# B4 - Synthesis Pool

B-SYN-400

# Abstract VM

a simple virtual machine for a simple assembler

1.11

{EPITECH.}

# Abstract VM

| | |
|---|---|
| **binary name:** | abstractVM |
| **repository name:** | SYN_abstractVM_*$ACADEMICYEAR* |
| **repository rights:** | ramassage-tek |
| **language:** | C++ |
| **compilation:** | via Makefile, including re, clean and fclean rules |

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

The goal of this project is to create a simple virtual machine that is able to interpret programs that are written in a simplified assembler language.
To be precise, it consists of a stack machine that is able to calculate simple arithmetical expressions.
These very arithmetical expressions are given to the machine in the form of simple assembler programs.

## Machines

Whether virtual or not, a machine, such as the one you will read about in this description, has a specific structure.
The only real difference between a *virtual* machine and a physical machine is that the physical machine functions with actual electronic elements, while the virtual machine simulates these elements with a program.

A virtual machine is no more (and no less) than a program that simulates a physical machine or another virtual machine.
However, it is obvious that a virtual machine simulating a physical machine (like a computer, for example), in the ordinary sense, is an excessively complicated program that requires an enormous amount of programming experience and a deep understanding of structure.

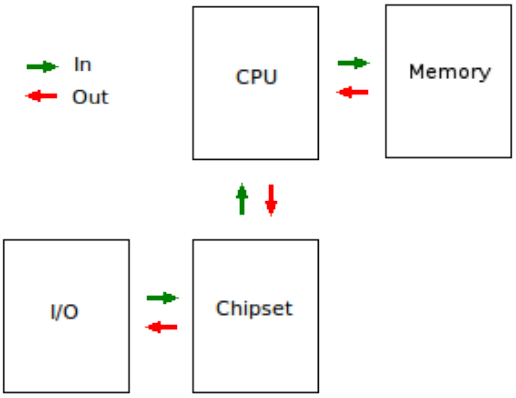
This project will be limited to an extremely simple virtual machine: executing basic arithmetic programs written in an equally basic assembler language.
If you want an idea of what your program will be capable of doing once its finished, type the man dc command in your shell.

## Structure

Our machine will have a classic structure.
However, you are entitled to ask yourself what a classic structure is.
There isn't a simple answer to this question ; actually, it all depends on the exactness you want to take on and the type of problem you want to take into consideration.
Each "organ" of a machine can be translated into a more or less complex program, and the program's complexity depends on what your machine's purpose will be in the end.

Take, for example, the memory.
We all agree that between a virtual machine that is able to run an exploitation system like Linux or Windows, and a CoreWar virtual machine, the machine's memory simulation complexity will be entirely different!

Let's consider the following structure.
Even though this structure is far from being precise, it is correct and could be used as the structure's foundation for this project's machine.




You should ask yourself if its sufficient. It's obvious that you'll need to increase the precision, but perhaps certain elements are missing. It's an open-ended question… The important thing is that you think about it.

Whatever you decide to do, check out some of these Wikipedia pages…
Central Processing Unit, Chiset, Data Storage, I/O.

## ASSEMBLY LANGUAGE

Here's an example of an assembler program that your machine should be able to execute, which is worth more than all the explanations in the world:

```
; file: example.avm

push int32(42)
push int32(42)
add

push float(44.55)
mul

push double (42.42)
push int32(42)
dump
pop
assert double(42.42)
exit
```

Like every assembly language, abstractVM's is composed of a sequence of instructions, but with only one instruction per line; a notable difference between other assembly languages. AbstractVM's is typical.

**comments**
Begin with a semicolon (;) and end at the end of a line. A comment may be indifferent at the beginning of a line or after an instruction.

**push v**
Stacks the v value at the top. The v value will naturally take one of the following forms: int8(n), int16(n), int32(n), float(z), double(z) or bigdecimal(z).
For example, `int16(n)` creates an signed 16-bit integer with the value n.

**pop**
Unstacks the value at the top of the stack. If the stack is empty the program's execution must stop error.

**clear**
Clears the stack, making it empty.

**dup**
Duplicates the value on the top of the stack, and stacks the copy of the value.

**swap**
Reverses the order of (swaps) the top two values on the stack.

**dump**
Displays each value on the stack from the newest to the oldest, WITHOUT MODIFYING the stack.
Each value is separated from the next one with a line break.

**assert v**
Checks that the value at the top of the stack is equal to the one passed as parameter in this instruction.
If it's not the case, the program execution must stop error.
The v value, of course, has the same form as those passed as parameter during the push instruction.

**load v**
Copies the value from the register v and stacks it at the top. if the register v does not contain a value, the program execution must stop error.

**store v**
Unstacks the first value and stores it to the register v.

**print**
Makes sure that the value at the top of the stack is an 8-bit integer (if this is not the case, see the assert instruction), then interprets it like an ASCII value and displays the corresponding character on the standard output.

**sqrt**
Unstacks the first value in the stack, calculates its square root, and then stacks the result.

**add**
Unstacks the first two values in the stack, adds them, and then stacks the result.
If the number of values in the stack is strictly less than 2, the program execution must stop error.

**sub**
Unstacks the first two values in the stack, substracts them, and then stacks the result.
If the number of values in the stack is strictly less than 2, the program execution must stop error.

**mul**
Unstacks the first two values in the stack, multiplies them, and then stacks the result.
If the number of the values in the stack is strictly less than 2, the program execution must stop error.

**div**
Unstacks the first two values in the stack, divides them, and then stacks the result.
If the number of the values in the stack is strictly less than 2, the program execution must stop error.
Also, if the divisor is equal to 0, the program execution must also stop error.

**mod**
Unstacks the first two values in the stack, calculates their modulo, and then stacks the result.
If the number of the values in the stack is strictly less than 2, the program execution must stop error.
Also, if the divisor is equal to 0, the program execution must also stop error.

**exit**
Quits the program execution that is underway. If this instruction does not appear, despite the fact that all of the instructions have been executed, the execution must stop error.

In case you run into non-commutative operations· compute v2 op v1 (in infix notation), where v1 is on top of the stack and v2 under it.

If ever a calculation implies two different operands, the return value will be the more precise type of operand. Please note that if your machine has extensibility problems, the precision question will not be significant.

Our machine has the capacity to store 16 values of any form into registers from 0 to 15.
For example, `load int8(12)`.

## Grammar

The AbstractVM's assembler language is generated from the following simple grammar ('#' corresponds to the end of the input not to the '#' character):

```
S := [INSTR SEP]* #

INSTR : = push VALUE
    | pop
    | dump
    | clear
    | dup
    | swap
    | assert VALUE
    | add
    | sub
    | mul
    | div
    | mod
    | sqrt
    | load VALUE
    | store VALUE
    | print
    | exit

VALUE := int8(N)
    | int16(N)
    | int32(N)
    | float(Z)
    | double(Z)
    | bigdecimal(Z)

N := [-]?[0..9]+
Z := [-]?[0..9]+.[0..9]+

SEP := '\n'
```

## Errors

In the following case(s), AbstractVM must raise an exception and correctly stop the program execution:

- the assembler program has one, or several, lexical or syntactical errors,
- an instruction is unknown,
- overflow on a value,
- underflow on a value,
- pop instruction on an empty stack,
- division/modulo by 0,
- the program does not have an exit instruction,
- an assert instruction is not verified,
- the stack strictly has less than two values during the execution of an arithmetical instruction.

It is forbidden to make scalar exceptions and your exception classes should inherit the STL from std::exception.

## EXEXUTION

Your machine should be able to execute programs from files passed as parameter or from the standard input.
If ever it's read from the standard input, the end of the program will be flagged by the ';;' special symbol.

Be sure not to have coflict during the lexical and syntactical analyses between ';;' (end of a read program on standard input) and ';' (beginning of a comment)

Let's check out a few execution examples:

Terminal

```
~/B-SYN-400> ./abstractVM example.svm
42
42.42
3341.25
```

Terminal

```
~/B-SYN-400> ./abstractVM
push int32(2)
push int32(3)
add
assert int32(5)
dump
exit
;;
5
```

Terminal

```
~/B-SYN-400> ./abstractVM
pop
;;
Line 1 : error : Pop on empty stack
```

The error message is given as an example, feel free to use your own.

# Technical Considerations

In order to help you in the development of your project, you're going to have to comply with the following instructions.
Think about why each instruction is given to you.
None of them are there by chance or to annoy you.

## + IOperand Interface

All of your operand classes must IMPERATIVELY implement the IOperand interface below:

```
class IOperand
{
    public
        virtual std::string toString() const=0; //string that represents the instance
        virtual eOperandType getType() const=0; //returns the type of instance

        virtual IOperand* operator+(const IOperand &rhs) const = 0 // sum
        virtual IOperand* operator-(const IOperand &rhs) const = 0 // diff
        virtual IOperand* operator*(const IOperand &rhs) const = 0 // mul
        virtual IOperand* operator/(const IOperand &rhs) const = 0 // div
        virtual IOperand* operator%(const IOperand &rhs) const = 0 // mod

        virtual ~IOperand() {}
};
```

**Int8** class,
**Int16** class,
**Int32** class,
**Float** class,
**Double** class and
**BigDecimal** class
must implement the IOperand interface.

It is FORBIDDEN to manipulate pointers or references for each one of these five classes. You are ONLY allowed to manipulate IOperand pointers.

Given the similarity between the operands' classes, it could be relevant to use class templates. However, it is not mandatory.

## + Creating a new IOperand

You must write a class named Factory with a static member function that will enable you to create new operands in a general manner.
This member function should be prototyped as follows:

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

The **eOperandType** is an enum that can take the following values: **Int8**, **Int16**, **Int32**, **Float** or **Double**.

Depending on the enum's value passed as parameter, the createOperand member function will create a new IOperand by calling the following private member functions:

```
IOperand* createInt8(const std::string& value);
IOperand* createInt16(const std::string& value);
IOperand* createInt32(const std::string& value);
IOperand* createFloat(const std::string& value);
IOperand* createDouble(const std::string& value);
IOperand* createBigDecimal(const std::string& value);
```

To select the correct member function that is used to create a new IOperand, you should create and use an array (here, the vector has little interest) of pointers about member functions that is indexed by the enum's values.

Your Factory class will be tested independently by compiling a main with your files Factory.h and Factory.cpp.

## + Precision

When an operation takes place between two of the same real types, no problem. But what about if the real types are different?
The usual method is to order the types among themselves in precision order. In our machine's case, this will give the following order:

```
Int8 < Int16 < Int32 < Float < Double < BigDecimal
```

We can cleverly use the **eOperandType** to keep track of this order in our machine.
During an operation involving two operands of different types, the type of the result of the operation can be determined.
In this project, we will consider that the returned type is always the more precise of the two.

## + The Stack

Given that AbstractVM is a stack machine, it possesses a container that acts like a stack.
The choice will seem obvious now, but perhaps you will discover a subtlety that will make you ask yourself the question again later on.

Your container must ONLY contain IOperand abstract type pointers.
It is FORBIDDEN to stock real type operands in your container.

## + Resource Usage

Every resource used by our machine must be freed even when error occurs that includes memory allocations. A programming idiom named **RAII - Ressource Allocation Is Initialization** developed by Bjarne Stroustrup could be useful.

## + Restrictions

You are generally free to implement what you want.
Also, you are allowed to use C++11 standard (supported by gcc 4.8.3).
However, there are a few restrictions:

- all (external) libraries, except the STL are explicity forbidden,
- you must use the STL as much as possible. We should therefore find at least one container and a class exception
  produced from the STL in your project. The use of at least one algorithm from the STL would be appreciated.
  Pay close attention,
- the only authorized libc functions are those that encapsule the call systems and that have no equivalent in C++,
- "split", "strtowordtab" and the rest of them must NOT be used for parsing code (even the assembler!). We're
  providing you with a grammar…use it!
- any value that is passed by copy rather than by reference or pointer must be justified,
- any value that is not const and passed as parameter must also be justified,
- any member function or method that does not modify the standard instance, and that is not const must be
  justified,
- any conditional statements with more than 3 branches (if … else if .. else) must be justified,
- there is no norm in C++. However, any code that we find to be illegible or too messy will be penalized. Be serious!