CPSC 213 – Assignment 1
Numbers and Memory

Due: Saturday, September 19, 2020 at 11:59pm

Overview

One preliminary goal for this assignment is for you to familiarize yourself with the computing environment we will be using this term. You will learn about the UNIX command line, how to transfer files to the department UNIX servers, and how to submit assignments using the *handin* program. You will also setup an IntelliJ project for the *Simple Machine* processor simulator.

The assignment consists of five parts. The marks for each part and for each question within the parts are listed as a percentage as like this [xx%].

What You Need to Do

Part 1: The UNIX Command Line

You can get a UNIX Command Line (i.e., a *shell*) in one of the following ways:

1. You can login to one of the department server machines directly in the lab or remotely from your own machine. To do this from the Mac use *ssh*. To do this from Windows install and run *putty* or *xshell*.

2. If you run Mac or Linux you can get a shell directly on your computer via the built-in terminal app.

3. If you run a recent version of Windows 10, you can install the *Windows Subsystem for Linux* (WSL, aka Bash on Windows), which provides a usable Linux environment with a shell.

4. If you're on a previous version of Windows, you can install *Cygwin*, which emulates Linux to some degree, but this is not a recommended configuration.

You'll also need to install the Java JDK. This is preinstalled on the department servers, but on your own machine you might need to download and install it from Oracle's website, or using your system package manager.

Pick your environment. Then do the following.

1. Create a file called `HelloWorld.java` and edit it so that it will print "`Hello World`" when it *runs*. You'll do this by creating a static method called *main* that prints that string.

2. Compile this class **from the command line** using the `javac` command.

3. Run this class **from the command line** using the `java` command.

4. Follow the instructions in the final section to use *handin* to submit what you've done so far. Then use *handin* again to verify that your code was submitted properly. The goal here is to test out the process of running *handin*. You'll run *handin* again later when you finish the rest of the assignment.

Part 2: Install the Simple Machine

Most of you will want to use IntelliJ to edit and run the simple machine simulator we will use throughout the term. It is also possible to use Eclipse or another other IDE or editor you like (`emacs` and `vim` are possibilities).

To install the simulator in IntelliJ, download:

- *www.students.cs.ubc.ca/~cs-213/cur/resources/sm-student-213-intellij.zip*

Then follow instructions in *Companion Section B.1* for IntelliJ. Note that you'll find a link to the Companion readings on the course Piazza resources page.

To use an IDE other than IntelliJ, unpack one of these zip files to select the files `SimpleMachineStudent.jar`, `SimpleMachineStudentSrc.zip`, and `SimpleMachineStudentDoc213.zip`. Then install them into your IDE. The first file contains jar files you need to include in your path, the second contains the source of the simulator, and the third is Java Doc.

Finally, download the solution / reference implementation of the simulator from
- *www.students.cs.ubc.ca/~cs-213/cur/resources/SimpleMachine213.jar*.

Save this file for later assignments. *Companion Section B.5* describes how to run this version. You can, for example, run it from the UNIX command line by typing:

```
java –jar SimpleMachine213.jar
```

Now, check to see whether your installation was successful by following the instructions in Companion Section B.4 to run the simulator. If all went well, the simulator will start and show a window like this:
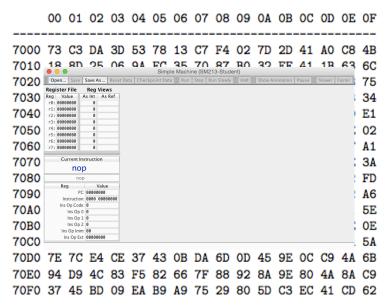
Part 3: Endianness and Hex Questions [20%]

```
     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
    --------------------------------------------------
7000 73 C3 DA 3D 53 78 13 C7 F4 02 7D 2D 41 A0 C8 4B
7010 18 8D 25 06 9A EC 35 70 87 B0 32 EE 41 1B 63 6C
7020                                              75
7030                                              34
7040                                              E1
7050                                              02
7060                                              A1
7070                                              3A
7080                                              FD
7090                                              A6
70A0                                              5E
70B0                                              0E
70C0                                              5A
70D0 7E 7C E4 CE 37 43 0B DA 6D 0D 45 9E 0C C9 4A 6B
70E0 94 D9 4C 83 F5 82 66 7F 88 92 8A 9E 80 4A 8A C9
70F0 37 45 BD 09 EA B9 A9 75 29 80 5D C3 EC 41 CD 62
```

Simple Machine (SM213-Student)

Open... | Save | Save As... | Reset Data | Checkpoint Data | Run | Step | Run Slowly | Halt | Show Animation | Pause | Slower | Faster

Register File — Reg Views

Reg	Value	As Int	As Ref
r0:	00000000	0	
r1:	00000000	0	
r2:	00000000	0	
r3:	00000000	0	
r4:	00000000	0	
r5:	00000000	0	
r6:	00000000	0	
r7:	00000000	0	

Current Instruction
nop
nop

Reg	Value
PC:	00000000
Instruction:	0000 00000000
Ins Op Code:	0
Ins Op 0:	0
Ins Op 1:	0
Ins Op 2:	0
Ins Op Imm:	00
Ins Op Ext:	00000000

Table 1: Contents of Memory from Address 0x7000 to 0x70FF.

Answer these questions.

1. [4%] What is the value (in hex) of the little-endian, 4-byte integer stored at location `0x70d8` in Table 1?

2. [4%] What is the value (in hex) of the big-endian, 8-byte long stored at location `0x7020` in the table above?

3. [4%] Give an example of a 4-byte integer whose big- and little-endian representations are identical. Can you generalize this example?

4. [8%] The Hubble Space Telescope labels the image data it collects with sky positions using *right ascension* / *declination* coordinates. It downloads this data as binary files that are accessible on the Internet. You've decided that you'd like to take an up-close look at *Proxima Centauri,* the nearest star to earth after the sun, whose position is RA 14h 29m 42.9s, D -62° 40' 46.1".

   Hubble image files encode position coordinates using two 4-byte integers, one for right ascension and the other for declination. And so the position of *Proxima Centauri* would be labeled as RA=521,829 and D=-2,256,461.

   ```
   RA =    14 * 36000 + 29 * 600 + 42.9 * 10  =    521,829
   D  = -(62 * 36000 + 40 * 600 + 46.1 * 10) = -2,256,461
   ```

   So you write a program to download a portion of the Hubble dataset and search it for images containing these coordinates. You discover, however, that Hubble apparently never took any images of *Proxima Centauri*. You call the head of NASA to complain bitterly. She tells you that they have taken thousands of pictures of *Proxima Centauri* and suggest that perhaps you are an idiot.

Then you note that the computer on the Hubble that generated the coordinates is the DF-224 manufactured by Rockwell Autonetics in the 1980's and the computer on which your program is running uses an Intel Core i7 processor that you recently purchased — and then you realize that something you learned in CPSC 213 might actually be useful.

What did you realize and what are the correct values of the two integers that you should use in your program to search for *Proxima Centauri*?

HINT: Use a calculator or a program to convert the numbers 521,829 and -2,256,461 to hex. Then think about how you might need to manipulate these hex values. You can give your answer in hex or convert it back to decimal; your choice. To print the hex value of a number in Java:

```
System.out.printf("0x%x\n", i);
```

Part 4: Programming Test of Endianness [30%]

Download the file *www.students.cs.ubc.ca/~cs-213/cur/assignments/a1/code.zip*. It contains the skeleton of an executable Java class in the file `Endianness.java`. Place this file directory on a UNIX machine (e.g., one of the lab machines, a Mac, or Windows running Cygwin/WSL) and compile it from the UNIX command line like this:

```
javac Endianness.java
```

You can now run this program from the command line. The program takes four command-line arguments. These arguments are the values of four consecutive bytes (in hex) of memory from which the program will construct both big-endian and the little-endian integers. For example, to see the value of the integer whose byte values are 0x01, 0x02, 0x03, and 0x04, in that order, you would type:

```
java Endianness 01 02 03 04
```

[25%] Write the code that transforms this memory into integers by replacing the TODOs with an implementation of `bigEndianValue` and `littleEndianValue.`

[5%] Test your program carefully by calling it from the command line with various memory values. Ensure that your tests provide good coverage and be sure to include some tests with bytes that have bit-eight set to 1 (*e.g.*, 0xff or 0x80). List and carefully describe each of the tests you perform and place this description in the file `P4.txt`. Your mark here will be based the list of tests you ran and the description you provide.

Part 5: Implement the Simple Machine *MainMemory* Class [50%]

Like a real processor, the simulator has a memory and a CPU. You will implement both of them as java classes. This week you will implement the memory.

Some portions of the memory are already implemented. Your job is to implement and test the five methods of `MainMemory.java` labeled with `TODO`'s. You will find this file in your IntelliJ environment in the `arch.sm213.machine.student` package.

[35%] Implement the following methods.

- [10%] `isAccessAligned` that determines whether an address is aligned.

- [10%] `bytestoInteger` and `integerToBytes` that translate between an array of bytes and a *big endian* integer. You can use your solution to Part 2 for this.

- [15%] `get` and `set` that provide array-of-byte access to memory used by the CPU to fetch and to store data. Follow the specification listed in the javadoc comments carefully. Note, for example, that the address provided to these methods is not required to be aligned.

[15%] Create a set of `JUnit` tests to test your implementation. Place all of your tests in a class named `MainMemoryTest` in the same package as the `MainMemory` class. Note that you will not actually be able to run the simulator itself yet beyond the initial screen, because you will still lack a CPU implementation.

Ensure that your tests provide good test coverage for each of five methods you implemented. Comment each test to explain what it is testing.

Plan Text Files

THIS IS IMPORTANT

All of the files you submit for this assignment and all future assignments must be *plain text* files.

Java files produced by IntelliJ are examples of this sort of file.

In this assignment you are also turning in several files ending in ".`txt`" that contains textual information. You might be tempted to created and edit these files in MS Word, for example. But, Word does not create plain text files. The files it creates contain formatting information in addition to the text and end in ".`docx`", ".`doc`", or maybe even ".`rtf`". These will not work and you may lose marks for submitting invalid files! Be warned that some operating systems hide file extensions by default, and may therefore show `.txt` when the file is really a `.txt.doc`.

Be sure that you create your files in a plain text editor or export them to plain text format from whatever editor you use.

What to Hand In

In your home directory create a directory named `cs-213` and then inside that directory create a subdirectory named named `a1` that contains the following files:

1. `HelloWorld.java`

2. `Q1.txt` … `Q4.txt` – containing your answers to Questions 1-4, respectively.

3. `Endianness.java` – your solution to Part 4.

4. `P4.txt` containing your test description for Part 4.

5. `MainMemory.java` – your `MainMemory` solution

6. `MainMemoryTest.java` – your `JUnit` tests for Part 5.

Be sure this directory contains these files and nothing else and then follow the instructions in the next section to submit your assignment.

How to Hand In

You must use a program called `handin` to submit your assignment. You will find instructions on how to do this here:

www.cs.ubc.ca/students/undergrad/services/account

You should run `handin` program from the UNIX command line. Instructions for using the command-line version are here (note that the course name is `cs-213`):

my.cs.ubc.ca/docs/handin-instructions

As indicated in the instructions, place your assignment files in a directory called `~/cs-213/a1`.

You can hand in as many times as you want up to the grace-period deadline. The last hand in wins.

The instructions also tell you how to verify that your handin was successful. You should always double check.

The hand-in script will perform certain automated marking tasks and assign a tentative mark for certain parts of the assignment. But, since automated marking has limitations, TAs will review these automated marks and make corrections where appropriate.