Chapter 2

1.1 Introduction

In Part 1 of the project, you learned about SaM, which is essentially a virtual computer with space for commands, data, and registers. Recall that computers can be instructed with ***machine code***, which is composed of an ***op code*** and an ***operand***. Since humans have trouble writing code as patterns of hexadecimal numbers, we write ***assembly code***, which is a mnemonic for machine code. For SaM, you write programs in our version of assembly code that we call ***Samcode***.

You wrote Samcode for a computer language with variables, declarations, assignments, and arithmetic. This assignment expands upon your previous work by introducing a high-level programming language that we call ***Bali***. Along with your previous work with operations and assignments, you will learn how to write control structures in Samcode. The programming for this part of the project involves writing a compiler that will translate Bali code into Samcode.

1.2 Not the Bali in your actual project!

Each semester we create a new Bali that looks like Java or C. As shown in lecture, the following "Bali--" program demonstrates an example grammar. **Note that the grammar in this chapter is different from the Bali in your project!** However, there are many common elements that you will find helpful:

```
main { // begin program

   { // begin varblock:
      int x ; int y ; int rv ;
   } // end varblock

   { // begin statblock:
      x = 2 ;
      if ( x > 1 )
         y = x ;
      rv = y ;
   } // end statblock

} // end program
```

Why do some elements seem weird? We have designed Bali for helping you write a compiler, not perform industrial programming. Note that this chapter focuses on control structures, leaving objects, functions, and other advanced concepts for later.

To handle SaM's return value of **`main`**, we use the following semantic requirements:

- **`main`** will have a return variable called **`rv`**.
- An assignment of **`rv`** must be the last statement.

1.3 Arithmetic, Declarations, and Assignments

Refer to Chapter 1 for examples of Samcode for arithmetic/logic (ALU) operations, variable declarations (and subsequent allocation), and variable assignments. You must use *relative*

addressing for this assignment to prepare for adding functions in future assignments with Bali. Recall the following rules:

- To allocate space for each variable, including a "return variable" (**`rv`**), adjust the stack pointer "up" the stack with **`ADDSP value`**.
- To place a value onto a stack, use **`PUSHIMM value`**.
- To store a variable value, which means assigning a variable, use **`STOREOFF address`**.
- To retrieve a variable value, use **`PUSHOFF address`**.
- To return a variable value, use **`ADDSP value`** to move the **`SP`** "down" to **`1`**.

Remember that we are assuming that all Samcode goes into only one function, **`main`**. In later assignments, we will introduce more instructions that deal with multiple functions.

1.4 Fundamentals of Control Structures

This section introduces the fundamentals of ***control structures*** for Bali and SaM, which are statements that allow you to perform selection and repetition.

1.4.1 Goto

In many programming "circles," the **`goto`** keyword has a bad reputation. You might be surprised to learn that not all **`goto`**s are created equally! SaM requires a form of **`goto`** called a ***jump***. So, we want you to know a little history of the **`goto`** before introducing the jump. You should read this classic paper: "Go To Statement Considered Harmful" by E. W. Dijkstra, *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147–148, http://www.acm.org/classics/oct95/.

1.4.1.1 Bad Goto

Suppose that you have a language that allows a **`goto`** statement. Such statements send control to another statement. To locate another statement, the language needs to allow ***labeling***, which means putting an address in front of a statement.* For instance, the following snippet of FORTRAN 77† demonstrates the wonders of **`goto`** and how a language might label a statement:

```
      PROGRAM ANTEDILUVEAN
      INTEGER N, MAX, SIGNAL
      PARAMETER (MAX=10, SIGNAL=5)
      DO 18 N = 1, MAX
         PRINT *, N
         IF (N .EQ. SIGNAL) THEN
            GOTO 19
         ELSE
            GOTO 20
         END IF
   18 CONTINUE
   19 PRINT *, 'LOOP ABORTED!'
   20 PRINT *, 'LOOP SUCCEEDED!'
      END
```

* See Page 45 of *Java in a Nutshell* for Java's implementation of a *labeled statement*.

† Yes, this is the first language DIS was taught in college.

Can you see the horror in using a **goto** indiscriminately to jump anywhere in your code?‡ In general, when a language supplies a robust control structure, such as **while** and **if**, you do not need to use a **goto**.

1.4.1.2 Good Goto

In your introductory programming course, you might have been told that languages, like Java, are *high level*, whereas a language, like assembly, is *low level*. One difference is that a high-level language provides more elegant control structures, whereas assembly typically provide bare bones, or primitive, instructions.

For instance, suppose that you need to print the values from 1 to 4. A simple **while** loop easily handles this problem, as shown below in pseudocode:

```
x ← 1
while x < 5
   print x
   x ← x + 1
```

However, assembly language generally provide instructions to perform *if-repeat*, which produces the same effects of the **while** loop, as shown below in pseudocode:

```
10 x ← 1
20 if x < 5 then
30       print x
40       x ← x + 1
50       goto 20
60 continue
```

In this case, we have a useful and necessary **goto** because the **goto** provides the mechanism for the looping control structure, which is essential for most programs.

1.5 SaM Instructions for Control Structures

In this assignment, we introduce two new SaM instructions, which assist with writing control structures. These instructions belong to the control classifications:

- **JUMP *t***: $PC \leftarrow t$. Jump to an instruction with address ***t***.
- **JUMPC *t***: If $V_{top} \neq 0$, $PC \leftarrow t$; else $PC \leftarrow PC + 1$. Pop the top of the stack. If the value is true, jump to the instruction with address ***t***. Otherwise, go to the next instruction.

To help remember the difference in names, the "**C**" in **JUMPC** stands for *condition*. These instructions are modeled after the "good **goto**" and are explained in more detail in this section.

1.5.1 Labels

SaM counts each instruction inside a Samcode file, starting with zero. As SaM runs each instruction, its **PC** register stores the number of the instruction currently being executed. Occasionally, you may need to go to an instruction besides the numerically subsequent instruction, as you would in cases of selection statement. In this case, we will apply the **goto** concept. Since a **goto**-like instruction requires an address of an instruction, you may either keep track of instruction numbers or write a ***label***. A label is a text string, above or to the left of the instruction to which you want another instruction to go.

‡ Does Java deliberately reserve the **goto** keyword with no meaning to prevent anyone from ever implementing it?

So, you may write either

```
label:
instruction
```

or

```
label: instruction
```

Either form of labeling (above or on the same line) is acceptable in terms of style. Internally, SaM replaces label with its actual instruction number.

1.5.2 Jumps

To provide a mechanism for **goto**, SaM uses a ***jump***, which is a statement that goes from one instruction to another in Samcode. So that an instruction "knows" where to jump, you will label Samcode instructions, as discussed above. To jump to the instruction below or to the right of ***label***, use **JUMP *label***. So, to provide jumping, do the following:

- Label a portion of Samcode with a text label followed by a colon (**:**)
- Use the Samcode statement **JUMP *label*** to have SaM jump to the labeled statement that begins with ***label:*** and start executing the statements following the label.

So, the overall structure of a jump and label might look something like the following Samcode:

```
stuff1
stuff2
JUMP hello
stuff3
hello:
stuff4
STOP
```

For the code above, assuming the instructions in ***stuff1*** and ***stuff2*** contain no jumps, SaM will execute code in ***stuff1*** and ***stuff2***, then jump to the first instruction below **hello:** (***stuff4***), and finally, stop. SaM will not execute ***stuff3***. Note that the following code is identical to the above example because you may write labels to the left of an instruction:

```
stuff1
stuff2
JUMP hello
stuff3
hello:stuff4
STOP
```

For a more specific example, try running the following Samcode:

```
PUSHIMM 2
JUMP blurt
PUSHIMM 100
blurt: PUSHIMM 3
ADD
STOP
```

After pushing **2** to the top of the stack, SaM jumps to the portion of the code that is labeled as **blurt**. Then, SaM executes the code in that block, which pushes the value of **3** and adds it to **2**. Thus, you will see that SaM returns a value of **5**. Note that **PUSHIMM 100** never executes. Be sure to run this example in SaM and keep track of the **SP** and **PC** registers.

1.5.3 Addresses

Upon loading a Samcode program, SaM stores each Samcode instruction along with its data (if any) in internal memory, as shown in SaM's `Program.java` and `SamProgram.java` files. Consequently, each instruction has an ***address***, which starts at **0** for the first instruction. After you load a SaM program, SaM shows the address of each instruction in the form ***address*:*instruction*** in the **Program Code** panel. SaM will show label information as well, as discussed below.

Do not think of labels as instructions! SaM numbers each Samcode statement with an address, but the labels become placeholders that are not numbered directly. For instance, run the following program, which has twelve lines of Samcode:

```
// Program for adding 2 and 3 in a convoluted fashion:
JUMP a      // line 1
d:          // line 2
STOP        // line 3
a:          // line 4
PUSHIMM 3   // line 5
JUMP b      // line 6
b:          // line 7
PUSHIMM 2   // line 8
JUMP c      // line 9
c:          // line 10
ADD         // line 11
JUMP d      // line 12
```

Do you see how this Samcode will produce a result of **5**? However, SaM knows that labels `a`, `b`, `c`, and `d` indicate a portion of code to which SaM will jump. Each instruction below a label gets the *next* address without counting the label itself. So, SaM condenses the written Samcode into a set of instructions and their numerical addresses. Figure 1.1 shows the **Program Code** panel for the above example:

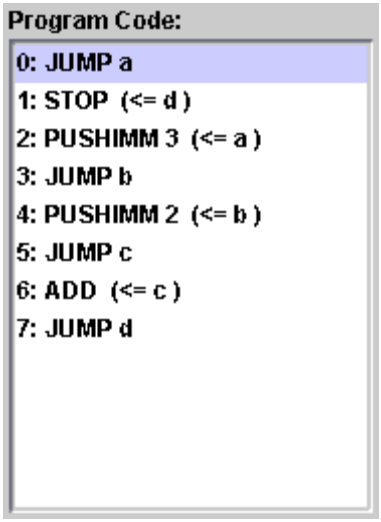

Figure 1.1: Convoluted Program in SaM

The labels **(<= a)**, **(<= b)**, **(<= c)**, and **(<= d)** indicate the names of the labels with which the instructions had originally been labeled. For instance, `PUSHIMM 3` begins with label **a** in the above example.

If you find labeling difficult to trace, inspect the **Capture Viewer** after execution. Figure 1.2 shows the order of the instructions, using their label names, for the example in this section.

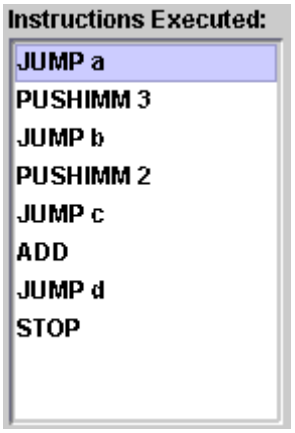

Figure 1.2: Execution Order for Convoluted Example

1.6 Selection Statements

Bali allows selection statements using **`if`** and **`else`** along with associated conditions and substatements. For instance, the following Bali program tests if a variable (**`x`**) exceeds a value (**`2`**):

```
main {
   {  int rv ; int x ; int flag ; }
   {  x = 3 ;
      flag = 0 ;           // give flag a dummy value
      if ( ( 2 < x ) ) {  // remember to surround expressions by ()
         flag = 1 ;
      }
      rv = flag ;          // return rv
   }
}
```

Since **`2 < 3`** is true, this code returns true (**`1`**).

As shown below in Step 1, the first part of the Samcode for this Bali code uses the techniques shown in Part 1 of the project. You will use a relative addressing approach to prepare for future assignments with function calls:

```
// Step 1: Start program and set variables
ADDSP 3      // adjust SP to account for rv, x, and flag
PUSHIMM 3    // push value of 3
STOREOFF 1   // store the value 3 in address 1 for x
PUSHIMM 0    // push the value of 0 (false)
STOREOFF 2   // store the value 0 in address 2 for flag
```

Below, Step 2 continues the Samcode for the Bali example. To execute the **`if`** statement, SaM first needs to evaluate a condition, which is placed on top of the stack before jumping to the correct portion of the Samcode. To evaluate the condition **`2 < x`**, you must retrieve the value of **`2`** and then push the value **`x`**. The top of the stack will be the result of evaluating the condition. By calling **`LESS`**, SaM will push **`0`** (false) or **`1`** (true), depending on the results of the comparison, which in this case will be **`1`**:

```
// Step 2: Check if 2 < x
PUSHIMM 2    // push the value 2 to compare with x (Vbot)
PUSHOFF 1    // push the value of x (Vtop)
LESS         // Push result of (Vbot < Vtop) to top of stack
```

Now that you have accounted for the condition part of the **if**-statement, you must jump to an appropriate instruction. Since a condition is either true or false, use **JUMPC**, which has the following behavior:

- If V_{top} is true, $PC \leftarrow t$. So, SaM will jump to the instruction with label ***t***.
- Otherwise, $PC \leftarrow PC + t$. So, SaM will execute the next instruction, which is directly underneath the **JUMPC** instruction.

Thus, calling **JUMPC *t*** will move to the label ***t*** if the condition is true. Otherwise, SaM will continue executing Samcode because the condition was false.

Step 3 finishes the Samcode for the Bali example. A block of Samcode for the case when **2 < x** is true is labeled as **correct**, though we could have used another name. In addition, when the **if** and **else** clauses finish, SaM needs to move to the statement after the entire selection statement. So, we have provided another block labeled as **continue** to finish the remaining program, which retrieves the value of **flag** (which will be **1**) and returns it:

```
// Step 3: Process if statement
JUMPC correct  // check if result of GREATER is true (1) or false (0)
               // false:
               //    if you had an else in Bali, you would handle it here
JUMP continue  //    continue with remaining program
correct:       // true:
PUSHIMM 1      //    push the value 1 (true)
STOREOFF 2     //    store the value true for flag
JUMP continue  //    continue with remaining program
continue:      // continue with program:
PUSHOFF 2      //    push the value of flag
STOREOFF 0     //    store the value of flag in rv
ADDSP -2       //    reset the SP
STOP           //    done with the program
```

As with the Bali code, you will find that the Samcode returns **1** (true). Actually, the code above is a bit redundant. You may remove the second **JUMP continue** statement since SaM will "fall through" the label after analyzing the **correct** block.

1.7 Repetition

Samcode uses an *if-repeat* structure of a loop (see Section 1.4.1.2) with judicious choice of labels and jumps:

- Evaluate a Boolean expression, which we call the ***condition***.
- If the condition is true, do the statements that follow the condition.
- If the condition is false, evaluate the statement that follows the **while** statement.
- After the statements are finished, go to test the condition again.

For example, a Bali program that increments a variable **x** from values **1** to **6** might be as follows:

```
main {
   { int x ; int rv ; }
   {
      x = 1 ;
      while ( ( x < 5 ) ) {
         x = ( x + 1 ) ;
      }
      rv = x ;
   }
}
```

In Samcode, you need to encode loops with jumps and labels. Since **JUMPC *label*** jumps to another portion of Samcode for a true condition, you will need to apply a label to the condition. Thus, SaM will be able to keep evaluating the condition until it becomes false. At that point, you need another jump to exit the loop. So, you need two labels in the following program:

- **looplabel**: the entry point for the loop in which the condition is evaluated and tested.
- **continue**: the statements that follow the **while** statement.

The program will return the final value of **x**, which will be **6**, as shown below:

```
ADDSP 2         // leave space for x and rv
PUSHIMM 1       // push 1 on the stack
STOREOFF 1      // store the value 1 for x
looplabel:      // label the loop starting at the condition
PUSHOFF 1       // retrieve the value of x
PUSHIMM 5       // push the value to compare x with
LESS            // is x < 5 ? push 1 if so; otherwise, 0
JUMPC continue  // if x < 5, do statements under continue
done:           // x >= 5, so move to statement after the while-block
PUSHOFF 1       // retrieve the value of x
STOREOFF 0      // store the value of x as the rv
ADDSP -1        // deallocate x
STOP            // stop processing and return the rv value
continue:       // the block of statements that follow the loop
PUSHOFF 1       // retrieve the value of x
PUSHIMM 1       // push 1 onto the stack
ADD             // add 1 to the current value of x
STOREOFF 1      // store the new value of x
JUMP looplabel // repeat the loop (goto loop condition)
```

Remember that you may choose arbitrary label names, though you should choose names that indicate their purpose.