A P P E N D I X

Assemblers, Linkers, and the SPIM Simulator

James R. Larus
Microsoft Research
Microsoft

Fear of serious injury cannot alone justify suppression of free speech and assembly.

Louis Brandeis
Whitney v. California, 1927

A.1 **Introduction** A-3
A.2 **Assemblers** A-10
A.3 **Linkers** A-18
A.4 **Loading** A-19
A.5 **Memory Usage** A-20
A.6 **Procedure Call Convention** A-22
A.7 **Exceptions and Interrupts** A-33
A.8 **Input and Output** A-38
A.9 **SPIM** A-40
A.10 **MIPS R2000 Assembly Language** A-45
A.11 **Concluding Remarks** A-81
A.12 **Exercises** A-82

A.1 Introduction

Encoding instructions as binary numbers is natural and efficient for computers. Humans, however, have a great deal of difficulty understanding and manipulating these numbers. People read and write symbols (words) much better than long sequences of digits. Chapter 2 showed that we need not choose between numbers and words because computer instructions can be represented in many ways. Humans can write and read symbols, and computers can execute the equivalent binary numbers. This appendix describes the process by which a human-readable program is translated into a form that a computer can execute, provides a few hints about writing assembly programs, and explains how to run these programs on SPIM, a simulator that executes MIPS programs. UNIX, Windows, and Mac OS X versions of the SPIM simulator are available on the CD.

Assembly language is the symbolic representation of a computer's binary encoding—**machine language**. Assembly language is more readable than machine language because it uses symbols instead of bits. The symbols in assembly language name commonly occurring bit patterns, such as opcodes and register specifiers, so people can read and remember them. In addition, assembly language

machine language Binary representation used for communication within a computer system.

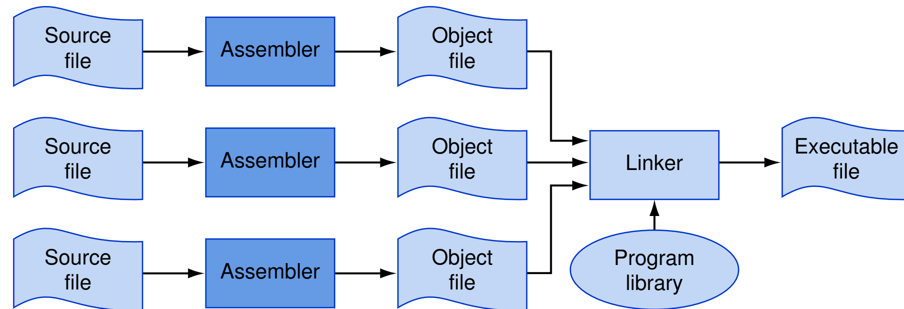

FIGURE A.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

permits programmers to use *labels* to identify and name particular memory words that hold instructions or data.

assembler A program that translates a symbolic version of an instruction into the binary version.

macro A pattern-matching and replacement facility that provides a simple mechanism to name a frequently used sequence of instructions.

A tool called an **assembler** translates assembly language into binary instructions. Assemblers provide a friendlier representation than a computer's 0s and 1s that simplifies writing and reading programs. Symbolic names for operations and locations are one facet of this representation. Another facet is programming facilities that increase a program's clarity. For example, **macros**, discussed in Section A.2, enable a programmer to extend the assembly language by defining new operations.

An assembler reads a single assembly language *source file* and produces an *object file* containing machine instructions and bookkeeping information that helps combine several object files into a program. Figure A.1.1 illustrates how a program is built. Most programs consist of several files—also called *modules*—that are written, compiled, and assembled independently. A program may also use prewritten routines supplied in a *program library*. A module typically contains *references* to subroutines and data defined in other modules and in libraries. The code in a module cannot be executed when it contains **unresolved references** to labels in other object files or libraries. Another tool, called a **linker**, combines a collection of object and library files into an *executable file*, which a computer can run.

unresolved reference A reference that requires more information from an outside source in order to be complete.

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

To see the advantage of assembly language, consider the following sequence of figures, all of which contain a short subroutine that computes and prints the sum of the squares of integers from 0 to 100. Figure A.1.2 shows the machine language that a MIPS computer executes. With considerable effort, you could use the opcode and instruction format tables in Chapter 2 to translate the instructions into a symbolic program similar to Figure A.1.3. This form of the

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

FIGURE A.1.2 MIPS machine language code for a routine to compute and print the sum of the squares of integers between 0 and 100.

routine is much easier to read because operations and operands are written with symbols, rather than with bit patterns. However, this assembly language is still difficult to follow because memory locations are named by their address, rather than by a symbolic label.

Figure A.1.4 shows assembly language that labels memory addresses with mnemonic names. Most programmers prefer to read and write this form. Names that begin with a period, for example `.data` and `.globl`, are **assembler directives** that tell the assembler how to translate a program but do not produce machine instructions. Names followed by a colon, such as `str` or `main`, are labels that name the next memory location. This program is as readable as most assembly language programs (except for a glaring lack of comments), but it is still difficult to follow because many simple operations are required to accomplish simple tasks and because assembly language's lack of control flow constructs provides few hints about the program's operation.

assembler directive An operation that tells the assembler how to translate a program but does not produce machine instructions; always begins with a period.

By contrast, the C routine in Figure A.1.5 is both shorter and clearer since variables have mnemonic names and the loop is explicit rather than constructed with branches. In fact, the C routine is the only one that we wrote. The other forms of the program were produced by a C compiler and assembler.

In general, assembly language plays two roles (see Figure A.1.6). The first role is the output language of compilers. A *compiler* translates a program written in a

```
addiu    $29, $29, -32
sw       $31, 20($29)
sw       $4,  32($29)
sw       $5,  36($29)
sw       $0,  24($29)
sw       $0,  28($29)
lw       $14, 28($29)
lw       $24, 24($29)
multu    $14, $14
addiu    $8,  $14, 1
slti     $1,  $8, 101
sw       $8,  28($29)
mflo     $15
addu     $25, $24, $15
bne      $1,  $0, -9
sw       $25, 24($29)
lui      $4,  4096
lw       $5,  24($29)
jal      1048812
addiu    $4,  $4, 1072
lw       $31, 20($29)
addiu    $29, $29, 32
jr       $31
move     $2,  $0
```

FIGURE A.1.3 The same routine written in assembly language. However, the code for the routine does not label registers or memory locations nor include comments.

source language The high-level language in which a program is originally written.

high-level language (such as C or Pascal) into an equivalent program in machine or assembly language. The high-level language is called the **source language**, and the compiler's output is its *target language*.

Assembly language's other role is as a language in which to write programs. This role used to be the dominant one. Today, however, because of larger main memories and better compilers, most programmers write in a high-level language and rarely, if ever, see the instructions that a computer executes. Nevertheless, assembly language is still important to write programs in which speed or size are critical or to exploit hardware features that have no analogues in high-level languages.

Although this appendix focuses on MIPS assembly language, assembly programming on most other machines is very similar. The additional instructions and address modes in CISC machines, such as the VAX, can make assembly programs shorter but do not change the process of assembling a program or provide assembly language with the advantages of high-level languages such as type-checking and structured control flow.

```
        .text
        .align    2
        .globl    main
main:
        subu      $sp, $sp, 32
        sw        $ra, 20($sp)
        sd        $a0, 32($sp)
        sw        $0,  24($sp)
        sw        $0,  28($sp)
loop:
        lw        $t6, 28($sp)
        mul       $t7, $t6, $t6
        lw        $t8, 24($sp)
        addu      $t9, $t8, $t7
        sw        $t9, 24($sp)
        addu      $t0, $t6, 1
        sw        $t0, 28($sp)
        ble       $t0, 100, loop
        la        $a0, str
        lw        $a1, 24($sp)
        jal       printf
        move      $v0, $0
        lw        $ra, 20($sp)
        addu      $sp, $sp, 32
        jr        $ra

        .data
        .align    0
str:
        .asciiz   "The sum from 0 .. 100 is %d\n"
```

FIGURE A.1.4 The same routine written in assembly language with labels, but no comments. The commands that start with periods are assembler directives (see pages A-47–A-49). `.text` indicates that succeeding lines contain instructions. `.data` indicates that they contain data. `.align n` indicates that the items on the succeeding lines should be aligned on a 2^n byte boundary. Hence, `.align 2` means the next item should be on a word boundary. `.globl main` declares that `main` is a global symbol that should be visible to code stored in other files. Finally, `.asciiz` stores a null-terminated string in memory.

When to Use Assembly Language

The primary reason to program in assembly language, as opposed to an available high-level language, is that the speed or size of a program is critically important. For example, consider a computer that controls a piece of machinery, such as a car's brakes. A computer that is incorporated in another device, such as a car, is called an *embedded computer*. This type of computer needs to respond rapidly and predictably to events in the outside world. Because a compiler introduces uncer-

```
#include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

FIGURE A.1.5 The routine written in the C programming language.

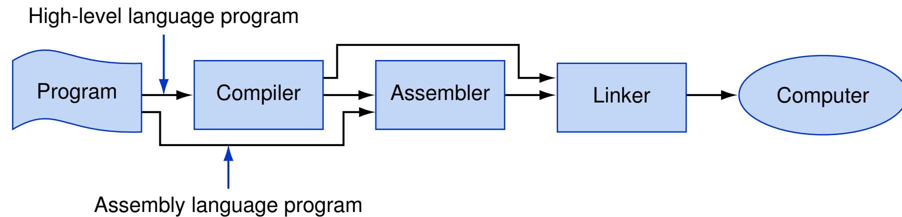

FIGURE A.1.6 Assembly language either is written by a programmer or is the output of a compiler.

tainty about the time cost of operations, programmers may find it difficult to ensure that a high-level language program responds within a definite time interval—say, 1 millisecond after a sensor detects that a tire is skidding. An assembly language programmer, on the other hand, has tight control over which instructions execute. In addition, in embedded applications, reducing a program's size, so that it fits in fewer memory chips, reduces the cost of the embedded computer.

A hybrid approach, in which most of a program is written in a high-level language and time-critical sections are written in assembly language, builds on the strengths of both languages. Programs typically spend most of their time executing a small fraction of the program's source code. This observation is just the principle of locality that underlies caches (see Section 7.2 in Chapter 7).

Program profiling measures where a program spends its time and can find the time-critical parts of a program. In many cases, this portion of the program can be made faster with better data structures or algorithms. Sometimes, however, significant performance improvements only come from recoding a critical portion of a program in assembly language.

This improvement is not necessarily an indication that the high-level language's compiler has failed. Compilers typically are better than programmers at producing uniformly high-quality machine code across an entire program. Programmers, however, understand a program's algorithms and behavior at a deeper level than a compiler and can expend considerable effort and ingenuity improving small sections of the program. In particular, programmers often consider several procedures simultaneously while writing their code. Compilers typically compile each procedure in isolation and must follow strict conventions governing the use of registers at procedure boundaries. By retaining commonly used values in registers, even across procedure boundaries, programmers can make a program run faster.

Another major advantage of assembly language is the ability to exploit specialized instructions, for example, string copy or pattern-matching instructions. Compilers, in most cases, cannot determine that a program loop can be replaced by a single instruction. However, the programmer who wrote the loop can replace it easily with a single instruction.

Currently, a programmer's advantage over a compiler has become difficult to maintain as compilation techniques improve and machines' pipelines increase in complexity (Chapter 6).

The final reason to use assembly language is that no high-level language is available on a particular computer. Many older or specialized computers do not have a compiler, so a programmer's only alternative is assembly language.

Drawbacks of Assembly Language

Assembly language has many disadvantages that strongly argue against its widespread use. Perhaps its major disadvantage is that programs written in assembly language are inherently machine-specific and must be totally rewritten to run on another computer architecture. The rapid evolution of computers discussed in Chapter 1 means that architectures become obsolete. An assembly language program remains tightly bound to its original architecture, even after the computer is eclipsed by new, faster, and more cost-effective machines.

Another disadvantage is that assembly language programs are longer than the equivalent programs written in a high-level language. For example, the C program in Figure A.1.5 is 11 lines long, while the assembly program in Figure A.1.4 is 31 lines long. In more complex programs, the ratio of assembly to high-level language (its *expansion factor*) can be much larger than the factor of three in this example. Unfortunately, empirical studies have shown that programmers write roughly the same number of lines of code per day in assembly as in high-level languages. This means that programmers are roughly x times more productive in a high-level language, where x is the assembly language expansion factor.

To compound the problem, longer programs are more difficult to read and understand and they contain more bugs. Assembly language exacerbates the problem because of its complete lack of structure. Common programming idioms, such as *if-then* statements and loops, must be built from branches and jumps. The resulting programs are hard to read because the reader must reconstruct every higher-level construct from its pieces and each instance of a statement may be slightly different. For example, look at Figure A.1.4 and answer these questions: What type of loop is used? What are its lower and upper bounds?

Elaboration: Compilers can produce machine language directly instead of relying on an assembler. These compilers typically execute much faster than those that invoke an assembler as part of compilation. However, a compiler that generates machine language must perform many tasks that an assembler normally handles, such as resolving addresses and encoding instructions as binary numbers. The trade-off is between compilation speed and compiler simplicity.

Elaboration: Despite these considerations, some embedded applications are written in a high-level language. Many of these applications are large and complex programs that must be extremely reliable. Assembly language programs are longer and more difficult to write and read than high-level language programs. This greatly increases the cost of writing an assembly language program and makes it extremely difficult to verify the correctness of this type of program. In fact, these considerations led the Department of Defense, which pays for many complex embedded systems, to develop Ada, a new high-level language for writing embedded systems.

A.2 Assemblers

An assembler translates a file of assembly language statements into a file of binary machine instructions and binary data. The translation process has two major parts. The first step is to find memory locations with labels so the relationship between symbolic names and addresses is known when instructions are translated. The second step is to translate each assembly statement by combining the numeric equivalents of opcodes, register specifiers, and labels into a legal instruction. As shown in Figure A.1.1, the assembler produces an output file, called an *object file*, which contains the machine instructions, data, and bookkeeping information.

An object file typically cannot be executed because it references procedures or data in other files. A **label** is **external** (also called **global**) if the labeled object can

external label Also called global label. A label referring to an object that can be referenced from files other than the one in which it is defined.

local label A label referring to an object that can be used only within the file in which it is defined.

be referenced from files other than the one in which it is defined. A label is *local* if the object can be used only within the file in which it is defined. In most assemblers, labels are local by default and must be explicitly declared global. Subroutines and global variables require external labels since they are referenced from many files in a program. **Local labels** hide names that should not be visible to other modules—for example, static functions in C, which can only be called by other functions in the same file. In addition, compiler-generated names—for example, a name for the instruction at the beginning of a loop—are local so the compiler need not produce unique names in every file.

Local and Global Labels

EXAMPLE

Consider the program in Figure A.1.4 on page A-7. The subroutine has an external (global) label `main`. It also contains two local labels—`loop` and `str`—that are only visible with this assembly language file. Finally, the routine also contains an unresolved reference to an external label `printf`, which is the library routine that prints values. Which labels in Figure A.1.4 could be referenced from another file?

ANSWER

Only global labels are visible outside of a file, so the only label that could be referenced from another file is `main`.

Since the assembler processes each file in a program individually and in isolation, it only knows the addresses of local labels. The assembler depends on another tool, the linker, to combine a collection of object files and libraries into an executable file by resolving external labels. The assembler assists the linker by providing lists of labels and unresolved references.

However, even local labels present an interesting challenge to an assembler. Unlike names in most high-level languages, assembly labels may be used before they are defined. In the example, in Figure A.1.4, the label `str` is used by the `la` instruction before it is defined. The possibility of a **forward reference**, like this one, forces an assembler to translate a program in two steps: first find all labels and then produce instructions. In the example, when the assembler sees the `la` instruction, it does not know where the word labeled `str` is located or even whether `str` labels an instruction or datum.

forward reference A label that is used before it is defined.

An assembler's first pass reads each line of an assembly file and breaks it into its component pieces. These pieces, which are called *lexemes*, are individual words, numbers, and punctuation characters. For example, the line

```
ble     $t0, 100, loop
```

contains six lexemes: the opcode `ble`, the register specifier `$t0`, a comma, the number `100`, a comma, and the symbol `loop`.

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

If a line begins with a label, the assembler records in its **symbol table** the name of the label and the address of the memory word that the instruction occupies. The assembler then calculates how many words of memory the instruction on the current line will occupy. By keeping track of the instructions' sizes, the assembler can determine where the next instruction goes. To compute the size of a variable-length instruction, like those on the VAX, an assembler has to examine it in detail. Fixed-length instructions, like those on MIPS, on the other hand, require only a cursory examination. The assembler performs a similar calculation to compute the space required for data statements. When the assembler reaches the end of an assembly file, the symbol table records the location of each label defined in the file.

The assembler uses the information in the symbol table during a second pass over the file, which actually produces machine code. The assembler again examines each line in the file. If the line contains an instruction, the assembler combines the binary representations of its opcode and operands (register specifiers or memory address) into a legal instruction. The process is similar to the one used in Section 2.4 in Chapter 2. Instructions and data words that reference an external symbol defined in another file cannot be completely assembled (they are unresolved) since the symbol's address is not in the symbol table. An assembler does not complain about unresolved references since the corresponding label is likely to be defined in another file

The BIG Picture

Assembly language is a programming language. Its principal difference from high-level languages such as BASIC, Java, and C is that assembly language provides only a few, simple types of data and control flow. Assembly language programs do not specify the type of value held in a variable. Instead, a programmer must apply the appropriate operations (e.g., integer or floating-point addition) to a value. In addition, in assembly language, programs must implement all control flow with *go to*s. Both factors make assembly language programming for any machine—MIPS or 80x86—more difficult and error-prone than writing in a high-level language.

Elaboration: If an assembler's speed is important, this two-step process can be done in one pass over the assembly file with a technique known as backpatching. In its pass over the file, the assembler builds a (possibly incomplete) binary representation of every instruction. If the instruction references a label that has not yet been defined, the assembler records the label and instruction in a table. When a label is defined, the assembler consults this table to find all instructions that contain a forward reference to the label. The assembler goes back and corrects their binary representation to incorporate the address of the label. Backpatching speeds assembly because the assembler only reads its input once. However, it requires an assembler to hold the entire binary representation of a program in memory so instructions can be backpatched. This requirement can limit the size of programs that can be assembled. The process is complicated by machines with several types of branches that span different ranges of instructions. When the assembler first sees an unresolved label in a branch instruction, it must either use the largest possible branch or risk having to go back and readjust many instructions to make room for a larger branch.

backpatching A method for translating from assembly language to machine instructions in which the assembler builds a (possibly incomplete) binary representation of every instruction in one pass over a program and then returns to fill in previously undefined labels.

Object File Format

Assemblers produce object files. An object file on UNIX contains six distinct sections (see Figure A.2.1):

- The *object file header* describes the size and position of the other pieces of the file.
- The **text segment** contains the machine language code for routines in the source file. These routines may be unexecutable because of unresolved references.
- The **data segment** contains a binary representation of the data in the source file. The data also may be incomplete because of unresolved references to labels in other files.
- The **relocation information** identifies instructions and data words that depend on **absolute addresses**. These references must change if portions of the program are moved in memory.
- The *symbol table* associates addresses with external labels in the source file and lists unresolved references.
- The *debugging information* contains a concise description of the way in which the program was compiled, so a debugger can find which instruction addresses correspond to lines in a source file and print the data structures in readable form.

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

data segment The segment of a UNIX object or executable file that contains a binary representation of the initialized data used by the program.

relocation information The segment of a UNIX object file that identifies instructions and data words that depend on absolute addresses.

absolute address A variable's or routine's actual address in memory.

The assembler produces an object file that contains a binary representation of the program and data and additional information to help link pieces of a pro-

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information

FIGURE A.2.1 Object file. A UNIX assembler produces an object file with six distinct sections.

gram. This relocation information is necessary because the assembler does not know which memory locations a procedure or piece of data will occupy after it is linked with the rest of the program. Procedures and data from a file are stored in a contiguous piece of memory, but the assembler does not know where this memory will be located. The assembler also passes some symbol table entries to the linker. In particular, the assembler must record which external symbols are defined in a file and what unresolved references occur in a file.

Elaboration: For convenience, assemblers assume each file starts at the same address (for example, location 0) with the expectation that the linker will *relocate* the code and data when they are assigned locations in memory. The assembler produces *relocation information*, which contains an entry describing each instruction or data word in the file that references an absolute address. On MIPS, only the subroutine call, load, and store instructions reference absolute addresses. Instructions that use PC-relative addressing, such as branches, need not be relocated.

Additional Facilities

Assemblers provide a variety of convenience features that help make assembler programs short and easier to write, but do not fundamentally change assembly language. For example, *data layout directives* allow a programmer to describe data in a more concise and natural manner than its binary representation.

In Figure A.1.4, the directive

```
.asciiz “The sum from 0 .. 100 is %d\n”
```

stores characters from the string in memory. Contrast this line with the alternative of writing each character as its ASCII value (Figure 2.21 in Chapter 2 describes the ASCII encoding for characters):

```
.byte 84, 104, 101, 32, 115, 117, 109, 32
.byte 102, 114, 111, 109, 32, 48, 32, 46
.byte 46, 32, 49, 48, 48, 32, 105, 115
.byte 32, 37, 100, 10, 0
```

The `.asciiz` directive is easier to read because it represents characters as letters, not binary numbers. An assembler can translate characters to their binary representation much faster and more accurately than a human. Data layout directives

specify data in a human-readable form that the assembler translates to binary. Other layout directives are described in Section A.10 on page A-45.

String Directive

EXAMPLE

Define the sequence of bytes produced by this directive:

```
.asciiz "The quick brown fox jumps over the lazy dog"
```

ANSWER

```
.byte 84,  104, 101, 32,  113, 117, 105, 99
.byte 107, 32,  98,  114, 111, 119, 110, 32
.byte 102, 111, 120, 32,  106, 117, 109, 112
.byte 115, 32,  111, 118, 101, 114, 32,  116
.byte 104, 101, 32,  108, 97,  122, 121, 32
.byte 100, 111, 103, 0
```

Macros are a pattern-matching and replacement facility that provide a simple mechanism to name a frequently used sequence of instructions. Instead of repeatedly typing the same instructions every time they are used, a programmer invokes the macro and the assembler replaces the macro call with the corresponding sequence of instructions. Macros, like subroutines, permit a programmer to create and name a new abstraction for a common operation. Unlike subroutines, however, macros do not cause a subroutine call and return when the program runs since a macro call is replaced by the macro's body when the program is assembled. After this replacement, the resulting assembly is indistinguishable from the equivalent program written without macros.

Macros

EXAMPLE

As an example, suppose that a programmer needs to print many numbers. The library routine `printf` accepts a format string and one or more values to print as its arguments. A programmer could print the integer in register `$7` with the following instructions:

```
         .data
int_str: .asciiz"%d"
         .text
         la    $a0, int_str # Load string address
                            # into first arg
```

```
        mov   $a1, $7  # Load value into
                       # second arg
        jal   printf   # Call the printf routine
```

The `.data` directive tells the assembler to store the string in the program's data segment, and the `.text` directive tells the assembler to store the instructions in its text segment.

However, printing many numbers in this fashion is tedious and produces a verbose program that is difficult to understand. An alternative is to introduce a macro, `print_int`, to print an integer:

```
        .data
int_str:.asciiz "%d"
        .text
        .macro print_int($arg)
        la $a0, int_str  # Load string address into
                         # first arg
        mov $a1, $arg    # Load macro's parameter
                         # ($arg) into second arg
        jal printf       # Call the printf routine
        .end_macro
print_int($7)
```

formal parameter A variable that is the argument to a procedure or macro; replaced by that argument once the macro is expanded.

The macro has a **formal parameter**, `$arg`, that names the argument to the macro. When the macro is expanded, the argument from a call is substituted for the formal parameter throughout the macro's body. Then the assembler replaces the call with the macro's newly expanded body. In the first call on `print_int`, the argument is `$7`, so the macro expands to the code

```
la  $a0, int_str
mov $a1, $7
jal printf
```

In a second call on `print_int`, say, `print_int($t0)`, the argument is `$t0`, so the macro expands to

```
la  $a0, int_str
mov $a1, $t0
jal printf
```

What does the call `print_int($a0)` expand to?

ANSWER

```
la  $a0, int_str
mov $a1, $a0
jal printf
```

This example illustrates a drawback of macros. A programmer who uses this macro must be aware that `print_int` uses register `$a0` and so cannot correctly print the value in that register.

Hardware Software Interface

Some assemblers also implement *pseudoinstructions*, which are instructions provided by an assembler but not implemented in hardware. Chapter 2 contains many examples of how the MIPS assembler synthesizes pseudoinstructions and addressing modes from the spartan MIPS hardware instruction set. For example, Section 2.6 in Chapter 2 describes how the assembler synthesizes the `blt` instruction from two other instructions: `slt` and `bne`. By extending the instruction set, the MIPS assembler makes assembly language programming easier without complicating the hardware. Many pseudoinstructions could also be simulated with macros, but the MIPS assembler can generate better code for these instructions because it can use a dedicated register (`$at`) and is able to optimize the generated code.

Elaboration: Assemblers *conditionally assemble* pieces of code, which permits a programmer to include or exclude groups of instructions when a program is assembled. This feature is particularly useful when several versions of a program differ by a small amount. Rather than keep these programs in separate files—which greatly complicates fixing bugs in the common code—programmers typically merge the versions into a single file. Code particular to one version is conditionally assembled, so it can be excluded when other versions of the program are assembled.

If macros and conditional assembly are useful, why do assemblers for UNIX systems rarely, if ever, provide them? One reason is that most programmers on these systems write programs in higher-level languages like C. Most of the assembly code is produced by compilers, which find it more convenient to repeat code rather than define macros. Another reason is that other tools on UNIX—such as `cpp`, the C preprocessor, or `m4`, a general macro processor—can provide macros and conditional assembly for assembly language programs.