CSCI 463 Assignment 4 – RV32I Disassembler

20 Points – Due Thursday, October 22, 2020 at 23:59

Abstract

In this assignment you will load and disassemble a binary RV32I executable file. This is the second of a multi-part assignment involving the creation of a machine capable of executing programs compiled with g++. The purpose of this assignment gain a thorough understanding of the RV32I 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 continue through to 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 any appropriate includes etc.) This version differs from the main() in assignment 3 by replacing the memory test function calls with the creation and use of the new rv32i class and by making the memory size an
optional command line argument (as clearly indicated in the usage statement by the fact that it is displayed surrounded by square brackets as seen on line 6 of Figure 1.)

```c
/**
 * Print a usage message and abort the program.
 *********************************************/
static void usage ()
{
    cerr << "Usage: rv32i [-m hex-mem-size] infile" << endl;
    cerr << " -m specify memory size (default = 0x10000)" << endl;
    exit(1);
}

/**
 * Read a file of RV32I instructions and execute them.
 *********************************************/
int main (int argc , char ** argv)
{
    uint32_t memory_limit = 0x10000; // default memory size = 64k
    int opt;
    while ((opt = getopt(argc , argv , "m:")) != -1)
    {
        switch (opt)
        {
            case 'm':
            {
                memory_limit = std::stoul(optarg , nullptr ,16);
                break;
            }
            default: /* '?' */
                usage();
        }
    }
    if (optind >= argc)
        usage();    // missing filename
    memory mem(memory_limit);
    if (!mem.load_file(argv[optind]))
        usage();
    rv32i sim(&mem);
    sim.disasm();
    mem.dump();
    return 0;
}
```

Figure 1: Example `main()` function.

The `usage()` function prints an appropriate error message and terminates 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 likely cause significant problems with future assignments implemented that will be implemented by extending this assignment.

2.4.1 rv32i Member Variables

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

- 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. jal, and the branch instructions), use this value when calculating memory addresses that appear in the operands.
  Initialize pc to zero.

- static constexpr uint32_t XLEN = 32;
  XLEN represents the number of bits in RV32I CPU registers.

- You might find it useful to define a number of additional constants along the lines of XLEN that represent the binary opcode, funct3, and funct7 values and some useful widths for print formatting. For example:

```cpp
static constexpr int mnemonic_width = 8;
static constexpr int instruction_width = 35;
static constexpr uint32_t opcode_lui = 0b0110111;
static constexpr uint32_t opcode_auipc = 0b0010111;
static constexpr uint32_t opcode_jal = 0b1101111;
static constexpr uint32_t opcode_jalr = 0b1100111;
static constexpr uint32_t opcode_btype = 0b1100011;
static constexpr uint32_t opcode_load_imm = 0b0000011;
...```

Note that these opcode constants are expressed with \textit{binary} literals. This notation was added in C++14. (Make sure you use the proper compiler flags.) A file with the above constants and others will be made available with the assignment test data files discussed below.

\subsection*{2.4.2 \texttt{rv32i} Member Functions}

\begin{itemize}
    \item \texttt{rv32i(memory *m);}
        
        Save the \texttt{m} argument in the \texttt{mem} member variable for use later when disassembling.
    \item \texttt{void disasm(void);}
        
        This method will be used to disassemble the instructions in the simulated memory.
        
        To perform this task, set \texttt{pc} to zero and then, for each 32-bit word in the memory:
        \begin{itemize}
            \item print the 32-bit hex address in the \texttt{pc} register
            \item \textit{fetch} the 32-bit instruction from memory at the address in the \texttt{pc} register
            \item print the instruction as a 32-bit hex value.
            \item pass the fetched instruction to \texttt{decode()} to \textit{decode} and render the instruction into a printable \texttt{std::string}
            \item print the decoded instruction string returned from \texttt{decode()}
            \item increment \texttt{pc} by 4 (point to the next instruction)
        \end{itemize}
        
        Continue until you have fetched, decoded, and printed every word from your simulated memory.
    \item \texttt{std::string decode(uint32_t insn) const;}
        
        This function must be capable of handling any possible \texttt{insn} value.
        
        It is the purpose of this function to return a \texttt{std::string} containing the disassembled instruction text. \textit{This function will not print anything}. You can implement this software decoder by using a \texttt{switch} statement. For this function, use the value of the opcode extracted with \texttt{get_opcode(insn)} as the switch expression.
        
        For each \texttt{case}, either format the instruction and its arguments (when the opcode is specific enough to do so such as for the \texttt{lui} instruction...) or use a sub-\texttt{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 error message shown below where the instruction mnemonic and arguments would otherwise appear:
        \begin{center}
            \textbf{ERROR: UNIMPLEMENTED INSTRUCTION}
        \end{center}
        
        Note that for each unique opcode value, the disassembled instruction format is the same. For example, all of the \texttt{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.
If you factor out ALL your instruction formatting then you will find it easier to reuse your code in future assignments. For example, consider creating member functions for print formatting code along the lines of:

```cpp
std::string render_illegal_insn() const;
std::string render_lui(uint32_t insn) const;
std::string render_auipt(uint32_t insn) const;
std::string render_jal(uint32_t insn) const;
std::string render_jalr(uint32_t insn) const;
std::string render_btype(uint32_t insn, const char *mnemonic) const;
std::string render_itype_load(uint32_t insn, const char *mnemonic) const;
std::string render_stype(uint32_t insn, const char *mnemonic) const;
std::string render_itype_alu(uint32_t insn, const char *mnemonic, int32_t imm_i) const;
std::string render_rtype(uint32_t insn, const char *mnemonic) const;
std::string render_fence(uint32_t insn) const;
std::string render_ecall(uint32_t insn) const;
std::string render_ebreak(uint32_t insn) const;
```

These are only suggestions. You should feel free to alter, add, or remove as you find useful.

2.4.3 rv32i Static Member Functions

These additional helper functions will make the decoding and print-formating logic easier to write, debug, and understand.¹

- **static uint32_t get_opcode(uint32_t insn);**
  Extract and return the opcode field from the given instruction. A suitable implementation might be implemented like this:

  ```cpp
  return (insn & 0x0000007f);
  ```

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

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

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

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

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

¹Recall that static member functions do not have a this and therefore don't have access to any rv32i object instance members.
• static int32_t get_imm_i(uint32_t insn);
  Extract and return the imm_i field from the given instruction.

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

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

• static int32_t get_imm_s(uint32_t insn);
  Extract and return the imm_s field from the given instruction. See Figure 4.

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

3 Input

Your program will accept one or two arguments on the command line as shown in the main() code snippet above.

The optional [-m hex-mem-size] argument is a hex number representing the amount of memory to simulate. Set the default value for this to 0x10000.

The last 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 a number of suitable executable test programs and associated output files.

4 Output

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

See Figure 2 for a list of every instruction and the proper disassembly format. Your program must match the reference output precisely.

The assembly language dump is rendered with a mix of decimal and hex values:

• The far left column is the 32-bit address printed in hex.

• The second column is the 32-bit hex full-word representation of the instruction that was fetched from your memory at the address in column 1.

• The third column is the instruction mnemonic.

• The presence and format of the fourth column depends on the type of the instruction.
Register numbers are printed as an 'x' followed by the decimal number of the register (0-31) as in x12, x0, and x31.

- Hexadecimal literals are printed with a leading 0x as in 0xabcde.
- Decimal literals are printed with the first character being a number (1-9) as in 1234.

It seems unfortunate that the letter x was chosen for the register indicator since it is so close to (and could be confused with) the indicator of hexadecimal numbers 0x. Be careful!

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 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:

```bash
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.

7 Hints

While this is the second part of a multi-part assignment, it too should be written in parts!

- Start by updating main.cpp and stub in your rv32i::disasm() such that it does nothing but return so you can just get your new framework to compile and verify that is OK by running it and see that mem.load_file() and mem.dump() still work OK.
- Implement the rv32i::disasm() loop to just print the pc and fetched instruction-word values.
- Add the call to rv32i::decode() and stub in just enough to treat all the instructions as illegal.
• Add each instruction opcode one at-a-time starting with the lui example code in Figure 5.

• 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.)

• 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 is located here: https://github.com/johnwinans/rvalp/releases/ (Click on the “Assets” menu for the latest/top shown release and open “book.pdf.”)

• Do all the the easy ones first such as the U-type and R-type instructions that appear at the begining of the tinyprog.bin example file.
winans@hopper:$ ./rv32i -m a0 allinsns.bin

Figure 2: Example output from running: ./rv32i a0 allinsns.bin
std::string rv32i::decode(uint32_t insn) const
{
    uint32_t opcode = get_opcode(insn);
    uint32_t funct3 = get_funct3(insn);
    uint32_t funct7 = get_funct7(insn);
    int32_t imm_i = get_imm_i(insn);

    switch(opcode)
    {
        default: return render_illegal_insn();
        case opcode_lui: return render_lui(insn);
        case opcode_auipc: return render_auipc(insn);
        case opcode_rtype:
            switch(funct3)
            {
                default: return render_illegal_insn();
                case funct3_add:
                    switch(funct7)
                    {
                        default: return render_illegal_insn();
                        case funct7_add: return render_rtype(insn, "add");
                        case funct7_sub: return render_rtype(insn, "sub");
                    }
                    assert(0 && "unhandled funct7");
                    case ...
                    ...
                }
                assert(0 && "unhandled funct3");
                case opcode_alu_imm:
                    switch (funct3)
                    {
                        default: return render_illegal_insn();
                        case funct3_sll: return render_itype_alu(insn, "slli", imm_i%XLEN);
                        case funct3_srx:
                            switch(funct7)
                            {
                                default: return render_illegal_insn();
                                case funct7_sra: return render_itype_alu(insn, "srai", imm_i%XLEN);
                                case funct7_srl: return render_itype_alu(insn, "srli", imm_i%XLEN);
                            }
                            assert(0 && "unhandled funct7");
                            case funct3_add: return render_itype_alu(insn, "addi", imm_i);
                            case ...
                            ...
                        }
                        assert(0 && "unhandled funct3");
                        case ...
                        ...
                    }
                    assert(0 && "unhandled opcode");
    }
}

Figure 3: Decoding the using a switch statement.
int32_t rv32i::get_imm_s(uint32_t insn)
{
    int32_t imm_s = (insn & 0x00000f80) >> (25 - 5); // extract & shift bits 5-11
    imm_s |= (insn & 0x80000000) // sign-extend
    imm_s |= 0xfffff000;
    return imm_s;
}

std::string rv32i::render_lui(uint32_t insn) const
{
    uint32_t rd = get_rd(insn);
    int32_t imm_u = get_imm_u(insn);
    std::ostringstream os;
    os << std::setw(mnemonic_width) << std::setfill(' ') << std::left << "lui"
        << "x" << std::dec << rd << ",0x" << std::hex << (imm_u >> 12) & 0xfffff;
    return os.str();
}

Figure 4: Extracting the imm_s field from an instruction.

Figure 5: Rendering the lui instruction.