You may work in teams of 1 to 4 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.   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, I wrote one. Our emulator is Linux only. I've placed a statically-linked copy on Pyrite in /home/sheaffer/legv8emul . I've also attached a copy here: legv8emul. Being statically linked, it should work on any Linux system (or, I suppose, in the Windows Linux Subsystem; if somebody knows how to use this, maybe you could post a short tutorial?).   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 and about VM access is at the bottom of this document, reproduced from my COM S 327 syllabus.

You will be implementing insertion sort in LEGv8 assembly.   In a high(er) level language, implementing the entirety of insertion sort 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 of these procedures, some with pseudocode, follow.

In the problem description below, all values are 8 bytes each. Procedures to be solved on LEGv8 assembly:

- Shift right: This procedure takes three parameters, the address of an array of ints, the index of the final element in the array, and a position in the array.   It overwrites the final element, shifting intermediate elements to the right, leaving a whole at the position.

ShiftRight(addr, pos, final):
  for i from final - 1 to pos:
    addr[i + 1] = addr[i]

- Find sorted position: This procedure takes three parameters, the address of an array of sorted ints, a value, and the index of the last element in the array. It searches the array for the sorted position of the value and returns that index.

FindSortedPos(addr, val, final):

  for i from 0 to final

    if addr[i] >= val

      break

  return i

- Insert sorted position: This procedure takes two parameters, the address of a sorted array of ints, the final element not in sorted position, and the index of the last element of the array. It moves the final element into its sorted position, shifting elements to the right as necessary such that the entire element is in sorted order and no data is lost.

InsertSortedPos(addr, final):

  v = addr[j]

  p = FindSortedPos(addr, v, final)

  ShiftRight(addr, p, final)

  addr(p) = v

- Insertion sort: This procedure takes two parameters, the address of an array of ints and the number of elements in the array. It sorts the array using the insertion sort algorithm.

InsertionSort(addr, length):

  for i from 1 to length:

    InsertSortedPos(addr, i)

- Fill: Fill(addr, length) will create an array at address addr of length elements containing length unique integers in reverse sorted order
- Main: Your main procedure takes no parameters, and doesn't strictly need to be named (its the start of your program, by default). It calls Fill to create a reverse sorted array and then calls InsertionSort to sort it.

How to get started:

1. Get an environment running that you are comfortable in.

2. Write all of the above procedures in C, or some similar language. If you do it in, e.g., Java, keep it C like (use only primitives and arrays). Make sure your code works. You'll use it to help you reason about your assembly. In a sense, you'll compile it by hand.   Please note that, while I believe it is correct, I have not tested the given pseudocode!   It is absolutely possible that it contains off-by-one or other such subtle errors!   Make your higher-level language implementation work!

3. Write Fill first. It will configure your memory in a way that makes it easy to check that other procedures work.

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, I 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 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, I never check 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).

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:

I wrote the emulator and got it tested to my 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, I only implemented them for SUBS and SUBIS). There are also probably implementation bugs, but none are known.

If you write code that crashes the emulator, please send it to me so that I can debug and fix it. If you write code that you are convinced is correct and does not behave to specification, similarly send that to me so that I can fix the emulator.

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

The rest of this document is reproduced from my 327 syllabus:

Linux VMs:

I have created two different Linux VM images. One is Slackware 14.1, which is the current version of the distribution of Linux that I run. The other is Fedora 23, which is the current version of the distribution of Linux that runs on Pyrite. I have installed the drivers that allow Linux to interface with the host operating system through the VM, so you should be able to configure shared directories, cut-and-paste between host and client, and other, similarly convenient things.

I have done a very small amount of configuration on these installations, including installing Valgrind (a memory debugging tool that you will want), Valkyrie (a GUI frontend to Valgrind), Google Chrome, and compiler updates to the current version of GCC, GCC 8.2.0.

I have created a user account, "student" with password "student" and did a small amount of configuration of that account in the .bashrc and .bash_profile files. The root (administrator) account uses the password "root". You will want to change both of these passwords with the "passwd" command! In order to change the root password, you will first need to become root with the su command. You will need to become root if you want to, say, install software or configure system services, but you should not work as root; that would be asking for trouble. On my first Linux system, back in 1995, I worked as root. My second Linux system (same hardware, fresh install) became necessary less than 2 weeks after the first.

Both images are shared in a single file on Box with the following link:

https://iastate.box.com/s/mbt9e1a8wnfmhajfxjim0p4s9xwk6bd8

This file is large, slightly over 12 GB. It is a VirtualBox file. You will need to install VirtualBox:

https://www.virtualbox.org/wiki/Downloads

and import the images (File -> Import Appliance). Once you have successfully imported, you can delete the OVA file.

Both systems are configured to use 4GB of memory. The virtual hard drives grow as needed up to a maximum of 32 GB. Much of the configuration can be changed, but only if the machine is not actually running.

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).