SECOND EDITION

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

TRADEMARKS

The following trademarks are the property of the following organizations:

TeX is a trademark of Americal Mathematical Society.

Apple II and Macintosh are trademarks of Apple Computers, Inc.

CDC 6600, CDC 7600, CDC STAR-100, CYBER-180, CYBER-180/990, and CYBER-205 are trademarks of Control Data Corporation.

The Cosmic Cube is a trademark of California Institute of Technology.

CP3100 is a trademark of Conner Peripherals.

Cray, CRAY-1, CRAY J90, CRAY T90, CRAY X-MP/416, and CRAY Y-MP are trademarks of Cray Research.

Alpha, AlphaServer, AlphaStation, DEC, DECsystem, DECsystem 3100, DECstation, PDP-8, PDP-11, Unibus, VAX, VAX 8700, and VAX11/780 are trademarks of Digital Equipment Corporation.

MP2361A, Super Eagle, VP100, VP200, and VPP300 are trademarks of Fujitsu Corporation.

Gnu C Compiler is a trademark of Free Software Foundation.

Goodyear MPP is a trademark of Goodyear Tire and Rubber Co., Inc.

Apollo DN 300, Apollo DN 10000, Convex, HP, HP Precision Architecture, HPPA, HP850, HP 3000, HP 300/70, PA-RISC, and Precision are registered trademarks of Hewlet-Packard Company.

432, 960 CA, 4004, 8008, 8080, 8086, 8087, 8088, 80186, 80286, 80386, 80486, Delta, iAPX 432, i860, Intel, Intel486, Intel Hypercube, iPSC/2, MMX, Multibus, Multibus II, Paragon, and Pentium are trademarks of Intel Corporation. Intel Inside is a registered trademark of Intel Corporation.

360, 360/30, 360/40, 360/50, 360/65, 360/85, 360/91, 370, 370/158, 370/165, 370/168, 370-XA, ESA/370, 701, 704, 709, 801, 3033, 3080, 3080 series, 3080 VF, 3081, 3090, 3090/100, 3090/200, 3090/400, 3090/600, 3090/600S, 3090 VF, 3330, 3380, 3380D, 3380 Disk Model AK4, 3380J, 3390, 3880-23, 3990, 7090, 7094, IBM, IBM PC, IBM PC-AT, IBM SVS, ISAM, MVS, PL.8, PowerPC, POWERstation, RT-PC, RAMAC, RS/6000, Sage, Stretch, System/360, Vector Faility, and VM are trademarks of International Business Machines Corporation. POWERserver, RISC System/6000, and SP2 are registered trademarks of International Business Machines Corporation.

ICL DAP is a trademark of International Computers Limited.

Inmos and Transputer are trademarks of Inmos.

FutureBus is a trademark of the Institute of Electrical and Electronic Engineers.

KSR-1 is a trademark of Kendall Square Research.

MASPAR MP-1 and MASPAR MP-2 are trademarks of MasPar Corporation.

MIPS, R2000, R3000, and R10000 are registered trademarks of MIPS Technology, Inc.

Windows is a trademark of Microsoft Corporation.

NuBus is a trademark of Massachusetts Institute of Technology.

Delta Series 8608, System V/88 R32V1, VME bus, 6809, 68000, 68010, 68020, 68030, 68881, 68882, 88000, 88000 1.8.4m14, 88100, and 88200 are trademarks of Motorola Corporation.

Ncube and nCube/ten are trademarks of Ncube Corporation.

NEC is a registered trademark of NEC Corporation.

Network Computer is a trademark of Oracle Corporation.

Parsytec GC is a trademark of Parsytec, Inc.

Imprimis, IPI-2, Sabre, Sabre 97209, Seagate, and Wren IV are trademarks of Seagate Technology, Inc.

NUMA-Q, Sequent, and Symmetry are trademarks of Sequent Computers.

Power Challenge, Silicon Graphics, Silicon Graphics 43/240, Silicon Graphics 4D/60, Silicon Graphics 4D/240, and Silicon Graphics 4D Series are trademarks of Silicon Graphics. Origin2000 is a registered trademark of Silicon Graphics.

SPEC is a registered trademark of the Standard Performance Evaluation Corporation.

Spice is a trademark of University of California at Berkeley.

Enterprise, Java, Sun, Sun Ultra, Sun Microsystems, and Ultra are trademarks of Sun Microsystems, Inc. SPARC and UltraSPARC are registered trademarks of SPARC International, Inc., licensed to Sun Microsystems, Inc.

Connection Machine, CM-2, and CM-5 are trademarks of Thinking Machines.

Burroughts 6500, B5000, B5500, D-machine, UNIVAC, UNIVAC I, and UNIVAC 1103 are trademarks of UNISYS.

Alto, PARC, Palo Alto Research Center, and Xerox are trademarks of Xerox Corporation.

The UNIX trademark is licensed exclusively through X/Open Company Ltd.

All other product names are trademarks or registered trademarks of their respective companies. Where trademarks appear in this book and Morgan Kaufmann Publishers was aware of a trademark claim, the trademarks have been printed in initial caps or all caps.

SECOND EDITION

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

John L. Hennessy
Stanford University

David A. Patterson
University of California, Berkeley

With a contribution by
James R. Larus
University of Wisconsin

Morgan Kaufmann Publishers, Inc.
San Francisco, California

Sponsoring Editor Denise Penrose
Production Manager Yonie Overton
Production Editor Julie Pabst
Editorial Coordinator Jane Elliott
Text and Cover Design Ross Carron Design
Illustration Alexander Teshin Associates, with second edition modifications by Dartmouth Publishing, Inc.
Chapter Opener Illustrations Canary Studios
Copyeditor Ken DellaPenta
Composition Nancy Logan
Proofreader Jennifer McClain
Indexer Steve Rath
Printer Courier Corporation

Morgan Kaufmann Publishers, Inc.
Editorial and Sales Office:
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205
USA

Telephone 415/392-2665
Facsimile 415/982-2665
Email *mkp@mkp.com*
WWW *http://www.mkp.com*
Order toll free 800/745-7323

02 01 10 9 8 7

Advice, Praise, and Errors: Any correspondence related to this publication or intended for the authors should be sent electronically to *cod2bugs@mkp.com*. Information regarding error sightings is encouraged. Any error sightings that are accepted for correction in subsequent printings will be rewarded by the authors with a payment of $1.00 (U.S.) per correction at the time of their implementation in a reprint.

Library of Congress Cataloging-in-Publication Data
Patterson, David A.
Computer organization and design : the hardware/software interface
/ David A. Patterson, John L. Hennessy.—2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 1-55860-428-6 (cloth).—ISBN 1-55860-491-X (paper)
1. Computer organization. 2. Computers—Design and construction.
3. Computer interfaces. I. Hennessy, John L. II. Title
QA76.9.C643H46 1997
004.2'2—dc21 97-16050

TO LINDA AND ANDREA

3

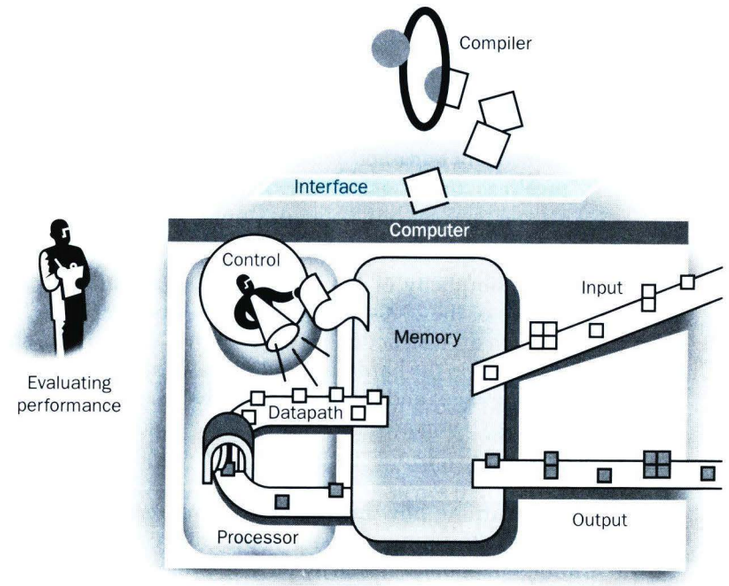
Instructions: Language of the Machine

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

Charles V, King of France
1337–1380

3.1 **Introduction** 106
3.2 **Operations of the Computer Hardware** 107
3.3 **Operands of the Computer Hardware** 109
3.4 **Representing Instructions in the Computer** 116
3.5 **Instructions for Making Decisions** 122
3.6 **Supporting Procedures in Computer Hardware** 132
3.7 **Beyond Numbers** 142
3.8 **Other Styles of MIPS Addressing** 145
3.9 **Starting a Program** 156
3.10 **An Example to Put It All Together** 163
3.11 **Arrays versus Pointers** 171
3.12 **Real Stuff: PowerPC and 80x86 Instructions** 175
3.13 **Fallacies and Pitfalls** 185
3.14 **Concluding Remarks** 187
3.15 **Historical Perspective and Further Reading** 189
3.16 **Key Terms** 196
3.17 **Exercises** 196

The Five Classic Components of a Computer



3.1 Introduction

To command a computer's hardware, you must speak its language. The words of a machine'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 humans and in the form read by the machine. 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.

You might think that the languages of machines would be as diverse as those of humans, but in reality machine languages are quite similar, more like regional dialects than like independent languages. Hence once you learn one, it is easy to pick up others. This similarity occurs because all computers are constructed from hardware technologies based on similar underlying principles and because there are a few basic operations that all machines 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. 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 the machines of the 2000s 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 the hardware and the relationship between high-level programming languages and this more primitive one. We are using the C programming language. (If you are familiar with Pascal, you may wish to refer to Web Extension III, available at *www.mkp.com/cod2e.htm*, for a short comparison of C with Pascal.)

By learning how instructions are represented, you will also discover the secret of computing: the stored-program concept. And you will exercise your "foreign language" skills by writing programs in the language of the machine and running them on the simulator that comes with this book. We conclude with a look at the historical evolution of instruction sets and an overview of other machine dialects.

The chosen instruction set comes from MIPS, used by NEC, Nintendo, Silicon Graphics, and Sony, among others, and is typical of instruction sets designed since the early 1980s. We reveal the MIPS instruction set a piece at a time, giving the rationale along with the machine structures. This step-by-step tutorial weaves the components with their explanations, making assembly language more palatable. To keep the overall picture in mind, each section ends with a figure summarizing the MIPS instruction set revealed thus far, highlighting the portions presented in that section.

3.2 Operations of the Computer Hardware

> *There must certainly be instructions for performing the fundamental arithmetic operations.*
>
> Burks, Goldstine, and von Neumann, 1947

Every computer must be able to perform arithmetic. The 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 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 give a more detailed and realistic picture.)

The following sequence of instructions adds the 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 take the sum of four variables.

The words to the right of the sharp symbol (#) on each line above are *comments* for the human reader, and they are ignored by the computer. Note that unlike other programming languages, each line of this language can contain at most one instruction. Another difference 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 four 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. Figure 3.1 summarizes the portions of MIPS assembly language described in this section.

Example

Compiling Two C Assignment Statements into MIPS

This segment of a C program contains the five variables `a`, `b`, `c`, `d`, and `e`:

```
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 C compiler.

Answer

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

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

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add a,b,c`	`a = b + c`	Always three operands
	subtract	`sub a,b,c`	`a = b - c`	Always three operands

FIGURE 3.1 MIPS architecture revealed in section 3.2. The real machine operands will be unveiled in the next section. Highlighted portions in such summaries show MIPS assembly language structures introduced in this section; for this first figure, all is new.

Example

Compiling a Complex C Assignment into MIPS

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

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

What would a C compiler produce?

Answer

The compiler must break this C 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 C operation is subtract, we need to calculate the sum of `i` and `j` before we can subtract. Thus the second instruction places the sum `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 result in the variable `f`, completing the compiled code:

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

These instructions are symbolic representations of what the MIPS processor actually understands. In the next few sections we will evolve this symbolic representation into the real language of MIPS, with each step making the symbolic representation more concrete.

3.3 Operands of the Computer Hardware

Unlike programs in high-level languages, the operands of arithmetic instructions cannot be any variables; they must be from a limited number of special locations called *registers*. Registers are the bricks of computer construction, for registers are primitives used in hardware design that are also visible to the programmer when the computer is completed. 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.

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers. MIPS has 32 registers. (See section 3.15 for the history of the number of

registers.) Thus, continuing in our 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 to 32 registers may be found in the second of our four underlying design principles of hardware technology:

Design Principle 2: Smaller is faster.

A very large number of registers would 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.

Chapters 5 and 6 show the central role that registers play in hardware construction; as we shall see in this chapter, effective use of registers is key 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 3.6 will explain the reasons behind these names. For now we will use `$s0`, `$s1`, ... for registers that correspond to variables in C 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 C assignment statement from our earlier example:

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

The variables `f`, `g`, `h`, `i`, and `j` can be assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. What is the compiled MIPS assembly code?

Answer

The compiled program is very similar to the prior example, except we replace the variables with the registers 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)
```


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

Recall the five components of a computer introduced in Chapter 1 and depicted on page 105. The processor can keep only a small amount of data in registers, but computer memory contains millions of data elements. Hence data structures, such as arrays, are kept in memory.

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 3.2, the address of the third data element is 2, and the value of Memory[2] is 10.

The data transfer instruction that moves 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 memory address is formed by the sum of the constant portion of the instruction and the contents of the second register. The actual MIPS name for this instruction is `lw`, standing for *load word*.

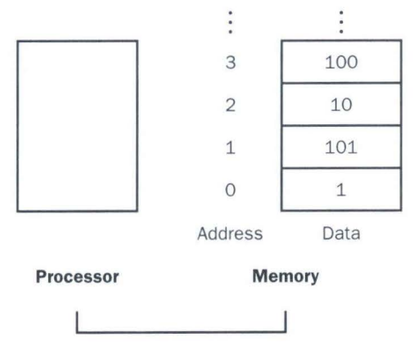


FIGURE 3.2 Memory addresses and contents of memory at those locations. This is a simplification of the MIPS addressing; Figure 3.3 shows MIPS addressing for sequential words in memory.

Compiling an Assignment When an Operand Is in Memory

Example

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`. Translate this C assignment statement:

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

Answer

Although there is a single operation in this C 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 3.2, the first compiled instruction is

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

(On the next page we'll make 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]` (`$t0`) and put the sum in the register corresponding to `g` (associated with `$s1`):

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

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

Hardware Software Interface

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

Since 8-bit *bytes* are useful in many programs, most architectures address individual bytes. Therefore the address of a word matches the address of one of the 4 bytes within the word. Hence, addresses of sequential words differ by 4. For example, Figure 3.3 shows the actual MIPS addresses for Figure 3.2; the byte address of the third word is 8.

Words must always start at addresses that are multiples of 4 in MIPS. This requirement is called an *alignment restriction*, and many architectures have it. (Chapter 5 suggests why alignment leads to faster data transfers.)

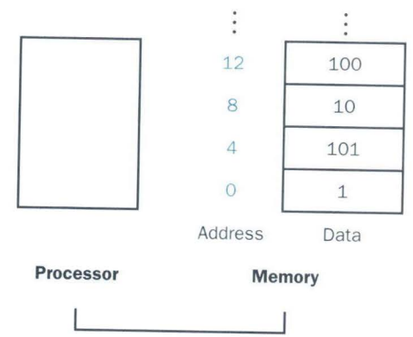


FIGURE 3.3 Actual MIPS memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 3.2. Since MIPS addresses each byte, word addresses are multiples of four (there are four bytes in a word).

Machines with byte addresses are split 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. (Appendix A, page A-48, 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 × 8, or 32,** so that the load address will select `A[8]` and not `A[8/4]`.

The instruction complementary to load is traditionally called *store*; it transfers 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*.

Compiling Using Load and Store

Example

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];
```

Answer

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 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 as the offset and register `$s3` as the base register.

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

Arrays are often accessed with variables instead of constants, so that the array element being selected can change while the program is running.

Compiling Using a Variable Array Index

Example

Here is an example of an array with a variable index:

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

Assume `A` is an array of 100 elements whose base is in register `$s3` and that the compiler associates the variables `g`, `h`, and `i` with the registers `$s1`, `$s2`, and `$s4`. What is the MIPS assembly code corresponding to this C segment?

Answer

Before we can load `A[i]` into a temporary register, we need to have its address. Before we can add `i` to the base of array `A` to form the address, we must multiply the index `i` by 4 due to the byte addressing problem. We will see a multiply instruction in the next chapter; for now we will get the effect of multiplying `i` by 4 by first adding `i` to itself (`i + i = 2i`) and then adding that sum to itself (`2i + 2i = 4i`):

```
add $t1,$s4,$s4    # Temp reg $t1 = 2 * i
add $t1,$t1,$t1    # Temp reg $t1 = 4 * i
```

To get the address of `A[i]`, we need to add `$t1` and the base of `A` in `$s3`:

```
add $t1,$t1,$s3    # $t1 = address of A[i] (4 * i + $s3)
```


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

```
lw   $t0,0($t1)  # Temporary reg $t0 = A[i]
```

The final instruction adds `A[i]` and `h`, and places the sum in `g`:

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

Hardware Software Interface

Many programs have more variables than machines 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 registers are smaller. This is indeed the case; data accesses are faster if data is kept 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 MIPS registers take both less time to access *and* have higher throughput than memory—a rare combination—making data in registers both faster to access and simpler to use. To achieve highest performance, MIPS compilers must use registers efficiently.

Figure 3.4 summarizes the portions of the symbolic representation of the MIPS instruction set described in this section. Load word and store word are the instructions that transfer words between memory and registers in the MIPS architecture. Other brands of computers use instructions in addition to load and store to transfer data. An architecture with such alternatives is the Intel 80x86, described in section 3.12.

Elaboration: The offset plus base register addressing is an excellent match to structures as well, 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 3.10.

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.

MIPS operands

Name	Example	Comments
32 registers	`$s0, $s1, . . . , $t0, $t1, . . .`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	three operands; data in registers
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`	three operands; data in registers
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Data from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Data from register to memory

FIGURE 3.4 MIPS architecture revealed through section 3.3. Highlighted portions show MIPS assembly language structures introduced in section 3.3.

3.4 Representing Instructions in the Computer

We are now ready to explain the difference between the way humans instruct machines and the way machines see instructions. But first, let's quickly review how a machine 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.

Instructions are also 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 part of almost all 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.

Translating a MIPS Assembly Instruction into a Machine Instruction

Example

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

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

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.

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. (This term is explained in Chapter 4 when we see the shift instructions; it will not be used until then, and hence the field contains zero.)
- *funct*: Function. This field selects the specific variant of the operation in the op field, and is sometimes called the *function code*.

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 2^5 or 32. This constant is used to select elements from large 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 third 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* or *I-format* and is used by the data transfer instructions. The fields of I-format are

op	rs	rt	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 (2^{13} or 8192 words) of the address in the base register rs.

Let's take a look at the load word instruction from page 114:

```
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 fourth field in I-type is equal to the length 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 3.5 shows the numbers used in each field for the MIPS instructions covered through section 3.3.

Instruction	Format	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32	n.a.
`sub` (subtract)	R	0	reg	reg	reg	0	34	n.a.
`lw` (load word)	I	35	reg	reg	n.a.	n.a.	n.a.	address
`sw` (store word)	I	43	reg	reg	n.a.	n.a.	n.a.	address

FIGURE 3.5 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 machine executes. Assuming that $t1 has the base of the array `A` and that `$s2` corresponds to `h`, the C 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 3.5 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 3.5) 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 × 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.

The binary equivalent to the decimal form is the following (1200 in base 10 is 0000 0100 1011 0000 base 2):

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 found in the third bit from the left.

Figure 3.6 summarizes the portions of MIPS assembly language described in this section. As we shall see in Chapters 5 and 6, the similarity of the binary representations of related instructions simplifies hardware design. These instructions are another example of regularity in the MIPS architecture.

Elaboration: Representing decimal numbers in base 2 gives an easy way to represent positive integers in computer words. Chapter 4 explains how negative numbers can be represented, but for now take it on faith that a 32-bit word can represent integers between -2^{31} and $+2^{31} - 1$ or –2,147,483,648 to +2,147,483,647. Such integers are called *two's complement* numbers.

MIPS operands

Name	Example	Comments
32 registers	`$s0, $s1, ..., $s7` `$t0, $t1, ..., $t7`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers `$s0–$s7` map to 16–23 and `$t0–$t7` map to 8–15.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
Data transfer	load word	`lw $s1,100($s2)`	`$s1` = Memory[`$s2` + 100]	Data from memory to register
	store word	`sw $s1,100($s2)`	Memory[`$s2` + 100] = `$s1`	Data from register to memory

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`
`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 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 3.6 MIPS architecture revealed through section 3.4. Highlighted portions show MIPS machine language structures introduced in section 3.4. 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; *shamt* field, which is unused in Chapter 3 and hence always is 0; and the *funct* field, which specifies the specific operation of R-format instructions. I-format keeps the last 16 bits as a single *address* field.

The Big Picture

Today's computers are built on two key principles:

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

These principles lead to the *stored-program* concept; its invention let the computing genie out of its bottle. Figure 3.7 shows the power of the concept; specifically, memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code.

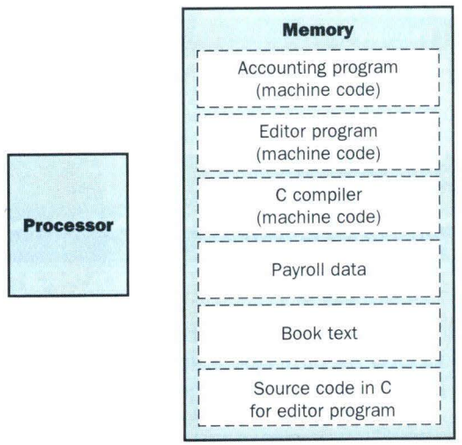


FIGURE 3.7 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given 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 machine can understand.

3.5 Instructions for Making Decisions

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. When the iteration is completed a different sequence of [instructions] is to be followed, so we must, in most cases, give two parallel trains of [instructions] preceded by an instruction as to which routine is to be followed. 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

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during the computation, different instructions are executed. 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*.

Example

Compiling an *If* Statement into a Conditional Branch

In the following C code segment, `f`, `g`, `h`, `i`, and `j` are variables:

```
        if (i == j) go to L1;
        f = g + h;
L1:     f = f - i;
```

Assuming that the five variables `f` through `j` correspond to the five registers `$s0` through `$s4`, what is the compiled MIPS code?

Answer

The first C statement compares for equality and then branches to the subtract operation. Since both the operands are in registers, this maps exactly to a branch if equal instruction (we'll define the label `L1` later):

```
beq $s3,$s4, L1  # go to L1 if i equals j
```

The following C 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 equals j)
```

The final statement can again be compiled into a single instruction. The problem is how to specify its address so that the conditional branch can skip the add instruction above.

Instructions are stored in memory in stored-program computers; hence instructions must have memory addresses just like other words in memory. The last instruction simply appends the label `L1` that was forward-referenced by the `beq` instruction.

```
L1:     sub $s0,$s0,$s3  # f = f - i (always executed)
```

The label `L1` thus corresponds to the address of the subtract instruction.

Notice that the assembler relieves the compiler or 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 3.9).

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.

Compiling *if-then-else* into Conditional Branches

Example

Using the same variables and registers from the previous example, compile this C *if* statement:

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

Answer

Figure 3.8 is a flowchart of what the MIPS code should do. The first C expression compares for equality, so it would seem that we would want the `beq` as before. 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):

```
bne $s3,$s4,Else   # go to Else if i ≠ j
```

The next C 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 machine 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:
```

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.

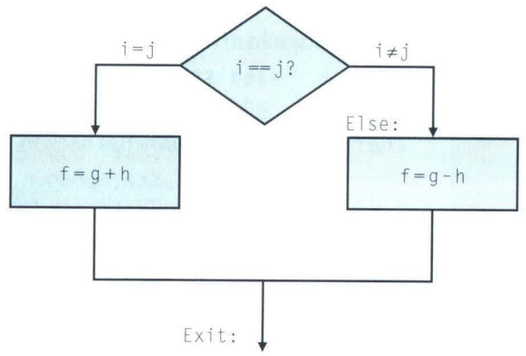


FIGURE 3.8 Illustration of the options in the *if* statement above. The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

Compiling a Loop with Variable Array Index

Example

Here is a loop in C:

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto Loop;
```

Assume that `A` is an array of 100 elements and that the compiler associates the variables `g`, `h`, `i`, and `j` with the registers `$s1`, `$s2`, `$s3`, and `$s4`, respectively. Let's assume that the base of the array `A` is in `$s5`. What is the MIPS assembly code corresponding to this C loop?

Answer

The first step is to load `A[i]` into a temporary register. We borrow the code from the similar example that starts on page 114. We need only add the label `Loop` to the first instruction so that we can branch back to that instruction at the end of the loop:

```
Loop:   add $t1,$s3,$s3     # Temp reg $t1 = 2 * i
        add $t1,$t1,$t1     # Temp reg $t1 = 4 * i
        add $t1,$t1,$s5     # $t1 = address of A[i]
        lw  $t0,0($t1)      # Temporary reg $t0 = A[i]
```

The next two instructions add `A[i]` to `g` and then `j` to `i`:

```
        add $s1,$s1,$t0     # g = g + A[i]
        add $s3,$s3,$s4     # i = i + j
```

The final instruction branches back to `Loop` if `i` ≠ `h`:

```
        bne $s3,$s2, Loop   # go to Loop if i ≠ h
```

Since the body of the loop modifies `i`, we must multiply its value by 4 each time through the loop. (Section 3.11 shows how to avoid these "multiplies" when writing loops like this one.)

Hardware Software Interface

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

Compiling a *while* Loop

Example

Of course, programmers don't normally write loops with *go to* statements, so it is up to the compiler to translate traditional loops into MIPS language. Here is a traditional loop in C:

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

Assume that `i`, `j`, and `k` correspond to registers `$s3`, `$s4`, 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. It starts with code similar to the prior example:

```
Loop:   add $t1,$s3,$s3     # Temp reg $t1 = 2 * i
        add $t1,$t1,$t1     # Temp reg $t1 = 4 * i
        add $t1,$t1,$s6     # $t1 = address of save[i]
        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 `j` to `i`:

```
        add  $s3,$s3,$s4    # i = i + j
```

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 Exercise 3.9 for an optimization of this sequence.)

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
```

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.

Hardware Software Interface

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

Compiling a Less Than Test

Example

What is the code to test if variable `a` (corresponding to register `$s0`) is less than variable `b` (register `$s1`) and then branch to label `Less` if the condition holds?

Answer

The first step is to use the set on less than instruction and a temporary register:

```
slt $t0,$s0,$s1          # $t0 gets 1 if $s0 < $s1 (a < b)
```

Register `$t0` is set to 1 if `a` is less than `b`. Hence, a branch to see if register `$t0` is not equal to 0 will give us the effect of branching if `a` is less than `b`. Register `$zero` always contains 0, so this final test is accomplished using the `bne` instruction and comparing register `$t0` to register `$zero`:

```
bne $t0,$zero, Less      # go to Less if $t0 ≠ 0
                         #   (that is, if a < b)
```

This pair of instructions, `slt` and `bne`, implements branch on less than.

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 this instruction would take extra clock cycles per instruction. Two faster instructions are more useful.

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. One way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements. But sometimes the alternatives may be efficiently encoded as a table of addresses of alternative instruction sequences, called a *jump address 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.

To support such situations, computers like MIPS include a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register. The program loads the appropriate entry from the jump table into a register, and then it jumps to the proper address using a jump register.

Compiling a *switch* Statement by Using a Jump Address Table

Example

This C version of a *case* statement is called a *switch* statement. The following C code chooses among four alternatives depending on whether `k` has the value 0, 1, 2, or 3.

```
switch (k) {
      case 0:  f = i + j; break; /* k = 0 */
      case 1:  f = g + h; break; /* k = 1 */
      case 2:  f = g - h; break; /* k = 2 */
      case 3:  f = i - j; break; /* k = 3 */
}
```

Assume the six variables `f` through `k` correspond to six registers `$s0` through `$s5` and that register `$t2` contains 4. What is the corresponding MIPS code?

Answer

We use the *switch* variable `k` to index a jump address table, and then jump via the value loaded. We first test `k` to be sure it matches one of the cases ($0 \le k \le 3$); if not, the code exits the *switch* statement.

```
slt  $t3,$s5,$zero      # Test if k < 0
bne  $t3,$zero,Exit     # if k < 0, go to Exit
slt  $t3,$s5,$t2        # Test if k < 4
beq  $t3,$zero,Exit     # if k >= 4, go to Exit
```

Since we are using the variable k to index into this table of words, we must first multiply by 4 to turn k into its byte address:

```
add  $t1,$s5,$s5        # Temp reg $t1 = 2 * k
add  $t1,$t1,$t1        # Temp reg $t1 = 4 * k
```

Assume that four sequential words in memory, starting at an address contained in $t4, have addresses corresponding to the labels L0, L1, L2, and L3. We can now load the proper jump address this way:

```
add  $t1,$t1,$t4        # $t1 = address of JumpTable[k]
lw   $t0,0($t1)         # Temp reg $t0 = JumpTable[k]
```

A jump register instruction jumps via the register to the address from the jump table.

```
jr   $t0                # jump based on register $t0
```

The first three *switch* cases in this example are the same: a label, a single instruction performing the *case* statement, and then a jump to exit the *switch* statement:

```
L0: add  $s0,$s3,$s4    # k = 0 so f gets i + j
    j    Exit           # end of this case so go to Exit
L1: add  $s0,$s1,$s2    # k = 1 so f gets g + h
    j    Exit           # end of this case so go to Exit
L2: sub  $s0,$s1,$s2    # k = 2 so f gets g - h
    j    Exit           # end of this case so go to Exit
```

A more complex example might have several instructions for each case.

For the final case we drop the jump to exit (since this is the last instruction of the switch code) and append an Exit label afterwards to mark the end of the switch statement:

```
L3: sub  $s0,$s3,$s4    # k = 3 so f gets i - j
Exit:                   # end of switch statement
```

Figure 3.9 summarizes the portions of MIPS assembly language described in this section. This step along the evolution of the MIPS language has added branches and jumps to our symbolic representation, and fixes the useful value 0 permanently in a register.

Elaboration: If you have heard about *delayed branches*, covered in Chapter 6, don't worry: The MIPS assembler makes them invisible to the assembly language programmer. Also, for C programmers not familiar with the infinitely abusable `go to` statement, it transfers control from wherever it appears to the label.

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, . . . , $s7 $t0, $t1, . . . , $t7, $zero	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0–$s7 map to 16–23 and $t0–$t7 map to 8–15. MIPS register $zero always equals 0.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory
Conditional branch	branch on equal	beq $s1,$s2,L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; used with beq, bne
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $t1	go to $t1	For *switch* statements

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
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see section 3.8)
jr	R	0	9	0	0	0	8	jr $t1
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

FIGURE 3.9 MIPS architecture revealed through section 3.5. Highlighted portions show MIPS structures introduced in section 3.5. The J-format, used for jump instructions, is explained in section 3.8. Section 3.8 also explains the proper values in address fields of branch instructions.

3.6 Supporting Procedures in Computer Hardware

A procedure or subroutine 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, with parameters acting as a barrier between the procedure and the rest of the program and data, allowing it to be passed values and return results.

You can think of a procedure like a spy who leaves with a secret plan, acquires resources, performs the task, covers his 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. Place 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. Place the result value in a place where the calling program can access it.
6. Return control to the point of origin.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. Hence MIPS software allocates the following of its 32 registers for procedure calling:

- `$a0–$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`, is called the *return address*. The

return address is needed because the same procedure could be called from several parts of the program.

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 saves PC + 4 in register `$ra` to link to the following instruction to set up the procedure return.

We already have an instruction to do the return jump:

```
jr   $ra
```

The jump register instruction, which we used above in the *switch* statement, 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–$v1`, and returns control to the caller using `jr $ra`.

Using More Registers

Suppose a compiler needs more registers for a procedure than the four argument and two return value registers. Since we are supposed to cover our tracks after our mission is complete, any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked. This situation is an example in which we need to spill registers to memory, as mentioned in the Hardware Software Interface section on page 115.

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 can be found. The stack pointer is adjusted by one word for each register that is saved or restored. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: placing data onto the stack is called a *push*, and removing data from the stack is called a *pop*.

MIPS software allocates another register just for the stack: the *stack pointer* (`$sp`), used to save the registers needed by the callee. By historical precedent, stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

Compiling a Procedure that Doesn't Call Another Procedure

Example

Let's turn the example on page 109 into a procedure:

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

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

For the rest of this section we assume we can add or subtract constants like 4, 8, or 12. (Section 3.8 reveals how constants are handled in MIPS assembly language.) 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 109, which uses two temporary registers. Thus we need to save three registers: `$s0`, `$t0`, and `$t1`. We create space for three words on the stack and then store the old values:

```
sub $sp,$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 3.10 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 109:

```
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 old values of the registers we saved and then "pop" the stack to its original value:

```
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
add $sp,$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 example above 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 offers two classes of registers:

- `$t0–$t9`: 10 temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- `$s0–$s7`: 8 saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

This simple convention reduces register spilling. In the example above, since the caller (procedure doing the calling) 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.

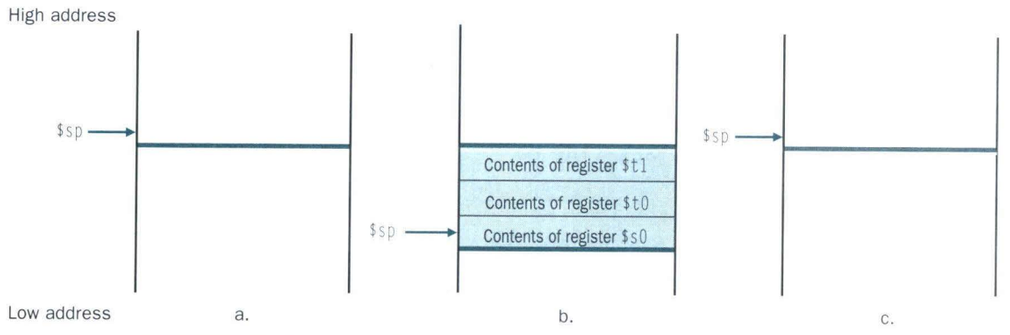


FIGURE 3.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

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 non-leaf 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 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));
}
```

Assume that you can add or subtract constants like 1 or 8, as we will show in section 3.8. What is the MIPS assembly code?

Answer

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`:

```
fact:
        sub     $sp,$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 if `n` is less than 1, going to `L1` if $n \geq 1$.

```
        slt     $t0,$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:

```
        add     $v0,$zero,1   # return 1
        add     $sp,$sp,8     # pop 2 items off stack
        jr      $ra           # return to after jal
```

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:     sub     $a0,$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
add  $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 4:

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

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

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

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 3.11 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 sections, they are also preserved.

Figure 3.11 summarizes what is preserved across a procedure call. Note that several schemes are used to preserve the stack. The stack above $sp is preserved simply by making sure the callee does not write above $sp; $sp is itself preserved by the callee adding exactly the same amount that was subtracted from it, and the other registers are preserved by saving them on the stack (if they are used) and restoring them from there. These actions also guarantee that the caller will get the same data back on a load from the stack as it put into the stack on a store: because the callee promises to preserve $sp and because the callee also promises not to modify the caller's portion of the stack, that is, the area above the $sp at the time of the call.

Allocating Space for New Data

The final complexity is that the stack is also used to store variables that are local to the procedure that 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 3.12 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 $fp by avoiding changes to $sp within a procedure: in our examples, the stack is adjusted only on entry and exit of the procedure.

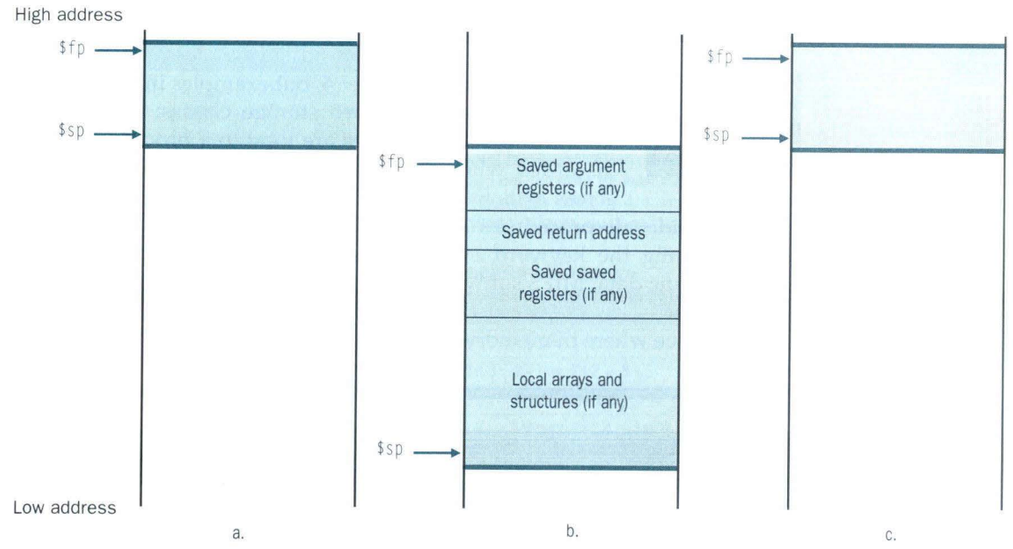


FIGURE 3.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer ($fp) points to the first word of the frame, often a saved argument register, and the stack pointer ($sp) points to the top of the stack. The stack is adjusted to make room for 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.

Figure 3.13 summarizes the register conventions for the MIPS assembly language, and Figure 3.14 summarizes the parts of the MIPS instruction set described so far.

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 3.12, 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/Silicon Graphics does not; it uses register 30 as another save register ($s8).

Elaboration: jal actually saves the address of the instruction that *follows* jal into register $ra, thereby allowing a procedure return to be simply jr $ra.

Hardware Software Interface

A C variable is a location in storage, and its interpretation depends both on its *type* and *storage class*. Types are discussed in detail in Chapter 4, but examples include integers and characters. 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`.

We will see where in memory the static data is allocated in section 3.9.

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 3.13 MIPS register convention. Register 1, called `$at`, is reserved for the assembler (see section 3.9), and registers 26–27, called `$k0–$k1`, are reserved for the operating system.

MIPS operands

Name	Example	Comments
32 registers	`$s0–$s7`, `$t0–$t9`, `$zero`, `$a0–$a3`, `$v0–$v1`, `$gp`, `$fp`, `$sp`, `$ra`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. `$gp` (28) is the global pointer, `$sp` (29) is the stack pointer, `$fp` (30) is the frame pointer, and `$ra` (31) is the return address.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`	Three operands; data in registers
Data transfer	load word	`lw $s1,100($s2)`	`$s1` = Memory[`$s2` + 100]	Data from memory to register
	store word	`sw $s1,100($s2)`	Memory[`$s2` + 100] = `$s1`	Data from register to memory
Conditional branch	branch on equal	`beq $s1,$s2,L`	if (`$s1 == $s2`) go to `L`	Equal test and branch
	branch on not equal	`bne $s1,$s2,L`	if (`$s1 != $s2`) go to `L`	Not equal test and branch
	set on less than	`slt $s1,$s2,$s3`	if (`$s2 < $s3`) `$s1`=1; else `$s1` = 0	Compare less than; for `beq`, `bne`
Unconditional jump	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

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`
`lw`	I	35	18	17	100			`lw $s1,100($s2)`
`sw`	I	43	18	17	100			`sw $s1,100($s2)`
`beq`	I	4	17	18	25			`beq $s1,$s2,100`
`bne`	I	5	17	18	25			`bne $s1,$s2,100`
`slt`	R	0	18	19	17	0	42	`slt $s1,$s2,$s3`
`j`	J	2	2500					`j 10000` (see section 3.8)
`jr`	R	0	31	0	0	0	8	`jr $ra`
`jal`	J	3	2500					`jal 10000` (see section 3.8)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

FIGURE 3.14 MIPS architecture revealed through section 3.6. Highlighted portions show MIPS assembly language structures introduced in section 3.6. The J-format, used for jump and jump-and-link instructions, is explained in section 3.8. This section also explains why putting 25 in the address field of `beq` and `bne` machine language instructions is equivalent to 100 in assembly language.

3.7 Beyond Numbers

Computers were invented to crunch numbers, but as soon as they became commercially viable they were used to process text. Most computers today use 8-bit bytes to represent characters, with the American Standard Code for Information Interchange (ASCII) being the representation that nearly everyone follows. Figure 3.15 summarizes ASCII.

A series of instructions can be used to 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 special 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
```

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
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 3.15 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, 9 represents a tab character and 13 represents a carriage return. Other useful ASCII values are 8 for backspace and 0 for Null, the value the programming language C uses to mark the end of a string.

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.

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 and test byte */
        i = i + 1;
}
```

What is the MIPS assembly code?

Answer

Below is the basic MIPS assembly code segment. We again assume we can add or subtract constants like 1 or 4, which we cover in section 3.8. 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:
    sub   $sp,$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   $t1,$a1,$s0  # 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, which puts the character into `$t2`:

```
lb    $t2, 0($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   $t3,$a0,$s0 # 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, 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.

```
add   $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
    add   $sp,$sp,4   # pop 1 word off stack
    jr    $ra         # return
```

String copies are usually done with pointers instead of arrays in C to avoid the operations on `i` in the code above. See section 3.11 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 the registers it must save.

Elaboration: There is a universal encoding of the characters of most human languages called *Unicode*, which needs 16 bits to represent a character. The programming language Java, for example, uses Unicode. The full MIPS instruction set has explicit instructions to load and store 16-bit quantities, called *halfwords*. We skip halfword instructions in this book to keep the instruction set as easy to understand as possible, although section A.10 starting on page A-49 includes the full instruction set.

Also, 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 will be allocated 4 bytes, even though it needs just 1 byte. A string variable or an array of bytes *will* pack 4 bytes per word, however.

3.8 Other Styles of MIPS Addressing

Designers of the MIPS architecture provided two more ways of accessing operands. The first is to make it faster to access small constants, and the second is to make branches more efficient.

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, counting iterations of a loop, or adjusting the stack in a nested procedure call. In fact, in two programs, more than half of the arithmetic instructions have a constant as an operand: in the C compiler gcc, 52% of arithmetic operations involve constants; in the circuit simulation program spice, it is 69%.

Using only the instructions in Figure 3.14, we would have to load a constant from memory to use it. (The constants would have been placed in memory when the program was loaded.) For example, to add the constant 4 to register `$sp`, we could use the code

```
lw    $t0, AddrConstant4($zero) # $t0 = constant 4
add   $sp,$sp,$t0               # $sp = $sp + $t0 ($t0 == 4)
```

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

An alternative that avoids memory accesses is to offer versions of the arithmetic instructions in which one operand is a constant, with the novel constraint that this constant is kept inside the *instruction* itself. Following the recommendation urging regularity, we use the same format for these instructions as for the data transfer and branch instructions. In fact, the *I* in the name of the I-type format is for *immediate*, the traditional name for this type of operand. The MIPS field containing the constant is 16 bits long.

Example

Translating Assembly Constants into Machine Language

The add instruction that has one constant operand is called *add immediate* or `addi`. To add 4 to register `$sp`, we just write

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

The op field value for `addi` is 8. Try to guess the rest of the corresponding MIPS machine instruction.

Answer This instruction is the following machine code (using decimal numbers):

op	rs	rt	immediate
8	29	29	4

(Figure 3.13 on page 140 shows that register 29 corresponds to `$sp`.) In binary `addi` is

001000	11101	11101	0000 0000 0000 0100

Immediate or constant operands are also popular in comparisons. Since register `$zero` always has 0, we can already compare to 0. To compare to other values, there is an immediate version of the set on less than instruction. To test if register `$s2` is less than the constant 10, we can just write

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

Similar to the earlier example on page 128 (Hardware Software Interface), this instruction can be followed by `bne $t0,$zero` to branch if register `$s2` is less than the constant 10.

Immediate addressing illustrates the final hardware design principle, first mentioned in Chapter 2:

Design Principle 4: Make the common case fast.

Constant operands occur frequently, and by making constants part of arithmetic instructions, they are much faster than if they were loaded from memory.

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 3.16 shows the operation of `lui`.

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 3.16 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. As we shall see in Chapter 4, this instruction is like multiplying the constant by 2^{16} before loading it into the register.

Loading a 32-Bit Constant

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

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Answer 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 add the lower 16 bits, whose decimal value is 2304:

```
addi $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
```

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 is a reason for the register `$at`, which is reserved for the assembler.

Hence the symbolic representation of the MIPS machine language is no longer limited by the hardware, but to whatever the creator of an assembler chooses to include (see section 3.9). We stick close to the hardware to explain the architecture of the machine, noting when we use the enhanced language of the assembler that is not found in the machine.

Elaboration: We need to be careful about creating 32-bit constants. The instruction `addi` will copy the leftmost bit of the 16-bit immediate field of the instruction into the upper 16 bits of a word. An instruction we will see in the next chapter, `ori`, for *logical or immediate*, 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 simplest addressing is found in the MIPS jump instructions. 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
```

is assembled into this format:

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:

Program counter = Register + 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. The question is then, 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, almost half of all conditional branches in gcc and spice 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 5, 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).

Like most recent machines, 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, and 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.

Showing Branch Offset in Machine Language

Example The *while* loop on page 127 was compiled into this MIPS assembler code:

```
Loop: add $t1,$s3,$s3       # Temp reg $t1 = 2 * i
      add $t1,$t1,$t1       # 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
      add $s3,$s3,$s4       # i = i + j
      j   Loop              # go to Loop
Exit:
```

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

Answer The assembled instructions and their addresses would look like this:

80000	0	19	19	9	0	32
80004	0	9	9	9	0	32
80008	0	9	22	9	0	32
80012	35	9	8	0		
80016	5	8	21	8		
80020	0	19	20	19	0	32
80024	2	80000				
80028	...					
80012	35	9	8	0		

Remember that MIPS instructions use byte addresses, so addresses of sequential words differ by four, the number of bytes in a word. The bne instruction on the fifth line adds 8 bytes to the address of the *following* instruction (80020), specifying the branch destination relative to that instruction (8) instead of the current instruction (16) or using the full destination address (80028). The jump instruction on the last line does use the full address (80000), corresponding to the label Loop.

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. Hence the address field in the `bne` instruction at location 80016 in the example above should have 2 instead of 8. (Relative word addressing is the reason that the machine language versions of `beq` and `bne` in Figures 3.9 and 3.14 have 25 in their address fields instead of 100, as in the assembly language versions.)

Hardware Software Interface

Nearly every conditional branch is to a nearby location, but occasionally it branches 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 the condition is inverted so that the branch decides whether to skip the jump.

Branching Far Away

Example 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 It can be replaced by these instructions:

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

Elaboration: The 26-bit field in jump instructions is also a word address, meaning that it represents a 28-bit byte address. Since the PC is 32 bits, 4 bits must come from someplace else. 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 3.9) must be careful to avoid placing a program across an address boundary of 256 MB (64 million instructions), for 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.

MIPS Addressing Mode Summary

We have seen two new forms of addressing in this section. Multiple forms of addressing are generically called *addressing modes*. The MIPS addressing modes are the following:

1. *Register addressing*, where the operand is a register
2. *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
3. *Immediate addressing*, where the operand is a constant within the instruction itself
4. *PC-relative addressing*, where the 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

Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (`addi`) and register (`add`) addressing. Figure 3.17 shows how operands are identified for each addressing mode. Section 3.12 expands this list to show addressing modes found in other styles of computers.

Hardware Software Interface

Although we show the MIPS architecture as having 32-bit addresses, nearly all microprocessors (including MIPS) have 64-bit address extensions. (See Web Extension I at *www.mkp.com/cod2e.htm*.) These extensions were in response to the needs of software for larger programs. The process of instruction set extension allows architectures to be expanded in a way that lets software move 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 a core dump. Figure 3.18 shows the MIPS encoding of the fields for the MIPS machine language. This figure can be used to translate by hand between assembly language and machine language.

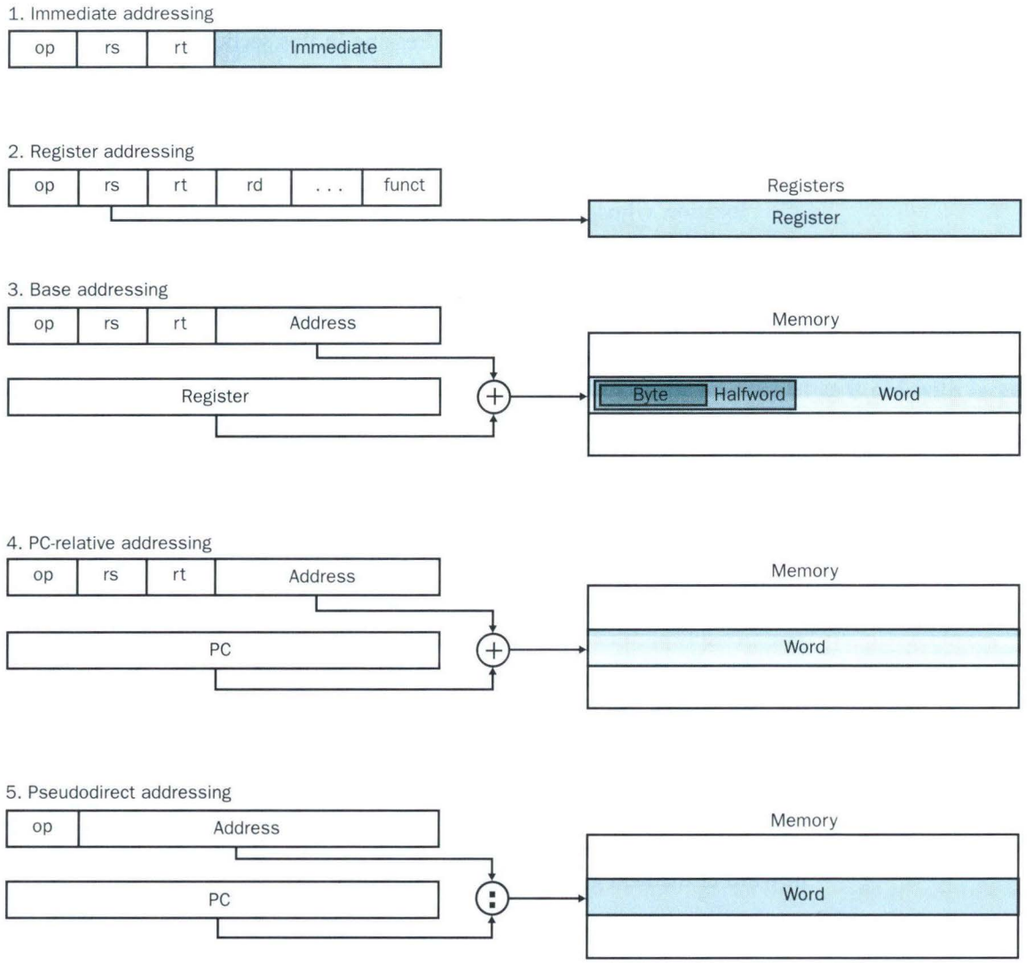


FIGURE 3.17 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1 the operand is 16 bits of the instruction itself. Modes 4 and 5 are used to address instructions in memory, with mode 4 adding a 16-bit address to the PC and mode 5 concatenating a 26-bit address with the upper bits of the PC.

op(31:26)

31–29 \ 28–26	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	lh	lwl	load word	lbu	lhu	lwr	
5(101)	store byte	sh	swl	store word			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26)=010000 (TLB), rs(25:21)

25–24 \ 23–21	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)

5–3 \ 2–0	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	sll		srl	sra	sllv		srlv	srav
1(001)	jump reg.	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	nor
5(101)			set l.t.	sltu				
6(110)								
7(111)								

FIGURE 3.18 MIPS instruction encoding. This notation gives the value of a field by row and by column. For example, in the top portion of the figure load word is found in row number 4 (100_{two} for bits 31–29 of the instruction) and column number 3 (011_{two} for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011_{two}. Underscore means the field is used elsewhere. For example, R-format in row 0 and column 0 (op = 000000_{two}) 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_{two} and the op field (bits 31–26) is 000000_{two}. The FlPt value in row 2, column 1 is defined in Figure 4.48 on page 292 in Chapter 4. Bltz/gez is the opcode for four instructions found in Appendix A: bltz, bgez, bltzal, and bgezal. Instructions given in full name using color are described in Chapter 3, while instructions given in mnemonics using color are described in Chapter 4. Appendix A covers all instructions.

Decoding Machine Code

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

(Bits: *31 28 26* *5 2 0)*

```
0000 0000 1010 1111 1000 0000 0010 0000
```

Answer The first step is to look at the op field to determine the operation. Referring to Figure 3.18, 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 3.19:

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

The bottom portion of Figure 3.18 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, 16 for rd (shamt is unused). Figure 3.13 on page 140 says these numbers represent registers `$a1`, `$t7`, and `$s0`. Now we can show the assembly instruction:

```
add $s0,$a1,$t7
```

Figure 3.20 shows the MIPS assembly language revealed in Chapter 3; the remaining hidden portion of MIPS instructions deals mainly with arithmetic, covered in the next chapter.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
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 3.19 MIPS instruction formats in Chapter 3. Highlighted portions show instruction formats introduced in this section.

MIPS operands

Name	Example	Comments
32 registers	`$s0–$s7, $t0–$t9, $zero, $a0–$a3, $v0–$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. Register `$at` is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1,$s2,100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load byte	`lb $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1,100`	`$s1 = 100 * 2^16`	Loads constant in upper 16 bits
Conditional branch	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 less than immediate	`slti $s1,$s2,100`	if (`$s2 < 100`) `$s1 = 1`; else `$s1 = 0`	Compare less than constant
Unconditional jump	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 3.20 MIPS assembly language revealed in Chapter 3. Highlighted portions show portions from sections 3.7 and 3.8.

3.9 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 3.21 shows the translation hierarchy. Some systems combine these steps to reduce translation time, but these are the logical four phases that all programs go through. This section follows this translation hierarchy.

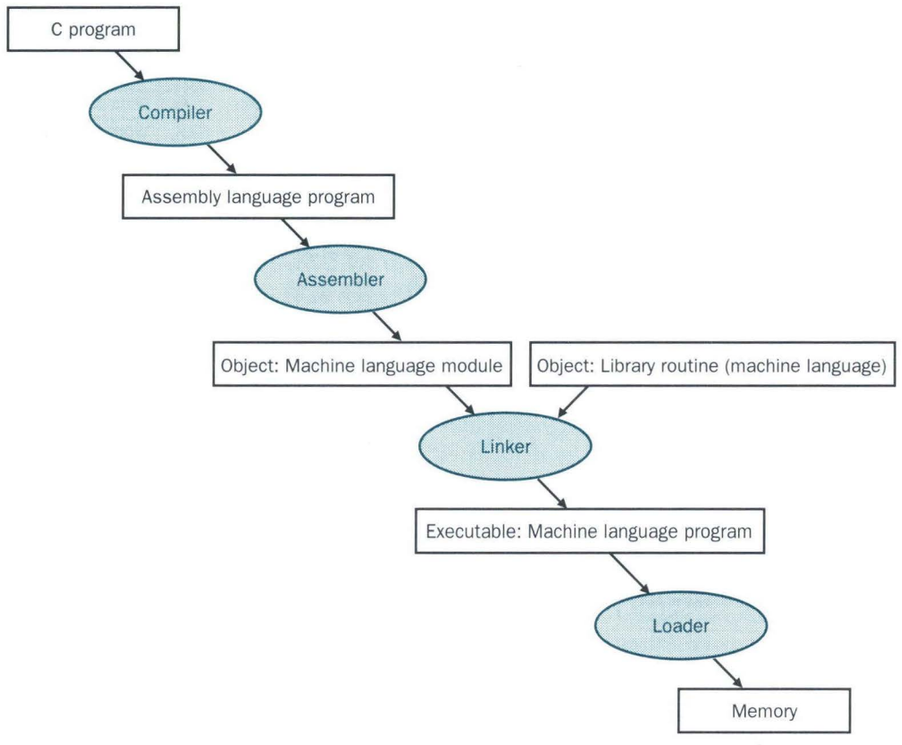


FIGURE 3.21 A translation hierarchy. A high-level-language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined together. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, Unix follows a suffix convention for files: C source files are named `x.c`, assembly files are `x.s`, object files are named `x.o`, and an executable file by default is called `a.out`. MS-DOS uses the suffixes `.C`, `.ASM`, `.OBJ`, and `.EXE` to the same effect.

Compiler

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

In 1975 many operating systems and assemblers were written in assembly language because memories were small and compilers were inefficient. The 16,000-fold increase in memory capacity per DRAM chip has reduced program size concerns, and optimizing compilers today can produce assembly language programs nearly as good as an assembly language expert, and sometimes even better for large programs.

Assembler

As mentioned on page 147, since assembly language is the 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. These instructions need not be implemented in hardware; however, their appearance in assembly language simplifies translation and programming. Such instructions are called *pseudoinstructions*.

For example, 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 `move` 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
```

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 128. 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 can even allow 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 to be sure to get best performance, however, study the real MIPS instructions found in Figures 3.18 and 3.20.

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 can easily be converted to a bit pattern. MIPS assemblers use base 16, called *hexadecimal*; we use the subscript "hex" to indicate a hexadecimal number. The hexadecimal digits are 0 to 9 for the first 10 digits and then the letters *a* to *f* for the last 6 digits. For example, the bit pattern from the example on page 154 is shown as both binary and hexadecimal numbers:

$$0000\ 0000\ 1010\ 1111\ 1000\ 0000\ 0010\ 0000_{two} = 00af\ 8020_{hex}$$

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 symbol and address.

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 *data segment* contains whatever data that comes with the program: either *static data*, which is allocated throughout the program, or *dynamic data*, which can grow or shrink as needed by the program.
- The *relocation information* identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.
- The *symbol table* contains the remaining labels that are not defined, such as external references.
- The *debugging information* contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable.

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

Linker

What we have presented so far suggests that a single change to one line of one procedure requires compiling and assembling the whole program. Complete retranslation is a terrible waste of computing resources. This repetition is particularly wasteful for standard library routines because programmers would be compiling and assembling routines that by definition almost never change. An alternative is to compile and assemble each procedure independently, so that a change to one line would require compiling and assembling only one procedure. This alternative requires a new systems program, called a *link editor* or *linker*, that 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 makes sense 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. Figure 3.22 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 will 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, relocation information, symbol table, or debugging information. It is possible to have partially linked files, such as library routines, which still have unresolved addresses and hence result in object files.

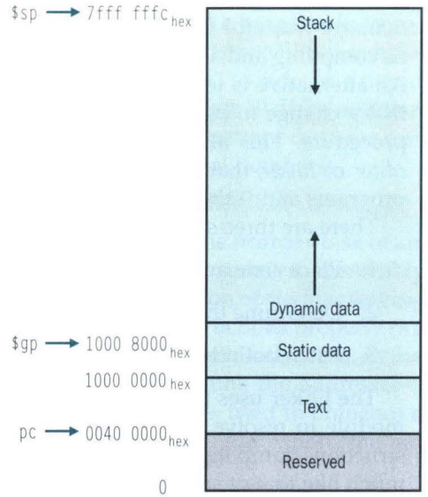


FIGURE 3.22 The MIPS memory allocation for program and data. Starting top down, the stack pointer is initialized to $7fff\ fffc_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at $0040\ 0000_{hex}$. The static data starts at $1000\ 0000_{hex}$. Dynamic data, allocated by `malloc` in C, is next and grows up toward the stack. 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` (see two's complement addressing in Chapter 4).

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_{hex}	
	Data size	20_{hex}	
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_{hex}	
	Data size	30_{hex}	
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.

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

From Figure 3.22 we know that the text segment starts at address $40\ 0000_{hex}$ and the data segment at $1000\ 0000_{hex}$. The text of procedure A is placed at the first address and its data at the second. The object file header for procedure A says that its text is 100_{hex} bytes and its data is 20_{hex} bytes, so the starting address for procedure B text is $40\ 0100_{hex}$, and its data starts at $1000\ 0020_{hex}$.

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 jals are easy because they use pseudodirect addressing. The jal at address $40\ 0004_{hex}$ gets $40\ 0100_{hex}$ (the address of procedure B) in its address field, and the jal at $40\ 0104_{hex}$ gets $40\ 0000_{hex}$ (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. In this example, the global pointer is used as the base register. Figure 3.22 shows that $gp is initialized to $1000\ 8000_{hex}$. To get the address $1000\ 0000_{hex}$ (the address of word X), we place 8000_{hex} in the address field of lw at address $40\ 0000_{hex}$. Chapter 4 explains 16-bit two's complement computer arithmetic, which is why 8000_{hex} in the address field yields $1000\ 0000_{hex}$ as the address. Similarly, we place 8020_{hex} in the address field of sw at address $40\ 0100_{hex}$ to get the address $1000\ 0020_{hex}$ (the address of word Y).

Loader

Now that the executable file is on disk, the operating system reads it to memory and starts it. It 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.

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

3.10 An 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 and the next, we derive the MIPS code from two procedures written in C: one to swap array elements and one to sort them.

The Procedure swap

Let's start with the code for the procedure swap in Figure 3.23. This procedure simply swaps two locations in memory. When translating from C to assembly language, 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.

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

FIGURE 3.23 A C procedure that swaps two locations in memory. This procedure will be used in the sorting example in the next section. Web Extension II at *www.mkp.com/cod2e.htm* shows the C and Pascal versions of this procedure side by side.

Register Allocation for `swap`

As mentioned on page 132, 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 136). This register allocation corresponds to the variable declarations in the first part of the `swap` procedure in Figure 3.23.

Code for the Body of the Procedure `swap`

The remaining lines of C code in `swap` are

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

Recall that the memory address for 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:

```
add   $t1, $a1,$a1   # reg $t1 = k * 2
add   $t1, $t1,$t1   # 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)
```


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

FIGURE 3.24 MIPS assembly code of the procedure `swap` in Figure 3.23.

Now we have allocated registers and written the code to perform the operations of the procedure. The only missing code is the code that preserves for the caller the saved registers that are 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 3.24 each block of code with its purpose in the procedure.

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. Figure 3.25 shows the C version of the program. Once again we present this procedure in several steps, concluding with the full procedure.

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 = i + 1) {
```

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

FIGURE 3.25 A C procedure that performs a sort on the array v. In case you are unfamiliar with C, the three parts of the first *for* statement are the initialization that happens before the first iteration (`i = 0`), the test if the loop should iterate again (`i < n`), and the operation that happens at the end of each iteration (`i = i + 1`). Web Extension II at *www.mkp.com/cod2e.htm* shows the C and Pascal versions of this procedure side by side.

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 157.) It also takes just one instruction to increment `i`, the last part of the *for* statement:

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

The loop should be exited if `i < n` is *not* true, or, said another way, should be exited if `i ≥ n`. The set on less than instruction sets register `$t0` to 1 if `$s0 < $a1` and 0 otherwise. Since we want to test if `$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 = i + 1
         j    for1tst        # jump to test of outer loop
exit1:
```


Voila! Exercise 3.9 explores 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 = 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 = 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 ≥ 0`.

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

```
add   $t1, $s1,$s1      # reg $t1 = j * 2
add   $t1, $t1,$t1      # 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] ≤ v[j + 1]` is the same as `v[j + 1] ≥ 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 together, 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)
   add   $t1, $s1,$s1       # reg $t1 = j * 2
```