Chapter 2[1]

SAL -- Simple Abstract Language

A programming language provides a method for the programmer to describe precisely the data structures and the algorithms to be performed on those data structures. This chapter introduces a powerful assembly language, SAL, that allows the creation of useful programs. Its power comes from allowing a high level of abstraction. It is similar to high-level languages such as Pascal or C in the amount of work that is accomplished with individual instructions. Its syntax is similar to a traditional assembly language. The capabilities of SAL are demonstrated by comparing examples in Pascal and in SAL.

Any programming language must provide ways of specifying four types of operations. First, the language must provide a way to specify the **type** of a variable. This implies the range of values the variable can be assigned and the ways it can be used. Some languages, such as FORTRAN, allow implicit declaration depending on the name chosen for the variable. Second, the language must have a way of specifying arithmetic operations such as addition and multiplication. Third, the language must provide control structures that allow looping and conditional execution. Fourth, a programming language must provide a way to communicate with the user of the program created in the programming language.

This chapter discusses aspects of assembly language programming, such as when and why assembly language code is written. It then focuses on the details of the SAL programming language. Most programming examples are given in both Pascal and in SAL. Each of the four necessary operation types are discussed in turn. At the end of the chapter is a description of a procedure call and return mechanism, followed by a complete program.

2.1 On Assembly and Compilation

A goal of programming language design is to provide an environment to maximize the efficiency of the programmer. The structure of the programming language should make it easy to write programs correctly and quickly. The programming language should also foster programs that make it easy for a programmer unfamiliar with a program to read and understand how it works for the purpose of modifying it. In addition to assisting in the programming process, however, a programming language should be designed so that it can be executed efficiently on a computer. Programs should make the best possible use of the hardware so that they execute as rapidly as possible, using as few resources as possible.

Unfortunately, these two goals — programmer efficiency and hardware efficiency — are frequently incompatible. Often, an unsophisticated algorithm is easily written and easily understood, but slow to execute. A more obscure algorithm might use resources more efficiently or take advantage of certain features of the computer that make the algorithm run efficiently. For example, a program that uses a temporary variable called `temp` to save many different intermediate results may be difficult to understand. If the variable is **overloaded** by using it to take on the value of different variables during the course of a single program, it may be difficult to analyze the program or identify a programming error. It is more easily understood when different variables are given independent variable names. Yet declaring numerous variables when one would suffice wastes of computing resources. Memory space is not used efficiently much of the time since most of the variables are not defined. CPU resources are not used efficiently if this data is loaded and stored many times.

Why Write Assembly Language Programs?

Years ago, hardware efficiency was extracted at the expense of the programmer's time. If a fast program was needed, then it was written in assembly language. Compilers were capable of translating programs from high-level languages, but generated assembly language programs that were relatively inefficient. Programmers often found it necessary to optimize the assembly language code created by the compiler for two reasons. The first is that memory space was often quite limited. A programmer could write code that fit in the available space where a compiler could not. The second reason that assembly code was written was to achieve acceptable performance. A programmer could write code that executed faster than the code generated by compilers.

This is no longer the case. Compilers have improved to the point that they can generate code comparable, or better than the code most programmers can generate. There are two main reasons why the use of compiler-generated code has become common. Advances in compiler technology have greatly improved the quality of the assembly language code generated. Writing in assembly language may result in little or no improvement over the best code a compiler can generate. In many cases it is hard to find ways to improve the code generated by a high quality compiler. The second reason is that there is little benefit derived by improving the execution speed of the assembly language. Many computers today execute so rapidly that it is not necessary to optimize code at the assembly language level.

It has become increasingly rare that programmers find it necessary to write assembly language code. However, there are several special reasons why it might be necessary. First, there are features of the computer that can be accessed with assembly language that are not well captured in a high-level language. Programs that must use those features may need to be written, at least partially, in assembly language. Critical parts of an operating system are an example of code that is often written in assembly language for this reason. Second, some programs have critical constraints, for example, a program that must fit in a very small amount of memory. Another example is a program that must execute in a highly predictable amount of time. Sometimes the reason for writing in assembly language is simply the unavailability of a good compiler. This last reason should become increasingly rare as compiler technology becomes more widely established.

Compiler writers must understand how to write programs in assembly language before they can write compilers. For a compiler to produce efficient code, a compiler writer must be able to assess the costs and benefits of various code implementations. There are often several ways to implement the same code, and the best way often depends on details that are specific to the implementation of the targeted computer. These details vary from machine to machine.

Where SAL Fits In

SAL is similar to the intermediate language that a compiler might generate. It is not difficult to translate high-level language code into SAL, and it is straightforward to translate SAL code into MAL (or even TAL) code. SAL therefore provides a good starting point for the introduction of computer architecture for a programmer who knows a high level language.

In general, this book presents the simplest, most obvious sequence of instructions. As is typical of modern high-level languages, ease of understanding is emphasized over efficiency of the program. This is consistent with the way compilers generate efficient assembly language code. Using a more-or-less direct translation, a compiler initially creates a program in an intermediate language that is often an abstraction of the assembly language of the targeted computer. Then the compiler invokes a program, known as an **optimizer** to improve the speed of the program without changing its behavior. Either during the optimizations or afterward, the assembly language of the abstract computer is translated into the assembly language of the target computer, then translated into machine code.

The MIPS RISC assembler is somewhat unusual in that the language it accepts (MAL) is not the true assembly language (TAL) of the hardware. This is because the MIPS RISC assembler performs additional optimizations before it generates the machine language code. Nevertheless, the process of writing straightforward assembly language programs that can be translated into MAL is a realistic way to write programs for a MIPS RISC-based computer.

2.2 Variable Declaration

Like all high-level languages, C provides a means for declaring the type of a variable. Declaration is for the benefit of the compiler or assembler, which must know (among other things) how much space to allocate for specific variables. Different variable types can take on different values and require different amounts of space. It is important that sufficient space, but not more, be set aside.

SAL understands three simple types: integers, characters, and floating point. The declaration of a variable is accomplished by giving a variable a name and a type. Integers are declared using the following syntax. An integer declaration in C is

```
int variablename {= value};
```

The SAL declaration of an integer variable looks like

```
{ variablename: } .word { value }
```

For the definitions, keywords are indicated in **`boldface`**. Optional words are in braces ({}). When an identifier is used to give a name to a variable, such as `variablename`, it is called a **label**. In both C and SAL, identifiers follow the rule that they start with a letter, and can be followed by letters or digits. The colon (:) marks the end of the variable name. The **`.word`** identifies the variable as an integer. It indicates how much space must be provided for the variable, whereas the label indicates how the variable is to be referenced. If `value` is present, it represents an integer constant. The value will be assigned to the variable as an initial value. If there is no integer value in the variable declaration, then the value of the variable will be initialized to zero.

The declaration

```
ten:     .word 10
```

sets aside space for a variable named `ten`, and initializes its value to be 10. The declaration

```
counter: .word
```

sets aside space for a variable named `counter`, and initializes its value to be 0. Both `ten` and `counter` are type integer. Notice that the value of a SAL variable is *always* defined.

The SAL character type declaration is similar to the SAL integer type declaration.The SAL declaration of a character type variable looks like

```
{variablename:} .byte {value}
```

The word **.byte** identifies the variable as a character. The label specifies the variable name. Like integer declarations, the value portion of the declaration is optional. If present, the syntax of `value` is that of a single character enclosed in single quote marks.

The declaration

```
sentinel: .byte 'z'
```

identifies the variable named `sentinel` to be a character, and initializes its value to be the character z. A declaration without a value portion will set aside enough space for a single character, bind the variable's name to the space, and assign the null character as a value.

Other characters such as the linefeed (newline) character are specified using the same escape sequences as in C. The linefeed character in SAL can be declared:

```
.linefeed: .byte '\n'
```

Real (noninteger) variables are declared in the same format as the other types. A C declaration of a variable of type floating point is

```
float variablename { = value };
```

The SAL declaration of a type real variable looks like this:

```
{variablename:} .float {value}
```

The .float identifies the variable as a real number. The variable's name is given by `variablename` and `value` is optional. If present, the variable is initialized with the value given. Otherwise, the variable is assigned the value 0. The `value` is given by the following syntax. A floating point value contains an optional sign (`+` or `-`) and a set of digits that may contain a decimal point, and may be followed by an exponent specification. The exponent specification is the letter `E` or `e` followed by an optional sign and an integer. The following examples are all legal floating point values, and they all specify the same value.

```
136.42
1.3642E2
+13.642e1
0.13642e+3
13642.e-2
```

Declarations are information given to the assembler about how to *create* the program, not how to execute it. They are therefore set apart within a program in a section that specifies how memory is to be allocated. The memory is divided into two distinct areas, one for instructions, known as the **code** or **text** space, and one for variables, known as the **data** space. In SAL, declarations can occur anywhere, but they must be separated from code by the use of **directives** or **pseudo-instructions.** This is indicated by preceding one or more declarations by the pseudo-instruction `.data`, as in

```
.data
var1:    .word
var2:    .byte
```

Code is distinguished in SAL by preceding it with the pseudo-instruction `.text`. There may be multiple `.data` and `.text` sections in a program.

2.3 Arithmetic Operations

The assignment statement in Pascal involves the evaluation of expressions composed of operators, variables, and constants. In Pascal, as in most languages, all operators are either monadic or dyadic. Addition and multiplication are not inherently dyadic operations, but subtraction and division are inherently dyadic operations. The longhand methods for performing addition and multiplication are dyadic, however, so it generally seems natural to make this restriction. High-level languages such as C and Pascal go to great lengths to define how to evaluate an expression by defining the order in which the operators are applied. Thus the C statement

```
answer = a - b + c;
```

is defined precisely to be

```
answer = ( a - b ) + c;
```

and not

```
answer = a - ( b + c );
```

In fact, the evaluation of a C statement involves a series of dyadic or monadic operations, performed on constants and variables in a well-defined order. SAL makes this order explicit by requiring that each operation be specified explicitly, and that the result be assigned to a variable.

Table 1 gives SAL's arithmetic instructions and C equivalents. An instruction consists of an

SAL Instructions	Equivalent C Statement
`move x, y`	`x = y;`
`add x, y, z`	`x = y + z;`
`sub x, y, z`	`x = y - z;`
`mul x, y, z`	`x = y * z;`
`div x, y, z`	`x = y / z;`
`rem x, y, z`	`x = y % z;`

Table 1: Arithmetic operations in SAL.

operation specification, known as the **mnemonic** or **opcode** and two or three operand specifications. An operand is either (1) the name of a variable or (2) a constant. For example, consider the C statement

```
int area_triangle, width, height;
...
area_triangle = ( width * height ) / 2;
```

This statement could be translated into the following SAL code:

```
area_triangle:    .word
width:            .word
height:           .word
tmp:              .word
...
mul  tmp, width, height
div  area_triangle, tmp, 2
```

For all arithmetic instructions, the first operand specifies the destination of the result, and the following operands are sources. The `move` instruction is equivalent to a simple C assignment statement. The value assigned to the first (destination) operand variable given in the `move` instruction is the value of the second (source) variable. The value of the source variable is unchanged by the move instruction. The C assignment statement

```
A := B;
```

could be translated to the SAL instruction

```
move    A, B
```

The `add`, `sub`, and `mul` instructions perform the operations that are specified in C by the operators +, -, and *, respectively. Instructions equivalent to these operators are defined in SAL where the operands are either integers or real numbers. Operand types should not be mixed in one instruction. Integer division is specified by the `div` instruction, and the modulus (or remainder) function is specified by the `rem` instruction. For integer variables, the `div` corresponds to integer division operator in C, and `rem` corresponds to the C `%` function. In addition, `div` can be applied to real variables to obtain the floating point quotient.

A Simple SAL Program

Figure 1 contains the code for an exceptionally simple program that finds the average of the

```
/* a simple C program to average 3 integers */
#include <stdio.h>
main()
{
      int avg;
      int i1 = 20;
      int i2 = 13;
      int i3 = 82;

      avg = (i1 + i2 + i3) / 3;
}
```

```
# a simple SAL program to average 3 integers

      .data
avg:  .word              # integer average
i1:   .word 20           # first number in the average
i2:   .word 13           # second number in the average
i3:   .word 82           # third number in the average

      .text
__start:    add   avg, i1, i2
            add   avg, avg, i3
            div   avg, avg, 3
            done
```

Figure 1: C and SAL versions of a program that averages three integers.

three integers, contained in the variables `i1`, `i2`, and `i3`. This example illustrates several important parts of a program not yet specified. The program is shown in both C and in SAL.

So far, labels have been used only to identify variable names. Labels can also be used to identify any instruction or variable declaration. When the program is assembled, the assembler allocates storage space for both program instructions and data. Each label must be unique. When a label is attached to an instruction or to data, the assembler associates a memory location with the label. The sample program has the label `__start` attached to the first instruction in the program. All SAL programs must have the label `__start` to identify where execution of the program begins. It usually is the first instruction in the program, but need not be.

SAL programs can be documented by adding comments. A comment in C is marked by surrounding it with the character strings /* and */. SAL comments are formed on a line-by-line basis. Within any line of a program, anything that follows a # symbol is considered to be a comment. Therefore, a comment may appear on the same line as an instruction or declaration by placing a # character between the end of the instruction or declaration and the comment itself. A comment may also appear by itself on a line that begins with the # character. Comments may not span lines.

Because the variable `avg` is declared to be an integer, the instruction `div` only gives the integer portion of the average. The remainder is lost. For example, if the variables were declared as `i1 = 10`, `i2 = 5` and `i3 = 5`, the result in `avg` would be 6, not $6\frac{2}{3}$. If the variable `avg` were declared to be a real (.float), the value would be extremely close to, but not exactly, $6\frac{2}{3}$.

The end of a program is indicated by the single word `done`. The word `done` is not a directive; it is a macro. Chapter 10 explains macros. The end of a program must be marked so that the computer understands that the program has been completed to allow another program to be run. A SAL program may have more than one `done`, but it must have at least one.

2.4 Control Structures

The assembly language instructions presented so far are not sufficient to form a usable programming language. C provides two categories of **structured statements** or **control structures:** conditionals and iteratives. An example of the first category is an **if statement**. It provides the capability for conditionally executing a statement. If the condition in the if statement evaluates to true, then the statement is executed. Otherwise it is skipped. Here is a C if statement.

```
if (a < b)
        c = a + b;
```

When the if statement is executed, the first thing that occurs is a comparison. The value of `a` is compared against the value of `b`. If `a` is indeed less than `b`, then the conditional evaluates to `true`, and the statement associated with the if statement is executed. In this case, the sum of `a` and `b` is calculated and assigned to variable `c`. If the conditional evaluated to false, then the statement is not executed. It is skipped. This is conditional execution; depending on the value of the condition, a statement may or may not be skipped.

An example of the second category is a **repetitive statement** which is used to implement a **loop.** C examples of repetitive statements are `for`, `while`, and `do-while` loops. Both of these categories of statements are made possible by a single assembly language construct called a **branch**. The simplest branch instruction is the equivalent of the C `goto` statement, which branches to a label.

More complex branch instructions combine conditional execution with a `goto` statement. This powerful set of instructions is the only mechanism provided in SAL to enable looping constructs. While this limitation may seem restrictive initially, there are very good reasons for it, since this restriction closely reflects the underlying hardware restrictions. The use of `goto` is generally discouraged in high-level languages because it makes programs difficult to analyze and debug. If the compiler is implemented correctly, however, and the high-level language program is well-structured, the use of branch instructions at the assembly language level introduces no new concerns.

Table 2 summarizes SAL's branch instructions. The variables `x` and `y` may be of type integer

SAL instructions	Equivalent C Statement
`b label`	`goto label;`
`beq x, y, label`	`if (x == y) goto label;`
`bne x, y, label`	`if (x != y) goto label;`
`blt x, y, label`	`if (x < y) goto label;`
`bgt x, y, label`	`if (x > y) goto label;`
`ble x, y, label`	`if (x <= y) goto label;`
`bge x, y, label`	`if (x >= y) goto label;`
`bltz x, label`	`if (x < 0) goto label;`
`bgtz x, label`	`if (x > 0) goto label;`
`blez x, label`	`if (x <= 0) goto label;`
`bgez x, label`	`if (x >= 0) goto label;`
`beqz x, label`	`if (x == 0) goto label;`
`bnez x, label`	`if (x != 0) goto label;`

Table 2: Branch instructions in SAL.

or character, and they can be constants or variables. Note that many of the branch instructions are redundant. In fact, the instructions in the latter half of the table are simply special cases of those instructions in the first half of the table, where the second operand is implicitly zero. Thus the instruction

```
ble sum, 0, L1
```

is equivalent to the instruction

```
blez sum, L1
```

These instructions are included because tests against zero are so common that many computers are optimized to handle them efficiently. Also note that an unconditional branch can be constructed from a special case of a conditional branch. As an example, the SAL instruction

```
b next
```

is equivalent to

```
beqz 0, next
```

The SAL instruction

```
ble x, y, L1
```

is also equivalent to

```
blt x, y, L1    # Branch if x < y.
beq x, y, L1    # Branch if x = y.
```

This alternative requires two instructions to be executed and is therefore a less attractive alternative.

Figure 2 shows a C `if-then-else` statement and two possible assembly language equiva-

C statement

```
if (A > 0)
    B = C / A;
else
    B = A + 10;
```

Possible SAL equivalent

```
            blez  A, elsepart
            div   B, C, A
            b     endif
elsepart:   add   B, A, 10
endif:
```

Another possible SAL equivalent

```
            bgtz    A, ifpart
            add     B, A, 10
            b       endif
ifpart:     div     B, C, A
endif:
```

Figure 2: SAL code implementing the C `if-then-else` statement.

lents. The statement tests if `A` is positive. If `A` is positive, it assigns to `B` the value of `C/A`. .Otherwise, it assigns `B` the value `A + 10`. All three code fragments implement the same function. Note that the first SAL equivalent reverses the sense of the comparison, and the second reverses the order of the `if` and `else` statements.

Two versions of SAL code are given to illustrate a point. There are numerous ways to program any given high-level language control structure. Based on the specific program, code written one way might execute more efficiently than code written another. This fact can be used to advantage by a sophisticated compiler or assembly language programmer.

A compound conditional can be built out of multiple branch instructions. Figure 3 shows an

C statement

```
if ( (A = B) || (C < D) ) {
      A = A + 1;
      B = B - 1;
      D = A + C;
}
```

SAL equivalent

```
            beq   A, B, do_if
            blt   C, D, do_if
            b     end_if
do_if:      add   A, A, 1
            add   B, B, -1
            add   D, A, C
end_if:
```

Figure 3: SAL code implementing a C compound conditional.

example of a C compound conditional statement. One of the two conditions must evaluate to `true` if the statements within the `if` statement are to be executed. The SAL code uses three branch statements to implement the structure of the compound conditional. If `A` is not equal to `B`, then the `beq` branch is not taken, and the second instruction (`blt`) is executed. If `A` and `B` are equal, then the branch is taken to the code within the `if` statement. If both conditionals turn out to be false, then the unconditional branch instruction, `b`, modifies the PC such that it contains the address `endif`.

A second example of a compound conditional is given in Figure 4. It shows an example of a

C statement

```
if ( (A == B) && (C == D) || (E < 0) ) {
      A = A + 1;
      C = E;
}
```

SAL equivalent

```
              bne   A, B, check_E
              beq   C, D, do_if
check_E:      bgez  E, end_if
do_if:        add   A, A, 1
              move  C, E
end_if:
```

Figure 4: SAL code implementing a C compound conditional.

logical and together with a logical or. In C, the evaluation of the `and` is completed before the `or`. The equivalent SAL code to implement the `if` statement reverses some of the conditions. This reversal has the effect of reducing the number of instructions necessary to implement the complete test.

An equivalent to a C `while` loop is straightforward to build out of SAL instructions. Figure 5

C statement

```
result = 1;
counter = exponent;
while (counter > 0) {
        result = result * base;
        counter = counter - 1;
}
```

SAL equivalent

```
            move  result, 1
            move  counter, exponent
while:      blez  counter, endwhile
            mul   result, result, base
            sub   counter, counter, 1
            b     while
endwhile:
```

Figure 5: SAL code to caculate $\text{base}^{\text{exponent}}$ using a while loop.

contains both a C version and a SAL version of a while loop that implements a power function. It calculates $\text{base}^{\text{exponent}}$, where `exponent` is assumed to be a positive integer. The result is assigned to the variable `result`. Note that the variables `base` and `exponent` are not changed by the execution of the loop, like the C implementation.

A C `for` loop can also be formed from SAL instructions. Figure 6 contains a C `for` loop and

C statement

```
      result = 1;
      for (counter = 1; counter <= exponent; counter++) {
            result = result * base;
      }
```

SAL equivalent

```
            move    result, 1    # initialize result
            move    counter, 1   # initialize loop induction variable
            # exit loop when counter > exponent
      for:  bgt     counter, exponent, endfor
            mul     result, result, base
            # increment loop induction variable
            add     counter, counter, 1
            b       for
      endfor:
```

Figure 6: SAL code to calculate $base^{exponent}$ using a `for` loop.

a SAL translation of the loop. Before the loop is entered, the loop induction variable `counter` is initialized to 1. At the top of the loop is a test to see if the loop induction variable is greater than the given ending value (`exponent`). If it is greater, the branch is taken, and the loop is exited. This is done by a conditional branch instruction in the SAL code. The last statement in the for loop is an unconditional branch back to the top of the loop. Before branching back to the top, the loop induction variable is incremented by 1. Notice that although the high-level language looping construct can define that a `for` loop implicitly increments the loop induction variable, SAL does not. A SAL equivalent must explicitly contain an instruction to add one to the loop induction variable. Incrementing an induction variable is such a common operation that some assembly languages provide a mechanism for implicitly incrementing a variable, just as high-level languages do.

2.5 Communication with the User

The final necessary item for an assembly language is some form of communication with the user. The communication is between the computer and the user of the program. For simplicity, assume that all communication from the user comes from a keyboard. All communication from the computer to the user goes to a display (or screen).

Table 3 contains SAL communication instructions. The only **input** instruction is `get`. It

SAL instructions	Equivalent C Statement	Notes
`get x`	`scanf("%d\n", &x);` `x = getc(stdin);`	`x` is type `.word` `x` is type `.byte`

`put x`	`printf("%d", x);`
	`printf("%c", x);`
	`printf("%f", x);`
		`x` is type `.word`
		`x` is type `.byte`
		`x` is type `.float`

`getc x`	`x = getc(stdin);`	
`putc x`	`printf("%c", x);`	
`puts string`	`printf("%s", string);`	

Table 3: SAL communication instructions.

reads some amount of data from a keyboard, and places the data in the variable specified as an operand. There are two **output** instructions, `put` and `puts`. Each displays the data specified by the operand variable.

The output operation `puts` takes a special form of string, and prints it to the screen. The string is essentially an array of characters, and the final character of the string is the null character '`\0`'. A string ended this way is often called a **null-terminated string**.

A simple way to declare a string that is automatically null terminated is by using a directive. The `.asciiz` directive allows a string to be specified, and null terminates the string. Consider the directive

```
string1: .asciiz "howdy!\n"
```

This directive declares a string of 8 characters, and labels it `string1`. The first 7 characters are assigned to be the characters in the string, and the final character is the null character. When declared using the `.asciiz` directive, the string is printed out to a display by using the single instruction

```
puts string1
```

The `puts` instruction is a powerful instruction for displaying messages, but it is in fact a simple procedure that calls `put` repeatedly. Here is the SAL code to write the message `howdy!`, followed by the newline character, using only `put` instructions.

```
put 'h'
put 'o'
put 'w'
put 'd'
put 'y'
put '!'
put '\n'
```

The structure of the input and output instructions is similar to than that of C. The `get` instruction works on a line-by-line basis for variables of type integer and floating point, making it like the C statement

```
void scanf("%d\n", &user_int);
```

or

```
void scanf("%f\n", &user_float);
```

Even if there is more than one value on a line, a `get` instruction will read the first value and throw away the rest. When the first value in the input read does not match the type, the value zero is placed in the operand.

When the operand of a `get` instruction is a character (declared as `.byte`) the SAL get instruction is equivalent to the C getc statement. No characters in the input are thrown away. The SAL instruction

```
get    user_char
```

is equivalent to the C `getc` statement

```
user_char = getc(stdin);
```

SAL also contains a `getc` instruction. It works exactly the same as the SAL instruction `get` where the operand is of type character (declared in SAL as `.byte`).

The SAL `put` instruction does not work on a line-by-line basis. It displays the operand in a format appropriate to the type of its operand. The C `printf` statement accomplishes the same operation as the SAL `put` instruction. The SAL instruction

```
put variable
```

has different output depending on the type of variable. If the operand called `variable` were of type character (declared in SAL as `.byte`), the equivalent C statement is

```
printf("%c", &variable);
```

The SAL `putc` instruction is identical in function to the SAL `put` instruction where the operand is of type character (declared in SAL as `.byte`).

If the variable in the SAL `put` instruction were declared of type integer (declared in SAL as `.word`), the equivalent C statement is

```
printf("%d", &variable);
```

If the variable were declared of type floating point (declared in SAL as `.float`), the equivalent C statement is

```
printf("%f", &variable);
```

As in C, in order to inject a new line into the output, the newline character, ‘\n’ is explicitly printed. Printing out this character forces the cursor to move to the beginning of the next line.

Figure 7 gives both C and SAL code (not a complete program) that reads characters typed on

C code

```
while ( (ch = getc(stdin)) != 'Z' );
printf("\nZ encountered\n");
```

SAL equivalent

```
.data
message:    .asciiz "\nZ encountered\n"
.text
loop:       get   ch
            bne   ch, 'Z', loop
            puts  message
            done
```

Figure 7: C and SAL code to read characters until the character 'Z' is encountered.

the keyboard until the character 'Z' is encountered. It then prints out the message

```
Z encountered
```

and quits.

2.6 A SAL Program

Figure 8 and Figure 9 contain a simple, complete program that prints out for the user the sum

```
#include <stdio.h>
main()
{
      int n; /* user entered integer */
      int sum; /* running sum of the first n integers */
      int i; /* integer to be added into sum, from 0 to n */

      /* prompt for input */
      printf("Please enter a positive integer: ");
      void scanf("%d\n", &n);
      printf("\n");

      /* calculate the sum */
      sum = 0;
      for (i=0; i<=n; i++)
            sum = sum + i;
      printf("The sum of the first %d integers is %d\n", n, sum);
}
```

Figure 8: C program that sums the first n positive integers.

```
# a SAL program to add up the first n integers,
# where n is a positive integer entered by the user.

 .data
# strings for making the output look nice
str1:    .asciiz "Please enter a positive integer:  "
str2:    .asciiz "The sum of the first "
str3:    .asciiz " integers is "
newline: .byte '\n'
# variable declarations
n:       .word   0      # user entered integer
sum:     .word   0      # running sum of the first n integers
i:       .word   0      # integer to be added into sum,
                        #   runs from 0 to n
tmp:     .word          # used for comparisons of i and n

.text
__start:    puts    str1            # prompt for input
            get     n
            put     newline

for:        sub     tmp, n, i       # for i:= 0 to n do
            bltz    tmp, endfor     #     sum := sum + i;
            add     sum, sum, i
            add     i, i, 1
            b       for
endfor:     puts    str2 # print the sum in nice form
            put     n
            puts    str3
            put     sum
            put     newline
            done
```

Figure 9: SAL program that sums the first n positive integers.

of the first n positive integers, where n is a positive integer that is input by the user. Figure 9 contains a SAL version of the C program given in Figure 8. While the program does exactly what is stated, it has one major drawback. There is no error checking on the user's input. If the user enters something other than an integer, the program may either crash, or it may calculate and print out an unexpected result.

2.7 Procedures and Functions

Any programmer who undertakes the writing of a large program understands the need for program modularization. Procedures and functions provide a useful abstraction. A mechanism to facilitate function calls and returns is often provided in an assembly language.

The Parts of a Procedure

The various parts of a function and function call are identified in the following C code. This program fragment contains a function call and the function. Function `switch` is a trivial function that switches the values pointed to by its parameters.

```
main()
{
      .
      .
      switch(&a, &b);
      c = a + 1;
      .
      .
}

void switch(x, y)
int *x;
int *y;
{
      int temp;

      temp = *x;
      *x = *y;
      *y = temp;
}
```

In order to gain insight into the implementation of a function in assembly language, it is useful to go over the steps involved in the execution of a function. Four steps are required to execute a simplified function. The function execution to be discussed is one that passes no parameters, and is not recursive. Here are the four steps in the execution.

1. Save return address
2. Procedure call
3. Execute procedure
4. Return

Step 2, the function call, is really a branch instruction. The control of the program must be transferred to the first instruction within the function. Once the function's code has been executed, control must be transferred back to the instruction following the function call. This return might be accomplished by using another branch instruction as follows.

```
.text
      .
      .
      .
call:       b proc
rtnaddr:
      .
      .
      .
            done

proc:       # procedure’s code here
      .
      .
      .
            b rtnaddr
```

The problem with this scheme for calls and returns becomes apparent when multiple call locations are considered. One of the important features of a procedure or function is that it can be called multiple times, from various locations in a program. The use of an explicit branch back to the address following the call does not work if there is more than one call location. This is because there can be only one label identifying the return location.

Addresses

The solution to this problem requires that the program remember a return address. The address remembered is different for each call location. SAL provides an instruction that places an address into a variable.

```
la  saved_address, rtnaddr
```

The `la` (load address) instruction assigns the value of the label in its second variable into the location given by the first variable. The address corresponding to the label `rtnaddr` is placed into the variable labelled `saved_address`. Variable `saved_address` must be an integer type variable. It could be declared as

```
.data
saved_address:    .word
```

A comparison of the SAL `move` and `la` instructions highlights the function of the `la` instruction. Assume that addresses can be represented by integers. Let the value of variable `x` be placed at the integer address 3. The value of the variable `y` will be at address 5.

label	address	contents
x	3	25
y	5	7

Consider the result of executing the instructions

```
move x, y
```

and

```
la  x, y
```

The result of the move instruction will be to copy the value of `y` into the variable `x`. So `x` would contain the value 7 after execution of the move instruction. The result of the `la` instruction will be to copy `y` into the variable `x`. The label `y` is the address 5 in this example. So `x` would contain the value 5 after execution of the `la` instruction.

Remembering Return Addresses

The solution to the problem of multiple calls to a function or procedure is to save a return address before a function is called, and then use the saved value when it is time to return from the function. A return address is saved in a variable associated with a function. Before the branch to a function's first instruction, the correct return address is copied into that variable. A function call example is the following:

```
            la  proc1_ret, ret_addr1
            b  proc1
ret_addr1:
```

A second call to the same function is the same, except for the different return address label:

```
            la  proc1_ret, ret_addr2
            b  proc1
ret_addr2:
```

Return Mechanism

The final piece of a function call and return mechanism is the return. As given above, one variable will now be associated with each function. That variable will contain the address of the next instruction to be executed when the function is done. But, the following branch instruction will *not* work as a return.

```
b  proc1_ret
```

This branch instruction would cause the program to branch to a variable. `proc1_ret` is the label of a variable, not an instruction. What is desired is to branch to the address contained within the variable `proc1_ret`. An extension to the functionality of the unconditional branch instruction will have the desired effect. The parentheses around the variable in the instruction

```
b  (proc1_ret)
```

have the effect of branching back to the correct location. The contents of variable `proc1_ret` are used instead of the address itself. Only the unconditional branch instruction can use this syntax of parentheses to branch to the address contained within a variable.

2.8 A Modular SAL Program

A program made modular by the use of procedures is given in Figures 2.10 and 2.11. Figure

```
/* A C program to calculate the longest, shortest, */
/* and average length of strings entered by the user.   */

#include <stdio.h>

main()
{
int str_count:  integer; /*number of user entered strings*/
int sum; /* running sum of the string lengths */
int ave; /* average of the string lengths */
int str_length; /* length of each string */
int shortest, longest;
char ch; /* used to read characters */

/* initialize variables */
      str_count = 0;
      sum = 0;
      ave = 0;
      shortest = 1000;
      longest = -1;

      getstring; /* prompt for input */
      while (str_length != 0) {
            calculate;
            getstring;
      }
      if (str_count > 0) {
            average;
            printresults;
      }
}

void getstring()
{
      str_length = 0;
      printf("Enter a string (<CR> to stop): ");
      while ( (ch = getc(stdin)) != '\n')
            str_length = str_length + 1;
}

void calculate()
{
      str_count = str_count++;
      sum = sum + str_length;
      if (str_length > longest)
            longest = str_length;
      if (str_length < shortest)
            shortest = str_length;
}
```

Figure 10: C program that calculates longest, shortest and average string lengths.

```
void average()
{
      ave = sum div str_count;
}

void printresults()
{
      printf("The longest string entered was %d characters long.\n",
longest);
      printf("The shortest string entered was %d characters long.\n",
shortest);
      printf("The average string length was %d characters.\n", ave);
}
```

Figure 10: C program that calculates longest, shortest and average string lengths.

2.10 contains a C implementation of the program, and Figure 2.11 contains a SAL implementa-

```
# A SAL program to calculate the longest, shortest,
# and average length of strings entered by the user.

.data
str_count:  .word 0      # number of user entered strings
sum:  .word 0            # running sum of the string lengths
ave:  .word 0            # average of the string lengths
str_length:  .word       # length of each string
shortest:    .word 1000
longest: .word -1
ch: .byte                #  used to read characters
newline: .byte '\n'
getstring_ra:.word    # return address for procedure getstring
calculate_ra:.word    # return address for procedure calculate
average_ra:.word    # return address for procedure average
printresults_ra:.word    # return address for procedure printresults
str1:.asciiz  "Enter a string (<CR> to stop):"
str2:.asciiz  "The longest string entered was "
str3:.asciiz  " characters long.\n"
str4:.asciiz  "The shortest string entered was "
str5:.asciiz  "The average string length was "
str6:.asciiz  " characters.\n"

.text
#  main program
__start:la  getstring_ra, rtn1
      b     getstring               # prompt for input
rtn1: beqz  str_length, endwhile     # while str_length<>0 do
      la    calculate_ra, rtn2       #
      b     calculate                #     calculate
rtn2: la    getstring_ra, rtn3       #
      b     getstring                #     getstring
rtn3: b     rtn1                    # endwhile
endwhile:blezstr_count, rtn5  # if str_count>0 then
      la    average_ra, rtn4         #
      b     average                  #     average
rtn4: la    printresults_ra, rtn5    #
      b     printresults             #     printresults
rtn5: done                          # endif
```

Figure 11: SAL program that calculates longest, shortest and average string lengths.

```
#  procedure getstring -- reads characters on 1 line until the
#                         newline character is encountered.  It
#                         also figures out the length of the string,
#                         not including the newline character.
getstring:  move str_length, 0
            puts  str1
            get   ch
while:      beq   ch, newline, getstr_rtn
            add   str_length, str_length, 1
            get   ch
            b     while
getstr_rtn: b     (getstring_ra)

#  procedure calculate -- adds current string length into the running
#                         total, and sets variables longest and
#                         shortest appropriately if this string is
#                         the longest or shortest so far.
calculate:  add   str_count, str_count, 1
            add   sum, sum, str_length
            ble   str_length, longest, nextif
            move  longest, str_length
nextif:     bge   str_length, shortest, calc_rtn
            move  shortest, str_length
calc_rtn:   b     (calculate_ra)

#  procedure average -- calculates an integer average by dividing
#                       the running total by the number of strings.
average:    div   ave, sum, str_count
            b     (average_ra)

#  procedure printresults -- prints the results of the program
#                            in a reasonable format.
printresults:putsstr2
            put   longest
            puts  str3
            puts  str4
            put   shortest
            puts  str3
            puts  str5
            put   ave
            puts  str6
            b     (printresults_ra)
```

Figure 11: SAL program that calculates longest, shortest and average string lengths.

tion of the program. The program reads in user generated strings, and figures out which one is the shortest, which is the longest, and the integer average length of the strings. Parameters are not passed to the C functions, since the SAL implementation does not provide for parameter passing. All variables are global. The goal in presenting both C and SAL versions of the same program is to see how the various pieces correspond.

Summary

SAL implements all the features of a high-level language: declarations, arithmetic operations, control structures, and communication with the user. SAL code looks like assembly language code. Each instruction or declaration is on its own line, and instructions are written with a mnemonic followed by one or more operands. The SAL language acts like an assembly language. Each instruction has a fixed number of operands, and performs a single, well-defined operation. All operations in an assembly language are explicit, unlike some operations in high-level languages.

Problems

1. Draw a diagram of a skeleton SAL program. Identify the different parts of the program, what pieces are optional, and where instructions and data belong.
2. Explain how to implement a boolean type variable in SAL. What is the variable's type, and how is it used?
3. Write SAL code for the following C for loop.

```
for (i=2; i<=z ; i++) {
      a = i mod 2;
      if (a == 0) then
            sum = sum + i;
}
```

4. Write SAL code that implements the following C code.

```
{
int   a, b, c, d, i;

      b = 13;
      for (i = 2; i <= a; i++) {
            c = b * i;
            if ( c != 0 )
            {
                  d = b - a;
                  d = d % c;
            }
      }
}
```

5. Are constants included in SAL? How is a constant specified and used in SAL?
6. From Figure 2, which of the two assembly language constructs would be more efficient if the C statement contained no else part? Why?
7. Write a SAL program that prints out a sequence of *n* Xs, where *n* is a positive integer entered by the user.
8. Write a SAL program that calculates average high and low temperatures for the month of February. Have the user enter high and low temperatures for each day.
9. Rewrite the SAL program that calculates average high and low temperatures in a modular way, using procedures.
10. The code given in Figure 5,

```
            move    result, 1
            move    counter, exponent
while:      blez    counter, endwhile
            mul     result, result, base
            sub     counter, counter, 1
            b       while
endwhile:
```

can be rewritten thus:

```
            move    result, 1
            move    counter, exponent
            blez    counter, endwhile
loop:       mul     result, result, base
            sub     counter, counter, 1
            bgtz    counter, loop
continue:
```

While these two methods have the same number of instructions, the number of instructions *executed* will differ. Give a value for `counter` for which the second method would execute fewer instructions than the first method. Give a value for `counter` for which the second method would execute more instructions than the first method.

11. The code in Figure 6 can be rewritten to eliminate the unconditional branch instruction at the end of the loop. Rewrite the SAL code segment so that fewer instructions are executed if `exponent` is ten.

12. In Figure 2, another possible way of writing the code would be the following:

```
      div   B, C, A
      bgtz  A, endif
      add   B, A, 10
endif:
```

Under what circumstances would this code be superior to the two versions given in Figure 2? Is there any reason for *not* using this code?

13. Write a SAL procedure that decides if the integer variable `value` is evenly divisible by 3. If `value` is evenly divisible, it should set the variable `flag` to 1, and if `value` is not evenly divisible, then it should set `flag` to 0.

14. Design and write a SAL program that calculates the area of a triangle. What information does the user need to enter for this program?

15. Design and write a SAL program that counts the number of punctuation marks in a paragraph entered by the user.