**Simple Virtual Machine**

This project takes us from the world of high-level object-oriented language programming to the low-level operations of a computer. We are going to emulate how a machine operates. And to do that, we are going to study a simple machine language and write several programs on it. That is a valuable experience that allows us to understand how *a virtual machine actually works*.

Virtual Machines solve a wide range of tasks from emulating the old hardware to providing a portable platform for computation. Examples of virtual machine include JVM, Dalvik, .NET CLR, Squeak (Smalltalk), CPython, DosBox, VMWare etc...

Let us create an emulator called Simple Virtual Machine (or SVM for short). As its name implies, it’s a simple machine, but also a powerful one as well. It has a CPU, memory and an external debugger with a lot of features. We can program this machine by loading bytecode images from the disk or by entering opcode mnemonics in the debugger.

Following sections give SVM specifications and explain how it works. By implementing a SVM emulator, you will actually be able to run, test and debug the SVM programs.

**Random Access Memory**

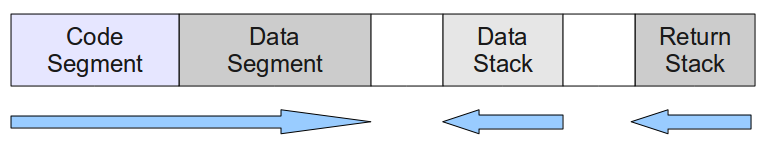
Random Access Memory (RAM) is the memory where program code and data are stored. The memory is addressed by a single positive integer value and stores signed integer values. That means you could address more than 8GB of virtual RAM. In reality, you have a small fraction of that actually allocated and any addressing out of that range MUST raise an exception.

You can allocate to the emulator as much memory as needed for a particular computation during the start-up. As an example, you could allocate 1k words of RAM to the emulator by passing options from the command line. In that case you could access the memory using addresses in the range 0..1023.

Any addressable value stored in memory is a signed 32-bit integer. Since 32-bit integers are units of work for our CPU, they are called ***words***.

A word could be interpreted in a number of way, depending on the context. It could be an operation code for the CPU, a single character in the string, an integer variable, a return address in the Return Stack or a local variable in the Data Stack.

Typical SVM memory map is displayed on the following figure.



CPU starts execution from an opcode located at address 0x00000000, so usually lowest regions of memory is occupied by the bytecode. The end of the Code Segment usually marked by the OP register of the CPU - this register is used by the debugger to place the newly entered opcode. Code Segment is followed by the Data Segment which contains all strings and global variables. End of the Data Segment is marked by the DP register and each time we read a new string from the console, the CPU allocates it at the end of Data Segment and adjust the DP register accordingly. Therefore Data Segment grows up.

SVM has a two-stack organization for simplicity. So we don’t have to store local values and return addresses in one place.

Data Stack contains all values for operation stack and grows down each time we push a new values. SVM is a stack-based machine, so all computation is based on the stack. Each time SVM executes ADD instruction, it takes two values from the Data Stack, adds them, and pushes the result back.

Return stack is used by the CALL opcode to store the return address before a subroutine call and by the RET opcode to go back from a subroutine. The Return Stack grows down the same way, as the Data Stack.

It is essential, that mentioned segments of memory do not overlap. Otherwise one area could damage another and cause an error in the program execution. It is a responsibility of the initial memory allocation and the SVM program itself to keep memory segments separated.

**CPU**

The CPU can be considered as the brain of our virtual machine. It processes and executes all the instructions in the RAM.

An instruction is represented in memory by a single word containing the opcode constant. An instruction also got a symbolic mnemonic which is used by the debugger to assemble and disassemble SVM programs. For example, “ADD” is a mnemonic for an opcode 0xC0 which is the CPU instruction to add two numbers on the Data Stack.

**Registers**

The SVM CPU has 5 registers: CP, OP, DP, RP, SP.

**CP** - points on the next opcode to execute. After the start-up it points at 0x00, so the execution begins from there. Each time an opcode gets executed it is incremented by 1 (exception is the opcode PUSH which has a parameter and increments CP by 2). The JMP, JMZ and JMN opcodes could affect the CP register directly. They set new value to CP and the execution continues from an opcode at assigned address.

**OP** - a special register used by the debugger as the address to place the next opcode. Basically, indicates the end of a Code Segment.

**DP** - used by the READ as the address to place the input string into the memory. Points at the end of the Data Segment.

**RP** - points at the top of the Return Stack. Each time the CPU executes the CALL opcode, the RP register is decremented by 1 and return address (current value of CP) is placed at the address it points. The RET opcode makes everything backwards - loads the value currently pointed by RP into CP and increments RP by 1.

**SP** - points at the top of the Data Stack. Most of the opcodes work only with the values on the Data Stack. Therefore, Data Stack increases and decreases all the time.

**Instruction set**

Most of the instructions operate on the data stack. They pop parameters from the top and push the result back.

All instructions occupy a single word in memory with their opcode. The only exception is the PUSH instruction (0xA0) which is always followed by the value to push on the Data Stack.

You can find out more about instructions, their opcodes and mnemonics in the Instruction Set reference.

**Input and Output**

SVM could work with a console. It could read strings and integer values from the console and print string and integer values on the console. There are 4 opcodes to do that: READ, WRITE, READI, WRITEI.

Strings are represented by the internal format:

*[Length][Character][Character][..][..]*

Length is a single word, each character of the string also occupies exactly one word.

Each READ opcode reads the string from the console and allocates it at the address pointed by the DP register. The DP register gets increased by [LENGTH + 1] in this case.

WRITE opcode pops the address of the string and prints it on the console.

READI reads the string from the console, parses first token as integer and pushes that value on top of the Data Stack.

WRITEI pops the word from the Data Stack and prints it on the console.

**Execution Cycle**

The CPU executes opcodes one-by-one with exception of subroutine calls/returns(*CALL* and *RET*) and conditional/unconditional jumps (*JMP, JMZ, JMN*).

JMP, JMZ and JMN instruction make relative jumps. They are not jump at particular address, rather make a shift to the current address in CP. Therefore, it is a relational jump instructions which take relative address and not the absolute one. The CALL instruction, on the other hand, expects absolute address on the top of the Data Stack.

Each step of the CPU execution is called execution cycle. CPU takes an opcode at the address pointed by CP, switches between possible opcodes to choose the right one and execute associated actions. In case there is no such opcode, SVM must raise an exception with details of the error (address and opcode which caused that problem) and switch SVM into the debug mode.

Lets consider a program to show the “OK” message: *PUSH 0x10 WRITE DEBUG*. It assumes that OK message is placed at the 0x10 address.

Following dump shows the memory for “OK” message program.

CP: 00000000

OP: 00000004 DP: 00000013

RP: 00000020 SP: 0000001C

IE: 0000000000000002

00000000: 000000A0 00000010 000000A4 EEEEEEEE 00000000 00000000 00000000 00000000

00000008: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000010: 00000002 0000004F 0000004B 00000000 00000000 00000000 00000000 00000000

00000018: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

At 0x10 address you can see the “OK” string itself - 0x02 is the length of the string and 0x4f and 0x4B are the two characters. First instruction is 0xA0 (PUSH) with argument 0x10. After that step we will have the following dump:

CP: 00000002

OP: 00000004 DP: 00000013

RP: 00000020 SP: 0000001B

IE: 0000000000000003

00000000: 000000A0 00000010 000000A4 EEEEEEEE 00000000 00000000 00000000 00000000

00000008: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000010: 00000002 0000004F 0000004B 00000000 00000000 00000000 00000000 00000000

00000018: 00000000 00000000 00000000 00000010 00000000 00000000 00000000 00000000

IE is the CPU instruction counter. It has been increased by 1. Also, the CP register get increased by 2 (since PUSH instruction has an argument). Now we can see 0x10 value appeard on the data stack and the data stack register SP decreased from 1C to 1B. The CP register now points at the A4 opcode (WRITE). Next step will show “OK” on the console and we can get the following dump:

CP: 00000003

OP: 00000004 DP: 00000013

RP: 00000020 SP: 0000001C

IE: 0000000000000004

00000000: 000000A0 00000010 000000A4 EEEEEEEE 00000000 00000000 00000000 00000000

00000008: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000010: 00000002 0000004F 0000004B 00000000 00000000 00000000 00000000 00000000

00000018: 00000000 00000000 00000000 00000010 00000000 00000000 00000000 00000000

Note, that the data stack register SP got increased to 0x1C. We no longer have hte 0x10 value on the stack, however it still stays in the memory untouched. Next push to the data stack will eventually replace that value with a new one. The CP register now points at the 0xEEEEEEEE opcode (DEBUG) which instructs SVM to switch to the debug mode.

**Debugger**

Debugger allows us to look inside the SVM, modify memory, registers etc.

The SVM goes into the debug mode after executing the *DEBUG* opcode (0xEEEEEEEE) or after the start-up if *-debug* option was specified in the command line. In the debug mode you could change the registers, dump the memory, modify the memory, load or store memory images. Debug allows you to make step-by-step execution of the program (when you return back into the debug mode after next opcode is executed). Also you could restore the SVM execution cycle by issuing the ‘start’ command. The execution will continue from the address currently pointed by the *CP* register.

Typing the ‘help’ command must show following options:

Possible commands are:

start - restart execution cycle from current CP position

restart - restart execution cycle from the 0x00 position

step - execute single instruction execution cycle

dump - CPU and RAM dump (optional parameters - address and length)

load [filename] - loads ROM image from file

store [filename] - store RAM on the file

read .. - read values/mnemonics from console to memory starting from OP

code [filename] - read and translate values from file starting from OP

uncode [address, length] - dissasemble memory region

erase [shift][length] - fill area of memory with 0x00

cp [xx] - assigns new value to CP register

op [xx] - assigns new value to OP register

dp [xx] - assigns new value to DP register

rp [xx] - assigns new value to RP register

sp [xx] - assigns new value to SP register

help - display this message

exit - stop SVM and exit

Lets debug a simple program: READI READI ADD WRITEI HALT. It read two values from the user, adds them and prints the result on the console.

\*\* Welcome to Simple Virtual Machine \*\*

\*\* SVM is in the Debug Mode \*\*

>> dump

CP: EEEEEEEE

OP: 00000000 DP: 00000010

RP: 00000020 SP: 0000001C

IE: 0000000000000001

00000000: EEEEEEEE 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000008: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000010: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000018: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

>> read readi readi add writei halt

>> dump

CP: EEEEEEEE

OP: 00000005 DP: 00000010

RP: 00000020 SP: 0000001C

IE: 0000000000000001

00000000: 000000A5 000000A5 000000C0 000000A6 FFFFFFFF 00000000 00000000 00000000

00000008: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000010: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000018: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

>> restart

4

5

9

First we use debugger *dump* command to show current status of the machine, then *read* command to read values and mnemonics and place that into the memory. Then dump to see the result of assembly. And finally, we use *restart* command to run the SVM from 0x00. We entered two values and get their sum as the result.

**Video Memory**

As advanced feature, you could implement video memory and a simple display system emulation.

Video Memory is mapped on the regular RAM address space at 0x0A000000. Values starting from that address represents colors on the display system. Each word is a pixel 0xAARRGGBB, where AA is a alpha channel value (should always be 0xFF), RR - red color intencity, GG - green color intencity and BB - blue color intencity. To change a pixel you just need to change the value with regular STORE instruction. CPU will ask RAM to store the word at pixels address, RAM will map that address as an address in the video memory range and readdress the call to the video subsystem.

The video memory is an array of values which match current display resolution. So, if our emulation will have a display 120x240 pixels (28800 or 0x7080) we have the video memory from 0x0A000000 to 0x0A00707F.

It would be ineffective, if we would repaint the display each time some pixel gets changed. Rather we have to invoke the repaint procedure on the update of the last pixel in the video memory. In our example that would be the pixel at 0x0A00707F.

**Implementation Steps**

You can follow next implementation plan to complete this project. This is not a complete plan and you can choose the path that suits you and your team. But it would be more convenient to have a debugger and the dump option before you start to implement most of the opcodes.

1. Prototype classes for RAM, OPCODE, CPU, Debugger

2. Memory initialization (copy ROM with initial program to RAM), memory dump

3. Implement execution cycle for the first instruction set (*NOP, HALT, DEBUG, ADD, SUB, MUL, DIV, MOD*)

4. Implement *read* option in the debugger with a simple assembler. It will use OP and DP registers to place opcodes, numbers and string to the memory.

5. Add input and output instructions (*READ, READI, WRITE, WRITEI*)

6. Implement flow control opcodes, comparators and bitwise operations (*JMP, JMZ, JMN, EQ, NEQ, AND, OR, XOR* etc...)

7. Implement procedures to store a memory image to a file and load a memory image from a file.

8. Add a symbol table for the debugger

9. Implement loading and compilation of the SVM assembly code from the file

10. Implement the rest of the opcodes

11. Implement disassembler

12. Map video memory and implement simple display emulation

13. Run, test and debug some SVM code ;)

**Suggestions for improvement**

You can improve the SVM emulator in a number of ways. The instruction set is quite complete, but you can expand it anyway by adding new opcodes. Also you can add new input/output capabilities by mapping unoccupied memory areas to new input/output devices. And you can expand the capabilities of the internal debugger.

**References**

You could read more about virtual machines in following sources:

1. The Art of Computer Programming by Donald E. Knuth, description of MIX computer

2. Structured Computer Organization by Andrew S. Tanenbaum

3. The Java Virtual Machine Specification by Tim Lindholm and Frank Yellin [java.sun.com/docs/books/jvms/second\_edition/html/VMSpecTOC.doc.html](http://www.google.com/url?q=http%3A%2F%2Fjava.sun.com%2Fdocs%2Fbooks%2Fjvms%2Fsecond_edition%2Fhtml%2FVMSpecTOC.doc.html&sa=D&sntz=1&usg=AFQjCNFWS69o70dga-PXhbKHe3ExJjvNdQ)

4. Starting Forth by Leo Brodie - Forth is a good example of a two-stack system