CSCI 463 Assignment 5 – RISC-V Simulator

30 Points – Due Thursday, April 16, 2020 at 23:59

Abstract
In this assignment, you will extend the functionality of your RISC-V disassembler by using it to load and execute a binary file.

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

1 Problem Description

Execute binary file by loading it 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 instruction-count limit is reached or an \texttt{ebreak} instruction is encountered.

2 Files You Must Write

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

Your source files \textit{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 \texttt{a5} and place within it a copy of all the the source files from assignment 4 as discussed below.

- \texttt{main.cpp} Your \texttt{main()} and \texttt{usage()} function definitions. (See Figure 1.)
- \texttt{hex.h} The declarations of your hex formatting functions (copied from assignment 4.)
- \texttt{hex.cpp} The definitions of your hex formatting functions (copied from assignment 4.)
- \texttt{memory.h} The definition of your \texttt{memory} class (copied from assignment 4.)
- \texttt{memory.cpp} The \texttt{memory} class member function definitions (copied from assignment 4.)
- \texttt{rv32i.h} The definition of the \texttt{rv32i} class (copied from assignment 4.)
- \texttt{rv32i.cpp} The definitions of member functions of class \texttt{rv32i} (copied from assignment 4.)
- \texttt{registerfile.h} The definition of the \texttt{registerfile} class will go here.
- \texttt{registerfile.cpp} The \texttt{registerfile} class member function definitions will go here.

2.1 \texttt{main.cpp}

You must provide a \texttt{main()} function that is implemented as shown in Figure 1 (plus appropriate documentation.)

Note that an additional command line argument has been added. See section 3 for details.

Your \texttt{usage()} function must print an appropriate error message and terminate the program in the traditional manner. (See \url{https://en.wikipedia.org/wiki/Usage_message})
```c
int main(int argc, char **argv)
{
    if (argc != 4)
        usage();

    memory mem(stoul(argv[1], 0, 16));
    if (!mem.load_file(argv[3]))
        usage();

    rv32i sim(&mem);
    sim.run(stoul(argv[2]));
    mem.dump();

    return 0;
}
```

![Figure 1: Example main() function.](image)

### 2.2 hex.h and hex.cpp

See assignment 3 handout.

### 2.3 memory.h and memory.cpp

See assignment 3 handout.

### 2.4 registerfile.h and registerfile.cpp

The purpose of this class is to store the state of the CPU registers.

Recall that the RISC-V CPU 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 written into it (such data is simply ignored/discarded.)

Implement registerfile with a private array of 32 `int32_t` elements (one for each register), a constructor that initializes register x0 to zero, and all other registers to 0xf0f0f0f0.

It must provide the following member functions:

- **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;**
  Implement a dump of the registers with the following format:

    x0 00000000 f0f0f0f0 00002000 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

  Use your `hex32()` utility function to simplify printing the register values!
2.5  rv32i.h and rv32i.cpp

See also the assignment 4 handout.

Assignment 4 included member functions to extract the instruction fields and decode instructions. In this assignment, copy and use the switch statement(s) from decode() as the starting point for exec() and make the necessary alterations to execute the instructions.

Implement a helper method for each instruction with names like exec_lui() and exec_jalr().

2.5.1  rv32i Member Functions

Your rv32i must include all the member functions from assignment 4 plus the following:

- void dump() const;
  As a member of the rv32i class, this dump method will dump the state of the CPU. It will dump the GP-regs (by making use of registerfile::dump()) and then add a dump of the PC register (as it is part of the rv32i class) in the following format:

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

- void run(uint32_t limit);
  Update the run() method to include an argument that specifies the limit on the number of instruction executions. (This will be VERY handy when debugging your project!)

  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.\footnote{By convention, x2 is used as the program’s full-descending stack pointer. Setting it to the address of the first non-existant memory address is suitable for allocating the top range of memory addresses to a stack that can be used to hold the program’s activation records.} In the dump above it can be observed that x2 is set to 0x00001000.

  Once x2 is set, enter a loop that will run until an ebreak instruction is encountered (or limit number of instructions have been executed) that will perform the following operations:

  1. Dump the CPU state.
  2. Fetch the instruction from the address in the PC register.
  3. Print the instruction address.
  4. Decode print the instruction (including space padding on the right using std::setw() and std::setfill() to align the execution comment string.)
  5. Print a comment spearator: “//”.
  6. Execute the instruction and print the operational details (returned in the string from exec().)

  When the run-loop has completed, print a message indicating how many instructions have been executed, a final dump of the CPU state, and return.

  For example, running your simulator with an execution limit of 2 and using allinsns.bin from your last assignment will result in the output shown in Figure 2.
CSCI 463 Assignment 5 – RISC-V Simulator

std::string exec(uint32_t insn);

Your exec() function will execute the given RISC-V insn by making use of the get_xxx() methods to extract the needed instruction fields and the current machine state in your memory, registerfile, and rv32i objects.

This function must be capable of handling any legal 32-bit instruction value. If an illegal instruction is encountered, simulation must halt.

See Figure 4 for details on how to halt execution when an illegal instruction is encountered.

The returned std::string must contain a comment that matches the operations described in the “Detailed Description” in the reference card at the end of RVALP. Note that for sake of space, the incrementing of the pc register is not shown except for the branch and jump instructions, where the updating 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 CPU dump before and after each instruction execution, it should provide everything necessary to verify that the instruction is implemented properly.

See Figure 5 for examples of comments for each type of instruction.

2.5.2 rv32i Member Variables

• registerfile regs;

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

Figure 2: Example output from running: ./rv32i a0 2 allinsns.bin

std::string exec(uint32_t insn);

Your exec() function will execute the given RISC-V insn by making use of the get_xxx() methods to extract the needed instruction fields and the current machine state in your memory, registerfile, and rv32i objects.

This function must be capable of handling any legal 32-bit instruction value. If an illegal instruction is encountered, simulation must halt.

See Figure 4 for details on how to halt execution when an illegal instruction is encountered.

The returned std::string must contain a comment that matches the operations described in the “Detailed Description” in the reference card at the end of RVALP. Note that for sake of space, the incrementing of the pc register is not shown except for the branch and jump instructions, where the updating 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 CPU dump before and after each instruction execution, it should provide everything necessary to verify that the instruction is implemented properly.

See Figure 5 for examples of comments for each type of instruction.

2.5.2 rv32i Member Variables

• registerfile regs;

The GP-regs (general purpose registers) for your simulation.
• bool halt = { false };
  A flag to use to stop your instruction execution. Set it any time the execution should halt and use it
  in your run() loop as one of the conditions to stop executing instructions.
  See the ebreak() instruction example and Figure 4.

• uint64_t insn_counter = { 0 };
  Use this to count the number of instructions executed. It is printed after the run() loop has completed
  executing instructions.
  Use it in your run() loop as one of the conditions to stop executing instructions.

• memory * mem;
  This will contain a pointer 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.

• uint32_t pc;
  Use this to contain the address of the instruction being decoded/disassembled. When decoding in-
  structures 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 pc to zero.

3 Input

Your program will accept three arguments on the command line as shown in the main() code snippet in
Figure 1.

• The first argument is a hex number representing the amount of memory to simulate.
• The second argument is a decimal number indicating the maximum number of instructions to execute.
• The third argument is the name of a file to load into the simulated memory. In this assignment, this
  will be the name of a binary rv32i executable program.

You will be provided with these executable test programs:

• allinsns5.bin
  This includes at least one of each instruction. It is short and suitable for testing instructions imple-
  mented one at-a-time. Run it like this:

        ./rv32i 200 50 allinsns5.bin > allinsns5.log

• torture5.bin
  This test program executes a true and false case of each conditional branch and a number of other test
  instructions with parameters to verify that each instruction has been implemented properly. Run it like
  this:

        ./rv32i 9000 250 torture5.bin > torture5.log
• sieve.bin

This is a C++ program that uses the set template class, sprintf(), iterators and other features to implement a sieve of Eratosthenes to generate a list of prime numbers from 2 to 1000. As can be seen below, it will output over 7,000,000 lines of output and takes 20+ seconds to run. If it runs properly, the tail and head commands will discard all of the output but a portion of the memory dump that includes the char output[1000][16] array of prime numbers that it formats in ASCII:

```
./rv32i 20000 2000000 sieve.bin | tail -n +7360402 | head -n 170 > sieve.log
```

4 Output

This program’s output will be a trace of instructions executed, starting at address zero and a memory dump after its termination.

See Figure 2 for an example of the full output of a program terminating after reaching a limit of two instructions.

See Figure 5 for examples of an execution operation comment for every instruction.

Your program match the reference output.

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 to your TA like this:

```
mailprog.463 a5
```

If mailprog.463 detects and problems, it will inform you that you have not followed the instructions given above and provide some hints how to proceed. If you followed these instructions you will see the following:

```
winans@hopper:"$ mailprog.463 a5
******************************************************************************
* WARNING : Do NOT use this program to mail notes to your Instructor *
* Doing so may result in the loss of your program !! *
******************************************************************************
Enter program number for your assignment : 5
shar: Saving /tmp/mailprog.11111 (text)
winans@hopper:"$
```

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 -ansi -pedantic -Wall -Wextra -Werror -std=c++14 -c -o main.o main.cpp

g++ -g -ansi -pedantic -Wall -Wextra -Werror -std=c++14 -c -o rv32i.o rv32i.cpp

g++ -g -ansi -pedantic -Wall -Wextra -Werror -std=c++14 -c -o memory.o memory.cpp

g++ -g -ansi -pedantic -Wall -Wextra -Werror -std=c++14 -c -o registerfile.o registerfile.cpp

g++ -g -ansi -pedantic -Wall -Wextra -Werror -std=c++14 -o rv32i main.o rv32i.o memory.o registerfile.o hex.o

Your program will then be run multiple times using different memory sizes and test data files as shown in the section discussing your program output above.

7 Hints

• Start by updating main.cpp and rv32i::run() to call a stub version of rv32i::exec() to sanity-check your new run() function:

```cpp
std::string rv32i::exec(uint32_t insn)
{
    pc += 4;
    return "ERROR: UNIMPLEMENTED INSTRUCTION";
}
```

(When necessary, you can temporarily remove the -Wextra compiler argument to shut off warnings about unused variables.) Such an exec() function will result in the same output from assignment 4 except it will have an error message in the exec operation comment.

```plaintext
00000000: abcd237 lui x4,0xabcd237 // ERROR: UNIMPLEMENTED INSTRUCTION
00000004: abcd217 auipc x4,0xabcd217 // ERROR: UNIMPLEMENTED INSTRUCTION
00000008: 004000ef jal x1,0xc // ERROR: UNIMPLEMENTED INSTRUCTION
0000000c: 00408267 jalr x4,4(x1) // ERROR: UNIMPLEMENTED INSTRUCTION
...
```

• Add the registerfile class and the dump() methods to it and the rv32i class. You should then be able to finish your run() loop to include the machine state dumps so that each instruction execution looks like this:

```plaintext
x0 00000000 f0f0f0f0 000000a0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 pc 00000000
00000000: abcd237 lui x4,0xabcd237 // ERROR: UNIMPLEMENTED INSTRUCTION
```

• Use the big switch statement from your decode() as a template structure for your exec() method. The first instruction you should implement should be ebreak (so the test programs can stop your simulator.) At this stage, your exec() might look like Figure 3.

This example code snippet suggests that I implemented each instruction in a method of their own and that each one returns their own operational details. For example, my exec_ebreak() does this:

```cpp
std::string rv32i::exec_ebreak(uint32_t insn)
{
    (void)insn; // shut up unused argument compiler warning
    halt = true; // stop the simulator
    return "HALT"; // the operation comment
}
Figure 3: Implementing `exec()` and `ebreak`

You might also consider setting `halt = true` in the bottom of your `exec()` to stop execution if an as-yet unimplemented instruction is encountered.

- 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), here is a way to implement `exec_lui()`. It is a more typical instruction that uses the `registerfile` and updates `pc` to point to the next instruction as described in the reference card at the end of RVALP as:

  \[
  rd \leftarrow \text{imm}_u, \quad pc \leftarrow pc+4
  \]

To implement this instruction extract the relevant fields using the `get_rd()` and `get_imm_u()` methods, set the destination register using `regs.set()`, and increment the `pc` register as shown in Figure 4.
std::string rv32i::exec_lui(uint32_t insn) {
    std::ostringstream os;
    uint32_t rd = get_rd(insn); // get the rd register number
    int32_t imm_u = get_imm_u(insn); // get the imm_u value
    regs.set(rd, imm_u); // store the imm_u value into rd
    pc += 4; // advance the pc past the end of this instruction

    // Return a string indicating the operation that has taken place
    os << std::dec << "x" << rd << " = " << hex (imm_u);
    return os.str();
}

std::string rv32i::exec(uint32_t insn) {
    uint32_t opcode = get_opcode(insn);
    uint32_t funct3 = get_funct3(insn);
    uint32_t funct7 = get_funct7(insn);

    switch(opcode) {
    case 0b0110111: return exec_lui(insn);
    ...
    case 0b1110011: // ECALL / EBREAK
        // treat both ECALL and EBREAK as an EBREAK and halt the program
        if (insn&0x00100000)
            return exec_ebreak(insn);
        else
            return exec_ebreak(insn);
    break;
    }

    pc += 4;
    halt = true; // ABORT THE EXECUTION LOOP!
    return "ERROR: UNIMPLEMENTED INSTRUCTION";
}
```assembly
00000000: abcede237 lui x4,0xabcdef // x4 = 0xabcdef000
00000004: abcede217 auipc x4,0xabcdef // x4 = 0x00000004 + 0xabcdef000 = 0xabcdef004
00000008: 090000ef jal x1,0x98 // x1 = 0x0000000c, pc = 0x00000008 + 0x0000000c = 0x00000008 + 0x0000000c = 0x000000084
0000000c: 0f408267 jalr x4,244(x1) // x4 = 0x00000010, pc = (0xf0f0f0f0 + 0x0000000c) & 0xfffffffe = 0x00000100
00000010: 08b50c63 beq x10,x11,0xa8 // pc += (0xf0f0f0f0 == 0xf0f0f0f0 ? 0x00000098 : 4) = 0x000000a8
00000014: 08b51a63 bne x10,x11,0xa8 // pc += (0xf0f0f0f0 != 0xf0f0f0f0 ? 0x00000094 : 4) = 0x00000018
00000018: 08b54863 blt x10,x11,0xa8 // pc += (0xf0f0f0f0 < 0xf0f0f0f0 ? 0x00000090 : 4) = 0x0000001c
0000001c: 08b55663 bge x10,x11,0xa8 // pc += (0xf0f0f0f0 >= 0xf0f0f0f0 ? 0x0000008c : 4) = 0x00000024
00000020: 08b56463 bltu x10,x11,0xa8 // pc += (0xf0f0f0f0 <U 0xf0f0f0f0 ? 0x00000088 : 4) = 0x00000024
00000024: 08b57263 bgeu x10,x11,0xa8 // pc += (0xf0f0f0f0 >=U 0xf0f0f0f0 ? 0x00000084 : 4) = 0x000000a8
00000028: 4d204203 lbu x4,1234(x0) // x4 = zx(m8(0x00000000 + 0x000004d2)) = 0x000000a5
0000002c: 4d205203 lhu x4,1234(x0) // x4 = zx(m16(0x00000000 + 0x000004d2)) = 0x0000a5a5
00000030: 4d200203 lb x4,1234(x0) // x4 = sx(m8(0x00000000 + 0x000004d2)) = 0xffffffa5
00000034: 4d201203 lh x4,1234(x0) // x4 = sx(m16(0x00000000 + 0x000004d2)) = 0xffffa5a5
00000038: 4d202203 lw x4,1234(x0) // x4 = sx(m32(0x00000000 + 0x000004d2)) = 0xa5a5a5a5
0000003c: 4c400923 sb x4,1234(x0) // m8(0x00000000 + 0x000004d2) = 0x000000a5
00000040: 4c401923 sh x4,1234(x0) // m16(0x00000000 + 0x000004d2) = 0x0000a5a5
00000044: 4c402923 sw x4,1234(x0) // m32(0x00000000 + 0x000004d2) = 0xa5a5a5a5
00000048: 4d260213 addi x4,x12,1234 // x4 = 0xf0f0f0f0 + 0x000004d2 = 0xf0f0f5c2
0000004c: 4d262213 slti x4,x12,1234 // x4 = (0xf0f0f0f0 < 1234) ? 1 : 0 = 0x00000001
00000050: 4d263213 sltiu x4,x12,1234 // x4 = (0xf0f0f0f0 <U 1234) ? 1 : 0 = 0x00000000
00000054: 4d264213 xor x4,x12,1234 // x4 = 0xf0f0f0f0 ^ 0xf0f0f0f0 = 0x00000000
00000058: 4d265213 or x4,x12,1234 // x4 = 0xf0f0f0f0 | 0xf0f0f0f0 = 0xf0f0f0f0
0000005c: 4d267213 and x4,x12,1234 // x4 = 0xf0f0f0f0 & 0xf0f0f0f0 = 0xf0f0f0f0
00000060: 0c69213 slli x3,x14,12 // x3 = 0xf0f0f0f0 << 16 = 0xf0f00000
00000064: 0c6d213 srl x3,x14,12 // x3 = 0xf0f0f0f0 >> 16 = 0x0000f0f0
00000068: 0c6d213 srai x3,x14,12 // x3 = 0xf0f0f0f0 >> 12 = 0xfffff0f0
0000006c: 00f70233 add x4,x14,x15 // x4 = 0xf0f0f0f0 + 0xf0f0f0f0 = 0xe1e1e1e0
00000070: 00f71233 slt x4,x14,x15 // x4 = 0xf0f0f0f0 < 0xf0f0f0f0 ? 1 : 0 = 0x00000000
00000074: 00f72233 srl x3,x14,15 // x3 = 0xf0f0f0f0 >> 16 = 0xf0f0f0f0
00000078: 00f73233 or x4,x14,x15 // x4 = 0xf0f0f0f0 | 0xf0f0f0f0 = 0xf0f0f0f0
0000007c: 00f75233 fence iorw,i // fence
```

Figure 5: `exec()` Instruction operation comment format.

---

Copyright © 2020 John Winans. All Rights Reserved

`~/NIU/courses/463/2020-sp/assignments/a5/handout.tex`

jwinans@niu.edu 2020-04-08 19:31:58 -0500 v2.0-480-gafe1002

Page 10 of 10