<u>LAB ASSIGNMENT – 3 – ASSEMBLER</u>

<u>REPORT</u>

The RISC-V Assembler Project is a C-based program that translates RISC-V assembly language code into machine code. This project serves as a foundational tool for understanding the assembly language and its corresponding machine representation. It aims to provide users with a practical understanding of the translation process, enabling them to work with low-level programming efficiently.

HEADERS AND FUNCTION PROTOTYPES:

I have created separate files for each of the instruction type, namely R, IS, B, U, J which has the function for processing each instruction type. So, the headers of all these files are included in the final.c file. The function prototypes of all the functions inside the files are also written before the main function.

MAIN FUNCTION:

The main function serves as the entry point of the program. It begins by opening the input file, input.s, in read mode, and then opens (or creates if it doesn't exist) the output.hex file in write mode to store the translated machine codes.

Once the files are opened successfully, the main function calls the process_instr function, which handles the translation of instructions from assembly to machine code. After the translation process is complete, the input file is closed, ensuring proper cleanup and termination of the program. The entire process ends upon successful generation of the output.hex file.

PROCESS_INSTR FUNCTION:

The process_instr function is responsible for reading and processing the assembly instructions from the input file line by line. It follows a two-pass process, with the first loop scanning for labels and recording their addresses, while the second loop translates the instructions into their corresponding machine codes. Here's a breakdown:

<u>First Loop: Label Detection</u>

The first loop goes through each line in the file to find labels (marked by a colon :).

If a label is found, the add_label function is called, arguments the label found along with the current address (curaddr), which is incremented by 4 for each instruction as RISCV instruction takes 4 bytes.

The add_label function is responsible for this. It does the following:

- If there is already a label with the same label name, it adds the current address (i.e, the address of the duplicate label) into an array named dup.
- Otherwise, it adds the label name and its corresponding address to the array of Label structs, named labels.

Each Label struct contains the label name (label[]) and its associated address (address), which are stored in the labels array. This allows for easy lookup and reference of labels during the second loop.

<u>Second Loop: Instruction Processing</u>

After the labels have been scanned, the second loop begins processing the actual instructions in the file. The steps in this loop are as follows:

- Line-by-Line Reading: Each line is read again, and the instruction is identified. The current address (curaddr) is updated for each instruction, incremented by 4 bytes for every line.
- Label Handling: If the line contains a label, the for loop checks if it's a duplicate label (i.e., if the label name has been used before). If a duplicate is found, a "Multiple label error" message is printed in the output file, indicating an error in label declaration.
- Instruction Identification: If the line does not contain a label, it is assumed to be a regular instruction. The program then skips the label check and moves to identify the specific instruction by checking the content of the instruction variable.
- Instruction Processing: For each recognized instruction (e.g., sra, sub, or), the corresponding function is called (e.g., process_instructionR) to handle the translation of that instruction into machine code. These processing functions generate the appropriate hexadecimal machine code for the instruction, which is eventually written to the output file.

This structured approach ensures that labels are accurately processed first, and then instructions are correctly translated into machine code according to their format.

INSTRUCTION TYPE FORMATS:

PROCESS_INSTRUCTIONR (instr_R.c):

- The function opens the output file output.hex in append mode to write the machine code. If the file cannot be opened, an error message is displayed.
- The sscanf function is used to extract up to three register operands from the assembly instruction.
- get_register_number function is used to convert the register names into their corresponding numeric values, which are needed for the machine code.
- The machine instruction is constructed using bitwise operations to combine the various fields (funct7, rs2, rs1, funct3, rd, and opcode) into a 32-bit instruction.

Like:
funct7 is shifted left by 25 bits (funct7 << 25),
rs2 is shifted left by 20 bits (rs2 << 20),
rs1 is shifted left by 15 bits (rs1 << 15),
funct3 is shifted left by 12 bits (funct3 << 12),
rd is shifted left by 7 bits (rd << 7),
The opcode (0b0110011 for R-type) occupies the last 7 bits.

- The generated machine code is written to the output.hex file in hexadecimal format, and the file is closed after each operation.

PROCESS_INSTRUCTIONIS (instr_IS.c):

- The function opens the output file output.hex in append mode to write the machine code. If the file cannot be opened, an error message is displayed.
- As the IS-Format instructions has 3 different opcodes, I divided the process into 3 if condition on basis of opcode:
    - **For arithmetic instructions** (opcode 0b0010011): These instructions have two operands (registers) and one immediate value. The instruction format involves the rd, rs1, immediate value, and opcode.
    - **For load instructions** (opcode 0b0000011): The format involves an offset (immediate), base register rs1, destination register rd, and the opcode.
    - **For store instructions** (opcode 0b0100011): The format involves splitting the immediate value into high and low parts, and then combining it with rs1, rs2, and the opcode.
- The sscanf function is used to extract up to two register operands and one immediate value from the assembly instruction.
- Bitwise operations are used to assemble the machine code:
    - For **arithmetic and load instructions**: The immediate value is shifted left by 20 bits, rs1 is shifted left by 15 bits, funct3 is shifted left by 12 bits, and rd is shifted left by 7 bits. The opcode occupies the last 7 bits.
    - For **store instructions**: The immediate value is split into high and low parts. The high part is shifted left by 25 bits, rs2 is shifted left by 20 bits, rs1 is shifted left by 15 bits, funct3 is shifted left by 12 bits, and the low part is shifted left by 7 bits, with the opcode in the last 7 bits.

The machine code is generated as a 32-bit instruction, which is written in hexadecimal format to the output.hex file. The file is closed after each operation.

PROCESS_INSTRUCTIONSRAI_IS (instr_IS.c):

The SRAI instruction is handled in a separate function because it requires a specific treatment for the immediate value, but has the same opcode value. So for better readability, unlike other IS-Type instructions where the immediate value is directly encoded, SRAI uses a special encoding:

- The upper 7 bits (specifically 0b0100000) indicate a right arithmetic shift.

- The lower 5 bits represent the shift amount.

The resulting 32-bit instruction is written to output.hex in hexadecimal format, and the file is closed.

PROCESS_INSTRUCTIONB (instr_B.c):

- The function opens the output file output.hex in append mode to write the machine code.
- sscanf reads the two register operands (rs1, rs2) and a label, which indicates the branch target. The branch offset is computed as the difference between the target label address (labeladd) and the current instruction address (curaddr), divided by 4 to account for the 4-byte instruction size.
- Bitwise operations are used to assemble the machine code:
    - The 12-bit signed immediate value is split into multiple parts: imm[12], imm[11], imm[10:5], and imm[4:1].
    - These are combined with rs1, rs2, funct3, and the branch opcode (0b1100011).
- The machine code is generated as a 32-bit instruction, which is written in hexadecimal format to the output.hex file. The file is closed after each operation.

PROCESS_INSTRUCTIONJ (instr_J.c):

- The function opens the output.hex file in append mode to write the generated machine code. If the file cannot be opened, an error message is displayed.
- J-Type instructions (such as jal) involve one register operand (rd) and a label for the jump target.
- The difference between the label address (labeladd) and the current instruction address (curaddr) is computed to determine the jump offset.
- The jump offset is a signed 21-bit value, and the function ensures it falls within the valid range (-2097152 to 2097151). If not, an error message is logged.
- The 21-bit offset is split into multiple parts: imm[20], imm[19:12], imm[11], and imm[10:1]. These parts are then combined with the destination register (rd) and the opcode using bitwise operations.
- The machine code is generated as a 32-bit instruction, which is written in hexadecimal format to the output.hex file. The file is closed after each operation.

PROCESS_INSTRUCTIONU (instr_U.c):

- This function is used to process U-Type instructions (such as lui or auipc), which involve a destination register (rd) and a 20-bit immediate value (imm).
- The function opens the output.hex file in append mode to write the generated machine code. If the file cannot be opened, an error message is displayed.
- It uses sscanf to extract two operands from the instruction: the destination register and the immediate value (in hexadecimal format). If more than two operands are provided, or if fewer operands are found, an error message is written to the output file.

- The register is cleaned by removing any trailing commas, and the immediate value is checked to ensure it fits within the 20-bit range (0 to 0xFFFFF). If the value exceeds this range, an error is reported.
- The instruction is constructed by shifting the 20-bit immediate value by 12 bits, shifting the destination register by 7 bits, and combining these with the opcode for U-Type instructions (0b0110111 for lui).
- The machine code is then written to the output.hex file in hexadecimal format, and the file is closed after the operation.

ERROR HANDLING FEATURES:

- When the output file output.hex is opened in append or write mode to store the machine code, the function checks if the file can be opened successfully. If it cannot, an error message is displayed on the console, and the program returns to prevent further execution.
- During label processing, if multiple labels with the same name are detected, an error is logged in the output.hex file. This ensures that duplicate labels are flagged and prevents further processing for that line. The execution stops there.
- When processing instructions, if an unrecognized instruction (i.e., not a valid RISC-V instruction) is encountered, an error message is printed in the output.hex file, including the line number where the error occurred. This allows the user to easily identify and correct invalid instructions.
- In each instruction-processing function (e.g., R-type, I-type, etc.), checks are in place to validate:
    - If the **register number** is not valid, an error message will be printed in the output.hex file.
    - The **number of operands** is correct for the given instruction format. If the instruction has too few or too many operands, an error is logged in the output file with details about the issue.
    - The **immediate value** is checked for validity, ensuring it fits within the allowable range for that instruction type. If the value is out of range, an error message is printed in the output.hex file.
    - In the case of branch (B-type) and jump (J-type) instructions, the function verifies if the label referred to in the instruction actually exists in the set of labels. If the label is missing or undefined, an error is recorded in the output.hex file, indicating the invalid label reference.

ASSUMPTIONS:

- The instructions are properly entered with valid spaces ie, there is only 1 space in between the instruction and the first operand. Also, only 1 space between "," and second operand and so on.
- Maximum number of labels in a file of instructions is assumed to be 100 and the maximum length of the label is assumed to be 50 characters length.
- The instruction inside the label is to be written in the same line.

- There are no blank lines in the input.s file.
- Pseudo Instructions are not supported for translation.
- EOL/EOF has to be properly entered.
- Immediate/offset values provided with instructions is assumed to be in decimal only.

FUTURE FEATURES:

- Comments: A line starting with a semicolon (;) can be treated as a comment and omitted. Similarly, anything after a semicolon (;) within a line can be omitted. This feature could be added.
- Pseudo Instructions: Support translation of specific pseudo instructions like mv, j etc.. This feature could be added.