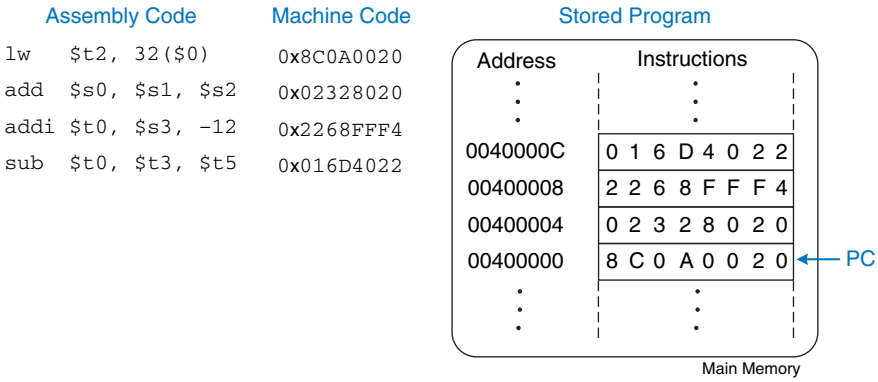

Figure 6.13 Stored program

To run or *execute* the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in a 32-bit register called the *program counter* (`PC`). The `PC` is separate from the 32 registers shown previously in Table 6.1.

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

The *architectural state* of a microprocessor holds the state of a program. For MIPS, the architectural state consists of the register file and `PC`. If the operating system saves the architectural state at some point in the program, it can interrupt the program, do something else, then restore the state such that the program continues properly, unaware that it was ever interrupted. The architectural state is also of great importance when we build a microprocessor in Chapter 7.

Ada Lovelace, 1815–1852. Wrote the first computer program. It calculated the Bernoulli numbers using Charles Babbage's Analytical Engine. She was the only legitimate child of the poet Lord Byron.

6.4 PROGRAMMING

Software languages such as C or Java are called high-level programming languages because they are written at a more abstract level than assembly language. Many high-level languages use common software constructs such as arithmetic and logical operations, `if/else` statements, `for` and `while` loops, array indexing, and function calls. See Appendix C for more examples of these constructs in C. In this section, we explore how to translate these high-level constructs into MIPS assembly code.

6.4.1 Arithmetic/Logical Instructions

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

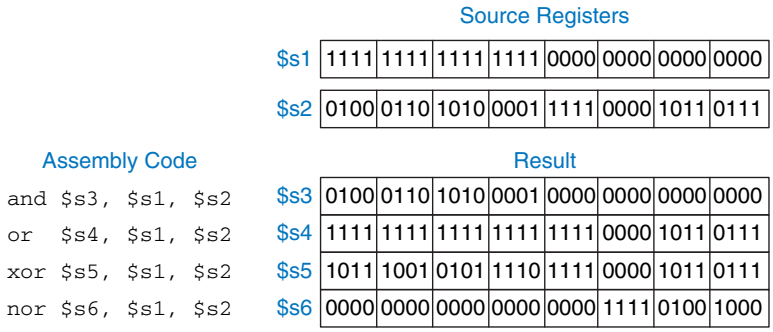

Figure 6.14 Logical operations

Logical Instructions

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

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

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

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

Logical operations can also operate on immediates. These I-type instructions are `andi`, `ori`, and `xori`. `nori` is not provided, because the same functionality can be easily implemented using the other instructions, as will be explored in Exercise 6.16. Figure 6.15 shows examples of the `andi`, `ori`, and `xori` instructions. The figure gives the values of the source

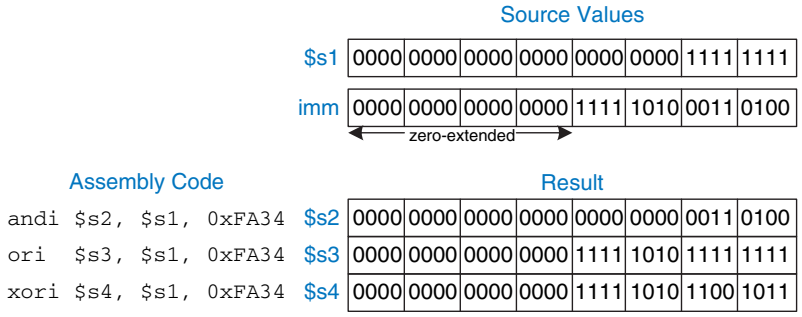

Figure 6.15 Logical operations with immediates

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

Shift Instructions

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

As discussed in Section 5.2.5, left shifts always fill the least significant bits with 0's. However, right shifts can be either logical (0's shift into the most significant bits) or arithmetic (the sign bit shifts into the most significant bits). Figure 6.16 shows the machine code for the R-type instructions `sll`, `srl`, and `sra`. `rt` (i.e., `$s1`) holds the 32-bit value to be shifted, and `shamt` gives the amount by which to shift (4). The shifted result is placed in `rd`.

Figure 6.17 shows the register values for the shift instructions `sll`, `srl`, and `sra`. Shifting a value left by N is equivalent to multiplying it by 2^N. Likewise, arithmetically shifting a value right by N is equivalent to dividing it by 2^N, as discussed in Section 5.2.5.

MIPS also has variable-shift instructions: `sllv` (shift left logical variable), `srlv` (shift right logical variable), and `srav` (shift right arithmetic variable). Figure 6.18 shows the machine code for these instructions. Variable-shift assembly instructions are of the form `sllv rd, rt, rs`. The order of `rt` and `rs` is reversed from most R-type instructions. `rt` (`$s1`) holds the value to be shifted, and the five least significant bits of `rs` (`$s2`) give the amount to shift. The shifted result is placed in `rd`, as before. The `shamt`

Assembly Code	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	Machine Code
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
`sll $t0, $s1, 4`	0	0	17	8	4	0	000000	00000	10001	01000	00100	000000	(0x00114100)
`srl $s2, $s1, 4`	0	0	17	18	4	2	000000	00000	10001	10010	00100	000010	(0x00119102)
`sra $s3, $s1, 4`	0	0	17	19	4	3	000000	00000	10001	10011	00100	000011	(0x00119903)

Figure 6.16 Shift instruction machine code

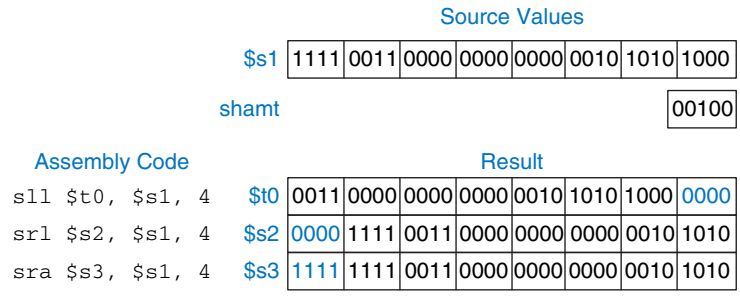

Figure 6.17 Shift operations

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

Figure 6.18 Variable-shift instruction machine code

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

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

Figure 6.19 Variable-shift operations

field is ignored and should be all 0's. Figure 6.19 shows register values for each type of variable-shift instruction.

Generating Constants

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

Code Example 6.10 16-BIT CONSTANT

High-Level Code	MIPS Assembly Code
`int a = 0x4f3c;`	`# $s0 = a` `addi $s0, $0, 0x4f3c  # a = 0x4f3c`

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

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

Code Example 6.11 32-BIT CONSTANT

High-Level Code	MIPS Assembly Code

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

Multiplication and Division Instructions*

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

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

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

MIPS provides another multiply instruction that produces a 32-bit result in a general purpose register. `mul $s1, $s2, $s3` multiplies the values in `$s2` and `$s3` and places the 32-bit result in `$s1`.

6.4.2 Branching

An advantage of a computer over a calculator is its ability to make decisions. A computer performs different tasks depending on the input. For example, `if/else` statements, `switch/case` statements, `while` loops, and `for` loops all conditionally execute code depending on some test.

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

Conditional Branches

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

When the program in Code Example 6.12 reaches the branch if equal instruction (`beq`), the value in `$s0` is equal to the value in `$s1`, so the branch is *taken*. That is, the next instruction executed is the `add` instruction just after the *label* called `target`. The two instructions directly after the branch and before the label are not executed.[2]

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

[2] In practice, because of pipelining (discussed in Chapter 7), MIPS processors have a *branch delay slot*. This means that the instruction immediately after a branch or jump is always executed. This idiosyncracy is ignored in MIPS assembly code in this chapter.

Code Example 6.12 CONDITIONAL BRANCHING USING `beq`

MIPS Assembly Code

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

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

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

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

Code Example 6.13 CONDITIONAL BRANCHING USING `bne`

MIPS Assembly Code

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

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

Jump

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

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

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

Code Example 6.14 UNCONDITIONAL BRANCHING USING `j`

MIPS Assembly Code

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

Code Example 6.15 UNCONDITIONAL BRANCHING USING `jr`

MIPS Assembly Code

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

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

6.4.3 Conditional Statements

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

If Statements

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

Code Example 6.16 `if` STATEMENT

High-Level Code

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

f = f - i;
```

MIPS Assembly Code

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

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

If/Else Statements

`if/else` statements execute one of two blocks of code depending on a condition. When the condition in the `if` statement is met, the *if block* is executed. Otherwise, the *else block* is executed. Code Example 6.17 shows an example `if/else` statement.

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

Code Example 6.17 `if/else` STATEMENT

High-Level Code

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

else
  f = f - i;
```

MIPS Assembly Code

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

Switch/Case Statements*

`switch/case` statements execute one of several blocks of code depending on the conditions. If no conditions are met, the `default` block is executed. A `case` statement is equivalent to a series of *nested* `if/else` statements. Code Example 6.18 shows two high-level code snippets with the same functionality: they calculate the fee for an ATM (automatic teller machine) withdrawal of $20, $50, or $100, as defined by `amount`. The MIPS assembly implementation is the same for both high-level code snippets.

6.4.4 Getting Loopy

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

Code Example 6.18 `switch/case` STATEMENT

High-Level Code

```
switch (amount) {
  case 20:   fee = 2; break;
  case 50:   fee = 3; break;
  case 100:  fee = 5; break;
  default: fee = 0;
}

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

MIPS Assembly Code

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

While Loops

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

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

Code Example 6.19 `while` LOOP

High-Level Code

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

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

MIPS Assembly Code

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

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

In Code Example 6.19, the `while` loop compares `pow` to 128 and exits the loop if it is equal. Otherwise it doubles `pow` (using a left shift), increments `x`, and jumps back to the start of the `while` loop.

`do/while` loops are similar to `while` loops except they execute the loop body once before checking the condition. They are of the form:

```
do
  statement
while (condition);
```

For Loops

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

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

The `initialization` code executes before the `for` loop begins. The `condition` is tested at the beginning of each loop. If the condition is not met, the loop exits. The `loop operation` executes at the end of each loop.

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

Magnitude Comparison

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

Code Example 6.20 `for` LOOP

High-Level Code

```
int sum = 0;



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


}

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

MIPS Assembly Code

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

Example 6.6 LOOPS USING `slt`

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

```
// high-level code

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

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

```
# MIPS assembly code

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

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

done:
```

Exercise 6.17 explores how to use `slt` for other magnitude comparisons including greater than, greater than or equal, and less than or equal.

6.4.5 Arrays

Arrays are useful for accessing large amounts of similar data. An array is organized as sequential data addresses in memory. Each array element is identified by a number called its *index*. The number of elements in the array is called the *size* of the array. This section shows how to access array elements in memory.

Array Indexing

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

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

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

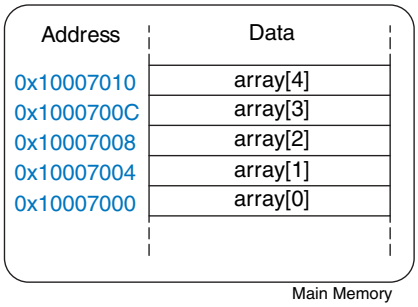

Figure 6.20 Five-entry array with base address of 0x10007000

Code Example 6.21 ACCESSING ARRAYS

High-Level Code

```
int array[5];

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

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

MIPS Assembly Code

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

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

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

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

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

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

Figure 6.21 shows the 1000-element array in memory. The index into the array is now a variable (`i`) rather than a constant, so we cannot take advantage of the immediate offset in `lw`. Instead, we compute the address of the *i*th element and store it in `$t0`. Remember that each array element is a word but that memory is byte addressed, so the offset from the base address is `i * 4`.

Code Example 6.22 ACCESSING ARRAYS USING A `for` LOOP

High-Level Code

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

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

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

MIPS Assembly Code

```
# $s0 = array base address, $s1 = i
# initialization code
  lui  $s0, 0x23B8       # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000  # $s0 = 0x23B8F000
  addi $s1, $0, 0        # i = 0
  addi $t2, $0, 1000     # $t2 = 1000
loop:
  slt  $t0, $s1, $t2     # i < 1000?
  beq  $t0, $0, done     # if not, then done
  sll  $t0, $s1, 2       # $t0 = i*4 (byte offset)
  add  $t0, $t0, $s0     # address of array[i]
  lw   $t1, 0($t0)       # $t1 = array[i]
  sll  $t1, $t1, 3       # $t1 = array[i] * 8
  sw   $t1, 0($t0)       # array[i] = array[i] * 8
  addi $s1, $s1, 1       # i = i + 1
  j    loop              # repeat
done:
```

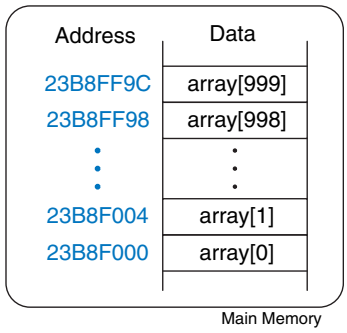

Figure 6.21 **Memory holding `array[1000]` starting at base address 0x23B8F000**

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

Bytes and Characters

Numbers in the range [–128, 127] can be stored in a single byte rather than an entire word. Because there are much fewer than 256 characters on an English language keyboard, English characters are often represented by bytes. The C language uses the type `char` to represent a byte or character.

Early computers lacked a standard mapping between bytes and English characters, so exchanging text between computers was difficult. In 1963, the American Standards Association published the *American Standard Code for Information Interchange* (*ASCII*), which assigns each text character a unique byte value. Table 6.2 shows these character encodings for printable characters. The ASCII values are given in hexadecimal. Lower-case and upper-case letters differ by 0x20 (32).

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

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

Table 6.2 ASCII encodings

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

ASCII codes developed from earlier forms of character encoding. Beginning in 1838, telegraph machines used Morse code, a series of dots (.) and dashes (–), to represent characters. For example, the letters A, B, C, and D were represented as . – , – . . . , – . – . , and – . . , respectively. The number of dots and dashes varied with each letter. For efficiency, common letters used shorter codes.

In 1874, Jean-Maurice-Emile Baudot invented a 5-bit code called the Baudot code. For example, A, B, C, and D were represented as 00011, 11001, 01110, and 01001. However, the 32 possible encodings of this 5-bit code were not sufficient for all the English characters. But 8-bit encoding was. Thus, as electronic communication became prevalent, 8-bit ASCII encoding emerged as the standard.

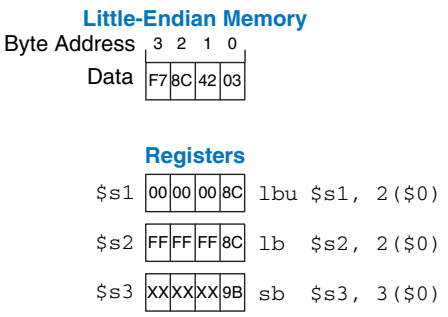

Figure 6.22 Instructions for loading and storing bytes

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

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

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

```
// high-level code

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

Solution:

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

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

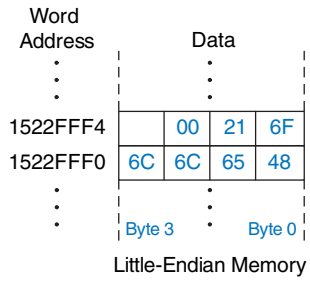

Figure 6.23 The string "Hello!" stored in memory

A series of characters is called a *string*. Strings have a variable length, so programming languages must provide a way to determine the length or end of the string. In C, the null character (0x00) signifies the end of a string. For example, Figure 6.23 shows the string "Hello!" (0x48 65 6C 6C 6F 21 00) stored in memory. The string is seven bytes long and extends from address 0x1522FFF0 to 0x1522FFF6. The first character of the string (H = 0x48) is stored at the lowest byte address (0x1522FFF0).

6.4.6 Function Calls

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

When one function calls another, the calling function, the *caller*, and the called function, the *callee*, must agree on where to put the arguments and the return value. In MIPS, the caller conventionally places up to four arguments in registers `$a0`–`$a3` before making the function call, and the callee places the return value in registers `$v0`–`$v1` before finishing. By following this convention, both functions know where to find the arguments and return value, even if the caller and callee were written by different people.

The callee must not interfere with the function of the caller. Briefly, this means that the callee must know where to return to after it completes and it must not trample on any registers or memory needed by the caller. The caller stores the *return address* in `$ra` at the same time it jumps to the callee using the jump and link instruction (`jal`). The callee must not overwrite any architectural state or memory that the caller is depending on. Specifically, the callee must leave the saved registers, `$s0`–`$s7`, `$ra`, and the *stack*, a portion of memory used for temporary variables, unmodified.

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

Function Calls and Returns

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

Code Example 6.23 `simple` FUNCTION CALL

High-Level Code

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

MIPS Assembly Code

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


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

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

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

Input Arguments and Return Values

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

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

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

Code Example 6.24 has some subtle errors. Code Examples 6.25 and 6.26 on pages 328 and 329 show improved versions of the program.

Code Example 6.24 FUNCTION CALL WITH ARGUMENTS AND RETURN VALUES

High-Level Code

```
int main()
{
  int y;

  . . .

  y = diffofsums(2, 3, 4, 5);

  . . .
}

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

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

MIPS Assembly Code

```
# $s0 = y

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

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

The Stack

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

The stack is a *last-in-first-out* (*LIFO*) *queue*. Like a stack of dishes, the last item *pushed* onto the stack (the top dish) is the first one that can be pulled (*popped*) off. Each function may allocate stack space to store local variables but must deallocate it before returning. The *top of the stack*, is the most recently allocated space. Whereas a stack of dishes grows up in space, the MIPS stack grows *down* in memory. The stack expands to lower memory addresses when a program needs more scratch space.

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

The stack pointer (`$sp`) starts at a high memory address and decrements to expand as needed. Figure 6.24(b) shows the stack expanding to allow two more data words of temporary storage. To do so, `$sp` decrements by 8 to become 0x7FFFFFF4. Two additional data words, 0xAABBCCDD and 0x11223344, are temporarily stored on the stack.

One of the important uses of the stack is to save and restore registers that are used by a function. Recall that a function should calculate a return value but have no other unintended side effects. In particular, it should not modify any registers besides the one containing the return value `$v0`. The `diffofsums` function in Code Example 6.24 violates this rule because it modifies `$t0`, `$t1`, and `$s0`. If `main` had been using `$t0`, `$t1`, or `$s0` before the call to `diffofsums`, the contents of these registers would have been corrupted by the function call.

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

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

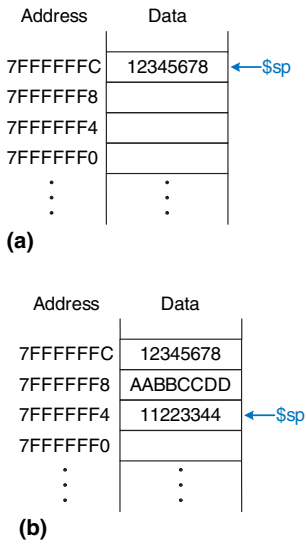

Figure 6.24 The stack

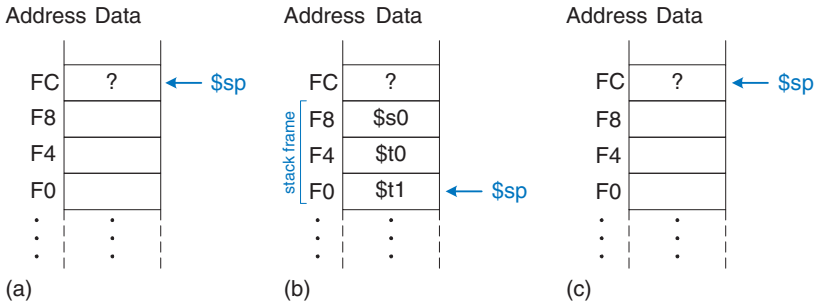

Figure 6.25 **The stack (a) before, (b) during, and (c) after `diffofsums` function call**

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

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

Code Example 6.25 FUNCTION SAVING REGISTERS ON THE STACK

MIPS Assembly Code

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

Preserved Registers

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

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

Remember that when one function calls another, the former is the *caller* and the latter is the *callee*. The callee must save and restore any preserved registers that it wishes to use. The callee may change any of the nonpreserved registers. Hence, if the caller is holding active data in a nonpreserved register, the caller needs to save that nonpreserved register before making the function call and then needs to restore it afterward. For these reasons, preserved registers are also called *callee-save*, and nonpreserved registers are called *caller-save*.

Code Example 6.26 FUNCTION SAVING PRESERVED REGISTERS ON THE STACK

MIPS Assembly Code

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

Table 6.3 summarizes which registers are preserved. `$s0`–`$s7` are generally used to hold local variables within a function, so they must be saved. `$ra` must also be saved, so that the function knows where to return. `$t0`–`$t9` are used to hold temporary results before they are assigned to local variables. These calculations typically complete before a function call is made, so they are not preserved, and it is rare that the caller needs to save them. `$a0`–`$a3` are often overwritten in the process of calling a function.

Table 6.3 Preserved and nonpreserved registers

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

Hence, they must be saved by the caller if the caller depends on any of its own arguments after a called function returns. `$v0–$v1` certainly should not be preserved, because the callee returns its result in these registers.

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

Recursive Function Calls

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

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

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

Code Example 6.27 `factorial` RECURSIVE FUNCTION CALL

High-Level Code

```
int factorial(int n) {


  if (n <= 1)
    return 1;


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

MIPS Assembly Code

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

instruction (`mul $v0, $a0, $v0`) multiplies `$a0` and `$v0` and places the result in `$v0`.

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

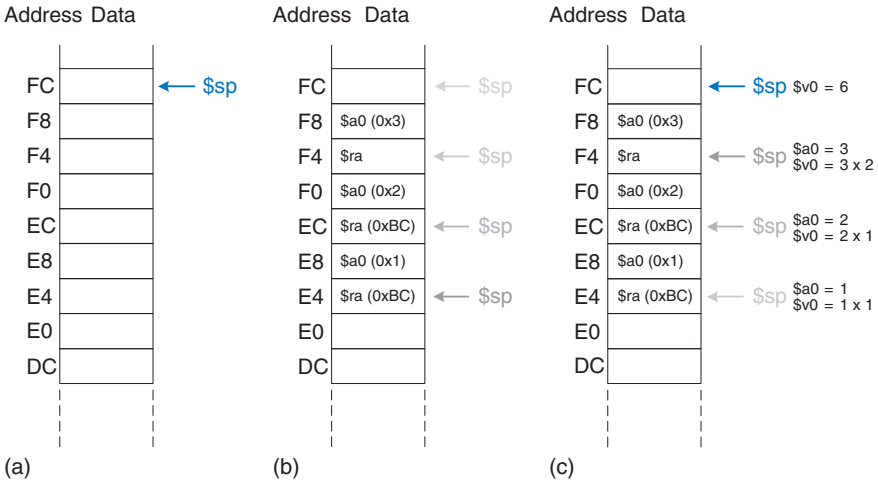

Figure 6.26 Stack during `factorial` function call when `n` = 3: (a) before call, (b) after last recursive call, (c) after return

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

Additional Arguments and Local Variables*

Functions may have more than four input arguments and local variables. The stack is used to store these temporary values. By MIPS convention, if a function has more than four arguments, the first four are passed in the argument registers as usual. Additional arguments are passed on the stack, just above `$sp`. The *caller* must expand its stack to make room for the additional arguments. Figure 6.27(a) shows the caller's stack for calling a function with more than four arguments.

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

Figure 6.27(b) shows the organization of a callee's stack frame. The stack frame holds the function's own arguments, the return address, and any of the saved registers that the function will modify. It also holds local arrays and any excess local variables. If the callee has more than four

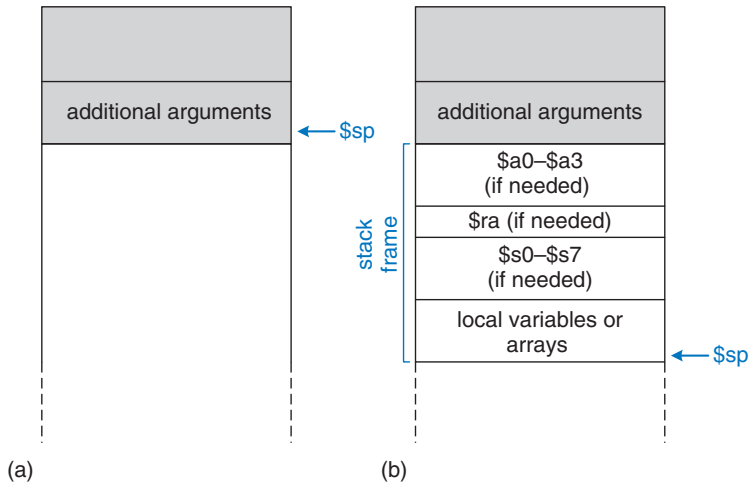

Figure 6.27 Stack usage: (a) before call, (b) after call

arguments, it finds them in the caller's stack frame. Accessing additional input arguments is the one exception in which a function can access stack data not in its own stack frame.

6.5 ADDRESSING MODES

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

Register-Only Addressing

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

Immediate Addressing

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

Base Addressing

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

PC-Relative Addressing

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

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

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

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

Code Example 6.28 CALCULATING THE BRANCH TARGET ADDRESS

MIPS Assembly Code

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

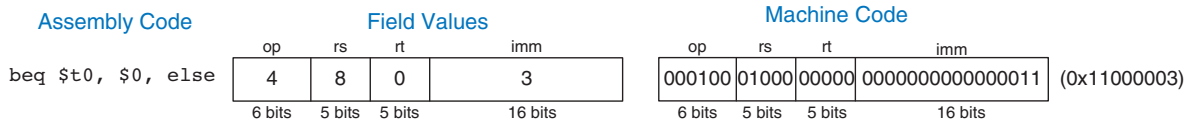

Figure 6.28 **Machine code for** `beq`

Example 6.8 CALCULATING THE IMMEDIATE FIELD FOR PC-RELATIVE ADDRESSING

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

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

Solution:

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

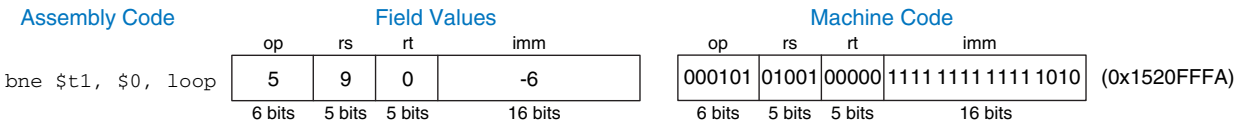

Figure 6.29 `bne` **machine code**

Pseudo-Direct Addressing

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

Code Example 6.29 CALCULATING THE JUMP TARGET ADDRESS

MIPS Assembly Code

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

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

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

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

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

Because the four most significant bits of the JTA are taken from `PC` + 4, the jump range is limited. The range limits of branch and jump instructions are explored in Exercises 6.29 to 6.32. All J-type instructions, `j` and `jal`, use pseudo-direct addressing.

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

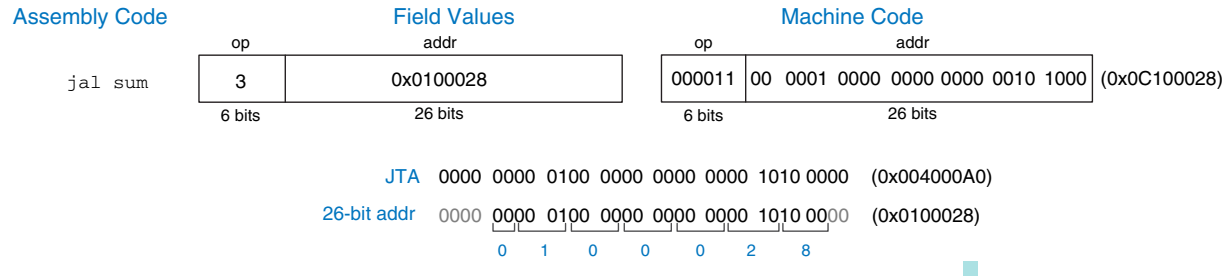

Figure 6.30 `jal` **machine code**

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

Up until now, we have shown how to translate short high-level code snippets into assembly and machine code. This section describes how to compile and assemble a complete high-level program and how to load the program into memory for execution.

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

6.6.1 The Memory Map

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

The Text Segment

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

The Global Data Segment

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

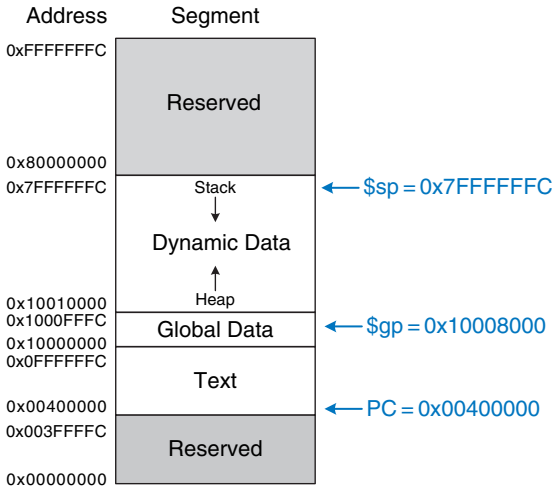

Figure 6.31 MIPS memory map

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

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

The Dynamic Data Segment

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

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

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

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

The Reserved Segments

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

Grace Hopper, 1906–1992. Graduated from Yale University with a Ph.D. in mathematics. Developed the first compiler while working for the Remington Rand Corporation and was instrumental in developing the COBOL programming language. As a naval officer, she received many awards, including a World War II Victory Medal and the National Defense Service Medal.

6.6.2 Translating and Starting a Program

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

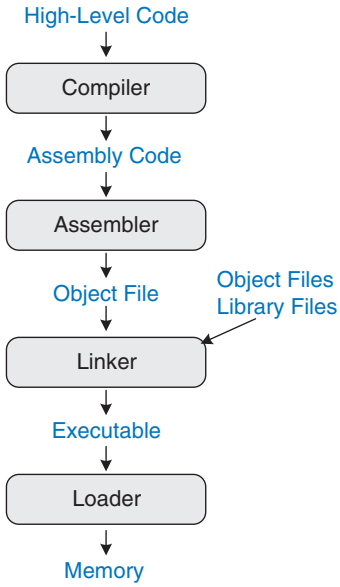

Figure 6.32 Steps for translating and starting a program

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

Step 1: Compilation

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

Code Example 6.30 COMPILING A HIGH-LEVEL PROGRAM

High-Level Code

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



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



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

MIPS Assembly Code

```
.data
f:
g:
y:

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

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

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

Step 2: Assembling

The assembler turns the assembly language code into an *object file* containing machine language code. The assembler makes two passes through the assembly code. On the first pass, the assembler assigns instruction addresses and finds all the *symbols*, such as labels and global variable names. The code after the first assembler pass is shown here.

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

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

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

Table 6.4 Symbol table

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

Step 3: Linking

Most large programs contain more than one file. If the programmer changes only one of the files, it would be wasteful to recompile and reassemble the other files. In particular, programs often call functions in library files; these library files almost never change. If a file of high-level code is not changed, the associated object file need not be updated.

The job of the linker is to combine all of the object files into one machine language file called the *executable*. The linker relocates the data and instructions in the object files so that they are not all on top of each other. It uses the information in the symbol tables to adjust the addresses of global variables and of labels that are relocated.

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

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

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

Figure 6.33 Executable

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

Step 4: Loading

The operating system loads a program by reading the text segment of the executable file from a storage device (usually the hard disk) into the text segment of memory. The operating system sets `$gp` to 0x10008000 (the middle of the global data segment) and `$sp` to 0x7FFFFFFC (the top of the dynamic data segment), then performs a `jal 0x00400000` to jump to the beginning of the program. Figure 6.34 shows the memory map at the beginning of program execution.

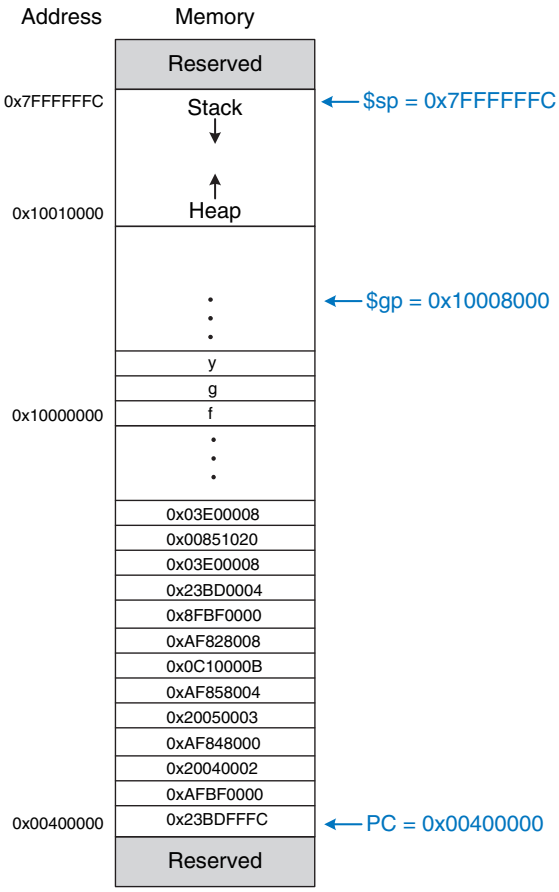

Figure 6.34 Executable loaded in memory

6.7 ODDS AND ENDS*

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

6.7.1 Pseudoinstructions

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

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

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

Some pseudoinstructions require a temporary register for intermediate calculations. For example, the pseudoinstruction `beq $t2,` $\text{imm}_{15:0}$`, Loop` compares `$t2` to a 16-bit immediate, $\text{imm}_{15:0}$. This pseudoinstruction requires a temporary register in which to store the 16-bit immediate. Assemblers use the assembler register, `$at`, for such purposes. Table 6.6 shows

Table 6.5 Pseudoinstructions

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

Table 6.6 Pseudoinstruction using `$at`

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

how the assembler uses `$at` in converting a pseudoinstruction to real MIPS instructions. We leave it as Exercises 6.38 and 6.39 to implement other pseudoinstructions such as rotate left (`rol`) and rotate right (`ror`).

6.7.2 Exceptions

An *exception* is like an unscheduled function call that jumps to a new address. Exceptions may be caused by hardware or software. For example, the processor may receive notification that the user pressed a key on a keyboard. The processor may stop what it is doing, determine which key was pressed, save it for future reference, then resume the program that was running. Such a hardware exception triggered by an input/output (I/O) device such as a keyboard is often called an *interrupt*. Alternatively, the program may encounter an error condition such as an undefined instruction. The program then jumps to code in the *operating system* (*OS*), which may choose to terminate the offending program. Software exceptions are sometimes called *traps*. Other causes of exceptions include division by zero, attempts to read nonexistent memory, hardware malfunctions, debugger breakpoints, and arithmetic overflow (see Section 6.7.3).

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

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

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

Table 6.7 Exception cause codes

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

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

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

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

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

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

6.7.3 Signed and Unsigned Instructions

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

Addition and Subtraction

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

As mentioned in Section 1.4.6, if two large signed numbers are added together, the result may incorrectly produce the opposite sign. For

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

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

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

Multiplication and Division

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

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

Set Less Than

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

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

Loads

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

6.7.4 Floating-Point Instructions

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

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

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

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

Table 6.8 MIPS floating-point register set

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

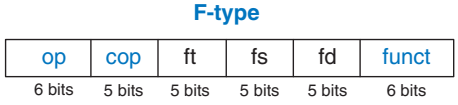

Figure 6.35 F-type machine instruction format

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

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

6.8 REAL-WORLD PERSPECTIVE: x86 ARCHITECTURE*

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

The x86 architecture has a long and convoluted history dating back to 1978, when Intel announced the 16-bit 8086 microprocessor. IBM selected the 8086 and its cousin, the 8088, for IBM's first personal computers. In 1985, Intel introduced the 32-bit 80386 microprocessor, which was backward compatible with the 8086, so it could run software developed for earlier PCs. Processor architectures compatible with the 80386 are called x86 processors. The Pentium, Core, and Athlon processors are well known x86 processors. Section 7.9 describes the evolution of x86 microprocessors in more detail.

Various groups at Intel and AMD over many years have shoehorned more instructions and capabilities into the antiquated architecture. The result is far less elegant than MIPS. As Patterson and Hennessy explain, "this checkered ancestry has led to an architecture that is difficult to explain and impossible to love." However, software compatibility is far more important than technical elegance, so x86 has been the *de facto* PC standard for more than two decades. More than 100 million x86 processors are sold every year. This huge market justifies more than $5 billion of research and development annually to continue improving the processors.

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