Indian Institute of Technology Gandhinagar

ES 215
COMPUTER ORGANISATION AND ARCHITECTURE

COA Project Report Group - 5
Group Name: Assembly Alliance
Project Title: Assembler for the MIPS Processor

Group Members:

22110063 - Srijahnavi Chinthalapudi

22110069 - Deepanjali Kumari

22110083 - Siri Gangannagudem

22110293 - Vubbani Bharath Chandra

Under the Supervision of
Prof. Sameer Kulkarni

Github link : https://github.com/Siri-gangannagudem/COA_Assembler

Abstract:

This project aims to develop an assembler for an MIPS (Microprocessor without Interlocked Pipeline Stages) architecture, widely recognised for its simplicity and utilised in academic and practical contexts. The assembler we are making will translate MIPS assembly language programs into binary format and then to hexadecimal code. Developing this assembler requires an understanding of the processor's Instruction Set Architecture (ISA), as well as designing, implementing, and testing the assembler code. This assembler takes a file of the user's choice as input which contains assembly instructions in each line and then gives the outputs as hexadecimal machine code of each instruction. The type of instructions are taken from the MIPS reference sheet provided.

Introduction:

An assembler is a computer program which converts assembly language code into machine code. Assembly language is a human-readable intermediary between high-level programming languages like Python and machine code, offering a structured yet approachable way to interact with computer hardware. It uses mnemonics to represent the operations that a processor has to do. Machine code is the binary code that a computer's or a system's processor can directly execute. Different processors have different assemblers because each of the processors have their own unique Instruction Set Architecture (ISA), with different types of instructions. In this project, we are designing an assembler for the MIPS processor. MIPS architecture (Also known as Load/Store Architecture) is a family of RISC (Reduced Instruction Set Computer).

MIPS uses a simplified set of instructions. This makes it more efficient and easier to design than processors with complex instruction sets. The Underlying design principles, as articulated by Hennessy and Patterson: Simplicity favours regularity, Make the common case fast, Smaller is faster, Good design demands good compromises. To design an assembler for an architecture, knowing about the ISA of the processor is of utmost importance. MIPS instructions are divided into three main formats : R-type, I-type, J-type. The instructions are of fixed length (32 bits) and it has 32 "integer" registers and 32 "floating point" registers.

R-Type (Register Type): Used for arithmetic and logical operations. These instructions take operands directly from registers, and the result is stored in a register.
I-Type (Immediate Type): Used for instructions that require a constant value (immediate) or memory address calculations. I-type instructions include load and store operations, conditional branches, and arithmetic with immediate values.
J-Type (Jump Type): Used for jump instructions, which change the program control flow by setting the program counter to a new address.

Key Idea:

This assembler for the MIPS instruction set, converts assembly language programs with instructions into

8 digit hexadecimal (32 bit binary) instructions. Our assembler handles 49 instruction encoding requirements of the MIPS architecture, translating readable assembly instructions into binary format and later to hexadecimal format that the processor can execute.
In addition to converting instructions, the assembler processes labels, maps them to memory addresses or instruction offsets in the hexadecimal code. The assembler we developed performs a two-pass approach:

- **First Pass:** The assembler scans through each line of assembly code, storing addresses and corresponding instructions while cleaning up extraneous blank lines using Python's **"Strip()"** function.
- **Second Pass:** The assembler finalises the translation of assembly instructions into machine code by referencing the symbol, command, and address tables constructed during the first pass.

To ensure accuracy, we implemented a reporting feature that flags errors in the code. For example, it will alert users if a base address is invalid, such as 1001, which is not divisible by 4, ensuring alignment requirements are met. This project has equipped us with practical insights into MIPS architecture, instruction encoding, and binary generation, consolidating our knowledge from coursework and enhancing our low-level programming skills.

Implementation of the project:

Task 1: Research about MIPS ISA

- Research on the ISA of MIPS to understand instruction formats, addressing modes, opcodes and registers, and understand how to handle assembly syntax, labels, comments, and directives in the input.

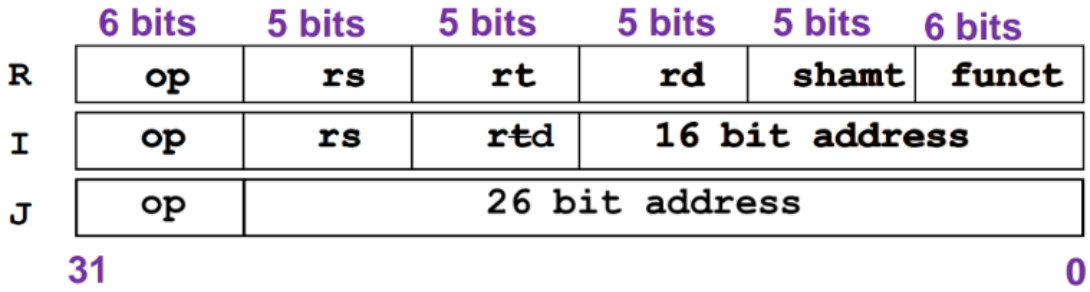
- We analysed the three primary instruction formats—R-type, I-type, and J-type—each with a distinct structure and operand requirements. This study encompassed various addressing modes used by MIPS, including immediate, register, and base addressing, to understand how operands are accessed within memory and registers.

- Further, we delved into MIPS opcodes and their role in defining instruction functionality alongside the architecture's register conventions. MIPS registers follow a straightforward convention, with designated purposes for general use, saved values, and specific operations like argument passing and return addresses.

- To handle assembly syntax accurately, we investigated MIPS syntax rules, which dictate operand order, spacing, and the handling of labels. Labels and comments were carefully reviewed, as labels enable branching and jumping, allowing for effective control flow, while comments ensure code readability. Finally, we explored assembler directives, which facilitate memory allocation, data initialization, and symbolic reference in the MIPS environment, thus providing a structured approach to writing and interpreting MIPS assembly code. This understanding forms a foundation for developing and validating assembly programs that adhere to MIPS standards.

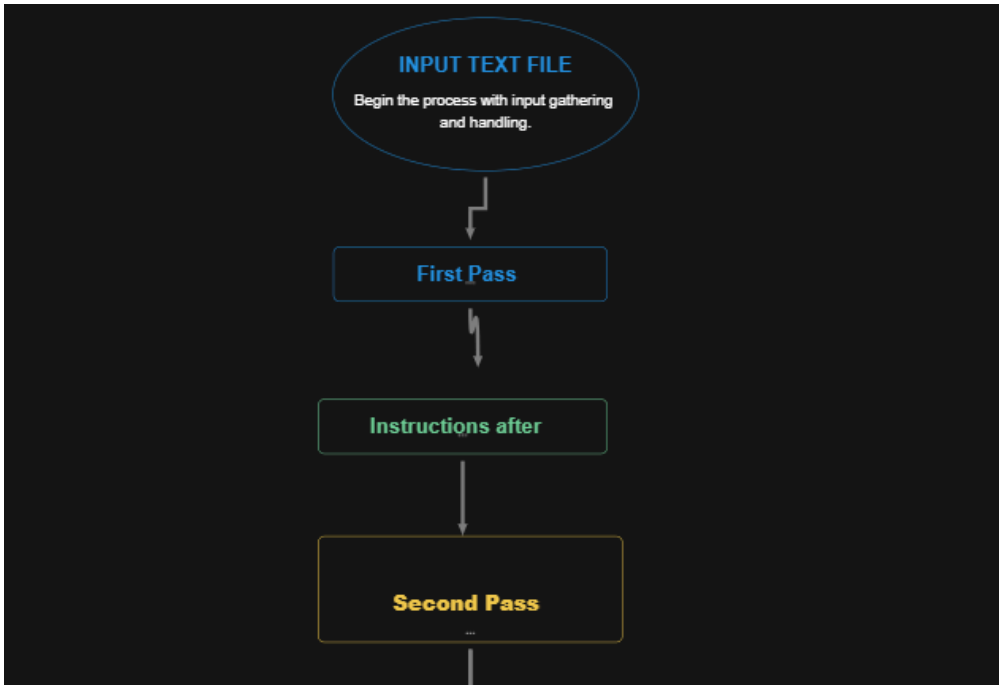
- We then also researched on how to convert these different types of instructions into their respective machine code. Each format has unique fields—like opcode, source and destination registers, shift amount (shamt), and function code (funct)—that define the 32-bit structure of the instruction.

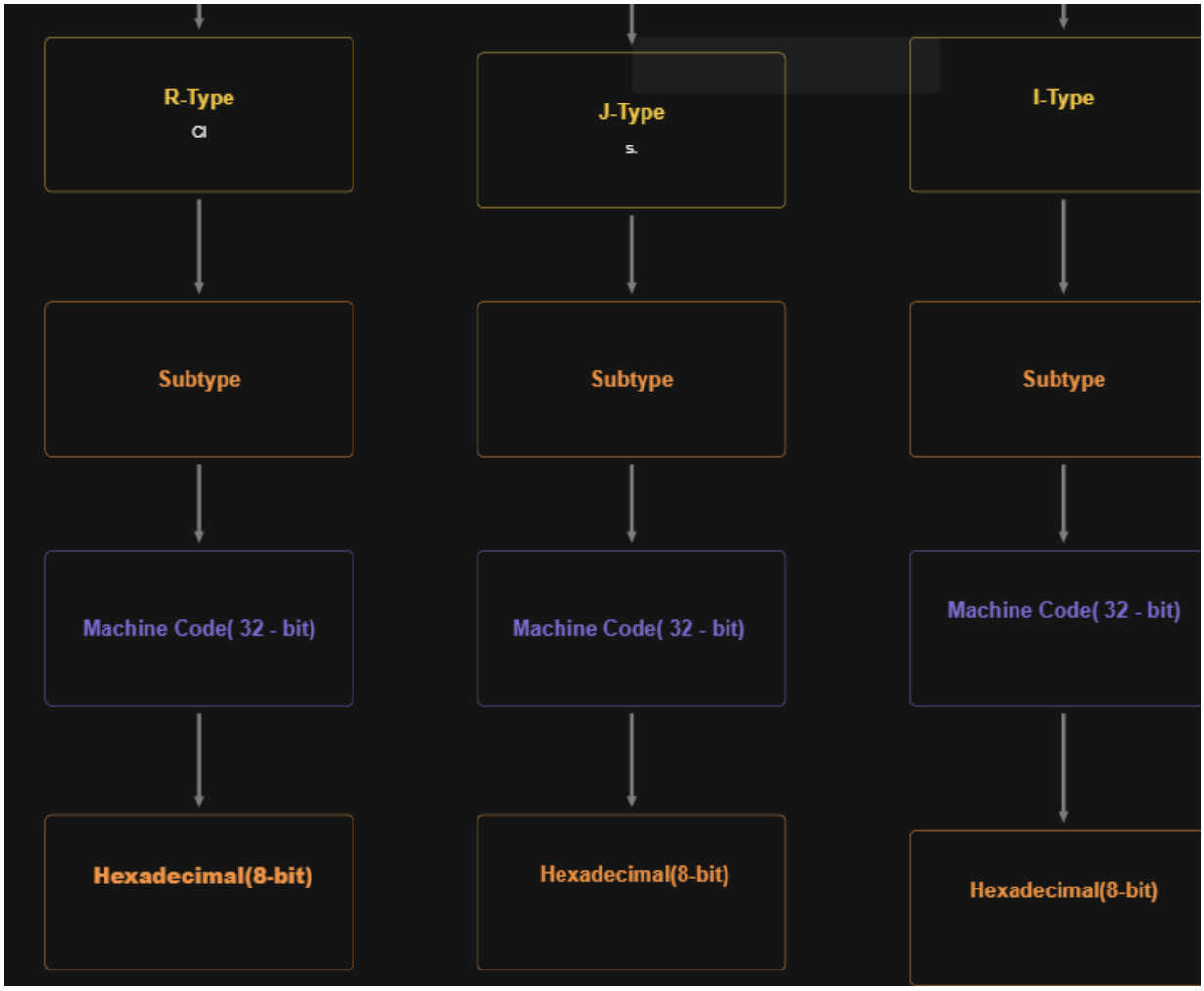
  The conversion of various formats of instructions in MIPS are given by the following figure,



  (Ref : Lecture slides)

Flow of the Code:





Task 2: First Pass Implementation:

First Pass: The first pass function in our MIPS assembler reads each line of assembly code from a user selected file and breaks it down into manageable components, or commands. It identifies and categorises them as instructions, labels, or operands, discarding comments, extra spaces, and blank lines in the process.

1. Comment Handling: If a # symbol appears anywhere in a line, the tokenizer ignores everything that follows in that line of input. This applies to both full-line comments (lines starting with #) and inline comments (code followed by # and comment text). Only the portion of the line before the '#' is processed as code, allowing for inline documentation without interference.

```
cleaned_command = re.sub(r'\s+', ' ', command.strip())
cleaned_command = re.sub(r'\s*,\s*', ', ', cleaned_command)
```

2. Label Identification: Labels, which serve as jump destinations within the code, are identified by a colon (`:`) at the end of the token (e.g., `LOOP:`). These are recorded in the symbol table with their

corresponding memory addresses to facilitate branching and jumping in the second pass.

3. Instruction Recognition: The first pass clear extraneous space in each instruction i.e, if there are any extra spaces between the operands or mnemonics (but not in between the mnemonics and operands - that causes an error). It stores them in a python list named “commands”.

4. Addresses: The address variable is initialised at 0, simulating the memory address of the first instruction. For each instruction encountered (whether it follows a label or stands alone), the address is stored and incremented by 4 to mimic MIPS's 4-byte instruction spacing. The “addresses” list has the addresses of each instruction stored in the command table. In the symbol table, it is a dictionary mapping labels and their address.

5. Labels are not stored in the commands, because they are not used for converting into machine code as only their addresses are required. If there are two labels with the same name, then the first pass gives an error as labels are supposed to be unique in the program for any jump or branch instructions.

6. Note : Label names should not have spaces, colon(:), or comma(,) in between them and Label names are case sensitive. Any mnemonic from the MIPS instructions should not be used as a label name. Mnemonics given in the input are not case sensitive (Example: ‘Add’ is considered as a valid mnemonic whereas ‘LOOP’ and ‘Loop’ are considered as two different label names).

   At the end of first pass, our output looks like this:

```
Enter the file path: Instructions.txt
Symbol Table: {'START': 0, 'loop': 12, 'siri': 20}
Commands: ['add $t0, $t1, $t2', 'lw $t0, 0($sp)', 'sw $t1, 8($gp)', 'beq $t0, $t1, start', 'j loop', 'add $t0, $t1, $t2']
Addresses: [0, 4, 8, 12, 16, 20]
```

The first pass function filters each line by stripping unnecessary elements, and organising these instructions, labels and their addresses in a structured format. By the end of the first pass, the assembler has a complete symbol table with all label-to-address mappings and a structured list of instructions, each linked to a memory address. This layout sets up the assembler for a smooth second pass by isolating essential components.

Task 3: Second Pass Implementation:

The main part of the assembler is the second pass as it plays a crucial role in converting the assembly instructions into machine code and also identifies different types of errors in the input so that the user can correct them.

During the instruction encoding phase of the second pass, the assembler converts parsed assembly instructions into binary format according to the MIPS Instruction Set Architecture (ISA). It identifies the type of instruction—R-type, I-type, or J-type—and maps each opcode to its corresponding binary representation. For R-type instructions, the assembler constructs a 32-bit word by combining the opcode, source, and destination registers, shift amount, and function code. For I-type instructions, it encodes the

opcode, source, and destination registers, along with the immediate value. J-type instructions are encoded with the target address. This transformation is essential for generating machine-readable binary instructions, enabling execution by the MIPS processor.

After identifying the mnemonic (once it is valid, if its not valid an error is raised telling the user that the mnemonic of that particular command is invalid along with the line in which the error occurred) and classifying it as one of the above mentioned type, the second pass does the following respectively:

R-Type Instructions:

1. Identifying the R-format type:
    a. Based on the instr_format (identified from the instruction_set table), the function proceeds to handle R-format instructions, categorizing them further by their specific operation (e.g., shifts, multiply/divide, etc.).

2. Shift Instructions **(sll, srl, sra)**:
    a. In the shift instructions, the instruction format is: "sll $rd, $rt, shamt".
    b. As it only has rd and rt and shamt, rs is set to 0 and function and opcode are taken from the instruction_set table.
    c. The shamt in this case is not zero unlike other R instructions. It is encoded from the instruction.

3. Multiply/Divide Instructions **(mult, multu, div, divu)**:
    a. For mult, multu, div, and divu, the format is : **"mult $rs, $rt"**.
    b. The destination register (rd) is always set to 0 as these operations do not store results in a specific register directly. Instead, the results are stored in the special-purpose registers HI and LO.
    c. rs is assigned the first operand(from parts[1]), and rt is the second operand (from parts[2]). Shamt is 0 as there is no shift operation.

4. Move Instructions **(mfhi, mflo, mthi, mtlo)**:
    a. These instructions involve special registers like HI and LO. Format: **"mflo $rd"**
        - mfhi : Moves the contents of the HI register to a destination register (rd).
        - mthi : Moves the contents of a general-purpose register (rs) into the HI register.
        - mflo : Moves the contents of the LO register to a destination register (rd).
        - mtlo : Moves the contents of a general-purpose register (rs) into the LO register.
    b. For mfhi and mflo, rd is set to the destination register, while rs, rt, and shamt are set to 0.
    c. For mthi and mtlo, rd is set to 0, while rs is the source register.

5. Jump Register **(jr)**:
    a. In jr, rd and rt and shamt are set to 0, while rs is the target register for jump.
    Format: **jr $rs**
    b. The instruction does not involve an immediate value or a function to return to, unlike jal or jalr.

6. Jump and Link Register **(jalr)**:

a. Jumps to the address specified in a register (rs), and optionally stores the return address (the address of the next instruction) in a register (rd).
b. jalr sets rd to 31 (for the return address register, ra) if not mentioned **("jalr $rs")** else it is encoded from the instruction **("jalr $rd, $rs").**
c. rs is the target register for the jump, and rt and shamt are set to 0.

7. System Call **(syscall)**:
    a. syscall does not require any operands; thus, rd, rs, rt, and shamt are set to 0.as it relies on values that are already present in registers,
    b. The func field is hardcoded to 12, as that is the func code for syscall. Format: **(syscall)**
8. Standard R-Type Instructions:
    a. For other R-type instructions (e.g., add, sub, and, or), rd is set to the destination register, while rs and rt are set to the first and second source registers, respectively.
    b. shamt is set to 0 as these instructions don't involve shifting. Format : **("add $t0, $t1, $t2")**

9. Function (func) and Opcode:
    a. The func code is retrieved from instr_info, specifying the operation to be performed in R-format.
    b. The opcode for R-type instructions is typically 0.

10. Machine Code Encoding: The final 32-bit machine code for the R-format instruction is constructed as follows:
    a. (opcode << 26) | (rs << 21) | (rt << 16) | (rd << 11) | (shamt << 6) | func
    b. This structure follows the MIPS encoding format for R-type instructions, positioning each field at the correct bit offset.

I-Type Instructions:

1. Target Register Extraction:

- The code retrieves the target register (rt) from parts[1] after removing any commas.
- It uses the registers dictionary to check if this register is valid.
- If rt is not found in the registers, an error is raised, indicating an invalid target register for this specific command and instruction number.

2. Immediate Instructions (Type 1) **(addi $rd, $rs, imm):**

- For instructions that directly use an immediate value, the code sets rs (the source register) by encoding it from the instruction. If rs is missing, an error is raised, specifying the source register as invalid.
- The immediate value (imm) is encoded from the instruction and converted into an integer and it checks if the imm falls within the valid range of -32768 to 32767. If not, it raises an error for the out-of-range immediate value.

3. Branch Instructions (Type 2) **(beq $rs, $rt, imm):**

- For branch instructions, both rs and rt are required as source and target registers.
- The code retrieves rs and rt from instructions and validates them in registers. If either register is invalid, it raises an error specific to the incorrect register for the command.
- The third part (parts[3]) is either a label or an address.
    - If it's a label, it looks up the address in the symbol_table, calculates the offset as imm by subtracting the next instruction's address (address_map[command] + 4) and dividing by 4.
    - If it's an address, it validates it to ensure it's a multiple of 4 and falls within the allowed address range.
    - If imm falls outside the -32768 to 32767 range, an error is raised for an out-of-range offset.

4. Load/Store Instructions (Type 3) **(lw $rt, imm($rs)):**

- For load/store instructions, the code extracts imm and rs (base register) from the instruction using parts[2].
- It converts imm to an integer and checks if it falls within the range -32768 to 32767. If not, an error is raised. The base register (rs) is validated in registers, raising an error if it's invalid.

5. Load Upper Immediate or Branch with Single Register (Type 4) **(blez $rs, imm):**

- For these instructions, rs or rt is set to 0, whichever is not being used in the instruction, imm needs to be processed. (Example: rt is not used in blez, rs is not used in lui $rt, imm)
- If parts[2] is a label, imm is calculated as an offset relative to the current command. If it's a direct immediate, it's converted to an integer. An invalid conversion raises an error.
- An out-of-range imm value also raises an error.

7. Opcode and Machine Code Construction:

- For each I-format instruction, opcode is extracted from instr_info['opcode'].
- The machine code for each I-format instruction is generated by shifting and combining the opcode, rs, rt, and imm. Then this 32-bit instruction is appended into the machine_code list.

J-Type Instructions:

Here's a step-by-step breakdown of what's happening in the second pass of the assembler code for J-format instructions, with a focus on how the code processes labels or addresses for jump instructions:

The code block executes when the instruction is 'j' or 'jal'.

2.. Identification of Target Address or Label:

- The operand after the instruction (`parts[1]`) is expected to be either a label or a direct address. This is saved in `target_str`.
- If `target_str` is found in the `symbol_table`, it's treated as a label.

- Labels in `symbol_table` are mapped to specific memory addresses. These addresses represent word addresses (so they're already shifted right by 2 to fit the MIPS J-format instruction requirement).
- The code retrieves the word address of the label from `symbol_table` and assigns it to `target`.

3. Validating the Address Requirements:

- If the target_str is not present in the symbol table, it now converts into integer and checks if it is an address.
- It checks if the address is a multiple of 4 (since MIPS instructions are word-aligned) and that the address is within the allowable program memory range (`0 <= target <= max_address`).
- If the address is valid, it divides `target` by 4 to convert it to a word address (as required by MIPS J-format).

4. Assembling the Machine Code:

- The line `(opcode << 26) | (target & 0x3FFFFFF)` creates the 32-bit binary instruction.
  a. Opcode << 26` shifts the 6-bit opcode to the leftmost position.
  b. Target & 0x3FFFFFF` ensures that only the lower 26 bits of `target` are kept.
- The final 32-bit instruction is then appended to the `machine_code` list as a binary string formatted to 32 bits (`f"{machine_instr:032b}"`).

Library used:

- **`time`:** Use to track the performance of different parts of the assembler, such as parsing, assembling, and execution. It allows us to measure the execution time, which can help in optimizing the code
- **`re` (Regular Expressions):** Used for parsing and validating assembly instructions, labels, registers, help to detect errors and ensure input follows expected syntax by identifying instruction components accurately
- **`tabulate`:** Used for presenting data in a readable, table format within the command line or reports.

Special Features:

1. The instruction file is given through a path, the user can use the file of his choice.
2. The mnemonics given in the input are not case sensitivity problems.
3. The number of labels and the number of different types of instructions(R, I, J) of each type is calculated.
4. The number of types of each instruction is displayed in the output.
5. The symbol, commands and address table is printed for better understanding.
6. The execution time and Throughput time (in seconds) is calculated and shown in the output to estimate the performance.
7. For branch and J instructions, the address can be taken in hexadecimal or integer

8. The output is in the form of a tabular form, having the address, command, binary and hexadecimal representation for better visualisation.
9. The output is also stored in a .bin file in the binary format which helps us in simulating for further operations.
10. Our MIPS simulator supports the NOP (No Operation) instruction, which performs no action for one clock cycle. It has been implemented to occupy a single clock cycle without performing any modifications to the register file or memory.
11. The users can run MIPS assembly code directly from the command prompt.

    Format : Go to that directory or folder where this 'assembler.py' input is there then give 'python assembler.py input.txt output.bin'.

Points to note while giving the instructions (Specifications for the input):

1. Do not start a command with '#', else it is considered a comment.
2. Anything after the '#' is not read by the code in that line.
3. Do not end a command with ','.
4. Label name should not have spaces (' '), colon (:), or commas(,).
5. Each line should have only one instruction, even if a label is used.
6. Instructions should be given from the MIPS reference sheet (attached) only.

Task 4: Error Handling and Validation

Developed and implemented error detection mechanisms for invalid instructions, incorrect operand usage, and incorrect syntax. Handled errors related to undefined labels, out-of-range immediate values, and other logical issues.
Every time an error is detected, it is shown to the user indicating the line(the instruction line in the input) and the command where the error is raised.

1. Invalid Instruction Detection:
    - Check if each parsed instruction exists within the MIPS instruction set. If it is not present an error message is displayed:

      ```
      ValueError: Error: Invalid mnemonic 'adu' found in command 'adu $t0, $t1, $t2'at
      instruction number 2.
      ```

2. Operand Validation:
    - Ensure the correct number of operands and counts for each instruction.
    - For every command, once the type and subtype is identified, it checks for the number of operands. If they are more or less than required, an error is raised
    - Example:

      ```
      ValueError: Error: Incorrect number of operands in command 'ori $t0, $t1, 100,
      0x60' at instruction number 31. Expected 3 operand, got 4.
      ```

3. Undefined or Duplicate Label:

- If a label or the immediate value given for a branch or jump type instruction and that label is not present in the instruction file, an error is generated.

```
ValueError: Error: Invalid immediate value or label 'label1' in command 'blez $t0,
label1' at instruction number 46.
```

- If any label is given twice:

```
ValueError: Error: Duplicate label 'START' found at address 20
```

4. Invalid Register:
   - If the registers in the commands are out of bounds (not from the defined registers), an error is generated.

```
ValueError: Error: Invalid base register '$t40' in command 'lw $t0, 100($t40)' at
instruction number 36.
```

5. Out of range address:
   - If the address given in the J or I type command is out of range or is not a multiple of 4 then an error is generated.

```
ValueError: Error: Address '6' is not a multiple of 4 in command 'j 0x6' at
instruction number 48.

During handling of the above exception, another exception occurred:

ValueError: Error: '0x6' is neither a valid label nor a valid address in command 'j
0x6' at instruction number 48.
```

   - If the address given is out of range, then error generated will be:

```
ValueError: Error: Address '567892300' is out of the valid program range (0 to 24) in
command 'j 567892300' at instruction number 5.
```

6. Immediate out of range or invalid:
   - If immediate value given in any instruction is out of range, it gives the error message:

```
ValueError: Error: Immediate value '5678903456' out of range (-32768 to 32767) in command
'addi $t1, $t2, 5678903456' at instruction number 7.
```

   - If the immediate value is not a valid integer, then it gives the error:

```
ValueError: invalid literal for int() with base 10: 'si'
```

Objectives Planned VS Achieved:

1. Planned to research on ISA of MIPS to understand instruction formats, addressing modes, opcodes, and registers, and we achieved it for all the instruction given in the reference list .
2. Planned to handle labels, comments, and directives, and we achieved all these including extraneous spaces, tabs, lines, commas, duplicate labels.

3. We planed to Develop and implement error detection mechanisms for invalid instructions, incorrect operand usage, and incorrect syntax. Handle errors related to undefined labels, out-of-range immediate values, and other logical issues and we achieved all that with some specification.
4. Prepare documentation on how to use the assembler, including input and output formats with clarity and maintainability of the code, and then review the assembler to ensure all requirements are met and it is achieved and we achieved
5. We planed to Test and Simulate MARS MIPS simulator (for testing and validating the generated binary code) but as we are not using the directives like '.text', '.data' before the instruction the program may not recognize where instructions start, causing the PC to be undefined/ invalid.

Challenges Faced:

We faced the following challenges while developing this project:

1. **Issue**: Initially, spaces before commas in operands (e.g., `add $t0, $t1 ,$t2`) caused the assembler to throw an error: "Invalid register '$t1 ' found in instruction 'add $t0, $t1 , $t2'."
   Cause: The assembler was not trimming spaces before commas, resulting in invalid input.
   Solution: We modified the code to remove spaces before and after commas, ensuring that only valid operands are processed. the assembler now correctly processes operands with spaces around commas.

2. **Issue**: The assembler previously allowed extra operands or random labels in instructions (e.g., add $t0, $t1, $t3, $67, random), and only processed the first valid operands, ignoring the extras.
   Solution: We added a check to validate the number of operands for each instruction type (R-type, I-type, J-type).
   Error Handling: If an instruction contains more operands than required, an error is thrown, ensuring that only valid operands are processed.

3. **Issue**: Commands like mflo, mtlo, etc., which have only one operand, were initially classified as R_single in the instruction set table, leading to incorrect machine code generation.
   Cause: These commands were not processed correctly because they were not recognized as valid R-type instructions.
   Solution: We updated the instruction set to classify these commands as R-type and added a subtype within R-type to handle them properly.

4. **Issue**: The assembler initially allowed duplicate labels without throwing an error, using only the first instance for conversion.
   Cause: There was no duplicate check in the first pass of the assembler.
   Solution: We added a duplicate label check during the first pass.

5. **Issue**: The assembler processed text after a comment marker (#) as part of the instruction, leading to invalid instruction errors.
   Cause: The assembler did not recognize # as a comment marker if it appeared at the end of the line.

Solution: Updated code to strip out text following `#`, treating it as a comment regardless of its position, thus ensuring only valid instruction parts were parsed.

6. **Issue**: Hexadecimal for jump values (e.g., `0x10`) caused errors as they were not recognized properly.
   Cause: The assembler initially expected only decimal values for jump and branch.
   Solution: Added support for hexadecimal parsing by detecting `0x` and converting such values appropriately, allowing both decimal and hexadecimal immediates for most cases.

7. **Issue**: Labels similar to instruction mnemonics (e.g., a label named `add`) caused the assembler to misinterpret them as instructions.
   Cause: No distinction was made between labels and instruction mnemonics in certain parsing conditions.
   Solution: Added additional checks to differentiate labels from mnemonics, ensuring accurate identification of each instruction and label.

8. **Issue:** First we tried to do error validation in the first pass which took a lot of time and was inefficient.
   Cause: As we need to account for many errors and edge cases in the instructions.
   Solution: It was easy to do in the second pass, as the second pass checks for each type and subtypes of instructions and can easily identify and throw an error directly.

Open Issues:

1. One of the open issues is related to inconsistent handling of spaces within register names. For example, an input like "`$t 1`" should ideally be interpreted as "`$t1`" if spaces are removed, making it a valid register. However, the implementation inconsistently removes spaces in some parts of the code while leaving them intact in others. This inconsistency leads to ambiguity and can cause the assembler to misinterpret or reject otherwise valid register names**.**

Core Technical Contributions:

Srijahnavi Chinthalapudi - Developed a module to preprocess the input assembly code by eliminating unnecessary spaces and output instructions in the form of commands, symbol table and addresses. In the second pass, developed the code for I-type instructions (for all the subtypes) with specifications such as to include hexadecimal input for branch type instructions etc. Worked on the code for J-type instructions in second pass and to statistical analysis of of the instructions in the text file.

Siri Gangannagudem - Developed a module to preprocess the input assembly code by eliminating unnecessary spaces output instructions in the form of commands, symbol table and addresses.In second pass worked on developing the code for R-type instructions (for all subtypes) and the special instructions

such as “nop” and “syscall”. Added the feature to run the instruction file from command prompt and generate output in the form of .bin file.

Deepanjali Kumari - Made the second pass efficient by adding error checks at different edge cases, tabulated the output for better visualisation, worked on simulating the code on MARS MIPS simulator to test the code. Worked on the code for J-type instructions in the second pass.

Bharath Chandra Vubbani - Worked on the first pass to remove spaces and separate the labels and store addresses of the labels and instructions.

Summary of the Project:

1. The project is mainly divided in two parts:
   First Pass: It reads and processes each line of the assembly code and cleans up the commands
   Second Pass: This converts each instruction into its corresponding machine code based on its format (R, I, or J). The assembler verifies the validity of mnemonics, the count and types of operands, and the correctness of register names.

2. The code includes the following:
   - Validation for mnemonic and label usage.
   - Binary to hexadecimal conversion for the generated machine code.
   - Error messages that indicate specific issues like incorrect operand counts, invalid registers, and out-of-range immediate values.

Takeaways from the Project:

1. Understanding Instruction Formats: Implementing the assembler required a deep understanding of MIPS instruction formats (R, I, and J types and their sub-types).
2. Effective error handling was crucial to ensure accurate translation and provide meaningful feedback to the user to enhance code robustness.

Performance Testing:

- **Complex Program Testing**: Used small programs that combine various instructions, loops, branches, and memory operations. Check if the assembler produces the expected binary output.
- **Basic Instruction Testing:** Tested each MIPS instruction type (R, I, J) individually to ensure correct parsing, validation, and conversion to binary. This includes common instructions like `add`, `sub`, `lw`, `sw`, `j`, etc.
- **Error Message Clarity**: Intentionally introduced errors such as giving invalid registers, opcodes, instructions, duplicate labels, and large values in branch or jump instructions in the test cases to verify if the assembler’s error messages are descriptive and pinpoint the exact issue to make debugging easier.
- **Code Changes Testing**: Each time we have updated the assembler’s code (e.g., optimizations, bug fixes), we run

- a full set of tests to ensure new changes haven't introduced any errors.

Tools & Technology Used:

- **Programming Language:** Python Programming
- **Development Environment:** Google Colab, VS Code.
- **Version Control:** GitHub (for collaboration and version control of the project code).
- **Documentation:** Google Docs, Google Slides
- **Output File:** Output is generated using bin file
- **Code Implementation:** Command Prompt

References :

1.S. Kulkarni, *Computer Organisation and Architecture*, Lecture slides, IIT Gandhinagar, Available: 2024-25 Sem1 ES 215 COA | General | Microsoft Teams.

2.NPTEL, "Computer Architecture and Organisation," YouTube, National Programme on Technology Enhanced Learning. Available: https://www.youtube.com/user/nptelhrd

3.Patterson, D. A., & Hennessy, J. L. (2014). *Computer organisation and design: The hardware/software interface* (5th ed.). Morgan Kaufmann.

4.Stallings, W. (2017). *Computer organisation and architecture: Designing for performance* (10th ed.). Pearson.