Virtual Machine 2.0 Specification

Version 1.0.5

Batburger

July 9, 2018

1 Definitions

Value A *value* shall be a 32bit number.

Stack The *stack* is a data structure, from which one may only add a *value* or may read, and than always remove, the lastly added *value*.[1]

Pushing *pushing* refers to the act of adding a new value onto the stack.

Poping *poping* refers to the act of reading the lastly pushed *value* from the stack and removing it.

Register A *register* stores one *value* at a time. One can push the stored *value* onto the *stack* or pop a value from the *stack* and store the poped value in the register.

Address An *address* shall be a 32bit number which describes the location of stored data.

Linear Memory *linear memory* refers to a data structure where one can write arbitrary many *values* to an arbitrary *address*[2] [3]. One can also push any *value* stored in the *linear memory* on to the stack.

State The *state* of the vm refers to the data which is stored in the *linear memory*, *registers*, *instruction pointer* and *stack*.

Instruction An *instruction* shall be callable, and modify the state of the vm, when called.

ByteCode The *bytecode* is an array of 8bit numbers which refer to instructions or static *values*[4].

Opcode An *opcode* is a 8bit number[5], which uniquely defines an *instruction*.

Argument A *argument* shall be a *value* which is stored on the *stack* and gets poped by an instruction.[6]

Instruction Pointer The *instruction pointer* is a pointer which points into the *bytecode*.

Float A *float* refers to single-precision float as defined by the IEEE Std 754-2008 - IEEE Standard for Floating-Point Arithmetic[7]

[1] We may refer to the *value* which was added lastly as the *top value*.
[2] Address are numbered starting from 0.
[3] The *linear memorys address's* are in no way related to the *address's* of other data structure.
[4] 4 8bit values are read and sticked together to build one 32bit number.
[5] Usually represented in hex.
[6] We say the poped *value* is the *argument* of the instruction poping it.
[7] http://standards.ieee.org/findstds/standard/754-2008.html

2 High Level Function

The vm keeps track of the Instruction Pointer (IP). The IP points into the bytecode starting at the first byte. The vm functions in a loop, executing the instruction which is defined by the opcode, to which the IP is currently pointing. Every instruction may or may not modify the state of the vm, but must always modify the ip. The vm stops executing when the ip points to the instruction 0x11.

3 Technical notes

Every instruction is defined to be one byte in size. The signage and type of a value is interpreted by the instruction to which it is being passed.[8] The VM only exposes the values stored on the stack to the instructions, but internally the vm stores, besides the value, what instruction pushed that value onto the stack. (This is required so the *return* instruction knows what value was pushed onto the stack by the *call* instruction.)

4 Instruction Set

We use "..." to specify a range of numbers. The first and last values are both included in the range.

Opcode	Name	Description
0xa0...0xa9	readRegister0...readRegister9	Pushes the value stored in a register onto the stack.
0xb0...0xb9	setRegister0...setRegister9	Writes one argument to the given register.
0xc0	setSize	Takes one argument which sets by how many addresses the linear memory gets expanded when calling 0xc3 (alloc)[9]. YOU CAN CALL THIS INSTRUCTION EXACTLY ONES.
0xc1	move	Writes the first argument into linear memory at the address given by the second argument.
0xc2	read	Pushes the value, specified by the address given as first argument, from linear memory onto the stack.
0xc3	alloc	Expands the writeable address range in the linear memory by the size that was set with the instruction 0xc0 a.k.a. setSize.[10]
0xd0	push	Pushes a value, specified by the next 4 bytes in the bytecode, onto the stack.
0xd1	remove	Removes a value from the stack.
0xe0	uadd	Adds two arguments and pushes the result onto the stack. (All used values are interpreted to be unsigned)
0xe1	sadd	Adds two arguments and pushes the result onto the stack. (All used values are interpreted to be signed)
0xe2	fadd	Adds two arguments and pushes the result onto the stack. (All used values are interpreted to be floats)

[8] E.g. some instructions might interpret a value as n float or as a signed/unsigned integer

[9] Addresses are 32bit in size, therefore if you were to set this value to 32 you would, every time you call alloc, increase the linear memory by 1024bit (1 Kibibit).

[10] If you want to write to linear memory you have to call this instruction at least ones; there would be no address to which you cloud write to otherwise.

0xe3	usub	Subtracts two arguments and pushes the result onto the stack. (All used values are interpreted to be unsigned)
0xe4	ssub	Subtracts two arguments and pushes the result onto the stack. (All used values are interpreted to be signed)
0xe5	fsub	Subtracts two arguments and pushes the result onto the stack. (All used values are interpreted to be floats)
0xe6	umult	Multiples two arguments and pushes the result onto the stack. (All used values are interpreted to be unsigned)
0xe7	smult	Multiples two arguments and pushes the result onto the stack. (All used values are interpreted to be signed)
0xe8	fmult	Multiples two arguments and pushes the result onto the stack. (All used values are interpreted to be floats)
0xe9	udiv	Divides two arguments and pushes the result onto the stack. (All used values are interpreted to be unsigned)
0xea	sdiv	Divides two arguments and pushes the result onto the stack. (All used values are interpreted to be signed)
0xeb	fdiv	Divides two arguments and pushes the result onto the stack. (All used values are interpreted to be floats)
0xec	utof	One argument is transformed to be stored in float representation.[11] (Value is interpreted to be unsigned)
0xed	stof	One argument is transformed to be stored in float representation.[11] (Value is interpreted to be signed)
0xee	abs	One argument is turned into a positive number.
0xef	ucmp	Compares two argument; Than pushes a value onto the stack, which represents if arg1<arg2, or if arg1>arg2, or if arg1 = arg2.[12] (All values are interpreted to be unsigned)
0xf0	scmp	Compares two argument; Than pushes a value onto the stack, which represents if arg1<arg2, or if arg1>arg2, or if arg1 = arg2.[12] (All values are interpreted to be signed)
0xf1	fcmp	Compares two argument; Than pushes a value onto the stack, which represents if arg1<arg2, or if arg1>arg2, or if arg1 = arg2.[12] (All values are interpreted to be floats)
0x01	jmp	Sets the ip to the address given by the first argument of this instruction.
0x02	jless	Functions like the *jmp* instruction if, and only if, the last *cmp* determinant that arg1<arg2 (Arguments of *cmp*).

[11] The instruction will store the number represented in its first argument with as much precision as a float allows. If precision is lost, no further actions are taken.

[12] *0* represents arg1 = arg2, *1* represents arg1<arg2, *2* represents arg1>arg2.

0x03	jgreater	Functions like the *jmp* instruction if, and only if, the last *cmp* determinant that arg1>arg2 (Arguments of *cmp*).
0x04	jequal	Functions like the *jmp* instruction if, and only if, the last *cmp* determinant that arg1=arg2 (Arguments of *cmp*).
0x05	jNequal	Functions like the *jmp* instruction if, and only if, the last *cmp* determinant that arg1≠arg2 (Arguments of *cmp*).
0x06	call	Sets the ip to the address specified in its first argument and pushes the address of this instruction in the bytecode onto the stack.
0x07	return	Looks through the stack, top to bottom, and jumps to the address which was put there by a previous *call* instruction.
0x10	int	Triggers the interrupt specified by its first argument.[13] [14]
0x11	halt	Stops the vm.
0x12	nop	Iterates the IP by one with no other changes to the state of the vm.

Table 1: Instruction Set

5 Interrupts

[13]See Section 5

[14]It is not sure which, if any, interrupts will exist.