# The Virtual Machine: Part 1

Prof. Naga Kandasamy
ECE Department, Drexel University

Develop a VM translator that implements stack arithmetic and memory-access commands. This problem is due December 4, 2023, by 11:59 pm. Please submit original work.

The name of the VM-language file is supplied to the translator as a command-line argument. If you choose to implement the translator in Python, usage is as follows:

```
> Python vm_translator.py some-program.vm
```

The *some-program.vm* file consists of text lines, each representing a VM command. Your translator must translate these VM commands into corresponding Hack assembly-language commands and store them in an output text file called *some-program.asm*. You may use the code provided in *vm_translator.py* as a starting point for your solution. Edit the file as desired.

## Memory-Access Commands

Your translator must support *push* and *pop* stack operations from/to the eight memory segments discussed in lecture:

- *push segment index* pushes the value of *segment*[*index*] to the stack.
- *pop segment index* pops the top stack value and stores it in *segment*[*index*].

The functionality of various memory segments are summarized as follows:

- The *argument* segment stores function arguments, with the base address stored in the `ARG` register.
- The *local* segment stores the function's local variables, with base address stored in the `LCL` register.
- The *static* segment stores static variables shared by all functions located in the same `.vm` file. The segment starts at `RAM[16]`.
- The *constant* pseudo-segment holds constants in the range 0 through $2^{16} - 1$.
- Segments *this* and *that* correspond to different areas in the heap, and the base addresses are stored in `THIS` and `THAT` registers, respectively.
- The two-entry *pointer* segment holds the base addresses of *this* and *that* segments. The $0^{\text{th}}$ entry corresponds to *this* and the $1^{\text{st}}$ entry to *that* segment, respectively.
- The *temp* segment holds eight temporary variables for general-purpose use, beginning at `RAM[5]`.

The following table lists the use of registers in the context of VM programs.

| Register | RAM location | Usage |
|---|---|---|
| `SP` | `RAM[0]` | Stack pointer; points to topmost location in the stack. |
| `LCL` | `RAM[1]` | Points to base of the current VM function's *local* segment. |
| `ARG` | `RAM[2]` | Points to base of the current VM function's *argument* segment. |
| `THIS` | `RAM[3]` | Points to base of the current VM function's *this* segment. |
| `THAT` | `RAM[4]` | Points to base of the current VM function's *that* segment. |
| | `RAM[5]`–`RAM[12]` | Holds contents of the *temp* segment. |
| | `RAM[13]`–`RAM[15]` | General-purpose registers for use by the VM implementation. |

- The *local*, *argument*, *this*, and *that* segments are mapped to RAM, and the corresponding locations are maintained by keeping the base addresses in registers `LCL`, `ARG`, `THIS`, and `THAT`. Any access to the $i^{\text{th}}$ entry of any one of these segments should be translated to assembly code that accesses `RAM[(base + i)]`, where $base$ is the current value stored in the register dedicated to that particular segment.
- The *pointer* segment is mapped to `RAM[3]` and `RAM[4]` — also called `THIS` and `THAT` — and the *temp* segment to RAM locations 5–12. Therefore, 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 $5 + i$.
- The *constant* segment is virtual; it occupies no physical space on the hardware. The VM implementation should handle any access to this segment by simply supplying the requested constant.

## Arithmetic and Logic Commands

Your VM translator must support the following arithmetic and logic commands.

| **Command** | **Return value** (after popping operands) | **Comment** |
|---|---|---|
| *add* | $x + y$ | Integer addition (in two's complement) |
| *sub* | $x - y$ | Integer subtraction (in two's complement) |
| *neg* | $-x$ | Arithmetic negation (in two's complement) |
| *eq* | if $x = y$ then true else false | Test for equality |
| *gt* | if $x > y$ then true else false | Test for greater than |
| *lt* | if $x < y$ then true else false | Test for less than |
| *and* | $x \& y$ | Bit-wise AND |
| *or* | $x \mid y$ | Bit-wise OR |
| *not* | $\neg x$ | Bit-wise NOT |

Seven commands are binary in that they pop two items off the stack, compute a binary function on them, and push the result back onto the stack; and the remaining two commands are unary in that they pop a single item off the stack, compute a unary function on it, and push the result back onto the stack. The *eq*, *gt*, and *lt* commands return Boolean values — true is represented as `0xFFFF` (or -1 in two's complement) and false is represented as `0x0000` (or 0).

## Testing Your VM Translator

Test the correctness of the assembly code generated by your translator using the following test programs. The expected output is included in the source code as part of the comments.

1. **(5 points)** *assignment.vm* contains VM commands that perform assignment operations.
2. **(5 points)** *arithmetic_test.vm* performs various arithmetic operations.
3. **(5 points)** *arithmetic_logic_test.vm* performs various arithmetic and logic operations.
4. **(5 points)** *memory_test.vm* performs memory access operations.
5. **(5 points)** *pointer_test.vm* performs pointer operations on the *this* and *that* segments.
6. **(5 points)** *static_test.vm* performs stack operations using the *static* segment.

In each of the above cases, use the Assembler tool to convert the generated `.asm` file into a `.hack` file containing the corresponding machine code. Then simulate the Hack machine code using the CPUEmulator tool and verify that the correct result is generated by the processor.

## Submission Instructions

Submit via BBLearn, the source code for the VM translator along with a README file that details how to build and execute it.