ELEMENTS OF COMPUTING

GROUP ASSIGNMENT-5

Contributed By:-

BATCH-A	TEAM-7
GAJULA SRI VATSANKA	CB.EN.U4AIE.21010
GUNNAM HIMAMSH	CB.EN.U4AIE.21014
M.PRASANNA TEJA	CB.EN.U4AIE.21035
VIKHYAT BANSAL	CB.EN.U4AIE.21076

1. Locate the following VM programs in the nand2teris folder **[CO2]**

- **SimpleAdd**
- **StackTest**
- **BasicTest**
- **PointerTest**
- **StaticTest**

a) Check the correctness of the programs using the VM emulator and supplied test script. Comprehend the lines of codes. (You may be asked to explain the lines of codes).
b) Write a VM-to-Hack translator in python or Java, conforming to the '*Standard VM-on-Hack Mapping*'
c) Use your VM translator to translate the above *.vm* programs to *.asm* programs
d) Execute the generated *.asm* programs on the CPU emulator using the supplied test script.

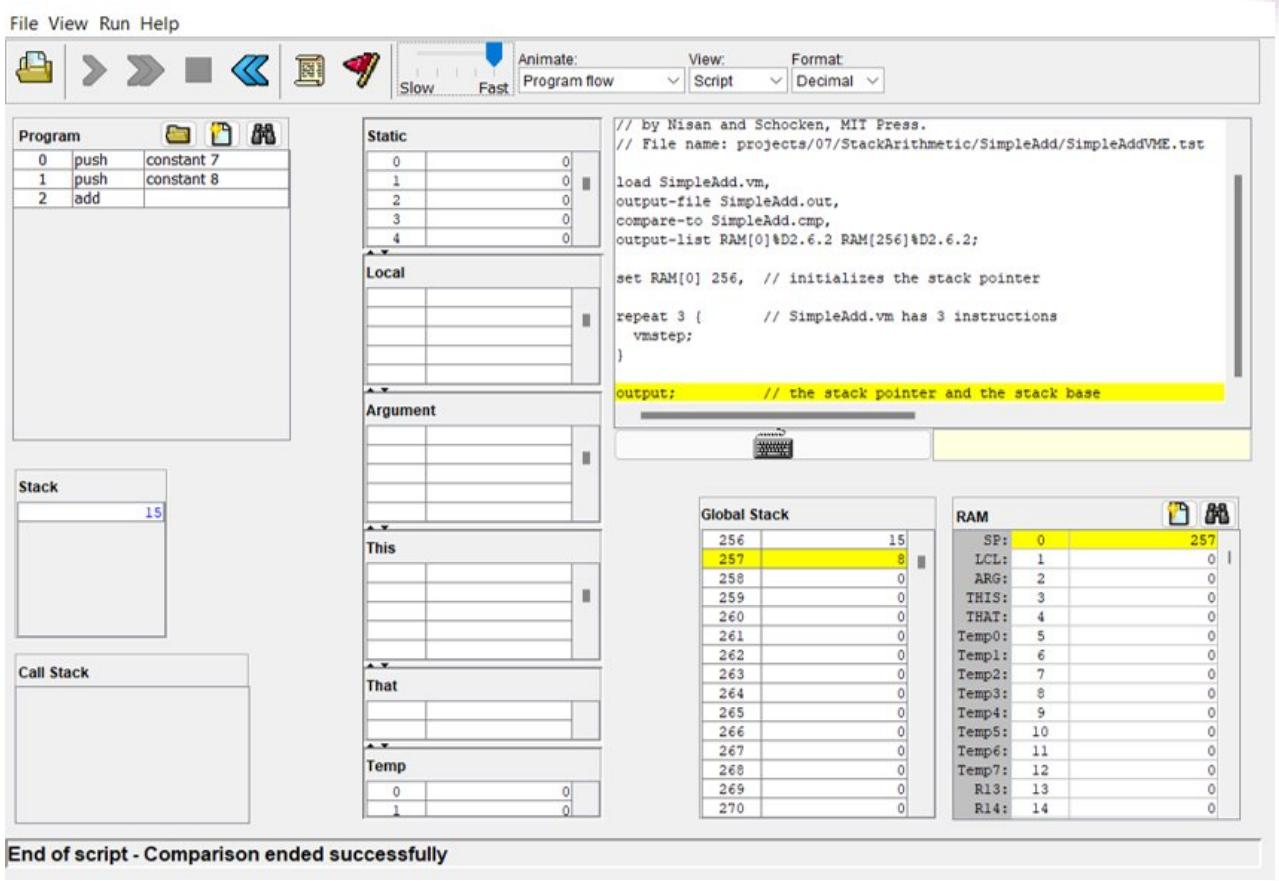
M.Prasanna Teja ,G.sri vatsanka, Vikhyat Bansal

1

PART-A

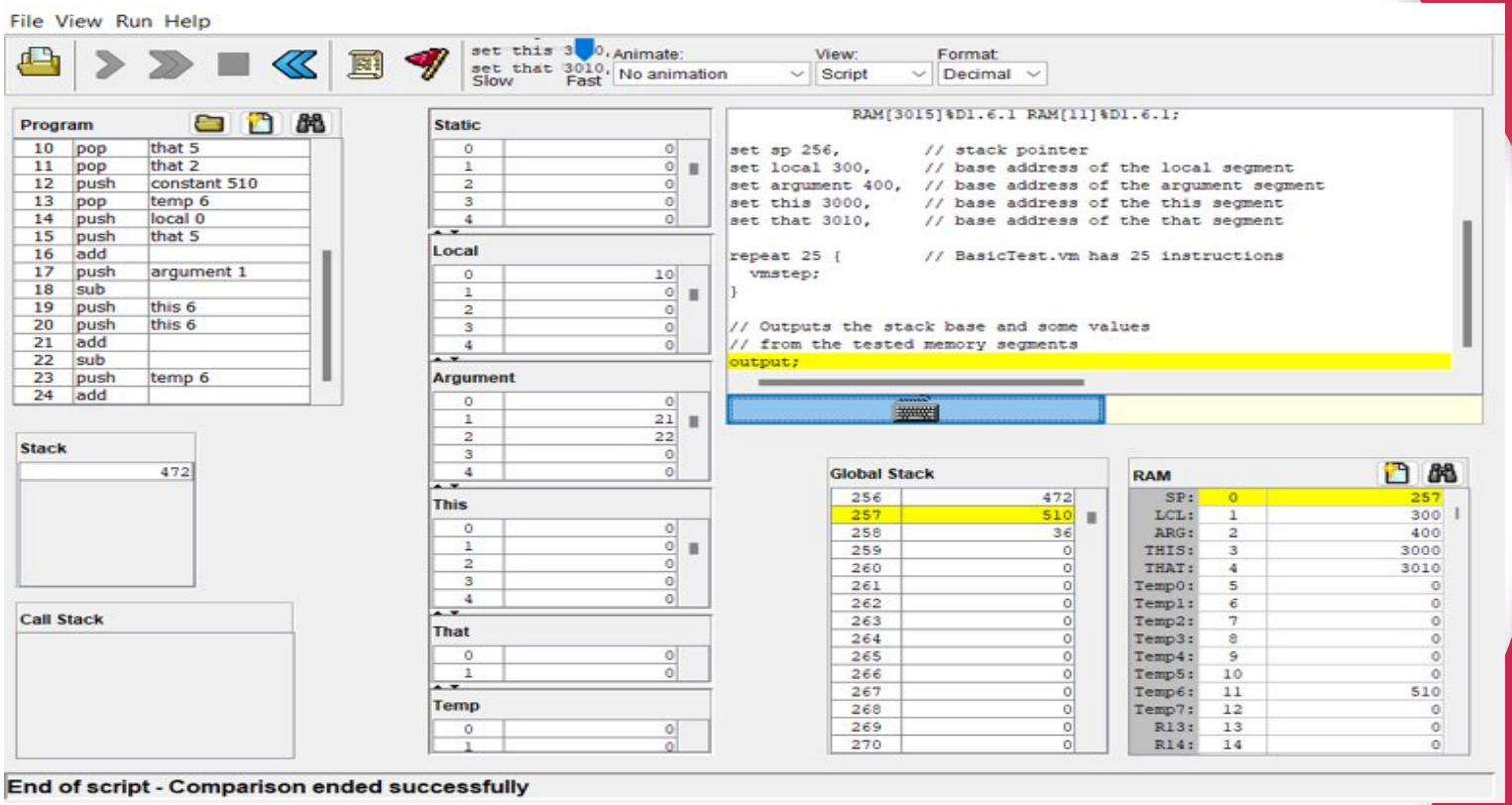
Check the correctness of the programs using the VM emulator and supplied test script. Comprehend the lines of codes. (You may be asked to explain the lines of codes).

Simple add



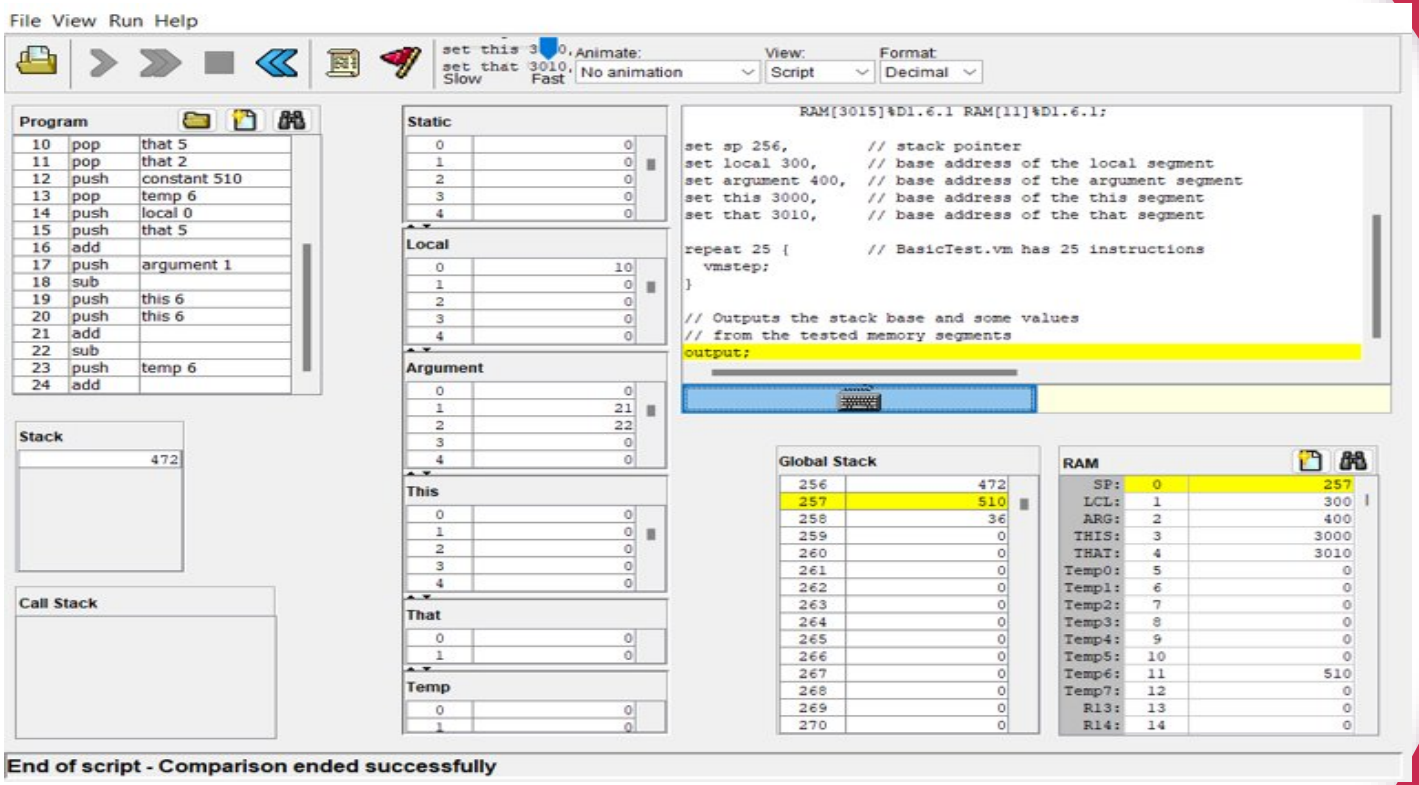
VM code is located: nand2tetris→ projects→07→Stack Arithmetic

Stack test



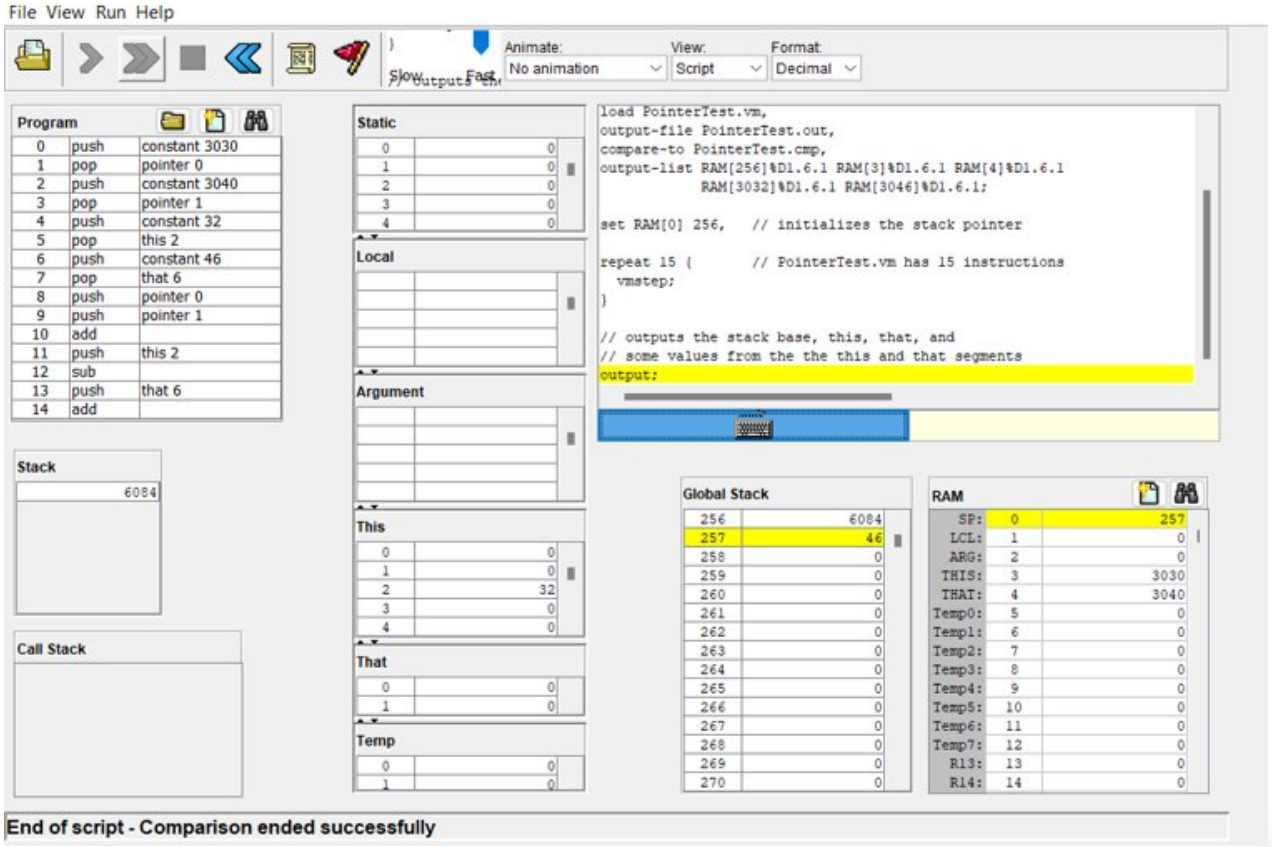
VM code is located: nand2tetris→ projects→07→Stack Arithmetic

Basic test



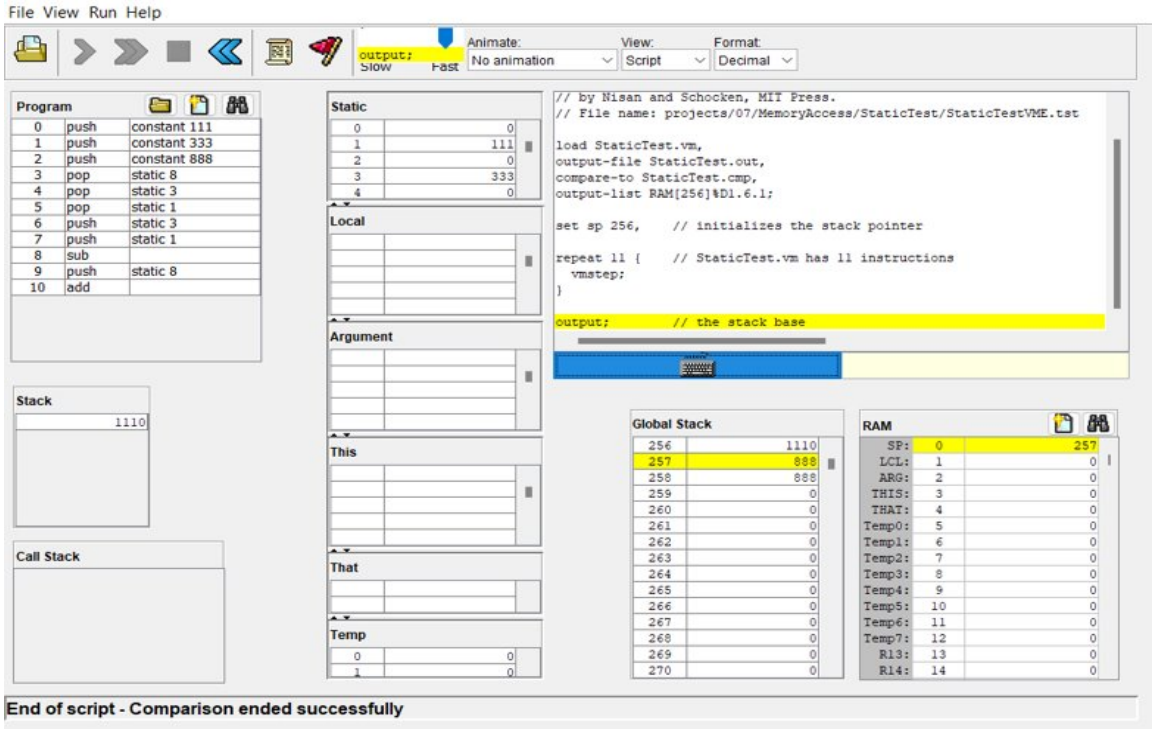
VM code is located: nand2tetris→ projects→07→MemoryAccess

Pointer test



VM code is located: nand2tetris→ projects→07→MemoryAccess

Static test



VM code is located: nand2tetris→ projects→07→MemoryAccess

PART-B

Write a VM-to-Hack translator in python or Java, conforming to the ‘Standard VM-on-Hack Mapping’

B) we have used python program to convert VM to hack

```python
import sys
import re

class VMCommand():
    """
    provides simpler interface and encapsulation for inspecting current command
    """
    COMMENT_SYMBOL = '//'
    NEWLINE_SYMBOL = '\n'
    EMPTY_SYMBOL = ''

    def __init__(self, text):
        self.raw_text = text
        self.text = text.strip()
        self.parts = text.strip().split(' ')

    def is_pushpop_command(self):
        return self.operation() == 'push' or self.operation() == 'pop'

    def is_comment(self):
        return self.raw_text[0:2] == self.COMMENT_SYMBOL

    def is_whitespace(self):
        return self.raw_text == self.NEWLINE_SYMBOL

    def is_empty(self):
        return self.raw_text == self.EMPTY_SYMBOL

    def segment(self):
        # only for memory access commands
        if len(self.parts) != 3:
            return

        return self.parts[1]

```

```
36      def index(self):
37          # only for memory access commands
38          if len(self.parts) != 3:
39              return
40
41          return self.parts[2]
42
43      def operation(self):
44          return self.parts[0]
45
46  class VMParser():
47      """
48      Encapsulates access to the input code in the file
49      Reads VM commands, parses them and provides a convenient access to their components
50      Ignores Whitespace and Comments
51      """
52      def __init__(self, input_file):
53          self.input_file = open(input_file, 'r')
54          self.has_more_commands = True
55          self.current_command = None
56          self.next_command = None
57
58      def has_valid_current_command(self):
59          return not self.current_command.is_whitespace() and not self.current_command.is_comment()
60
61      def advance(self):
62          self._update_current_command()
63          self._update_next_command()
64          self._update_has_more_commands()
65
66      def _update_has_more_commands(self):
67          if self.next_command.is_empty():
68              self.has_more_commands = False
69
70      def _update_next_command(self):
```

```
71          text = self.input_file.readline()
72          self.next_command = VMCommand(text)
73
74      def _update_current_command(self):
75          # initialization
76          if self.current_command == None:
77              text = self.input_file.readline()
78              self.current_command = VMCommand(text)
79          else:
80              self.current_command = self.next_command
81
82  class VMWriter():
83      """
84      simply wrapper for interacting with output file
85      """
86      def __init__(self, input_file):
87          self.output_file = open(self._output_file_name_from(input_file), 'w')
88
89      def write(self, command):
90          self.output_file.write(command)
91
92      def close_file(self):
93          self.output_file.close()
94
95      def _output_file_name_from(self, input_file):
96          return input_file.split('.')[0] + '.asm'
97
98
99  class VMArithmeticTranslator():
100     OPERATION_INSTRUCTIONS = {
101         'add': 'M=M+D',
102         'sub': 'M=M-D',
103         'neg': 'M=-M',
104         'or' : 'M=M|D',
105         'not': 'M=!M',
```

```
        'and': 'M=M&D'
    }

    COMP_COMMANDS = {
        'eq': { 'jump_directive': 'JNE'},
        'lt': { 'jump_directive': 'JGE'},
        'gt': { 'jump_directive': 'JLE'}
    }

    def __init__(self):
        self.comp_counters = {
            'eq' : { 'count': 0 },
            'lt' : { 'count': 0 },
            'gt' : { 'count': 0 }
        }

    def translate(self, command):
        if command.text in self.COMP_COMMANDS:
            return self.comp_translation(command.text)
        else:
            return self.arithmetic_translation(command.text)

    def arithmetic_translation(self, command_text):
        # binary operation
        if command_text in [ 'add', 'sub', 'and', 'or' ]:
            return [
                *self.pop_top_number_off_stack_instructions(),
                # put in temp D for operation
                'D=M',
                *self.pop_top_number_off_stack_instructions(),
                self.OPERATION_INSTRUCTIONS[command_text],
                *self.increment_stack_pointer_instructions()
            ]
        else: # unary operation
            return [
                *self.pop_top_number_off_stack_instructions(),
                self.OPERATION_INSTRUCTIONS[command_text],
                *self.increment_stack_pointer_instructions()
            ]

    def comp_translation(self, command_text):
        counter = self.comp_counters[command_text]
        counter['count'] += 1
        label_identifier = '{}{}'.format(command_text.upper(), counter['count'])
        jump_directive = self.COMP_COMMANDS[command_text]['jump_directive']

        return [
            *self.pop_top_number_off_stack_instructions(),
            # set D to top of stack
            'D=M',
            *self.pop_top_number_off_stack_instructions(),
            # set D to x-y
            'D=M-D',
            # load not true label
            '@NOT_{}'.format(label_identifier),
            # jump to not true section on directive
            'D;{}'.format(jump_directive),
            # load stack pointer
            '@SP',
            # set A to top of stack address
            'A=M',
            # set it to -1 for true
            'M=-1',
            # load inc stack pointer
            '@INC_STACK_POINTER_{}'.format(label_identifier),
            # jump uncoditionally
            '0;JMP',
            # not true section
            '(NOT_{})'.format(label_identifier),
            # load stack pointer
```

```
176             '@SP',
177             # set A to to top of stack address
178             'A=M',
179             # set to 0 for false
180             'M=0',
181             # define inc stack pointer label
182             '(INC_STACK_POINTER_{})'.format(label_identifier),
183             *self.increment_stack_pointer_instructions()
184         ]
185
186     def pop_top_number_off_stack_instructions(self):
187         return [
188             # load stack pointer
189             '@SP',
190             # decrement stack pointer and set address
191             'AM=M-1'
192         ]
193
194     def increment_stack_pointer_instructions(self):
195         return [
196             # load stack pointer
197             '@SP',
198             # increment stack pointer
199             'M=M+1'
200         ]
201
202
203 class VMPushPopTranslator():
204     VIRTUAL_MEMORY_SEGMENTS = {
205         'local'    : { 'base_address_pointer': '1' },
206         'argument' : { 'base_address_pointer': '2' },
207         'this'     : { 'base_address_pointer': '3' },
208         'that'     : { 'base_address_pointer': '4' }
209     }
210
```

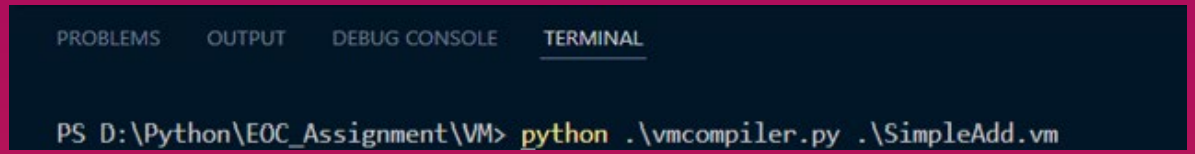
```
    POINTER_SEGMENT_BASE_ADDRESS = '3'

    HOST_SEGMENTS = {
        'temp'  : { 'base_address': '5' },
        'static': { 'base_address': '16'}
    }

    def translate(self, command):
        if command.operation() == 'push':
            # Push the value of segment[index] onto the stack

            return [
                *self.load_desired_value_into_D_instructions_for(segment=command.segment(), index=command.index()),
                *self.place_value_in_D_on_top_of_stack_instructions(),
                *self.increment_stack_pointer_instructions()
            ]
        else: # command operation is pull
            # Pop the top-most value off the stack store in segment[index]

            return [
                *self.store_top_of_stack_in_D_instructions(),
                *self.store_top_of_stack_first_temp_register_instructions(),
                *self.load_base_address_instructions_for(segment=command.segment()),
                *self.add_index_to_base_address_in_D_instructions(index=command.index()),
                *self.store_target_address_in_second_temp_register_instructions(),
                *self.set_target_address_to_value_instructions()
            ]


    def load_desired_value_into_D_instructions_for(self, segment, index):
        if segment == 'constant':
            return [
                *self.load_value_in_D_instructions(value=index)
            ]
        else:
```

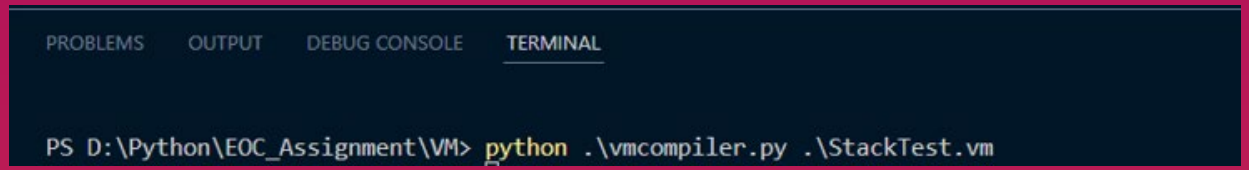
```
            return [
                *self.load_base_address_instructions_for(segment=segment),
                *self.add_index_to_base_address_in_D_instructions(index=index),
                *self.load_value_at_memory_address_in_D_instructions()
            ]

    def load_base_address_instructions_for(self, segment):
        if segment in self.VIRTUAL_MEMORY_SEGMENTS:
            pointer_to_segment_base_address = self.VIRTUAL_MEMORY_SEGMENTS[segment]['base_address_pointer']
            return self.load_referenced_value_in_D_instructions(address=pointer_to_segment_base_address)
        elif segment in self.HOST_SEGMENTS:
            host_segment_base_address = self.HOST_SEGMENTS[segment]['base_address']
            return self.load_value_in_D_instructions(value=host_segment_base_address)
        elif segment == 'pointer':
            return self.load_value_in_D_instructions(value=self.POINTER_SEGMENT_BASE_ADDRESS)


    def place_value_in_D_on_top_of_stack_instructions(self):
        return [
            # load stack pointer
            '@SP',
            # Get current address
            'A=M',
            # Store constant in address
            'M=D'
        ]

    def increment_stack_pointer_instructions(self):
        return [
            # load stack pointer
            '@SP',
            # increment stack pointer
            'M=M+1'
        ]

```

```
281     def load_value_in_D_instructions(self, value):
282         return [
283             # load value
284             '@' + value,
285             # store value in D
286             'D=A'
287         ]
288 
289     def load_referenced_value_in_D_instructions(self, address):
290         return [
291             # load address
292             '@' + address,
293             # store address value
294             'D=M'
295         ]
296 
297     def add_index_to_base_address_in_D_instructions(self, index):
298         return [
299             '@' + index,
300             'D=D+A'
301         ]
302     def load_value_at_memory_address_in_D_instructions(self):
303         return [
304             # set A to address stored in D
305             'A=D',
306             # now put value at new address in D
307             'D=M'
308         ]
309 
310     def set_address_to_top_of_stack_instructions(self, address):
311         return [
312             # load segment address
313             '@' + address,
314             # set segment equal to top of stack
315             'M=D'
```

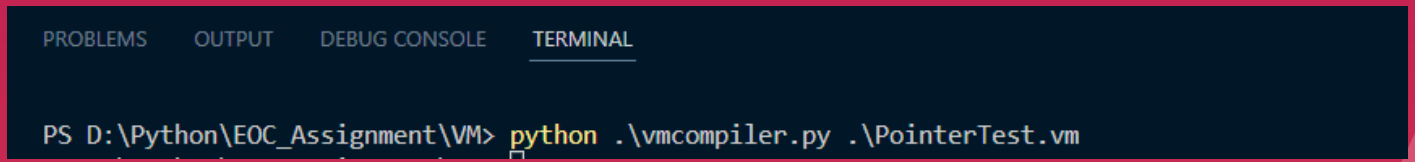
```
        ]

    def set_target_address_to_value_instructions(self):
        return [
            # load top of stack value
            '@R5',
            # store in D
            'D=M',
            # load segment + index address
            '@R6',
            # set as current address register
            'A=M',
            # set segment[index] to stack top
            'M=D'
        ]

    def store_target_address_in_second_temp_register_instructions(self):
        return [
            # load temp
            '@R6',
            # store segment + index address
            'M=D'
        ]

    def store_top_of_stack_first_temp_register_instructions(self):
        return [
            # load temp register
            '@R5',
            # store top of stack in temp register
            'M=D'
        ]

    def store_top_of_stack_in_D_instructions(self):
        return [
            # load stack pointer
```

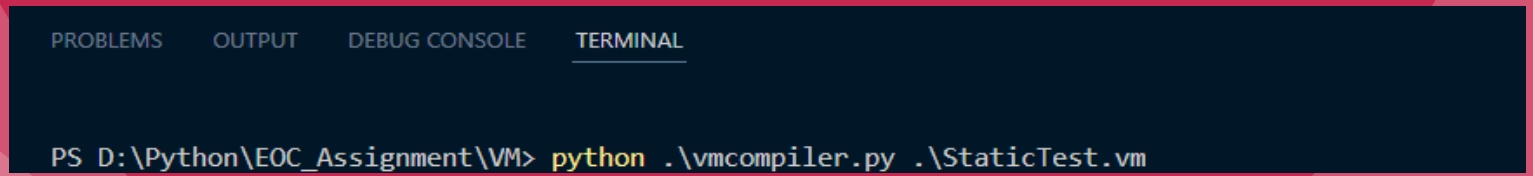
```
            '@SP',
            # decrement pointer to top of stack
            'AM=M-1',
            # store value in D
            'D=M'
        ]


if __name__ == "__main__" and len(sys.argv) == 2:
    vm_code_file = sys.argv[1]

    parser = VMParser(vm_code_file)
    writer = VMWriter(vm_code_file)
    arithmetic_translator = VMArithmeticTranslator()
    push_pop_translator = VMPushPopTranslator()

    while parser.has_more_commands:
        parser.advance()
        translation = []

        if parser.has_valid_current_command():
            if parser.current_command.is_pushpop_command():
                translation = push_pop_translator.translate(parser.current_command)
            else:
                translation = arithmetic_translator.translate(parser.current_command)

            for line in translation:
                writer.write(line + '\n')

    writer.close_file()
```


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PS D:\Python\EOC_Assignment\VM> python .\vmcompiler.py .\SimpleAdd.vm


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PS D:\Python\EOC_Assignment\VM> python .\vmcompiler.py .\StackTest.vm


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PS D:\Python\EOC_Assignment\VM> python .\vmcompiler.py .\BasicTest.vm


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PS D:\Python\EOC_Assignment\VM> python .\vmcompiler.py .\PointerTest.vm


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PS D:\Python\EOC_Assignment\VM> python .\vmcompiler.py .\StaticTest.vm

PART-C

Use your VM translator to translate the above .vm programs to .asm programs

Simple add

SimpleAdd.asm - Notepad

File Edit Format View Help

```
// push constant 7
@7
D=A
@SP
A=M
M=D
@SP
M=M+1


// push constant 8
@8
D=A
@SP
A=M
M=D
@SP
M=M+1


// add
@SP
AM=M-1
D=M
A=A-1
M=M+D


(EXIT)
@EXIT
0;JMP
```

Stack Test

StackTest.asm - Notepad

File Edit Format View Help

```
// push constant 17
@17
D=A
@SP
A=M
M=D
@SP
M=M+1


// push constant 17
@17
D=A
@SP
A=M
M=D
@SP
M=M+1


// eq
@SP
AM=M-1
D=M
A=A-1
D=M-D
@c0_if_eq
D;JEQ
D=0
@c0_else
0;JMP
(c0_if_eq)
D=-1
(c0_else)
@SP
A=M-1
M=D
```

StackTest.asm - Notepad

File Edit Format View Help

```
// push constant 16
@16
D=A
@SP
A=M
M=D
@SP
M=M+1


// push constant 17
@17
D=A
@SP
A=M
M=D
@SP
M=M+1


// eq
@SP
AM=M-1
D=M
A=A-1
D=M-D
@c2_if_eq
D;JEQ
D=0
@c2_else
0;JMP
(c2_if_eq)
D=-1
(c2_else)
@SP
A=M-1
M=D
```

StackTest.asm - Notepad

File Edit Format View Help

```
// push constant 892
@892
D=A
@SP
A=M
M=D
@SP
M=M+1


// push constant 891
@891
D=A
@SP
A=M
M=D
@SP
M=M+1


// lt
@SP
AM=M-1
D=M
A=A-1
D=M-D
@c3_if_lt
D;JLT
D=0
@c3_else
0;JMP
(c3_if_lt)
D=-1
(c3_else)
@SP
A=M-1
M=D
```

StackTest.asm - Notepad

File Edit Format View Help

```
// push constant 891
@891
D=A
@SP
A=M
M=D
@SP
M=M+1


// push constant 892
@892
D=A
@SP
A=M
M=D
@SP
M=M+1


// lt
@SP
AM=M-1
D=M
A=A-1
D=M-D
@c4_if_lt
D;JLT
D=0
@c4_else
0;JMP
(c4_if_lt)
D=-1
(c4_else)
@SP
A=M-1
M=D
```

StackTest.asm - Notepad

File Edit Format View Help

```
// push constant 32767
@32767
D=A
@SP
A=M
M=D
@SP
M=M+1


// push constant 32766
@32766
D=A
@SP
A=M
M=D
@SP
M=M+1


// gt
@SP
AM=M-1
D=M
A=A-1
D=M-D
@c6_if_gt
D;JGT
D=0
@c6_else
0;JMP
(c6_if_gt)
D=-1
(c6_else)
@SP
A=M-1
M=D
```

StackTest.asm - Notepad

File Edit Format View Help

```
// push constant 32766
@32766
D=A
@SP
A=M
M=D
@SP
M=M+1


// push constant 32767
@32767
D=A
@SP
A=M
M=D
@SP
M=M+1


// gt
@SP
AM=M-1
D=M
A=A-1
D=M-D
@c7_if_gt
D;JGT
D=0
@c7_else
0;JMP
(c7_if_gt)
D=-1
(c7_else)
@SP
A=M-1
M=D
```

StackTest.asm - Notepad

File Edit Format View Help

```
// push constant 57
@57
D=A
@SP
A=M
M=D
@SP
M=M+1


// push constant 31
@31
D=A
@SP
A=M
M=D
@SP
M=M+1


// push constant 53
@53
D=A
@SP
A=M
M=D
@SP
M=M+1


// add
@SP
AM=M-1
D=M
A=A-1
M=M+D
```

StackTest.asm - Notepad

File Edit Format View Help

```
// push constant 112
@112
D=A
@SP
A=M
M=D
@SP
M=M+1


// sub
@SP
AM=M-1
D=M
A=A-1
M=M-D


// neg
@SP
A=M-1
M=-M


// and
@SP
AM=M-1
D=M
A=A-1
M=M&D


// push constant 82
@82
D=A
@SP
A=M
M=D
@SP
M=M+1
```

StackTest.asm - Notepad

File Edit Format View Help

```
// neg
@SP
A=M-1
M=-M


// and
@SP
AM=M-1
D=M
A=A-1
M=M&D


// push constant 82
@82
D=A
@SP
A=M
M=D
@SP
M=M+1


// or
@SP
AM=M-1
D=M
A=A-1
M=M|D


// not
@SP
A=M-1
M=!M


(EXIT)
@EXIT
0;JMP
```

Basic Test

BasicTest.asm - Notepad

File Edit Format View Help

```
// push constant 10
@10
D=A
@SP
M=M+1
A=M-1
M=D
// pop local 0
@0
D=A
@LCL
AD=D+M
@R13
M=D
@SP
M=M-1
A=M
D=M
@R13
A=M
M=D
// push constant 21
@21
D=A
@SP
M=M+1
A=M-1
M=D
// push constant 22
@22
D=A
@SP
M=M+1
A=M-1
M=D
```

BasicTest.asm - Notepad

File Edit Format View Help

```
// pop argument 2
@2
D=A
@ARG
AD=D+M
@R13
M=D
@SP
M=M-1
A=M
D=M
@R13
A=M
M=D
// pop argument 1
@1
D=A
@ARG
AD=D+M
@R13
M=D
@SP
M=M-1
A=M
D=M
@R13
A=M
M=D
// push constant 36
@36
D=A
@SP
M=M+1
A=M-1
M=D
```

BasicTest.asm - Notepad

File Edit Format View Help

```
// pop this 6
@6
D=A
@THIS
AD=D+M
@R13
M=D
@SP
M=M-1
A=M
D=M
@R13
A=M
M=D
// push constant 42
@42
D=A
@SP
M=M+1
A=M-1
M=D
// push constant 45
@45
D=A
@SP
M=M+1
A=M-1
M=D
```

BasicTest.asm - Notepad

File Edit Format View Help

```
// pop that 5
@5
D=A
@THAT
AD=D+M
@R13
M=D
@SP
M=M-1
A=M
D=M
@R13
A=M
M=D
// pop that 2
@2
D=A
@THAT
AD=D+M
@R13
M=D
@SP
M=M-1
A=M
D=M
@R13
A=M
M=D
// push constant 510
@510
D=A
@SP
M=M+1
A=M-1
M=D
```

BasicTest.asm - Notepad

File Edit Format View Help

```
// pop temp 6
@11
D=A
@R13
M=D
@SP
M=M-1
A=M
D=M
@R13
A=M
M=D
// push local 0
@0
D=A
@LCL
AD=D+M

D=M
@SP
M=M+1
A=M-1
M=D
// push that 5
@5
D=A
@THAT
AD=D+M

D=M
@SP
M=M+1
A=M-1
M=D
// add
@SP
M=M-1
A=M
D=M
A=A-1
```

BasicTest.asm - Notepad

File Edit Format View Help

```
// push argument 1
@1
D=A
@ARG
AD=D+M

D=M
@SP
M=M+1
A=M-1
M=D
// sub
@SP
M=M-1
A=M
D=M
A=A-1
M=M-D
// push this 6
@6
D=A
@THIS
AD=D+M

D=M
@SP
M=M+1
A=M-1
M=D
// push this 6
@6
D=A
@THIS
AD=D+M

D=M
@SP
M=M+1
A=M-1
M=D
```

```
// add
@SP
M=M-1
A=M
D=M
A=A-1
M=D+M
// sub
@SP
M=M-1
A=M
D=M
A=A-1
M=M-D
// push temp 6
@11

D=M
@SP
M=M+1
A=M-1
M=D
// add
@SP
M=M-1
A=M
D=M
A=A-1
M=D+M
```

Pointer Test

PointerTest.asm - Notepad

File Edit Format View Help

```
// push constant 3030
@3030
D=A
@SP
A=M
M=D
@SP
M=M+1


// pop pointer 0
@R3
D=A
@R13
M=D
@SP
AM=M-1
D=M
@R13
A=M
M=D


// push constant 3040
@3040
D=A
@SP
A=M
M=D
@SP
M=M+1
```

PointerTest.asm - Notepad

File Edit Format View Help

```
// push constant 3030
@3030
D=A
@SP
A=M
M=D
@SP
M=M+1


// pop pointer 0
@R3
D=A
@R13
M=D
@SP
AM=M-1
D=M
@R13
A=M
M=D


// push constant 3040
@3040
D=A
@SP
A=M
M=D
@SP
M=M+1
```

PointerTest.asm - Notepad

File Edit Format View Help

```
// push constant 46
@46
D=A
@SP
A=M
M=D
@SP
M=M+1


// pop that 6
@THAT
D=M
@6
A=D+A
D=A
@R13
M=D
@SP
AM=M-1
D=M
@R13
A=M
M=D


// push pointer 0
@R3
D=M
@SP
A=M
M=D
@SP
M=M+1
```

PointerTest.asm - Notepad

File Edit Format View Help

```
// push pointer 1
@R4
D=M
@SP
A=M
M=D
@SP
M=M+1


// add
@SP
AM=M-1
D=M
A=A-1
M=M+D


// push this 2
@THIS
D=M
@2
A=D+A
D=M
@SP
A=M
M=D
@SP
M=M+1


// sub
@SP
AM=M-1
D=M
A=A-1
M=M-D
```

Static Test

StaticTest.asm - Notepad

File Edit Format View Help

```
// push constant 111
@111
D=A
@SP
A=M
M=D
@SP
M=M+1


// push constant 333
@333
D=A
@SP
A=M
M=D
@SP
M=M+1


// push constant 888
@888
D=A
@SP
A=M
M=D
@SP
M=M+1
```

StaticTest.asm - Notepad

File Edit Format View Help

```
// pop static 8
@StaticTest.8
D=M
D=A
@R13
M=D
@SP
AM=M-1
D=M
@R13
A=M
M=D


// pop static 3
@StaticTest.3
D=M
D=A
@R13
M=D
@SP
AM=M-1
D=M
@R13
A=M
M=D


// pop static 1
@StaticTest.1
D=M
D=A
@R13
M=D
@SP
AM=M-1
D=M
@R13
A=M
M=D
```

StaticTest.asm - Notepad

File Edit Format View Help

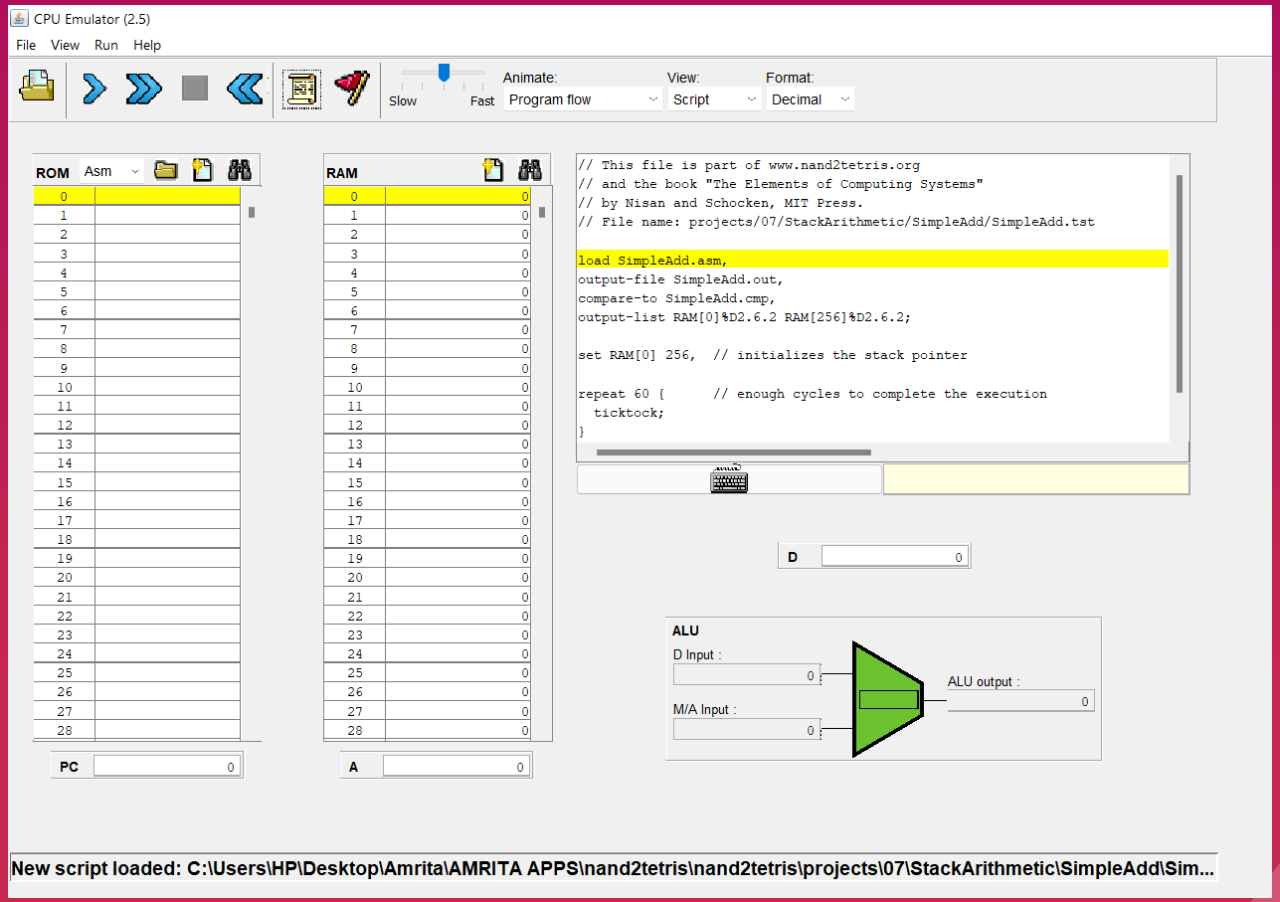
```
// push static 3
@StaticTest.3
D=M
D=M
@SP
A=M
M=D
@SP
M=M+1


// push static 1
@StaticTest.1
D=M
D=M
@SP
A=M
M=D
@SP
M=M+1


// sub
@SP
AM=M-1
D=M
A=A-1
M=M-D


// push static 8
@StaticTest.8
D=M
D=M
@SP
A=M
M=D
@SP
M=M+1
```

```
// add
@SP
AM=M-1
D=M
A=A-1
M=M+D


(EXIT)
@EXIT
0;JMP
```

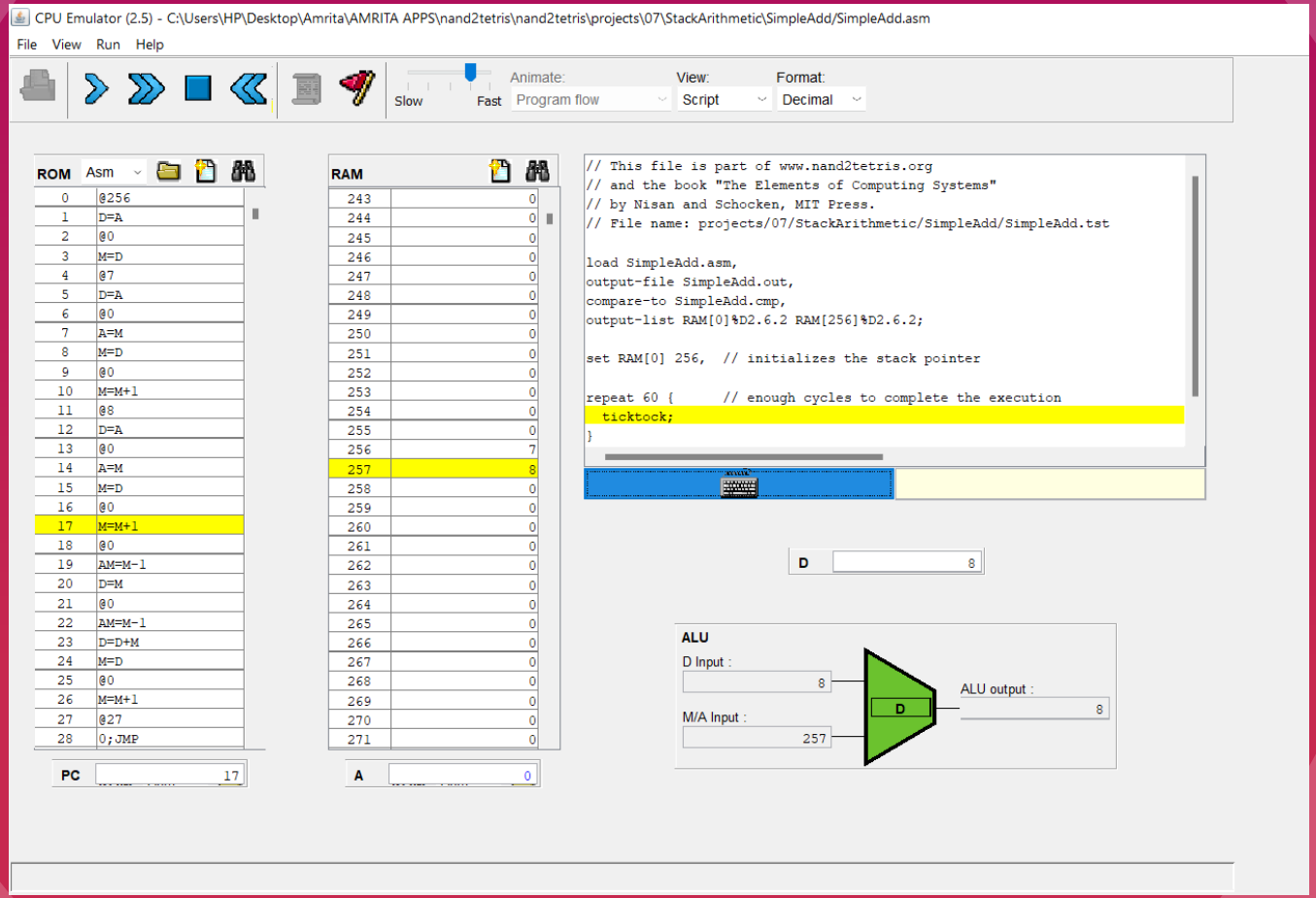
PART-D

Execute the generated .asm programs on the CPU emulator using the supplied test script.

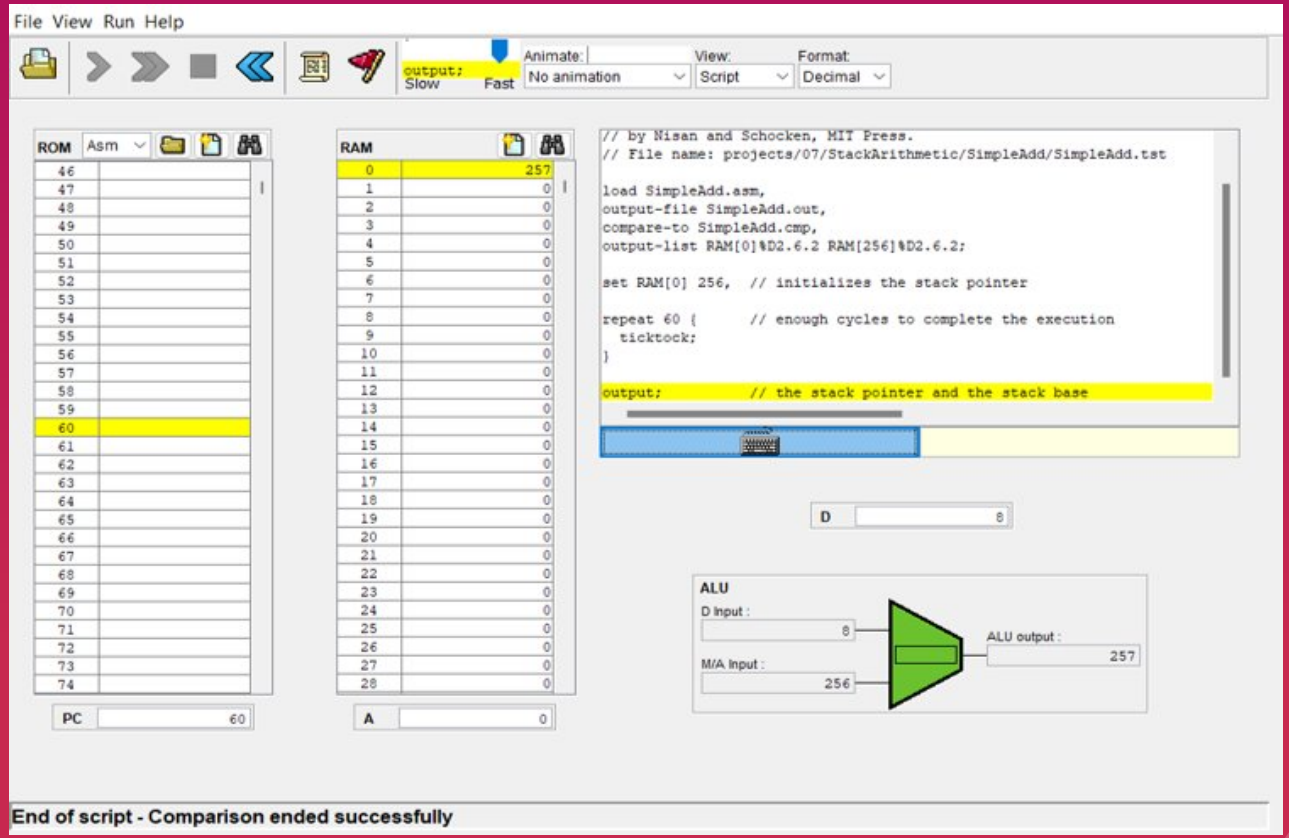
SIMPLE ADD


CPU Emulator (2.5)
File View Run Help
Slow Fast
Animate: Program flow
View: Script
Format: Decimal
ROM Asm
RAM
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/07/StackArithmetic/SimpleAdd/SimpleAdd.tst
load SimpleAdd.asm,
output-file SimpleAdd.out,
compare-to SimpleAdd.cmp,
output-list RAM[0]%D2.6.2 RAM[256]%D2.6.2;
set RAM[0] 256, // initializes the stack pointer
repeat 60 { // enough cycles to complete the execution
ticktock;
}
D
ALU
D Input :
M/A Input :
ALU output :
PC
A
New script loaded: C:\Users\HP\Desktop\Amrita\AMRITA APPS\nand2tetris\nand2tetris\projects\07\StackArithmetic\SimpleAdd\Sim...

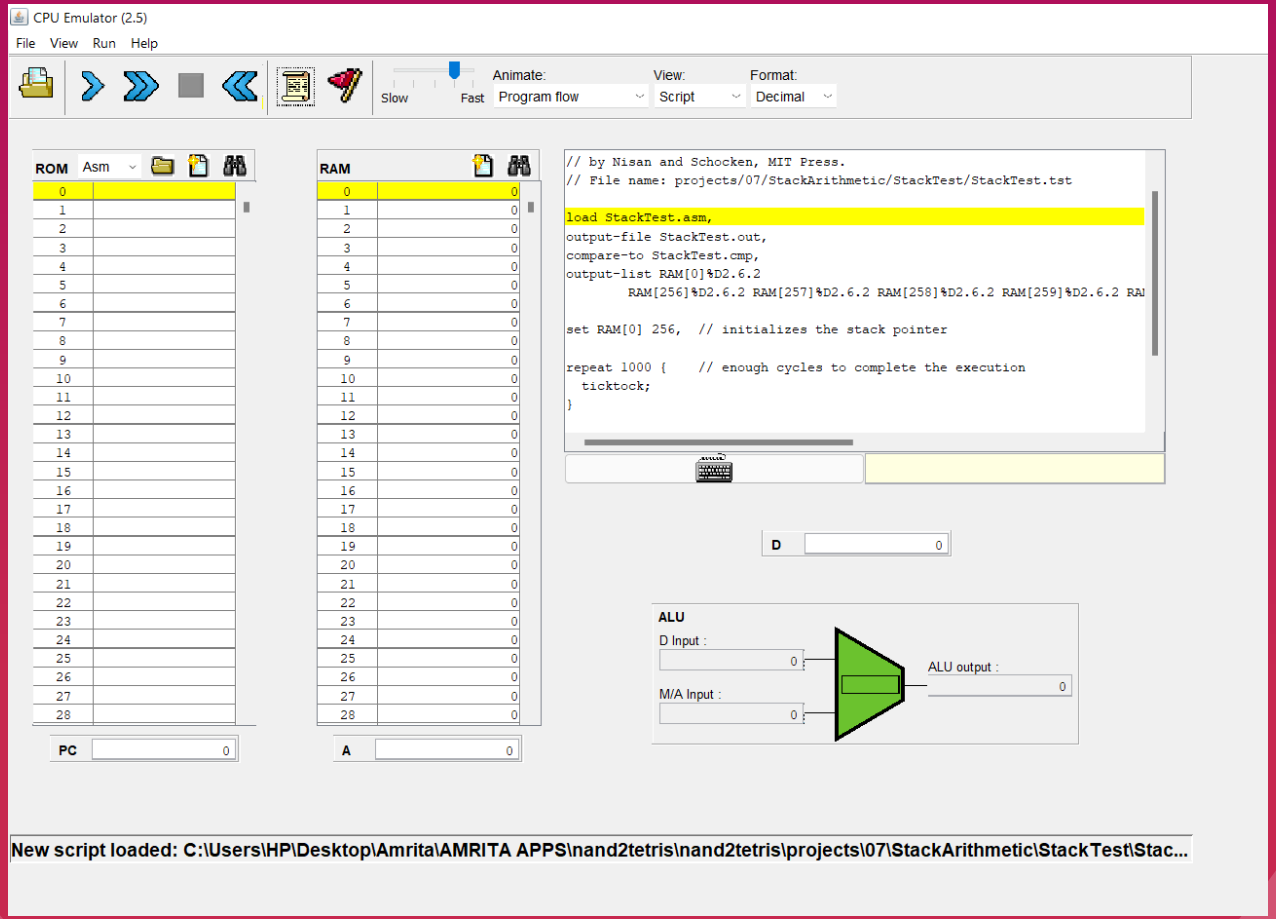
SIMPLE ADD


CPU Emulator (2.5) - C:\Users\HP\Desktop\Amrita\AMRITA APPS\nand2tetris\nand2tetris\projects\07\StackArithmetic\SimpleAdd/SimpleAdd.asm
File View Run Help
Animate: Slow Fast
Program flow
View: Script
Format: Decimal
ROM Asm
0 @256
1 D=A
2 @0
3 M=D
4 @7
5 D=A
6 @0
7 A=M
8 M=D
9 @0
10 M=M+1
11 @8
12 D=A
13 @0
14 A=M
15 M=D
16 @0
17 M=M+1
18 @0
19 AM=M-1
20 D=M
21 @0
22 AM=M-1
23 D=D+M
24 M=D
25 @0
26 M=M+1
27 @27
28 0;JMP
PC 17
RAM
243 0
244 0
245 0
246 0
247 0
248 0
249 0
250 0
251 0
252 0
253 0
254 0
255 0
256 7
257 8
258 0
259 0
260 0
261 0
262 0
263 0
264 0
265 0
266 0
267 0
268 0
269 0
270 0
271 0
A 0
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/07/StackArithmetic/SimpleAdd/SimpleAdd.tst
load SimpleAdd.asm,
output-file SimpleAdd.out,
compare-to SimpleAdd.cmp,
output-list RAM[0]%D2.6.2 RAM[256]%D2.6.2;
set RAM[0] 256, // initializes the stack pointer
repeat 60 { // enough cycles to complete the execution
ticktock;
}
D 8
ALU
D Input : 8
M/A Input : 257
D
ALU output : 8

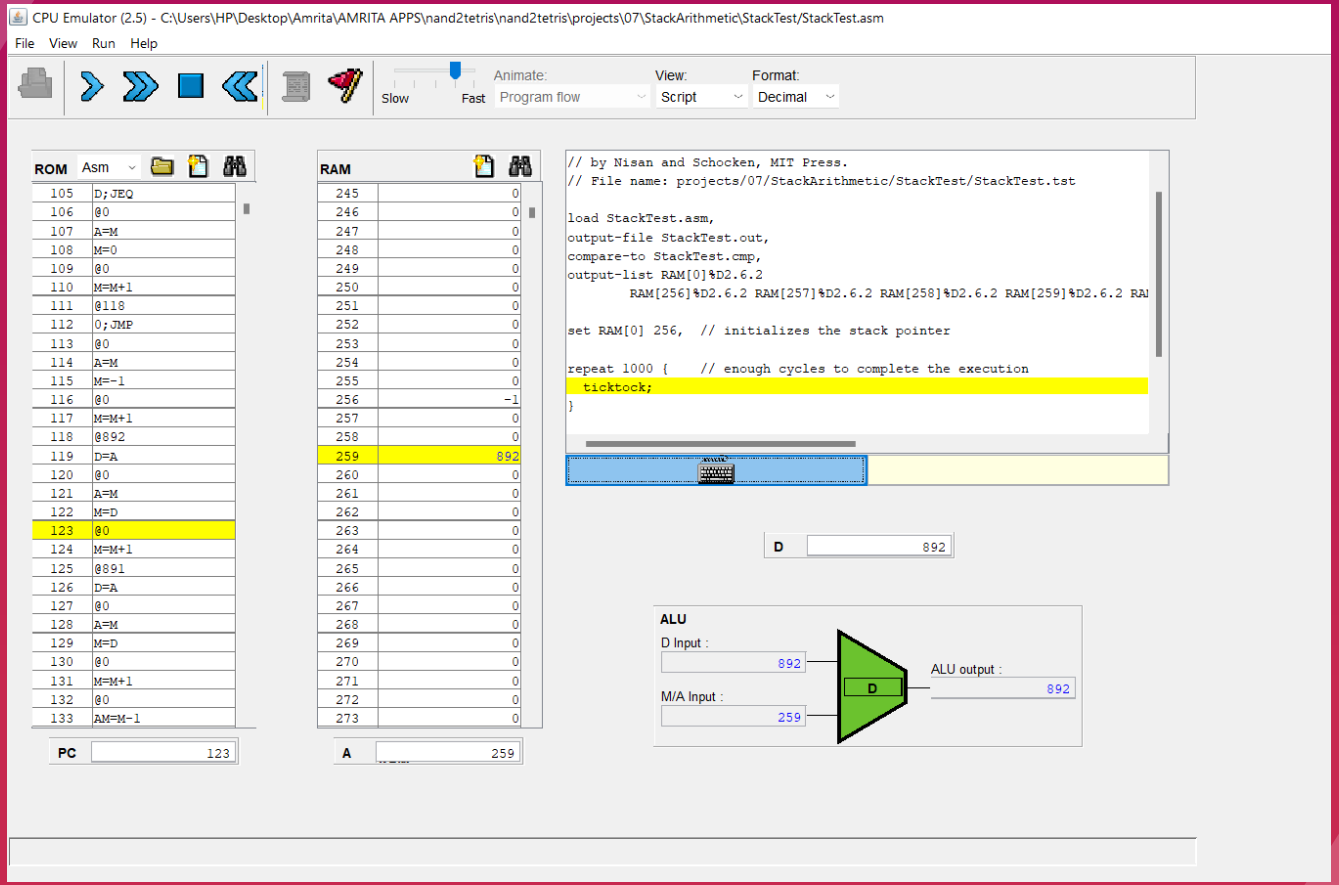
SIMPLE ADD


File View Run Help
output;
Slow
Fast
Animate:
No animation
View:
Script
Format:
Decimal
ROM
Asm
RAM
// by Nisan and Schocken, MIT Press.
// File name: projects/07/StackArithmetic/SimpleAdd/SimpleAdd.tst
load SimpleAdd.asm,
output-file SimpleAdd.out,
compare-to SimpleAdd.cmp,
output-list RAM[0]%D2.6.2 RAM[256]%D2.6.2;
set RAM[0] 256, // initializes the stack pointer
repeat 60 { // enough cycles to complete the execution
ticktock;
}
output; // the stack pointer and the stack base
D
8
ALU
D Input :
8
M/A Input :
256
ALU output :
257
PC
60
A
0
End of script - Comparison ended successfully

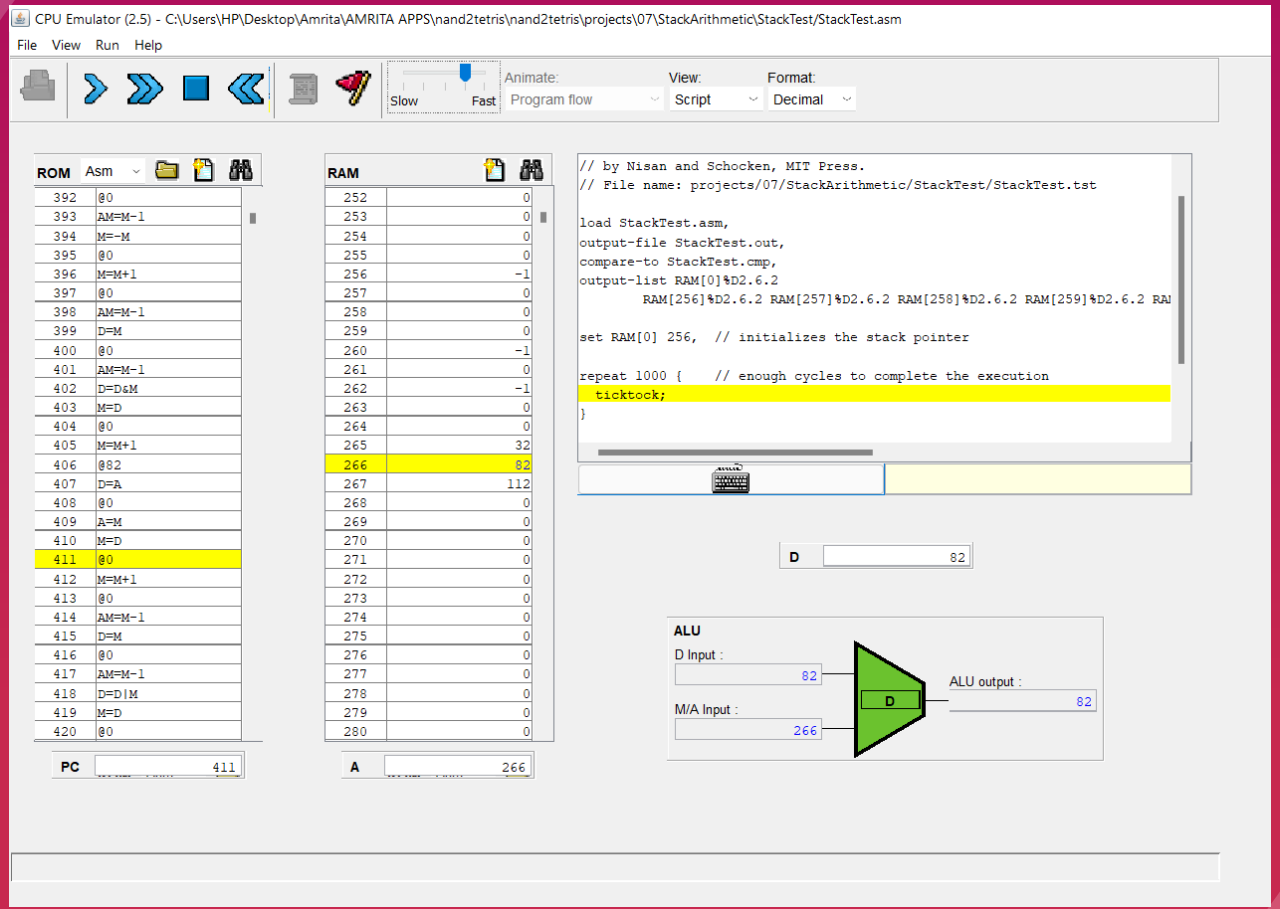
STACK TEST


CPU Emulator (2.5)
File View Run Help
Animate: View: Format:
Slow Fast Program flow Script Decimal
ROM Asm
RAM
// by Nisan and Schocken, MIT Press.
// File name: projects/07/StackArithmetic/StackTest/StackTest.tst
load StackTest.asm,
output-file StackTest.out,
compare-to StackTest.cmp,
output-list RAM[0]%D2.6.2
RAM[256]%D2.6.2 RAM[257]%D2.6.2 RAM[258]%D2.6.2 RAM[259]%D2.6.2 RA
set RAM[0] 256, // initializes the stack pointer
repeat 1000 { // enough cycles to complete the execution
ticktock;
}
D
ALU
D Input :
M/A Input :
ALU output :
PC
A
New script loaded: C:\Users\HP\Desktop\Amrita\AMRITA APPS\nand2tetris\nand2tetris\projects\07\StackArithmetic\StackTest\Stac...

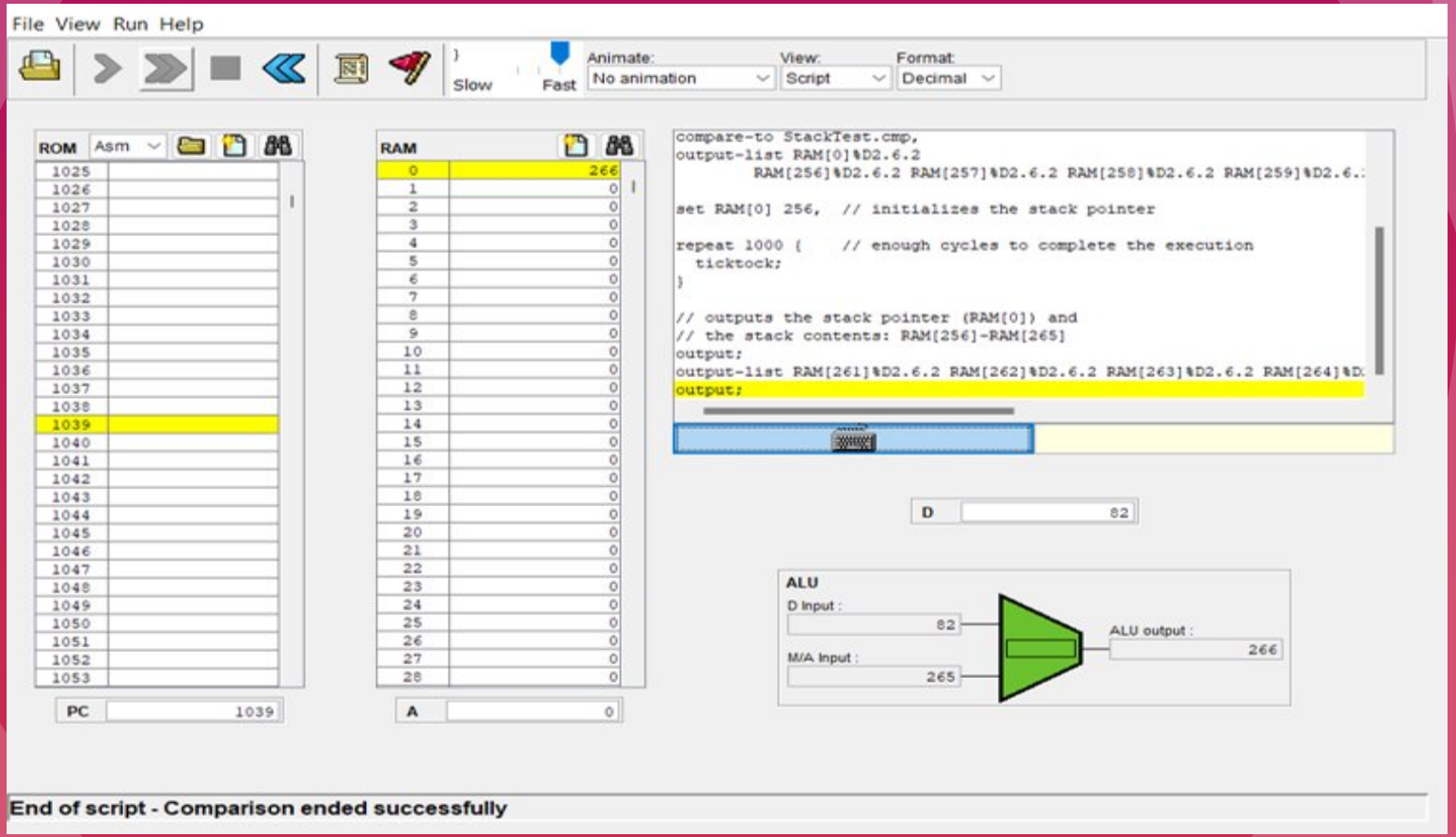
STACK TEST


CPU Emulator (2.5) - C:\Users\HP\Desktop\Amrita\AMRITA APPS\nand2tetris\nand2tetris\projects\07\StackArithmetic\StackTest/StackTest.asm
File View Run Help
Slow Fast
Animate: Program flow
View: Script
Format: Decimal
ROM Asm
105 D;JEQ
106 @0
107 A=M
108 M=0
109 @0
110 M=M+1
111 @118
112 0;JMP
113 @0
114 A=M
115 M=-1
116 @0
117 M=M+1
118 @892
119 D=A
120 @0
121 A=M
122 M=D
123 @0
124 M=M+1
125 @891
126 D=A
127 @0
128 A=M
129 M=D
130 @0
131 M=M+1
132 @0
133 AM=M-1
PC 123
RAM
245 0
246 0
247 0
248 0
249 0
250 0
251 0
252 0
253 0
254 0
255 0
256 -1
257 0
258 0
259 892
260 0
261 0
262 0
263 0
264 0
265 0
266 0
267 0
268 0
269 0
270 0
271 0
272 0
273 0
A 259
// by Nisan and Schocken, MIT Press.
// File name: projects/07/StackArithmetic/StackTest/StackTest.tst
load StackTest.asm,
output-file StackTest.out,
compare-to StackTest.cmp,
output-list RAM[0]%D2.6.2
RAM[256]%D2.6.2 RAM[257]%D2.6.2 RAM[258]%D2.6.2 RAM[259]%D2.6.2 RA
set RAM[0] 256, // initializes the stack pointer
repeat 1000 { // enough cycles to complete the execution
ticktock;
}
D 892
ALU
D Input :
892
M/A Input :
259
D
ALU output :
892

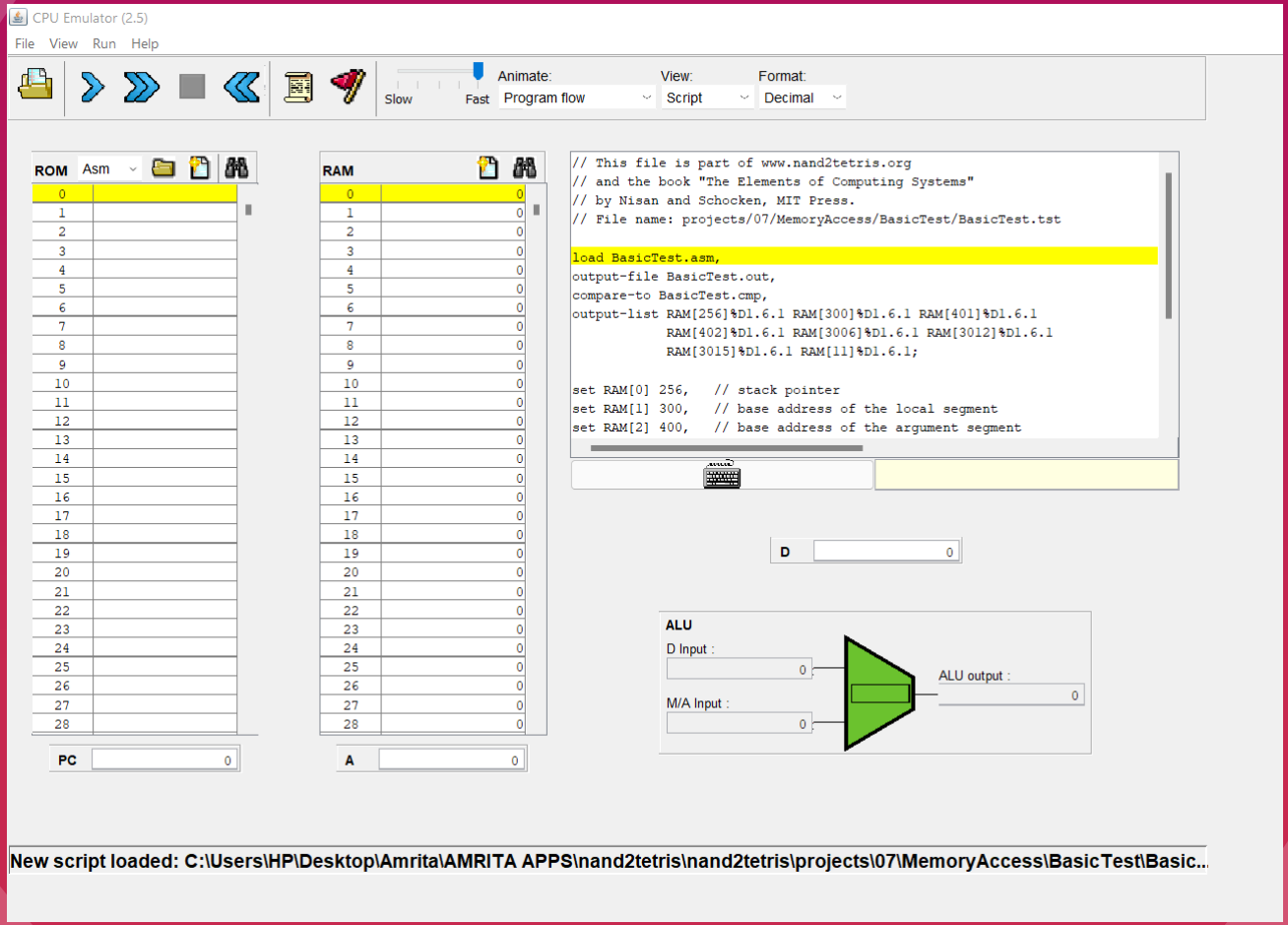
STACK TEST


CPU Emulator (2.5) - C:\Users\HP\Desktop\Amrita\AMRITA APPS\nand2tetris\nand2tetris\projects\07\StackArithmetic\StackTest/StackTest.asm
File View Run Help
Slow Fast
Animate: Program flow
View: Script
Format: Decimal
ROM Asm
392 @0
393 AM=M-1
394 M=-M
395 @0
396 M=M+1
397 @0
398 AM=M-1
399 D=M
400 @0
401 AM=M-1
402 D=D&M
403 M=D
404 @0
405 M=M+1
406 @82
407 D=A
408 @0
409 A=M
410 M=D
411 @0
412 M=M+1
413 @0
414 AM=M-1
415 D=M
416 @0
417 AM=M-1
418 D=D|M
419 M=D
420 @0
PC 411
RAM
252 0
253 0
254 0
255 0
256 -1
257 0
258 0
259 0
260 -1
261 0
262 -1
263 0
264 0
265 32
266 82
267 112
268 0
269 0
270 0
271 0
272 0
273 0
274 0
275 0
276 0
277 0
278 0
279 0
280 0
A 266
// by Nisan and Schocken, MIT Press.
// File name: projects/07/StackArithmetic/StackTest/StackTest.tst
load StackTest.asm,
output-file StackTest.out,
compare-to StackTest.cmp,
output-list RAM[0]%D2.6.2
RAM[256]%D2.6.2 RAM[257]%D2.6.2 RAM[258]%D2.6.2 RAM[259]%D2.6.2 RAI
set RAM[0] 256, // initializes the stack pointer
repeat 1000 { // enough cycles to complete the execution
ticktock;
}
D 82
ALU
D Input :
82
M/A Input :
266
D
ALU output :
82

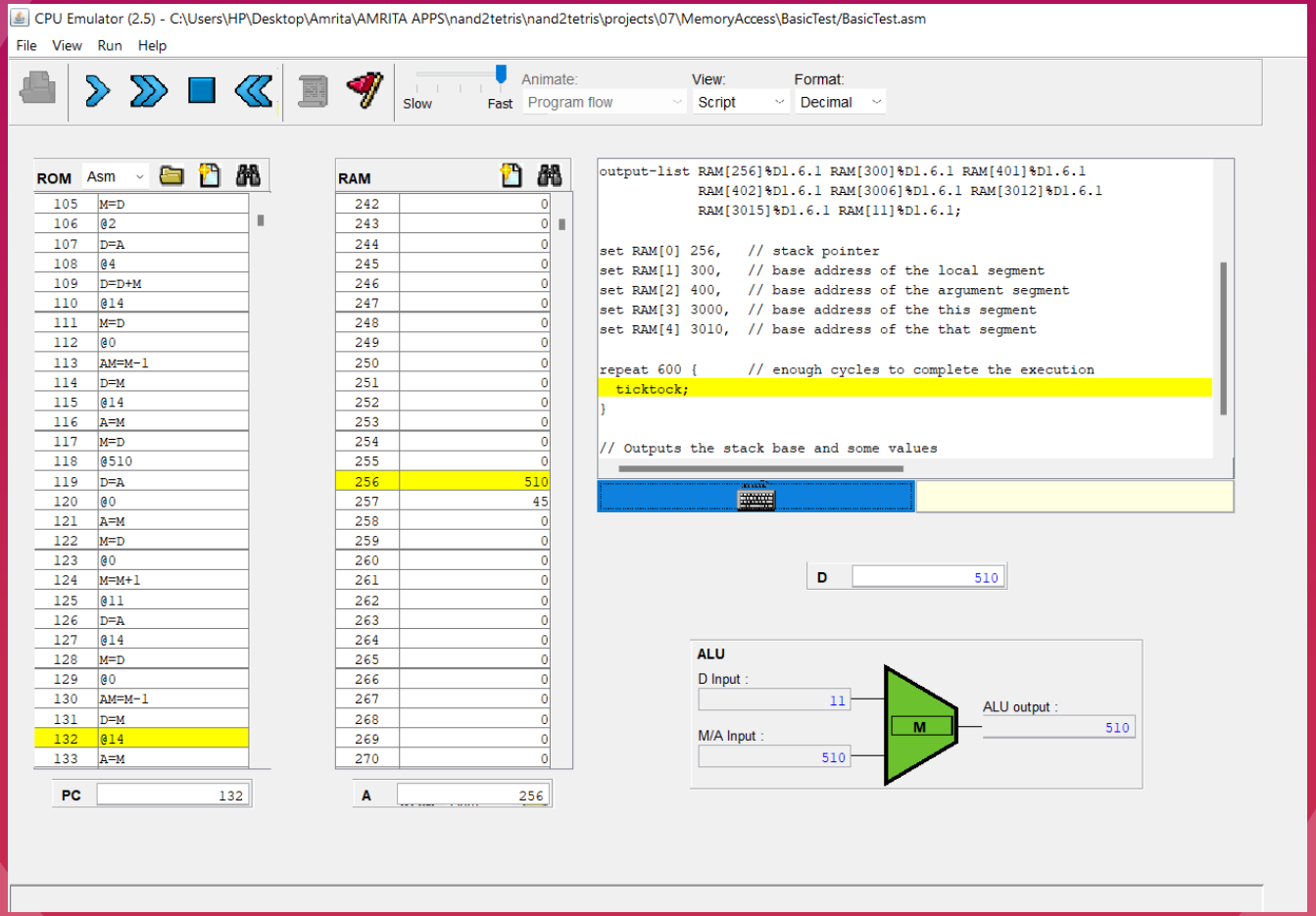
STACK TEST


File View Run Help
Slow
Fast
Animate:
No animation
View:
Script
Format:
Decimal
ROM
Asm
RAM
compare-to StackTest.cmp,
output-list RAM[0]%D2.6.2
RAM[256]%D2.6.2 RAM[257]%D2.6.2 RAM[258]%D2.6.2 RAM[259]%D2.6.
set RAM[0] 256, // initializes the stack pointer
repeat 1000 { // enough cycles to complete the execution
ticktock;
}
// outputs the stack pointer (RAM[0]) and
// the stack contents: RAM[256]-RAM[265]
output;
output-list RAM[261]%D2.6.2 RAM[262]%D2.6.2 RAM[263]%D2.6.2 RAM[264]%D
output;
D
82
ALU
D Input :
82
M/A Input :
265
ALU output :
266
PC
1039
A
0
End of script - Comparison ended successfully

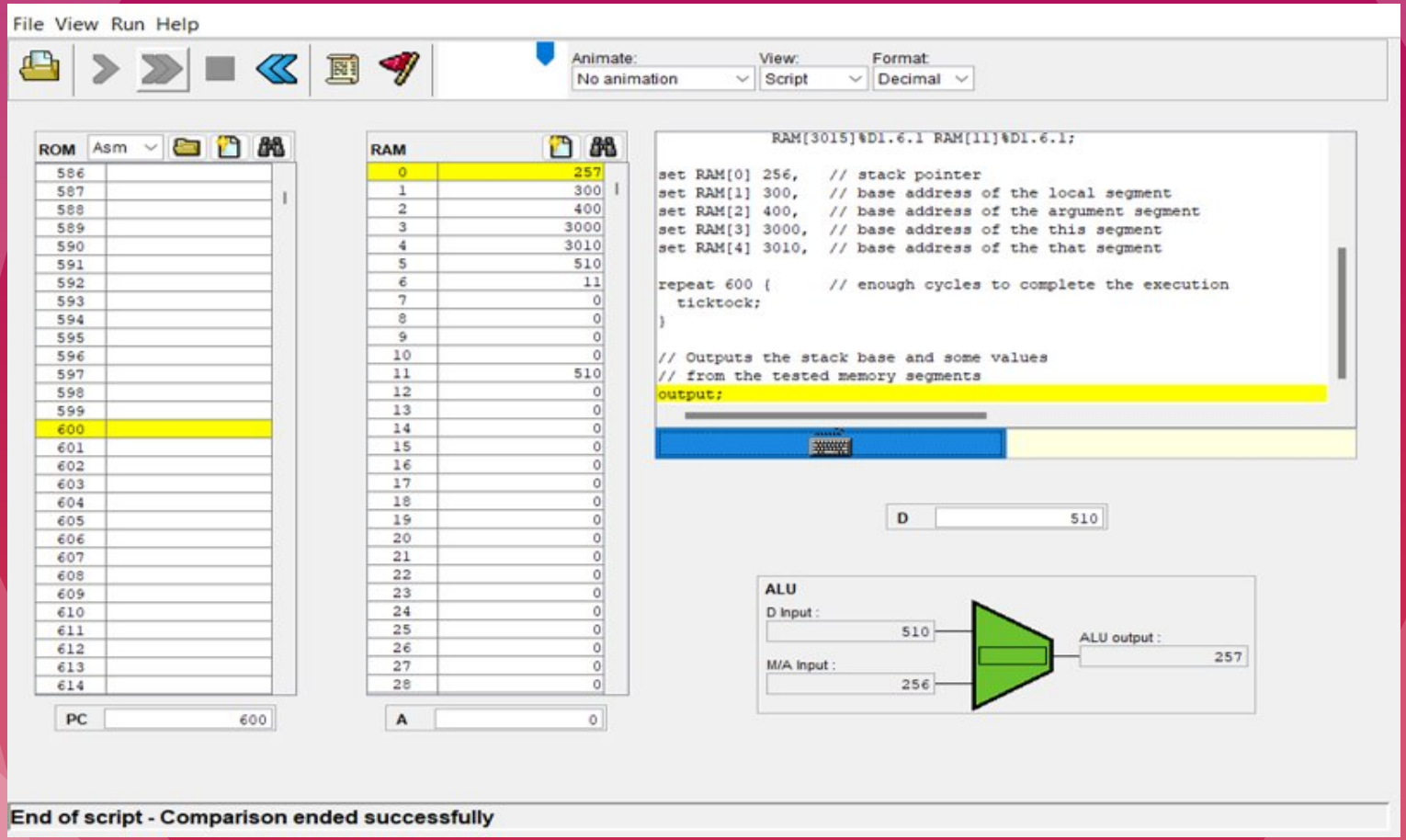
BASIC TEST


CPU Emulator (2.5)
File View Run Help
Slow
Fast
Animate:
Program flow
View:
Script
Format:
Decimal
ROM
Asm
RAM
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/07/MemoryAccess/BasicTest/BasicTest.tst
load BasicTest.asm,
output-file BasicTest.out,
compare-to BasicTest.cmp,
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 RAM[401]%D1.6.1
RAM[402]%D1.6.1 RAM[3006]%D1.6.1 RAM[3012]%D1.6.1
RAM[3015]%D1.6.1 RAM[11]%D1.6.1;
set RAM[0] 256, // stack pointer
set RAM[1] 300, // base address of the local segment
set RAM[2] 400, // base address of the argument segment
D
ALU
D Input :
M/A Input :
ALU output :
PC
A
New script loaded: C:\Users\HP\Desktop\Amrita\AMRITA APPS\nand2tetris\nand2tetris\projects\07\MemoryAccess\BasicTest\Basic..

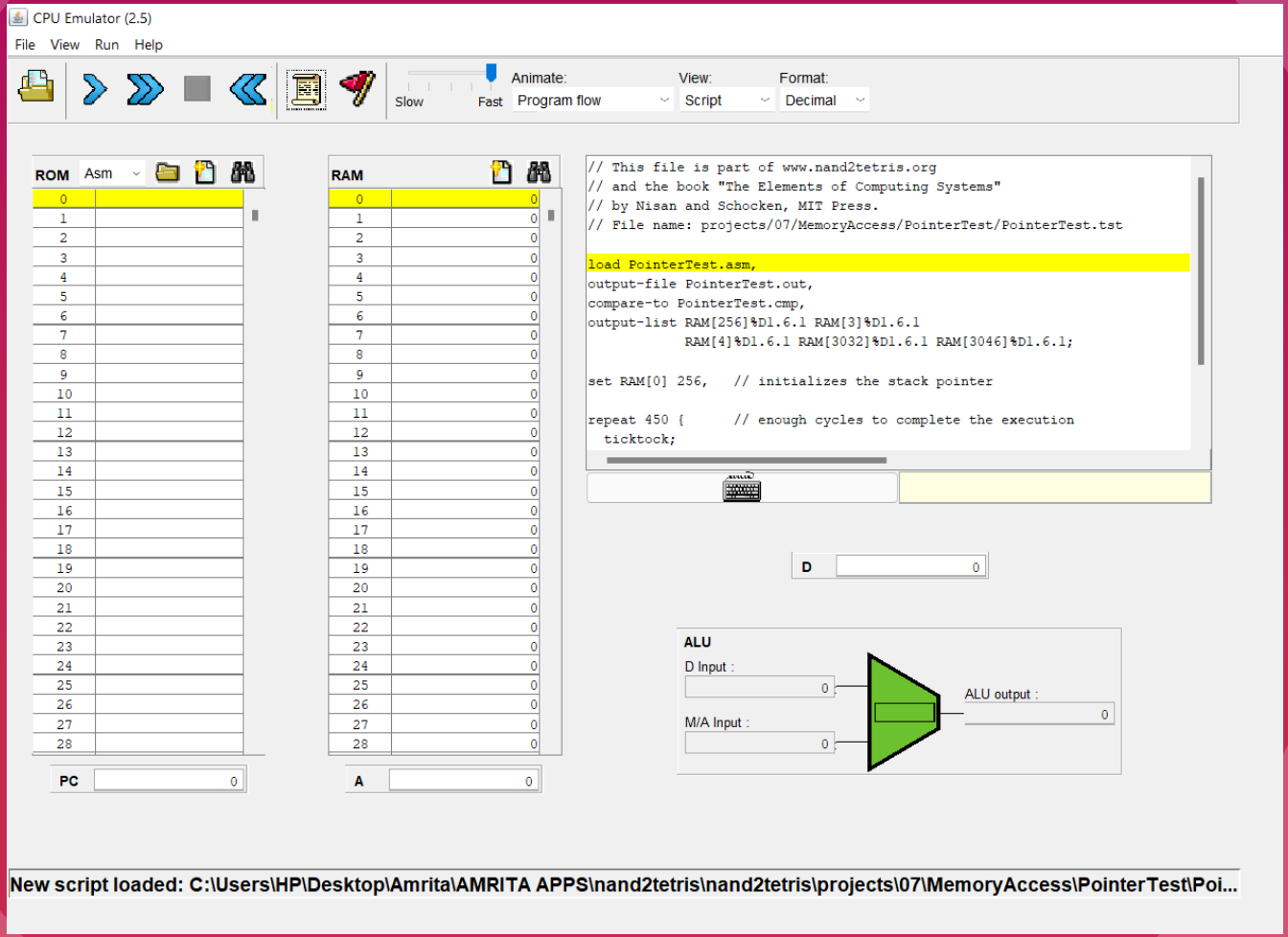
BASIC TEST


CPU Emulator (2.5) - C:\Users\HP\Desktop\Amrita\AMRITA APPS\nand2tetris\nand2tetris\projects\07\MemoryAccess\BasicTest/BasicTest.asm
File View Run Help
Animate:
Slow
Fast
Program flow
View:
Script
Format:
Decimal
ROM
Asm
105 M=D
106 @2
107 D=A
108 @4
109 D=D+M
110 @14
111 M=D
112 @0
113 AM=M-1
114 D=M
115 @14
116 A=M
117 M=D
118 @510
119 D=A
120 @0
121 A=M
122 M=D
123 @0
124 M=M+1
125 @11
126 D=A
127 @14
128 M=D
129 @0
130 AM=M-1
131 D=M
132 @14
133 A=M
PC 132
RAM
242 0
243 0
244 0
245 0
246 0
247 0
248 0
249 0
250 0
251 0
252 0
253 0
254 0
255 0
256 510
257 45
258 0
259 0
260 0
261 0
262 0
263 0
264 0
265 0
266 0
267 0
268 0
269 0
270 0
A 256
output-list RAM[256]%D1.6.1 RAM[300]%D1.6.1 RAM[401]%D1.6.1
RAM[402]%D1.6.1 RAM[3006]%D1.6.1 RAM[3012]%D1.6.1
RAM[3015]%D1.6.1 RAM[11]%D1.6.1;
set RAM[0] 256, // stack pointer
set RAM[1] 300, // base address of the local segment
set RAM[2] 400, // base address of the argument segment
set RAM[3] 3000, // base address of the this segment
set RAM[4] 3010, // base address of the that segment
repeat 600 { // enough cycles to complete the execution
ticktock;
}
// Outputs the stack base and some values
D 510
ALU
D Input :
11
M/A Input :
510
M
ALU output :
510

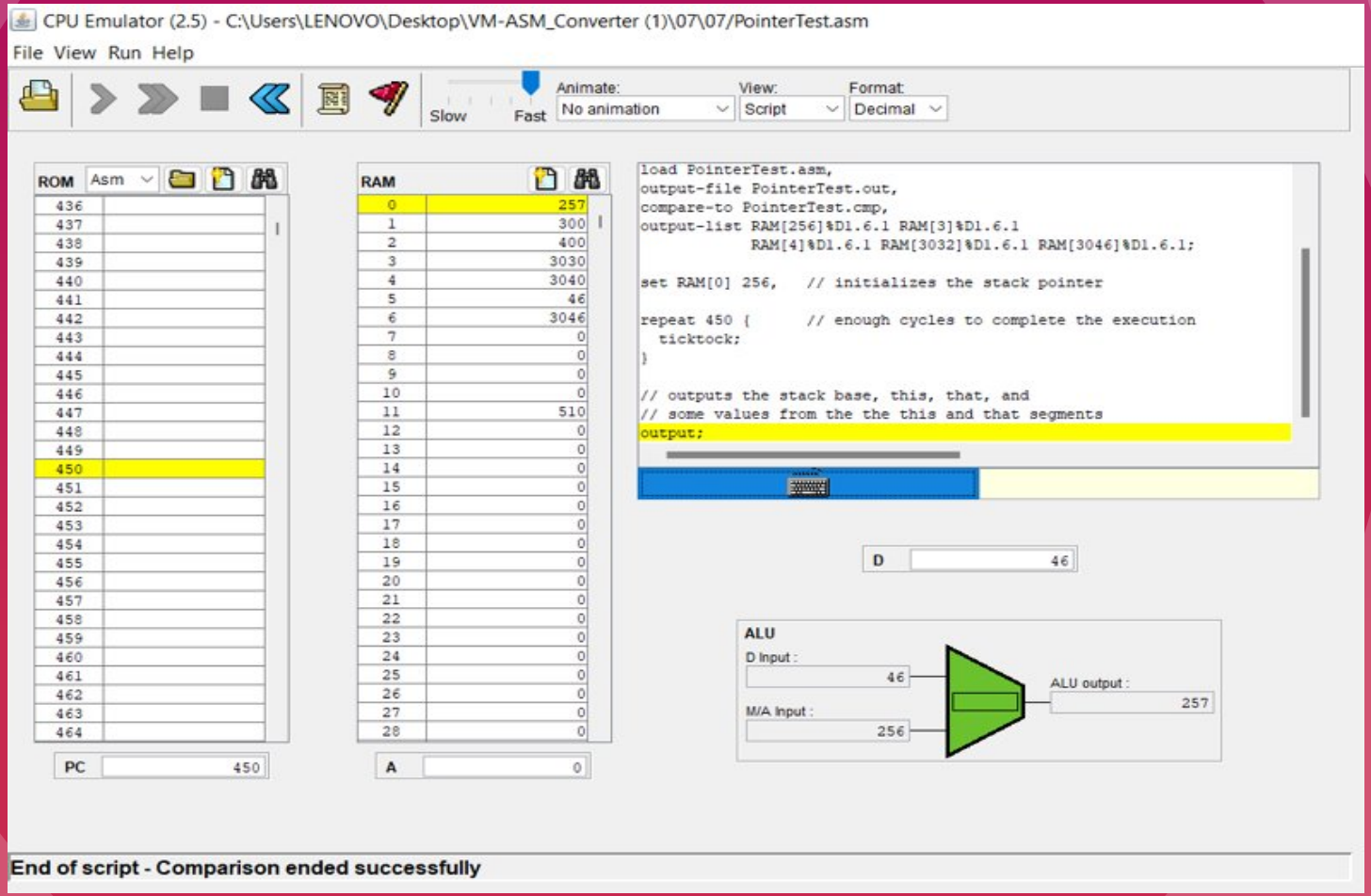
BASIC TEST


File View Run Help
Animate:
No animation
View:
Script
Format:
Decimal
ROM
Asm
RAM
RAM[3015]%D1.6.1 RAM[11]%D1.6.1;
set RAM[0] 256, // stack pointer
set RAM[1] 300, // base address of the local segment
set RAM[2] 400, // base address of the argument segment
set RAM[3] 3000, // base address of the this segment
set RAM[4] 3010, // base address of the that segment
repeat 600 { // enough cycles to complete the execution
ticktock;
}
// Outputs the stack base and some values
// from the tested memory segments
output;
D
510
ALU
D Input :
510
M/A Input :
256
ALU output :
257
PC
600
A
0
End of script - Comparison ended successfully

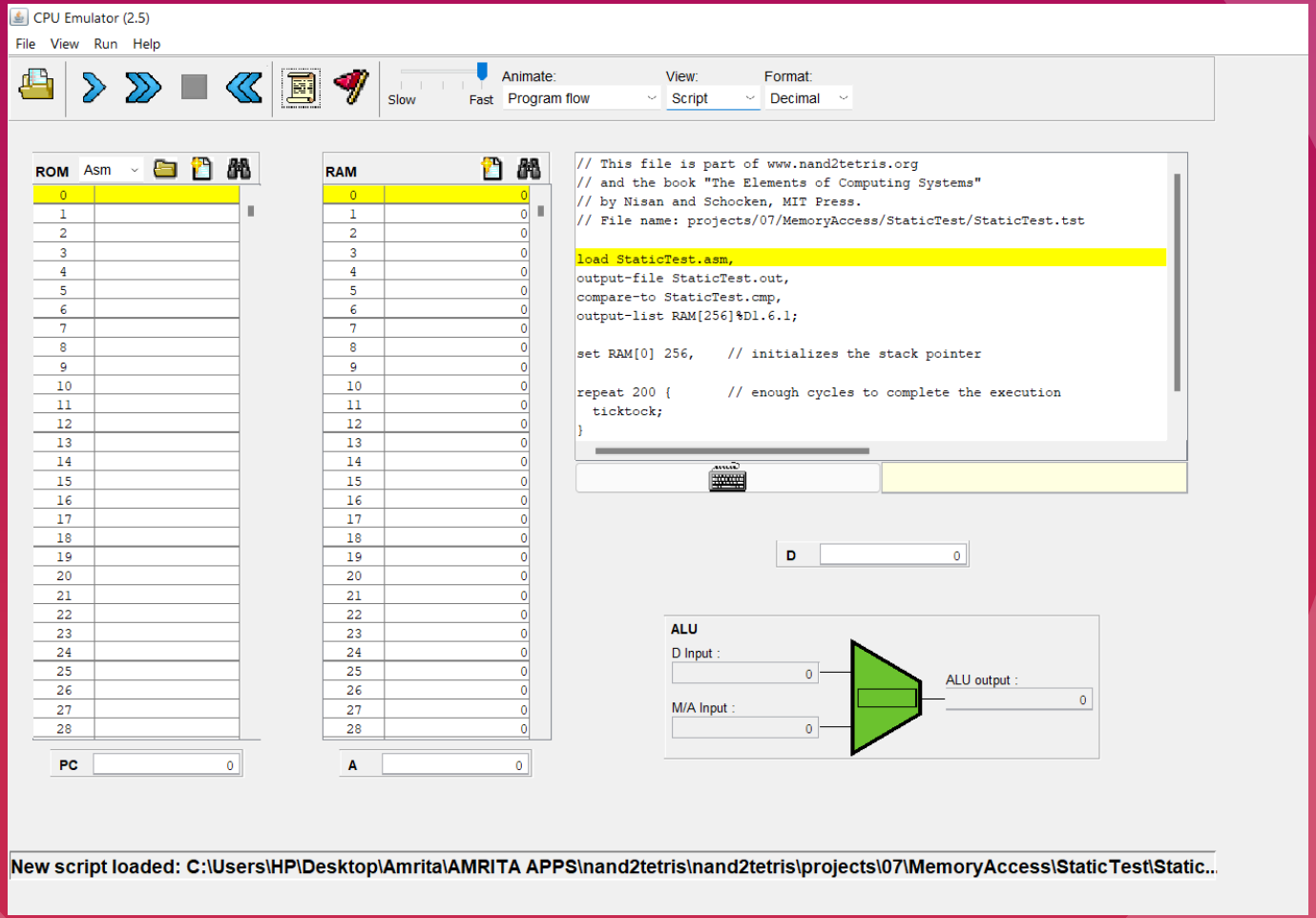
POINTER TEST



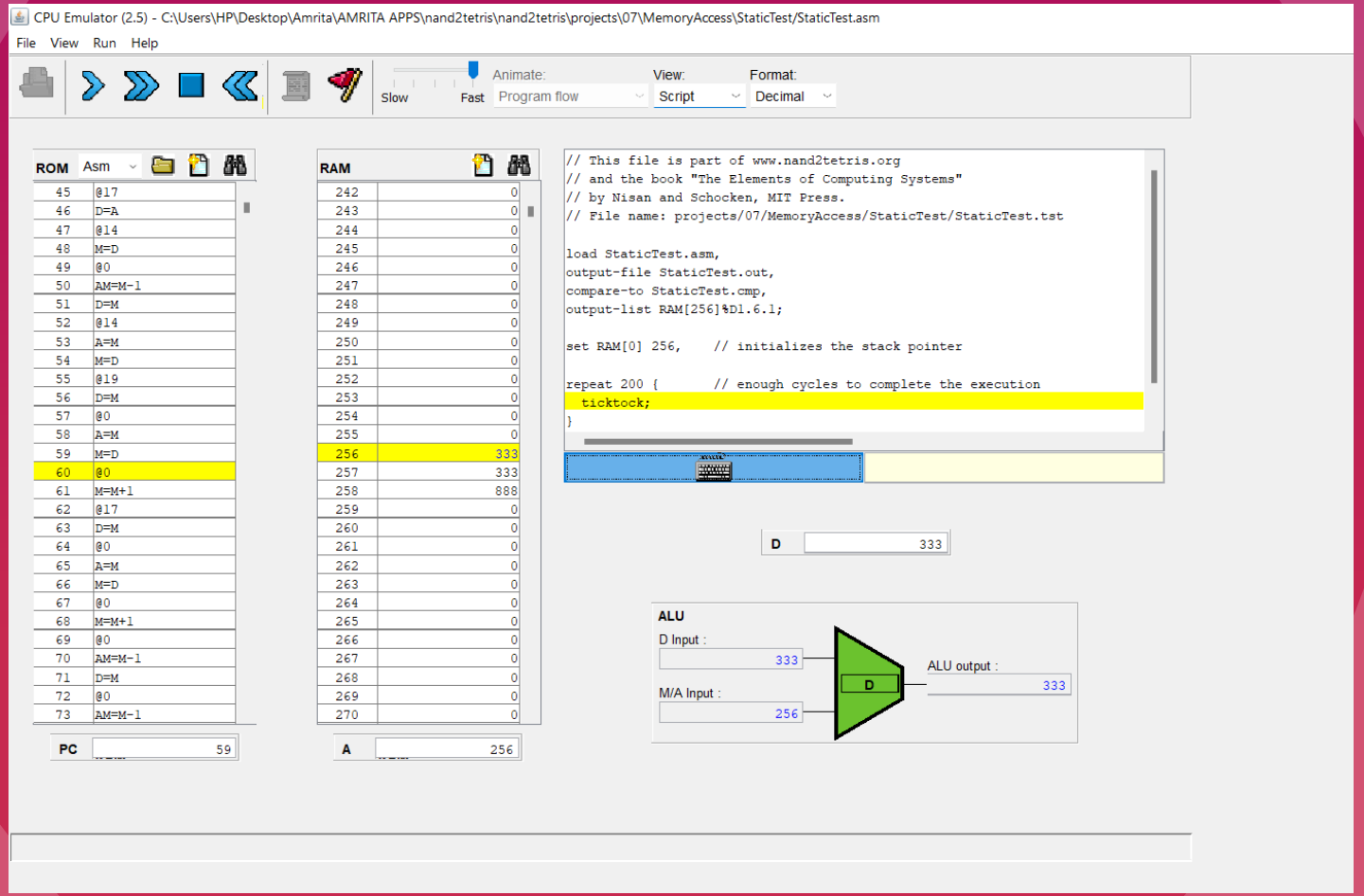
POINTER TEST



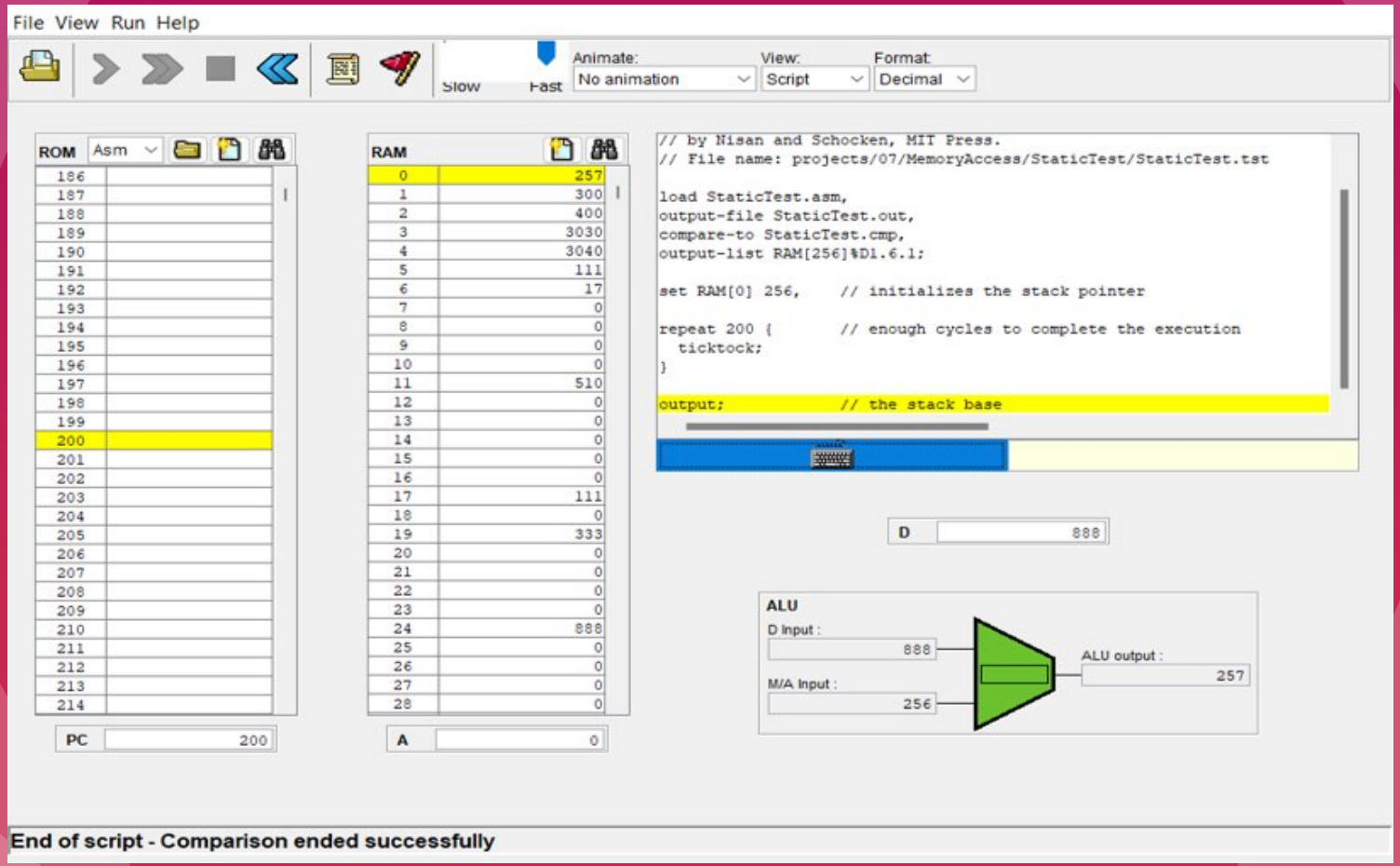
STATIC TEST


CPU Emulator (2.5)
File View Run Help
Animate:
View:
Format:
Slow
Fast
Program flow
Script
Decimal
ROM
Asm
RAM
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/07/MemoryAccess/StaticTest/StaticTest.tst
load StaticTest.asm,
output-file StaticTest.out,
compare-to StaticTest.cmp,
output-list RAM[256]%D1.6.1;
set RAM[0] 256, // initializes the stack pointer
repeat 200 { // enough cycles to complete the execution
ticktock;
}
D
ALU
D Input :
M/A Input :
ALU output :
PC
A
New script loaded: C:\Users\HP\Desktop\Amrita\AMRITA APPS\nand2tetris\nand2tetris\projects\07\MemoryAccess\StaticTest\Static...

STATIC TEST


CPU Emulator (2.5) - C:\Users\HP\Desktop\Amrita\AMRITA APPS\nand2tetris\nand2tetris\projects\07\MemoryAccess\StaticTest/StaticTest.asm
File View Run Help
Animate: Slow Fast Program flow
View: Script
Format: Decimal
ROM Asm
45 @17
46 D=A
47 @14
48 M=D
49 @0
50 AM=M-1
51 D=M
52 @14
53 A=M
54 M=D
55 @19
56 D=M
57 @0
58 A=M
59 M=D
60 @0
61 M=M+1
62 @17
63 D=M
64 @0
65 A=M
66 M=D
67 @0
68 M=M+1
69 @0
70 AM=M-1
71 D=M
72 @0
73 AM=M-1
PC 59
RAM
242 0
243 0
244 0
245 0
246 0
247 0
248 0
249 0
250 0
251 0
252 0
253 0
254 0
255 0
256 333
257 333
258 888
259 0
260 0
261 0
262 0
263 0
264 0
265 0
266 0
267 0
268 0
269 0
270 0
A 256
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/07/MemoryAccess/StaticTest/StaticTest.tst
load StaticTest.asm,
output-file StaticTest.out,
compare-to StaticTest.cmp,
output-list RAM[256]%D1.6.1;
set RAM[0] 256, // initializes the stack pointer
repeat 200 { // enough cycles to complete the execution
ticktock;
}
D 333
ALU
D Input : 333
M/A Input : 256
D
ALU output : 333

STATIC TEST


File View Run Help
Animate:
No animation
View:
Script
Format:
Decimal
Slow
Fast
ROM
Asm
RAM
// by Nisan and Schocken, MIT Press.
// File name: projects/07/MemoryAccess/StaticTest/StaticTest.tst
load StaticTest.asm,
output-file StaticTest.out,
compare-to StaticTest.cmp,
output-list RAM[256]%D1.6.1;
set RAM[0] 256, // initializes the stack pointer
repeat 200 { // enough cycles to complete the execution
ticktock;
}
output; // the stack base
D
888
ALU
D Input :
888
M/A Input :
256
ALU output :
257
PC
200
A
0
End of script - Comparison ended successfully

Explain in detail various memory segments and its VM Implementation and assembly pseudo codes.

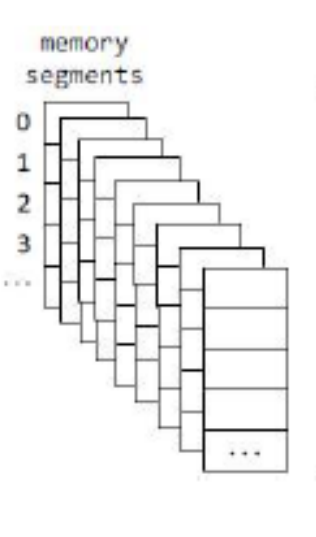
Vikhyat Bansal

2

Virtual Memory Segments

Virtual:
not physically existing as such but made by software to appear to do so.

Memory Segments:
Memory is one of the most important resources on a computing system, and its management is primary in every environment. In a bid to use memory efficiently and effectively, a number of techniques have been developed to properly manage it. One of these memory management techniques is known as memory segmentation (MS).
Memory segmentation is a system of segmenting processes that loads information into different addressed spaces in memory.
Once the process of segmentation occurs, the entire process can be loaded into different areas in memory instead of one contiguous space. Loading smaller segments of the process into memory allows the physical memory to be used more efficiently.



Both the definitions together create Virtual memory segments of VM in HACK programming language

Different Memory Segments

- Argument:

Segment	Purpose	Comments
`argument`	Stores the function's arguments.	Allocated dynamically by the VM implementation when the function is entered.

- Local:

Segment	Purpose	Comments
`local`	Stores the function's local variables.	Allocated dynamically by the VM implementation and initialized to 0's when the function is entered.

- Static:

Segment	Purpose	Comments
`static`	Stores static variables shared by all functions in the same `.vm` file.	Allocated by the VM imp. for each `.vm` file; shared by all functions in the `.vm` file.

Different Memory Segments

- This/That:

Segment	Purpose	Comments
`this` `that`	General-purpose segments. Can be made to correspond to different areas in the heap. Serve various programming needs.	Any VM function can use these segments to manipulate selected areas on the heap.

- Pointer:

Segment	Purpose	Comments
`pointer`	A two-entry segment that holds the base addresses of the `this` and `that` segments.	Any VM function can set `pointer` 0 (or 1) to some address; this has the effect of aligning the `this` (or `that`) segment to the heap area beginning in that address.

Different Memory Segments

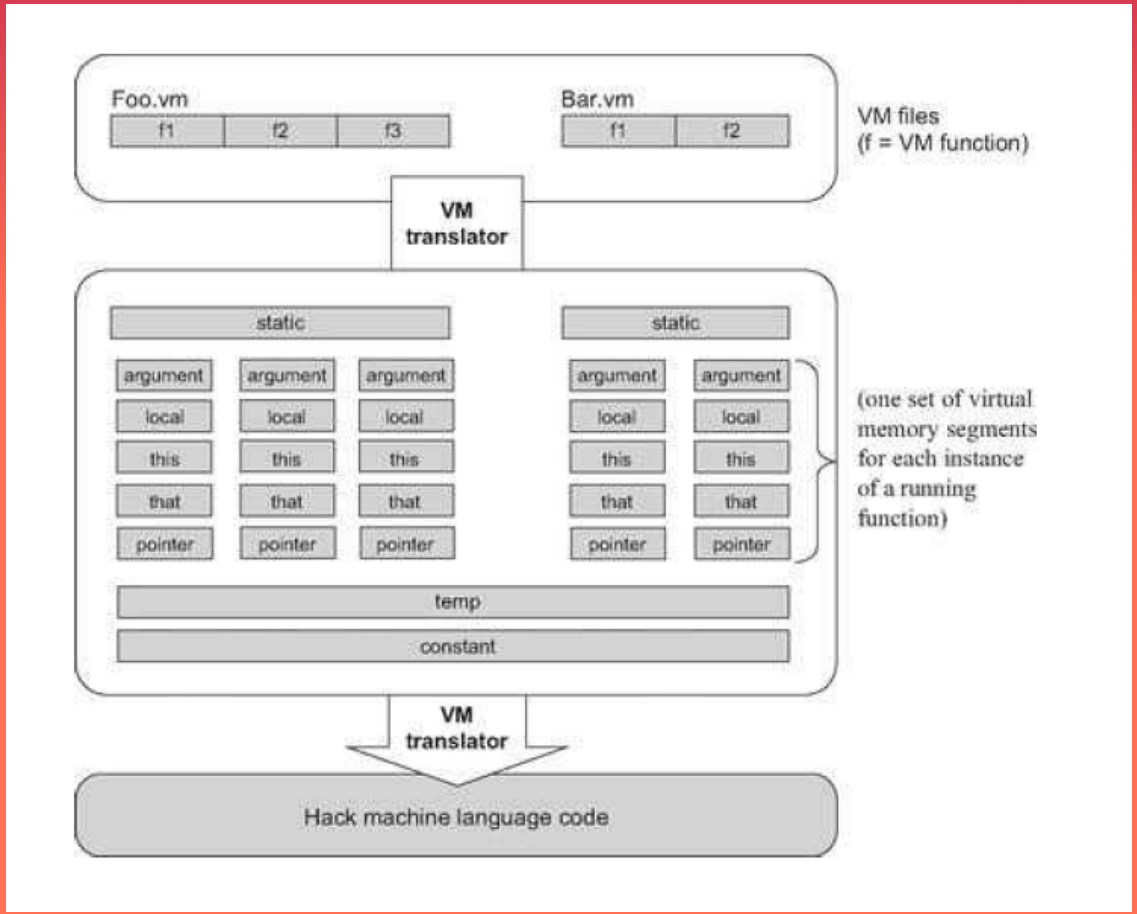
- Temp:

Segment	Purpose	Comments
temp	Fixed eight-entry segment that holds temporary variables for general use.	May be used by any VM function for any purpose. Shared by all functions in the program.

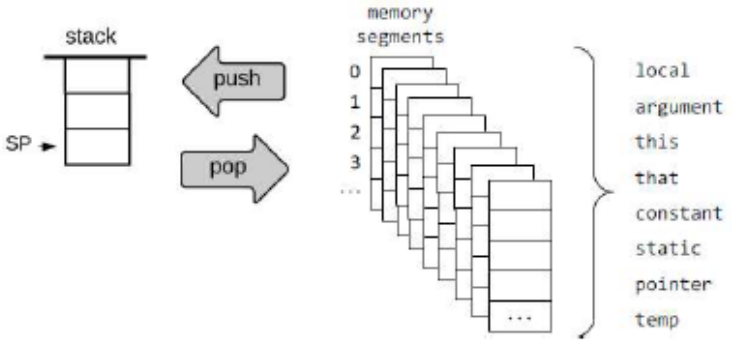
- Constant

Segment	Purpose	Comments
constant	Pseudo-segment that holds all the constants in the range 0...32767.	Emulated by the VM implementation; Seen by all the functions in the program.

The virtual memory segments are maintained by the VM implementation.



Memory segments



Syntax

push *segment i*

Where *segment* is: argument, local, static, constant, this, that, pointer, or temp

and *i* is a non-negative integer.

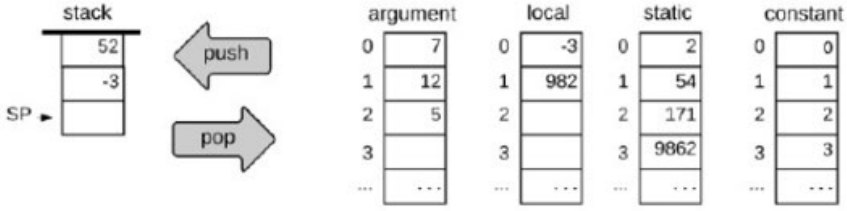
pop *segment i*

Where *segment* is: argument, local, static, this, that, pointer, or temp

and *i* is a non-negative integer.

Eight memory segments, which are managed explicitly by, VM push and pop commands

Example:

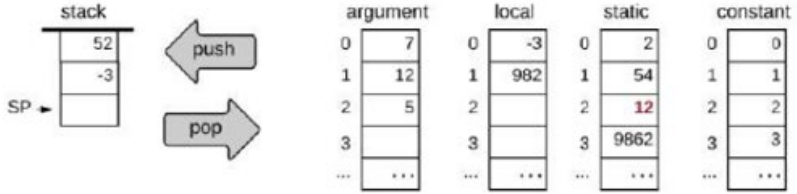


let static 2 = argument 1

?

let static 2 = argument 1

```
push argument 1
pop static 2
```



Variable kinds

Source code (Jack)

```
class Foo {
   static int s1, s2;
   function int bar (int x, int y) {
      var int a, b, c;
      ...
      let c = s1 + y;
      ...
   }
}
```

Compiled VM code

```
...
...
...
...
push s1
push y
add
pop c
...
...
```

Variable kinds

- Argument variables
- Local variables
- Static variables

Variable kinds and memory segments

Source code (Jack)

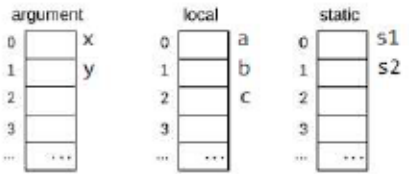
```
class Foo {
   static int s1, s2;
   function int bar (int x, int y) {
      var int a, b, c;
      ...
      let c = s1 + y;
      ...
   }
}
```

compile

Compiled VM code

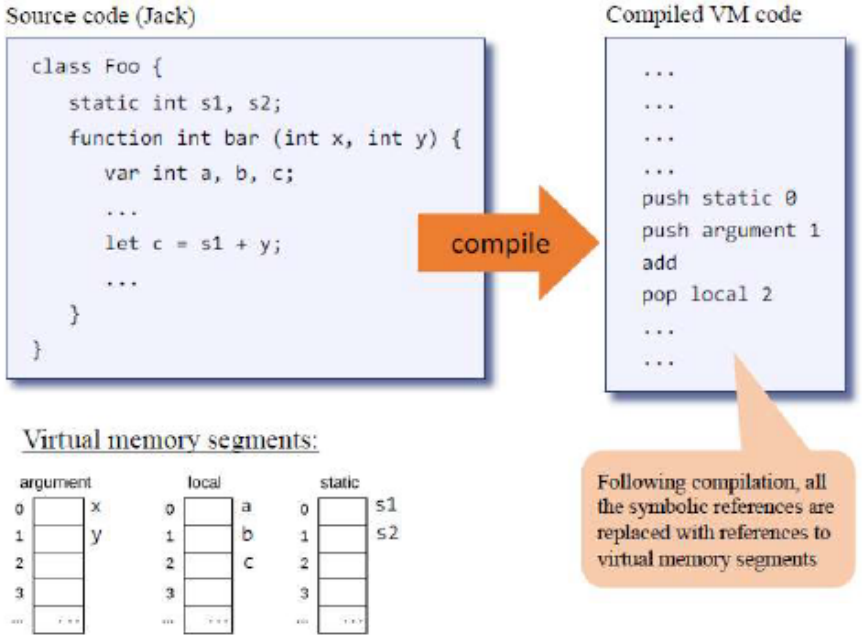
```
...
...
...
...
push s1
push y
add
pop c
...
...
```

Virtual memory segments:



Jack code compiled to VM code using JackCompiler in software suite of nand2tetris.

Variable kinds and memory segments



All variable kinds in form of symbols are replaced by references to memory segments.

RAM USAGE by Memory Segments in VM

RAM Usage The data memory of the Hack computer consists of 32K 16-bit words. The first 16K serve as general-purpose RAM. The next 16K contain memory maps of I/O devices. The VM implementation should use this space as follows:

RAM addresses	*Usage*
0–15	Sixteen virtual registers, usage described below
16–255	Static variables (of all the VM functions in the VM program)
256–2047	Stack
2048–16483	Heap (used to store objects and arrays)
16384–24575	Memory mapped I/O

Recall that according to the Hack Machine Language Specification, RAM addresses 0 to 15 can be referred to by any assembly program using the symbols R0 to R15, respectively. In addition, the specification states that assembly programs can refer to RAM addresses 0 to 4 (i.e., R0 to R4) using the symbols SP, LCL, ARG, THIS, and THAT. This convention was introduced into the assembly language with foresight, in order to promote readable VM implementations. The expected use of these registers in the VM context is described as follows:

Register	*Name*	*Usage*
RAM[0]	`SP`	Stack pointer: points to the next topmost location in the stack;
RAM[1]	`LCL`	Points to the base of the current VM function's `local` segment;
RAM[2]	`ARG`	Points to the base of the current VM function's `argument` segment;
RAM[3]	`THIS`	Points to the base of the current `this` segment (within the heap);
RAM[4]	`THAT`	Points to the base of the current `that` segment (within the heap);
RAM[5–12]		Holds the contents of the `temp` segment;
RAM[13–15]		Can be used by the VM implementation as general-purpose registers.

Memory Segment Mapping

Memory Segments Mapping

local, argument, this, that: Each one of these segments is mapped directly on the RAM, and its location is maintained by keeping its physical base address in a dedicated register (LCL, ARG, THIS, and THAT, respectively). Thus any access to the *i*th entry of any one of these segments should be translated to assembly code that accesses address (*base* + i) in the RAM, where base is the current value stored in the register dedicated to the respective segment.

pointer, temp: These segments are each mapped directly onto a fixed area in the RAM. The pointer segment is mapped on RAM locations 3-4 (also called THIS and THAT) and the temp segment on locations 5-12 (also called R5, R6,..., R12). Thus access to pointer i should be translated to assembly code that accesses RAM location 3 + *i*, and access to temp i should be translated to assembly code that accesses RAM location 5 + i.

constant: This segment is truly virtual, as it does not occupy any physical space on the target architecture. Instead, the VM implementation handles any VM access to ⟨*constant i*⟩ by simply supplying the constant i.

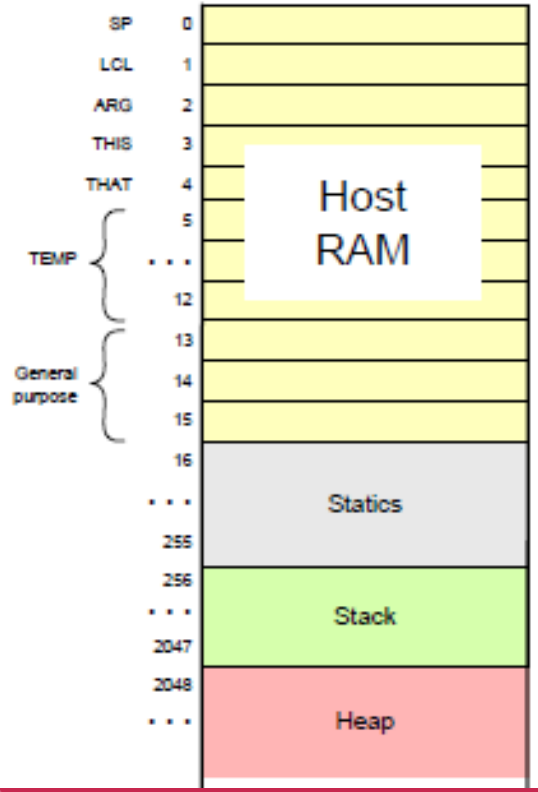
static: According to the Hack machine language specification, when a new symbol is encountered for the first time in an assembly program, the assembler allocates a new RAM address to it, starting at address 16. This convention can be exploited to represent each static variable number j in a VM file f as the assembly language symbol f.j. For example, suppose that the file Xxx.vm contains the command push static 3. This command can be translated to the Hack assembly commands@Xxx.3 and D=M, followed by additional assembly code that pushes D's value to the stack. This implementation of the static segment is somewhat tricky, but it works.

The VM implementation manages two implicit data structures called stack and heap. These data structures are never mentioned directly, but their states change in the background, as a side effect of VM commands.

The Stack Consider the commands sequence push argument 2 and pop local 1, mentioned before. The working memory of such VM operations is the stack. The data value did not simply jump from one segment to another—it went through the stack. Yet in spite of its central role in the VM architecture, the stack proper is never mentioned in the VM language.

The Heap Another memory element that exists in the VM's background is the *heap*. The heap is the name of the RAM area dedicated for storing objects and arrays data. These objects and arrays can be manipulated by VM commands, as we will see shortly.

VM implementation on the Hack platform (memory)



Basic idea: the mapping of the stack and the global segments on the RAM is easy (fixed); the mapping of the function-level segments is dynamic, using pointers

`The stack`: mapped on `RAM[256 ... 2047]`;
The stack pointer is kept in RAM address SP

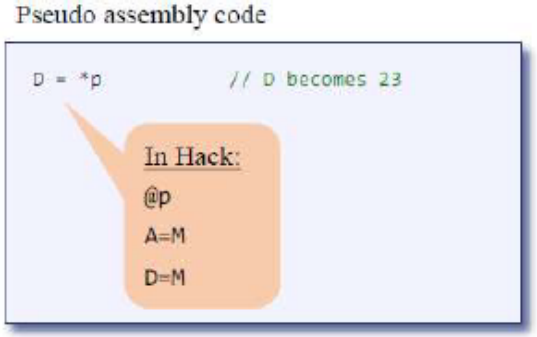
`static`: mapped on `RAM[16 ... 255]`;
each segment reference `static i` appearing in a VM file named `f` is compiled to the assembly language symbol `f.i` (recall that the assembler further maps such symbols to the RAM, from address 16 onward)

`local, argument, this, that`: these method-level segments are mapped somewhere from address 2048 onward, in an area called "heap". The base addresses of these segments are kept in RAM addresses LCL, ARG, THIS, and THAT. Access to the i-th entry of any of these segments is implemented by accessing `RAM[segmentBase + i]`

`constant`: a truly a virtual segment:
access to `constant i` is implemented by supplying the constant i.

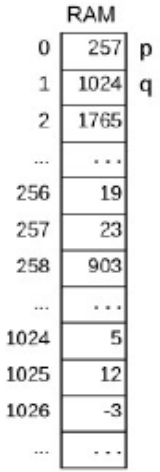
Pointer Manipulation

Before moving ahead with implementation of memory segments, it is necessary to go through and learn about pointer manipulation and stack implementation.



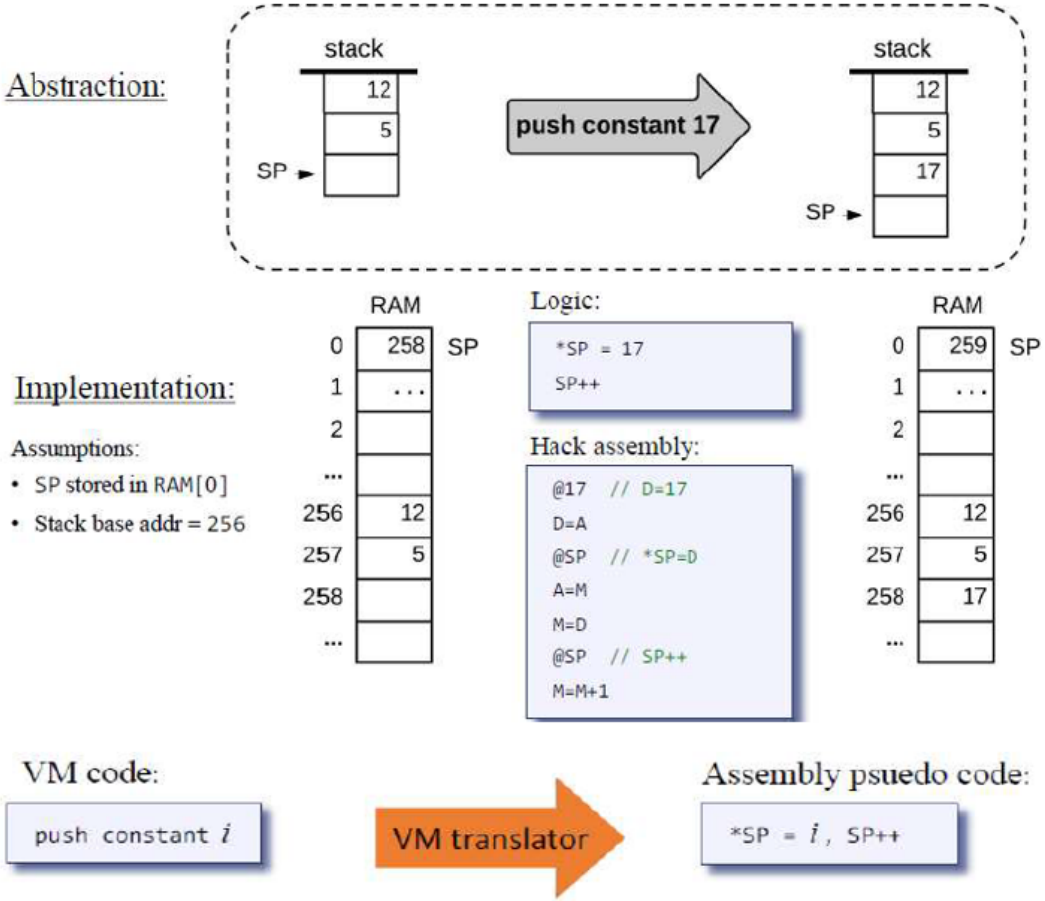
Notation:

*p // the memory location that p points at



P is pointing at RAM[0]
now A = 257
now M = RAM[257]
that is why D = 23

Stack Implementation



Memory Segment Implementation

Implementing Local

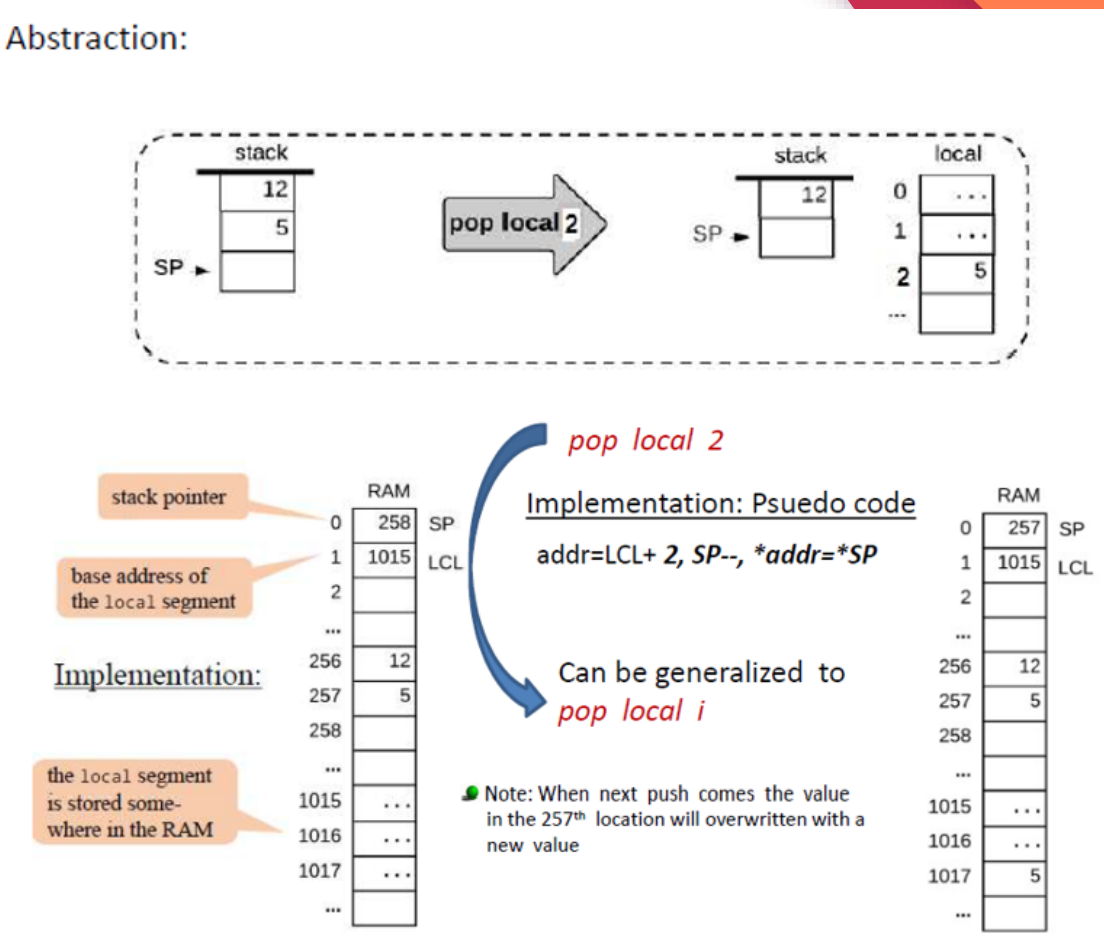
RAM[1] points to the base of the current VM function's local segment.

Here, pop is used to take value from stack (starting from RAM[256]) and store that value in Local memory segment address in RAM[1].

Push is used to store value in stack (starting from RAM{256}), value is taken from RAM[1] i.e., from LCL memory segment address.

Working of Pseudo Code for pop:

- (addr = LCL +2): Access to the i^{th} entry of Local segment is implemented by accessing RAM[LCL +i].
- (SP--) : A particular address from stack is accessed after performing (SP – 1) on stack pointer address at RAM[0].
- (*addr=*SP): Local i^{th} entry (RAM address of LCL segment) has now become equal to value stored in Address present in stack pointer.



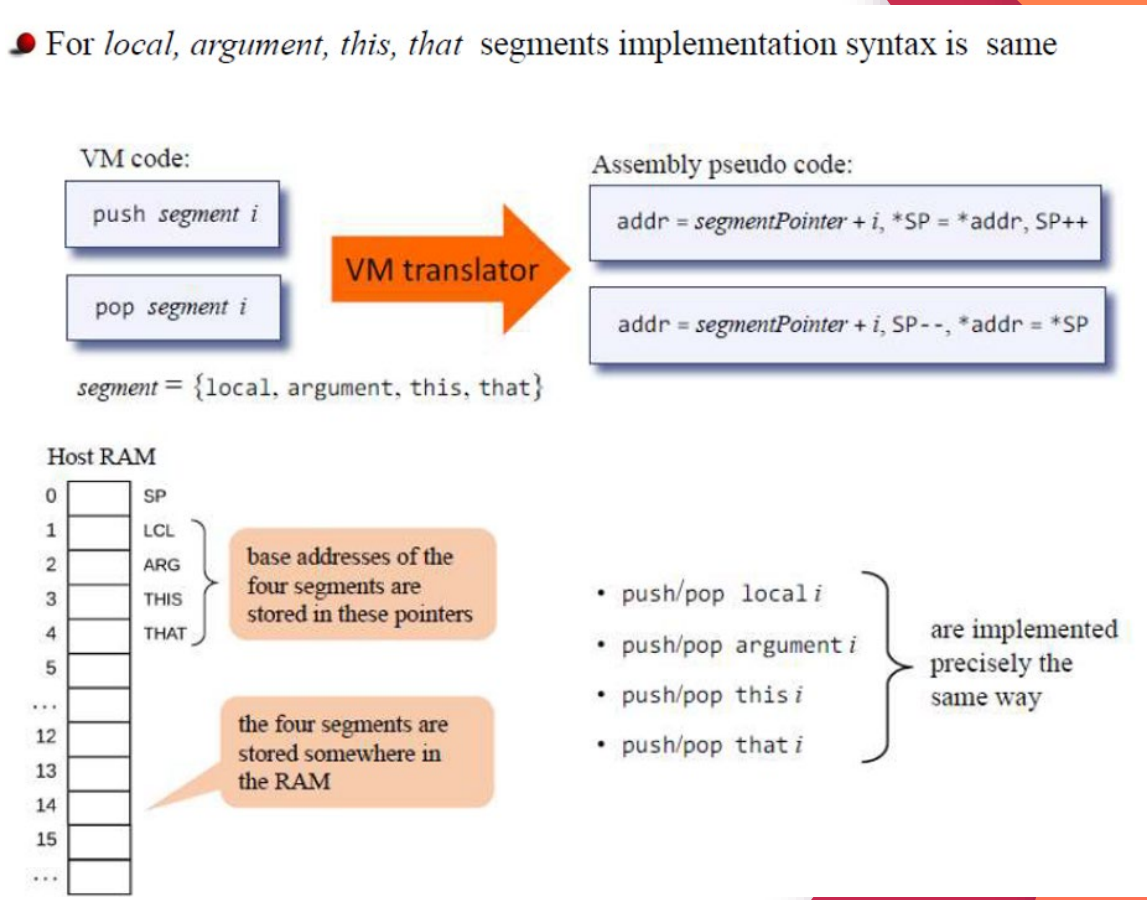
Memory Segment Implementation

Implementing: *argument, this, that*

RAM[2],RAM[3],RAM[4] points to the base of the current VM function's argument, this and that segment respectively.

Working of Pseudo Code for push:

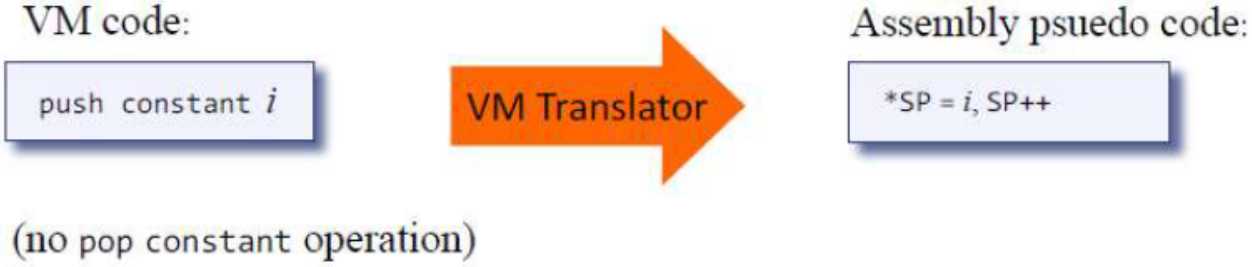
- (addr = segmentPointer +i):
  Access to the i^{th} entry of any of these segments is implemented by accessing RAM[segmentbase +i].
- (*SP=*addr):
  Value stored in Address present in stack pointer becomes equal to value stored in RAM address of memory segments.
- (SP++) : A particular address from stack is accessed after performing (SP + 1) on stack pointer address at RAM[0].



Memory Segment Implementation

Implementing: *Constant*

- Constant segments are used when the HL compiler encounters constants
- VM code: *push constant i* ***(there is no such command pop constant)***



This segment is truly virtual and it simply just supply constant I and does not take any physical space.

Memory Segment Implementation

Implementing: *Static*

Mapped from RAM[16] to RAM[255]

> static: According to the Hack machine language specification, when a new symbol is encountered for the first time in an assembly program, the assembler allocates a new RAM address to it, starting at address 16. This convention can be exploited to represent each static variable number j in a VM file f as the assembly language symbol f.j. For example, suppose that the file Xxx.vm contains the command push static 3. This command can be translated to the Hack assembly commands@Xxx.3 and D=M, followed by additional assembly code that pushes D's value to the stack. This implementation of the static segment is somewhat tricky, but it works.

VM code:

```
// File Foo.vm
...
pop static 5
...
pop static 2
...
```

VM translator

Generated assembly code:

```
...
// D = stack.pop (code omitted)
@Foo.5
M=D
...
// D = stack.pop (code omitted)
@Foo.2
M=D
...
```

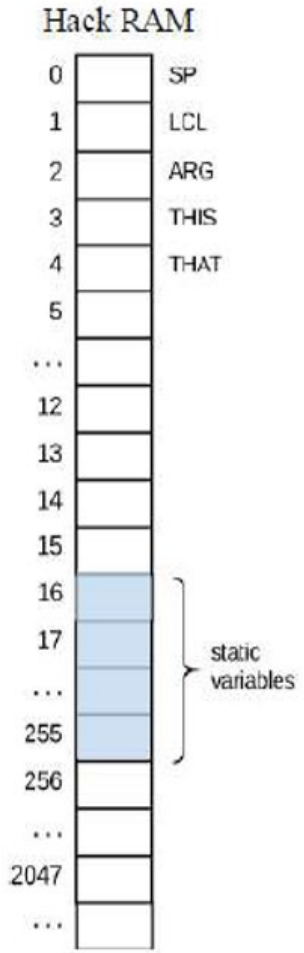
The challenge:

static variables should be seen by all the methods in a program

Solution:

Store them in some "global space":

- Have the VM translator translate each VM reference `static i` (in file `Foo.vm`) into an assembly reference `Foo.i`
- Following assembly, the Hack assembler will map these references onto `RAM[16]`, `RAM[17]`, ..., `RAM[255]`
- Therefore, the entries of the `static` segment will end up being mapped onto `RAM[16]`, `RAM[17]`, ..., `RAM[255]`, in the order in which they appear in the program.

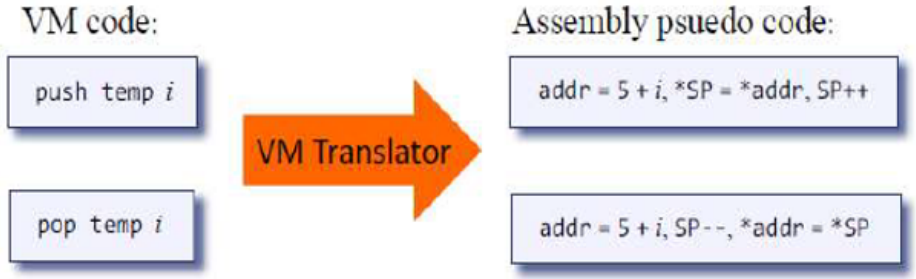


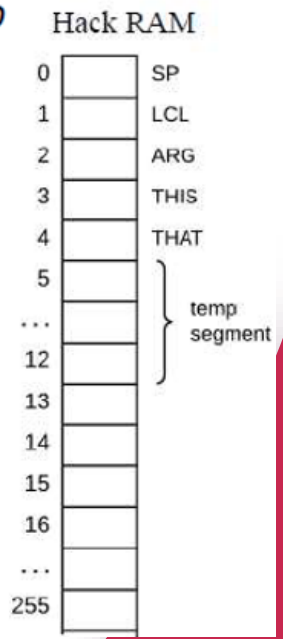
Memory Segment Implementation

Implementing: *temp*

- Fixed eight entry segment that holds temporary variables for general use.
- Mapped from RAM[5] to RAM[12]
- Working of assembly code is similar to that of local, argument, this and that segment.

• Our VM provides 8 such variables, stored in a segment named *temp*





• stored in RAM locations 5 to 12

Memory Segment Implementation

Implementing: *pointer*

RAM[3] points to the base of the current *this* segment (within the heap).
RAM[4] points to the base of the current *that* segment (within the heap).

- We use it for storing the base addresses of the segments 'this' and 'that'

Working of Pseudo Code for push:

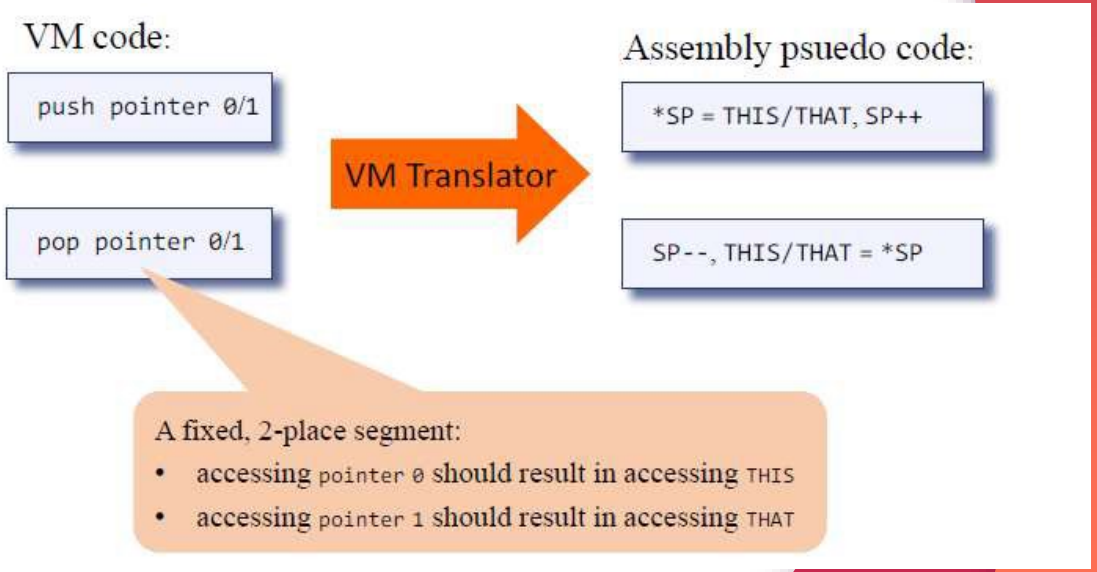
- (*SP=THIS/THAT):
  Value stored in Address present in stack pointer becomes equal to base address of segments this/that.

- (SP++) : A particular address from stack is accessed after performing (SP + 1) on stack pointer address at RAM[0].

Working of Pseudo Code for pop:

- (SP--): A particular address from stack is accessed after performing (SP – 1) on stack pointer address at RAM[0].

- (THIS/THAT=*SP): Base address of segments this/that has now become equal to value stored in Address present in stack pointer.



3. Explain with the help of a stack diagram, argument how the following snippet of code works. [CO2]

```
function main 0
   push constant 3
   call factorial 1
   return
function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```

High level programme

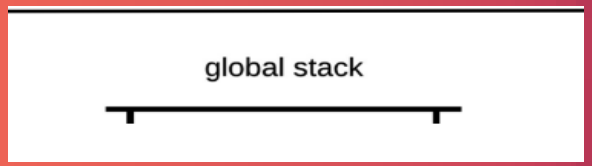
```
// Tests the factorial function
int main() {
   return factorial(3);
}

// Returns n!
int factorial(int n) {
   if (n==1)
      return 1;
   else
      return n * factorial(n-1);
}
```

Explanation

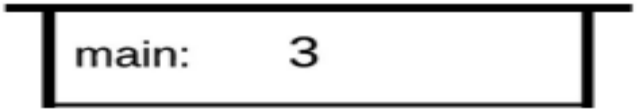
So basically we have Main function and factorial function this factorial function is inside the main function .

In the given code first line function main 0 represents that there are no local variables and global stack is empty

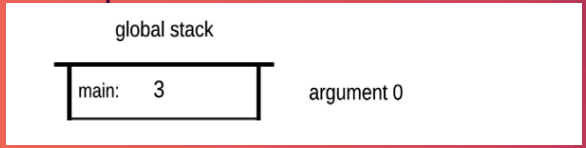
The picture of stack is given below



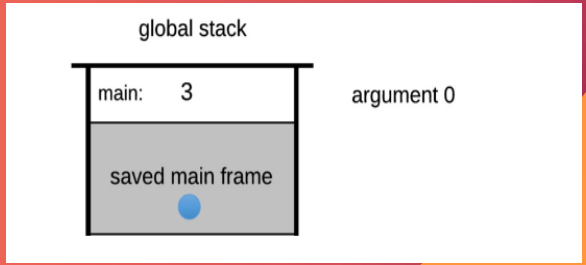
We can see that global stack is empty and after that we are pushing 3 into the stack so we can see below that 3 is pushed into global stack



So 3 is pushed into the stack and after that we have call factorial 1 so when
There is call factorial we should look for number here it is 1 what it is indicating
Is the argument present above so here there is only one 1 argument in the main
Function that is 3 the global stack picture is as below



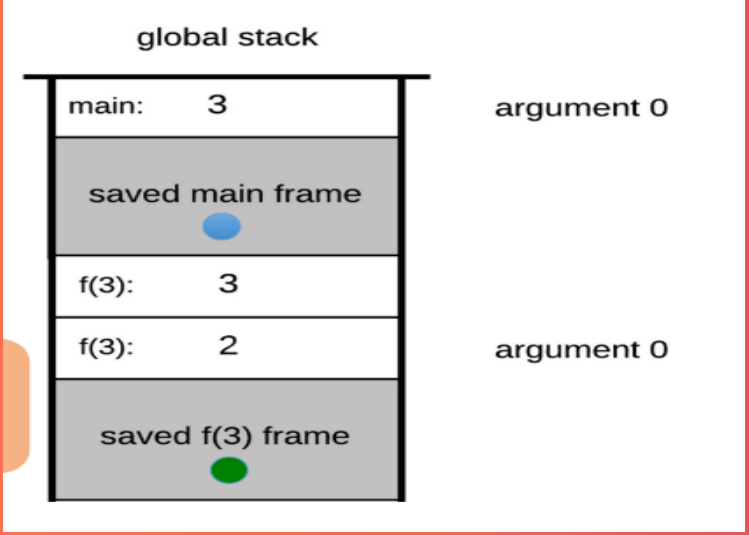
After that we need to return that and we have to save that main frame the stack diagram is as below



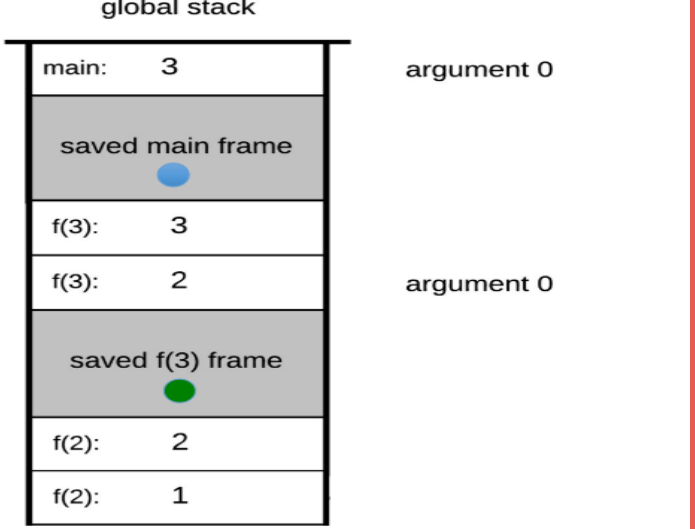
Next is factorial function

Here in the code it is mentioned that function factorial 0 it means that we don't Have any local variables here after that we are pushing argument 0 into the stack Here argument 0 is 3 and next we are pushing constant 1 into the stack.

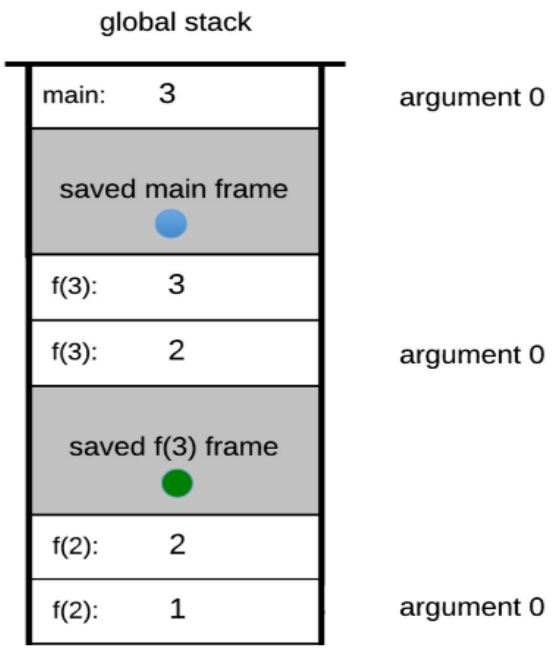
Next we are checking whether they are equal or not if they are equal then if-goto Base case otherwise goto else condition. So in this case argument 0 is 3 and Constant 1 both are not equal so go to else condition we are pushing argument 0 2 times in order to perform sub operation so by giving push argument 2 times and Constant 1 and sub condition it is nothing but we have to consider last 2 conditions That is 3-1 = 2 (here it is for condition that argument is 3) and stack picture is like Shown below

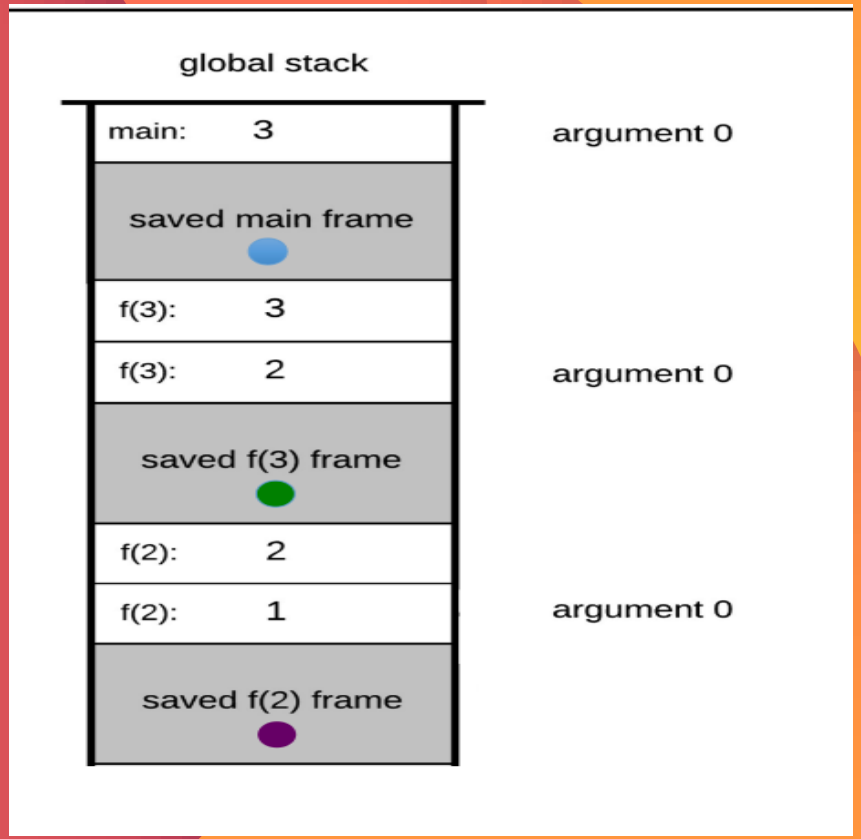


Now again jump to factorial function here in this case argument is 2 and repeat the same What we did in the previous case push argument 0 (2) 2 times and push constant 1 and Perform sub function that is basically 2-1 =1 so the stack picture is as shown below



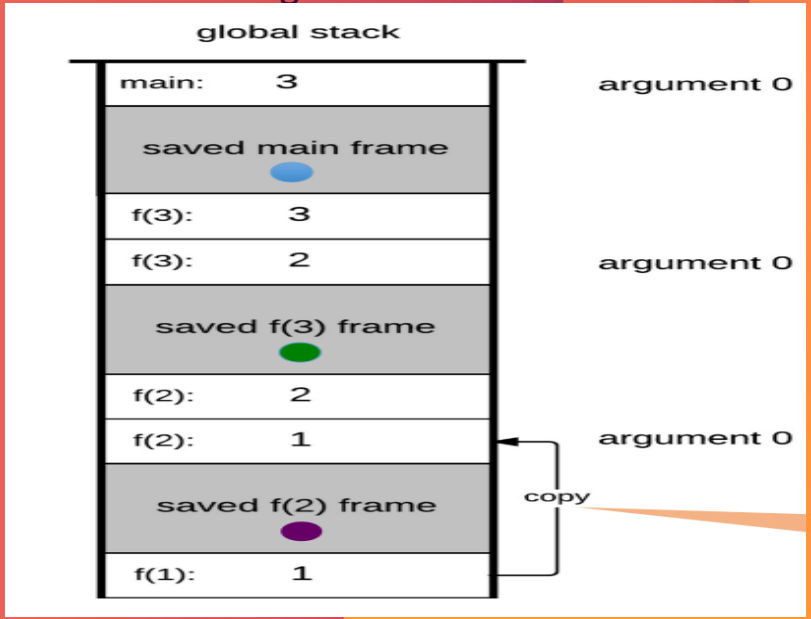
Here we can see the stack picture and after that we are again jumping to factorial Function so when we are doing this here argument will be 0 will be 1 same as what we Did in the previous cases we are pushing argument 0 that is value 1, 2 times and constant 1 into the stack so in this case argument 0 and constant 1 values are Same so as written in code we have to go to BASECASE .

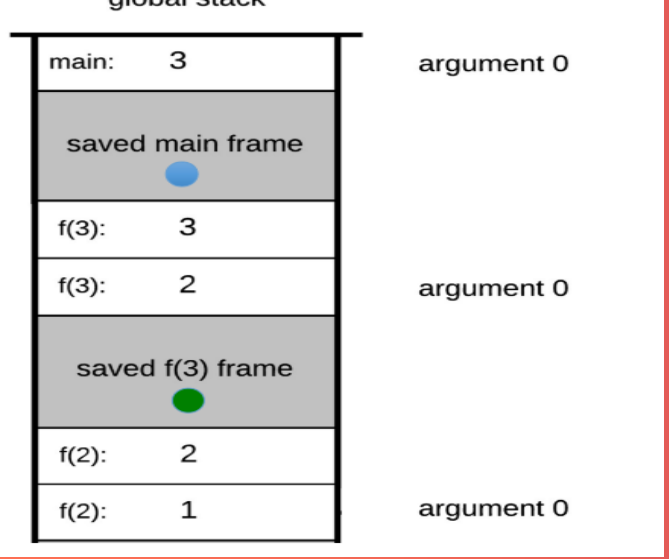

global stack
main: 3
saved main frame
f(3): 3
f(3): 2
saved f(3) frame
f(2): 2
f(2): 1
argument 0
argument 0
argument 0


global stack
main: 3
saved main frame
f(3): 3
f(3): 2
saved f(3) frame
f(2): 2
f(2): 1
saved f(2) frame
argument 0
argument 0
argument 0

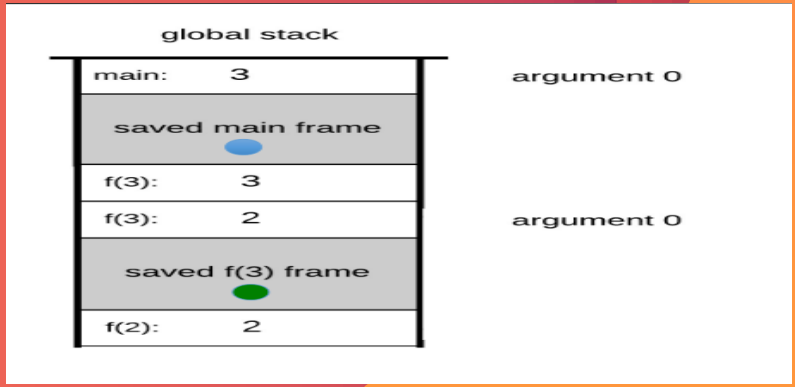
So according to the code we have to push constant 1 into the stack. We have to Proceed according to the Base case and then we have return address so what return Is we have to retrive the return address here it is call mult .

We have to copy this return address to the argument value.

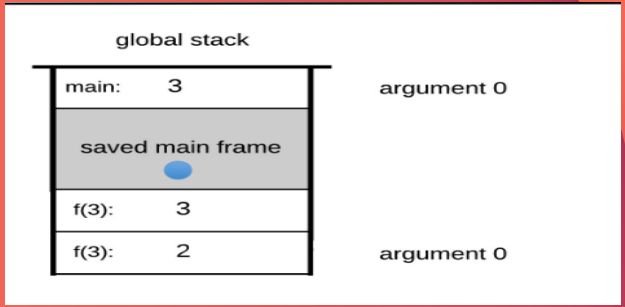




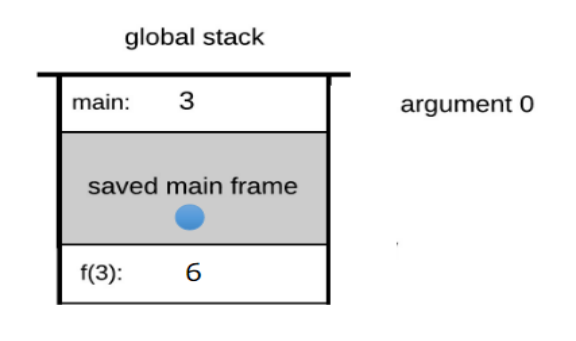
Here we can see that it is copied to the argument next there is call mult function so we Should multiply them and copy that to argument this is what happening here



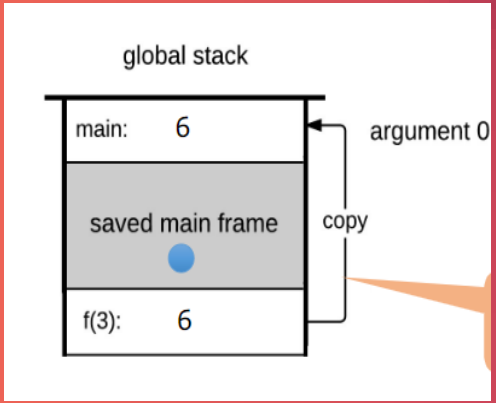
Apply the same concept here also with return statement present in function address
We have to copy that to the argument



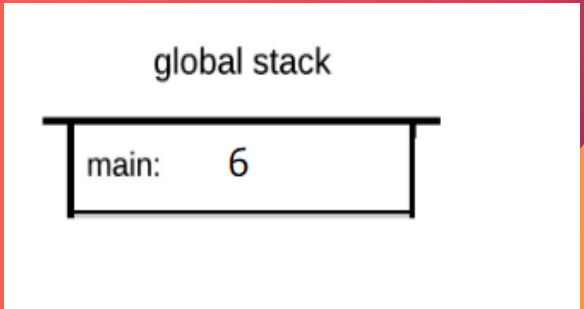
Here is the pic of stack after doing that . After that we have call mult function so multiply those 2 , 3*2=6.



After that there is a return statement in the main function so now we have to copy that to The main function .



So after doing that we get the stack picture as below

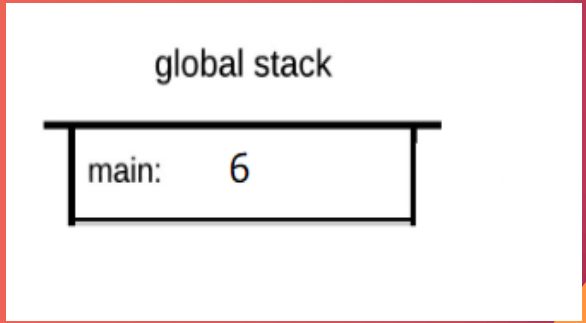


Conclusion

So at last we can see that in the global stack only 6 is remained

That is value of 3 factorial that is 3*2*1=6

That is what we can see in the global stack after executing this code.



THANK YOU