S 2025-COMS-3210 | Piazza QA

Due Friday 3/28 10 pm. Late deadline (with 10 percentage point penalty) 3/29 10 pm.

The emulator is on Pyrite (ssh to pyrite.cs.iastate.edu) at /share/cs321/legv8emul. It can be executed in place there. You don't need to copy it.

You may work in teams of 1 or 2 on this assignment. EXACTLY ONE MEMBER OF YOUR TEAM SHOULD SUBMIT THE ASSIGNMENT IN CANVAS! THE TAs WILL ASSESS A PENALTY IF MORE THAN ONE MEMBER OF YOUR GROUP SUBMITS THE ASSIGNMENT, AS THIS CREATES A SIGNIFICANT OVERHEAD FOR THEM WHILE GRADING. Similarly, you will be assessed a penalty if you fail to correctly denote all members of your team with your submission. Your submission should be headed by a comment giving the names and NetIDs of each member of your team.

You will be writing a program in LEGv8 assembly. Contrary to the claims in the textbook, there is not an emulator that comes with the book; in its place, one was developed in house. Our emulator is Linux only. There is a statically-linked copy on Pyrite in /share/cs321/legv8emul (you can execute it directly from there). We've also attached a copy here: legv8emul. Being statically linked, it should work on any Linux system and the Windows Linux Subsystem. Since your U drive is mapped to Pyrite, simply running your programs on Pyrite and editing them on your favorite platform will probably be simplest for most of you. Information about how to access Pyrite is given at the bottom of this document.

You will be implementing heapsort in LEGv8 assembly. In a high(er) level language, implementing the heapsort in a single function should be straightforward for students with your level of experience, but in assembly, it gets complicated fairly quickly. To ameliorate some of the complication, we'll break the implementation into a number of procedures, none of which require nested loops. Descriptions and implementations of these functions are given in the attached C file.

All procedures are to be implemented in LEGv8 assembly. All integers may be assumed to be 8 bytes. **All procedures must be implemented as procedures**, meaning that they adhere to the ARMv8 calling and register use conventions. **You will be penalized for failing to adhere to conventions.**

cs3210_s2025_pa1_heapsort.c

How to get started:

1. Get an environment running that you are comfortable in. We recommend editing on your own machine with the source on your U drive, and running through a terminal connected to pyrite.

2. Write *fill* first. It will configure your memory in a way that makes it easy to check that the other procedures work.
3. From there, implement procedure one at a time and devise methods of testing them (unit tests?). Be confident in the correctness of each procedure before attempting to call if from another procedure. You do not have the debugging tools that you're used to!
4. You have complete, unmanaged access to memory. Memory starts at address 0 and goes (by default, but adjustable via a command line switch) through byte 4095, inclusive (default 4 kB). To allocate storage, you simply use it. e.g., to access a 10 element array at address 100, simply put 100 into a register and then use that register to index your array.

Gotchas:

Be very careful about the 8s (8 byte integers!) you're going to need them all over the place. It's easy to forget them, and then things simply don't work!

What to turn in:

A single file, assignment1.legv8asm, containing your program. Don't forget the comment at the top with team members' names and NetIDs.

Using legv8emul:

Running the emulator with no parameters will give usage instructions. Code may have comments. Comments start with // and continue to the end of the line (actually, the parser never checks for the second slash, so technically they start with /).

There are three debugging pseudo-instructions available to you:

```
PRNT reg
```

will print the contents of register reg,

```
PRNL
```

will print a blank line, and

```
DUMP
```

will print a complete core dump, including all registers, and display the program code with an arrow (-->) indicating the line where the dump was produced. This arrow is not particularly useful when you explicitly DUMP, but it is useful when the emulator automatically generates a core dump for you (e.g., in the event of a crash).

There is also a HALT pseudo-instruction, which will produce a core dump and shut down the emulator.

Unlike a real computer, the emulator will start up with all registers and memory initialized to zero (except for SP and FP, which are initialized to the size of the stack).

Also unlike a real computer, the emulator will instantly crash when you attempt to access an address outside your address space. Upon crashing, the emulator will dump core with the arrow indicating the line that attempted to make the erroneous access.

Additionally--and also unlike a real computer--the emulator separates stack and "main memory" as if they are physically separate spaces; thus, it is impossible to "smash" your stack. This separation means (and is implemented by requiring) that stack is accessed exclusively through SP and FP. On a real computer, any register could be used to access the stack should one desire to flaunt convention.

The last thing that is unlike a real computer: your program is not stored in main memory! Indeed, your program's code cannot be accessed by the emulator in any way except by the part of the framework which executes it. This makes it (sadly) impossible to write self-modifying code; it also makes it impossible to accidentally overwrite your own running program. A consequence of the separated stack and program memories is that what we refer to as "main memory" begins at address zero, runs through address main-memory-size-minus-one, and is entirely yours to use as you please without worrying about corrupting anything (except your own data).

Bugs in the emulator:

Prof. Sheaffer wrote the emulator and got it tested to his satisfaction on a tight schedule. It certainly has bugs. Some of those bugs come from incomplete hardware specifications: for instance, the B.cond instructions are almost certainly not correctly implemented due to poor documentation on the condition registers (and, come to think of it, they are only implemented for SUBS and SUBIS). There are also probably implementation bugs, but none are known. Students have been using this for several years with no issues.

If you write code that crashes the emulator, please post it to the instructors on Piazza so that we can debug and fix it (not your code, but the emulator!). If you write code that you are convinced is correct and does not behave to specification, similarly post that so that we can fix the emulator.

If emulator updates occur, we will announce it to the class so that you can all move to the new version with as little disruption as possible.

Remote Server (Pyrite):

It is always possible to complete the programming assignments by working directly on pyrite.cs.iastate.edu or on any of the Linux machines in the labs. Connect to Pyrite using ssh. A nice, free Windows ssh client is PuTTY (http://www.putty.org/).

To resolve pyrite from off campus, you will need to first connect to the Iowa State VPN (https://vpn.iastate.edu/).

Editing code:

In a Linux environment, you can edit with emacs, vi, or pico (the first two are powerful and used by professional programmers all over the world, while the latter is simple and used by undergraduate computer science students with tunnel vision all over the world). In some other environment, mount your U drive, save your program there, and edit with your favorite tool, whatever it may be. You'll be able to log into pyrite in a terminal and run your program there while editing locally.

Link to the LEGv8 reference sheet: https://www.usna.edu/Users/cs/lmcdowel/courses/ic220/S20/resources/ARM-v8-Quick-Reference-Guide.pdf