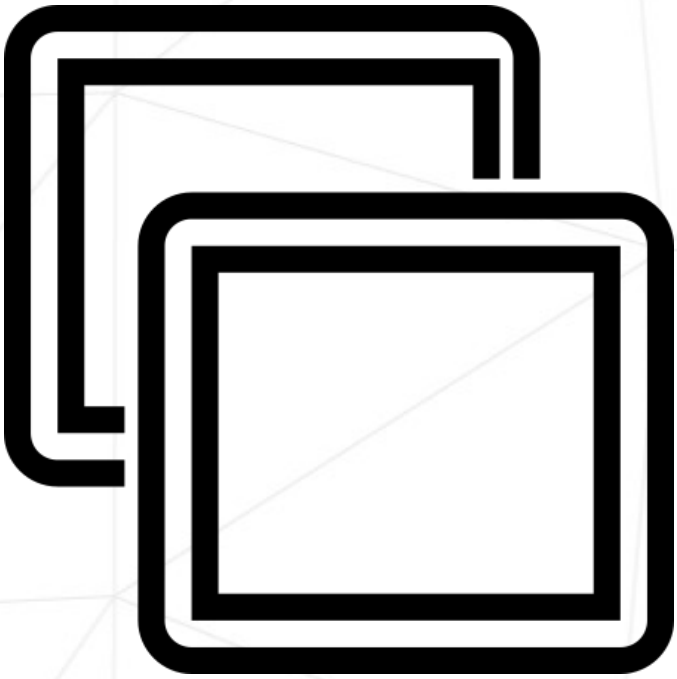

# Elementary VM

*Summary: The purpose of this project is to create a simple virtual machine that can interpret programs written in a basic assembly language.*

# Contents

# Chapter I

# A Machine

A machine, virtual or not, has a specific architecture. The only real difference between a virtual machine and a physical one is that the physical one uses real electronic components, while a virtual one emulates them by using to a program.

A virtual machine is nothing more than a program that simulates a physical machine, or another virtual machine. Nevertheless, it is clear that a virtual machine that emulates a physical machine such as a desktop computer is a very advanced program that requires an important programming experience as well as a very in-depth architectural knowledge.

For this project, requirements will be limited to a very simple virtual machine: it will run some basic arithmetic programs coded in a very basic assembly language. If you want to have an idea of what your program’s capabilities should look like, type the command man dc in your shell.

The virtual machine we are describing has a classical architecture. However you may wonder what a classical architecture is...

There is no easy answer to this question. It depends on the precision you want to adopt, as well as the kind of problem you are looking at. Each "organ" of a machine can be translated into a program or set of (more or less) complex functions. Moreover, this complexity is linked to what your machine is used for in the end. Let’s look at memory for instance. We do agree that the emulation complexity of a machine’s memory between a virtual machine running an operating system, such as Linux or Windows is completely different!

Whatever decision you make, you really should have a look at these articles:

1. http://en.wikipedia.org/wiki/Central_processing_unit
2. http://en.wikipedia.org/wiki/Chipset
3. http://en.wikipedia.org/wiki/Computer_data_storage
4. http://en.wikipedia.org/wiki/Input/output

# Chapter II

# The project

Elementary VM is a machine that uses a stack to compute simple arithmetic Expressions. These arithmetic expressions are provided to the machine as basic assembly programs.

## II.1 The assembly language

### II.1.1 Example

As an example is still better that all the possible explanations in the world, this is an example of an assembly program that your machine will be able to compute:

```
# -------------
# example.evm -
# -------------

put int32(42)
put int32(33)

add

put float(44.55)

mul

put float64(42.42)
put int32(42)

trace

pop

assert float64(42.42)

end
```

## II.1.2 Description

As for any assembly language, the language of Elementary VM is composed of a series of instructions, with one instruction per line. However, Elementary VM's assembly language has a limited type system, which is a major difference from other real world assembly languages.

- **Comments**: Comments start with a '**#**' and finish with a newline. A comment can be either at the start of a line, or after an instruction.

- **put** <*v*>: Pushes the value *v* at the top of the stack. The value *v* must have one of the following form:
  - **int8(*n*)**: Creates an 8-bit integer with value *n*.
  - **int16(*n*)**: Creates a 16-bit integer with value *n*.
  - **int32(*n*)**: Creates a 32-bit integer with value *n*.
  - **int64(*n*)**: Creates a 64-bit integer with value *n*.
  - **int128(*n*)**: Creates a 128-bit integer with value *n*.
  - **float32(*z*)**: Creates a 32-bit float with value *z*.
  - **float64(*z*)**: Creates a 64-bit float with value *z*.

- **pop**: Remove the value from the top of the stack. If the stack is empty, the program execution must stop with an error.

- **trace**: Displays each value of the stack, from the most recent one to the oldest one WITHOUT CHANGING the stack. Each value is separated from the next one by a newline.

- **assert** <*v*>: Asserts that the value at the top of the stack is equal to the one passed as parameter for this instruction. If it is not the case, the program execution must stop with an error. The value *v* has the same form that those passed as parameters to the instruction **put**.

- **add**: Unstacks the first two values on the stack, adds them together and stacks the result. If the number of values on the stack is strictly inferior to 2, the program execution must stop with an error.

- **sub**: Unstacks the first two values on the stack, subtracts them, then stacks the result. If the number of values on the stack is strictly inferior to 2, the program execution must stop with an error.

- **mul**: Unstacks the first two values on the stack, multiplies them, then stacks the result. If the number of values on the stack is strictly inferior to 2, the program execution must stop with an error.

- **div**: Unstacks the first two values on the stack, divides them, then stacks the result. If the number of values on the stack is strictly inferior to 2, the program execution must stop with an error. Moreover, if the divisor is equal to 0, the program execution must stop with an error too. Chatting about floating point values is relevant a this point. If you don't understand why, some will understand. The linked question is an open one, there's no definitive answer.
- **mod**: Unstacks the first two values on the stack, calculates the modulus, then stacks the result. If the number of values on the stack is strictly inferior to 2, the program execution must stop with an error. Moreover, if the divisor is equal to 0, the program execution must stop with an error too. Same note as above about floating point values.

- **print**: Asserts that the value at the top of the stack is an 8-bit integer. (If not, see the instruction assert), then interprets it as an ASCII value and displays the corresponding character on the standard output.

- **end**: Terminate the execution of the current program. If this instruction does not appears while all others instruction has been processed, the execution must stop with an error.

For non commutative operations, consider the stack v1 on v2 on stack_tail, the calculation in infix notation v2 op v1.

When a computation involves two operands of different types, the value returned Has the type of the more precise operand. Please do note that because of the extensibility of the machine, the precision question is not a trivial one. This is covered more in details later in this document.

### II.1.3 Grammar

The assembly language of Elementary VM is generated from the following grammar (# corresponds to the end of the input, not to the character '#'):

```
 1  S := INSTR [SEP INSTR]* #
 2
 3  INSTR :=
 4        put VALUE
 5      | pop
 6      | trace
 7      | assert VALUE
 8      | add
 9      | sub
10      | mul
11      | div
12      | mod
13      | print
14      | end
15
16  VALUE :=
17       int8(N)
18     | int16(N)
19     | int32(N)
20     | int64(N)
21     | int128(N)
22     | float32(Z)
23     | float64(Z)
24
25  N := [-]?[0..9]+
26
27  Z := [-]?[0..9]+.[0..9]+
28
29  SEP := '\n'+
```

### II.1.4 Errors

When one of the following cases is encountered, Elementary VM must raise an exception and stop the execution of the program cleanly. It is forbidden to raise scalar exceptions. Moreover your exception classes must inherit from std::exception.

- The assembly program includes one or several lexical errors or syntactic errors.
- An instruction is unknown
- Overflow on a value
- Underflow on a value
- Instruction pop on an empty stack
- Division/modulo by 0
- The program doesn't have an exit instruction
- An assert instruction is not true
- The stack is composed of strictly less that two values when an arithmetic Instruction is executed.

Perhaps there are more errors cases. However, you machine must never crash (segfault, bus error, infinite loop, unhandled exception, ...).

### II.1.5 Execution

Your machine must be able to run programs from a file passed as a parameter and from the standard input. When reading from the standard input, the end of the program is indicated by the special symbol "**##**" otherwise absent.

Be very careful avoiding conflicts during lexical or syntactic analysis between "**##**" (end of a program read from the standard input) and "**#**" (beginning of a comment.)

Now let see some examples of execution:

```
1 $> ./evm
2 put int32(2)
3 put int32(3)
4 add
5 assert int32(5)
6 trace
7 end
8 ##
9 5
10 $>
```

```
 1 $> cat sample.evm
 2 # ------------
 3 # sample.evm -
 4 # ------------
 5
 6 put int32(42)
 7 put int32(33)
 8 add
 9 put float32(44.55)
10 mul
11 put float64(42.42)
12 put int32(42)
13 trace
14 pop
15 assert float64(42.42)
16 end
17 $> ./evm ./sample.evm
18 42
19 42.42
20 3341.25
21 $>
```

```
1 $> ./evm
2 pop
3 ##
4 Line 1 : Error : Pop on empty stack
5 $>
```

The error message is given as an example. Feel free to use yours instead.

# Chapter III

# Mandatory part

In order to help you with your project, we provide you with the following instructions that you **MUST** respect.

## III.1 Generic instructions

- You are free to use any compiler you like.
- Your code must compile with: **-Wall -Wextra -Werror**.
- You shall use **C++14** version.
- You shall not use any library except **STL**.
- You shall use **CMake** as a build system.
- Any class that declares at least one attribute must be written in canonical form. Inheriting from a class that declares attributes does not count as declaring attributes.
- It’s forbidden to implement any function in a header file, except for templates and the virtual destructor of a base class.
- The “keyword” "***using namespace***" is forbidden.

## III.2 The IOperand interface

Elementary VM uses 7 operand classes that must be generated by compiler from the single template class which you should define in your code:

- **Int8**: Representation of a signed integer coded on 8bits.
- **Int16**: Representation of a signed integer coded on 16bits.
- **Int32**: Representation of a signed integer coded on 32bits.
- **Int64**: Representation of a signed integer coded on 64bits.
- **Int128**: Representation of a signed integer coded on 128bits.
- **Float32**: Representation of a float coded on 32bits.
- **Float64**: Representation of a float coded on 64bits.

Each one of these operand classes **MUST** implement the following *IOperand* interface:

```
class IOperand {

public:

    virtual int             getPrecision( void ) const = 0;     // Precision of the type of the instance
    virtual eOperandType    getType( void ) const = 0;          // Type of the instance

    virtual IOperand const *  operator+( IOperand const & rhs ) const = 0; // Sum
    virtual IOperand const *  operator-( IOperand const & rhs ) const = 0; // Difference
    virtual IOperand const *  operator*( IOperand const & rhs ) const = 0; // Product
    virtual IOperand const *  operator/( IOperand const & rhs ) const = 0; // Quotient
    virtual IOperand const *  operator%( IOperand const & rhs ) const = 0; // Modulo

    virtual std::string const & toString( void ) const = 0; // String representation of the instance

    virtual ~IOperand( void ) {}

};
```

## III.3 Creation of a new operand

New operands **MUST** be created via a "*factory method*". Search Google if you don't know what it is. This member function must have the following prototype:

```
IOperand const * createOperand( eOperandType type, std::string const & value ) const;
```

The ***eOperandType*** type is an enum defining the following values: Int8, Int16, Int32, Int64, Int128, Float32 and Float64.

Depending on the enum value passed as a parameter, the member function *createOperand* creates a new *IOperand* by calling one of the following private member functions:

```
IOperand const * createInt8( std::string const & value ) const;
IOperand const * createInt16( std::string const & value ) const;
IOperand const * createInt32( std::string const & value ) const;
IOperand const * createInt64( std::string const & value ) const;
IOperand const * createInt128( std::string const & value ) const;
IOperand const * createFloat32( std::string const & value ) const;
IOperand const * createFloat64( std::string const & value ) const;
```

In order to choose the right member function for the creation of the new *IOperand*, you **MUST** create and use an array (here, a vector shows little interest) of pointers on when an operation happens between two operands of the same type, there is no problem. Member functions with enum values as index.

## III.4 The precision

However, what about when the types are different? The usual method is to order types using their precision. For this machine you should use the following order: *Int8 < Int16 < Int32 < Int64 < Int128 < Float32 < Float64*. This order may be represented as an enum, as enum values evaluate to integers.

The pure method *getPrecision* from the interface *IOperand* allows to get the precision of an operand. When an operation uses two operands from two different types, the comparison of theirs precision allows to figure out the result type of the operation.

## III.5 The Stack

Elementary VM is a stack based virtual machine. Whereas the stack is an actual stack or another container that behaves like a stack is up to you. Whatever the container, it **MUST** only contain pointers to the abstract type *IOperand*.