Project Report 07

Virtual Machine I

Name: MIDN 2/C Eshleman

Course / Section: SY303 / 3321

Enclosures: (1) Python Implementation of VMTranslator.py
(2) Screenshot for CPU Emulator comparison of SimpleAdd.asm
(3) Screenshot for CPU Emulator comparison of StackTest.asm
(4) Screenshot for CPU Emulator comparison of BasicTest.asm
(5) Screenshot for CPU Emulator comparison of PointerTest.asm
(6) Screenshot for CPU Emulator comparison of StaticTest.asm

RUBRIC

Section	0%	50%	80%	100%	Max Points
Used Template	No	Partially		Yes	10
Introduction	Purpose and objectives unclear	Discussion lacking	Progressing	Well discussed	5
Procedure	Discussed procedure could not be reproduced	Some steps and tools missing	Progressing	Comprehensive and clear	10
Results	Not present or not discussed	Incorrect results without explanation	Progressing	Correct, well-explained results	10
Discussion	Questions not answered or answers are off-topic.	Discussions lack complete consideration. Terse responses.	Progressing	Well-developed discussions with interesting insights	30
Python Code	Missing or not commented	Incomplete, has errors, missing name or comments	Progressing	Includes identification and clear descriptions of all code	20
CPU Emulator Screenshots	Missing	Does not capture full simulator screen or wrong	Progressing	All present, shows status of test script, full legible screen capture	10
Grammar/Professionalism	Poor grammar or use of slang		Progressing	Professional writing	5

Project Report 07

Virtual Machine I

INTRODUCTION:

The main objective for project 7 is to build the first part of the VM translator that was shown in chapter 7. We are trying to build the first portion of the VM translator focusing on the implementation of the stack arithmetic and memory access commands of the VM language. The project is trying to demonstrate that we can successfully build the translator using python, the VM translator is designed to generate Hack Assembly Code.

PROCEDURE:

For this project I followed along with Dr. Browns walk through video to fully grasp and understand the concepts that we were trying to program. The initial portion of the code is the main function, this is where we first open a file from the command line argument and read and write the contents of that file into a new file that is an .asm. We then read each line of instruction and determine whether it is an instruction or if it is a comment and or not an instruction.

For the parser function we take our list of instructions and we come to a two way street for example. We have to deter which instruction will go to either Memory Access or Arithmetic instructions. We determine this by checking the line of instruction and seeing if there is either a "push" or "pop" in that particular line. If there is then the instruction line is separated to Memory Access. If neither of the key words are in the line of instruction then that line is a Arithmetic code. We return the tuple of instType and instList.

The code function uses the tuple return from the parser function and then again like the parser sends the different codes to their respective generate function. If the line of instruction is Arithmetic then we generate the hack assembly instruction to be implemented in the VM Instruction. If the line of instruction is Memory Access then we generate the hack assembly instructions into VM instructions of Push and Pop. Then the function returns the hack assembly code with complies with the requested VM instruction.

The Generate Arithmetic function takes in a single command line and corresponds the command to the hack machine code. The Generate Memory Access does the same thing but it is broken down into two main portions, the push and the pop. If the command has push in it then it will push the data onto the stack and move down the pointer. If the command has a pop then the stack will remove the last item and move the pointer.

RESULTS:

Succeeded	Failed
• SimpleAdd.asm • StackTest.asm • BasicTest.asm	• StaticTest.asm

Project Report 07

Virtual Machine I

• PointerTest.asm •	

If you were successful, describe an issue or a confusing concept you had to overcome to get this to work.

One confusing topic I ran into during this project was the global variable uniqueNum, I had initialized it at the beginning of the function but did not put the value in the beginning of the code. After having my code not work for a while I finally figured out my errors and then got it to work.

DISCUSSION (For these questions you may seek quality information from the internet or other resources. Be sure to cite all sources used for research, give the exact URL(s) visited):

1. **Discuss the decision a computer architect can make to implement a function like multiplication at one of various levels (from Hardware in ALU, to Assembly, to Virtual Machine, to High-Level language). What might influence their choice in this matter?** (10 pts) An implementation of multiplication that a computer architect can use is booth's algorithm. It uses binary integers to represent in an efficient way. It uses the examination of the multiplier bits and shifting the product. The algorithm uses ALU hardware by using registers and then is translated into assembly code.

SOURCE: https://www.geeksforgeeks.org/computer-organization-booths-algorithm/

2. **How does the concept of a Virtual Machine, as introduced in this chapter, compare to and differ from the Virtual Machine that you use to run Linux on your PC (VMWare or VirtualBox)?** (5 pts)
A Virtual Machine like VMWare is stored on a host computer in a set of files. The VM we used in this project is a single file of instructions while the VM computer is sets of many files complied together to make a Linux machine.

SOURCE: https://www.vmware.com/support/ws55/doc/ws_learning_files_in_a_vm.html

3. **a) As we've discussed, *many* programming languages us a Virtual Machine to process their high-level code efficiently, including Python! Check out the C style implementation of `mult` at the top of pg. 136. Open your Python3 interpreter in the terminal (just type "`python3`") and define a `mult` function which is implemented the same way, make minor adjustments to switch to Python3 syntax where necessary, but keep the same variable names and program flow. Test your `mult` function with a couple examples to make sure it is correct (eg. `mult(5,7)` and `mult(12, 0)`). Then, import the Python disassembly library "`import dis`" and have it display your `mult` function Virtual Machine bytecode, "`dis.dis(mult)`". Paste both routines below.** (5 pts)

Python3 mult	mult bytecode

Project Report 07

Virtual Machine I

<pre>def mult(x,y):
   product = x * y
print(product)</pre>
	Def
	
	Could not get this to work

b) <u>Fully</u> explain what is happening in the Virtual Machine bytecode to the best of your ability. Don't forget to discuss how `push` and `pop` are handled here, what are they called in the Python VM? These resources (and others you may find) can help: (10 pts)

CPython uses a stack-based virtual machine, it is oriented entirely around stack data structures where you can push an item onto the top of the structure or pop an item off of the top. CPython uses three different kinds of stacks, the first is the call stack where the main structure of the running python program. The second stack is an evaluation stack(data stack) this is where the execution of a python function occurs and pushing and popping things into the stack. The third stack is a block stack where python keeps track of the control structures like loops.

- **Python Bytecode**
- **Disassembly Module**

COMMENTS:

What did YOU gain from this project? If you felt like the project did not add to your understanding leave the feedback below. Your help is critical to improving the course!

I felt like this project did not have a very good explanation of what was required of us, we have to try to figure it out on our own. It is helpful when the professor goes through what exactly is needed for us and what we need to execute.

Project Report 07

Virtual Machine I

ENCLOSURE (1): Python Implementation

```
#!/usr/bin/env python3

'''
Name: MIDN McKenzie Eshleman
Course: SY303
Section: 3321
Description: <Describe the program here>
Usage:
  $ ./VMtranslator.py <program.vm>

Tips:
# See Python Software Foundation, Python 3 Documentation (URL:
https://docs.python.org/3/ ):
# - open()
# - str.strip()
# - str.format()
# - string concatenation
# You may want to consider incorporating some of these functions / data structures
'''
import sys
#Global variable
uniqueNum = 0

def main():
	'''
	Description: main Function (main control flow)
	Arguments:
	  ...
	'''
	# Read filename from command line argument
	filename = sys.argv[1]
	# Open the input .vm file with that filename
	fin = open(filename, 'r')
	# Create and open the output .asm file with the same basename
	basename = filename.split('.')[0]
	fout = open(basename + '.asm' ,'w')
	# Read each VM instruction in the input file
	# For each instruction:
	for instruction in fin.readlines(): #reads each line of instruction
```

```
			instruction = instruction.strip()
			if not instruction: #if not an instruction then we ignore
				continue
			if "//" in instruction: #if line is a comment we ignore
				continue
			else:
				instruction = instruction.split("//")[0].strip()
			# Parse the instruction into its type (eg. Arithmetic, Push, Pop) and
arguments and return them in a tuple
			instruction_type, instruction_arguments = parser(instruction)
			# Generate the corresponding Hack Assembly code of that parsed instruction
as a string
			assembly_code = code(instruction_type, instruction_arguments)
			# Write the assembly string(s) to the output file
			fout.write(assembly_code) #assembly_code + '\n'
		# Be nice and close the files when you are done!
		for f in (fin,fout):
			f.close()

def parser(inst):
		'''
		Description: Checks to see if the line of instruction is either
		Memory Access or Arithmetic code. Then it is sent through the code
function.
		Arguments: inst
		  ...
		Returns:instType and instList
		  ...
		'''
		# Your code here
		instList = inst.split() #splits the lines of code
		if instList[0] in ('push', 'pop'): #if there is a push or pop then the line is memory
access
			instType = "MemoryAccess"
		else: #if not push or pop then the line is arithmetic
			instType = "Arithmetic"
		# Return a tuple containing the instruction type and the arguments (arg1, arg2)
		return(instType, instList) #returing the instType and inst List


def code(instType, instArgs):
		'''
		Description: Takes the two different kinds of instruction
```

```
	and sends them to their respective generate functions.
	Arguments: instType, instArgs
	  ...
	Returns: assembly
	  ...
	'''
	# Determine the class of code that must be generated based on the type, then
create it in an appropriate function
	if instType == "Arithmetic":
		# Need to generate Hack Assembly instructions to implement an Arithmetic
or Logical VM instruction (suggested arguments included)
		[command] = instArgs
		assembly = generateArithmetic(command)
	else:
		# Need to generate Hack Assembly instructions to implement a Memory
Access VM instruction (push, pop) (suggested arguments included)
		command, segment, index = instArgs
		assembly = generateMemoryAccess(command, segment, index)
	...
	# Return the line(s) of Hack Assembly code which accomplish the requested VM
instruction
	return assembly

def generateArithmetic(command):
	'''
	Description: Takes a single command and corresponds
	the command to that specific hack machine code.
	Arguments: command
	  ...
	Returns: acode
	  ...
	'''
	# Your code here
	global uniqueNum #declaring the unique number
	acode = '\n'
	acode += "@SP\t//Performing Arithmetic\n"
	acode += "A=M-1\n"
	if (command == "not"):
		acode += "M=!M\t//not\n" #Not Hack machine code
	elif (command == "neg"):
		acode += "M=-M\t//negation\n" #Neg hack machine code
	else: #Must be a binary operation
		acode += "D=M\n"
```

```
                acode += "A=A-1\n"
                if (command == "add"):
                        acode += "M=D+M\t// add\n" #machine code for adding
                elif (command == "sub"):
                        acode += "M=M-D\t// sub\n" #machine code for subtracting
                elif (command == "and"):
                        acode += "M=D&M\t// and\n" #machine code for and
                elif (command == "or"):
                        acode += "M=D|M\t// or\n" #machine code for or
                else: #Jump hack machine codes
                        acode += "D=M-D\n"
                        acode += "@CMP" + str(uniqueNum) + "\n"
                        if (command == "eq"):
                                acode += "D;JEQ\t// equality\n" #hack code for the Jump
equal to
                        if (command == "gt"):
                                acode += "D;JGT\t// greater than\n" #hack code for the Jump
greater than
                        if (command == "lt"):
                                acode += "D;JLT\t// less than\n" #hack code for the Jump less
than
                        acode += "D=0\n"
                        acode += "@FINCMP" + str(uniqueNum) + "\n"
                        acode += "0;JMP\n"
                        acode += "(CMP" + str(uniqueNum) + ")\n"
                        acode += "D=-1\n"
                        acode += "(FINCMP" + str(uniqueNum) + ")\n"
                        acode += "@SP\n"
                        acode += "A=M-1\n"
                        acode += "A=A-1\n"
                        acode += "M=D\n"
                        uniqueNum += 1
                acode += "@SP\n" #pointer location
                acode += "M=M-1\n"
        # Return a representation of Hack Assembly instruction(s) which implement the
passed in command
        return(acode)

def generateMemoryAccess(command, segment, index):
        '''
        Description: Broken down by two portions

        Push:push the data onto the stack and moves the pointer down
```

```
        Pop:Pop the latest item on the stack and moves the pointer up
        Arguments: command, segment, index
          ...
        Returns: mcode
          ...
        '''
        # Your code here
        mcode = "\n"
        mcode += "// " + command + " " + segment + " " + index + "\n" #Makes mcode the
three inputs
        argDict = {"local" : "LCL", "argument":"ARG", "this": "THIS", "that": "THAT",
"static":"16"}
        #creates a argument dictionary to reference

        #push command
        if (command == "push"):
                if(segment == "constant"): #checks if the code is a constant
                        mcode += "@"+index+"\n"
                        mcode += "D=A\n"#makes constant = to D
                if(segment in argDict): #checks to see if the code is in the arg dict
                        mcode += "@"+index+"\n"
                        mcode += "D=A\n"#makes index = to D
                        mcode += "@"+argDict[segment]+"\n"
                        mcode += "A=D+M\n"
                        mcode += "D=M\n"#sets the memory address of the segment = to D
                if(segment == "temp"): #checks if code it a temp
                        mcode += "@"+str(int(index)+5)+"\n"
                        mcode += "D=M\n" #makes the memory of the temp + 5 equal to D
                if(segment == "pointer"): #checks to see if the code is a pointer
                        mcode += "@"+str(int(index) +3) +"\n"
                        mcode += "D=M\n" #makes the pointer + 3 = to D
                if(segment == "static"):
                        mcode += "@static_" + index + "\nD=A\n" + "D=M\n" + "@SP\n" +
"A=M\n" + "M=D\n" + "@SP\n" + "M=M+1\n"
                        return mcode

                #ALWAYS OCCURS DURING PUSH
                mcode += "@SP\n"
                mcode += "M=M+1\n"
                mcode += "A=M-1\n"
                mcode += "M=D\n"
        #POP Command
```

Virtual Machine I

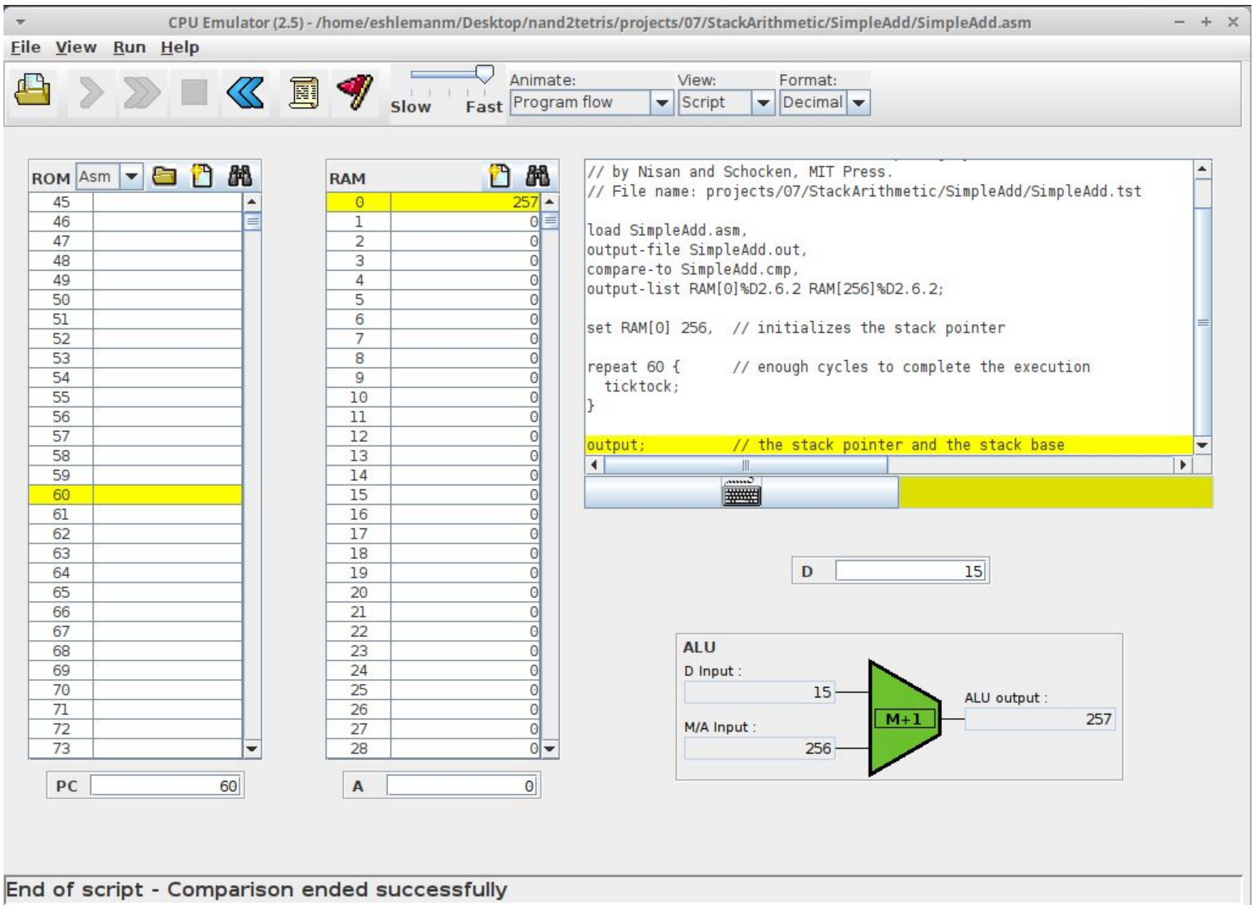
```
        else: #must be a pop command
                if(segment in argDict):#code in the arg Dict
                        mcode += "@"+index+"\n"
                        mcode += "D=A\n"#sets D equal to the index
                        mcode += "@" + argDict[segment] + "\n"
                        mcode += "D=D+M\n" #D is stored as the segment in arg Dict
                        mcode += "@R13\n" #D is stored in RAM[13]
                        mcode += "M=D\n" #Memory is stored as D
                mcode += "@SP\n" #Pointer
                mcode += "AM=M-1\n" #Incrementor
                mcode += "D=M\n"
                if(segment in argDict): #arg Dict portion
                        mcode += "@R13\n"
                        mcode += "A=M\n" #memory of RAM[13]
                if(segment == "temp"): #if the code is a temp
                        mcode += "@"+str(int(index)+5)+"\n"
                if(segment == "pointer"): #if the code is a pointer
                        mcode += "@"+str(int(index)+3)+"\n"
                mcode += "M=D\n"
                if(segment == "static" ):
                        mcode += "@SP\n" + "AM=M-1\n" + "D=M\n" + "@static_" + index +
"\nM=D\n"
                        return mcode


# Return a representation of Hack Assembly instruction(s) which implement the passed in
command
        return mcode

###############################################################################
###########################
# Define any additional helper functions here (include function descriptions)
###############################################################################
###########################


# Include code below to ONLY call the main function when the program is run from the
command line; i.e. a standalone program
if __name__ == "__main__": #calls the main functions
        main()
```

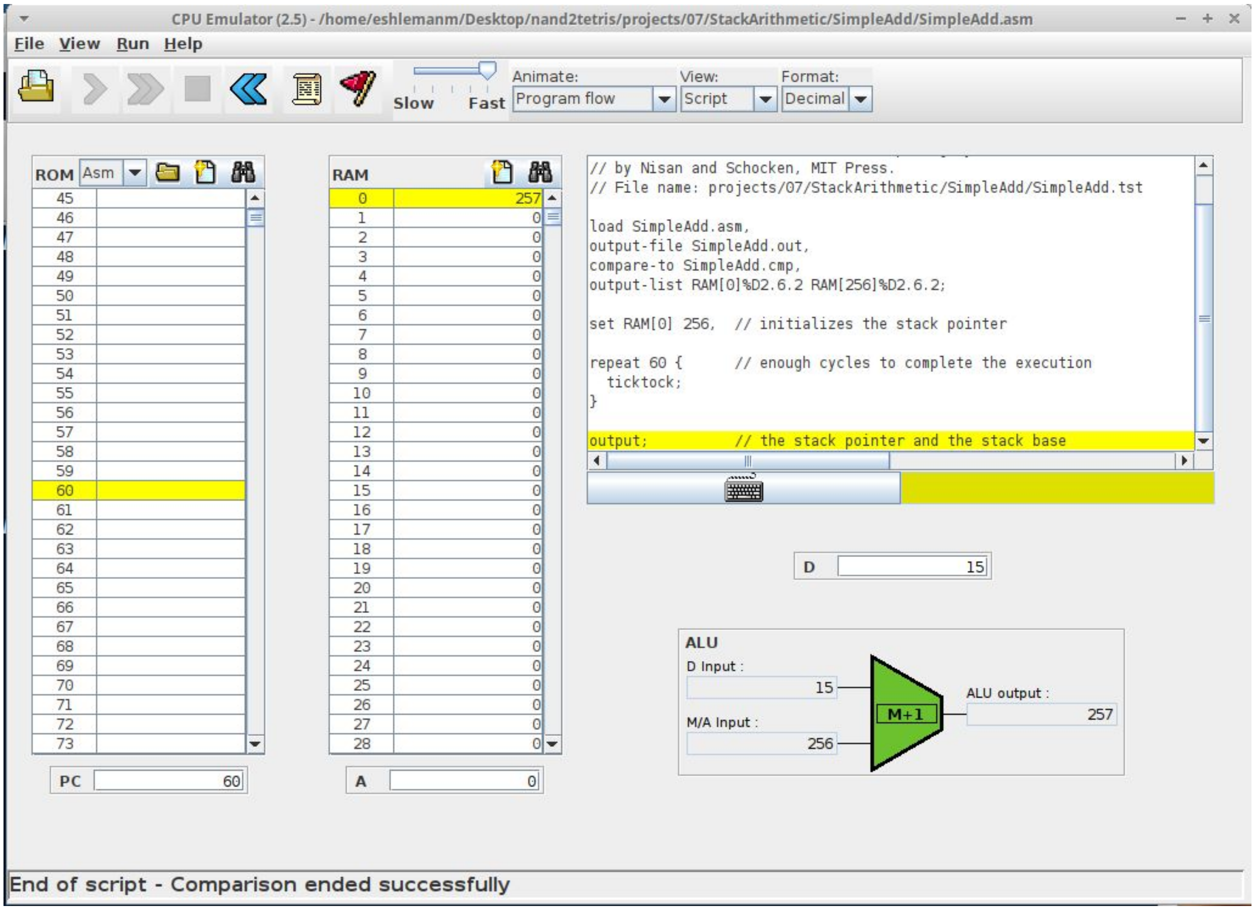
ENCLOSURE (2): CPU Emulator comparison of SimpleAdd.asm



ENCLOSURE (3): CPU Emulator comparison of StackTest.asm

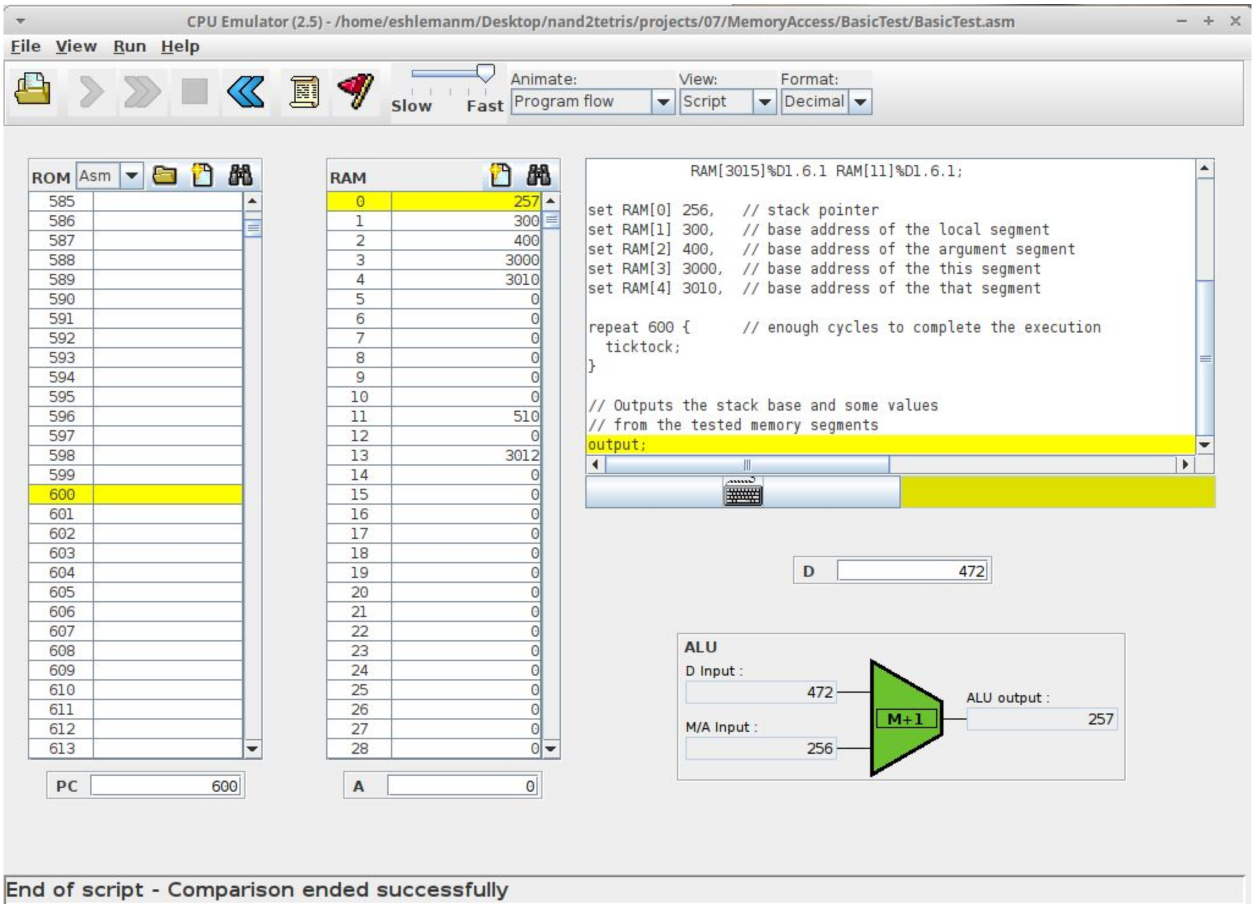
Project Report 07

Virtual Machine I

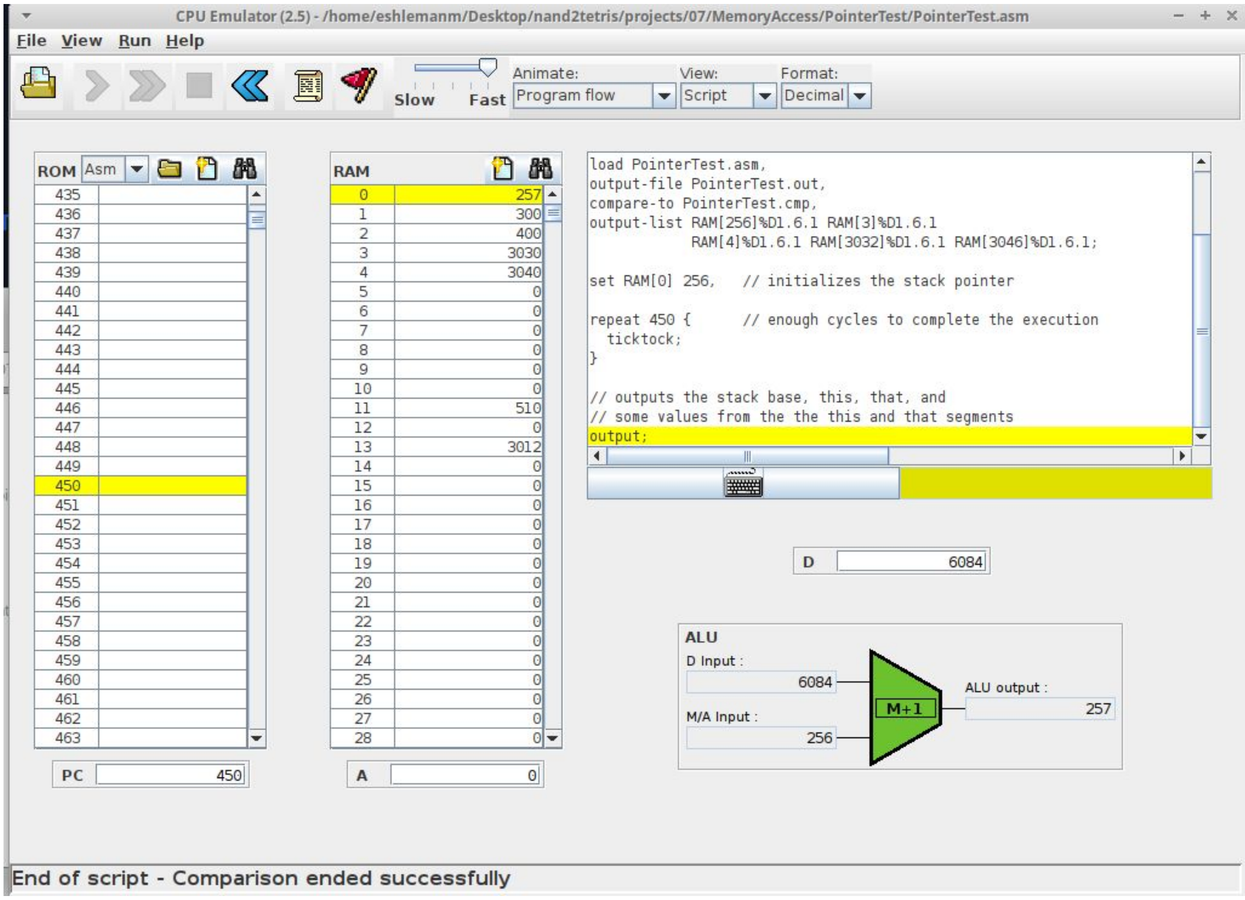


ENCLOSURE (4): CPU Emulator comparison of BasicTest.asm

Virtual Machine I



ENCLOSURE (5): CPU Emulator comparison of PointerTest.asm



ENCLOSURE (6): CPU Emulator comparison of StaticTest.asm

Project Report 07

Virtual Machine I