CSCI 463 Assignment 5 – RISC-V Simulator

30 Points

Abstract

In this assignment, you will extend the functionality of your RISC-V disassembler to also simulate the execution of RV32I instructions.

This is the third of a multi-part assignment creating computing machine capable of executing real programs compiled with gcc. The purpose is to gain an understanding of a machine and its instruction set while exercising your programming skills.

1 Problem Description

Load a binary file into a simulated memory of sufficient size and then decode and execute each 32-bit instruction one-at-a-time starting from address zero and continuing until an `ebreak` instruction is encountered, an instruction-count limit is reached, or an illegal instruction has been encountered.

2 Files You Must Write

You will write a C++ program suitable for execution on `hopper.cs.niu.edu` (or `turing.cs.niu.edu`.)

Your source files *MUST* be named exactly as shown below or they will fail to compile and you will receive zero points for this assignment.

Create a directory named `a5` and place within it a copy of all the the source files from assignment 4 and add the additional files discussed below.

- `hex.h` (see assignment 4.)
- `hex.cpp` (see assignment 4.)
- `memory.h` (see assignment 4.)
- `memory.cpp` (see assignment 4.)
- `rv32i_decode.h` (see assignment 4.)
- `rv32i_decode.cpp` (see assignment 4.)
- `rv32i_hart.cpp` The definition of the class `rv32i_hart`.
- `rv32i_hart.h` The definitions of member functions of class `rv32i_hart`.
- `registerfile.h` The definition of the `registerfile` class will go here.
- `registerfile.cpp` The `registerfile` class member function definitions.
- `cpu_single_hart.h` The definition of the class `cpu_single_hart`.
- `cpu_single_hart.cpp` The `cpu_single_hart` class member function definitions.
- `main.cpp` Your `main()` and `usage()` function definitions.

Provided that no mistakes are present in the files for Assignment 4 then no changes to those files are necessary.

2.1 registerfile.h and registerfile.cpp

The purpose of this class is to store the state of the general-purpose registers of one RISC-V *hart*.[1]

Recall that a RISC-V hart has 32 registers and that every one is identical except for register `x0`.

Register `x0` will always contain the value zero when ever it is read and it will never store anything that is written into it (such data is simply ignored/discarded.)

Implement `registerfile` with a private vector of `int32_t` elements (one for each register) and a constructor that uses the `reset()` method to initialize register `x0` to zero, and all other registers to `0xf0f0f0f0`.

It must provide the following member functions:

- `void reset();`

  Initialize register `x0` to zero, and all other registers to `0xf0f0f0f0`.

- `void set(uint32_t r, int32_t val);`

  Assign register `r` the given `val`. If `r` is zero then do nothing.

- `int32_t get(uint32_t r) const;`

  Return the value of register `r`. If `r` is zero then return zero.

- `void dump(const std::string &hdr) const;`

  Implement a dump of the registers. The `hdr` parameter is a string that must be printed at the begining of the output lines. For example, if called as `dump("")` then the output must be formatted precisely as:

  ```
   x0 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
   x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  ```

  if called as `dump("HEADER-")` then the output must be formatted precisely as:

  ```
  HEADER- x0 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  HEADER- x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  HEADER-x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  HEADER-x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  ```

  Note the space-gap on the first two lines.

  Inherit the `hex` class and use its `hex32()` utility function to simplify printing the register values!

2.2 rv32i_hart.h and rv32i_hart.cpp

Define `rv32i_hart` as a subclass of `rv32i_decode` to represent the execution unit of a RV32I hart as seen in Figure 1

Implement a member function named `exec` (using a similar design as that used in `rv32i_decode::decode`) to simulate the execution of RV32I instructions and helper methods for each instruction with names like `exec_lui` and `exec_jalr` to perform the simulated execution.

[1] The term *hart* means "hardware thread." As part of the simple CPU you are creating for this assignment, this term is the same as what is often referred to as a *core*.

~/NIU/courses/463/2022-fa/assignments/a5/handout.tex

jwinans@niu.edu 2022-10-26 15:45:40 -0500 v2.1-95-g034c48a0

```
class rv32i_hart : public rv32i_decode
{
public:
    rv32i_hart(memory &m) : mem(m) { }
    void set_show_instructions(bool b) { show_instructions = b; }
    void set_show_registers(bool b) { show_registers = b; }
    bool is_halted() const { return halt; }
    const std::string &get_halt_reason() const { return halt_reason; }
    uint64_t get_insn_counter() const { return insn_counter; }
    void set_mhartid(int i) { mhartid = i; }

    void tick(const std::string &hdr="");
    void dump(const std::string &hdr="") const;
    void reset();

private:
    static constexpr int instruction_width          = 35;
    void exec(uint32_t insn, std::ostream*);
    void exec_illegal_insn(uint32_t insn, std::ostream*);
  ...

    bool halt = { false };
    std::string halt_reason = { "none" };
  ...
    uint64_t insn_counter = { 0 };
    uint32_t pc = { 0 };
    uint32_t mhartid = { 0 };

protected:
    registerfile regs;
    memory &mem;
};
```

Figure 1: `rv32i_hart()`

2.2.1 `rv32i_hart` Public Member Functions

- `rv32i_hart(memory &m);`

  The constructor must initialize `mem` as shown in Figure 1 (because the `mem` member variable is a reference.)

- `void set_show_instructions(bool b);`

  Mutator for `show_instructions`. When `true`, show each instruction that is executed with a comment displaying the register values used (as seen in Figure 8.)

- `void set_show_registers(bool b);`

  Mutator for `show_registers`. When `true`, dump the registers before instruction is executed.

- `bool is_halted() const;`

  Accessor for `halt`. Return `true` if the hart has been halted for any reason.

- `const std::string &get_halt_reason() const;`

  Return a string indicating the reason the hart has been halted. Values returned are one of the following:

  – `"none"`

  – `"EBREAK instruction"`

  – `"ECALL instruction"`

  – `"Illegal CSR in CSRRS instruction"`

  - "Illegal instruction"
  - "PC alignment error"

- `void reset();`

  Reset the `rv32i` object and the `registerfile`.

  To reset a hart:

  - Set the `pc` register to zero.
  - Call `regs.reset()` to reset the register values.
  - Set the `insn_counter` to zero.
  - Set the the `halt` flag to false.
  - Set the the `halt_reason` to `"none"`.

- `void dump(const std::string &hdr="") const;`

  Dump the entire state of the hart. Prefix each line printed by the given `hdr` string (the default being to not print any prefix.) It will dump the GP-regs (making use of the `regs` member variable by calling `regs.dump(hdr)`) and then add a dump of the `PC` register in the following format:

```
 x0 00000000 f0f0f0f0 00001000 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 pc 00000000
```

  If the `hdr` string is set to `"[XYZ] "` then the output would look like:

```
[XYZ]  x0 00000000 f0f0f0f0 00001000 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
[XYZ]  x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
[XYZ] x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
[XYZ] x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
[XYZ]  pc 00000000
```

- `uint64_t get_insn_counter() const;`

  Accessor for `insn_counter`. Return the number of instructions that have been executed by the simulator since the last `reset()`.

- `void set_mhartid(int i);`

  Mutator for `mhartid`. This is used to set the ID value to be returned by the `csrrs` instruction for CSR register number `0xf14`. (*This will always be zero on processors that only have a single-hart.*)

- `void tick(const std::string &hdr="");`

  The `tick()` method function is how to tell the simulator to execute and instruction. The `hdr` parameter is required to be printed on the left of any and all output that is displayed as a result of calling this method.

  If the hart is halted then return immediately without doing anything. Otherwise, simulate the execution of one single instruction:

  - If `show_registers` is true then dump the state of the hart with the given `hdr`.
  - If the `pc` register is *not* a multiple of 4 then set the `halt` flag to true, the `halt_reason` to `"PC alignment error"`, and return without further processing.
  - Increment the `insn_counter` variable (not the `pc` register.)

  - Fetch an instruction from the memory at the address in the `pc` register.
  - If `show_instructions` is true then
    * Print the `hdr`, the `pc` register (in hex), and the 32-bit fetched instruction (in hex).
    * Call `exec(insn, &std::cout)` to execute the instruction and render the instruction and simulation details.
  - else
    * Call `exec(insn, nullptr)` to execute the instruction *without* rendering anything.

Note that the `reset()` and `tick()` methods are the *only* way to change the state of the simulated hart hardware. (Which is similar to but not to be confused with changing the state of the C++ `rv32i_hart` object! For example, the notion of calling `set_show_instructions()` can change the state of the `rv32i_hart` object. But it does *not* change the state of the simulated hart hardware.)

2.2.2 `rv32i_hart` Private Member Functions

- **`void exec(uint32_t insn, std::ostream*);`**

  This function will execute the given RV32I instruction by making use of the `get_xxx()` methods to extract the needed instruction fields to decode the instruction and invoke the associated `exec_xxx()` helper function by using the same sort of `switch`-logic from assignment 4. See Figure 2.

  This function must be capable of handling any 32-bit `insn` value. If an illegal instruction is encountered then call an `exec_illegal_insn()` method to take care of the situation.

```
void rv32i_hart::exec(uint32_t insn, std::ostream* pos)
{
    ...

    switch(opcode)
    {
    default:                exec_illegal_insn(insn, pos); return;
    case opcode_lui:        exec_lui(insn, pos); return;
    case opcode_auipc:      exec_auipc(insn, pos); return;

    ...
    }
}
```

Figure 2: Implementing `exec()`

- **`void exec_illegal_insn(uint32_t insn, std::ostream* pos);`**

  Set the `halt` flag and, if the `ostream*` parameter is not `nullptr` then use `render_illegal_insn()` to render the proper error message by writing it to the `pos` output stream. See Figure 3.

```
void rv32i_hart::exec_illegal_insn(uint32_t insn, std::ostream* pos)
{
    if (pos)
        *pos << render_illegal_insn(insn);
    halt = true;
    halt_reason = "Illegal instruction";
}
```

Figure 3: `exec_illegal_insn()`

- `void exec_xxx(uint32_t insn, std::ostream*);`

  Your `exec_xxx()` helper functions perform a similar role as the `render_xxx()` helpers. However, the `exec` helpers will simulate the execution of an instruction.

  Each `exec` helper function must simulate the execution of an instruction and, optionally, render the details of what it is simulating.

  The rendering of the simulation details for each instruction can be seen in Figure 8.

  Use the `render_xxx()` helpers from assignment 4 to render the decoded instructions when needed by the `exec_xxx()` helpers.

  To align the comment column when adding the simulation details to those instructions that have them, consider using the `std::setw()` I/O manipulator to add padding on the right as seen in Figure 4

  Note that the simulation-description comments are modeled on the way that the operations are described in the "Detailed Description" column of the reference card at the end of RVALP. *Note that for sake of space, the incrementing of the* `pc` *register is not shown by this simulator except in the branch and jump instructions, where the updating of the* `pc` *register is a significant aspect of the instruction.*

  When rendering the `exec` operations comment, the data values displayed are those of the registers, fields, or data involved in the instruction. When combined with the hart dumps before and after each instruction execution, they provide everything necessary to verify that an instruction has been implemented properly.

  See Figure 8 for examples of the comment format of each type of instruction.

  Your output must precisely match the reference output or it will be ungradable and you will receive a zero for the output portion of your grade.

  The correct value for the `instruction_width` constant is 35;

  See Figure 4.

```
void rv32i_hart::exec_slt(uint32_t insn, std::ostream* pos)
{
    uint32_t rd = get_rd(insn);
    uint32_t rs1 = get_rs1(insn);
    uint32_t rs2 = get_rs2(insn);

    int32_t val = (regs.get(rs1) < regs.get(rs2)) ? 1 : 0;

    if (pos)
    {
        std::string s = render_rtype(insn, "slt    ");
        *pos << std::setw(instruction_width) << std::setfill(' ') << std::left << s;
        *pos << "// " << render_reg(rd) << " = (" << hex::to_hex0x32(regs.get(rs1)) << " < " << hex::
             to_hex0x32(regs.get(rs2)) << ") ? 1 : 0 = " << hex::to_hex0x32(val);
    }
    regs.set(rd, val);
    pc += 4;
}
```

Figure 4: `exec_slt()`

2.2.3 `rv32i_hart` Protected Member Variables

- `registerfile regs;`

  The GP-regs (general purpose registers) for your simulation.

- `memory &mem;`

  This will contain a reference to the `memory` object from assignment 3. It will be used by the disassembler and execution logic to fetch the instructions and to read/write data in the load and store instructions.

2.2.4 `rv32i_hart` Private Member Variables

- `bool halt;`

  A flag to stop the hart from executing instructions. Set it any time that the execution should halt and use it in `tick()` to prevent further instructions from executing until/unless `reset()` is invoked.

- `std::string halt_reason;`

  If `halt` is set to true, also set this to contain a string describing the reason for the halt. Initialize to `"none"` if `reset()` is called.

- `uint32_t mhartid;`

  This contains the CSR register value to return by a `csrrs` instruction that reads register `0xf14`. Set the default value for this to zero. (*In this assignment, this default value will never change.*)

- `bool show_instructions;`

  A flag with a default value of false. When true, print each instruction when simulating its execution.

- `bool show_registers;`

  A flag with a default value of false. When true, print a dump of the hart state (by calling `dump()`) before executing each instruction.

- `uint64_t insn_counter;`

  This will count the number of instructions that have been executed. Initialize to zero and if/when `reset()` is called.

  Use this to count the number of instructions executed.

- `uint32_t pc;`

  Use this to contain the address of the instruction being decoded/disassembled. When decoding instructions that refer to the `pc` register to calculate a target address (e.g. `auipc`, `jal`, and branch instructions) use this value to determine the instruction's memory address.

  Initialize to zero and if/when `reset()` is called.

2.3 `cpu_single_hart.h` and `cpu_single_hart.cpp`

This is a subclass of `rv32i_hart` that is used to represent a CPU with a single hart.

2.3.1 `cpu_single_hart` Public Member Functions

- `cpu_single_hart(memory &mem) : rv32i_hart(mem) {}`

  Implement this constructor as shown above in order to pass the memory class instance to the constructor in the base class.

- `void run(uint64_t exec_limit);`

  Since code that executes on this simulator has no (practical) way to determine how much memory the machine has, set register `x2` to the memory size (get it with `mem.get_size()`) before executing any instructions in your `run()` method. Note that the number of bytes in the memory is also the address of the first byte past the end of the simulated memory.[2]

If the `exec_limit` parameter is zero, call `tick()` in a loop until the `is_halted()` returns `true`.

If the `exec_limit` parameter is not zero then enter a loop that will call `tick()` until `is_halted()` returns `true` or `exec_limit` number of instructions have been executed.

If the hart becomes halted then print a message indicating why by using `get_halt_reason()` to get the reason message.

Regardless of why the execution has terminated, print the number of instructions that have been executed by using `get_insn_counter()`.

For example running the simulator with an execution limit of 2, dumps enabled by the `-ir` command-line options, and simulating the `allinsns5.bin` example program will result in the output shown in Figure 5.

```
winans@x570:~$ ./rv32i -m100 -irl2 allinsns5.bin
 x0 00000000 f0f0f0f0 00000100 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 pc 00000000
00000000: abcde237  lui     x4,0xabcde                 // x4 = 0xabcde000
 x0 00000000 f0f0f0f0 00000100 f0f0f0f0  abcde000 f0f0f0f0 f0f0f0f0 f0f0f0f0
 x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 pc 00000004
00000004: abcde217  auipc   x4,0xabcde                 // x4 = 0x00000004 + 0xabcde000 = 0xabcde004
2 instructions executed
```

Figure 5: Example output from running: `rv32i -m100 -irl2 allinsns5.bin`

2.4 `main.cpp`

Provide a `main()` function so that it accepts the command-line parameters (and reflect them in a proper Usage statement) as discussed below. See the example logic in the `main()` from the last assignment and the on-line manual for getopt(3) for details on how to use it to parse the arguments.

The command line arguments you must provide are:

- `[-d]`

  Show a disassembly of the entire memory before program simulation begins. By default, do not disassemble the program memory.

- `[-i]`

  Show instruction printing during execution. By default, do not print instructions during execution.

- `[-l execution-limit]`

  Specifies the maximum limit of instructions to execute. If set to zero then there is no limit (run forever.) By default there is no limit.

[2] By convention, `x2` is used as the program's full-descending stack pointer. Setting it to the address of the first *non-existent* memory address is suitable for allocating the top range of memory addresses to a call-stack used to hold the program's activation records.

- `[-m hex-mem-size]`

  Specifies the size of the simulated memory. By default the size must be `0x100`.

- `[-r]`

  Show a dump of the hart (GP-registers and `pc`) status before each instruction is simulated.

- `[-z]`

  Show a dump of the hart status and memory after the simulation has halted.

- The last argument is the name of the binary file to load into the memory before the simulation begins.

Keep in mind that any of the command-line arguments may appear in any order and both on their own:

```
-d -i -r -z -l 1234 -m efc0
```

as well as in groups or stuck together:

```
-dirz -l1234 -mefc0
```

The getopt(3) function can deal with these situations. Make sure that your solution does too.

If any command-line arguments are invalid then your `usage()` function must print an appropriate error and Usage messages and terminate the program in the traditional manner. (See https://en.wikipedia.org/wiki/Usage_message and Figure 6.)

```
winans@x570:~$ ./rv32i -X allinsns5.bin
./rv32i: invalid option -- 'X'
Usage: rv32i [-d] [-i] [-r] [-z] [-l exec-limit] [-m hex-mem-size] infile
    -d show disassembly before program execution
    -i show instruction printing during execution
    -l maximum number of instructions to exec
    -m specify memory size (default = 0x100)
    -r show register printing during execution
    -z show a dump of the regs & memory after simulation
```

Figure 6: Example output from running: `rv32i -X allinsns5.bin`

3 Input

You will be provided with multiple executable test programs and the command-line arguments used to run them on the course web site.

4 Output

Your program will be tested with a combination of the command-line arguments and runs with dumps and traces of instructions executed will be `diff`'d against the output from a reference implementation.

Your program must precisely match the reference output to be considered perfect.

~/NIU/courses/463/2022-fa/assignments/a5/handout.tex

jwinans@niu.edu 2022-10-26 15:45:40 -0500 v2.1-95-g034c48a0

5 How To Hand In Your Program

When you are ready to turn in your assignment, make sure that the only files in your a5 directory is/are the source files defined and discussed above. Then, in the parent of your a5 directory, use the mailprog.463 command to send the contents of the files in your a5 project directory in the same manner as we have used in the past.

6 Grading

The grade you receive on this programming assignment will be scored according to the syllabus and its ability to compile and execute on the Computer Science Department's computer.

It is your responsibility to test your program thoroughly.

When we grade your assignment, we will compile it on `hopper.cs.niu.edu` using these exact commands:

```
g++ -g -Wall -Werror -std=c++14 -c -o main.o main.cpp
g++ -g -Wall -Werror -std=c++14 -c -o rv32i_decode.o rv32i_decode.cpp
g++ -g -Wall -Werror -std=c++14 -c -o memory.o memory.cpp
g++ -g -Wall -Werror -std=c++14 -c -o hex.o hex.cpp
g++ -g -Wall -Werror -std=c++14 -c -o registerfile.o registerfile.cpp
g++ -g -Wall -Werror -std=c++14 -c -o rv32i_hart.o rv32i_hart.cpp
g++ -g -Wall -Werror -std=c++14 -c -o cpu_single_hart.o cpu_single_hart.cpp
g++ -g -Wall -Werror -std=c++14 -o rv32i main.o rv32i_decode.o memory.o hex.o registerfile.o rv32i_hart.o cpu_single_hart.o
```

Your program will then be run multiple times using different memory sizes, test data files, and command line options.

7 Hints

As always, build up a solution one step at a time. Some times you can start with what you already have and build upon it. Other times you must create something new and (should) unit test it before trying to integrate it with the rest of your code.

- Start by updating `main.cpp` to accept the new command line options and test it by printing out their values. Then add the conditional call to `disassemble(mem)` and test that your new `-d` command line option works.
- After the disassembly, construct and reset() your CPU like this:

  ```
  cpu_single_hart cpu(mem);
  cpu.reset();
  ```

- Stub in `void dump(const std::string &hdr="") const;` that prints only a message to let you know it has been called. Then add conditional logic to call it and `mem.dump()` based on the `-z` command line argument and test it.
- Add the flags to the `rv32i_hart` class and set them based on the associated `-i` and `-r` command line options.
- Stub in the `rv32i_hart::exec()` method that treats every instruction as illegal and use it to develop and debug your `rv32i_hart::tick()` and `cpu_single_hart::run()` methods. (If you can not execute *one illegal* instruction and halt the simulation then you can't possibly expect anything else to work.)

If your simulation ends with an `ebreak` instruction, then the application program your simulator is running will have terminated gracefully. If so then your `cpu_single_hart::run()` method loop should end and print the message:

```
Execution terminated. Reason: EBREAK instruction
```

The other reasons for halting the simulation should print similar messages.

Regardless of why the simulated application has ended, print out the instruction counter as seen at the end of Figure 8.

Note that it should be trivial to create a file for testing your logic for handing illegal instructions… just leave one of the actual instructions unimplemented in `rv32i_hart::exec()` and see that it is treated accordingly. Alternately, you can also run the simulator on just about any random (preferably small or even empty) file and odds are that it will include illegal instruction values. (No, you will not be given a test file with illegal instructions for testing this specific feature. It is your job to think creatively to solve this sort of problem.)

- Write the `registerfile` class add add it as a member to the `rv32i_hart` class. You should then be able to finish your `rv32i_hart::dump()` method so that it prints out the GP-regs and the `pc` register as seen in Figure 5.

- Finish any remaining work left undone in your in your `rv32i_hart::tick()` method so that it properly calls `rv32i_hart::exec()` as discussed in section 2.2.1.

- Use the big `switch` statement from the `rv32i_decode::decode()` as a template structure for your `rv32i_hart::exec()` method. The first instruction you should implement should be `ebreak` (so the test programs can stop your simulator) as seen in Figure 7.

```
void rv32i_hart::exec_ebreak(uint32_t insn, std::ostream* pos)
{
    if (pos)
    {
        std::string s = render_ebreak(insn);
        *pos << std::setw(instruction_width) << std::setfill(' ') << std::left << s;
        *pos << "// HALT";
    }
    halt = true;
    halt_reason = "EBREAK instruction";
}
```

Figure 7: Executing the `ebreak` instruction.

Note that `rv32i_hart::exec()` is called differently than `rv32i_decode::decode()` in that it is void and takes different arguments. Re-using the already-debugged switch structure from assignment 4 should work well, but keep in mind that the cases may have to return differently as seen in Figure 2.

- At this point, add one instruction at-a-time comparing your output against the reference files.

  Since `ebreak` is a bit simplistic (and a special case that is almost identical to the way you should implement the illegal instruction method), a close look at a possible implementation logic of a more typical instruction is shown in Figure 4.

  The `slt` instruction is described in the reference card at the end of RVALP as:

  `rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4`

  Therefore the instruction and simulation details will be rendered as shown in Figure 8 and can be summarized as:

  `slt x4,x14,x15 // x4 = (0xf0f0f0f0 < 0xf0f0f0f0) ? 1 : 0 = 0x00000000`

The `= 0x00000000` at the right end of the above simulation detail comment represents the value that is assigned to `x4`. In other words, it is the final value of the expression:

```
(0xf0f0f0f0 < 0xf0f0f0f0) ? 1 : 0
```

Consider what happens when the the instruction: `slt x4,x4,x15` is simulated. In order to be able to render the simulation summary comment that shows the values of all the registers involved before and after the instruction simulation, it will be necessary to extract the associated register values before `val` in Figure 4 is calculated and it must *not* be stored into the `rd` register `x4` until *after* the simulation comment has been printed.

The code in Figure 4 addresses this problem by its use of the `val` variable as a holder for the calculated result of the instruction. Then it prints the simulation comment (if needed). Finally, it stores the result into the `rd` register using `regs.set(rd, val)` and increments the `pc` register.

The same problem occurs with the `pc` register in the jump and branch instructions. Always be careful that your renderings are of the correct (before/after) values of any registers involved.

```
winans@x570:~$ ./rv32i -i allinsns5.bin
00000000: abcde237  lui     x4,0xabcde                // x4 = 0xabcde000
00000004: abcde217  auipc   x4,0xabcde                // x4 = 0x00000004 + 0xabcde000 = 0xabcde004
00000008: 008000ef  jal     x1,0x00000010             // x1 = 0x0000000c,  pc = 0x00000008 + 0x00000008 = 0x00000010
00000010: 01008267  jalr    x4,16(x1)                 // x4 = 0x00000014,  pc = (0x00000010 + 0x0000000c) & 0xfffffffe = 0x0000001c
0000001c: feb59ce3  bne     x11,x11,0x00000014        // pc += (0xf0f0f0f0 != 0xf0f0f0f0 ? 0xfffffff8 : 4) = 0x00000020
00000020: fe004ae3  blt     x0,x0,0x00000014          // pc += (0x00000000 < 0x00000000 ? 0xfffffff4 : 4) = 0x00000024
00000024: fe0558e3  bge     x10,x0,0x00000014         // pc += (0xf0f0f0f0 >= 0x00000000 ? 0xfffffff0 : 4) = 0x00000028
00000028: fe0066e3  bltu    x0,x0,0x00000014          // pc += (0x00000000 <U 0x00000000 ? 0xffffffec : 4) = 0x0000002c
0000002c: fea074e3  bgeu    x0,x10,0x00000014         // pc += (0x00000000 >=U 0xf0f0f0f0 ? 0xffffffe8 : 4) = 0x00000030
00000030: 00000463  beq     x0,x0,0x00000038          // pc += (0x00000000 == 0x00000000 ? 0x00000008 : 4) = 0x00000038
00000038: 00b01463  bne     x0,x11,0x00000040         // pc += (0x00000000 != 0xf0f0f0f0 ? 0x00000008 : 4) = 0x00000040
00000040: 00054463  blt     x10,x0,0x00000048         // pc += (0xf0f0f0f0 < 0x00000000 ? 0x00000008 : 4) = 0x00000048
00000048: 00005463  bge     x0,x0,0x00000050          // pc += (0x00000000 >= 0x00000000 ? 0x00000008 : 4) = 0x00000050
00000050: 00a06463  bltu    x0,x10,0x00000058         // pc += (0x00000000 <U 0xf0f0f0f0 ? 0x00000008 : 4) = 0x00000058
00000058: 00007463  bgeu    x0,x0,0x00000060          // pc += (0x00000000 >=U 0x00000000 ? 0x00000008 : 4) = 0x00000060
00000060: 01000313  addi    x6,x0,16                  // x6 = 0x00000000 + 0x00000010 = 0x00000010
00000064: 01034203  lbu     x4,16(x6)                 // x4 = zx(m8(0x00000010 + 0x00000010)) = 0x000000e3
00000068: 00134203  lbu     x4,1(x6)                  // x4 = zx(m8(0x00000010 + 0x00000001)) = 0x00000082
0000006c: 01035203  lhu     x4,16(x6)                 // x4 = zx(m16(0x00000010 + 0x00000010)) = 0x00004ae3
00000070: 00a35203  lhu     x4,10(x6)                 // x4 = zx(m16(0x00000010 + 0x0000000a)) = 0x0000feb0
00000074: 01030203  lb      x4,16(x6)                 // x4 = sx(m8(0x00000010 + 0x00000010)) = 0xffffffe3
00000078: 01130203  lb      x4,17(x6)                 // x4 = sx(m8(0x00000010 + 0x00000011)) = 0x0000004a
0000007c: 01031203  lh      x4,16(x6)                 // x4 = sx(m16(0x00000010 + 0x00000010)) = 0x00004ae3
00000080: 00a31203  lh      x4,10(x6)                 // x4 = sx(m16(0x00000010 + 0x0000000a)) = 0xfffffeb0
00000084: 01032203  lw      x4,16(x6)                 // x4 = sx(m32(0x00000010 + 0x00000010)) = 0xfe004ae3
00000088: fff00293  addi    x5,x0,-1                  // x5 = 0x00000000 + 0xffffffff = 0xffffffff
0000008c: 0e500ea3  sb      x5,253(x0)                // m8(0x00000000 + 0x000000fd) = 0x000000ff
00000090: 0e501823  sh      x5,240(x0)                // m16(0x00000000 + 0x000000f0) = 0x0000ffff
00000094: 0e502a23  sw      x5,244(x0)                // m32(0x00000000 + 0x000000f4) = 0xffffffff
00000098: 4d260213  addi    x4,x12,1234               // x4 = 0xf0f0f0f0 + 0x000004d2 = 0xf0f0f5c2
0000009c: 4d262213  slti    x4,x12,1234               // x4 = (0xf0f0f0f0 < 1234) ? 1 : 0 = 0x00000001
000000a0: 4d263213  sltiu   x4,x12,1234               // x4 = (0xf0f0f0f0 <U 1234) ? 1 : 0 = 0x00000000
000000a4: 4d264213  xori    x4,x12,1234               // x4 = 0xf0f0f0f0 ^ 0x000004d2 = 0xf0f0f422
000000a8: 4d266213  ori     x4,x12,1234               // x4 = 0xf0f0f0f0 | 0x000004d2 = 0xf0f0f4f2
000000ac: 4d267213  andi    x4,x12,1234               // x4 = 0xf0f0f0f0 & 0x000004d2 = 0x000000d0
000000b0: 00c69213  slli    x4,x13,12                 // x4 = 0xf0f0f0f0 << 12 = 0x0f0f0000
000000b4: 00c6d213  srli    x4,x13,12                 // x4 = 0xf0f0f0f0 >> 12 = 0x000f0f0f
000000b8: 40c6d213  srai    x4,x13,12                 // x4 = 0xf0f0f0f0 >> 12 = 0xffff0f0f
000000bc: 00f70233  add     x4,x14,x15                // x4 = 0xf0f0f0f0 + 0xf0f0f0f0 = 0xe1e1e1e0
000000c0: 40f70233  sub     x4,x14,x15                // x4 = 0xf0f0f0f0 - 0xf0f0f0f0 = 0x00000000
000000c4: 00f711b3  sll     x3,x14,x15                // x3 = 0xf0f0f0f0 << 16 = 0xf0f00000
000000c8: 00f72233  slt     x4,x14,x15                // x4 = (0xf0f0f0f0 < 0xf0f0f0f0) ? 1 : 0 = 0x00000000
000000cc: 00f73233  sltu    x4,x14,x15                // x4 = (0xf0f0f0f0 <U 0xf0f0f0f0) ? 1 : 0 = 0x00000000
000000d0: 00f74233  xor     x4,x14,x15                // x4 = 0xf0f0f0f0 ^ 0xf0f0f0f0 = 0x00000000
000000d4: 00f751b3  srl     x3,x14,x15                // x3 = 0xf0f0f0f0 >> 16 = 0x0000f0f0
000000d8: 40f751b3  sra     x3,x14,x15                // x3 = 0xf0f0f0f0 >> 16 = 0xfffff0f0
000000dc: 00f76233  or      x4,x14,x15                // x4 = 0xf0f0f0f0 | 0xf0f0f0f0 = 0xf0f0f0f0
000000e0: 00f77233  and     x4,x14,x15                // x4 = 0xf0f0f0f0 & 0xf0f0f0f0 = 0xf0f0f0f0
000000e4: f14022f3  csrrs   x5,0xf14,x0               // x5 = 0
000000e8: 00100073  ebreak                            // HALT
Execution terminated. Reason: EBREAK instruction
50 instructions executed
```

Figure 8: `exec()` Instruction operation comment format.