ECEG 431: Project 7

Stewart Thomas

October 7, 2025

Project 7: VM (stage 1)

Overview

Overview

High level language gets translated to code that can run on some "abstracted" or virtual machine. This "VM Code" is a program, that we want to execute.

Many modern systems may perform "JIT" compilation, but we will use the VM code (other systems would call this "byte code") and compile to a machine language program by way of assembly.

A note about Virtual Machines

The virtual machine we are talking about is *NOT* the same as what you are used to! Think of an "imagined" or "idealized" machine or maybe even "abstracted" machine. In higher level languages, we are designing to some thought of machine that does not physically exist, and then the job of the lower-level compilation routines is to make the physical hardware meet the imagined (virtual) interface. Other systems (like `LLVM`) call this an *Intermediate Representation*.

Quick tips *BEFORE* you begin

This chapter is a bit confusing on how to test things. For each of the five test programs supplied to you:

- To understand the program Xxx.vm, rut it on the VM Emulator using the supplied XxxVME.tst script
- Use *your* VM translator to translate the Xxx.vm file to the assembly file Xxx.asm
- Look at your Xxx.asm program to see if there are visible errors. If so, debug and fix your VM translator
- To check if everything runs correctly, use the supplied Xxx.tst and Xxx.cmp files to run your Xxx.asm program on the CPU Emulator.

Second, follow the order of the test programs. They are built to help you incrementally develop the features. We will be extending this VM translator for project 8. And boy howdy, do things get tricky next project!

Order of programs

Program	Description	Test Scripts
SimpleAdd.vm	Pushes two constants onto the stack and adds them up.	SimpleAddVME.tst SimpleAdd.tst SimpleAdd.cmp
StackTest.vm	Executes a sequence of arithmetic and logical operations on the stack.	StackTestVme.tst StackTest.tst StackTest.cmp

Testing how the VM translator handles memory access commands:

Program	Description	Test Scripts
BasicTest.vm	Executes push/pop operations using the virtual memory segments constant, local, argument, this, that, and temp.	BasicTestVME.tst BasicTest.tst BasicTest.cmp
PointerTest.vm	Executes push/pop operations using the virtual memory segments pointer, this, and that.	PointerTestVME.tst PointerTest.tst PointerTest.cmp
StackTest.vm	Executes push/pop operations using the virtual memory segment static.	StaticTestVME.tst StaticTest.tst StaticTest.cmp

Slides

I'm mostly going to present from the textbook slides for this chapter. There are some animations that are quite nice. However, I'll include some information here about what it is in the slides.

PDF File: 01-HackExamples.pdf

- Overview of abstraction between high-level language and the machine
- Modeling of abstraction in a higher-level language vs implementation
  - Done using classes and modeling interface-implementation paradigms
- Very brief view of what compilation looks like for our system

- Short set of slides showing what the 2-tier compilation means
- We compile a high-level language to an abstracted machine
- We can then implement the abstracted machine on different hardware
- For modern systems, this means that
  - we can take advantage of existing lower-level implementations for new languages!
  - a single low-level implementation can bring in multiple high-level languages!

- Introduces a stack data structure
  - A stack "builds" towards the top
  - The SP (stack pointer) points to free space at the top of a stack
- We can "push" new data to the top of the stack
- We can "pull" data from the top of the stack to a variable
- Arithmetic is performed on the top items (usually 2) on the stack

- Starts to get into the central aspects of *doing* the project
- Primary goal is translating VM Code to Assembly code
- Lists the VM Commands we must implement
  - Arithmetic/Logical commands (Project 7)
  - Memory access commands (Project 7)
  - Branching commands
  - Function commands
- Introduces the "Standard VM mapping" on the HACK platform for memory segments

- Short set of slides that demonstrate arithmetic commands
  - Implements $d = (2 - x) + (y + 9)$ on a stack (with VM'ish commands!)
  - Implements $(x < 7)$ or $(y == 8)$ on a stack (with VM'ish commands!)
  - Shows list of arithmetic/logic commands with expected stack return values

Starting to get juicy!

- Shows context of VM Memory segments in a high-level language program
  - Argument, Local, Static in the first example
- Shows how to think of the "abstracted" memory segments in the VM
- The VM abstraction has multiple, separate memory segments (SEPARATE FROM THE STACK!)
  - We can push-to and pop-from these memory segments through the stack

Useful to know!

The difference between *memory segments* and the *stack* can be very confusing. You need to have a mental model where you can understand these as separate ... for now.

PDF File: 07-Pointers.pdf

Here is where we really get into what memory is and how things work! You will understand pointers after this project!

- Introduces (C-style) pointer notation and basic increment/decrement pointer operations
- Then, shows how this actually looks like in HACK assembly
- Next comes in memory segments and how to interact with these (via pointers)
  - Doesn't quite give you the assembly code, but gets you where you can start
- Talks about unique things for implementing the memory segments on the HACK
  - `local, argument, this, that` are implemented the same way
  - `constant` is "faked" and is not an actual segment
  - `static` are for "global" variables and are implemented by assembler variables
  - `temp` are mapped to RAM locations 5 through 12
  - `pointer` store base addresses of `this` and `that`

- Mostly shows how to use the test scripts and begin testing your code and provides a project overview.
- Proposed design structure using a `Parser` class, `CodeWriter` class and the `Main` class/starting-point
  - `Parser` handles input file and breaks apart VM command into elements
  - `CodeWriter` writes assembly commands for each parsed VM element and handles the output file
  - `Main` Mostly responsible for getting input file setup and getting things running
- The general structure for each class is suggested here as well. It might be a bit complex, but is a nice starting point for you. You will probably not need to use all the methods that this structure is suggesting.

Second day of lecture

Notes from readings

- Mostly more confusion about the VM
  - Take a look at LLVM and see a more standard example
  - This VM language is not a known standard, but illustrates a stack-based language
  - Stacks remain important in systems! (Stack Overflow, for example)
- You all really like to do these at the last minute
- General view of what all this is meaning
  - THERE ARE TWO DIFFERENT PERSPECTIVES or ABSTRACTIONS GOING ON!

Plan today

- Brief review of stack-based processing
- Discuss memory segments
- Discuss details of implementing this on our CPU

3rd day of lecture

Things we have covered

Things I think we know well

- The stack
  - Pushing, popping
  - Moving data through the stack to/from memory
- Stack Arithmetic
  - Adding, subtracting, negative'ing

Things we have covered

Things I think we know well

- The stack
  - Pushing, popping
  - Moving data through the stack to/from memory
- Stack Arithmetic
  - Adding, subtracting, negative'ing

Things I think we maaaay need to work on

- Stack Boolean operations
- Memory segments
  - I think we get SP, LCL, ARG, THIS, THAT fairly well –> pointer manipulation
- Static variables
- Initialization
- Assembling a directory

Stack Boolean operations

YOU NEED TO USE *LABELS* FOR THESE

But Prof. Thomas, how do I make a label???

```
@file.vm_TRUE5
D;JLT
D=0
@SP
A=M
M=D
@file.vm_END5
0;JMP
(file.vm_TRUE5)
D=-1
@SP
A=M
M=D
(file.vm_END5)
```

Simple: Generate them!

- I suggest `filename.vm_lbl###` where "###" is a number you just keep incrementing.
- Make a function that keeps track of how many labels you've used, and just spit out a string and wrap it in (`label###`) in your emitted ASM file.

Memory Segments

I think we get SP, LCL, ARG, THIS, THAT fairly well. Constant is not a true memory segment, but we get it.

However, there are some slightly weirder ones. For example: `Static` is shown below

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

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

"Foo" is chosen from the file name, and the argument (5, 2) is chosen based on which static variable. These are "global" variables and accessible in a global space

Memory segments – temp

The `temp` segment is provided so that the compiler can use these variables when compiling programs. These variables are found in RAM locations 5 through 12.

Things like `push temp i` becomes

```
addr = 5 + i, *SP = *addr, SP++
```

We just add the i-th variable to 5.

Memory Segments – pointer

Last is the `pointer` segment. This stores the base location of `this` and `that`. It seems odd, but will be useful later.

- Code like `push pointer 0` becomes `*SP = THIS, SP++`
- Code like `push pointer 1` becomes `*SP = THAT, SP++`

In other words, just treat "pointer 0" as "THIS", and "pointer 1" a "THAT"

Initialization

When we run an actual program, we'd have to take care of initializing the variables and memory segments, setting up the stack pointer, etc. For this project, we are just ignoring all of that.

We can do this because many of the test files include the necessary setup items to put these in place (setting SP, setting LCL and ARG segments, etc.)

Assembling a directory

You need to be able to operate on an entire directory. For each VM file, you will translate each VM file. Eventually, we will just emit one giant ASM file with the entire code.

Handle directories, and put the ASM file in the same location as the VM file