Computer Architecture
Instructor: Gedare Bloom
Project 1

You must do this assignment in a group of 2 or 3. You and your team members should work jointly and collaboratively on ALL parts of this assignment. You may not use code downloaded from the Internet without prior permission. Always watch the course website for updates on the assignments.

Read all instructions carefully. **Start early.**

Always watch the course website for updates on the assignments.

Your team should use a new private repository on Bitbucket with shared write-access among the members of the team. As you work, **git-add** and **git-commit** your source files to the repository. Make sure you set up your local **git-configure** with your name and email address for proper credit and attribution. Use descriptive messages for your commits. **git-push** to the repository frequently and **git-pull** before working to minimize conflicting commits. I would avoid adding any compiled/binary files to your repository, because those (1) increase the size of repositories greatly, and (2) make it much more likely to have merge conflicts that you cannot fix.

Due Sunday, September 26

It's your first day on the job at *Foo, Inc*. You are part of a design team that is creating a processor simulator for a subset of the RISC-V core RV32I Instruction set.

Group Formation

Agree to form a group with 1 to 2 of your classmates, and arrange to join the same Project1_Groups in Canvas (People → Groups). Do this ASAP.

I require that you use git source version control. We will be using Bitbucket. If you do not have an account on Bitbucket (https://bitbucket.org) you will need to create a new account. You should also create an ssh key for your account by clicking on your account icon, then go to "Manage Account" and find the "SSH Keys" menu option. Read the links on how to use and create keys, and then add a key for your computer. Don't share private ssh keys!

One member of your team should visit https://bitbucket.org/gedare/cs4200_risc-v_mas/src/main/ and create a ***private*** fork[1] of the repository. Give access to your team member(s) by adding them based on the email address linked to their Bitbucket account.

[1] Forking a repository creates a copy in your account so that you can **clone** the repository to your local machine from your account, make changes and **commit**, and **push** your changes back to the repository stored on the Bitbucket server. You may have to login first using your Bitbucket account.

You may like to use branches and pull requests to synchronize your work with each other, or you may also work directly on the same repository (e.g., the first team member's) and push directly to a branch (or even just use the main branch) there by giving write access to the other team member. Commit and push frequently as you work with your team. Take care to avoid merge conflicts if working on the same file simultaneously and independently. Pulling, committing, and pushing frequently will help, as will communicating with your team members when you work concurrently.

Part 1: The Assembler (75 points)

Your first task at *Foo, Inc.* is to complete a program known as an assembler that is written in the C language, which is named **`mas`** (short for My Assembler). The assembler has the job of converting input from an assembly source (.S) file into a binary format. In addition to converting assembly instructions to their machine instruction equivalent, the assembler is responsible for laying out the instructions correctly in the program memory and generating the correct address values for symbols (functions, branch target labels). Thankfully, management only needs the assembler to work on a single input source, which will be used to generate a complete program. (Note: You should however write your assembler as multiple source code files, which may help you with modularity and software engineering.)

The output of the assembler is a binary program file in a special format. By default the filename should be a.mxe and it should be a binary (non-ASCII) file. If an a.mxe file exists in the current directory, your assembler should replace it. The format of the a.mxe file should be as follows:

- The first 4096 bytes of the input file are the data segment of the program and should be assumed to start at address 0x10000000 (for the purposes of calculating the addresses of symbols in the data segment).
- The second 4096 bytes are the text segment, and should be assumed to start at address 0x00400000 (for the purposes of calculating the addresses of symbols in the text segment).
- There is a maximum of 4 KiB of code/data, for a maximum size program of 8 KiB.
- The first instruction located in the text segment (at address 0x00400000) will be the first instruction the program will execute.

The current state of the assembler is that it is incomplete. There is a simple parser that can read in an assembly source file and creates linked lists of assembly lines, and linked lists of tokenized strings from each line. You'll need to figure out how the parser is capturing the source input in order to determine how to generate the encodings of the program. There is also a writer module that, given two arrays of 1024 32-bit words, can produce the output of the program in the expected format. You will need to design and implement the glue to link together the parser and the writer in order to produce the correct program output for a given assembly source file. There is also a very simple disassembler in the util subdirectory that, when run on an mxe formatted file, should show the addresses and contents of the data and text sections, as well as the instruction

mnemonics for the decoded 32-bit words. (Since the .data section likely doesn't hold valid instructions, the decoding of the .data section won't really be that helpful, but the hexadecimal representation of it should be.)

The syntax and encoding for instructions is as given in the COD "Green Sheet". The instructions that the assembler must support are:

```
add, addi, and, andi, auipc, beq, bne, jal, jalr, lui, lw,
or, ori, slt, slti, sll, slli, sra, srai, srl, srli, sub,
sw, xor, xori
```

In addition, the assembler needs to support the following pseudo instructions:

```
j, la, li, mv, neg, nop, not, ret
```

The assembler implements pseudo instructions by using the base instructions. We may talk about this in class, or you may find information about how to realize these pseudo instructions in the book or online. Just remember to cite any resources you may use.

All 32 general-purpose RISC-V registers must be supported with both their Register number form (x0-x31) and with their Mnemonic names (zero, ra, sp, …).

Immediates should be supported in signed decimal, hexadecimal (Written as 0x008 for example to represent 8), and binary (Written as 0b01000 for example to represent 8).

The assembler also needs to support the following assembler directives:

- .align n
  - Aligns the next item in memory on a 2^n byte boundary. This directive can only be used within the data segment.
- .asciiz str
  - Stores the string *str* in memory with a NIL terminator. The str must be encapsulated in double quotes (non-hanging, plaintext ASCII, e.g., "Hello, world.").
- .data
  - Specifies that the following memory items should be placed in the data segment.
- .space n
  - Allocates *n* bytes in the currently active segment (i.e., data segment).
- .text
  - Specifies that the following memory items are put in the text segment. Only instructions or words (using .word) are valid to put in .text.
- .word w1, …, wn
  - Stores the *n* 32-bit word values in consecutive memory locations.

Assembler line comments are made with a # character, which causes the assembler to ignore the remainder of the line until a newline character.

Labels are indicated by ASCII containing any number of characters and digits, followed by a colon. Labels associate with the address of the next memory item in the assembly file, which may come after any number of whitespace or newline characters.

Whitespace characters may be spaces or tabs, and there may be any number of them between labels, operations, and operands. There does not need to be whitespace following commas in lists of operators or in the .word directive.

You also need to submit test cases that validate your work. You need to create one or more source files that are used as input to your assembler. Place all your test source files in the subdirectory tests of your repository, and use the file extension .S for each assembly source file. Be sure to test for correct and error cases. You may like to start with creating a few simple tests, and use a Test Driven Development approach to your project.

You will need to update the Makefile that compiles your assembler, producing an executable program named **`mas`**. Given your repository, someone should be able to run the following successfully:

```
$ make
$ ./mas tests/example1.S
```

For example if one of your tests is called example1.S then the output should be a.mxe in the current directory containing the program for this source file.

Here is an example of an assembly source file:

```
.data
myvar:   .word 5, 10,    15

.text
_start:
     la t0, myvar
     lw t1, 0(t0)
     lw t2, 4(t0)
     add x2, t1, t2
     lw t3, 8(t1)
     bne x2, t3, _start
     nop
     ret
```

Part 2: Report and Log (25 points)

Create a written report in PDF format that explains:

- Your team composition and roles of each member
- Design notes/diagrams you created.
- Any resources you used for this assignment.
- Challenges you faced and whether you overcame them.
- Documentation of functionality, bugs, etc.
- Anything else you would like to say to management.

In addition, generate a text file with the contents of your git log using the command:

```
$ git log --pretty=format:"%h %an (%ad): %s" --date=short > log.txt
```

Submit the log.txt file for 10 points.

Submission Instructions (read carefully)

Make a tar and gzip file with your group members' UCCS usernames (email address without the @uccs.edu part) as the name.

– tar -zcvf ${USERNAME1}-{USERNAME2}-project1.tgz ${USERNAME1}-{USERNAME2}-project1/

Where all your files are in the ${USERNAME1}-${USERNAME2}-project1 directory. For example, I would replace ${USERNAME1} above with gbloom, and ${USERNAME2} with the second member of my team. If your team has a third member, add their username as well. **Do not include compiled output in your submission!** Include the PDF report, your source code, and the log.txt file. Upload the tgz file to Project 1 on Canvas.