

# CS2323 Lab4 REPORT

Pakki Sai Kaushik - AI23BTECH11017  
Samagalla Sukesh Kumar - AI3BTECH11024

## Contents

1. <a href="#">Introduction</a> .....	2
2. <a href="#">Implementation</a> .....	2
(a) <a href="#">Overview</a> .....	2
(b) <a href="#">Implementation</a> .....	2
3. <a href="#">Assumptions or Limitations</a> .....	4
4. <a href="#">Challenges arised and Excess Learning</a> .....	5
5. <a href="#">Additional Context and Considerations</a> .....	5

# 1. Introduction

Welcome to a detailed report on a C program which simulates a mini simulator(when executed) to be used while executing an Assembly instruction(or an assembly line code in a file(strictly adhering to the **RISC-V ISA**)). The code is human-readable to interpret the logic behind it by simply reading the code files.

## 2. Implementation

### (i) Overview

The Zip contains a list of files :

- **Lab4.h** :a header file which contains the list of function declarations and global variables such as `Jal_r_return` , `offset_global`, etc.
- **helpers.c** : a '.C' file containing the function definitions of a few functions that can handy while programming.
- **commands.c** : a '.C' file containing the function definitions of the functions used to handle input commands.
- **ins\_fun\_RI.c** : a '.C' file containing the function definitions of the functions which mimic the functionality/property of RISC-V instructions of type **R** & **I**.
- **ins\_fun\_BJ.c** : a '.C' file containing the function definitions of the functions which mimic the functionality/property of RISC-V instructions of type **B** , **J** & `jalr` instruction.
- **ins\_fun\_ULS.c** : a '.C' file containing the function definitions of the functions which mimic the functionality/property of RISC-V instructions of type **U** , **S** & **load** type in **I**.

The list of functions and it's functionality is explained in detail in upcoming pages of this document.

Two structures are defined:

- **Instruction**: Stores the array `instructions[]` of instructions(their names) along with their type.
- **Label**: Stores the labels and their line numbers, their relative **offsets** from the starting of the input document(an important part of this project!!) from the input file into an array `labels[]`.
- **Stack\_Calls**: stores the **name**, **position of calling**, **top index** of the stack.

### (ii) Implementation

moving onto details of the code, The following headers are included:

- **stdio.h**: For input and output streams.
- **stdlib.h**: For functions like `atoi()` and `strtol(,,)`.
- **ctype.h**: For functions like `isxdigit()`.
- **string.h**: For functions like `strstr(,,)`, `strchr(,,)`, `strcmp(,,)`, `strcpy(,,)`, and `strlen()`.

- `stdbool.h`: For the `bool` data type and constants like `true` and `false`.
- `stdint.h`: for using pre-defined datatypes like `textbfint32_t`, `int64_t`, etc.

## Structures Defined

- **Instruction:**

```
typedef struct Instruction
{
    char name[7]; // name of instruction
    char type; // Type of instruction
};
```

- **Label:**

```
typedef struct Label {
    char name[21]; /*name of the label , assumed to have atmost 20
    characters*/
    int32_t line_num;
    int64_t offset;
};
```

Stores("append" in strict sense) label names, their line numbers from `.text` section and their offsets(the number of characters from the start of the file so that we can point to any character of the file using pre-defined function like `fseek(, , )`).

- **Stack\_Calls:**

```
typedef struct Stack_Calls
{
    char name[30][20];
    int pos_of_calling[30];
    int top_index;
} Stack_Calls;
```

- **Register\_list:**

```
typedef struct Register_list
{
    char reg_name Og[4];
    char reg_name_alias[5];
    int64_t reg_value;
    int64_t jal_offset;
} Register_list;
```

- **add\_label function:**

```
void add_label(char *name, int64_t line_num , int64_t offset);
```

Adds a found label and its line number of the occurrence of that label to the `labels[]` array.

- **find\_label\_address function:**

```
int find_label_address(char *name);
```

Returns the offset of a label if found in the `labels[]` array or returns `-1` if not found, also updates the PC value of the destination.

- **Other Functions:**

- `bool is_hexadecimal(char *str);` Checks if a string is a valid hexadecimal.
- `void add_label(char *name, int64_t line_num, int64_t offset);`
- `void execute_instruction(char Instruction[], int32_t id, char rd[], char rs1[], char imm_str[], char line[]);`
- `void initialization();` initializes all global variables and stack values to their respective initial values.
- `void data_segregation();` segregates all the memory values into the respective array.
- `void PC_check();` checks if EOF is reached , else updates the PC.
- `int32_t assign_reg_val(char reg[]);` returns the index of the register to be used.
- `int32_t get_instruction(char *str);` returns the index of the instruction.
- `int64_t get_reg_val(char reg[]);` returns the register value of the register `reg[]`.
- `bool break_check();` checks if break statement is reached.
- `void separator(char *str);` separates each command to be given in terminal.
- `void show_stack();` functionality of a stack displayer.
- `void stack_update(int64_t dummy);` updates the stack.
- `void Pop_Stack();` pops the top of the stack.
- `void Push_Stack(char name[], int pos);`
- `void Print_Memory(char mem_start_addr[], char count_in_str[]);` prints the memory values from 0x10000 till what is requested.
- `void display_register(void);` displays the register values of all 32 registers.
- `void run(bool flag);` a run function that runs based on flag. if flag is `true`, full run. if false, only single step run!
- `void load_file(char file_name[]);` loads the required file to execute.
- `void set_break_point(int64_t line_num);` sets break point at required line number.
- `void del_break_point(int64_t line_num);` deletes the break point at required line number.

## 4. Assumptions or Limitations

- data is only specified after `.dword/.word/etc..`
- A variable `MAX_Labels` is defined to be 10000 using a macros. and the number of characters to define a label is restricted to 20. Hence, any label having more characters than 20, and having more than 10000 labels may result in a segmentation fault.
- The Typical format of `jalr` instruction is taken to be `jalr rd, n(rs1)` which is the traditional notation way of writing a `jalr` instruction which differs from that of RIPS simulator which is `jalr rd, rs1, n`.
- In case of multiple label definitions, for calculating the immediate value of a 'B' or 'J' type instruction, the line number of first occurrence of the label is used while reading the file from top to bottom. For remaining labels having the same name as of the previous one, the terminal shows the error for repeating the label but does print the hexadecimal machine code for the instruction followed by that label definition.

- **Line numbers:** it has been made an assumption that when `show-stack` is invoked, the number to be printed after " : " is nothing but the line number which has been counted from the starting of the input file given.

**But** if the user wants to have that number printed, without actually counting from the start of file, but actually from the assembly code, exchange the commented lines respectively ,

```
printf("main:%ld\n",dot_line_num);
//printf("main:0\n");
```

```
printf("%s:%d\n",Stack.name[i],Stack.pos_of_calling[i]+dot_line_num);
//printf("%s:%d\n",Stack.name[i], Stack.pos_of_calling[i]);
```

- **The code assumes that the input file has no logical/syntactical errors in it.**
- For I-type instruction(except the **load** part), the immediates are to be taken only in Integer-type
- During branches/ jumps / jump returns, only label names are to be given.
- Immediate values for U-type and B/J-type instructions are assumed to be integers unless otherwise specified.
- The supported RISC-V assembly instructions include `add`, `sub`, `and`, `or`, `xor`, `sll`, `srl`, `sra`, `addi`, `andi`, `ori`, `xori`, `slli`, `srli`, `srai`, `ld`, `lw`, `lh`, `lb`, `lwu`, `lhu`, `lbu`, `sd`, `sw`, `sh`, `sb`, `beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`, `jal`, `lui` and `jalr`  
**NOTE:** *pseudo instructions are not supported!*

## 4. Challenges Arised and Excess Learning

This assignment introduced challenges like:

- Extensive use of structures to store complex data.
- File handling functions such as `fgets`, `sscanf`, and `fseek`.
- all Bitwise operations have to be learnt
- Familiarization with string and character handling functions like `strcpy`, `strcmp`, `strlen`, `isxdigit`, `atol`, and `strtol`.

## 5. Additional Context and Considerations

**Number of lines:** Our C-program is typically capable of producing desired output even with excess of input instructions( i.e. atmost 10000 number of lines of assembly code in `input.s` file.

**Challenges and Learning:** This lab assignment presented several challenges, particularly in the realm of bit-level manipulation and the correct positioning of bits for generating machine code. Specific difficulties were encountered with:

- Using bitwise shifts (`<<`, `>>`) and logical operations (`|`, `&`) , etc.
- Handling label-based branches in B-type and J-type instructions by calculating the offset from the current line to the line where the label occurs.

**New Concepts Learned:** Through this assignment, the following advanced programming concepts were mastered:

- Handling and manipulating structures to store data with multiple attributes.
- Extensive use of file handling functions in C, including `fgets`, `sscanf`, and `fseek`, to read from and write to files.
- Bitwise operations for manipulating data at the binary level.
- String handling functions such as:
  - `strcpy`: To copy strings.
  - `strcmp`: To compare two strings and return whether they are identical.
  - `strlen`: To calculate the length of a string.
  - `strstr`: To find a substring within a string.
  - `strchr`: To locate the first occurrence of a character within a string.
- Character checking functions such as:
  - `isxdigit`: To check if a character is a valid hexadecimal digit.
- Number conversion functions such as:
  - `atol`: To convert a string to a long integer.
  - `strtol`: To convert a string to an integer or hexadecimal number, depending on the base provided.

In summary, this lab gave good learning exercise, particularly in handling low-level operations, file I/O, file handling, and structure manipulation in C programming.