Name: McKenzie Eshleman

Partner(s): Jon Robinson (in company), Caitlynn Stringer (in class partner)

Section: 3321

**Objective:** In this lab, you will apply what you have learned about how modern systems break up the complex task of code compilation by using the intermediate step of implementing a stack-based **Virtual Machine (VM)**. You will first demonstrate an understanding of the VM operations, then implement those operations on the Hack computer using Hack Assembly routines. To receive full credit, properly commented code will be required (Hack Assembly comments begin with “//”).

1. **Pre-Lab: SimpleAdd VM** 
   1. The **SimpleAdd.vm** program consists of 3 basic instructions. The first two push a number from the “**constant**” segment (**constant** is a virtual memory segment) onto the **stack** segment of memory. The final instruction takes those top two numbers from the stack, performs addition with them, and then returns the sum to the stack. Based on the initial **RAM** state shown below, show how each instruction affects the system.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Address** | **RAM** | | **0 (SP)** | 300 | | **1** |  | | **2** |  | |  | … | |  |  | | **300 ->** |  | | **301** |  | | **302** |  | | **303** |  | |

* 1. Update the **RAM** diagram following the instruction: **push constant 11**

NOTE: Make sure an arrow (->) indicates the location referenced by the **Stack Pointer (SP)** as shown in part a. (5 pts)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Address** | **RAM** | | **0 (SP)** | 0 | | **1** |  | | **2** |  | |  | … | |  |  | | **300** | 11 | | **301-->** |  | | **302** |  | | **303** |  | |

* 1. Update the **RAM** diagram following the instruction: **push constant 17**

NOTE: Make sure an arrow (->) indicates the location referenced by the **Stack Pointer (SP)** as shown in part a. (5 pts)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Address** | **RAM** | | **0 (SP)** |  | | **1** |  | | **2** |  | |  | … | |  |  | | **300** | 11 | | **301** | 17 | | **302🡪** |  | | **303** |  | |

* 1. Update the **RAM** diagram following the instruction: **add**

NOTE: Make sure an arrow (->) indicates the location referenced by the **Stack Pointer (SP)** as shown in part a. (5 pts)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Address** | **RAM** | | **0 (SP)** |  | | **1** |  | | **2** |  | |  | … | |  |  | | **300** | 28 | | **301🡪** | 17 | | **302//** |  | | **303** |  | |

* 1. Open the **VMEmulator** from the nand2tetris tools directory. Load the test script **SimpleAddVME.tst**. This VM program for the project is very similar to what we did in the previous steps, albeit with different values and addresses. Step through the 3 instructions paying close attention to how the “**Stack**”, “**Global** **Stack**”, and “**RAM**” windows change. Discuss the final state of the “**Global** **Stack**”. Where is the final total? What happened to the 7? Why is the 8 still there? (10 pts)

The final state of the global stack is for RAM[256] is the value of 15 which is 7 + 8, the second location in the stack RAM[257] is the value of 8. The final total is the first position in our stack which is RAM[256]. The seven is overwritten from the stack by the addition total, the seven is not in the global stack but it remains in the stack. The 8 is still in our global stack because that position was not overwritten.

* 1. Include a screenshot below of the full Virtual Machine Emulator window after it completes the **SimpleAddVME.tst** test script. (5 pts)

|  |
| --- |
|  |

1. **SimpleAdd: push constant X**

**Purpose:** Describe the function and purpose of a **push** instruction. (5 pts)

The push function pushes the value of segment [index] onto the stack.

What is unique about pushing a value from the **constant** segment of memory?

When pushing a constant segment of memory the memory is not store it simply just exist to be used.

What type of memory addressing does **push** **constant** **X** use to retrieve the constant? Make your choice **bold**.

**Immediate** / Direct / Indirect

**You now need to write a Hack Assembly routine that will manipulate the values in RAM as you showed in steps b and c of the Pre-Lab. This sequence of Assembly instructions will execute whenever a *push constant X* command is called. We use X here to represent any valid integer value.**

1. Write a snippet of Hack Assembly to move the constant represented by X into the D Register (you may, for now, act as if X is an actual number to simplify this code). (3 pts)

|  |
| --- |
| **@7 //Constant variable**  **D=A //puts constant 7 into varable** |

1. Write a snippet of Hack Assembly to increment the value of the **Stack Pointer (SP)**. (3 pts)

|  |
| --- |
| **@SP //address of stack pointer** |

1. Write a snippet of Hack Assembly to set the A Register to the **RAM** address of the location within the **Stack** which will be assigned the constant value X. (3 pts)

|  |
| --- |
| **A = M //putting pointer into memory address** |

1. Write a snippet of Hack Assembly to assign the X value (held in the D Register) to this location at the head of the **Stack**. (3 pts)

|  |
| --- |
| **M = D //makes D register equal to the location of the head of the stack** |

1. Combine these snippets of Hack Assembly code into a series of lines of code which will be written to the **SimpleAdd.asm** file you create whenever a **push constant X** instruction appears in the **SimpleAdd.vm** program. Recall that your Python Translator script is responsible for performing this **VM->ASM** code translation (just as project 06 performed an **ASM->HACK** translation). Add code to the **generateMemoryAccess**() function that will, if the command is a **push** from the **constant** segment, return the string(s) of Assembly code which will be written during the following call: **generateMemoryAccess**(“push”, ”constant”, X ), where X will be a valid integer value. Paste your updated **generateMemoryAccess**() function below**.** (3 pts)

|  |
| --- |
| **@7 //Constant variable**  **D=A //puts constant 7 into varable**  **@SP //address of stack pointer**  **A = M //putting pointer into memory address**  **M = D //makes D register equal to the location of the head of the stack**  **@SP //address of stack pointer**  **M=M+1 //increments the Ram address by 1** |

1. **SimpleAdd: add**

**Purpose:** Describe the function and purpose of an **add** instruction. (5 pts)

The function of the add instruction is to add two integer together. The purpose of the instruction is add the integers and pop the return value after the operands.

**You now need to write a Hack Assembly routine that will manipulate the values in RAM as you showed in step d of the Pre-Lab. This sequence of Assembly instructions will execute whenever an add command is called.**

1. Write a snippet of Hack Assembly to assign the address of the head element (last element in) of the **stack** to the A Register. (3 pts)

|  |
| --- |
| **@SP //assigning the address of the head element**  **AM=M-1 //places addition variable above the last item in the stack** |

1. Write a snippet of Hack Assembly to store the head element of the **stack** to the D Register. (3 pts)

|  |
| --- |
| **D = M //stores head element into the D register** |

1. Write a snippet of Hack Assembly to set the A Register to the **RAM** address of the next element within the **stack**, then add the value in the D Register to this element. (3 pts)

|  |
| --- |
| **@SP //variable that is located at SP**  **AM=M-1 //going up 1 from the pointer to save the varaible**  **A=M //storing the memory location of the pointer**  **D=A+D //adding the two variable together and stores as the D register** |

1. Write a snippet of Hack Assembly to decrement the value of the **Stack Pointer (SP)**. (3 pts)

|  |
| --- |
| **@SP //location of the pointer**  **A=M //stores location of pointer into RAM**  **M=D //Storing the register D (Sum varaible) into a Ram address**  **//one above our pointer** |

1. Combine these snippets of Hack Assembly code into a series of lines of code which will be written to the **SimpleAdd.asm** file you create whenever an **add** instruction appears in the **SimpleAdd.vm** program. Add code to the **generateArithmetic**() function that will, if the command is **add**, return the string(s) of Assembly code which will be written to implement that command. Paste your updated **generateArithmetic**() function below**.** (3 pts)

|  |
| --- |
| **@SP //assigning the address of the head element**  **AM=M-1 //places addition variable above the last item in the stack**  **D=M //stores head element into the D register**  **@SP //variable that is located at SP**  **AM=M-1 //going up 1 from the pointer to save the varaible**  **A=M //storing the memory location of the pointer**  **D=A+D //adding the two variable together and stores as the D register**    **@SP //location of the pointer**  **A=M //stores location of pointer into RAM**  **M=D //Storing the register D (Sum varaible) into a Ram address**  **//one above our pointer**    **@SP //loads pointer into A**  **M=M+1 //incrementing the pointer to the next RAM location by one** |

1. **Testing SimpleAdd**

Save your Python **VMTranslator.py** script and exit the text editor. Make your script executable. Now navigate to the project07/StackArithmetic/SimpleAdd directory. Make a symbolic link (**ln -s**) to your Python script and then run the command below.

|  |
| --- |
| **./VMTranslator.py SimpleAdd.vm** |

Resolve any errors generated while executing your code. You will need to fill in more setup code than described here to create a viable script just as you did in Lab 06, refer there for File I/O as well as general interfacing with the **parser** and **code** functions.

Once the errors are resolved and your program runs successfully, run the “**ls -l**” command and verify that the “**SimpleAdd.asm**” file was properly created. Examine the output file for any obvious errors with the “**cat SimpleAdd.asm**” command.

Now open the CPU Emulator from the nand2tetris **tools** directory. Run the **SimpleAdd.tst** test script and include a screenshot below showing whether the test succeeded. (10 pts)

|  |
| --- |
|  |

1. **Several popular programming languages use a virtual machine to simplify compiler writing and to allow for cross-platform source code. One example is Java and its** [**JVM**](https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings)**. Show what instructions the SimpleAdd.vm program would contain if it were, instead, written in the JVM. Compare these instructions and the approach to the provided Jack VM code. Which VM language is easier to use? Which language more precisely specifies what the physical machine is doing?** (10 pts) The JVM instruction that would best suit our SimpleAdd.vm program would be iadd. This instruction adds two integers together, the JVM also has other instructions to add different kinds of variables like floats for example. I am not too sure which is easier to use in my experience I would say the SimpleAdd only because that is what I am most familiar with, however the JVM seems to be more precise when specifying what the physical machine is doing.
2. **Explain how a** [**virtual machine**](https://www.blackhat.com/presentations/bh-asia-02/LSD/bh-asia-02-lsd.pdf) **that translates an intermediate language to target architecture assembly language increases the security of the resulting program and how it can potentially introduce new vulnerabilities.** (10 pts) JVM security architecture becomes the first line of defense against malicious java codes, and helps protect from spoofing attacks that can take place. It also has a system that will target classes that are “bogus.” There can also be new vulnerabilities that are found in every coding language, java has a field of continuous research being done to search for any new vulnerabilities as well as general attack techniques.