Chapter 1: Introduction to SaM

1.1 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.

1.2 Stack Machine

1.2.1 SaM

We do not have the time to study the complete JVM[*], so we will be simulating the JVM with ***SaM***. SaM is a software package developed by Professor Keshav Pingali to simplify the JVM. SaM stands for **St**a**ck** **M**achine. A ***stack machine*** is essentially a device that pushes and pops information from a ***stack***, which is a data structure that stores items in “last-in, first-out” order. Since the JVM uses stacks to store information, SaM provides a relatively easy and graceful way to learn about the JVM and compilers.

You will write code in ***Samcode***, which is SaM’s pseudo-assembly code that mimics actual assembly code. However, you will not actually generate binary files as you do for the JVM. Samcode mimics assembly code, which has the form ***`op-code operand`***. For instance, **`PUSHIMM 3`** has the op-code **`PUSHIMM`** and operand `3`. When SaM encounters this instruction, SaM puts the value `3` on top of its stack of data. As you work your way through this document you will learn more about SaM’s environment, internal structure, and more Samcode instructions.

1.2.2 SaM Values and Types

SaM has a variety of data types. For now, we will focus on *integer* and *memory*, though we will learn others later. The limits of SaM’s integer type are identical to those of Java. To simulate Boolean values, you can use integer `0` for *false* non-zero integers for *true*. Value of type memory are represented as integers and refer to locations in SaM’s memory, which is explained in the next section.

1.2.3 SaM Memory

SaM has two areas of memory that you will use. These are called ***program*** and ***stack***:

- ***Program***: Main memory where SaM loads the user’s program prior to execution. The user’s program is a collection of Samcode instructions, which you will see in Section 1.3.
- ***Stack***: Memory in CPU where SaM stores data during the execution of the program.

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

SaM also contains other smaller areas of memory in the processor called ***registers*** to store counters that keep track of information about a program. SaM's **PC**, **SP**, **FBR**, **HP**, and **HALT** registers all store non-negative integers and have the following purposes:

- ***PC*** (Program Counter): contains the address of the Samcode instruction that SaM is currently executing.
- ***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 `0` for this assignment.
- ***HP***: contains information about objects. We do not need this register in this assignment.
- ***HALT***: SaM has a special internal register that keeps track of whether or not SaM should stop processing instructions.

1.3 SaM Instructions

This section provides an overview of a portion of the SaM instruction set. You may not need all of these instructions for this assignment. We will introduce more instructions later as we develop functions and classes.

1.3.1 Classifications

SaM has four classifications of instructions:

- ***ALU (Arithmetic/Logic Unit) instruction***: These instructions perform arithmetic and logical operations, which include addition, subtraction, logical and, logical or, etc. on integers. When SaM performs an ALU instruction, the operands of the instruction are popped from the stack. If there is a result, it is pushed on top of the stack. Note that some ALU instructions have only one operand.
- ***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.
- ***I/O instruction***: Input and output instructions to communicate with the user and devices.

The instructions are stored in the program area of 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" execution to another instruction in program memory.

1.3.2 The Stack Memory and Stack Pointer

SaM's ***stack*** area of memory is essentially a column in which you can place "items" (integer values) in ***cells*** above each other. SaM addresses each cell, starting from **0**. For the first instruction that generates a result, SaM will store that value in this first cell. The next result gets stored in the cell with the address of **1**, which is above the first cell, and so forth. To help keep track of which cell is empty, SaM uses the **SP** (stack pointer) register to "point" to the next free cell. How does **SP** "point" to a cell? Since the **SP** stores only nonnegative integers, by storing the *address* of the next free cell in **SP**, SaM can retrieve that free address from the **SP** whenever a program requires something pushed or popped from the stack.

For an example of this process, refer to Figure 1.1. You can see the results of two instructions that pushed the values of `10` and then `20` onto the stack and moved **SP** at each step. The process went as follows:

- First, **SP** started at **0**.
- Next, SaM pushed `10` into that cell address and incremented **SP** to **1**.
- Next, SaM pushed `20` and incremented **SP** to **2**.

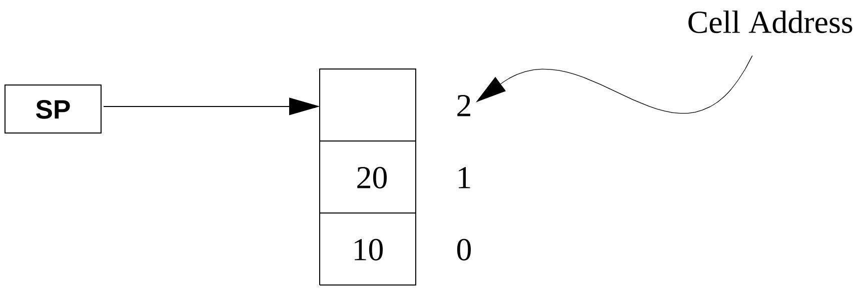


Figure 1.1: Stack Memory

SaM keeps track of cell addresses and register values, as you will see in later sections.

1.3.3 Instruction Set: "Some of SaM"

An ***instruction set*** is the complete collection of the instructions that a CPU uses. We provide a portion of SaM's instruction set in Section 1.3.3.1, Section 1.3.3.2, Section 1.3.3.3, and Section 1.3.3.4. Refer to the SaM documentation on the CS212 website for the complete set.

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. As discussed in the previous section, SaM's stack pointer (**SP**) always points to the first free location on the stack. Therefore, you can express V_{top} and V_{below} as follows:

$$V_{top} = V_{SP-1} \tag{1}$$

and

$$V_{below} = V_{SP-2} . \tag{2}$$

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! We do not show many of the available SaM instructions because we do not need them yet.

1.3.3.1 ALU Instructions

`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} .

`NOT`	Push $\neg V_{top}$.
`OR`	Push $V_{below} \vee V_{top}$.
`AND`	Push $V_{below} \wedge V_{top}$.
`GREATER`	Push $V_{below} > V_{top}$.
`LESS`	Push $V_{below} < V_{top}$.
`EQUAL`	Push $V_{below} = V_{top}$.

1.3.3.2 Stack Manipulation Instructions

`PUSHIMM` *`c`*	$V_{SP} \leftarrow c; SP \leftarrow SP + 1$. Push the value c, which is an integer.
`DUP`	$V_{SP} \leftarrow V_{SP-1}$; $SP \leftarrow SP + 1$. Duplicate top element.
`SWAP`	Exchange the top two elements on the stack.
`PUSHABS` *`k`*	Push V_k.
`STOREABS` *`k`*	$V_k \leftarrow V_{top}$.
`PUSHOFF` *`k`*	Push V_{k+FBR}. Push `V[`*`k`*`]` at an *offset* of ***FBR***; assume **FBR**=`0` for now.
`STOREOFF` *`k`*	$V_{k+FBR} \leftarrow V_{top}$. Store `V[`*`k`*`]` at an *offset* of ***FBR***; assume **FBR**=`0` for now.

1.3.3.3 Register Save/Restore Instructions

`ADDSP` *`c`*	$SP \leftarrow SP + c$. c is an integer.

1.3.3.4 Control Instructions

`STOP`	$HALT \leftarrow 1$.

1.3.4 SaM Programs

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

- Comments begin with `//`. Any text on the same line following `//` belongs to the comment.
- SaM ignores comments and blank lines.
- Each instruction must be written in entirety on the same line.
- An instruction may be given a label, which is discussed in the next assignment.

For example, you can write Samcode that continues the example shown in Figure 1.1. The following Samcode program (`equal.sam`) pushes two values (`10`, then `20`) on the stack, checks if they are equal, returns the result of `0` (because $20 \neq 10$) and then halts:

```
PUSHIMM 10   // push the value 10
PUSHIMM 20   // push the value 20
EQUAL        // push 0 because 20 != 10
STOP         // halt execution
```

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

1.4 The Simulator

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

1.4.1 How To Start SaM

We have programmed SaM for you to use. Please access the SaM link on the CS212 website. To run it, follow the steps listed on the website. You should see a window, as shown in Figure 1.2.

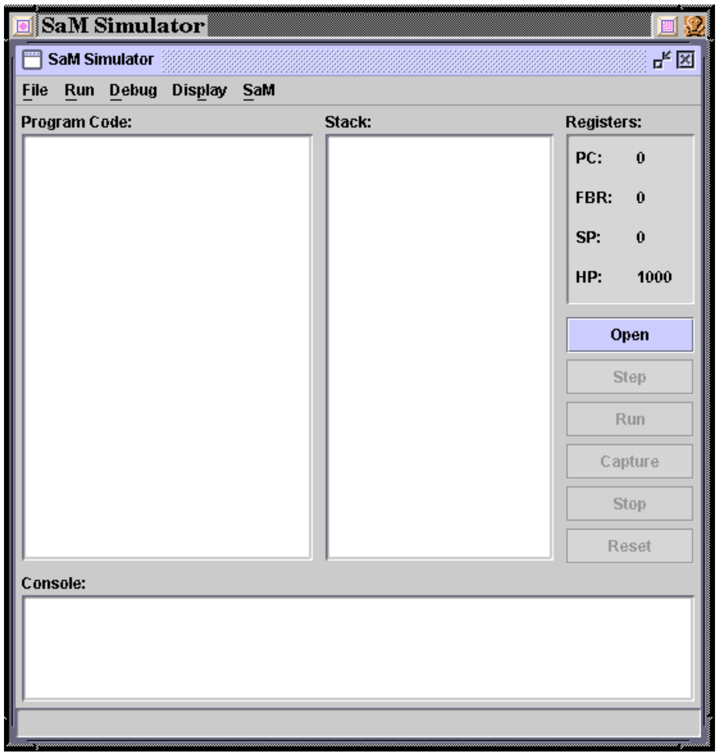


Figure 1.2: SaM

1.4.2 GUI Interface

The interface provides a rough approximation of a virtual machine 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**. We will provide a brief overview of the menus and buttons to help you get started. For this assignment, you will need to assume that **FBR** is 0 throughout the execution of a Samcode program. We will discuss more about the **FBR** and **HP** registers for functions and objects in later parts.

1.4.3 Running Samcode

To run a Samcode program, follow these steps:

- Create a Samcode program with a text editor.
- Use the **File** → **Open** menu option or **Open** button to load the file.
- 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 **Console** window will report an answer. To change your execution speed, select **Run** → **Execution Speed**.
- Use the **Step** button to execute each instruction manually, one at a time. If you already ran the program you will have to reset it using the **Reset** button.
- Click the **Reset** button to reset the simulator.
- Click the **Capture** button to execute the entire program and display a **Capture Window** that shows the stack during the execution of the individual instructions.
- Set breakpoints by double clicking on an instruction. If you run and capture a session, SaM will follow the breakpoints. To disable a breakpoint, double click on the same instruction.

As you start running examples, you will discover that values that remain in the stack are overwritten for other runs. Why? The **SP** starts "below" the old values.

1.4.4 SaM's Output

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

- **Assembler error**: alerts you that SaM does not recognize a particular op-code or operand, which is stated below the error.
- **Exit Code: *n***: Program terminated normally. The code ***n*** gives the return code or the program answer.

If you execute the program using the **Capture** button, the **Capture Viewer** window appears. This displays the program on the left and the stack view for the different execution stages on the right. This facility works well for Samcode programs with several instructions.

SaM will report the bottom of the **Stack** in the **Console** with the **Exit Code**. When SaM encounters a `STOP` command, it displays the value in address **0**. Thus, you should be careful to leave only value on the stack if you intend to return that value.

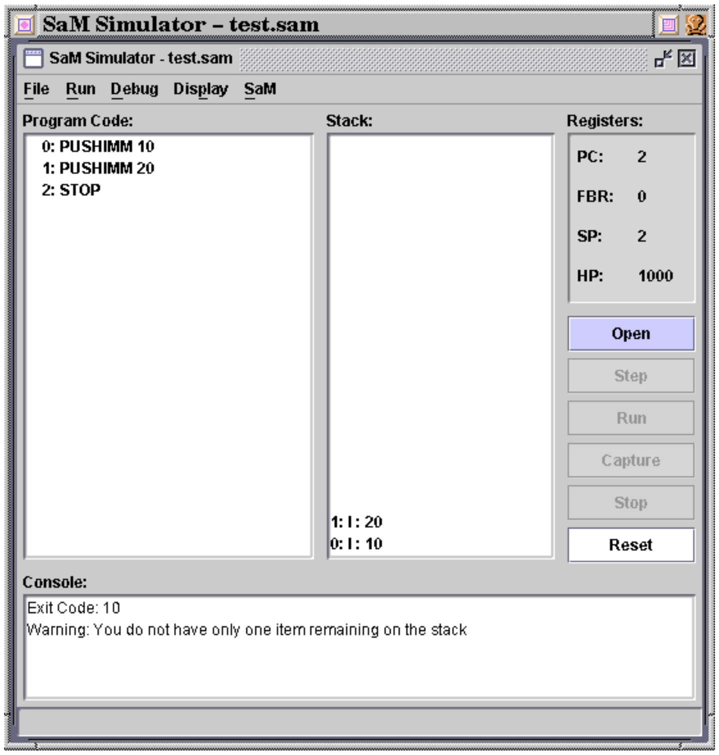
1.4.5 Stack

There are some design issues that you might find disconcerting at first. As SaM executes instructions, you will see values pushed and popped onto the stack in the **Stack** panel. For example, run the following Samcode:

```
PUSHIMM 10
PUSHIMM 20
STOP
```

As shown in Figure 1.3, after running the program `test.sam`, SaM has pushed the values `10` and `20` and then stopped. Along with each pushed value, SaM gives the value's address and type with the format ***address*:*type*:*value***. For example, **1:I:20** means that the value **20** has address **1** and type **I** (integer) on the stack. Note that we receive an error code because SaM excepts only one value to remain on the stack. In the next section, you will learn how to write a Samcode program that satisfies this constraint. Of course, we have not answered *why* SaM wants just one value–for now, consider the stack is where function calls go, and a function generally returns one value.

To keep track of the types, refer to the **Help** → **Stack Colors Reference**. For the stack address, keep track of the **SP** register in the **Registers** area. In this assignment, you will see just **I** for integer values and **M** for memory values.



```
0: PUSHIMM 10
1: PUSHIMM 20
2: STOP
```

```
1: I: 20
0: I: 10
```

```
PC:   2
FBR:  0
SP:   2
HP:   1000
```

```
Exit Code: 10
Warning: You do not have only one item remaining on the stack
```

Figure 1.3: SM's Stack and Other Output

1.4.6 Instruction Loader

As you develop your compiler, you will discover some handy tools that SaM sports. For example, you may wish to try making your own SaM instructions. Rather than recompiling all of SaM, you add an instruction directly into for a particular session using the ***Instruction Loader***. First, investigate SaM's source code to learn how we have written Samcode instructions in Java. Following the pattern that we used, write your own instruction in Java and compile it into bytecode, which is a `.class` file. To load the instruction into SaM, select **File→Load Instruction...**, which will pop open a window (Figure 1.3) that prompts you to load the bytecode file.

1.5 Samcode

As you would write any program, you will write programs in Samcode as a sequence of instructions. This assignment requires that you will use many of the SaM instructions in Section 1.3.3. However, you will see many more instructions in `Instruction.java` that we do not use in this assignment. You will write programs that would occur inside a single function, such as arithmetic expressions and variable assignments, to perform basic computations.

To calculate `10+20` using SaM, you need to think of the expression in ***postfix notation*** as `10 20 +`. Postfix notation is form of notation that placed an expression's operator at the *end* of an expression. Knowing that `+` operates on two values means that you should store the first two values somewhere, such as your brain, and then add them after encountering the `+` operator. In Samcode, you would write this arithmetic operation, as follows:

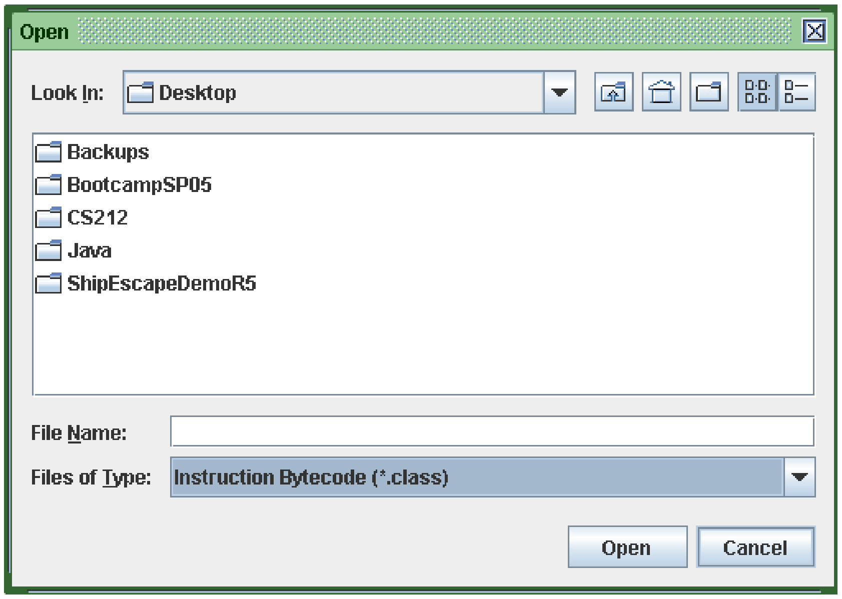


Figure 1.4: Instruction Loader

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

To help you visualize the results of pushing `10` and `20` onto the stack, refer to Section 1.1. 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 used element address. Isn't that neat?

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

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

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

1.6 Variables and Assignments

1.6.1 Compilers

For the CS212 project this semester, you will be writing a program called a ***compiler***, which translates one computer language into another. In our course, your compiler will convert a Java-like language into Samcode. For example, suppose you were trying to write a compiler to convert the following Java snippet into Samcode:

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

The next sections will demonstrate how to translate such high-level languages into Samcode.

1.6.2 Symbol Table

Writing the Samcode requires a bit of work because of the variables. You must reserve space for each variable in the stack memory because Samcode does not have explicit variables of its own. To help keep track of the source program's variables and their locations, you should manually draw a ***symbol table***, as shown in Section 1.1. 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`.

Table 1: Symbol Table

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

To follow changes in stack memory during execution, you should draw another figure to represent the stack, as shown in Figure 1.5. As your program calls functions and creates object, parts of the stack are filled with data. For now, we will assume that we are converting code from a `main` function, so we will not change the **FBR**'s initial value of zero. When we add more functions in Part 3 of the project, we will adjust the **FBR**. We will not need to worry about the **HP** until Part 4 of the project.

1.6.3 Variable Allocation

You need to allocate stack space for the source program's variables. By leaving the first cell open for the `rv`, the result of a program will have a place for SaM to store and return the value. The `main` variables are stacked on top of the first cell in the order in which they are declared. The address of a variable is its cell address on the stack.

So, to encode the declaration statements of the source language (`int x` and `int y`), you need to move **SP** "up" the stack. You have two choices:

- `ADDSP` *`slots`*: `ADDSP` will makes *`slots`* addresses available on the stack. Because these cells indicate allocated memory, they are labeled as type **M**. Previously-stored values will appear on the stack.
- `PUSHIMM` *`value`*: This instruction will allocate one memory location and initialize it with value. Typically, you should use `0` for *`value`*. This instruction will clear previously stored values on the stack. SaM already initializes each cell to `0` upon starting.

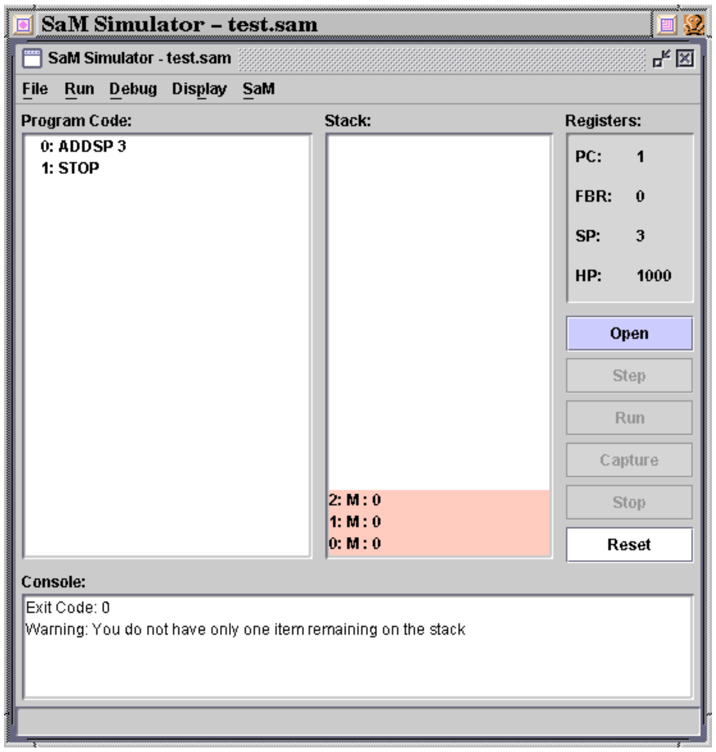
For our example, we use `ADDSP`. Since **SP** starts at **0**, you can move the stack pointer up three cells with `ADDSP 3`, which leaves space for `rv`, `x`, and `y`. **SP** will now point to the next free cell at the address of `3`. You may allocate the same amount of space use three `PUSHIMM 0` statements.

The following Samcode (`add_rel.sam`, `add_abs.sam`) allocates space for three variables:

```
ADDSP 3
// code to store and retrieve x and y values
//    will be developed later
STOP
```

The state of the stack after running is shown in Figure 1.5. There are two important ideas to note:

- Each cell is labeled with **M** because cells **0**, **1**, and **2** are addresses that you use to store and retrieve values from memory.
- **SP** points to the cell address **3**, which is above the last memory allocation **2:M:0**.



```
0: ADDSP 3
1: STOP
```

```
2: M : 0
1: M : 0
0: M : 0
```

```
PC:   1
FBR:  0
SP:   3
HP:   1000
```

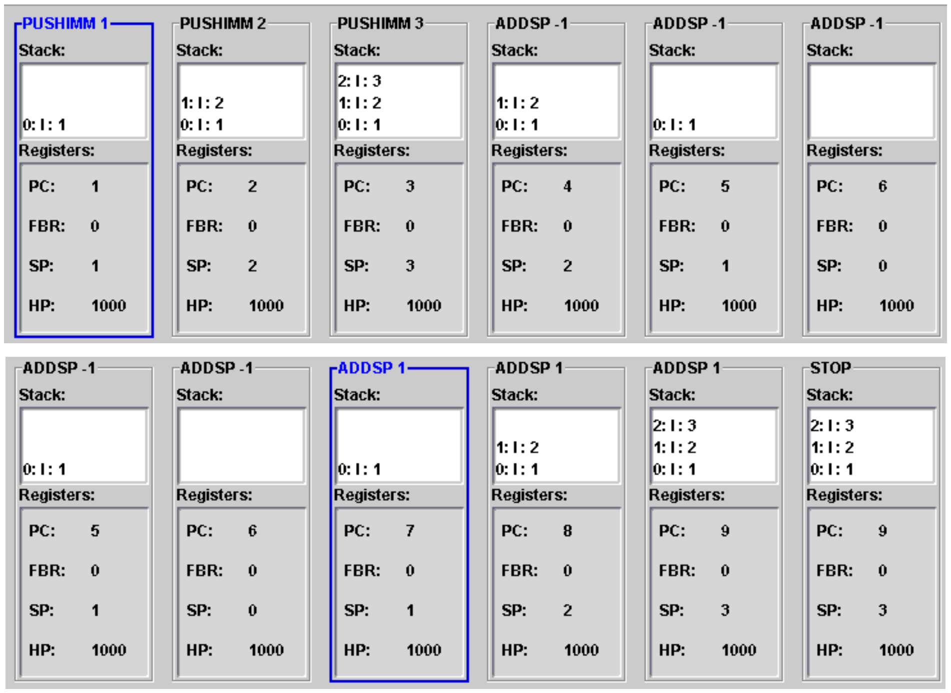
```
Exit Code: 0
Warning: You do not have only one item remaining on the stack
```

Figure 1.5: Stack Memory With Variables

If you wish to see the primary difference between approaches, step through this program, which is stored in `allocate.sam`:

```
PUSHIMM 1
PUSHIMM 2
PUSHIMM 3
ADDSP -1
ADDSP -1
ADDSP -1
ADDSP 1
ADDSP 1
ADDSP 1
STOP
```

SaM will display the initially pushed values after it executes `ADDSP 1` three times, as shown in the two **Capture** windows in Figure 1.6.



```
PUSHIMM 1
Stack:
0: I: 1
Registers:
PC:   1
FBR:  0
SP:   1
HP:   1000

PUSHIMM 2
Stack:
1: I: 2
0: I: 1
Registers:
PC:   2
FBR:  0
SP:   2
HP:   1000

PUSHIMM 3
Stack:
2: I: 3
1: I: 2
0: I: 1
Registers:
PC:   3
FBR:  0
SP:   3
HP:   1000

ADDSP -1
Stack:
1: I: 2
0: I: 1
Registers:
PC:   4
FBR:  0
SP:   2
HP:   1000

ADDSP -1
Stack:
0: I: 1
Registers:
PC:   5
FBR:  0
SP:   1
HP:   1000

ADDSP -1
Stack:
Registers:
PC:   6
FBR:  0
SP:   0
HP:   1000
```

```
ADDSP -1
Stack:
0: I: 1
Registers:
PC:   5
FBR:  0
SP:   1
HP:   1000

ADDSP -1
Stack:
Registers:
PC:   6
FBR:  0
SP:   0
HP:   1000

ADDSP 1
Stack:
0: I: 1
Registers:
PC:   7
FBR:  0
SP:   1
HP:   1000

ADDSP 1
Stack:
1: I: 2
0: I: 1
Registers:
PC:   8
FBR:  0
SP:   2
HP:   1000

ADDSP 1
Stack:
2: I: 3
1: I: 2
0: I: 1
Registers:
PC:   9
FBR:  0
SP:   3
HP:   1000

STOP
Stack:
2: I: 3
1: I: 2
0: I: 1
Registers:
PC:   9
FBR:  0
SP:   3
HP:   1000
```

Figure 1.6: Allocation and Deallocation of Memory

1.6.4 Addressing of Variables

So…how do you write Samcode for the assignment statements in the Java snippet in Section 1.6.1? Also, how do you retrieve a variable value? You have two options: ***absolute addressing*** and ***relative addressing***. In absolute addressing, you do not need to worry about the mysterious **FBR** other than you should not change its initial value of 0. When we introduce functions, you will run into trouble unless you use relative addressing to shift the **FBR** every time a new function is called. For now, you will focus on basic expressions that would be written in the body of a function.

1.6.4.5 Absolute Addressing

`STOREABS` and `PUSHABS` use an ***absolute cell address***, which means you can directly access any location on the stack without worrying about a specific function. For example, global variables use absolute addresses, as you will see later in the semester.

To store a value on the stack, use `STOREABS` ***`k`***, which takes a value from V_{top} and pushes it into location ***`k`***. In your Samcode, you would use the following sequence of instructions:

- `PUSHIMM` ***`value`***: Push the ***`value`*** you wish to store.
- `STOREABS` ***`address`***: Push ***`value`*** into ***`address`***.

For example, to store the value of `10` for `x` in address `1`, you would do the following:

```
PUSHIMM 10
STOREABS 1
```

To retrieve a value from the stack, use `PUSHABS` ***k***, which takes the address ***k*** and pushes a value from that location.

For example, to retrieve the value of `x`, you would do the following:

```
PUSHABS 1
```

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.

The following Samcode performs the operations of the Java code-snippet in Figure 1.6.1. As you read though the example, note how I methodically store and retrieve each variable value. Although I assign `rv` a “default” value, you may choose not to do so:

```
// Absolute addressing, add_abs.sam:

ADDSP 3      // allocate x,y,rv; you may also PUSHIMM 0 three times
PUSHIMM 0    // value to store in rv’s location
STOREABS 0   // store value 0 in address 0
PUSHIMM 10   // value to give x
STOREABS 1   // store value of x
PUSHIMM 20   // value to give y
STOREABS 2   // store value of y
PUSHABS 1    // retrieve value of x
PUSHABS 2    // retrieve value of y
ADD          // add values of x and y
STOREABS 0   // store value of x+y in address rv
ADDSP -2     // remove x and y from memory
STOP         // halt --> SaM 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.

1.6.4.6 Relative Addressing

By using the **FBR** to keep track of your current method call, you can keep your Samcode very general. Since we are assuming no other method than our “main,” **FBR** stays at `0`. Judicious use of the `STOREOFF` ***x*** command provides the best way to manage 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:

```
// Relative addressing, add_rel.sam:

ADDSP 3      // allocate 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 30
```

Note how 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 is not an issue for Part 1.