Project C++

Abstract VM

Koalab koala@epitech.eu

Abstract: *The purpose of this project is to create a simple Virtual Machine that can interpret programs coded with a basic assembly language.*

Contents

Chapter I

A Machine

A machine, virtual or not, has a particular 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 thanks 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 “computer” is a very advanced program requiring an important programing experience as well as a very deep architectural knowledge.

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

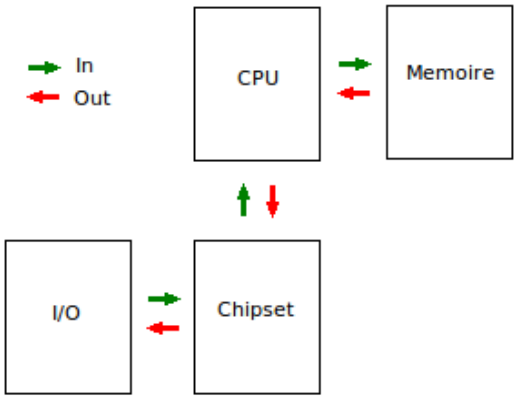
Chapter II

Architecture

The virtual machine we are describing has a classical architecture. However you are allowed to ask what is a classical architecture!?

There are no easy answer to this question. It depends of 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 more or less complex. Moreover, this complexity is linked to what your machine is used for in the end. Let's look at memory for example. We do agree that the emulation complexity of the machine memory between a virtual machine running an operating system, such as Linux or Windows, and a virtual machine running CoreWar is completely different!

Let's consider the following architecture:



Although this architecture is far from being precise, it is a correct one, and it can be used as the machine architecture base for this project.
However, you should seriously ask yourself if this architecture is sufficient or not. It is obvious that it is necessary to increase the precision, but it is also the case that some of the components can be missing. This is not a closed question...
What is important here is that you reflect on the architecture.

Whatever decision you make, we recommend you to have a look at these addresses:

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 III

The project

AbstractVM is a machine with a stack that is able to compute simple arithmetic expressions. These arithmetic expressions are provided to the machine as basic assembly programs.

III.1 The assembly language

III.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:

```
; -------------
; exemple.avm -
; -------------

push int32(42)
push int32(33)

add

push float(44.55)

mul

push double(42.42)
push int32(42)

dump

pop

assert double(42.42)

exit
```

III.1.2 Description

As for any assembly language, the language of AbstractVM is composed of a series of instructions, with one instruction per line. AbstractVM language has a type, which is a major difference from the others 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.

- **push *v***: Stacks the value v at the top of the stack. The value v must have one of the following form:
  - **int8(*n*)** : Creates an 8 bits integer with value n.
  - **int16(*n*)** : Creates a 16 bits integer with value n.
  - **int32(*n*)** : Creates a 32 bits integer with value n.
  - **float(*z*)** : Creates a float with value z.
  - **double(*z*)** : Creates a double with value z.

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

- **dump** : 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*** : Verifies 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 **push**.

- **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.

- **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.

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

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

For non commutative operations, you must consider for the following stack : v1 on v2 on reste_pile, the calculation in infix notation: v2 op v1.

When a calculation involves two operands from 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.

III.1.3 Grammar

The assembly language of AbstractVM is generated from the following grammar (# corresponds th the end of an entry, not to the character '#'):

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

III.1.4 Errors

When one of the following case happens, AbstractVM must raise an exception and stop the execution of the program neatly. It is forbidden to raise scalar exceptions. Moreover

exception classes must inherit from `std::exception` of the `STL`.

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

Perhaps there are others errors. However, you machine must never crash! (segfault, bus error, infinite loop ...)

III.1.5 Execution

You machine must be able to run programs from files passed as parameters or from the standard entry. When parameters are passed by the standard entry, the end of the program is indicated by the special symbol ";;".

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

Now let see some examples of execution:

```
1   >./avm
2   push int32(2)
3   push int32(3)
4   add
5   assert int32(5)
6   dump
7   exit
8   ;;
9   5
10  >
```

```
>cat sample.avm
; -------------
; sample.avm -
; -------------

push int32(42)
push int32(33)
add
push float(44.55)
mul
push double(42.42)
push int32(42)
dump
pop
assert double(42.42)
exit
>./avm ./sample.avm
42
42.42
3341.25
>
```

```
>./avm
pop
;;
Line 1 : Error : Pop on empty stack
>
```

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

Chapter IV

Technical considerations

In order to help you with your code, we give you the following instructions that you have to respect. For each instruction, you must understand why this particular one is imposed. We are not providing these instructions randomly or to bore you to death!

IV.1 The IOperand interface

Each of your operand classes MUST implement the following IOperand interface:

```
1  class IOperand
2  {
3  public:
4
5    virtual std::string const & toString() const = 0; // Renvoie une string reprensentant l'instance
6
7    virtual int getPrecision() const = 0; // Renvoie la precision du type de l'instance
8    virtual eOperandType getType() const = 0; // Renvoie le type de l'instance. Voir plus bas
9
10   virtual IOperand * operator+(const IOperand &rhs) const = 0; // Somme
11   virtual IOperand * operator-(const IOperand &rhs) const = 0; // Difference
12   virtual IOperand * operator*(const IOperand &rhs) const = 0; // Produit
13   virtual IOperand * operator/(const IOperand &rhs) const = 0; // Quotient
14   virtual IOperand * operator%(const IOperand &rhs) const = 0; // Modulo
15
16   virtual ~IOperand() {}
17 };
```

Operand classes implementing the IOperand interface are the following ones:

- **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.
- **Float** : Representation of a float.
- **Double** : Representation of a double.

Important : It is FORBIDDEN to manipulate pointers or references on each of these 5 classes. You are allowed to manipulate pointers ONLY on IOperand.

Considering similarities between operand classes, it can be relevant to use template classes. However, this is not mandatory.

IV.2 Creation of a new IOperand

You have to write a member function of a class **relevant** of your machine that will allows you to create new operands an a generic way. This function must have the following prototype:

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

The type `eOperandType` is an enum that can accept the following values:

- Int8
- Int16
- Int32
- Float
- Double

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

```
1   IOperand * createInt8(const std::string & value);
2   IOperand * createInt16(const std::string & value);
3   IOperand * createInt32(const std::string & value);
4   IOperand * createFloat(const std::string & value);
5   IOperand * createDouble(const std::string & value);
```

In order to choose the right member function for the creation of the new `IOperand`, you MUST create and use an array (here, `vector` shows little interest) of pointers on member functions with enum values as index.

IV.3 The precision

When an operation happens between two operands from the same type real, there is no problems. However, what about when the types real are different?

The usual method is to sequence types using their precision. For the machine you should use the following order:

```
1    Int8 < Int16 < Int32 < Float < Double
```

To use this sequence for the machine, it is possible to associate an integer with each type to maintain the order, thanks to an enum for example.

The method pure `getPrecision` of the interface `IOperand` allows to get the precision of an operand. When an operation use two operands from two different types, the comparison of theirs precisions allows to figure out the result type of the operation.
For the project, the type returned is the more precise type of the two operands.

IV.4 The Stack

AbstractVM is a machine with a stack. It possesses a container behaving as a stack, or a stack itself. The choice seems obvious now, but perhaps you'll discover a subtlety later that will make you reconsider your choice.

The container must contain **ONLY** pointers on the abstract type IOperand! It is **FORBIDDEN** to store types real of operands in your container.

Chapter V

Instructions

You are more or less free to implement your machine how you want it. However there are certain rules:

- All libraries except for `STL` are strictly prohibited.
- You must use `STL` as much as you can. Your project must contains at least one container and one exception class from `STL`. The use of at least one algorithm from `STL` will be appreciated. Be very careful!
- The only functions of the `libc` that are authorized are those one that encapsulate system calls, and that don't have a C++ equivalent.
- "split", "strtowordtab", and so on must **not** be used for parsing code (even assembler!). A grammar is provided, use it!
- Each value passed by copy instead of reference or by pointer must be justified. Otherwise, you'll loose points.
- Each non `const` passed as parameter must be justified. Otherwise, you'll loose points.
- Each member function or method that does not modify the current instance and which is not `const` must be justified. Otherwise, you'll loose points.
- There are no `C++` norm. However if a code is reckoned to be unreadable or dirty, this code will be penalized. Be serious please!
- Keep an eye on this project instructions. It can change with time!

Chapter VI

Turn in instructions

You must turn in you project in the repository provided by Epitech.

Repository will be cloned at the exact hour of the end of the project, intranet `Epitech` being the reference.

Only the code from your repository will be graded during the oral defense.

Good luck!