

Assignment Description for Project 6 (Virtual Machine I)

The project corresponds to Chapter 7 (Virtual Machine I) of the book. Please read this chapter of the book thoroughly. You will essentially write the first part of a compiler from the VM language to Hack assembly.

Objective: Get a deep understanding of the operation of the VM and of the process of compiling VM code into assembly code, which essentially corresponds to what is done in the 'backend' of a modern compiler.

Contract: Create a VM language parser and a VM to Hack assembly compiler by extending the C++ source code or the Python source code provided. Implement 1 member function of the VMParser class and 2 member functions of the CodeWriter class.

The functions in C++ are the following:

```
void VMParser::parse()
void VMCompiler::writePushPop(CmdType cmd, string segment, int index);
void VMCompiler::writeArithmetic(string command);
```

The functions in Python are the following:

```
Parser._Parse(self)
CodeWriter.WritePushPop(self, commandType, segment, index)
CodeWriter.WriteArithmetic(self, command)
```

Resources: You will extend the provided VMParser and the CodeWriter classes. You find the classes in the corresponding cpp or py files with the same name. You can choose the language you prefer.

For C++: You are also provided the corresponding header files (.h) and the VM2ASMCompiler.cpp file, which contains the main function of the program to be compiled. Finally, you are also provided a Makefile that you can use to compile the program.

For Python: You are also provided the hvm.py file, which contains the main() function of the code to be executed.

We recommend that you use the C++ or the Python code provided to implement the project, but you can of course port the code to Java, and implement the above functions in Java. For the rest of the resources (VMEulator and Hardware emulator) please refer to Chapter 7.5 of the book.

Background

The provided code is consistent with the interface specification provided in Chapters 7 in the book. It contains the implementation of the functionality that is not strictly related to the task of compiling VM language code into assembly language code.

Details

C++: The provided code consists of 5 C++ files and a makefile. The makefile is provided for Unix/Linux systems. On Linux you can use 'gcc' or 'c++' for compilation. On Windows systems you can use Visual C++ (available to KTH students for free via the Intranet, or the Express edition for free from Microsoft). If you are not familiar with the Visual C++ IDE, you can use *nmake* (part of Visual C++) together with the Makefile, but then you have to include the 'bin' directory of Visual C++ into the path.

VM2ASMCompiler.cpp: This file contains the 'main' function that uses the functionality implemented in the classes VMParser and CodeWriter to compile a set of input VM files into a single ASM file. The syntax for calling the compiled program is

```
./VM2ASMCompiler [-nosysinit] <outfile.ASM> <infile1.VM> <infile2.VM> ...
```

For Project 6 you will have to invoke the compiler with

```
./VM2ASMCompiler -nosysinit target.ASM source1.VM source2.VM source3.VM
```

Which will compile the VM files source1.VM, source2.VM and source3.VM into the ASM file target.ASM, and it would *not* insert a call to 'sys.Init'. You will deal with the implementation of calling 'sys.Init' in Project 7.

VMParser.h: Header file of the VMParser class. It contains the class definition with comments.

VMparser.cpp: Implementation of the VMParser class. The implementation of the function
VMParser::parse

is missing and should be done. The parameters and the expected behavior of the function to be implemented are described as comments in the file, and are the same as those specified in the book.

CodeWriter.h: Header file of the CodeWriter class. It contains the class definition with comments.

CodeWriter.cpp: Implementation of the CodeWriter class. The implementation of 9 functions is missing. Each of the nine functions is responsible for compiling a particular VM command type (there are 9 types). Out of these 9 functions you will have to implement 2 in this project. We recommend that you implement the 2 functions in the following order:

```
CodeWriter::WritePushPop
CodeWriter::WriteArithmetic
```

Three of the functions are responsible for creating label names to be used by other functions. The parameters and the expected behavior of the functions are described as comments in the file.

Python: The provided code consists of 4 files, compatible with Python 2.x.

hvm.py: This file contains the 'main' function that uses the functionality implemented in the classes VMParser and CodeWriter to compile a set of input VM files into a single ASM file. The program function takes 1 optional command line argument and one mandatory command line argument. The mandatory argument is either the name of the VM file to be processed or the name of a directory, in which case all .VM files in the directory will be processed. The syntax for calling the script is

```
python hvm.py [-nosysinit] <sourcefile(s)>
```

For Project 6 you will have to invoke the compiler with

```
python hvm.py -nosysinit source.VM
```

which will compile the VM file source into the ASM file source.ASM, and it would *not* insert a call to 'sys.Init'.

You will deal with the implementation of calling 'sys.Init' in Project 7.

hvmParser.ph: Implementation of the parser.

hvmCommands.py: Definitions of the command type values.

hvmCodeWriter.py: Implementation of the CodeWriter class. The implementation of 9 functions is missing. Each of the nine functions is responsible for compiling a particular VM command type (there are 9 types). Out of these 9 functions you will have to implement 2 in this project. We recommend that you implement the 2 functions in the following order:

```
CodeWriter::WritePushPop
CodeWriter::WriteArithmetic
```

Testing of the compiler

1. We recommend that you first implement the parse() function in VMParser.cpp. You can test the implementation by creating a VM code file that contains one of each command type and then by looking at the result of the parsing. The result appears on the screen (stdout) when running the compiler. If the parser is correct, you can switch off this feature for later use (steps 2-4) by commenting out line 12 of VM2ASMCompiler (#define __VERIFY_PARSER 1).
2. Once the parser is correct you can start with the first code writing function (WritePushPop) and then you can continue with the second code writer function (WriteArithmetic). We recommend that once you are done with the first function you test your compiler on the MemoryAccess program, and once you are done with the second function, you test your compiler on the StackArithmetic program provided for Chapter 7 using the CPU emulator (5 tests).

Notes for C++ programming

- The C++ implementation files contain #include "stdafx.h" on line 3, which has to be commented out if you are *not* using MS Visual C++.
- Please use standard C++ 1998, 2003 or 2007 to ensure that the code can be compiled using any C++ 2007 compliant compiler.

The easiest way to write the string 'line1' to the ASM output file is to use

`'outFile << "line1\n";'`. You can write several lines with a single command like this `'outFile << "line1\nline2\nline3\n";'`. To test whether a string (e.g., `string1`) contains a particular text you can use `'if (string1=="mytext") {}'`.

Given a string variable you can use the `'erase'` method to remove characters from the string. For example, given the variable `str="my dog"`, you would use `'str.erase(0,3)'` to remove 3 characters starting at position 0 (i.e., the first three characters of the string), and the results would be `str="dog"`.

Given a string variable you can use the `'find'` method to figure out whether the string variable contains a particular text and at what position. For example, given the variable `str="my dog is blue"` you could use `str.find(„dog“)` to check whether the string „dog“ is contained in `str`. The function would return the value 3, because „dog“ starts at position 3 in `str`.

You can convert a string to integer by using the function `atoi`. Given a string variable `string str="23"`, you would use `'int intValue = atoi(str.c_str());'` to convert it into an integer. The result of the conversion is 0 if the string does not contain an integer in textual form.

You can obtain the length of a string variable `str` by using the method `size()`, i.e., you would write `str.size()`. This will return the number of characters contained in `str`.

- `SimpleFunction.vm` may contain a single `'/'` character on the first line. Fix it to `'//'` before compiling the file to avoid errors.

Submission

You should submit your source files (for C++ submit `VMPParser.cpp` and `CodeWriter.cpp`, for Python submit `hvmParser.py` and `hcmCodeWriter.py`), which should include your implementation of the above named functions. If you port the code to another language you should include **all** source files.

In one zip-archive, include the source files and the filled in declaration sheet. Please use the convention `EP1200-seminarM-GroupN-firstname-familyname` for the filename.