Computer Architecture Assembler Report

Nishi Baranwal, CS23BTECH11041
Paidala Vindhya, CS23BTECH11044

Contents

1 Coding Approach 1
1.1 Design Decisions . . . 1
1.2 Implementation (Explanation of Code) . . . 3
2 Testing 5
3 Facilities 5
4 Limitations 6

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.
- **assembler.h** header file consists of all the function declarations which are written in other .cpp files. The header file is then *included* in the .cpp files
- Reading input file and storing those lines in a vector. It helps to know on which line the label was defined.
- We are storing the label information such as label name and on which line it was defined in a vector of type **label_info** to facilitate calculation of the offset for branch or jump instructions.
- For immediate values, it facilitates both decimal and hexadecimal numbers.
- For branch and jump instructions, we are allowing both labels and offsets of decimal or hexadecimal type.
- If there is any error in the input line, it prints the 'Error message' specifying what the error is. It outputs the error both in **output.hex** file and in the **terminal**. On the terminal it also gives the line number.

- Used 'string' library, for concatenation of strings, finding any character, extracting sub-string of string and erasing some part of string etc.
- Created some general functions in the **gen_func.cpp** file such as -
  - findformat
  - bin_to_hex
  - deci_to_bin
  - regname_to_binary
  - immediate_binary
  - hex_to_bin
- 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 function which takes the instruction as argument and returns the hex code of the instruction. If it encounters any error, it returns the error to the main function.
- Program flow:

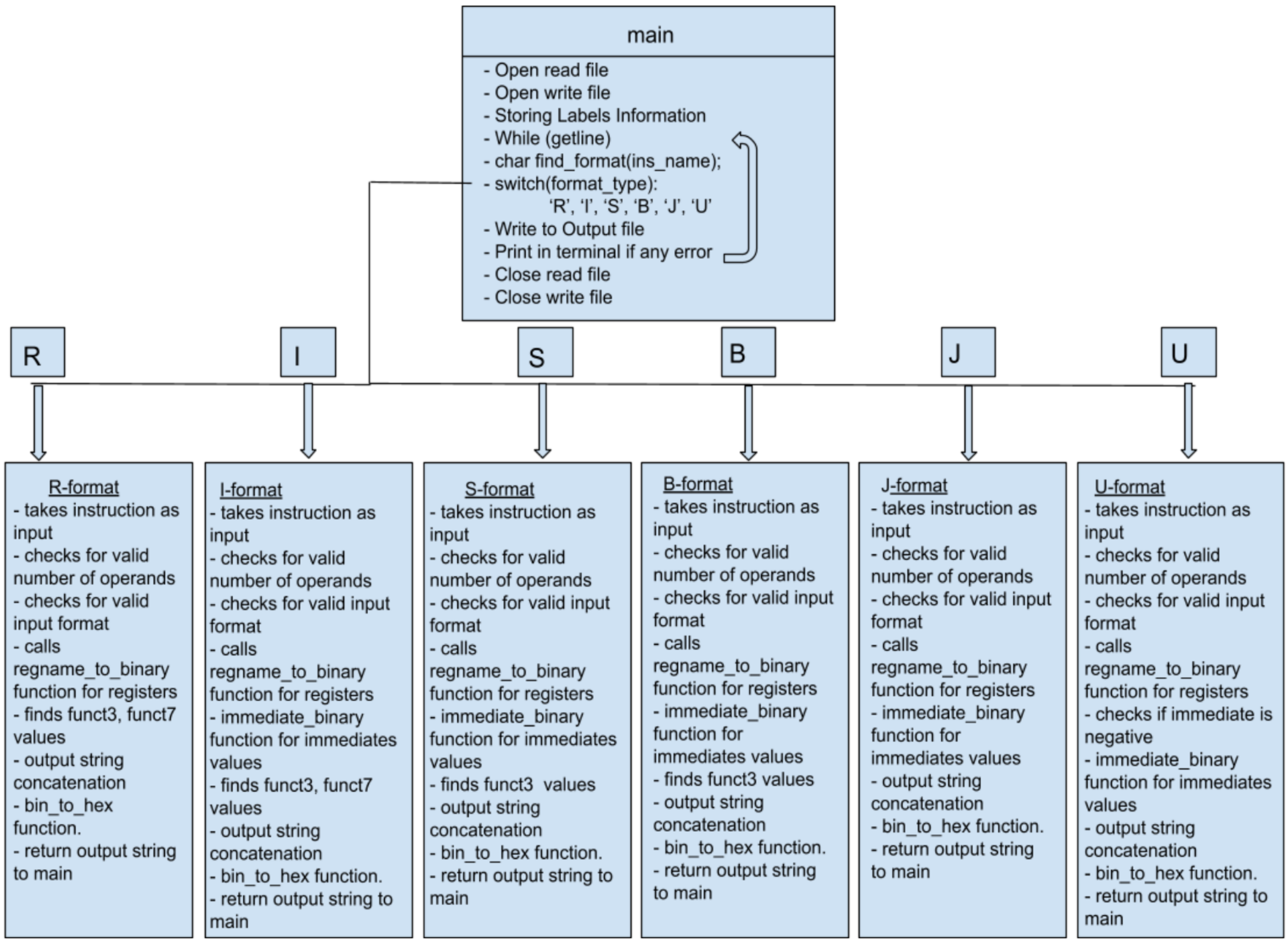
main
- Open read file
- Open write file
- Storing Labels Information
- While (getline)
- char find_format(ins_name);
- switch(format_type):
'R', 'I', 'S', 'B', 'J', 'U'
- Write to Output file
- Print in terminal if any error
- Close read file
- Close write file
R
I
S
B
J
U
R-format
- takes instruction as input
- checks for valid number of operands
- checks for valid input format
- calls regname_to_binary function for registers
- finds funct3, funct7 values
- output string concatenation
- bin_to_hex function.
- return output string to main
I-format
- takes instruction as input
- checks for valid number of operands
- checks for valid input format
- calls regname_to_binary function for registers
- immediate_binary function for immediates values
- finds funct3, funct7 values
- output string concatenation
- bin_to_hex function.
- return output string to main
S-format
- takes instruction as input
- checks for valid number of operands
- checks for valid input format
- calls regname_to_binary function for registers
- immediate_binary function for immediates values
- finds funct3 values
- output string concatenation
- bin_to_hex function.
- return output string to main
B-format
- takes instruction as input
- checks for valid number of operands
- checks for valid input format
- calls regname_to_binary function for registers
- immediate_binary function for immediates values
- finds funct3 values
- output string concatenation
- bin_to_hex function.
- return output string to main
J-format
- takes instruction as input
- checks for valid number of operands
- checks for valid input format
- calls regname_to_binary function for registers
- immediate_binary function for immediates values
- output string concatenation
- bin_to_hex function.
- return output string to main
U-format
- takes instruction as input
- checks for valid number of operands
- checks for valid input format
- calls regname_to_binary function for registers
- checks if immediate is negative
- immediate_binary function for immediates values
- output string concatenation
- bin_to_hex function.
- return output string to main

Figure 1: Flow of functions

1.2 Implementation (Explanation of Code)

- For reading or writing to a file, we are using the 'fstream' library.
- If initially there is no "**output.hex**" file, it creates one. Later whenever we make changes to the input.s file, it overwrites the existing output.hex file.
- Reading the input file into **vector<string>**, to analyze the label locations so that whenever branch or jump instructions use label names, we know the corresponding offset value.
- **vector<label_info>** stores the label names and line numbers of those corresponding label definitions. Here the **label_info** is a struct having data members - name and line_no (line number).
- **findformat** function : It takes the instruction name as argument and returns the format type as character. Ex : For add → 'R' ; beq → 'B' etc.
- Based on that particular format type, we are calling that specific type of function using switch cases of format character which returns the hex code of that instruction. If no such instruction name exists, it gives an error message as *"Incorrect Instruction Name"*. If blank line is present, it gives error *"Blank line"*
- If there is any error in that particular format type or number of arguments or immediate value exceeding the range, it stores that error message in hex code and displays that error in both output.hex file and in the terminal.
- For the formats **R, I, S, U** - we are directly calling those respective format functions but for **B** and **J** formats, we are modifying the labels into decimal offsets if the label exists. If the offset is directly given as a decimal or hexadecimal immediate then that format specific function is called directly.
  If it has a non existing label then it gives the error message that *"Label does not exist"*. If the label is defined multiple times, it gives an error message saying *"Multiple definitions of label"* in the lines where the label is defined and also where that label is used.
- All these format specific functions take the instruction string as an argument.
  - If there are incorrect number of operands (i.e registers and immediates/offsets combined), it gives an error message as *"Instruction expects 3 arguments"* except for the 'jal' and 'lui' instructions. For the later one's , it gives the error *"Instruction expects 2 arguments"*.
  - If the instruction is not in a valid format i.e, if there are no proper spacing etc., it gives an error message as *"Incorrect format"*.
- In these functions, it stores the instruction name, register names and immediate values (if any) in variables such as 'ins', 'rd', 'rs1', 'rs2', 'imm'. Using the 'substring' member function, these variables are assigned from the instruction string.
  - We pass the register variables to the function 'regname_to_binary' which returns a binary string of that register which is assigned back to the same register variable.
  - If that is not a valid register then it gives an error message as *"Invalid register: reg_name"*.

- For finding 'imm' variable in binary format, we are passing this as an argument to the 'immediate_binary' function. Along with this, we are also passing the size of the binary code it has to result in for that instruction (like 12 for I-format(except shift instructions), 21 bits for J format etc). This function returns the binary format of the immediate value in the specified bit size.
- If the given number cannot fit in that size, it returns an error message as *"Immediate value does not fit in specified_size bits"*.
- If the given 'imm' value is not a valid immediate value, then it gives an error as *"Invalid Immediate"*.

- **regname_to_binary** function : It takes the register name as argument. It can be a proper register name or alias name or any invalid register name. It converts alias names to proper register names. Depending on whether that register is empty or if it does not contain any valid register name, it gives an error message as *"Invalid register: reg_name"* . If that is a valid register, then we are calling the **deci_to_bin** function which takes the register number and size of the binary code it has to return as arguments. Since this deci_to_bin function can return binary conversions of both signed and unsigned decimal numbers, we are passing size as 6. So it returns a binary code of 6 bits with MSB as zero. So we have to erase the first bit and return the resulting 5-bit binary string.

- **deci_to_bin** function : It takes the decimal string value(can take both positive and negative decimal numbers) and size of the binary code it has to return(number of bits it has to be represented in), as arguments and returns a binary string of that. If that given decimal number cannot be represented in the given number of bits size, it returns as *"Error: Out of bound"*. It returns the 2's complement binary string of specified size.

- **bin_to_hex** function : It takes a 32 bit binary string and returns a hexadecimal string of that. It groups 4 digits from start and finds equivalent hexadecimal characters and concatenates them.

- **immediate_binary** function : It takes the immediate/offset string and the bit size it has to fit in, as arguments.
  - If the given immediate is hexadecimal, it calls the **hex_to_bin** function. If that hexadecimal does not fit in the specified bit size, then it returns *"Out of bound"* or if it is an invalid hexadecimal number then it returns *"Invalid Hexadecimal"*.
  - If it is a decimal number then it calls the **deci_to_bin** function. If that decimal does not fit in the specified bit size then it returns *"Immediate value does not fit in specified_size bits"*.
  - If the given immediate is not a valid number, then it returns an error as *"Invalid Immediate"*.

- **hex_to_bin** : It converts hexadecimal string to binary string. First, it converts the hexadecimal to a decimal number by the addition of product of hex digit with 16 raised to the power of bit position of that hex digit. Now passing this decimal number as an argument to the **deci_to_bin** function which returns the binary number. If it doesn't fit in the specified size, it gives the error *"Out of bound"*.

- **R_format, I_format, S_format, U_format, B_format, J_format** functions :
  In these functions, after finding binary representations of register names, immediate values (if any) and opcode, it gets their funct3, funct6 or funct7 values in binary format specific

to that instruction name. For ‘lui’ and shift instructions, since they take only positive immediate values, while calling **deci_to_bin** function we are passing size as 1 more than required. Thus, it returns one extra bit (MSB) which is then erased. Then it does string concatenation of these binary strings in the corresponding specific order for that instruction and passes this 32 bit binary string to the **bin_to_hex** function which returns the required hex code. The hex code is then returned back to the main function.

2 Testing

- 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
- Whenever there is an erroneous line in the input.s 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
- The code runs through the entire input file even when it encounters an error. So the errors are given all at once.
- We facilitate both hexadecimal immediate values and decimal immediate values as input.
- We facilitate both labels and immediate values (decimal or hexadecimal) as input for branch or jump instructions.
- 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

- 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. 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.