Computer Architecture Simulator Report

Nishi Baranwal, CS23BTECH11041
Paidala Vindhya, CS23BTECH11044

Contents

1 Coding Approach

1.1 Design Decisions

- We have divided the source file into multiple .cpp files and one .h file to have better readability of the code since altogether there are many 100s of lines of code. It would also be easy to make any changes if divided into smaller individual files. It is a kind of abstraction to one function against the other functions.
- **riscv_header.h** header file consists of all the function declarations and the global variables which are written in other .cpp files. The header file is then *included* in the .cpp files
- **hexcode_generator.cpp** file has all the functions handling the generation of hexcode of the assembly code given as input, loading the values from .data section into memory.
  - It has the functions - machine_code : to generate hexcode, store_in_memory : to store the values in .data section in memory.
  - Here, we deal with loading of file. The register values, and memory values are initialized with default values.
  - The values from data section is stored in data memory. The hexcode of the instructions is generated. If any errors, then 'load_error' is assigned 1 (to indicate some error has come while loading).
  - If no error then after this, in main function we store the hexcodes generated in text section of memory.

- In the **main.cpp** file, in the main function, we run an infinite loop waiting for commands from the user. Based on the input given by the user we provide that functionality using different functions like - **run(), step(), print_reg_values(), initializeTextMem(), print_memory_values()** etc.
- Created some general functions in the **assembler_gen_func.cpp** and the **simulator_gen_func.cpp** files such as -
  - assembler_gen_func.cpp
    - findformat
    - bin_to_hex
    - bin_to_hex_sz
    - deci_to_bin
    - regname_to_binary
    - immediate_binary
    - hex_to_bin
    - hex_to_deci
    - deci_to_hex
    - parse
  - simulator_gen_func.cpp
    - initializeTextMem
    - initialize_reg_values
    - print_reg_values
    - print_memory_values
    - step
    - run
- There are .cpp files specific to each of the instruction types - **R, I, S, B, J** and **U**. Each file has its corresponding format specific hexcode generating function which takes the instruction as argument and returns the hex code of the instruction. Each file has its corresponding format specific execute function which takes the vector containing the different parts of the parsed instruction as argument and updates register values, memory (as required) and returns offset by which the PC should update.

1.2 Implementation (Explanation of Code)

- This project uses few global variables which are extern to access across multiple files. They are:
  - **lineNo** : which stores line numbers from the start of the text section i.e, has starting value as zero. It is used to calculate PC value.
  - **data_mem** and **text_mem** : It is a map of 'int' to 'char' where the key value indicates the address location and value indicates each hex character it is storing i.e, each map location has a hex character.
  - **reg_value** : It is a map from binary representation of that register number to the long int it is storing.
  - **Ins_lines** : It stores each instruction from the .text section in a vector of string type.

- **Label** : Vector of label info(which contains name of the label and its corresponding line number)
- **break_points** : It is a map from int to bool. It is the map from the line number of the line where break point is present to where it has been executed or not. Bool value is updated accordingly.
- **data_line_count** : stores the line count of the .data section. Used for locating and setting breakpoints.
- **load_error** : It is used as an indicator for any errors. It is = 0 if no errors are there. It becomes 1 when error occurs while loading, -1 when we try to 'run' etc. without loading the file, -2 when the .text section (read-only memory) is being written to by the program, -3 when incorrect instruction address is being accessed.

- The functions used in **main** function while loop is running infinitely are:
  - In **'load file'** functionality : It gets the file name from the user. It generates hex code in output.hex file in hex format. If any errors in a given input file, it outputs them in both output.hex and also in terminal and tells the user that **"Input file given has error. Please load another file."** In this load part itself, it ensures that all data, text memory are empty and all register values are initialized to zeros and if no errors in output.hex file, then text memory stores this hex code and .data section provided values get stored in the data memory.
  - When other commands are used even though the input file has errors, it will inform the user about the error in the input file till it is resolved.
  - In **'run'** functionality : It runs till it reaches any breakpoint or executes all the instructions. It internally calls a **step** function to execute each instruction. It updates registers and memory.
  - In **'regs'** functionality : It prints all the values of registers in hexadecimal format.
  - In **'exit'** functionality : It gives a message **"Exited the simulator"** and gracefully exits the program.
  - In **'mem'** functionality : Depending on the address they gave, it prints the values of that memory address for the count they asked for. It can print for both text and the data section.
  - In **'step'** functionality : Depending on the instruction name of the instruction we need to execute, we call that instruction specific execution function which updates the register values and memory values if any and it returns a line number by which the global line number has to be incremented. Since this line number points to an instruction amongst all the instructions that are stored, it goes to that instruction to execute next. It also prints the instruction that just got executed along with PC value.
  - In **'break'** functionality, i.e, at line number at which break point should be applied, it stores all of them in a map. When reaching that break point for the first time while running it will pause execution over there but after running now again, it will continue its execution.
  - In **'del break'** functionality, it deletes the breakpoint if present otherwise tells the user that **"Breakpoint is not set at that line"**.
  - In **'show-stack'** functionality, it prints the current status of the stack. Note: stack is updated with a new function call only when the jal is hit.

- We changed the implementation of the generation of the hexcode from the previous lab assignment. Previously parsing of the string instruction was done in their respective format functions and the hexcode generation happened in the main function itself. But for clarity and reusablity of parsing in the simulator related code as well we shift the code of parsing different instructions to a different function 'parse'. For similar reasons we created a different .cpp which contains 'machine_code' function and the 'store_in_memory' function and all the code dealing with the hexcode generation was shifted here. We also added functions to load the .data section values in memory.

- **R_execute**

  - The above function takes the vector containing the pased parts of the string instruction.
  - It accesses the values of the source registers and performs the respective operations. It then stores the computed value in destination register.
  - If destination register is x0, we simply return (as x0's value cannot be changed).
  - Since our registers is implemented as a map of string to long ints, where the value stored in register is the long int value. Most operations can be done directly.
  - For srl, first we are bitwise shifting by right by number of bits required. Then we are doing bitwise 1and' with the number that has that many 0 bits in front.
  - for shift operations since the values in the registers can be negative or greater than 64, we are doing modulus 64 with it. To prevent any overflows we add 64 to it and then do modulus once more.

- **I_execute**

  - Similar to R_execute it handles the addi, xori, ori, andi, slli, srai, srli.
  - But here if destination register is x0 and the instruction is jalr, we do not directly return. Only when we come to the part of actually writing the value in x0, do we return.
  - For the load instructions, we are accessing the address using a relative difference with 0x10000, as for data memory we have a different map. Since we are storing the values in memory character by character, for 1 byte we need to read 2 values. By this logic we are loading values.
  - For all load instructions if it is signed we expand the msb otherwise we put 0 in front of it.
  - For jalr, the lineNo is given offset such that it jumps to the address (register value + immediate). Also if call_stack is non-empty then the function is popped off it.

- **S_execute**

  - Arguments are similar to R and I_execute is passed here.
  - Since we have stored data in memory character by character, for every byte we are taking 2 characters.
  - In the case of memory being written to the .text section of memory. We assign -2 to load_error and return. After that nothing can be run unless a new file is loaded (since the current one erroneous).

- **B_execute**

- Here, since the values store in the register map is long int, we can compare the values directly as required from the instruction. ex] if beq is there, we can directly check if rs1_value == rs2_value.
- If the required comparison results to true then the offset is updated accordingly to go to the instruction as per the immediate value or label.
- If the comparison results to false we just return offset as 1.
- If we are going to an instruction which does not exist i.e. either lineNo is negative or greater than the last instruction line number, then load_error is assigned -3. Then, unless another file is loaded, the code wont proceed.

- **U_execute**
  - The immediate value, if not in hexadecimal, is converted to hexadecimal. Then at end 3 '0's are added.
  - After this we compute the value of resulting hexadecimal string in decimal base, and store it in the destination register. If the destination register is x0, we directly just return.

- **J_execute**
  - The address of the next instruction is stored in the destination register (unless x0).
  - Also the label is pushed to stack, if we are jumping to a label. As this represents a function call.
  - The offset which we need to return to update the lineNo is calculated. If there is label then we calculate the offset using the lineNo of label line, current line. If it is immediate value then appropriate decimal value is retrieved. Finally we return the offset.

2 Testing

- For the simulator part:
  - We checked all types of instructions, if they were correctly updating respective registers, memory values. We compared our answers with Ripes to verify.
  - We checked that for all values of registers, the value is printing properly in hexadecimal. We even checked for maximum and minimum possible values (the edge cases)
  - For updating and printing memory values, we checked with multiple cases of how it is being updated and printed.
  - We checked for cases when text memory section could be written by the code. In such cases we print an error message.
  - We checked for when out of range values and the limiting (for respective datatype) when written in the .data section. And error is thrown when values are out of range.
  - We checked for when input file cannot be opened, in such cases appropriate error message is printed.
  - We checked running the breakpoints for normal cases as well as when it could be for a line exceeding the total lines. In such cases, error message is printed.

- We checked for deleting breakpoints, when it could be deleting a non-existent breakpoint and handled it by giving appropriate error.
- We checked show-stack for different jal, jalr jumps and returns and also for when we reach end of program.
- We checked for run and step that proper instructions are being executed, for when we reach end of the execution
- We ran our code for some of the previous lab assignment that we had submitted (like the gcd code) and verified the working by comparing with ripes.
- We checked for erroneous commands that can be run by the user like:
  * running any command before loading the input file.
  * If any error in input file, you are required to load another file. If without loading any other command (except for exit) is tried, it gives error message.
  * If while running the code, read-only memory is being attempted to be written, then error message is printed, and unless new file is loaded any other command (except for exit) does not run.
- We have included our test cases in the **Test Cases 2** text file. This file contains all the incorrect and edge test cases as well.

- For assembler part, we rechecked all the testcases from the previous Lab, Lab3:
  - We checked different types of inputs for each set of same format instructions
  - We checked for cases where the input format is not followed such as improper spacing, commas.
  - We checked for incorrect instructions like -
    * Blank lines
    * Incorrect instruction names
    * Incorrect number of operands
    * Incorrect type of operands - e.g.] instead of immediate value a register is given
    * Invalid Registers
    * Invalid Immediates (both decimal and hexadecimal immediates)
    * Jumping to Non-existing labels
    * A Label name having multiple definitions and it being used
    * Out of Range immediate values (we also tested all the edge cases)
    * We checked for when negative immediate values are given to an instruction that takes only positive values
  - We have included our test cases in the **Test Cases** text file. This file mainly contains all the incorrect and edge test cases.

3 Facilities

- We support all types - R-Type, I-Type, S-Type, U-Type, B-Type, J-Type of instructions
- We support all the functionalities - load, run, regs, exit, mem <addr> <count>, step, show-stack, break <line>, del break <line>.

- Whenever there is an erroneous line in the input file, the error with the line number is printed on the terminal. The error is also printed on the corresponding line in the output.hex file (except for the case of .data section errors)
- The code runs through the entire input file while loading it even when it encounters an error. So the errors are given all at once.
- For the .text section (the instructions) we facilitate both hexadecimal immediate values and decimal immediate values as input.
- For the .data section facilitate both decimal values as well as non-negative hexadecimal values. The hexadecimal number can have both uppercase and lowercase for the non - digit characters.
- We facilitate both labels and immediate values (decimal or hexadecimal) as input for branch or jump instructions.
- We facilitate storing of hexcode in the text section of memory and also the facility to view it.
- The Error message (both on the terminal and the output file) clearly states why that error is coming. So, it becomes easier for the user to fix their code.

4 Limitations

- It does not facilitate negative hexadecimal values for the .data section.
- There is strict format rules regarding spacing, commas, colons, blank lines. The program is not flexible enough to accept them as correct input and give out the machine code or to execute then. If the user does not follow the input format, an error message is printed and the program goes to the next instruction (if any).
- It facilitates only RISC-V (RV64I) variant assembly code.
- It does not support comments and pseudo instructions.