Systems II — Project 0
Completing a BIOS

1 Overview and motivation

This project will begin your work with *Fivish*, our (mostly) RISC-V-based assembler and system simulator. The goal is to get working with these tools and see a small group of sample assembly programs. The real work, though will be in the BIOS; it is mostly written, but you will need to complete it.

2 Getting started

2.1 System requirements and options

First, be sure that you have completed the *tools installation*, with a working *Micromamba* environment.

Expect to work significantly on the command-line and with fundamental text/code editors. Specifically, `vim`, `emacs`, *Sublime Text*, even *Notepad++* are reasonable options for editing code. Maybe *VSCode*. IDE's may make assumptions about the kind of software that you're trying to write that don't apply here, so find a good programmer's editor that allows you to read, write, and format assemly, C, and scripts.

Getting into your `sys2` environment: Remember that you want to be in the habit of starting a `bash` shell with your *Mamba environment*. First:

- **macOS/Linux:** Open a terminal/console.

- **Windows:** Then, in the *Start* menu, type `git bash`, and then select *Run as administrator*. Normally, just running this program as a user, in the usual way, is sufficient, but we are going to need administrator capabilities during this setup.

  Once you are at the `bash` shell prompt, do the following to enable the creation of symbolic links:[1]

  ```
  $ printf "export MSYS=winsymlinks:nativestrict\n" >> ~/.bash_profile
  $ source ~/.bash_profile
  ```

[1] If you don't know what a symbolic link is, that's OK, although you can look it up if you're curious. We will be using them, so we need them enabled.

Second, activate your environment:

```
$ micromamba activate sys2
```

2.2 Getting *Fivish*

On whatever system your using, use `git` to checkout a copy of the assembler and simulator:

```
(sys2) $ git clone git@gitlab.com:amherst-college/sfkaplan_grp/fivish.git
```

For this project, we are interested in two directories in this repository: `assembler` and `simulator`. Change into each directory in turn, building the source code:[2]

```
(sys2) $ cd fivish/assembler
(sys2) $ javac *.java
(sys2) $ cd ../simulator
(sys2) $ javac -cp jline.jar *.java
(sys2) $ cd ..
```

You then need to add *symbolic links* from your local binaries directory to the scripts that run these tools, like so:

```
(sys2) $ cd assembler
(sys2) $ ln -s $PWD/f-assemble ~/.local/bin/
(sys2) $ cd ../simulator
(sys2) $ ln -s $PWD/f-simulate ~/.local/bin/
(sys2) $ cd ../..
```

2.3 Getting the project source

Download and unpack the source files from the command-line like so, and then take a look at the files.

```
(sys2) $ curl -L https://bit.ly/cosc-275-25s-p0 -o project-0.tar.xz
(sys2) $ tar -xJvpf project-0.tar.xz
(sys2) $ cd project-0
(sys2) $ ls
bios.asm  conditional.asm  do-nothing.asm  find-max.asm  loop.asm
```

The latter four files are small assembly examples of basic programming constructs that we will go over in class. The `bios.asm` file is where you will do your real work.

[2]When you compile the simulator, you will see a warning about *unchecked or unsafe operations*; you may safely ignore this warning.

2.4 Assembling and running

As an example, we can use the `do-nothing.asm` code to see how to use the assembler and the simulator. First, open the source code in your favorite text editor; you will see that this little program loads a couple of constants into registers, and then it halts the processor.

To assemble this program, do the following:

```
(sys2) $ f-assemble do-nothing.asm
```

You should see the output of the assembler, which shows how each line of code is interpreted and translated into machine code. The assembler will create a file named `do-nothing.vmx` that contains the actual RISC-V machine code. We can now try running this code in the simulator:

```
(sys2) $ f-simulate do-nothing.vmx
[...]
[pc = 0x00015000]: step 10
```

You will see the processor carry out the instructions of the program (which actually takes fewer than 10 steps, but the halted processor ignores the request for any additional steps). You can match these steps to what the assembler showed. You can also examine the registers used to see that they contain the desired values, and then exit the simulator.

```
[pc = 0x00015018]: showregister a0
a0 = 0x0000000d
[pc = 0x00015018]: showregister a1
a1 = 0x1a2b3c4d
[pc = 0x00015018]: exit
```

Congratulations! You have used our little system. Now try the other sample programs and make sure that you see how they work.

3 Your assignment

Your goal with this assignment is to complete, in assembly, the code needed to bootstrap our simulated system. Much of this BIOS is written, but it lacks one critical portion of its code that you must fill in. So, to complete this assignment, you must:

1. **Grok:** Read the code already in `bios.asm` and grasp, to the greatest extent possible, what it does and how it does it. I recommend also writing down questions about parts you don't fully understand, and then asking about them. As a broad outline, here's what the BIOS, when complete, must accomplish:

(a) Find the **RAM** in the physical address space.

(b) Find the **second ROM** in the physical address space. Both your BIOS and users of it assume that this second ROM is the *kernel*.

(c) Copy the kernel (2^{nd} ROM) into main memory (RAM).

(d) Jump to the copied kernel's first machine code instruction.

2. **Complete:** The procedure `copy_kernel` is not written. Given a pointer to the device table entry for the kernel (the 2^{nd} ROM) in `a0`, as well as a pointer to the device table entry for RAM in `a0`, this procedure needs to copy the contents of that ROM into RAM. You can write a loop to perform this task or you can use the *DMA portal* for efficiency.

Notice that you must, for now, use a *dummy kernel*. Since we have not yet written any part of a true OS kernel, then any executable image (e.g., `do-nothing.vmx`, assembled from the provided `do-nothing.asm`) will suffice. To run the simulator with both your BIOS and this dummy program as a "kernel", do this:

```
(sys2) $ f-simulate bios.vmx do-nothing.vmx
```

You will see, when the simulator starts, that the device table has two entries for ROM-type devices; the first contains `bios.vmx`, the second `do-nothing.vmx`. That is, each `.vmx` file is turned into a separate ROM on the bus.

4 How to submit your work

For this course, I will create a Google Drive folder that is shared between you, me, and the TA. Within this folder will be subfolders for each project. For this project, there will be a `project-0` subfolder.

Copy your completed `bios.asm` file into this subfolder to submit your work.

This assignment is due at **11:59 pm** on **Sunday, February** 9^{th}**.**