ES 215 Computer Organization and Architecture Project

Report

ASSEMBLER AND DISASSEMBLER

April 26, 2022

Indian Institute of Technology, Gandhinagar

Harshvardhan Vala 20110075
harshvardhan.vala@iitgn.ac.in
Inderjeet Singh Bhullar 20110080
inderjeet.bhullar@iitgn.ac.in
Kalash Kankaria 20110088
kalash.kankaria@iitgn.ac.in
Kanishk Singhal 20110091
kanishk.singhal@iitgn.ac.in

CONTENTS

1) Abstract

The project enables the user to interactively convert MIPS code to machine understandable instructions just with a few clicks. The application is for anyone who frequently converts MIPS code to machine code or otherwise. The GUI platform provides a fast and easy way of conversion while saving the data into text files.

2) Introduction

- **The Goal of this Project**

  This project aims to make an assembler and disassembler along with a GUI interface for the MIPS 32 architecture using Python and the Tkinter library.

- **The Rationale of this Project**

  An assembly code is human readable and quite comprehensible but a machine code being written in binary is hard to read and comprehend without prior knowledge.

  We decided to take up this project in order to make this conversion from an assembly-level code to machine code and vice versa. This assembler and disassembler can be used for educational as well as research purposes since it reduces the labor-intensive work so that one can focus on more advanced applications and problems in this field.

- **The Importance of this Project**

  This project is the culmination of our coursework and knowledge earned during the semester. We have tried to implement all the concepts of MIPS 32 architecture and related assembly language and machine language, learned until now to make

a platform, that is easy to access and use, which enables one to assemble an assembly-level code and disassemble this assembled code (machine code) back to the assembly-level code.

To practically implement the theory we have learned all semester, will be an enriching experience for us in order to build these concepts stronger.

- **Our Main Contributions to the Project**

  We implemented this assembler and disassembler by writing basic Python codes that cater to various instruction formats available in the MIPS-32 Architecture and further coded the different cases that may occur for assembling a particular function and similarly the different cases that may occur on the input of a machine code.

  We designed and developed a complete GUI application that provides an easy-to-use interface for this assembling-disassembling. We also provide the user with a choice to deal in binary numbers or hexadecimal numbers. While the computer hardware only understands binary, we added this extra feature to enable the user to read even the machine code easily.

3) Literature Review

- Our computer hardware can only understand binary language. Conversing with the hardware would require us to use binary ourselves which is a very tedious task and would require high-level knowledge. To make this task easier, a human-readable assembly language is used by us to converse with the hardware. This assembly language needs to be translated to the binary machine code, and the machine code output by the hardware also needs to be converted back to the

assembly language, this is where the use of an assembler and disassembler comes in.

- Numerous efforts in a similar domain have been made by people since this is a very essential requirement for people who research in this domain as it reduces the hard labor of converting each instruction in assembly language to binary machine code. Here is one of the existing online assemblers/disassemblers: http://shell-storm.org/online/Online-Assembler-and-Disassembler/
- Our solution to this problem is a simple, accessible, free, and easy-to-use GUI application that can be used to assemble a MIPS 32 assembly code and disassemble this assembled code back into MIPS 32 assembly language to check its accuracy. This Assembler/Disassembler is more fit for educational purposes and not research since it implements only the basic MIPS 32 instructions. Our solution provides versatility in the way the machine code is input and output. It allows the user to choose between binary and hexadecimal input/output formats based on ease of readability. It also runs a check of the correctness of the input code since sometimes the user may enter the wrong instruction format.

4) Your Project Idea

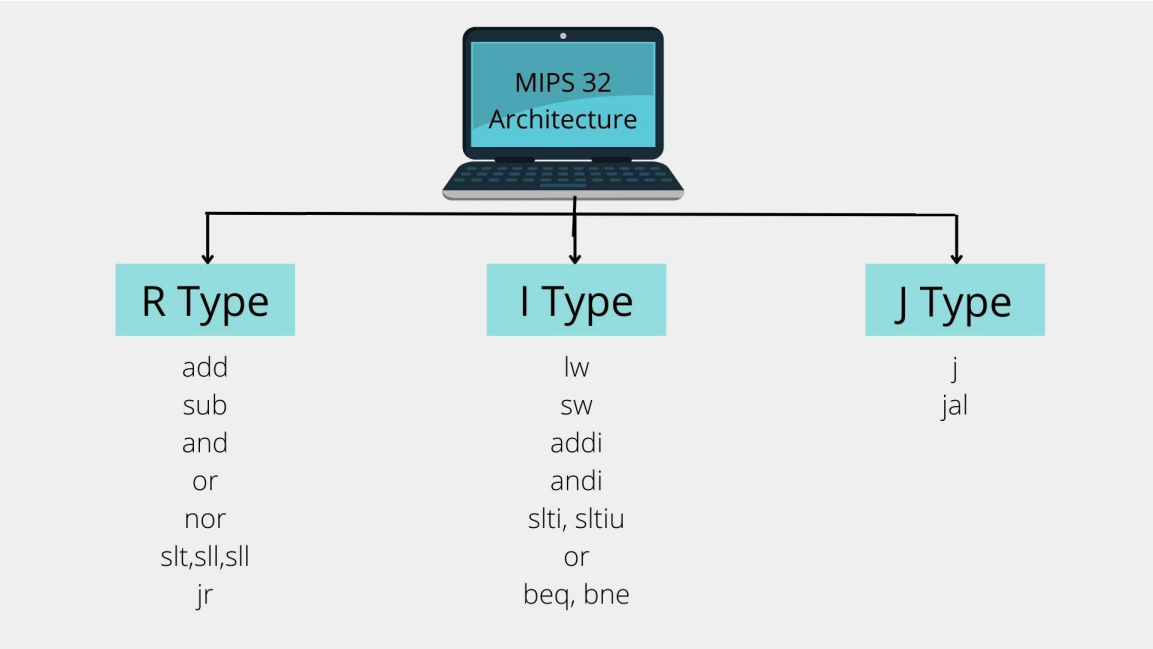
The objective of this project is to implement an Assembler-Disassembler that takes a MIPS 32 assembly code as input and outputs a binary machine code. The program then asks the user whether they want to again disassemble this binary machine code to the MIPS 32 assembly code.

Our Solution

We have three different files named assembler, disassembler, and GUI. The user only needs to run the GUI file and an application will open up that will contain

the interface where the MIPS assembly code needs to be entered along with the user input options to choose whether the output will be in binary or hexadecimal.

The assembler code then parses through the text written in the assembly language block and for each known expression as defined in the code, for example, register values, function abbreviations, immediates, labels, etc., the code assembles the machine code according to the MIPS 32 instruction format.



This assembled code is then output in the Machine Code block. The user can then choose to clear both the blocks or again disassemble the binary machine code to give the original assembly code.

The disassembler also parses through each line of machine code taking in a 32-bit binary input as one instruction and then disassembles it according to the MIPS 32 instruction format to give the original assembly code.

Pseudo Code for Assembler

Algorithm 1:

```
# define functions for numeric system conversions

bin_to _hex(bin_str, n_bits)
dec_to _bin(dec_str, n_bits)


# define dictionaries that assign the assembly code symbols the designated decimal values according to MIPS 32 architecture for instruction format, opcodes, function codes, registers, and also arrays too for classifying different instructions according to their formats

instruction_data = {}
registers = {}
R_format_0 = []
R_format_1 = []
R_format_2 = []
R_format_3 = []

for all instructions in instruction_data
     append instructions to appropriate arrays

# define the assemble function
open input_file
initial_address ← 0
pc ← initial_address

Instruction_memory=[]
Labels={}

create list instructions
create dictionary labels

for each instruction in Instruction_memory
   pc ← pc + 4
   operation ← instruction[operation]

   # R type instruction
```

```
if operation['type'] = 'R'
    opcode=operation[‘opcode’]
    funct  = operation[‘funct’]
    rs ← registers[instruction.rs]
    rt ← registers[instruction.rt]
    rd ← registers[instruction.rd]
    shamt← instruction.shamt
    output opcode + rs_code + rt_code + rd_code + shamt + funct


# I type instruction

if operation['type'] = 'I'
   opcode=operation[‘opcode’]
    rs ← registers[instruction.rs]
    rt ← registers[instruction.rt]
    immediate ←instruction.immediate
    output opcode + rs_code + rt_code + immediate



# J type instruction

if operation['type'] = 'J'
   opcode=operation[‘opcode’]
   address=instruction.address
   address ← dec_to_bin(address, 32)
   address ← address[4:]
   address ← int(address)
   address ← address/4
   address ←dec_to_bin(address,26)

   output opcode + address

close output_file
```

Pseudo Code for Disassembler

Algorithm 2:

```
# define functions for numeric system conversions

hex_to_bin(hex_str, n_bits)
dec_to _bin(dec_str, n_bits)
bin_to _dec(bin_str, n_bits)

# define dictionaries that assign the assembly code symbols the designated decimal values according to mips 32 architecture for opcodes, registers, and labels

opcode_data = {}
registers = {}
func_data = {}

# define the disassemble function
open input_file
open output_file

initial_address ← 0
pc ← initial_address

create list instructions
create list addresses
create dictionary labels

for each line in input_file
   if bin_input = 0
      convert line to hexadecimal

   opcode ← hex(line[0:6])
   data ← opcode_data[opcode]
   pc ← pc + 4

   # R type instruction

   if data['type'] = 'R'
      rs ← registers[line[6:11]]
      rt ← registers[line[11:16]]
      rd ← registers[line[16:21]]
```

```
        shamt ← line[21:26]
        funct ← line[26:32][2:]
        operation ← funct_data[funct][‘operation’]
        format ← funct_data[funct][‘format’]

        for different types of function types:
            instruction ← mips 32 defined instruction format for r

    # I type instruction

    if data['type'] = 'I'
        operation ← data[‘operation’]
        format ← data[‘format’]
        rs ← registers[line[6:11]]
        rt ← registers[line[11:16]]
        immediate ←line[16:]

         for different types of function types:
            instruction ← mips 32 defined instruction format for i

    # J type instruction

    if data['type'] = 'J'
        operation ← data[‘operation’]
        address ← shift left binary(line[6:]) by 2 bits
        address ← dec_to_bin(address, 28)
        address ← concatenate first 4 bits of pc to address

        if address not present in addresses then add address

    sort(addresses)
    for all address in addresses
        assign a label

    output each assembled instruction

close input_file
close output_file

disassemble(bin_input)
```

5) Project Implementation

In this section, describe details of your project design and implementation, document challenges and obstacles and how you overcome them in design and implementation.

Assembler

is a program that is used to convert the assembly language code into machine code. Assembly language is human readable and easily convertible into machine code. It takes in input the assembly language codes like, MIPS assembly code, and converts it into binary that the computer understands.

There are three types of instructions in MIPS Assembly- R type, I type and J type. A MIPS Assembly code is made up of an operation, registers and/or immediates and addresses. Each instruction has some operation that it needs to perform. Each operation has a binary code associated with it that uniquely identifies the operation. In R type of instruction, the opcode and the function code together define the operation, while in I type and J type, the op code is sufficient to determine the operation type. There are different formats in which each of the type of instructions appear:

- Operation reg, reg, reg
- Operation reg, reg, integer
- Operation reg, integer
- Operation reg, integer
- Operation reg, integer(reg)
- Operation reg, reg, address

Our implementation is based on parsing through the instruction, identifying the type of instruction, as well as the format in which it occurs, and then converting it into the binary code.

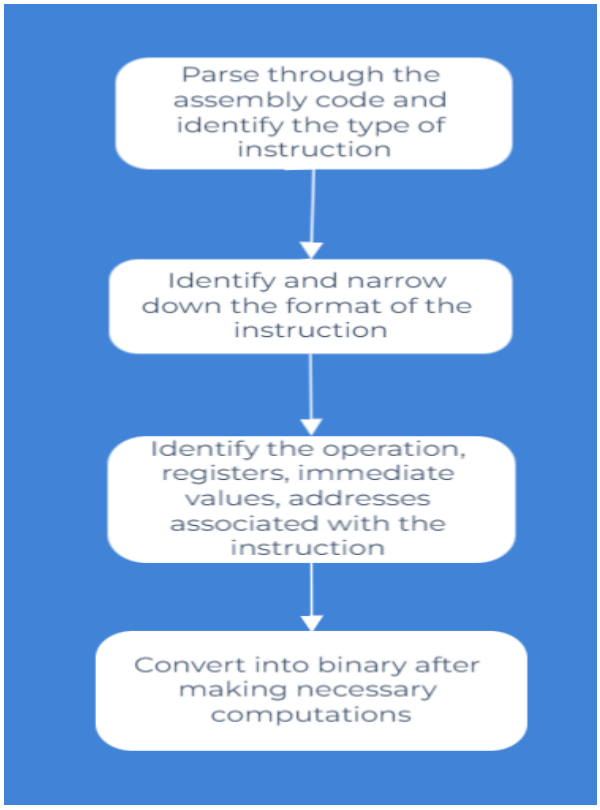


Figure 1: Flowchart of the Assembler program

Disassembler

The disassembler is the program that can convert Machine Code back into MIPS Assembly Code. The disassembler we have programmed can take machine code in the format of binary or hexadecimal and it will output assembly code. The format of the input should be such that each machine code instruction is in a different line.

In order to disassemble the code, the program first figures out the type of Instruction using the opcode. Depending on whether the instruction is R type or I type or J type, we apply conditional statements in order to decode the machine code to assembly code.

We will be using a lot of cases and conditions in our code in order to cover various formats of all the 3 types of instructions. For example,

Add is an R type instruction with an instruction format that takes 3 registers but the Slt instruction is also an R type which has information of only 2 registers in the instruction format.

The main crux of the code is of the format :

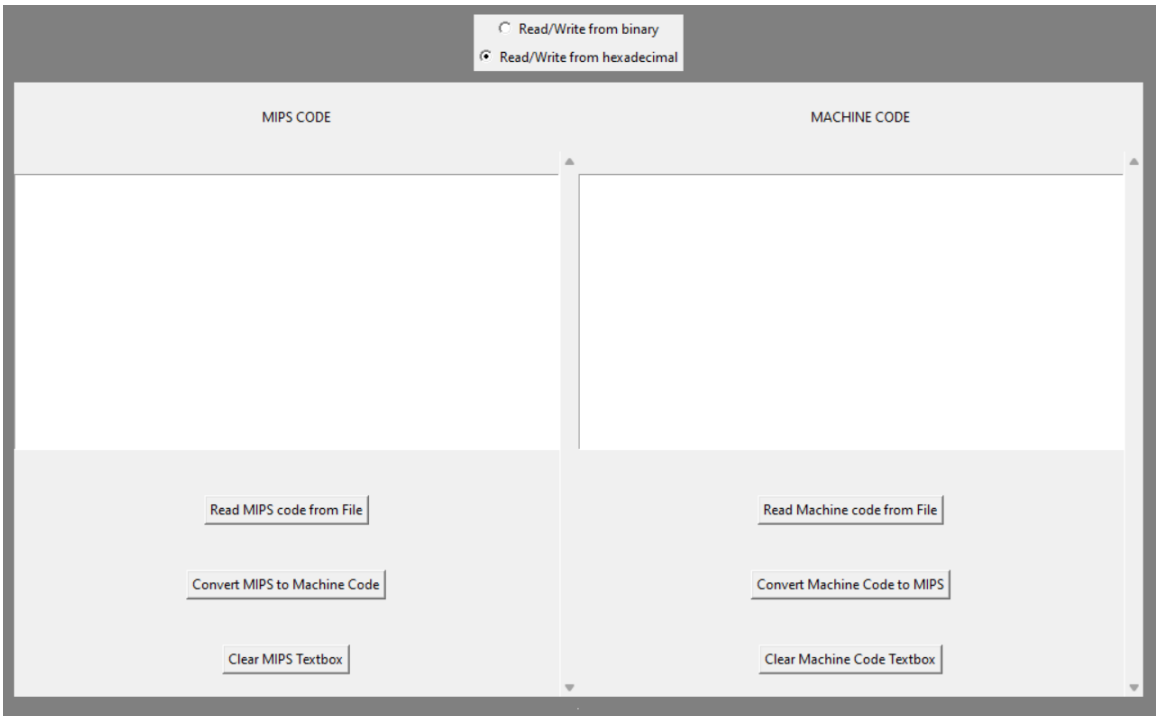
```
        if data['type'] == 'R':
            rs = registers[int(line[6:11], 2)]
            rt = registers[int(line[11:16], 2)]
            rd = registers[int(line[16:21], 2)]
            shamt = int(line[21:26], 2)
            funct = hex(int(line[26:32], 2))[2:]
            opperation = func_data[funct]['opperation']
            format = func_data[funct]['format']

            if format == 0:
                instruction = [opperation + ' ' +
                                    rd + ', ' + rs + ', ' + rt,
None]                if format == 1:
                instruction = [opperation + ' ' + rd + ', ' +
                                rt + ', ' + str(shamt), None]
            if format == 2:
                instruction = [opperation + ' ' + rs, None]
```

Here we have created the instruction for all the formats of the R type instruction.

Similarly we perform such conditional statements for I type and J type instructions.

Graphical User Interface (GUI):

We have implemented a python based GUI built on the tkinter module. The GUI makes the application more interactive for the user providing text boxes for input and output. The GUI also provides a click on button which instantly converts machine instructions to MIPS code and vice-versa. The requirements of an ideal GUI is that it should be interactive and user friendly, so that the user can read and write input and output easily.



Our GUI enables the user to write in the application itself or save the data in a given text file and the GUI will read from that file. The application also saves the output in the respective file for future reference.

MIPS.txt - Contains the MIPS code, the GUI reads the data for text box under MIPS section from this file.

machine.txt - Contains the machine code, the GUI reads the data for text box under machine section from this file.

Features

1. GUI provides a text box area for input machine instructions or MIPS code and to see the results in another box.
2. It can read from a text file by pressing the ‘Read from file’ button.
3. The application has a radio button which can be used to decide binary or hexadecimal values to be store in machine instructions file.
4. GUI also has a clear text button which clears the text box area for fresh use.

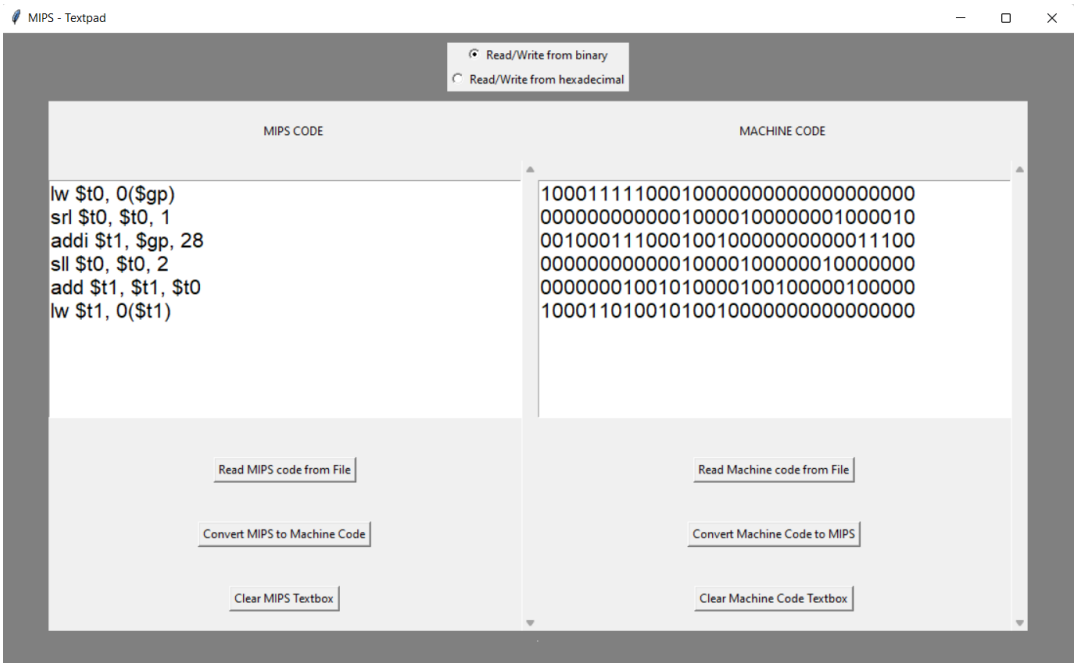
6) Testing and Experiments

Hereby we have attached snippets of our program when it converts assembly code to machine code and vice versa. We have manually checked these instructions and confirmed that they are correct.

- Example - We will try to convert the following Assembly code into Machine code

  lw $t0, 0($gp)
  srl $t0, $t0, 1
  addi $t1, $gp, 28
  sll $t0, $t0, 2
  add $t1, $t1, $t0
  lw $t1, 0($t1)

  Our program outputs the following machine code :



Machine code :

10001111100010000000000000000000

00000000000010000100000001000010

00100011100010010000000000011100

00000000000010000100000010000000

00000001001010000100100000100000

10001101001010010000000000000000

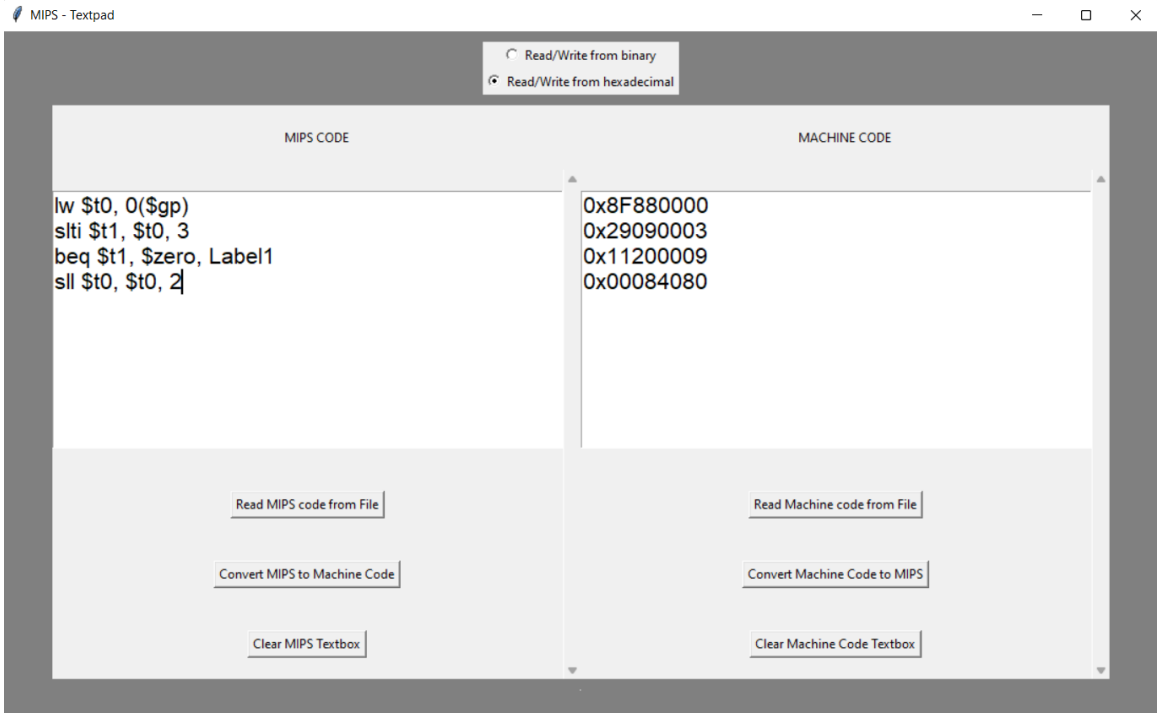
- Example - Here we will try to convert the following machine code to assembly code. We will also attempt it in hexadecimal format

  0x8F880000

  0x29090003

  0x11200009

  0x00084080



Assembly Code :

lw $t0, 0($gp)

slti $t1, $t0, 3

beq $t1, $zero, Label1

sll $t0, $t0, 2

Here we have verified that our program works upto most of the extents.

7) Limitations

Our program can easily convert most of the assembly code to machine code and vice versa, but we still face a few limitations. Some of which are:

- The error handling of our program is not upto the mark. Sometimes when we encounter an input that is not valid, we will either get an output or get no output rather than getting an error prompt.
- Whenever the disassembler gets an empty input line, it is not able to give the output for the lines coming after the empty line.
- If we click on convert on the same code multiple times then we get the same result added to the output text file. This should not happen.
- Even if we input a binary file of size greater than 32, our disassembler still works and outputs an assembly code instead of giving an error.
- We were not able to add all the operations and functions available for the MIPS assembly language. Due to this we have a limit to the operations we can assemble and disassemble.
- Another limitation of our program is that whenever we assemble code with a label in it that is not used, we will not get that label returned once we disassemble the assembled code.
- This is not a limitation but something to keep in mind is that once we assemble some assembly code with particular labels then we will not get the same labels once we disassemble the machine code.
- Another limitation with our program is that the assembly code must be in a particular format in order to get converted to machine code.

  The format required is of form - “add $t1, $t2, $t3”

  If we add spaces or commas to this format, then we will get an error.

8) Conclusion

- The Assembler-Disassembler developed under this project is a Graphic User Interface Application that takes a MIPS assembly code and assembles it into a Binary Machine Code and vice-versa. Simply put, it translates a human-readable set of instructions into a hardware compatible instruction set and also converts machine instructions to human-readable text. This code implements this assembling and disassembling for most of the MIPS 32 instructions.
- If we look at the wider scope of this project, it can be used to implement a compiler that directly converts the Assembly Language to the Binary Machine Code and vice versa.
- The more efficient and fast a compiler is, the faster the execution time for any program.
- Our project is a simple, free, and easy-to-use code/application that can be used to efficiently reduce the labor-intensive work of assembling and disassembling a code. That can be used for educational purposes such that students or professors can use it to check whether the solutions arrived at from manual calculations are correct or not.

  GitHub Repo Link for our Project

9) Future Scope

- If given more time, we could include a much larger set of instructions in the Assembler-Disassembler and even some of the pseudo instructions.
- We could also include a much more complex and efficient “Error Handling”. As the number of instructions increases, the different types of errors associated with

them do too, for example in the div instruction, division by 0 is an important factor that one needs to consider.

- In the availability of more time, we could use a faster language like C/C++. Writing such codes in these languages takes more time but is much faster as python is a comparatively slow language.
- We could also make a directory in the GUI Application that keeps a track of the addresses of the instructions in the memory after getting the input of the initial address of the first instruction. This can be used to further optimize one's code in order to occupy memory efficiently and also has its further scope in dealing with pipelining hazards, especially for Control Hazards.
    - Filling of the Delay Slot can be efficiently implemented using this tracker.
    - Branch Prediction can be improved by many folds by efficient stacking of code. This is what most modern compilers try to do.

10) References

1. David A Patterson and John Hennessey, Computer Organization and Design 5th Edition, Morgan Kaufman Publishers, 2017.
2. William Stallings, Computer Organisation and Architecture, 10th Edition, Pearson Edition, 2020.