VM Language Manual

Documentation for users

Introduction

VM is a programming langage inspired from assembly. It consists of a compiler and a virtual machine. The virtual machine emulates the functionning of a simple computer with basic operations and memory.

Both the virtual machine and the compiler are written in C.

The compilation process is divided in different stages :

- If you know the C language, you're probably familiar with the "preprocessor" instructions, which are intended to the compiler. For example, the "#define" instruction in C let the programmer replace certain words by others. My language enhance the same "define" instruction, which is processed through this first step called "precompilation"

- The precomiplation is also used to remove comments, indentation, and replace the name of instructions by numbers, to make the next steps of the compilation easier

- After this firsts steps, the real compilation takes places. It consists of translating the instructions and parameters in actual "bytecode"

The compilation creates a file that can then be runned by the virtual machine.

The VM as well as the compiler do their work with the files passed as an argument.

As the virtual machine interface is not a true terminal, it's an emulation of terminal created with the graphical library SDL 1.2.15. For the VM, the interface is close to a console but has no scroll bar : at the end of the screen, the display of a text causes its deletion then a return to the beginning of the screen for the cursor. The 128 characters of the reduced ASCII table are accepted by the VM. The console is dimensioned in boxes (width = 80 and height = 25)

The VM has 24mb of RAM, the beginning of the RAM contains the executable contained in the ".vm" file passed as an argument. The instructions are executed one by one, the end of the program triggers the closing of the program, therefore of the VM.

As my language is inspired by assembly in its syntax, I've copied its basic features.

Nevertheless, my language is not actually assembly, because the code is not compiled into an executable file runned by the processor but into a bytcode file interpreted by the virtual machine that emulates an actual real processor.

Now let's look at some other attributes of the virtual machine.

Registers

If you don't know what a register is, it's a technology used in CPUs to store data and make operations. High level programmers are not familiar with registers. The only programming language that let the programmer use directly registers is assembly.

The VM has 16 24-bit registers that can each contain an unsigned integer.

Registers R8 to R15 are used by the VM, it is therefore not recommended to place values there.

- R8: pointer to the top of the current stack.

- R9: pointer to the top of the function call stack.

- R10: result of the "test" instruction, also used by "je" and "jne" to know the result of a test.

- R11: this register contains the UNICODE value of the last key hurry.

- R12: this register contains the size of the stack in bytes.

- R13: this register is the instruction pointer.

- R14: register which contains the size of the executable in bytes.

- R15: function return register.

Stack

If you don't know what a stack is, it's also a technology used in CPU to facilitate functions calling.

When you call a function in C, parameters are sent to the function through the "stack". Which means, its pilled up on a "stack" of data, then, the function pop the parameters in the opposite order. It's really useful when you have to call a function inside an other function.

The VM has a 763 byte stack and a function call stack of the same lenght.

Syntax

The syntax of my language looks like assembly.

Example :

```
target "hello world.vm"

        str 100h "Hello world !"

        out 100h
```

```
wait 5000
```

The language consists of set of 40 instructions.

A source code must carry the extension ".vmasm" by default. The first line of a source code must not contain any spaces or tabs before the "target" keyword. "target" is followed by a space then by the name of the target file of the compilation (executable) in quotes. Target files must have the extension ".vm".

Example:

```
target "test.vm"
```

A source file must end with a newline, otherwise the last line will not be taken into account by the compiler. One instruction = one line, it is not possible to write on one line more than one instruction.

Comments also fit on a line and cannot be following an instruction, they begin with the character ";". They don't are not taken into account during compilation.

Example:

A incorrect comment:

```
mov R0, R1; this is a comment
```

A fair comment:

```
; this is a comment
mov R0, R1
```

Numeric values in instructions are written either in decimal, or in hexadecimal. In the first case, they are written normally, in the second, you have to add a 'h' at the end of the numeric value to specify to the compiler that it is with a value in hexadecimal.

Pointers are RAM addresses. A register can contain a pointer like any other numeric value. A pointer is located between square brackets ([R0] or [100h] for example). The dereference of this pointer is by default on 3 bytes.

You just have to add a '~' in front of the pointer so that the dereferencing is only done on a single byte. (~ [60] for example).

Labels are instructions that do not appear in the code compiled. For the compiler, they must start with a “.” who is their only identifier for the user, the case is respected, the numbers are allowed.

Instructions

The list of instructions is exhaustive but their use risks to vary (slightly) during versions:

- "off": this instruction turns off the VM, therefore ends the execution from the program. It is not necessary to call it at the end of the program.

- "time": this instruction places the number seconds elapsed since the start of the program at the point indicated:

time C
- C is a register.

- "add" (addition), "sub" (subtraction), "mul" (multiplication), "div" (division), "mod" (modulo), "shl" (shift left), "shr" (shift right), "or" (or), "and" (et), "xor" (or exclusive) are the instructions for making calculations. here is their mode of use:

op A, B: C

- "op" is the operation ("add", "sub", "mul", ...)
- C is the target memory (register, pointer, ...)

- "mov": this instruction allows you to move a contained value in a register or RAM or one to assign a numeric constant, it is used like this:

mov D, E

- D cannot be a numeric constant.

- "goto": this instruction is used to branch to a label. The label must of course exist, regardless of whether it precedes or not the call of "goto".

Here is an example :

---

target "goto.vm"

.label1

goto label1

; infinite loop

- "test", this instruction performs a test on two registers, here is its syntax:

F, G, H test

- H is a numeric value which corresponds to the operator
comparison:
0 => "! =" (Different from)
1 => "==" (equal)
2 => "<=" (sup. Or equal)
3 => "> =" (inf. Or equal)
4 => "<" (sup)
5 => ">" (inf)

The result is 0 (false) or 1 (true) and is placed in the register R10.

- "je" and "jne": these two instructions are conditional branches: they are used as "goto" with the difference that:

- for "je", the jump will only take place if the value contained in R10 is 1.

- for "jne", the jump will only take place if the value contained in R10 is 0.

- "in": this instruction is used to retrieve an entry, it is used in this way:

in I

- "out": this instruction works like "in", but it allows to display and not enter a string.

(N.B: a string is considered terminated at the first character '\ 0' (ASCII value 0 meet)

- "disp": this instruction displays the numerical value of a register, it is used as this:

disp J

- "str": this instruction creates a character string at a given address, here is its use :

str L "string"

Example: str 250h "character string"

(N.B: it is not necessary to add the character '\ 0' at the end, the comiler takes care of it)

- "char": this instruction is used to assign a character to a given address, it works like "str" but the "string" becomes a character (ex: 'c') placed between two apostrophes or a numerical value.

Example: char 100h 'A'

- "movcurs": this instruction allows you to modify the position of the console cursor. Here is its instructions for use:

movcurs M, N

M is the abscissa and N the ordinate of the new cursor coordinates. If those if are outside the screen, it will cause an error when execution that is supported by the VM.

- "putchar": this instruction has the effect of displaying on the screen a character stored in the RAM.

Here is how to use it:

putchar O
- O is the address of the character to display.

- "wait": this instruction has the effect of pausing the program for a number of milliseconds defined :

wait P
- P is the numerical value of the time in milliseconds.

- "clear": this instruction clears the screen.

- "push": this instruction has the effect of adding an element to the stack:

- push Q

- "pop": this instruction has the effect of retrieving the top element of the stack:

- pop R

- "call": this instruction has the effect of calling a function, in fact it works like the "goto" instruction, i.e. it jumps to a label, but it saves the instruction pointer (EIP) on the call stack to be able to return in case call of the "ret" instruction.

- "ret", this instruction retrieves the instruction pointer on the top of the stack to return to the instruction following the last "call".

- "numstr": the purpose of this instruction is to convert a character string into a number:

- numstr S, T
- S is the address of the chain and T the target.

- "mem": this instruction is used to define a memory block, it
is used like this:

- mem U {V}
- U is the address.
- V is a succession of 8bits value in hexadecimal.

example:

mem 100h {00ff0d59}

- "open": this instruction opens a file, knowing that it is possible to open at the same time 8 files (from 0 to 7):

- open f, C, M
- f is the file number (from 0 to 7)
- This is the address of the channel
which is the path of the file
- M contains the address of the
opening mode string ("r", "rb", "w", etc ...)

- "close": this instruction is used to close a file, by its identifier:

- close f
- f is the file number (from 0 to 7)

- "read": this instruction is used to read a piece of file:

- read f, A, s
- f is the number of the target file
- A is the write address of the block read
- s is the block size

- "write": this instruction works exactly like the read function, except that it allows not to read, but to write to a file. Warning to use the correct mode of opening the file in both cases.

- "seek": this function is used to modify the position of the cursor in the file manipulation:

- seek f, v, r
- f is the file number (between 0 and 7)
- v is the positive or negative displacement value
- r is the starting location
- (start) SEEK_SET = 0
- (current) SEEK_CUR = 1
- (end) SEEK_END = 2