FLOAT Stack-based VM Description

Gary T. Leavens
Department of Computer Science
University of Central Florida
Leavens@ucf.edu

March 31, 2023

Abstract

In creating the code generator (for the FLOAT calculator), we found that the stack machine as described previously (for homework 4) needed some revision. In this document, the changed and revised features (from the homework 4 VM) are **highlighted in bold font**. The major changes have to do with adapting the VM to deal with floating point numbers.

1 Overview

The virtual machine (VM) is a word-addressible stack-based machine that manipulates both floating-point numbers (C's **`float`** type) and integers (C's **`int`** type).

The following subsections specify the interface between the Unix operating system (as on Eustis) and the VM as a program.

1.1 Inputs

The VM understands the `-n` command line option, which turns off the printing of the program and the VM's tracing output; otherwise the VM prints the program and an execution trace, on `stderr`, by default. The execution trace can be turned off during a program's execution by using the NDB instruction.

The VM takes a single file name as a command line argument; this file should be the name of a (readable) text file containing the program that the VM should execute. For example, if the executable is named `vm/vm` and the program it should run is contained in the file named `test1.vmi` (and both the input file and the `vm` subdirectory are in the current directory), then the VM should execute the program in the file `test1.vmi` by executing the following command in the Unix shell (e.g., at the command prompt on Eustis):

```
vm/vm test1.vmi
```

When the program executes a CHI instruction to read a character, that character will be read from standard input (`stdin`). However, note that if you want the program to read a character, typing a single character (say `c`) into the terminal (i.e., to the shell) while the program is running will not send that character immediately to the program, as `stdin` is buffered. To send characters to the program it is best to use a pipe or file redirection in the Unix shell, for example, to send the two characters `c` and `e` to the VM running the program `progfile.vmi` one could use the following command at the Unix shell:

```
echo ce | vm/vm progfile.vmi
```

One could also put those characters in a file say `ce-input.txt` and then to use the following Unix command: `vm/vm progfile.vmi < ce-input.txt`

1.2 Outputs

The VM prints its tracing output to the Unix standard error output (`stderr`). However, characters printed using the CHO instruction are printed on standard output (`stdout`).

All error messages (e.g., for division by zero) are sent to standard error output (`stderr`).

1.3 Exit Code

When the machine halts normally, it exits with a success error code (zero on Unix). However, when the machine encounters an error it halts and the program should exits with a failure error code (non-zero on Unix).

2 VM Architecture

The VM is a stack machine that conceptually has two memory stores: the "stack," which is organized as a LIFO queue of words and contains the data to be used by instruction evaluation, and the "code," which is organized as a list of instructions. The code list contains the instructions for the VM in order of execution; however, that order can be changed by executing instructions (such as jump or call instructions).

2.1 Registers

The VM has a few built-in registers[1] used for its execution: The registers are named:

- base pointer (BP),
- stack pointer (SP), which points to the next location in the stack to allocate (i.e., one above the current top of the stack), and
- program counter (PC).

The use of these registers will be explained in detail below.

2.2 Instruction Format

The Instruction Set Architecture (ISA) of the VM has instructions that each have two components, which are integers (i.e., they have the C type int) named as follows:

OP	is the operation code (an unsigned integer)
M	a word, whose type depends on the operator it indicates, either: (a) A C **`float`** (when the instruction is a LIT instruction, so that OP is 1) or (b) A C **`int`** (for all other instructions)

The list of instructions and details on their execution appears in Appendix A. In some instructions the function toInt(w) is used, which returns the value of round(f), when w represents the float f, and otherwise just the integer value of w. Similarly, toFloat(w) returns the value of w as a float.

[1] What we call "registers" in this document are simply important concepts that simulate what would be registers in a hardware implementation of the virtual machine. In the VM as a C program, these are implemented as variables.

2.3 VM Cycles

The VM instruction cycle conceptually does the following for each instruction:

1. Let *IR* be the instruction at the location that PC indicates. (Note that *IR* could be considered to be the contents of a register.)
2. The PC is made to point to the next instruction in the code list.
3. The instruction *IR* is executed using the "stack" memory. (This does not mean that the instruction is stored in the "stack.") The OP component of this instruction (*IR*.OP) indicates the operation to be executed. For example, if *IR*.OP encodes the instruction ADD, then the machine adds the top two elements of the stack, popping them off the stack in the process, and stores the result in the top of the stack (so in the end SP is one less than it was at the start). Note that the arithmetic instructions operate on numbers as in C's **float** arithmetic.[2]

2.4 VM Initial/Default Values

When the VM starts execution, BP, SP, and PC are all 0. This means that execution starts with the "code" element 0. Similarly, the initial "stack" store values are all zero (0).

2.5 Size Limits

The following constants define the size limitations of the VM.

- `MAX_STACK_HEIGHT` is 2048
- `MAX_CODE_LENGTH` is 512

2.6 Invariant Properties

The VM enforces the following invariant properties and will halt with an error message (written to stderr) if one of them is violated:

- $0 \leq \mathsf{BP} \ \wedge \ \mathsf{BP} \leq \mathsf{SP} \ \wedge \ 0 \leq \mathsf{SP} \ \wedge \ \mathsf{SP} < \texttt{MAX\_STACK\_HEIGHT}$
- $0 \leq \mathsf{PC} \ \wedge \ \mathsf{PC} < \texttt{MAX\_CODE\_LENGTH}$

Although the VM tracks the format of data written on the stack (as a **`float`** or **`int`**), it tries to convert these types to whatever is needed for the instruction as needed, and it also allows writing a location in the stack without regard to the format of the data that was there previously.

A Appendix A

In the following tables, the meta-variable f refers to a C float used as data, while the meta-variables p refers to an (unsigned) integer used as an address and o and m are meta-variables that stand for an (unsigned) integer used as an offset or count. If an instruction's M field is notated as $-$, then its value does not matter (we use 0 as a placeholder for such values in examples). Note that stack[$SP - 1$] is the top element of the stack.

[2]The VM's arithmetic was changed to be a combination of C's **`int`** and **`float`** arithmetic in this revision.

A.1 Basic Instructions

OP Code	OP Mnemonic	M	Comment (Explanation)
0	NOP	–	do nothing (no-op)
1	LIT	f	Literal push: stack[SP] $\leftarrow f$; SP $\leftarrow$ SP $+ 1$
2	RTN	–	Returns from a procedure and restores the caller's AR: PC $\leftarrow$ stack[SP $- 1$]; BP $\leftarrow$ stack[SP $- 2$]; SP $\leftarrow$ SP $- 3$
3	CAL	p	Call the procedure at code index p, generating a new activation record and setting PC to p: stack[SP] $\leftarrow$ stack[BP]; // static link stack[SP $+ 1$] $\leftarrow$ BP; // dynamic link stack[SP $+ 2$] $\leftarrow$ PC; // return address BP $\leftarrow$ SP; SP $\leftarrow$ SP $+ 3$; PC $\leftarrow p$;
4	POP	–	Pop the stack: SP $\leftarrow$ SP $- 1$;
5	PSI	–	Push the element at address stack[SP $- 1$] on top of the stack: stack[SP $- 1$] $\leftarrow$ stack[toInt(stack[SP $- 1$])]
6	LOD	o	The value at the address in the top of the stack + offset o is put on top of the stack: stack[SP $- 1$] $\leftarrow$ stack[toInt(stack[SP $- 1$]) $+ o$]
7	STO	o	Store stack[SP $- 1$] into the stack at address toInt(stack[SP $- 2$] $+ o$) and pop the stack twice: stack[toInt(stack[SP $- 2$]) $+ o$] $\leftarrow$ stack[SP $- 1$]; SP $\leftarrow$ SP $- 2$
8	INC	m	Allocate m locals on the stack: SP $\leftarrow$ SP $+ m$
9	JMP	o	Jump relative to the current instruction's code index: PC $\leftarrow$ PC $- 1 + o$
10	JPC	o	Jump conditionally relative to the current instruction's code index: **if** stack[SP $- 1$] $\neq 0$ **then** {PC $\leftarrow$ PC $- 1 + o$} ; SP $\leftarrow$ SP $- 1$
11	CHO	–	Output the value in stack[SP $- 1$] (rounded if need be) to standard output as a character and pop: **putc**(toInt(stack[SP $- 1$]), `stdout`); SP $\leftarrow$ SP $- 1$
12	CHI	–	Read an integer, as character value, from standard input and push it in the top of the stack, but on EOF or error, push -1: stack[SP] $\leftarrow$ **getc**(`stdin`); SP $\leftarrow$ SP $+ 1$
13	HLT	–	Halt the program's execution
14	NDB	–	Stop printing debugging output

A.2 Arithmetic/Logical Instructions

For comparisons, note that the integer 0 represents false and 1 represents true. That is, the result of a logical operation, such as $A > B$ is defined as 1 if the condition was met and 0 otherwise. **Arithmetic operations are performed as C's `float` arithmetic.**[3] Errors such as division by 0 cause the VM to halt with an appropriate error message printed on stderr.

OP Code	Number Mnemonic	M	Comment (Explanation)
15	NEG	–	Negate the value in the top of the stack: $\text{stack}[\mathsf{SP}-1] \leftarrow -\text{toFloat}(\text{stack}[\mathsf{SP}-1])$
16	ADD	–	Add the top two elements in the stack: $\text{stack}[\mathsf{SP}-2] \leftarrow \text{toFloat}(\text{stack}[\mathsf{SP}-2]) + \text{toFloat}(\text{stack}[\mathsf{SP}-1]);$ $\mathsf{SP} \leftarrow \mathsf{SP}-1$
17	SUB	–	Subtract the top element from the 2nd to top one: $\text{stack}[\mathsf{SP}-2] \leftarrow \text{toFloat}(\text{stack}[\mathsf{SP}-2]) - \text{toFloat}(\text{stack}[\mathsf{SP}-1]);$ $\mathsf{SP} \leftarrow \mathsf{SP}-1$
18	MUL	–	Multiply the top two elements in the stack: $\text{stack}[\mathsf{SP}-2] \leftarrow \text{toFloat}(\text{stack}[\mathsf{SP}-2]) \times \text{toFloat}(\text{stack}[\mathsf{SP}-1]);$ $\mathsf{SP} \leftarrow \mathsf{SP}-1$
19	DIV	–	Divide the 2nd from top element by the top one: $\text{stack}[\mathsf{SP}-2] \leftarrow \text{toFloat}(\text{stack}[\mathsf{SP}-2])/\text{toFloat}(\text{stack}[\mathsf{SP}-1]);\ \mathsf{SP} \leftarrow \mathsf{SP}-1$
20	RND	–	**Round the result on top of the stack into an integer:** $\text{stack}[\mathsf{SP}-1] \leftarrow \text{round}(\text{stack}[\mathsf{SP}-1]);$
21	EQL	–	Are (the contents of) the top two elements equal? $\text{stack}[\mathsf{SP}-2] \leftarrow \text{toFloat}(\text{stack}[\mathsf{SP}-2]) = \text{toFloat}(\text{stack}[\mathsf{SP}-1]);$ $\mathsf{SP} \leftarrow \mathsf{SP}-1$
22	NEQ	–	Are (the contents of) the top two elements different? $\text{stack}[\mathsf{SP}-2] \leftarrow \text{toFloat}(\text{stack}[\mathsf{SP}-2]) \neq \text{toFloat}(\text{stack}[\mathsf{SP}-1]);$ $\mathsf{SP} \leftarrow \mathsf{SP}-1$
23	LSS	–	Is (the contents of) the second from the top element strictly less than the contents of the top element? $\text{stack}[\mathsf{SP}-2] \leftarrow \text{toFloat}(\text{stack}[\mathsf{SP}-2]) < \text{toFloat}(\text{stack}[\mathsf{SP}-1]);$ $\mathsf{SP} \leftarrow \mathsf{SP}-1$
24	LEQ	–	Is (the contents of) the 2nd from top element no greater than the contents of the top element? $\text{stack}[\mathsf{SP}-2] \leftarrow \text{toFloat}(\text{stack}[\mathsf{SP}-2]) \leq \text{toFloat}(\text{stack}[\mathsf{SP}-1]);$ $\mathsf{SP} \leftarrow \mathsf{SP}-1$
25	GTR	–	Is (the contents of) the 2nd from top element strictly greater than the contents of the top element? $\text{stack}[\mathsf{SP}-2] \leftarrow \text{toFloat}(\text{stack}[\mathsf{SP}-2]) > \text{toFloat}(\text{stack}[\mathsf{SP}-1]);$ $\mathsf{SP} \leftarrow \mathsf{SP}-1$
26	GEQ	–	Is (the contents of) the 2nd from top element no less than the contents of the top element? $\text{stack}[\mathsf{SP}-2] \leftarrow \text{toFloat}(\text{stack}[\mathsf{SP}-2]) \geq \text{toFloat}(\text{stack}[\mathsf{SP}-1]);$ $\mathsf{SP} \leftarrow \mathsf{SP}-1$

[3] **The definition of arithmetic for the arithmetic operators was changed to `float` arithmetic in this revision.**

A.3 VM State Examination Instructions

These instructions allow the state of the VM to be examined and used in computation.

The RBP instruction was added to restore the break pointer from the top of the stack.

OP Code	Number Mnemonic	M	Comment (Explanation)
27	PSP	–	Push SP (i.e., the address itself) on top of the stack: stack[SP] ← SP; SP ← SP + 1
28	PBP	–	Push BP (i.e., the address itself) on top of the stack: stack[SP] ← BP; SP ← SP + 1
29	PPC	–	Push PC (i.e., the address itself) on top of the stack: stack[SP] ← PC; SP ← SP + 1
30	JMI	–	jump to the address on top of the stack: stack[PC] ← toInt(stack[SP − 1]); SP ← SP − 1
31	**RBP**	–	**restore the break pointer: BP ← toInt(stack[SP − 1]); SP ← SP − 1**

A.4 Examples

As an example, consider the instruction `ADD 0`, which is input as the line `16 0`; then assuming SP is 10, this means to place in stack[8] the sum of the values in stack[8] and stack[9], and then setting SP to 9.

As another example: if we have instruction `LIT 9.0`, which is input as the line `1 9.0`, then this means to push the float 9.0 on the top of the stack: stack[SP] ← 9.0; SP ← SP + 1.

B Appendix B: Examples

B.1 A Simple Example Showing Output Formatting

The following very simple example shows the expected formatting. Suppose the input is the following file (`hw1-test0.vmi`, the name of this file is passed to the VM on the Unix command line):

Running the VM with the command line argument `hw1-test0.vmi` the following output (written to stderr). Note that there are two parts to the output: (1) a listing of the instructions in the program one per line, following a header, with mnemonics for each instruction and (2) a trace of the program's execution, following the line `Tracing ...` (all on standard error output). The trace of execution shows the state of the built-in registers (PC, BP, and SP) and the stack's values at addresses between BP and SP − 1 (inclusive), and then it shows the instruction being executed (following the text `==> addr:`); this consists of: (a) the address of the instruction being executed, then (b) the instruction with its mnemonic and M value, then after showing the instruction being executed (and after the instruction's execution by the VM) the state is again shown. The output of the instruction and the resulting state are show after is each instruction executed.

In this output, words that are floating-point numbers are printed to look like floating point numbers, and words that are integers are printed without a decimal point.

```
Addr  OP    M
0     INC   3
1     HLT   0
Tracing ...
PC: 0 BP: 0 SP: 0
stack:
==> addr: 0     INC   3
```

```
PC: 1 BP: 0 SP: 3
stack: [0]: 0.000000 [1]: 0.000000 [2]: 0.000000
==> addr: 1     HLT   0
PC: 2 BP: 0 SP: 3
stack: [0]: 0.000000 [1]: 0.000000 [2]: 0.000000
```

C Future Work

To better support blocks (sequences of statements with declarations), there should be an instruction for entering a block, which would push BP on the stack and set BP to point to that address (SP − 1). The RBP instruction already supports exit from a block.

The VM already understands the distinction between floating point numbers and integers, but it could also be made to distinguish Boolean values and print them using "true" and "false".

C.1 A Slightly More Involved Example

The following example is a bit more involved and shows some of the details of the machine's execution.

C.1.1 Input File

The following is the contents of the file `test1.vmi`:

```
8 2
1 0
1 1
1 5
1 7
16 0
1 12
22 0
10 2
13 0
1 78
11 0
1 13
11 0
13 0
```

C.1.2 Output (to stderr)

Running the VM with the above input produces the following output on `stderr` (assuming that the `-n` option is not used).

```
Addr  OP    M
0     INC   2
1     LIT   0.000000
2     LIT   1.000000
3     LIT   5.000000
4     LIT   7.000000
5     ADD   0
6     LIT   12.000000
7     NEQ   0
```

```
8     JPC   2
9     HLT   0
10    LIT   78.000000
11    CHO   0
12    LIT   13.000000
13    CHO   0
14    HLT   0
Tracing ...
PC: 0 BP: 0 SP: 0
stack:
==> addr: 0     INC   2
PC: 1 BP: 0 SP: 2
stack: [0]: 0.000000 [1]: 0.000000
==> addr: 1     LIT   0.000000
PC: 2 BP: 0 SP: 3
stack: [0]: 0.000000 [1]: 0.000000 [2]: 0.000000
==> addr: 2     LIT   1.000000
PC: 3 BP: 0 SP: 4
stack: [0]: 0.000000 [1]: 0.000000 [2]: 0.000000 [3]: 1.000000
==> addr: 3     LIT   5.000000
PC: 4 BP: 0 SP: 5
stack: [0]: 0.000000 [1]: 0.000000 [2]: 0.000000 [3]: 1.000000 [4]: 5.000000
==> addr: 4     LIT   7.000000
PC: 5 BP: 0 SP: 6
stack: [0]: 0.000000 [1]: 0.000000 [2]: 0.000000 [3]: 1.000000 [4]: 5.000000 [5]: 7.000000
==> addr: 5     ADD   0
PC: 6 BP: 0 SP: 5
stack: [0]: 0.000000 [1]: 0.000000 [2]: 0.000000 [3]: 1.000000 [4]: 12.000000
==> addr: 6     LIT   12.000000
PC: 7 BP: 0 SP: 6
stack: [0]: 0.000000 [1]: 0.000000 [2]: 0.000000 [3]: 1.000000 [4]: 12.000000 [5]: 12.00000
==> addr: 7     NEQ   0
PC: 8 BP: 0 SP: 5
stack: [0]: 0.000000 [1]: 0.000000 [2]: 0.000000 [3]: 1.000000 [4]: 0
==> addr: 8     JPC   2
PC: 9 BP: 0 SP: 4
stack: [0]: 0.000000 [1]: 0.000000 [2]: 0.000000 [3]: 1.000000
==> addr: 9     HLT   0
PC: 10 BP: 0 SP: 4
stack: [0]: 0.000000 [1]: 0.000000 [2]: 0.000000 [3]: 1.000000
```