CS 210, Fall 2015
buriscv Lab
Assigned: Tuesday Nov. 5
Part 1 Due: Nov. 12, Part 2 Due: Nov. 21 @ 1:30PM

1 Introduction

You have just joined a startup that is building a new line of phones. They have choosen to build their phones around the new open source RISC-V instruction set architecture (ISA) (`riscv.org`). Their plans are to fabricate their own chips with design custom ISA extensions – new instructions and specialized addon's, to enable advance features in their software. As part of this plan they have embarked on a project to build their own software simulator on which they can try out their extensions and develop, prototype and test their software.

Your arch nemesis from high school, who as it turns out is also now a hotshot programmer, was orignially hired to write the simulator. But he got over his head and was fired. You have been brought in to pick up the pieces. Your goal is to get a set of binary images, that uses the standard RISC-V ISA, running on the simulator for a demo, in two weeks, to an important investor. If you manage to do this you will be given a full time position to lead the software division.

Good luck your fortunes and reputation are on the line.

2 Purpose

In this lab we will be putting your knowledge of how a computer works to create a software simulator of a new computer that operates as per the model we have discussed. Rather than building the computer physically, our goal will be to write a C program that, given a memory image, will operate like the computer we are trying to build.

3 How to Work on the Lab

YOU MAY WORK IN GROUPS OF TWO ON THIS ASSIGNMENT.

Download the file

`http://www.cs.bu.edu/fac/jappavoo/Resources/210/assignments/buriscv.tar`

Start by copying `handout.tar` to a protected directory in which you plan to do your work. Then give the command: `tar xvf handout.tar`. This will cause a number of files to be unpacked into a directory called handout. You will be submitting a tar file of your updated version of this directory when you are done.

Your goal will be to create a program called 'buriscv' that implements a simple emulator for a RISC-V based computer that is invoked as follows:

```
./buriscv <input.img> <output.img> [count]
```

The program reads in the file specified by the 'input.img' argument as the initial image of its memory and then executes based on that memory image. When it stops it will dump its current memory into an output file specified by the 'output.img'. Optionally you should be able to specify a count of instructions that the emulator should execute and then quit. If it is not specified then the emulator should execute until a 'SBREAK' instruction is encountered. You are given the basic infrastructure of the emulator. You will need to complete it inorder to get the provided memory image files working.

The first thing you should do is update the loop.c file with your team information. After this you should poke around the code and get a feeling for how the basic emulator infrastructure works. Your success largely depends on your ability to read, study and understand the source code you are given. The source code is larger and more complex (and thus more realistic) that other things we have looked at. You will need to develop the skill of how one reads and sorts out a larger body of source code. This is not a single pass process; you need to take notes, draw diagrams, jump around the code as you are reading, and use the debugger. You will find it helpful to talk it out on a whiteboard with your partner as you are trying to figure out how it works.

See the hints section below for an overview of the emulator and its design. Then work on getting the phases as discussed in the evaluation section completed. Note after the 'Compile' phase you will need to start using the RISC-V documentation provided in the doc directory to implement the actual functioning of the emulator.

4 Evaluation

You will be evaluated on two parts, and each part is broken down into stages. Part 1 is due first and is comprised of stages that get you up and running on some simple memory images that have been constructed from simple assembly programs that exercise basic features of the processor and simple I/O subsystem of the computer. Part 2 is composed of stages that have you execute a considerably more complex unknown memory image. The makefile has a target for each stage that will test your emulator to see if it passes the stage. You can use these make targets to see if you have gotten things right. For each target you will either see it print out PASS or FAIL.

The stages are described below:

Part 1

1. **Compiling (10 Credits): 'make compile'**: Get the emulator to compile without any warnings. In other words when you run make, you should obtain an executable file named buriscv.
2. **Memory Copy (20 Credits): 'make memcpy')**: Get your emulator to correctly execute the memcpy.img file provided. You will find in the apps directory test memory images along with the 'unknown' image (unknown.img) that you are ultimately trying to get working. One of the test images is a simple image that contains a program that copies a string in memory from one location to another. The buriscv source assembly used to create this image is in apps/memcpy.S. The tool chain (compiler, assembler and linker) required to create the images are installed on the csa systems. The make files contain the commands to use these tools to recreate the images. You are free to create you own images to help test and develop your simulator. To get the memcpy image working you will need to get your main loop working and then implement the necessary functionality for the instructions used by the memcpy.s program. To test the emulator on the memcpy.img file by hand, use the following command:

   ```
   ./buriscv apps/memcpy.img out.img
   ```

   To poke around the image files, use the provided **./img -h** image command.
3. **Console Copy (20 Credits): 'make concpy')**: Get your emulator to correctly execute the concpy.img file provided. To get this image file working you will need to complete hooking up the IO support in the emulator. The emulator has built-in infrastructure for a simple console device (console.[ch]) that allows characters to be read in from the keyboard and characters to be written to the screen. To accomplish this, the emulator supports the idea of memory mapped devices (mem.[ch]). Where we read or write a certain range of the simulated memory will cause operations on the IO devices. To get the image file working, you need to complete the **console_init** routine (which main invokes) to correctly map the console to the right simulated address.
4. **'C' code (30 Credits): make 'func')**: Get your emulator to correctly execute the func.img file provided. To get this one working you will need to implement the instructions that are necessary to get basic 'C' working – procedures.
5. **printf (20 Credits): make 'printf')**: As we have learnt, printf is a very useful thing to have working. Your job here is to get a basic hello world image functioning.

Part 2

1. **Unknown 1 (100 Credits): 'make unknown1')**: You are now ready to start working on the unknown image. The first test will send the image a simple command to see that you have been able to get the image started up. You may find it useful to use the provided **./img** command to dump parts of the image file. In particular you can use it to dump the reset vector to determine where in the image file the opcodes are located. You can then dump the opcodes using the same command. You may also find it useful to use one of the online 6502 disassemblers to get a dump of the actual 6502 assembly code that the unknown image file contains. GOOD LUCK AND HAVE FUN!
2. **Unknown 2 (BONUS Credits): 'make unknown2')**: Congratulations! If you have gotten this far you now know what the program in the unknown image is. The unknown2 target will test more of its functionality thus requiring you to implement more instructions in the emulator to pass this stage.

5 Handin Instructions

You will do two handins for this lab. Part 1 will be due on Nov 12 and Part 2 on Nov 21. In each case, do the following:

> **Create a tar file of your emulator directory called 6502.tar and then use gsubmit to submit your solution. If you have any questions about how to create the tar file speak with the TF.**

Before submitting, ensure that your solution behaves as expected on *csa2.bu.edu* as this is where we will grade your solution.

6 Hints

In this section we provide you with some background about the emulator and some general hints for this assignment. But perhaps the most important of all is that this assignment requires you to explore the problem using the knowledge you have learnt in this class. There is no prescribed cookie-cutter solution; you must be creative and inquisitive.

6.1 Emulator

An emulator/simulator is a program that is designed to mimic the behavior of an entire computer system. In our case, we are trying to construct a program that emulates a simple computer system that is based on a cpu that conforms to the RISC-V ISA specification. This ISA was developed by researches at Berkley as a modern open source ISA. Along with the CPU, our emulator needs to implement the memory contents and I/O capabilities of the simple computer. Memory can be easily emulated by an array that is indexed by the simulated addresses. I/O devices are generally more complicated to emulate, but we take a very simple approach to I/O that is discussed in the next sub-subsection.

In header files (*.h) you will find the definition of the emulator's state. It defines a machine structure that contains the memory array along with the CPU state. The CPU state is define as a structure that contains the internal state used to implement the functioning of the CPU along with the registers that the RISC-V programs can manipulate through the instructions. The RISC-V ISA is very flexible and supports 32, 64 and 128 bit versions. We will be creating a 64 bit version (XLEN=64).

The basic idea of the emulator is to implement the CPU's behavior by implementing a loop that fetches, decodes, and executes instructions from the simulated memory. Each instruction should create a change in the simulated machine state, either the simulated memory or simulated CPU state. You will need to complete the definition of this `inst_loop` in instruction.c. To do this you will find that the functions that implement fetch, decode and execute already exist but you must structure their invocation in `inst_loop`. While these functions exist, the actual implementation of the instructions are not complete and the majority of your work will be to correctly implement the instructions that are needed by the memory images.

The `inst_loop` function is invoked by main. In main, the processor state and memory is initialized so that the simulated riscv core can begin execution at the point of reset. As per the manual, when the riscv is powered on, it follows a reset sequence that causes it to initialize the Program Counter (pc) register to the an address that is system specific. In our case the reset vector is 0x200.

The logic invoked in main prior to inst_loop will load the input memory image in (as specified as the first argument to the program) and loads the initialize the pc by simulating a reset. From that point on, the loop function controls the operation of the emulator. You may use the ./img command to dump the value of a memory image's reset.

6.1.1 Toolchain

On the csa* machines you will find a custom build of gcc and associated tools (objdump, objcopy, ...) that can compile code for the simulator. The make files in the apps directory use this tools. The are installed in /research/sesa/210/buriscv/multilib. You are free to play with them. The most important to you will be objdump. As it can be used to disassemble the elf files for the tests applications. See example below. However, you can also you the tools to compile your own c code for the emulator. The easiest way to go about this is to copy one of the app directories and then make modifications to the copy. If you are instrested in help on how to do this please see the TF or instructor.

```
csa3> /research/sesa/210/buriscv/multilib/bin/riscv64-unknown-elf-objdump -d apps/m

apps/memcpy/memcpy512K-64.elf:     file format elf64-littleriscv


Disassembly of section RAM:

0000000000000000 <_ram_start>:
...

0000000000000100 <_vector_start>:
     100: 00100073           ebreak
...

0000000000000140 <_vector_supervisor>:
     140: 00100073           ebreak
...

0000000000000180 <_vector_hypervisor>:
     180: 00100073           ebreak
...

00000000000001c0 <_vector_machine>:
     1c0: 00100073           ebreak
...
```

```
00000000000001fc <_vector_nmi>:
     1fc: 00100073           ebreak

0000000000000200 <_vector_reset>:
     200: 00080137           lui sp,0x80
     204: 00010113           mv sp,sp
     208: fe010113           addi sp,sp,-32 # 7ffe0 <_gp+0x6f710>
     20c: 00012e23           sw zero,28(sp)
     210: 00012c23           sw zero,24(sp)
     214: 00012a23           sw zero,20(sp)
     218: 00012823           sw zero,16(sp)
     21c: 00012623           sw zero,12(sp)
     220: 00012423           sw zero,8(sp)
     224: 00012223           sw zero,4(sp)
     228: 00012023           sw zero,0(sp)
     22c: 00010413           mv s0,sp
     230: 5d10f0ef           jal 10000 <_ftext>
...

0000000000010000 <_ftext>:
   10000: 000107b7           lui a5,0x10
   10004: 03078793           addi a5,a5,48 # 10030 <_fdata>
   10008: 000105b7           lui a1,0x10
   1000c: 07f58593           addi a1,a1,127 # 1007f <dst>

0000000000010010 <loop>:
   10010: 0007c683           lbu a3,0(a5)
   10014: 00d58023           sb a3,0(a1)
   10018: 00178793           addi a5,a5,1
   1001c: 00158593           addi a1,a1,1
   10020: fe0698e3           bnez a3,10010 <loop>
   10024: 00100073           ebreak

0000000000010028 <_etext>:
...

0000000000010030 <_fdata>:
   10030: 6548                 sd a0,136(a0)
   10032: 6c6c                 sd a1,216(s0)
   10034: 6f57206f           j 82f28 <_ram_end+0x2f28>
   10038: 6c72                 bnez s0,10114 <_bss_start+0x44>
   1003a: 2164                 fsd s1,192(a0)
   1003c: 2121                 fsd s0,128(sp)
   1003e: 000a                 j 10040 <_fdata+0x10>
```

```
...

000000000001007f <dst>:
...

00000000000100cf <_edata>:
...

00000000000100d0 <_bss_start>:
...

00000000000108d0 <_gp>:
...
csa3>
```

6.1.2 IO

6.2 GDB

As always, the debugger is your friend! However, things are a little more subtle when working on this project. Remember that GDB is used to debug the actual x86 RISC-V emulation program not the RISC-V software running in the emulator. That being said, clearly the data structures and state of the emulator can be used to trace and debug the RISC-V software. In this kind of situation, it is typical to use the programmability of GDB to define new commands that make your life easier. Specifically, GDB automatically looks for a file named `.gdbinit` in the directory you are in when executing GDB. In this file, you can define your own commands that then become available for your use within GDB. We have provided an example of such a file that has many useful commands. For instance, we can define a command `sinfo` that prints the contents of the RISC-V registers in a nicely formatted way including dumping the sr bits in binary and disassembling the current instruction that the pc is pointing to. We encourage you to look at this file and use these commands. Here is some info to get you started.

6.2.1 .gdbinit file

In addition to looking at the contents of the .gdbinit, you can issue `help user` to see a list of all the user define commands we have provided. `help <cmd>` can be used to print a documentation that we have provided for a particular command `<cmd>`.

Some commands of particular interest are `strace, sstep, sinfo, sdisassemble, sxbyte, sxshort`.
In general, you will want to start GDB and then issue `strace` in order to use these commands. This will set a default breakpoint in the right spot to allow you to single step 6502 instruction execution via the `sstep` command. The following is an example of a typical GDB session:

```
add stuff here
```

6.3 Additional Info

- Study machine.h, loop.c, and instruction.[ch]; you will be spending a lot of time on them.
- You will want to figure out how to use the types correctly.
- You may find it useful to run the emulator using the images by hand so that you can interact with it. You can do this by running the emulator directly. For example to directly run the emulator with the unknown image you would issue the following command:

  ```
  ./buriscv apps/unknown.img myout.img 2> mytrace
  ```

  This will run your emulatorin the current directory on the unknown image file in the apps directory such that the final output memory image will be placed in a file called myout.img in the current directory and any output written by the emulator to standard error will be sent to the file mytrace in the current directory. If you are unsure about what mytrace is and how to use it ask the TF.
- **Work incrementally.**
- Get comfortable locating and interpreting the information in the manuals.
- We have provided an `img` command that can be used to manipulate and dump the image files. You can use this to poke around the image files. Issue './img -h' to learn how to use it.