

CS 210, Summer 2019

buriscv Lab

1 Introduction

You have just joined a startup that is building a new line of phones. They have chosen to build their phones around the new open source RISC-V instruction set architecture (ISA) (<http://riscv.org>). Their plans are to fabricate their own chips with custom ISA extensions – new instructions and specialized add-on's – to enable advanced 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. They feel it is critical that they develop their own simulator to ensure that they fully understand the code and can quickly change it to explore the new features they are considering putting in their hardware.

Your arch nemesis from high school, who as it turns out is also now a hotshot programmer, was originally hired to write the simulator. But he got in over his head and was fired. You have been brought in to pick up the pieces. Your goal is to get a set of standard RISC-V ISA binary images running on the simulator. A demo has been scheduled with an important investor who is expecting to see a working simulator. If you manage to pull this demo off, you will be promoted to the prestigious position of lead software engineer, given a considerably higher salary and access to a company car.

Your fame and fortune are on the line. Good Luck!

2 Purpose

In this lab we will be putting your knowledge of computer mechanics to the tests by developing a software simulator of a new computer system. Rather than building the computer physically, our target is to create a complete C program that simulates the new computer. Much of code for the simulator has already been written, your job is to complete it. This assignment gives the experience of working with an existing code base, and an open-source ISA. By walking-through the code of the simulator and reading through RISC-V ISA manual, you will discover enough mechanics of the simulator to complete the development.

3 How to Work on the Lab

YOU MAY WORK IN GROUPS OF TWO ON THIS ASSIGNMENT.

Download the file buriscvlab.tgz from the piazza 210 site

<https://piazza.com/bu/spring2019/cs210/resources>

Start by copying buriscvlab.tgz to a protected directory in which you plan to do your work. Then give the command: `tar -zxvf buriscvlab.tgz`. This will cause a number of files to be unpacked into a directory called buriscvlab. 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 'buriscv64' that implements a simple simulation of a RISC-V based computer. The simulator is invoked as follows:

```
./buriscv64 <input.img> <output.img> [count]
```

The simulator program reads in the file specified by the 'input.img' argument and sets this image as the initial state of simulated memory. The simulator then executes based on the instructions and configuration of that base memory image. When the simulation is complete, it will dump final state of the simulated memory into an output file specified by the 'output.img'. Optionally, you can specify a maximum count of instructions the simulator should execute before quitting. If it is not specified then the simulator will execute until a 'SBREAK' instruction is encountered. You will start with a basic infrastructure for the simulator. You will need to complete it. The correctness and completeness of your simulator will be verified by simulating the execution of a set of RISC-V memory image files. The first thing you should do is update team.c and attempt to compile and build the simulator using the make (<http://www.gnu.org/software/make/manual/make.html>) command:

```
$ ls
buriscvlab.md5sum  buriscvlab.tgz
$ tar -zxf buriscvlab.tgz
$ ls
buriscvlab  buriscvlab.md5sum  buriscvlab.tgz
$ cd buriscvlab/
$ cat team.c
#include "team.h"

/*****
 * NOTE TO STUDENTS: Before you do anything else, please
 * provide your team information in the following struct.
 *****/
team_t team = {
    /* Team name */
    "",
    /* First member's full name */
    "",
    /* First member's email address */
```

```

    "",
    /* Second member's full name (leave blank if none) */
    "",
    /* Second member's email address (leave blank if none) */
    ""};
$ emacs -nw team.c # edit file filling in your team info
$ cat team.c
#include "team.h"

/*****
 * NOTE TO STUDENTS: Before you do anything else, please
 * provide your team information in the following struct.
 *****/
team_t team = {
    /* Team name */
    "TheDoctor",
    /* First member's full name */
    "Jonathan Appavoo",
    /* First member's email address */
    "jappavoo@bu.edu",
    /* Second member's full name (leave blank if none) */
    "",
    /* Second member's email address (leave blank if none) */
    ""};
$

```

Next try building the simulator by running make:

```

$ make
gcc -O1 -Wall -g -D IMP -c -DXLEN64 -MMD -MP -o mem.o64 mem.c
gcc -O1 -Wall -g -D IMP -c -DXLEN64 -MMD -MP -o instruction.o64 instruction.c
instruction.c: In function inst_loop:
instruction.c:1590:5: error: unknown type name ADD
    ADD CODE HERE;
    ^
instruction.c:1590:14: error: expected '=', ',', ';', asm or __attribute__ before HERE
    ADD CODE HERE;
    ^
instruction.c: At top level:
instruction.c:1235:16: warning: inst_decodeAndExecute
defined but not used [-Wunused-function]
    static INST_RC inst_decodeAndExecute(struct Machine* m) {
    ^
make: *** [instruction.o64] Error

```

As shown above, the initial build will fail as the simulator is not complete enough to function. Your job is to progressively complete the simulator until it can pass all the required tests. Each phase grows in complexity, requiring the capabilities of your simulation to be expanded.

Here is a basic outline of how you should proceed

1. **FIRST, READ THIS HANDOUT COMPLETELY!**
2. Poke around the code and get a basic feeling for how the simulator is designed to work. Understand the basic data structures that make up the simulator, and observe the way the RISC-V instructions are intended to be executed (see example instruction section ??). Next, skim over "The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.0" manual in the doc directory.
3. Fix and complete the function `inst_loop` in the file `instruction.c`. See the section ?? for a discussion of what this means. At this point your simulator should compile but it will not be functional enough to simulate the execution of a program.
4. When your changes to `inst_loop` are correct the simulator will read in and attempt to execute the past in image file. However, none of the functions that implement the working RISC-V instructions set have been implemented. The simulator exits when it attempted to execute an unimplemented instruction. A message written to the simulator trace file (`*.trc`) will identify the exact instruction that caused the simulation to fail. Your job is to implement the instructions as-needed to run each of the tests. The development will be incremental, as you only need to implement the instructions that the simulation requires (i.e., the instructions that cause the simulation to quit.) The first test image is `memcpy` (see later in this writeup for more information on the tests). Below is what you will see the first time you try and run the `memcpy` test after correctly fixing the `inst_loop` function.

```
$ make memcpy64
...
make: *** [memcpy64] Error 134
$ cat memcpy-64.trc
inst_AUIPC: NYI
buriscv64: instruction.c:131: inst_AUIPC: Assertion `0' failed.
$
```

5. At this point you need to find the unimplemented function named `inst_AUIPC` in the `instruction.c` file (line 131). Each of the instruction functions in `instruction.c` are intended to implement the simulated execution behaviour of the associated RISC-V instruction; In this case, we will need to implement the `AUIPC` instruction (see page 14 of the ISA manual). Once you implemented this instruction, rerun the test and see if the simulator encounters another unimplemented instruction (odds are, it will!) Your goal is to implement each instruction function successively until the simulation completes without any failing. If your system runs correctly when you run the test (i.e., your instructions are implemented correctly) you will see:

```
MEMCOPY64: PASS
```

Note you can also run the simulator directly instead of using the `make` command eg.:

```
$ ./buriscv64 apps/memcpy/memcpy512K-64.img out.img
inst_AUIPC: NYI
buriscv64: instruction.c:131: inst_AUIPC: Assertion `0' failed.
Aborted
```

4 Simulator Overview

The simulator source is composed of several parts, organized into files and subdirectories. The source code for this simulator itself is in the main directory:

```
$ ls *.*[ch]
buriscv.c          common.h          instruction.c     main.c          riscv.h         tty.c
buriscv.h          console.c        instruction.h     mem.c          team.c          tty.h
buriscv_defines.h console.h        interrupt.c      mem.h          team.h         ttymodes.c
buriscv_info.h    disassemble.c   interrupt.h      misc.c         test.c         ttymodes.h
buriscv_types.h   disassemble.h   machine.h        misc.h         test.h
```

There are three subdirectories: `doc`, `writup`, `apps`. The `doc` directory contains a copy of the RISC-V documentation that you will need to understand the behaviour of each instruction. You are also encouraged to poke around the RISC-V website and consult any additional documentation you can find. The `writup` directory contains a copy of this writup pdf. The `apps` directory contains several subdirectories, one for each RISC-V binary image that has been built for the simulation tests:

```
apps/memcpy apps/concpy apps/jalret apps/func
apps/printf apps/ptypes apps/unknown
```

Within each of these you find a `.img` file (eg. `apps/memcpy/memcpy512K-64.img` that is a exact copy of the initial simulated memory to execute the particular RISC-V program. We built each of these using a RISC-V cross tool-chain (compiler, assembler, linker, etc) which is installed in `/research/sesa/210/buriscv/multilib/bin`. You are free to use these to rebuild the images or construct your own test images (section ??). The first four apps; `memcpy`, `concpy`, `jalret` and `func` form the required portion of the assignment the others are bonus. Once you have fixed the `inst_loop` function you can try and run any of the images as follows: `./buriscv apps/<app>/<imagefile>.img <output image file>`. The simulator will load the specified image file as the starting contents of its memory and then simulator a processor reset and start executing instructions in contain in the memory image. If the simulator exits correctly, when it encounters an `SBREAK` instruction it will dump the current contents of its simulated memory to the output image file specified.

4.1 Simulator Structure

You should familiarize your self with the basic design and intended operation of the simulator. The following gives you a basic overview. You are encouraged to poke around the code and draw pictures.

The first thing you want to do is understand the core data structures of the simulator and how the simulator starts up. Look at `machine.h` to get started understanding the data structures. In particular look at the `struct Machine` an instance of this structure encapsulates the simulation state. Perhaps most importantly it contains the registers and memory state for the simulation. You will find that the actual definitions of the sub types are spread across the various other `.h` files. You will find that there are various functions defined for accessing and updating the data structures. Eg. There are functions for getting and setting the registers and there are functions for reading and writing memory. **YOU SHOULD USE THESE FUNCTIONS AND NOT DIRECTLY ACCESS THE DATASTRUCTURES.**

In `main.c` you will find the definition of the main function. Reading it will give you an overview of how the simulator starts up and the basic flow. The core functionality – the simulation loop is invoked near the end of `main` and it is called `inst_loop`.

`inst_loop` is within the `instruction.c` file. This implements that heart of the simulator and is where you will spend most of your time and effort. The `inst_loop` function should successively fetch, decode and execute instructions. Prior to call this function the main function will have ensured that the initial value of the PC register is initialized to a well known start location as specified by the RISC-V manual. You must complete the `inst_loop` so that it conducts the necessary steps. There are functions (which are complete and correct) for each of these steps: 1) `inst_fetch` – using the PC's current value reads from the simulated memory the instruction opcode to be executed next into the IR (instruction register). All RISC-V instructions are encoded in a single 32bit value. After the `inst_fetch` instruction is executed the IR register will have the 32 bit instruction value in it that needs to be decoded and executed. This work is conducted by the `inst_decodeAndExecute` function. This function decodes the opcode using a large switch statement that was generated from the manual. Each case that corresponds to a particular RISC-V valid encoding for an instruction will result in corresponding invocation to a instruction specific function (eg. `inst_AUIPC`). These instructions, however, are not complete.

Hint: Your first goal is to add the invocation of `fetch` and `decodeAndExecute` to the loop function. The `fetch` function will return a value greater than 0 if it was successful. You should record this value in the `rc` local variable. If `rc` is negative after the call to `fetch` you skip the `decodeAndExecute` invocation. Otherwise you should invoke `decodeAndExecute` again recording its return value in `rc`. The tests at the end of the while loop examine `rc` to determine if the simulation should exit or continue. So to fix the `inst_loop` function you should only need to add two lines!

4.2 Instructions

Most of your work will be in completing the instruction functions for the RISC-V instructions that are encountered when you try and simulate the RISC-V application images provided. To do this, you will need to understand the behaviour of each instruction, as defined in the RISC-V manual. Read very carefully, and use the functionality build into the simulator to your advantage (the simulator provides many "helper" functions you should use!)

The RISC-V instruction set is broken down into different subsets; we will be implementing instructions from the RV32I, RV64I and RV32M subsets. Our goal is a basic 64 bit RISC-V simulator where all our registers are represented as 64 bits values.

NOTE: BE AWARE THAT SOME OF THE RV32I DESCRIPTIONS INCORRECTLY REFER TO

THE VALUES BEING 32 BIT, IT REALLY SHOULD SAY REGISTER LENGTH (64, IN OUR CASE).

The simulator has code for decoding an immediate value from an instruction encoding, and functions for sign and zero extending different sized values. These supplementary functions are critical to get the instruction functions working quickly and correctly. The following provide you with working examples for several instructions so that you can get a sense of how to proceed.

4.2.1 inst_AUIPC

```
static INST_RC inst_AUIPC(struct Machine* m) {
    struct Core* core = &(m->state.cpu.core);
    struct RegisterFile* rf = &(core->regFile);
    Instruction ir;
    ir.raw = irGet(core);

    uint8_t rd = ir.U.rd;
    Register imm = decodeImmediate(ir, U);

    Register newpc = pcGet(rf) + imm;
    setGpr(rf, rd, newpc);
    return INST_PC_UNSET;
}
```

To get this instruction working you will want to replace Not Yet Implemented (NYI) code with the above code, and compare what it is doing with respect to the manual entry on the AUIPC. As expected, each instruction function is passed a pointer to the Machine structure (m) for the simulated machine. Using this pointer, it then declares some local variables to the subparts of the machine state it needs (a pointer to the core and the core's register file). It also declares a local ir variable that is of Instruction type. It loads this variable with the current instruction encoding. Using this variable it then determines the destination register and the immediate value that is specified in the encode instruction.

There are several ways of encoding a RISC-V instruction. These various encodings are reflected in the simulator through the Instruction type and the decodeImmediate function. All the instruction functions already pull out the values you need to implement the core logic of the instruction. We encourage you to look at how the Instruction type works and how the decodeImmediate function has been implemented. You can rely on them working correctly.

In this case, all we need to do to fetch the value of the PC from the register file is to use the pcGet(rf) call. We place this value into the local variable, newpc. To this we add the value of the immediate that was pulled from the encoding of this instruction call. We then update the destination General Purpose Register via the call setGpr(rf, rd, newpc). Finally we return a return-value back to the decodeAndExecute function that indicates we did not set the PC to a new value and that it should go ahead and advance the PC to the next sequentially ordered instruction (this function is already implemented in (BROKEN: LOOK UP FILENAME)).

It is critical that you look at `buriscv_types.h` to understand the basic types that are used to represent the various values that you will be manipulating.

4.2.2 `inst_JAL`

```
static INST_RC inst_JAL(struct Machine* m) {
    struct Core* core = &(m->state.cpu.core);
    struct RegisterFile* rf = &(core->regFile);
    Instruction ir;
    ir.raw = irGet(core);

    uint8_t rd = ir.J.rd;
    Register imm = decodeImmediate(ir, J);

    Register oldpc = pcGet(rf);
    Register newpc = oldpc + imm;
    Register link = oldpc + (Register)sizeof(Instruction);

    setGpr(rf, rd, link);
    pcSet(rf, newpc);
    return INST_PC_SET;
}
```

The above is a working implementation of the JAL instruction. This instruction updates the pc to the sum of the current value of the pc plus the encoded immediate. It also stores the next instruction that would have been executed into the destination register specified. Since every instruction is encoded in a fixed size encoding this is simply the current value of the PC plus sizeof an Instruction. After computing these values in local variables, the function updates the registers via calls to `setGpr(rf, rd, link)` and `pcSet(rf, newpc)`. Critically, this function returns back `INST_PC_SET` to the `decodeAndExecute` function, indicating that it should not touch the PC as it already has been set inside the instruction function.

4.2.3 `inst_BLT` and `inst_BLTU`

```
static INST_RC inst_BLT(struct Machine* m) {
    struct Core* core = &(m->state.cpu.core);
    struct RegisterFile* rf = &(core->regFile);
    Instruction ir;
    ir.raw = irGet(core);

    uint8_t rs1 = ir.B.rs1;
    uint8_t rs2 = ir.B.rs2;
    Register imm = decodeImmediate(ir, B);
```

```

SRegister rs1Val = getGpr(rf, rs1);
SRegister rs2Val = getGpr(rf, rs2);
Register newpc = pcGet(rf) + imm;

if (rs1Val < rs2Val) {
    pcSet(rf, newpc);
    return INST_PC_SET;
}
return INST_PC_UNSET;
}

```

The above is a functional implementation of the Branch Less Than (BLT). It is critical to note how it is careful to use the types to ensure that the comparison is conducted as a signed two's complement comparison. The Register type is a 64bit unsigned value, and the SRegister type is a signed 64bit value. To ensure the less-than comparison is done correctly, the local temporaries `rs1Val` and `rs2Val` are declared as SRegister.

Compare the implementation above carefully to the implementation of the `inst_BLTU` instruction below.

```

static INST_RC inst_BLTU(struct Machine* m) {
    struct Core* core = &(m->state.cpu.core);
    struct RegisterFile* rf = &(core->regFile);
    Instruction ir;
    ir.raw = irGet(core);

    uint8_t rs1 = ir.B.rs1;
    uint8_t rs2 = ir.B.rs2;
    Register imm = decodeImmediate(ir, B);

    Register rs1Val = getGpr(rf, rs1);
    Register rs2Val = getGpr(rf, rs2);
    Register newpc = pcGet(rf) + imm;

    if (rs1Val < rs2Val) {
        pcSet(rf, newpc);
        return INST_PC_SET;
    }
    return INST_PC_UNSET;
}

```

4.2.4 `inst_BL` and `inst_SB`

```

static INST_RC inst_LB(struct Machine* m) {
    struct Core* core = &(m->state.cpu.core);

```

```

struct RegisterFile* rf = &(core->regFile);
Instruction ir;
ir.raw = irGet(core);

uint8_t rd = ir.I.rd;
uint8_t rs1 = ir.I.rs1;
Register imm = decodeImmediate(ir, I);

Register rs1Val = getGpr(rf, rs1);
Address eaddr = rs1Val + imm;

Byte b = mem_getByte(m, eaddr);
Register value;
signExtendToRegister(b, value);
setGpr(rf, rd, value);
return INST_PC_UNSET;
}

```

Another important class of instructions are those that load and store values to and from memory. The above is a implementation of Byte Load (BL) instruction. There are several predefined functions for loading and storing various sized value to and from the simulated memory. In the above code we see the use of `mem_getByte` to get the byte-sized value from the target effective address. Read over the `mem.h` file to see what other functions exist to read and write to memory..

It is critical to note the use of `signExtendToRegister`. Carefully read the manual description of Byte Load to understand why this is. To this point, be careful when implementing LBU, as you do not want to sign extend, but rather zero extend. These points maybe subtle, but they are crucial to your simulation running correctly. Read the manual slowly and carefully.

Below is a function implementation of `inst_SB`

```

static INST_RC inst_SB(struct Machine* m) {
    struct Core* core = &(m->state.cpu.core);
    struct RegisterFile* rf = &(core->regFile);
    Instruction ir;
    ir.raw = irGet(core);

    uint8_t rs1 = ir.S.rs1;
    uint8_t rs2 = ir.S.rs2;
    Register imm = decodeImmediate(ir, S);

    Register rs1Val = getGpr(rf, rs1);
    Address eaddr = rs1Val + imm;

    uint8_t value = getGpr(rf, rs2);
}

```

```

    mem_putByte(m, eaddr, value);
    return INST_PC_UNSET;
}

```

4.2.5 inst_ADDIW

Perhaps the trickiest instructions to implement will be those in the RV64I section. They will test your knowledge of C types. Read the manual carefully and think about what you need to do. Remember that the RISC-V rule is to keep all 32-bit values as sign-extended 64 bit values in its registers. Computations should first be done in 32bits to ensure correct overflow and truncation behavior. Here is a implementation of `inst_ADDIW` to get you started.

```

static INST_RC inst_ADDIW(struct Machine* m) {
    struct Core* core = &(m->state.cpu.core);
    struct RegisterFile* rf = &(core->regFile);
    Instruction ir;
    ir.raw = irGet(core);

    uint8_t rd = ir.I.rd;
    uint8_t rs1 = ir.I.rs1;
    Register imm = decodeImmediate(ir, I);

    Register rs1Val = getGpr(rf, rs1);
    uint32_t v32 = (uint32_t)(rs1Val + imm);
    Register value;
    signExtendToRegister(v32, value);

    setGpr(rf, rd, value);
    return INST_PC_UNSET;
}

```

4.3 Input/Output

An important part of any computer is how it conducts Input and Output. The concept test will require you to understand how our simulator works in this regard. The test will read bytes from the input of simulator and write it back to the output.

The simulator adopts a memory mapped I/O model in which address after physical memory can be used to read and write I/O devices.

You will need to study and fix the console device code to get this working. The necessary change to the simulator is simple, but the real challenge is to understand how the simulator is intended to work, and where to do the fix.

5 Evaluation

Your grade will be determined by your simulator's operation on the tests that make up the required portion. The bonus stages are optional, and will earn you additional bonus points. To get credit for completing a stage your simulator must PASS the test for the stage. The stages are described below:

The stages are described below:

Required

1. **Compiling (10 Credits): 'make compile'**: Get the simulator to compile without any warnings. In other words when you run make, you should obtain an executable file named buriscv64.
2. **Memory Copy (20 Credits): 'make memcpy64'**: Get your simulator to correctly execute the memcpy.img file provided. This is a simple test that copies a string from one area in memory 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. To test the simulator on the memcpy.img file by hand, use the following command:

```
/buriscv64 apps/memcpy/memcpy512K-64.img out.img
```

To poke around the image files, use the provided **.img -h** image command. To see an exact disassembly use:

```
$ /research/sesa/210/buriscv/multilib/bin/riscv64-unknown-elf-objdump  
-d apps/memcpy/memcpy512K-64.elf
```

3. **Console Copy (20 Credits): 'make concpy64'**: Get your simulator to correctly execute the concpy.img file provided. This test copies bytes from the input to your program to the output. To get this image file working you will need to complete hooking up the IO support in the simulator. The simulator 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 simulator 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. **jalret (20 Credits): 'make jalret64'**: This function ensures that the basic instructions to implement 'C' functions work correctly.
5. **'C' code (30 Credits): make 'func64'**: Unlike the previous tests this application was created with the 'C' compiler and exercises the basics of running 'C' code on your simulator.

To run all the required tests you can use the command `make required`

Bonus Several bonus tests are included. If you will get additional points for each bonus test you can pass.

1. **Full C code: 'make bonus1'**: You are now ready to try something a more complicated – a real hello world C binary that includes a full C runtime startup.

2. **Types: 'make bonus2')** This tests does several signed and unsigned operations on variables of different and sizes.
3. **unknown 1: 'make bonus 3')** For this tests you are given only the binary image. If your simulator is working correctly you will be able to determine what exactly the program is.
4. **unknown 2: 'make bonus 4')** In this final test you your simulator will be required to correctly run the binary from bonus 3 with a more rigorous test.

6 Handin Instructions

Create a tar file of your simulator directory called buriscvlab.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.

7 Hints

7.0.1 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 simulator. 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/memcpy/memcpy512K-64.elf
```

```
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>
...

00000000000010000 <_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>

00000000000010010 <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>

```

7.1 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 simulator program not the RISC-V software running in the simulator. That being said, clearly the data structures and state of the simulator 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 allows you to load a file in our case named `buriscv.gdb` in the directory you are in when executing GDB. To do this start `gdb` and invoked the `'source buriscv.gdb'` command. In this file, are custom `gdb` users commands that then become available for your use within GDB. We have provided many useful commands that can help you debug your simulator. The TF will cover the use of some of these commands in your discussion section.