Abstract

In this assignment, you will extend the functionality of your memory simulator by using it to load and disassemble a binary RISC-V executable file. This is the second of a multi-part assignment concluding with 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

Disassemble an executable binary file by loading it into a simulated memory of sufficient size and decode each 32-bit instruction one-at-a-time starting from address zero and continuing through the end of the simulated memory.

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 a4 and place within it a copy of all the the source files from assignment 3 plus the additional files defined below.

main.cpp Your main() and usage() function definitions will go here.

hex.h The declarations of your hex formatting functions will go here (copied from assignment 3.)

hex.cpp The definitions of your hex formatting functions will go here (copied from assignment 3.)

memory.h The definition of your memory class will go here (copied from assignment 3.)

memory.cpp The memory class member function definitions will go here (copied from assignment 3.)

rv32i.h The definition of the rv32i class will go here.

rv32i.cpp The definitions of any member functions of class rv32i will go here.

2.1 main.cpp

You must provide a main() function that is implemented as shown in Figure 1 (plus appropriate documentation):
int main(int argc, char **argv)
{
    if (argc != 3)
        usage();

    memory mem(stoul(argv[1], 0, 16));

    if (!mem.load_file(argv[2]))
        usage();

    rv32i sim(&mem);
    sim.run();
    mem.dump();

    return 0;
}

Figure 1: Example main() function.

Your usage() function must print an appropriate error message and terminate the program in the traditional manner:
https://en.wikipedia.org/wiki/Usage_message

2.2 hex.h and hex.cpp

See assignment 3 handout.

2.3 memory.h and memory.cpp

See assignment 3 handout.

2.4 rv32i.h and rv32i.cpp

Your rv32i class must include the members discussed below plus appropriate documentation. Failure to follow this design will cause significant problems with future assignments implemented by extending this assignment!

2.4.1 rv32i Member Functions

- rv32i::rv32i(memory *m);
  Save the m argument in the mem member variable for use later when disassembling.

- void rv32i::run(void);
  In this assignment, this run method will be used to disassemble the instructions in the simulated memory.
  To perform this task, set pc to zero and then, for each 32-bit word in the memory:
- print the address in pc
- call decode(mem->get32(pc)) to fetch and decode the instruction into a printable
  std::string
- print the decoded instruction string
- increment pc by 4 (point to the next instruction)

- static uint32_t rv32i::get_opcode(uint32_t insn);
  Extract and return the opcode field from the given instruction.

- static uint32_t rv32i::get_rd(uint32_t insn);
  Extract and return the rd field from the given instruction.

- static uint32_t rv32i::get_funct3(uint32_t insn);
  Extract and return the funct3 field from the given instruction.

- static uint32_t rv32i::get_rs1(uint32_t insn);
  Extract and return the rs1 field from the given instruction.

- static uint32_t rv32i::get_rs2(uint32_t insn);
  Extract and return the rs2 field from the given instruction.

- static uint32_t rv32i::get_funct7(uint32_t insn);
  Extract and return the funct7 field from the given instruction.

- static int32_t rv32i::get_imm_i(uint32_t insn);
  Extract and return the imm_i field from the given instruction.

- static int32_t rv32i::get_imm_u(uint32_t insn);
  Extract and return the imm_u field from the given instruction.

- static int32_t rv32i::get_imm_b(uint32_t insn);
  Extract and return the imm_b field from the given instruction.

- static int32_t rv32i::get_imm_s(uint32_t insn);
  Extract and return the imm_s field from the given instruction.

- static int32_t rv32i::get_imm_j(uint32_t insn);
  Extract and return the imm_j field from the given instruction.

- std::string rv32i::decode(uint32_t insn) const;
  It is the purpose of this function to return a std::string containing the hex instruction
  value and the disassembled instruction text. This function will not print anything.
  Implement this software decoder by using a switch statement. For this function, use the
  value of the opcode extracted with get_opcode(insn) as the switch expression.
  For each case, either format the instruction and its arguments (when the opcode is specific
  enough to do so such as for the lui instruction...) or use a sub-switch statement to further
decode any additionally required fields (such as \texttt{funct3} and/or \texttt{funct7}) and then format the decoded instruction as shown in Figure 3.

For any invalid instructions, return a string with the hex instruction value and the word \texttt{ERROR} where the instruction mnemonic would otherwise appear.

Note that for each unique opcode value, the disassembled instruction format is the same. For example, all of the \textit{store} instructions have the same opcode: \texttt{0b0100011} and differ only in their mnemonic. Take advantage of this by factoring your string formatting logic to minimize the amount of replicated code as shown in Figure 3.

This function must be capable of handling any 32-bit value.

\section*{2.4.2 \texttt{rv32i} Member Variables}

- \texttt{memory \* mem;}
  
  This will contain a pointer to the \texttt{memory} object from assignment 3. It will be used by the disassembler logic to fetch the instructions.

- \texttt{uint32\_t pc;}
  
  Use this to contain the address of the instruction being decoded/disassembled. When decoding instructions that refer to the \texttt{pc} register to calculate a target address (e.g. \texttt{auipc}, \texttt{jal}, and branch instructions) use this value to determine the instruction’s memory address.
  
  Initialize \texttt{pc} to zero.

- \texttt{static constexpr uint32\_t XLEN = 32;}
  
  \texttt{XLEN} represents the number of bits in a RISC-V CPU registers.

\section*{3 Input}

Your program will accept two arguments on the command line as shown in the \texttt{main()} code snippet above.

The first argument is a hex number representing the amount of memory to simulate.

The second argument is the name of a file to load into the simulated memory. In this assignment, this will be the name of a binary \texttt{rv32i} executable program.

You will be provided with a number of suitable executable test programs.

\section*{4 Output}

Your program’s output will be a disassembly of the executable binary file given on the command line followed by a dump of the simulated memory.

See Figure 2 for a list of every instruction and the proper disassembly format. Your program match the reference output precisely.
5 How To Hand In Your Program

When you are ready to turn in your assignment, make sure that the only files in your a4 directory is/are the source files defined and discussed above. Then, in the parent of your a4 directory, use the mailprog.463 command to send the contents of the files in your a4 project directory to your TA like this:

mailprog.463 a4

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 a4
**********************************************************************
* 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 : 4
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 hex.o hex.cpp
g++ -g -ansi -pedantic -Wall -Wextra -Werror -std=c++14 -o rv32i main.o rv32i.o memory.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

While this is the second part of a multi-part assignment, it too should be written in parts!
• Write the `get_xxx()` methods that you are using as you encounter the need for them. (It is OK to have a declaration of a method in your `rv32i` class without actually defining it as long as you don’t call it.)

• Start by updating `main.cpp` and stub in your `rv32i::run()` such that it does nothing but return so you can just get your new framework to just compile and verify that is OK by running it and see that `mem.load_file()` and `mem.dump()` still work OK.

• You will extract the field data in your `get_xxx()` methods by using using a combination of the bitwise `and &`, `or |`, and shift operators: `>>` and `<<` as seen in Figure 4.

• Implement and test one opcode at-a-time using the reference card and the diagrams showing how the instruction fields are decoded in “RISC-V Assembly Language Programming” (AKA rvalp.) The latest version of rvalp is located here: https://github.com/johnwinans/rvalp/releases/ (Click on the “Assets” menu for the latest/top shown release and open “book.pdf.”)

  Start with the easy ones (that don’t use `funct3` or `funct7` such as the U-type instructions that appear at the beginning of the `tinyprog.bin` example file.

  Note the presence of `-std=c++14` on the compiler flags. Therefore we can use binary literals as shown in Figure 3. This should reduce typographical errors when copying the opcodes from the reference card on the last page of rvalp.
Figure 2: Example output from running: `.rv32i a0 allinsn.bin`
std::string rv32i::decode(uint32_t insn) const
{
    std::ostringstream os;
    os << hex32(insn) << " "; // the instruction hex value
    switch(get_opcode(insn))
    {
        case 0b0110111: // LUI
            os << " lui x" << std::dec << get_rd(insn) << " ,0x" << std::hex
                << ((get_imm_u(insn) >> 12) & 0xfffff);
            break;
        ...
        case 0b0100011: // S-type store instructions
            switch (get_funct3(insn))
            {
                case 0b000: os << "sb "; break;
                case 0b001: os << "sh "; break;
                case 0b010: os << "sw "; break;
                default:
                    os << " ERROR";
                    return os.str();
            }
            os << "x" << std::dec << get_rs2(insn) << "," << get_imm_s(insn) << "(x"
                << get_rsi(insn) << ");
            break;
        ...
        default:
            os << " ERROR";
    }
    return os.str();
}

Figure 3: Decoding the using a switch statement.

int32_t rv32i::get_imm_s(uint32_t insn)
{
    int32_t imm_s = (insn & 0xf0000000) >> (25-5); // extract & shift bits 5-11
    imm_s |= (insn & 0x00000f80) >> (7-0); // extract & shift bits 0-4
    if (insn & 0x80000000) // sign-extend
        imm_s |= 0xfffff000;
    return imm_s;
}

Figure 4: Extracting the imm_s field from an instruction.