CSCI 463 Assignment 4 – RV32I Disassembler

20 Points

Abstract

In this assignment you will load RV32I binary executable file into your memory simulator and decode/disassemble the instructions.

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 machine instructions.

1 Problem Description

Disassemble an executable binary file by loading it into a simulated memory of sufficient size and then 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 rv32i_decode 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.

hex.cpp The definitions of your hex formatting functions will go here.

memory.h The definition of your memory class will go here.

memory.cpp The memory class member function definitions will go here.

types.h The definition of a class named types will go here.

rv32i_decode.h The definition of a class named rv32i_decode will go here.

rv32i_decode.cpp The definitions of any member functions of class rv32i_decode will go here.

2.1 main.cpp

You must provide a main() function that is implemented as shown below plus a new utility function to disassemble the memory contents. This version of main() differs from the that in Assignment 3 by replacing the memory test function calls with the use of the new rv32i_decode class and the addition of a disassemble() helper function:
The `disassemble()` function is what will decode and print each instruction in a loop. For each 32-bit word in the simulated memory:

- Call `rv32i_decode::decode()` to format/render one instruction mnemonic and return it as a `std::string`.
- Print the memory address, instruction hex value, and the instruction mnemonic.

See the example output files below and on the course web page for details on how to align and space the output.
2.2  hex.h and hex.cpp

Same as Assignment 3 plus the following additional methods:

- \texttt{static std::string to\_hex0x20(uint32\_t i);}  
  This function must return a \texttt{std::string} beginning with \texttt{0x}, followed by the 5 hex digits representing the 20 least significant bits of the \texttt{i} argument. Implement this method in a similar fashion as \texttt{hex::to\_hex8()} but without the \texttt{static\_cast}.  
  This new method will be used when formatting the \texttt{lui} and \texttt{auipc} instructions.

- \texttt{static std::string to\_hex0x12(uint32\_t i);}  
  This function must return a \texttt{std::string} beginning with \texttt{0x}, followed by the 3 hex digits representing the 12 least significant bits of the \texttt{i} argument. See \texttt{to\_hex0x20()}.  
  This new method will be used when formatting the \texttt{csrrX()} instructions.

2.3  memory.h and memory.cpp

Same as for Assignment 3.

2.4  rv32i\_decode.h and rv32i\_decode.cpp

Your \texttt{rv32i\_decode} class must, at least, 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.

Note that \textit{all} of the members of this class are \texttt{static} and \textit{none} will actually “print” anything to the screen.

The \texttt{rv32i\_decode} class is a subclass of the \texttt{hex} class. Therefore it will inherit all the hex formatting methods.

2.4.1  rv32i\_decode Member Constants

- \texttt{static constexpr uint32\_t XLEN = 32;}  
  \texttt{XLEN} represents the number of bits in RV32I CPU registers.

- You will find it useful to define a number of constants that represent the binary values of the opcode, funct3, and funct7 fields of each of the instructions as well as some useful widths for formatting the returned string values. For example:
  \begin{verbatim}
  static constexpr int mnemonic_width = 8;
  static constexpr int opcode_lui = 0b0110111;
  static constexpr int opcode_auipc = 0b0010111;
  ...
  \end{verbatim}
Note that these opcode constants are expressed with binary literals. This notation was added in C++14. (Make sure you use the proper compiler flags.)

A file with the above and other useful constants will be made available on the course web page.

### 2.4.2 rv32i_decode Member Functions

- **static std::string decode(uint32_t addr, uint32_t insn);**

  It is the purpose of this function to return a std::string containing the disassembled instruction text.

  It must be capable of handling any possible 32-bit insn value (regardless of it representing a useful/legal instruction or not.)

  Note that the memory address parameter addr, from which the instruction has been fetched, is passed to decode() so that it may be used to calculate the PC-relative target address shown in the J-type and B-type instructions.

  You can implement this software decoder by using a switch statement with the value of the opcode field by extracting it from insn by calling get_opcode(insn).

  For each case, call a function to format/render 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 funct3 and/or funct7) and then call a function to format the instruction as shown in section 7.

  Note that, most of the instructions are in groups such that the disassembled instruction format is the same. For example, all of the store instructions differ only in their mnemonic.

  Take advantage of this by factoring your std::string formatting logic to minimize the amount of replicated code. To do this, add formatting helper-functions like these to the rv32i_decode class as needed:

    ```
    static std::string render_illegal_insn();
    static std::string render_auipc(uint32_t insn);
    static std::string render_jal(uint32_t addr, uint32_t insn);
    static std::string render_btype(uint32_t addr, uint32_t insn, const char *mnemonic);
    static std::string render_itype_load(uint32_t addr, uint32_t insn, const char *mnemonic);
    static std::string render_itype_alu(uint32_t insn, const char *mnemonic, int32_t imm_i);
    static std::string render_stype(uint32_t insn, const char *mnemonic);
    ...
    ```

  The simplest of these is the renderer for invalid instructions:

    ```
    std::string rv32i_decode::render_illegal_insn()
    {
      return "ERROR: UNIMPLEMENTED INSTRUCTION";
    }
    ```
See the `render_lui()` example in section 7 for a more interesting example.

These above `render_X()` functions should also use helper-functions like those below to eliminate messy replicated code:

- `static std::string render_reg(int r);`
  Render, into a `std::string`, the name of the register with the given number `r`.
  For example, `render_reg(23)` would return a string with the value: `x23`
  Calling this member is the one and only way that any code may format the name of a register into a string (for subsequent printing by the program.)

- `static std::string render_base_disp(uint32_t register, int32_t imm);`
  Use this to render, into a `std::string`, the operands of the form `imm(register)` for those instructions that have such an addressing mode.
  Note that, in this case, the `imm` value is printed in decimal. The register should be rendered into a string by `render_reg()`.
  Calling this member is the one and only way that any code may format an `imm(register)` operand in your application.

- `static std::string render_mnemonic(const std::string &mnemonic);`
  Render, into a `std::string`, the given instruction mnemonic with the proper space padding on right side to make it `mnemonic_width` characters long.
  By using this to make all mnemonics a uniform width when printing instructions that have operands, aligning the operands will be much easier.
  Note that `ecall` and `ebreak` are the only instructions that do not have operands and that must therefore not be rendered with padding on the right.

You should feel free to alter or add more helper-functions for rendering that you find useful.

The following helper-functions for extracting the fields from a 32-bit instruction will make the instruction decoding logic easier to write, debug, and understand:

- `static uint32_t get_opcode(uint32_t insn);`
  Extract and return the `opcode` field from the given instruction. A suitable implementation is:
  
  ```
  return (insn & 0x0000007f);
  ```

- `static uint32_t get_rd(uint32_t insn);`
  Extract and return the `rd` field from the given instruction as an integer value from 0 to 31.

- `static uint32_t get_rs1(uint32_t insn);`
  Extract and return the `rs1` field from the given instruction as an integer value from 0 to 31.

- `static uint32_t get_rs2(uint32_t insn);`
  Extract and return the `rs2` field from the given instruction an integer value from 0 to 31.
• static uint32_t get_funct3(uint32_t insn);
  Extract and return the funct3 field from the given instruction as an integer value from 0 to 7.

• static uint32_t get_funct7(uint32_t insn);
  Extract and return the funct7 field from the given instruction as an integer value from 0x00 to 0x7f.

• static int32_t get_imm_i(uint32_t insn);
  Extract and return the imm_i field from the given instruction as a 32-bit signed integer as shown in RVALP.

• static int32_t get_imm_u(uint32_t insn);
  Extract and return the imm_u field from the given instruction as a 32-bit signed integer as shown in RVALP.

• static int32_t get_imm_b(uint32_t insn);
  Extract and return the imm_b field from the given instruction as a 32-bit signed integer as shown in RVALP.

• static int32_t get_imm_s(uint32_t insn);
  Extract and return the imm_s field from the given instruction. A suitable implementation is:

  ```c
  int32_t rv32i_decode::get_imm_s(uint32_t insn)
  {
    int32_t imm_s = (insn & 0xfe000000) >> (25-5);
    imm_s |= (insn & 0x00000f80) >> (7-0);
    
    if (insn & 0x80000000)
      imm_s |= 0xfffff000;  // sign-extend the left
    
    return imm_s;
  }
  ```

  Note that the shifting distances in the above example have been specified as a calculation using the values shown in the castellated diagrams in RVALP. Showing the shift distance values such as: 25-5 is more clear than simply 20 because it a) illustrates your intent and b) prevents a silly subtraction mistake from creating a problem.

• static int32_t get_imm_j(uint32_t insn);
  Extract and return the imm_j field from the given instruction as a 32-bit signed integer as shown in RVALP.

3 Input

Your program will accept the same command-line arguments as Assignment 3.
The optional \[-m\ hex-mem-size\] argument is a hex number representing the amount of memory to simulate. When not given, the default value must be 0x100.

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 RV32I binary 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.

The disassembled instructions are 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 0xabcdef.
  - Decimal literals are printed with the first character being an optional negative sign ‘-’ followed by one or more digits (0-9) as in 1234 and -58.

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!

See below for a list of every instruction and the proper disassembly format. (A copy of this output and others are available on the course web site.) Your program must match the reference output precisely (including whitespace.)

```
winans@hopper:~$ ./rv32i -mc0 testdata/allinsns4.bin
00000000: abcde237 lui x4,0xabcdef
00000004: abcde217 auipc x4,0xabcdef
00000008: 004000ef jal x1,0x0000001c
0000000c: 00808267 jalr x4,8(x1)
00000010: ff808267 jalr x4,-8(x1)
00000014: 00000263 beq x0,x0,0x00000018
00000018: fe001ee3 bne x0,x0,0x00000014
0000001c: 00004263 blt x0,x0,0x00000020
00000020: 00005263 bge x0,x0,0x00000024
00000024: 00006263 bltu x0,x0,0x00000028
00000028: 00007263 bgeu x0,x0,0x0000002c
```
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 -Werror -std=c++14 -c -o main.o main.cpp
g++ -g -ansi -pedantic -Wall -Werror -std=c++14 -c -o rv32i_decode.o rv32i_decode.cpp
g++ -g -ansi -pedantic -Wall -Werror -std=c++14 -c -o memory.o memory.cpp
g++ -g -ansi -pedantic -Wall -Werror -std=c++14 -c -o hex.o hex.cpp
```

Your program will then be run multiple times using different memory sizes and binary test 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 the `disassemble()` method 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 `disassemble()` loop to just print the `pc` and fetched instruction-word values.

- Implement the `render_illegal_insn()` helper-function and then call it from a first draft stub version of `rv32i_decode::decode()` that treats all the instructions as illegal. It might look something like this:

```cpp
std::string rv32i_decode::decode(uint32_t addr, uint32_t insn)
{
    switch(get_opcode(insn))
    {
    default:      return render_illegalInsn();
    }
    assert(0 && "unrecognized opcode"); // It should be impossible to ever get here!
}
```

...and your output would look something like this:
Note that it is OK to have a declaration of a method in your `rv32i_decode` class without actually defining it as long as you don’t call it.

Note the use of an assertion to make sure that execution can never fall off the end of the switch statement without stopping the program. Given the style of this particular switch statement design that includes a default case with a `return` and, as will be shown below, any cases added as the solution is evolved will also include a return statement, execution of the code can never run past the end of the switch statement. If it does then it means that the code in the switch statement is broken and the failing assertion will terminate the program and let us know.

- Add each instruction opcode one at-a-time starting with `lui` like this:

```cpp
std::string rv32i_decode::decode(uint32_t addr, uint32_t insn)
{
    switch(get_opcode(insn))
    {
    default: return render_illegal_insn();
    case opcode_lui: return render_lui(insn);
    }
    assert(0 && "unrecognized opcode"); // It should be impossible to ever get here!
}

std::string rv32i_decode::render_lui(uint32_t insn)
{
    uint32_t rd = get_rd(insn);
    int32_t imm_u = get_imm_u(insn);

    std::ostringstream os;
    os << render_mnemonic("lui") << render_reg(rd) << "," << to_hex0x20((imm_u >> 12)&0x0fffff);
    return os.str();
}
```

The output would then be:

```
./rv32i -m48 testdata/tinyprog.bin
00000000: 0000137 lui x2,0x0000
00000004: 0000137 lui x2,0x0001
00000008: 80000137 lui x2,0x8000
...
When you write the `get_imm_X()` and `get_functX()` methods test them with a series of useful hard-coded values to make sure you have all the bits repositioned and working correctly. It can save you a HUGE amount of debugging time if you exhaustively test them with many bit patterns before using them.

Extract the field data in these methods by using using a combination of the bitwise `and`, `&`, `or`, `|`, and `shift` operators: `>>` and `<<` as seen in the `get_imm_s()` example code seen earlier.

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/](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 beginning of the `tinyprog.bin` example file.

When implementing instructions that share the same rendering code, use one helper for all of them. For example, the `render_btype()` can handle the rendering any of the branch instructions and a `render_itype_alu()` helper-function can handle the rendering any of I-type instructions that perform ALU operations as can be seen below:

```cpp
std::string rv32i_decode::decode(uint32_t addr, uint32_t insn) {
    switch(get_opcode(insn))
    {
    default: return render_illegal_insn();
    case opcode_lui: return render_lui(insn);
    ...  
    case opcode_btype:
        switch (funct3)
        {
        default: return render_illegal_insn();
        case funct3_beq: return render_btype(addr, insn, "beq");
        case funct3_bne: return render_btype(addr, insn, "bne");
        ...
        }
        assert(0 && "unrecognized funct3"); // impossible
    case opcode_alu_imm:
        switch (funct3)
        {
        default: return render_illegal_insn();
        case funct3_add: return render_itype_alu(insn, "addi", get_imm_i(insn));
        ...
        case funct3_sll: return render_itype_alu(insn, "slli", get_imm_i(insn)%XLEN);
        case funct3_srx:
            switch(funct7)
            {
            default: return render_illegal_insn();
            case funct7_sra: return render_itype_alu(insn, "srai", get_imm_i(insn)%XLEN);
            case funct7_srl: return render_itype_alu(insn, "srli", get_imm_i(insn)%XLEN);
            }
            assert(0 && "unrecognized funct7"); // impossible
    ```
assert(0 && "unrecognized funct3"); // impossible

Note that the `shamt` value passed to the `render_itype_alu()` helper-function for the shift-immediate instructions is the same thing as the `imm_i` value if we ignore all but the least-significant `XLEN` bits... which is the remainder if we divide the `imm_i` by `XLEN`. 