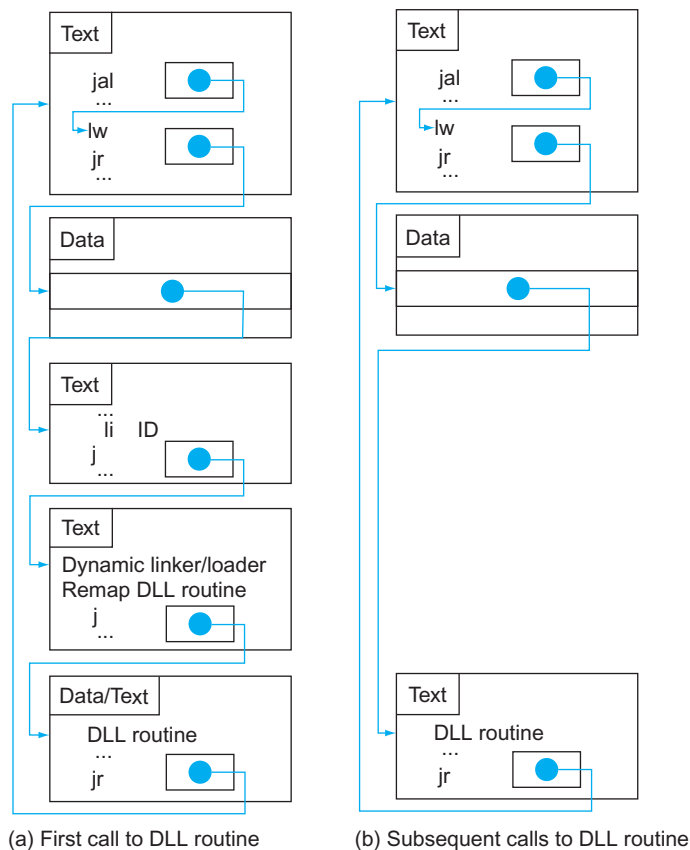
David Wheeler

**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 only 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 2.22 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 jump.

The first time the library routine is called, the program calls the dummy entry and follows the indirect jump. It points to code that puts a number in a register to



**FIGURE 2.22 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.

identify the desired library routine and then jumps to the dynamic linker/loader. The linker/loader finds the desired routine, remaps it, and changes the address in the indirect jump location to point to that routine. It then jumps to it. When the routine completes, it returns to the original calling site. Thereafter, the call to the library routine jumps indirectly to the routine without the extra hops.

In summary, DLLs require extra 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 jump 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 specific 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 2.23 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.

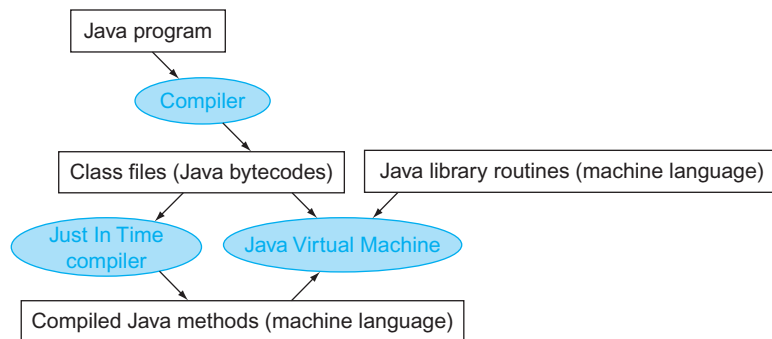
A software interpreter, called a **Java Virtual Machine (JVM)**, can execute Java bytecodes. An interpreter is a program that simulates an instruction set architecture.

### Java bytecode

Instruction from an instruction set designed to interpret Java programs.

### Java Virtual Machine (JVM)

The program that interprets Java bytecodes.



**FIGURE 2.23 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.




For example, the MIPS 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.

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 hundreds of millions 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 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.

As computers get faster so that compilers can do more, and as researchers invent better 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.

**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.

### Check Yourself

Which of the advantages of an interpreter over a translator do you think was 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 MIPS code from two procedures written in C: one to swap array elements and one to sort them.

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

**FIGURE 2.24 A C procedure that swaps two locations in memory.** This subsection uses this procedure in a sorting example.

## The Procedure `swap`

Let's start with the code for the procedure `swap` in [Figure 2.24](#). 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.

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

### Register Allocation for `swap`

As mentioned on pages 98–99, the MIPS convention on parameter passing is to use registers `$a0`, `$a1`, `$a2`, and `$a3`. Since `swap` has just two parameters, `v` and `k`, they will be found in registers `$a0` and `$a1`. The only other variable is `temp`, which we associate with register `$t0` since `swap` is a leaf procedure (see page 100). This register allocation corresponds to the variable declarations in the first part of the `swap` procedure in [Figure 2.24](#).

### 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 MIPS refers to the *byte* address, and so words are really 4 bytes apart. Hence we need to multiply the index `k` by 4 before adding it to the address. *Forgetting that sequential word addresses differ by 4 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 4 via a shift left by 2:

```
sll    $t1, $a1, 2      # reg $t1 = k * 4
add    $t1, $a0, $t1    # reg $t1 = v + (k * 4)
                        # reg $t1 has the address of v[k]
```

Now we load  $v[k]$  using  $\$t1$ , and then  $v[k+1]$  by adding 4 to  $\$t1$ :

```
lw     $t0, 0($t1)      # reg $t0 (temp) = v[k]
lw     $t2, 4($t1)      # reg $t2 = v[k + 1]
                        # refers to next element of v
```

Next we store  $\$t0$  and  $\$t2$  to the swapped addresses:

```
sw     $t2, 0($t1)      # v[k] = reg $t2
sw     $t0, 4($t1)      # v[k+1] = reg $t0 (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, which includes the procedure label and the return jump. To make it easier to follow, we identify in [Figure 2.25](#) each block of code with its purpose in the procedure.

Procedure body		
swap:	sll	\$t1, \$a1, 2
	add	\$t1, \$a0, \$t1
		# reg \$t1 = k * 4
		# reg \$t1 = v + (k * 4)
		# reg \$t1 has the address of v[k]
	lw	\$t0, 0(\$t1)
	lw	\$t2, 4(\$t1)
		# reg \$t0 (temp) = v[k]
		# reg \$t2 = v[k + 1]
		# refers to next element of v
	sw	\$t2, 0(\$t1)
	sw	\$t0, 4(\$t1)
		# v[k] = reg \$t2
		# v[k+1] = reg \$t0 (temp)
Procedure return		
	jr	\$ra
		# return to calling routine

**FIGURE 2.25** MIPS assembly code of the procedure `swap` in [Figure 2.24](#).

## 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 2.26 shows the C version of the program. Once again, we present this procedure in several steps, concluding with the full procedure.

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

**FIGURE 2.26** 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 `$a0` and `$a1`, and we assign register `$s0` to `i` and register `$s1` 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:

```
move    $s0, $zero    # i = 0
```

(Remember that `move` is a pseudoinstruction provided by the assembler for the convenience of the assembly language programmer; see page 124.) It also takes just one instruction to increment `i`, the last part of the *for* statement:

```
addi    $s0, $s0, 1    # i += 1
```

The loop should be exited if  $i < n$  is *not* true or, said another way, should be exited if  $i \geq n$ . The set on less than instruction sets register `$t0` to 1 if `$s0 < $a1` and to 0 otherwise. Since we want to test if `$s0 ≥ $a1`, we branch if register `$t0` is 0. This test takes two instructions:

```
for1tst:slt  $t0, $s0, $a1    # reg $t0 = 0 if $s0 ≥ $a1 (i≥n)
        beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $a1 (i≥n)
```

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

```
        j    for1tst        # jump to test of outer loop
exit1:
```

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

```
        move $s0, $zero      # i = 0
for1tst:slt $t0, $s0, $a1    # reg $t0 = 0 if $s0 ≥ $a1 (i≥n)
        beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $a1 (i≥n)
        . . .
        (body of first for loop)
        . . .
        addi $s0, $s0, 1     # i += 1
        j    for1tst        # jump 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    $s1, $s0, -1 # j = i - 1
```

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

```
        addi    $s1, $s1, -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: slti $t0, $s1, 0    # reg $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
```

This branch will skip over the second condition test. If it doesn't skip,  $j \geq 0$ .

The second test exits if  $v[j] > v[j + 1]$  is *not* true, or exits if  $v[j] \leq v[j + 1]$ . First we create the address by multiplying  $j$  by 4 (since we need a byte address) and add it to the base address of  $v$ :

```
sll    $t1, $s1, 2    # reg $t1 = j * 4
add    $t2, $a0, $t1  # reg $t2 = v + (j * 4)
```

Now we load  $v[j]$ :

```
lw     $t3, 0($t2)    # reg $t3 = v[j]
```

Since we know that the second element is just the following word, we add 4 to the address in register  $\$t2$  to get  $v[j + 1]$ :

```
lw     $t4, 4($t2)    # reg $t4 = v[j + 1]
```

The test of  $v[j] \leq v[j + 1]$  is the same as  $v[j + 1] \geq v[j]$ , so the two instructions of the exit test are

```
slt    $t0, $t4, $t3   # reg $t0 = 0 if $t4 ≥ $t3
beq    $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
```

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

```
j      for2tst    # jump to test of inner loop
```

Combining the pieces, the skeleton of the second *for* loop looks like this:

```
        addi $s1, $s0, -1    # j = i - 1
for2tst:slti $t0, $s1, 0     # reg $t0 = 1 if $s1 < 0 (j < 0)
        bne $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
        sll $t1, $s1, 2      # reg $t1 = j * 4
        add $t2, $a0, $t1    # reg $t2 = v + (j * 4)
        lw  $t3, 0($t2)      # reg $t3 = v[j]
        lw  $t4, 4($t2)      # reg $t4 = v[j + 1]
        slt $t0, $t4, $t3    # reg $t0 = 0 if $t4 ≥ $t3
        beq $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
        . . .
        (body of second for loop)
        . . .
        addi $s1, $s1, -1    # j -= 1
        j    for2tst        # jump 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    swap
```

### Passing Parameters in `sort`

The problem comes when we want to pass parameters because the `sort` procedure needs the values in registers `$a0` and `$a1`, 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 `$a0` and `$a1` available for the call of `swap`. (This copy is faster than saving and restoring on the stack.) We first copy `$a0` and `$a1` into `$s2` and `$s3` during the procedure:

```
move    $s2, $a0    # copy parameter $a0 into $s2
move    $s3, $a1    # copy parameter $a1 into $s3
```

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

```
move    $a0, $s2    # first swap parameter is v
move    $a1, $s1    # 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 `$ra`, since `sort` is a procedure and is called itself. The `sort` procedure also uses the saved registers `$s0`, `$s1`, `$s2`, and `$s3`, so they must be saved. The prologue of the `sort` procedure is then

```
addi    $sp, $sp, -20 # make room on stack for 5 registers
sw      $ra, 16($sp)  # save $ra on stack
sw      $s3, 12($sp)  # save $s3 on stack
sw      $s2, 8($sp)   # save $s2 on stack
sw      $s1, 4($sp)   # save $s1 on stack
sw      $s0, 0($sp)   # save $s0 on stack
```

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

### The Full Procedure `sort`

Now we put all the pieces together in [Figure 2.27](#), being careful to replace references to registers `$a0` and `$a1` in the *for* loops with references to registers `$s2` and `$s3`. 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 35 lines in the MIPS assembly language.

**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.

Saving registers		
	sort:	<pre> addi    \$sp,\$sp,-20      # make room on stack for 5 registers sw      \$ra,16(\$sp)# save \$ra on stack sw      \$s3,12(\$sp)      # save \$s3 on stack sw      \$s2,8(\$sp)# save \$s2 on stack sw      \$s1,4(\$sp)# save \$s1 on stack sw      \$s0,0(\$sp)# save \$s0 on stack </pre>
Procedure body		
Move parameters		<pre> move    \$s2,\$a0 # copy parameter \$a0 into \$s2 (save \$a0) move    \$s3,\$a1 # copy parameter \$a1 into \$s3 (save \$a1) </pre>
Outer loop	for1tst:	<pre> slt     \$t0,\$s0,\$s3 #reg\$t0=0if\$s0&lt;\$s3(i\$Ńn) beq     \$t0,\$zero,exit1# go to exit1 if \$s0 &lt;= \$s3 (i\$Ńn) </pre>
Inner loop	for2tst:	<pre> addi    \$s1,\$s0,-1# j = i - 1 slti    \$t0,\$s1,0  #reg\$t0=1if\$s1&lt;0(j&lt;0) bne     \$t0,\$zero,exit2# go to exit2 if \$s1 &lt; 0 (j &lt; 0) sll     \$t1,\$s1,2# reg \$t1 = j * 4 add     \$t2,\$s2,\$t1# reg \$t2 = v + (j * 4) lw      \$t3,0(\$t2)# reg \$t3 = v[j] lw      \$t4,4(\$t2)# reg \$t4 = v[j + 1] slt     \$t0,\$t4,\$t3 # reg \$t0 = 0 if \$t4 &lt;= \$t3 beq     \$t0,\$zero,exit2# go to exit2 if \$t4 &lt;= \$t3 </pre>
Pass parameters and call		<pre> move    \$a0,\$s2      # 1st parameter of swap is v (old \$a0) move    \$a1,\$s1      # 2nd parameter of swap is j jal     swap          # swap code shown in Figure 2.25 </pre>
Inner loop	j	<pre> addi    \$s1,\$s1,-1# j -= 1 j       for2tst      # jump to test of inner loop </pre>
Outer loop	exit2:	<pre> addi    \$s0,\$s0,1    # i += 1 j       for1tst      # jump to test of outer loop </pre>
Restoring registers		
	exit1:	<pre> lw      \$s0,0(\$sp)    # restore \$s0 from stack lw      \$s1,4(\$sp)# restore \$s1 from stack lw      \$s2,8(\$sp)# restore \$s2 from stack lw      \$s3,12(\$sp)   # restore \$s3 from stack lw      \$ra,16(\$sp)   # restore \$ra from stack addi    \$sp,\$sp,20    # restore stack pointer </pre>
Procedure return		
	jr	\$ra # return to calling routine

FIGURE 2.27 MIPS assembly version of procedure sort in Figure 2.26.



Understanding  
Program  
Performance

Figure 2.28 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.

Figure 2.29 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 MIPS 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 increases 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 ( $0.05 \times 2468$ , or 123 times faster than the unoptimized C code versus 2.41 times faster).

**Elaboration:** The MIPS compilers always save room on the stack for the arguments in case they need to be stored, so in reality they always decrement `$sp` by 16 to make room for all four argument registers (16 bytes). One reason is that C provides a `vararg` option that allows a pointer to pick, say, the third argument to a procedure. When the compiler encounters the rare `vararg`, it copies the four argument registers onto the stack into the four reserved locations.

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 2.28 Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort.** The programs sorted 100,000 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.

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	O1	2.37	1.50	1562
	Compiler	O2	2.38	1.50	1555
	Compiler	O3	2.41	1.91	1955
Java	Interpreter	–	0.12	0.05	1050
	JIT compiler	–	2.13	0.29	338

**FIGURE 2.29 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 2.28. 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 MIPS assembly versions of two procedures to clear a sequence of words in memory: one using array indices and one using pointers. Figure 2.30 shows the two C procedures.

The purpose of this section is to show how pointers map into MIPS 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 `$a0` and `$a1`, and that `i` is allocated to register `$t0`.

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

```
move    $t0,$zero    # i = 0 (register $t0 = 0)
```

To set `array[i]` to 0 we must first get its address. Start by multiplying `i` by 4 to get the byte address:

```
loop1: sll    $t1,$t0,2    # $t1 = i * 4
```

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     $t2,$a0,$t1    # $t2 = address of array[i]
```

```

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

```

**FIGURE 2.30 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 last 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 MIPS instructions to increment `p` by four, the number of bytes in a MIPS integer. The assignment in the loop places 0 in the object pointed to by `p`.

Finally, we can store 0 in that address:

```
sw    $zero, 0($t2)    # array[i] = 0
```

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

```
addi $t0,$t0,1        # i = i + 1
```

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

```
slt   $t3,$t0,$a1      # $t3 = (i < size)
bne   $t3,$zero,loop1  # if (i < size) go to loop1
```

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

```

move   $t0,$zero        # i = 0
loop1: sll   $t1,$t0,2     # $t1 = i * 4
add     $t2,$a0,$t1      # $t2 = address of array[i]
sw      $zero, 0($t2)     # array[i] = 0
addi    $t0,$t0,1        # i = i + 1
slt     $t3,$t0,$a1      # $t3 = (i < size)
bne     $t3,$zero,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 `$a0` and `$a1` and allocates `p` to register `$t0`. The code for the second procedure starts with assigning the pointer `p` to the address of the first element of the array:

```
move    $t0,$a0           # p = address of array[0]
```

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

```
loop2:  sw    $zero,0($t0)  # 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 word:

```
addi    $t0,$t0,4          # p = p + 4
```

Incrementing a pointer by 1 means moving the pointer to the next sequential object in C. Since `p` is a pointer to integers, each of which uses 4 bytes, the compiler increments `p` by 4.

The loop test is next. The first step is calculating the address of the last element of array. Start with multiplying `size` by 4 to get its byte address:

```
sll     $t1,$a1,2          # $t1 = size * 4
```

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

```
add     $t2,$a0,$t1        # $t2 = address of array[size]
```

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

```
slt     $t3,$t0,$t2        # $t3 = (p<&array[size])
bne     $t3,$zero,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:

```

        move    $t0,$a0           # p = address of array[0]
loop2:  sw      $zero,0($t0)       # Memory[p] = 0
        addi    $t0,$t0,4          # p = p + 4
        sll     $t1,$a1,2          # $t1 = size * 4
        add     $t2,$a0,$t1        # $t2 = address of array[size]
        slt     $t3,$t0,$t2        # $t3 = (p<&array[size])
        bne     $t3,$zero,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:

```


        move $t0,$a0          # p = address of array[0]
        sll  $t1,$a1,2        # $t1 = size * 4
        add  $t2,$a0,$t1      # $t2 = address of array[size]
loop2:  sw   $zero,0($t0)      # Memory[p] = 0
        addi $t0,$t0,4        # p = p + 4
        slt  $t3,$t0,$t2      # $t3 = (p<&array[size])
        bne  $t3,$zero,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):

move \$t0,\$zero	# i = 0	move \$t0,\$a0	# p = & array[0]
loop1: sll \$t1,\$t0,2	# \$t1 = i * 4	sll \$t1,\$a1,2	# \$t1 = size * 4
add \$t2,\$a0,\$t1	# \$t2 = &array[i]	add \$t2,\$a0,\$t1	# \$t2 = &array[size]
sw \$zero, 0(\$t2)	# array[i] = 0	loop2: sw \$zero,0(\$t0)	# Memory[p] = 0
addi \$t0,\$t0,1	# i = i + 1	addi \$t0,\$t0,4	# p = p + 4
slt \$t3,\$t0,\$a1	# \$t3 = (i < size)	slt \$t3,\$t0,\$t2	# \$t3=(p<&array[size])
bne \$t3,\$zero,loop1	# if () go to loop1	bne \$t3,\$zero,loop2	# if () 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 6 to 4. 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 add a jump just before the first instruction of the loop to the `slt` instruction.