SaM

1. Introduction

Have you ever heard the Java mantra, “write once, run everywhere?” So, how does that work? When you compile a Java program, you generate a binary file that contains ***byte-code***. Byte-code is a collection of instructions that resemble machine code. However, you cannot simply run the byte-code on your computer because the byte-code is written for a “virtual computer,” which we call the Java Virtual Machine (***JVM***). To run your actual Java program from the JVM, most computers have a byte-code interpreter that converts each byte-code instruction for the particular architecture that you use. By learning about the JVM, you can learn about how programs are compiled and executed on a computer.

2. Stack Machine

2.1 SaM

Although we do not have the time to study the complete JVM[*], one important facet is the JVM’s use of stacks to store instruction information. A ***stack*** is data structure that stores items in “last-in, first-out” order. So, to help you learn about how a computer executes your program, we are providing you with ***SaM***, which approximately stands for *a stack machine*. (It took me awhile to realize the rearrangement of the *a*, plus it sounds better than *ASM*.) You will be able to write code in pseudo-assembly code (***sam-code***) that mimics actual assembly code.

Sam-code mimics assembly code, which has the form ***`op-code operand`***. For instance, the sam-code **`PUSHIMM 3`** has the op-code **`PUSHIMM`** and operand **`3`**. When SaM encounters this instruction, SaM puts the value of **`3`** on top of its stack of data. As you work your way through this document you will learn more about SaM’s organization and further sam-code instructions.

2.2 SaM Values

SaM has only one data type, which is *integer*, though we might add others later. The limits of SaM’s integer type are identical to those of Java. To simulate Boolean values, you can use **`0`** for false and **`1`** for true.

2.3 SaM Memory

SaM has two areas of memory, which are called ***program*** and ***stack***:

- ***Program***: where SaM loads the user’s program prior to execution. The user’s program will be a collection of sam-code instructions, which you will see in Section 3.
- ***Stack***: where SaM stores data during the execution of the program.

SaM also contains four registers, which are called PC, SP, FBR, and HALT. These registers all store non-negative integers:

- ***PC*** (Program Counter): contains the address of the sam-code instruction that SaM is currently executing.

*. Interested in seeing actual JVM byte-codes? See http://java.sun.com/docs/books/vmspec/ for a free on-line book.

- ***SP*** (Stack Pointer): contains the address of the first *free* location on the stack. The first address in the stack is **`0`**. The subsequent addresses increment by one. For now, assume that the stack has unlimited address space.
- ***FBR*** (Frame Based Register): contains information to help keep track of function calls. We will assume an FBR of numerical zero for this assignment.
- ***HALT***: contains a signal for whether or not SaM should keep executing the program. SaM executes the program while HALT contains **`0`**. To stop execution, an instruction must insert **`1`** into the HALT register.

3. SaM Instructions

This section provides an overview of the complete SaM instruction set. You will not need many of these instructions for this assignment, though you will use more later.

3.1 Classifications

SaM has four classifications of instructions:

- ***ALU instruction***: These instructions perform arithmetic and logical operations, which include addition, subtraction, logical and, logical or, etc. on integers. The operands are popped from the stack and the result is pushed back to the stack.
- ***Stack manipulation instruction***: These instructions copy a value from one location in the stack to another.
- ***Register save/restore instruction***: These instructions permit the values of the registers to be pushed and popped from the stack.
- ***Control instruction***: These instructions implement conditional and unconditional transfer of control in the program.

The instructions are stored in the program memory. After the execution of an ALU, stack manipulation, or register instruction, control transfers to the next instruction in program memory. A control instruction "moves" to another instruction in program memory.

We provide lists of instructions in the following four sections, below. Many instructions operate on operands, which are stored in the stack and are, thus, sometimes called ***stack elements***. We denote a stack element at location i as V_i. Assume that V_{top} and V_{below} refer to the top-most element and the element below it, respectively, *before* an instruction is executed. For a command that needs the V_{top} and V_{below} operands, SaM will pop them before pushing any results onto the stack. Note that all op codes are strictly uppercase!

3.2 ALU Instructions

Instruction	**Pseudocode**
`ADD`	Push $V_{below} + V_{top}$.
`SUB`	Push $V_{below} - V_{top}$.
`TIMES`	Push $V_{below} \times V_{top}$.
`DIV`	Push V_{below} / V_{top}.

3.2 ALU Instructions

Instruction	Pseudocode
CMP	Push $\left\{\begin{array}{c} V_{top} > V_{below} \\ V_{top} = V_{below} \\ V_{top} < V_{below} \end{array}\right\} \rightarrow \left\{\begin{array}{c} 1 \\ 0 \\ -1 \end{array}\right\}$.
ISPOS	If $V_{top} > 0$, push 1. Otherwise, push 0.
ISNIL	If $V_{top} = 0$, push 1. Otherwise, push 0.
ISNEG	If $V_{top} < 0$, push 1. Otherwise, push 0.
NAND	Push $\neg(V_{below} \wedge V_{top})$. What is a "NAND?" you may ask? See http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=nand.

3.3 Stack Manipulation Instructions

Instruction	Pseudocode
PUSHIMM *c*	Push the value c, which is an integer.
DUP	Duplicate top element: $V_{SP} \leftarrow V_{SP-1}$; $SP \leftarrow SP + 1$.
SWAP	Exchange the top two elements on the stack.
PUSHIND	Push $V_{V_{top}}$.
STOREIND	$V_{V_{below}} \leftarrow V_{top}$.
PUSHOFF *k*	Push V_{k+FBR}.
STOREOFF *k*	$V_{k+FBR} \leftarrow V_{top}$.

3.4 Register Save/Restore Instructions

Instruction	Pseudocode
PUSHSP	$V_{SP} \leftarrow SP$; $SP \leftarrow SP + 1$.
POPSP	$SP \leftarrow SP - 1$; $SP \leftarrow V_{SP}$.
ADDSP *c*	c is an integer; $SP \leftarrow SP + c$.
PUSHFBR	Push FBR: $V_{SP} \leftarrow FBR$; $SP \leftarrow SP + 1$.
POPFBR	Pop FBR: $SP \leftarrow SP - 1$; $FBR \leftarrow V_{SP}$.
LINK	$V_{SP} \leftarrow FBR$; $FBR \leftarrow SP$; $SP \leftarrow SP + 1$

3.5 Control Instructions

Instruction	Pseudocode
`JUMP t`	$PC \leftarrow t$.
`JUMPC t`	If V_{top} is true, $PC \leftarrow t$; else $PC \leftarrow PC + 1$.
`JUMPIND`	$PC \leftarrow V_{top}$.
`JSR t`	Push $PC + 1$; $PC \leftarrow t$.
`JSRIND`	Push $PC + 1$; $PC \leftarrow V_{top}$.
`STOP`	$HALT \leftarrow 1$.

3.6 SaM Programs

A SaM program is a *text* file that contains a collection of sam-code instructions along with optional comments and labels:

- Comments begin with `//`. Any text that follows the `//` belongs to the comment.
- SaM ignores comments.
- Each instruction must be written in entirety on the same line.
- Each new line forms a new instruction.
- We will discuss labels in the next assignment.

For example, the following sam-code program pushes a value on the stack, checks if it is positive, and then halts:

```
// Sample sam-code
// sample.sam

PUSHIMM 10   // push the value 10
ISNIL        // check if the top of stack is zero
STOP         // halt execution
```

We recommend that you use the commenting style as shown in the above example.

4. The Simulator

To run a sam-code program, you need to use SaM. This section explains how to start the SaM simulator and run your sam-code programs inside of it.

4.1 GUI Interface

We have programmed SaM for you to use. To run it, follow these steps:

- Download the zip file that contains the Java source code.
- Compile *all* of the code. The Main Class is `GUISimulator`. In CodeWarrior, ensure that you set your target correctly!
- Run the program, using Java 2, SDK 1.2.2 or higher. In CodeWarrior, you should be using Java 1.3 stationery, or higher.

On the next page, Figure 1 shows the initial start-up window:

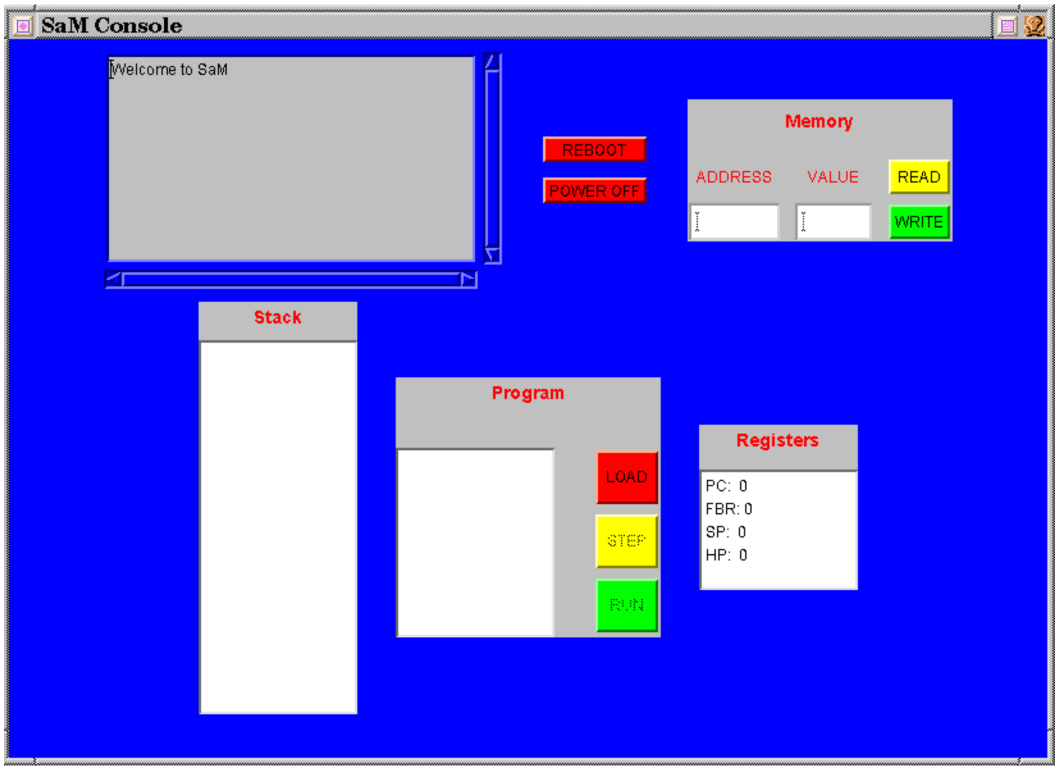

Figure 1: SaM Simulator

The interface provides a rough approximation of a virtual machine's architecture. The **Stack** and **Program** areas correspond to the stack and program memories that Section 2.3 introduces. You will also see four register values in **Registers**. Do not worry about the FBR and HP values for now – they relate to implementation of methods and objects. Also, ignore the **Memory** box. Instead, you will use the box that says **Welcome to SaM**, **Stack**, **Program**, the **Power Off** and **Reboot** buttons, and the **PC** and **SP** registers.

4.2 Running Sam-Code

To run a sam-code program, do the following:

- Create a sam-code program with a text editor.
- Press the **Load** button.
- Select your file by using the browser that pops up.
- Press **OK** in the browser.
- Press the **Run** button to execute the program. If your program returns a value, the welcome window will report an answer.
- Use the **Step** button to execute each instruction manually, one at a time.
- Click the **Reboot** button to reset the simulator.
- Click the **Power Off** button to shut down the simulator.

Practice these steps with the examples in Section 5. When running different programs, values that remain in the stack are overwritten for other runs. Why? The SP starts "below" the old values.

4.3 SaM’s Output

There are a few general messages that SaM reports in its console:

- welcome: **Welcome to SaM** indicates that you just started SaM.
- ready: **I am SaM** indicates that you are loading a file.
- loading: **Loading SaM Commands from file *name*** indicates the file that SaM is attempting to access.
- error: **Funny opcode *string*** and **Undefined name *string*** alert you that SaM does not recognize a particular opcode or operand, respectively.
- success: **Program terminated normally** indicates that the program ran without a return value. Otherwise, SaM will report **Answer is *value***.

To have SaM return a value as a result of executing your sam-code, you must ensure that the instructions leave only one value remaining in the stack. The simulator will detect that you have an answer when the first cell (address 0) has a value, the stack pointer (SP) contains the address of the next cell (address 1), and SaM encounters a **`STOP`** command. So, you must carefully choose your sam-code such that the last value is the value you intended for the program to compute.

5. Sam-Code Examples

As you would write any program, you will write programs in sam-code as a sequence of instructions. For the required portion of this assignment, you will not use **`PUSHSP`**, **`POPSP`**, **`PUSHFBR`**, **`POPFBR`**, **`JUMP`**, **`JUMPC`**, **`JUMPIND`**, **`JSR`**, **`JSRIND`**, and a maybe few others, because you will not be modeling function execution. You will write programs that would likely occur inside a single function, such as arithmetic expressions and variable assignments, to perform basic computations.

5.1 Postfix Notation

To calculate 10+20 using SaM, you need to think of the expression in ***postfix notation*** as 10 20 +. Postfix notation means that operators are placed at the end of an expression. Knowing that + operates on two values means that you should store the first two values somewhere, namely, your brain, and then add them after encountering the + operator. In sam-code, you would this arithmetic operation as:

```
PUSHIMM 10
PUSHIMM 20
ADD
STOP
```

To help you visualize the results of pushing **`10`** and **`20`** onto the stack, refer to Figure 2. If you step through this example in the simulator, you will see that the stack pointer (SP register) contains the value 2, which is one address higher than the address of the last instruction. When the code issues **`ADD`**, both **`10`** and **`20`** pop from the stack and SaM pushes the result of **`30`** into address 0. Since the **`30`** is only remaining value in the stack, the simulator will report **`30`** as the answer. Note that SP will contain **`1`**, because that address is one higher than the last instruction (**`STOP`**). Isn’t that neat?

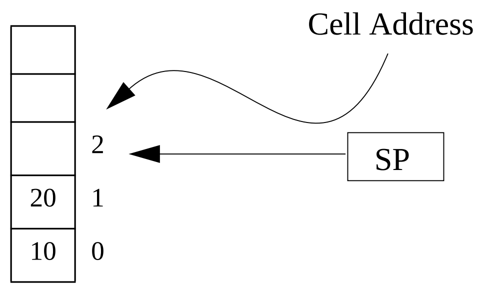

Figure 2: Stack Memory

Now, try a program that performs $1 + (2 \times 3)$:

```
PUSHIMM 1
PUSHIMM 2
PUSHIMM 3
TIMES
ADD
STOP
```

In this case, sam-code advantageously helps to avoid worrying about operator precedence because the operators appear after the operands.

5.2 Assignments and Sam-Code

Ultimately, you will be writing a program called a compiler that will translate one language into sam-code. For example, suppose you were trying to write a compiler to convert the following Java snippet into sam-code:

```
int x , y ;
x = 10 ;
y = 20 ;
return ( x + y ) ; // returns 30
```

Writing the sam-code requires a bit of work because of the variables. You must reserve space for each variable in the stack memory. To help keep track of the variables and their locations, manually draw a ***symbol table***, as shown in Figure 3. A symbol table is a collection of the program's variables plus an additional variable, `rv` (***return variable***), that stores the program's result. In this case, `rv` would represent the result of `x+y`.

Variable	Address
`rv`	0
`x`	1
`y`	2

Figure 3: Symbol Table

In stack memory, you should draw another figure to represent the stack, as shown in Figure 4. Since we will assume that we are converting code from a `main` function, the FBR is zero. When we add more functions, we will adjust this register. By leaving the first cell open for the **`rv`**, the results of the program will have a place for SaM to store and return the answer. The **`main`** variables are stacked on top of the first cell in the order in which they are declared. The address of a variable corresponds to a cell address on the stack.

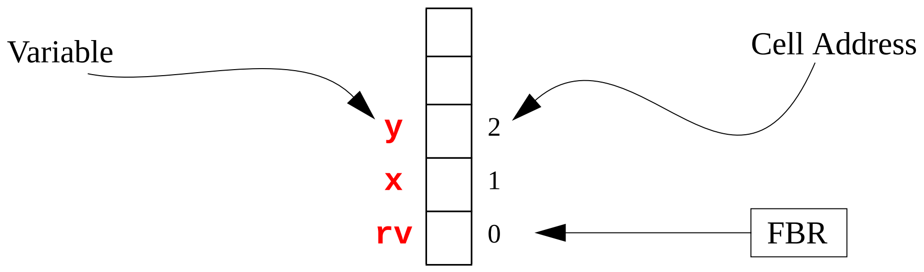

Figure 4: Stack Memory with Variables

How do you write sam-code for this program? You have two options: ***absolute addressing*** and ***relative addressing***. In absolute addressing, you do not need to worry about the mysterious FBR, because you forget all about functions. Otherwise, when we introduce functions, you run into trouble unless you use relative addressing to shift the FBR every time a new function is called.

5.2.1 Absolute Addressing

You need to make space for the variables. So, to replace the declaration statements, move the stack pointer up two cells with **`ADDSP 3`**, which leaves space for the two variables and the "return variable," **`rv`**. You may ensure that any old values remaining from previous runs are removed by pushing a value of zero into the first cell for **`rv`**, which will then start as zero. To do so, use **`STOREIND`**, which takes a value from V_{top} and pushes it into the location specified by V_{below}. Thus, you would do the following steps:

- Enter **`PUSHIMM 0`** to give the *address* in which to write
- Enter **`PUSHIMM 0`** for the *value* that you wish to store
- Enter **`STOREIND`** to move the value zero into the zeroth address.

You will repeat similar instructions to push and place the values of **`x`** and **`y`**. For assigning each variable, follow the same process: push the address, push the value, and store the value at the address.

To use a variable value, you must first retrieve it! Make sure that you remember the address where you put it. You will push that address with **`PUSHIMM`** again. Using that address, you enter **`PUSHIND`**, which forces SaM to retrieve the value from the address that you had previously pushed. Sam places the retrieve value on top of the stack. Having retrieved the needed variable values, you may then perform operations on them.

The following sam-code performs the operations of the Java code-snippet in this section. As you read though the example, note how I methodically stored and retrieved each variable value.

```
// Absolute addressing
// absolute.sam

ADDSP 3      // leave space for three variables
PUSHIMM 0    // location in which to store rv
PUSHIMM 0    // value to store in rv’s location
STOREIND     // store value 0 in address 0
PUSHIMM 1    // location to store x
PUSHIMM 10   // value to give x
STOREIND     // store value of x
PUSHIMM 2    // location to store y
PUSHIMM 20   // value to give y
STOREIND     // store value of y
PUSHIMM 0    // location to store result of x+y
PUSHIMM 1    // push address of x
PUSHIND      // retrieve value of x
PUSHIMM 2    // push address of y
PUSHIND      // retrieve value of y
ADD          // add values of x and y
STOREIND     // store value of x+y in address rv
ADDSP -2     // remove x and y from memory
STOP         // halt --> should return 30
```

To fully understand this approach, step through the code very carefully in the simulator. In particular, keep track of the SP register. You may also wish to draw your own stack to keep track of cell entries. Note that this technique limits the generality when accounting for functions, since each function will have its own variables. The next section demonstrates the technique that we prefer that you use.

5.2.2 Relative Addressing

By using the FBR to keep track of your current method call, you can keep your sam-code very general. Since we are assuming no other method than our “main,” FBR stays at zero. Judicious use of the **STOREOFF *x*** command provides the best way to help with variables. First, you need to advance the SP by the number of variables that you have, including the **rv**. To store a variable’s value, you push the value and then move it to the correct position in the stack. For instance, since **x** is the first variable, you would enter **PUSHIMM 10** and then **STOREOFF 1**. This instruction moves V_{top} (which is 10) into the cell with address of 1 (which refers to $V_{k+FBR} = V_{1+0} = V_1$). You would enter a similar instruction for **y**. After storing the variable values, you need to extract and add them. To extract a value, enter **PUSHOFF *k***, which pushes the value stored in address ***FBR+k*** to the top of the stack. Once you have both values pushed, you can then add them and move the result to the **rv** address. Since SaM will only return the value in the first cell (address 0), you need to alert SaM that you have finished by moving the SP to the second cell (address 1) and stopping the program (**STOP**). We have provided the code for you to test:

```
// CS212 Fall 2002
// Part 1
// relative addressing
// relative.sam


ADDSP 3      // make space for 3 variables (x, y, rv)
PUSHIMM 10   // push the value to store for x
STOREOFF 1   // store the value 10 in x
PUSHIMM 20   // push the value to store for y
STOREOFF 2   // store the value 20 in y
PUSHOFF 1    // retrieve the value of x
PUSHOFF 2    // retrieve the value of y
ADD          // x+y
STOREOFF 0   // store the value of x+y in rv
ADDSP -2     // remove x and y from the stack
STOP         // halt -- the return value should return 30ADDSP 3
```

The commands for relative addressing are very similar to those of absolute addressing. The main difference is whether or not you account for functions, which in this portion of the project, isn't an issue.