Indian Institute of Technology, Gandhinagar

ES 215 Project Report

Assembler and Disassembler

Authors

Chirag Sarda 20110047

Bhavesh Jain 20110038

Dhruv Parekh 20110058

Rahul Chembakasseril 20110158

26^{th} April, 2022

Under the guidance of

Prof. Sameer Kulkarni

Contents

0.1 Abstract

This project involves building an Assembler and Disassembler for the MIPS 32 ISA. The following report contains detailed description about

the design methodology, algorithms used, GUI implementation, test cases, results obtained, challenges and limitations.

0.2 Introduction

0.2.1 Project Goal

- The goal of our project is to create a MIPS Assembler and Disassembler. The MIPS assembler that has been coded by us takes in assembly code as input and outputs the corresponding machine language code in hexadecimal.

- Following if wished we can use the Disassembler program to take in the produced machine code as input and reproduce the original MIPS Assembly Code. We have also built a GUI to display these outputs, for the ease of the user.

0.2.2 Project Rationale

- The reason behind choosing this project was to make the tough process of inter-conversion of assembly and machine code much easier.

- We have also implemented a user friendly GUI which opens a wide range of applications for our project. One main application, which we had kept in mind while creating this was to use it for education purposes.

0.2.3 Importance of the Project

- The importance of this project lies in two domains. One refers to our development in this course and how this project has helped

us fine tune our knowledge and implement systematic application of the overall contents of the course.

- The second lies in the importance of this project for its application wherein the cumbersome inter-conversion of assembly and machine code becomes fast and easy. It was due to this project that we could implement the theoretical knowledge and get practical experience from it.

0.3 Literature Review

- A computer does not understand the English language and information is passed in only binary. To write instructions for the computer in binary directly is a very tedious task that requires knowledge and expertise. Even then it will be very time consuming. Hence instructions can be written by us in an assembly language which is easy to implement. This is then converted to binary for the sake of the computer. In the same way the binary code coming from the computer needs to be converted into assembly language for the user to understand easily. These conversions are brought about by the Assembler and the disassembler.

- This problem has been present for a long time and several attempts have been made to solve it. Here are some of the models freely available for us to study.

  - https://github.com/zeckendorf/Python-MIPS-Assembler
  - https://github.com/SamuelGong/MIPSAssemblerAndDisassembler
  - https://github.com/kashyapakshay/PyMIPS

  The pre-existing solutions that we came across our very calculative in their approach and have not designed a model for ease of

user, but rather just for the conversion process. Moreover they do not give an error message upon entering a wrong format of input. All the models do not even cover the rare MIPS instructions and majorly stick to the 5 or 6 major types of instructions.

- The basic idea behind the assembler and disassembler remains the same in general for all models. What we have changed is the interaction and depth of this code. Instead of picking up pre-existing models from the freely available sources we coded the entire logic behind our program from scratch based completely on the knowledge we acquired. We have also implemented a user friendly GUI which makes the input output process interactive and learning friendly. Moreover we output the machine code in a way that the user can learn about the different types of instructions and their formats in an applied way of learning.

0.4 Project Idea

- Our solutions to the Assembler and Disassembler Program are shown below. We have tried to convey this information through the Pseudocode for both the Assembler and Disassembler. We have also attached the Data Card to our Assembler and Disassembler. It contains the instructions supported by our program. A detailed explanation of our Algorithms is described in the the next section.

- Assembler Pseudocode

All the MIPS instructions is divided into these 5 sets

```
r_type=['add', "sllv", ...]
i_type=["addi","beq",...]
j_type=["j", "jal"]
s_type=["sw","lw","lb","sb"]
shift_type=["sll","sra"]

def assemble_it (a):
    l=[] → final output stored here
    for i in range(len(a)):
        a[i]=split the instruction
```

Now checking the 0th index of split instruction and mapping the instruction according to the bifurcation in the above 5 sets and printing the instruction accordingly

```
    for i in (a):
        if i[0] in r_type:  .....

        elif i[0] in i_type:  .....

        elif i[0] in j_type:  .....

        elif i[0] in s_type:  .....

        elif i[0] in shift_type: .....

        elif i[0]=="jr": .....

        elif i[0]=="syscall": ........


    return l
```

- Disassembler Pseudocode

```
if opcode_integer not in opcodes dictionary:
        output = "THE OPERATION IS NOT IN OUR DATA CARD"

elif opcode_integer == 0:  → r type instruction

        # special r type instruction
        if(funct_integer == 0 or funct_integer == 2 or funct_integer == 3): →shift instructions

        elif(funct_integer == 8 or funct_integer == 9):  → jr/jalr

        elif(funct_integer == 12 or funct_integer == 13):  → syscall or break

           # mfhi, mthi, mflo, mtlo
        elif(funct_integer == 24 or funct_integer == 26): → mult and div



elif (opcode_integer == 2 or opcode_integer == 3): → j type instruction

else: → i type instruction

        if (opcode_integer == 32 or opcode_integer == 35 or opcode_integer == 40 or
opcode_integer == 43):  → lw lb sb sw

        else: → For rest of the i type instructions
```

- Supported Data Card

Arithmetic Logic Unit							
ADD rd,rs,rt	Add	rd=rs+rt	0	rs	rt	rd	0
ADDI rt,rs,imm	Add Immediate	rt=rs+imm	1000	rs	rt		
AND rd,rs,rt	And	rd=rs&rt	0	rs	rt	rd	0
ANDI rt,rs,imm	And Immediate	rt=rs&imm	1100	rs	rt		
LUI rt,imm	Load Upper Immediate	rt=imm<<16	1111	rs	rt		
NOR rd,rs,rt	Nor	rd=~(rs|rt)	0	rs	rt	rd	0
OR rd,rs,rt	Or	rd=rs|rt	0	rs	rt	rd	0
ORI rt,rs,imm	Or Immediate	rt=rs|imm	1101	rs	rt		
SLT rd,rs,rt	Set On Less Than	rd=rs<rt	0	rs	rt	rd	0
SLTI rt,rs,imm	Set On Less Than Immediate	rt=rs<imm	1010	rs	rt		
SUB rd,rs,rt	Subtract	rd=rs-rt	0	rs	rt	rd	0
XOR rd,rs,rt	Exclusive Or	rd=rs^rt	0	rs	rt	rd	0
XORI rt,rs,imm	Exclusive Or Immediate	rt=rs^imm	1110	rs	rt		
Shifting Instructions							
SLL rd,rt,sa	Shift Left Logical	rd=rt<<sa	0	rs	rt	rd	sa
SRA rd,rt,sa	Shift Right Arithmetic	rd=rt>>sa	0	0	rt	rd	sa
Branch and Jump Instructions							
BEQ rs,rt,offset	Branch On Equal	if(rs==rt) pc+=offset*4	100	rs	rt		
BNE rs,rt,offset	Branch On Not Equal	if(rs!=rt) pc+=offset*4	101	rs	rt		
J target	Jump	pc=pc_upper|(target<<2)	10				
JAL target	Jump And Link	r31=pc; pc=target<<2	11				
JR rs	Jump Register	pc=rs	0	rs			0
SYSCALL	System Call	epc=pc; pc=0x3c	0				0
Load/Store Instructions							
LB rt,offset(rs)	Load Byte	rt=*(char*)(offset+rs)	100000	rs	rt		
LW rt,offset(rs)	Load Word	rt=*(int*)(offset+rs)	100011	rs	rt		
SB rt,offset(rs)	Store Byte	*(char*)(offset+rs)=rt	101000	rs	rt		
SW rt,offset(rs)	Store Word	*(int*)(offset+rs)=rt	101011	rs	rt		

0.5 Project Implementation and Algorithm

- Assembler
  - Our Assembler program is designed such that it takes assembly code as input and returns the corresponding machine code as the output.
  - The assembly code is available to us in the form of a string. The primary idea is determining that we will obtain five main types of strings. Note that we refer to the type of strings and not the type of instruction- there are only 3 types of instructions (R,J,I). We have assumed 7 main types of strings:-
    - R type- typical arithmetic instructions like add, sub, or, xor, sllv, jr, etc.

    * I type- addi, ori, beq, bne, etc.
    * J type- j, jal.
    * S type- sw, lw, sb, lb.
    * Shift type- sll, sra.
    * jr type- jr.
    * syscall type- syscall.
  - We first clean our string input and remove any white-spaces using the strip() method in Python. After this, we split each instruction into a list of it's component opcode, registers, shamt and funct values using the split() method in Python.
  - Next we check for whether the opcode in our split list is present in any one of our 5 string types.
  - Once we identify that the opcode belongs to a certain string type, we map it's corresponding registers, function values, shift amount to their binary values.
  - We then concatenate these binary strings to output the final machine code corresponding to the instruction code.

- Disassembler

  - Our Disassembler program is designed such that it takes machine code as an input and returns the assembly code corresponding to it.
  - In order for it to work we have coded the program such that it takes input in the format 0X' Hexadecimal Code'. After this we split the input at 0X and use only the hexadecimal machine code to convert to assembly language . This hexadecimal code is converted to 32 bit binary code.
  - In case the length turns out to be less than 32 bits we use the zfill function to make the length 32 bits by adding zeros.

After getting the 32 bit binary code we segregate on the basis of the most significant 6 bits, which is also called the opcode.

- In case opcode is =0, we conclude that it is a r type instruction. In case the opcode is the same as the opcode defined for the two j type instructions, we conclude that instruction is J type. In all other cases the instruction will be I type.
- After classifying as R/J/I type instruction we use the functional code and to find which R type instruction it is. We use a lot of conditions to land on the final classification of the instruction.
- After narrowing down our search to the type of instruction, we know the instruction format for each type. We then perform a one-to-one register mapping and opcode, funct mapping and concatenate the respective strings to print the instruction in assembly code.

- GUI Implementation
  - For GUI we have used the python Tkinter library. We have made the GUI user interactive and friendly. The first main window allows the user to choose between the assembler and disassembler.
  - Once the user chooses an option, a new window opens up where the user can input the code in the displayed text box. Next, the user can press the convert button to get their required result.
  - We are printing the result in the GUI only. To make it look appealing, we have added labels, changed the font and

applied padding to the elements.

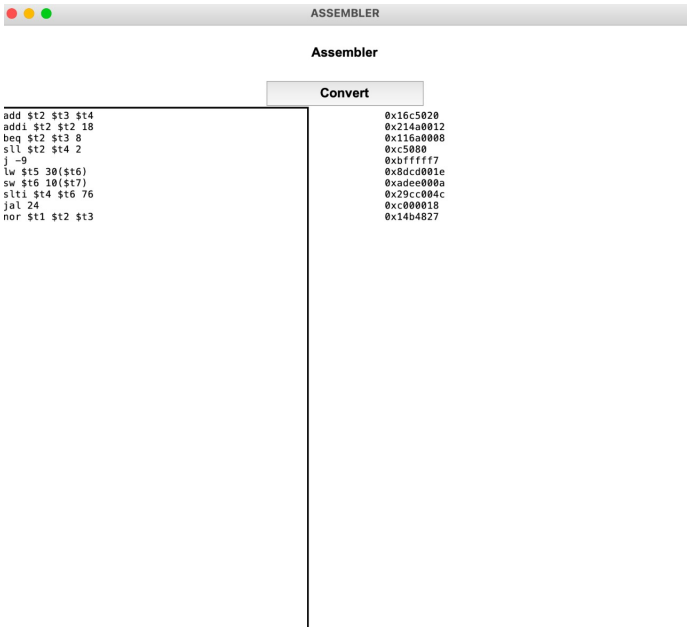
- Notably, we have been able to implement the use of negative immediates and offset values in our program.

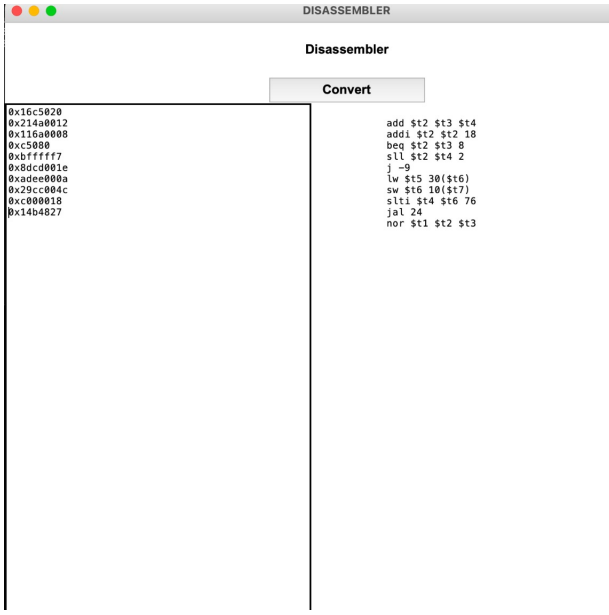
0.6 Testing and Experiments

- We tested our Assembler and Disassembler on the following set of test-cases. It goes without saying that any combination of the instructions can be used as test cases for our program.

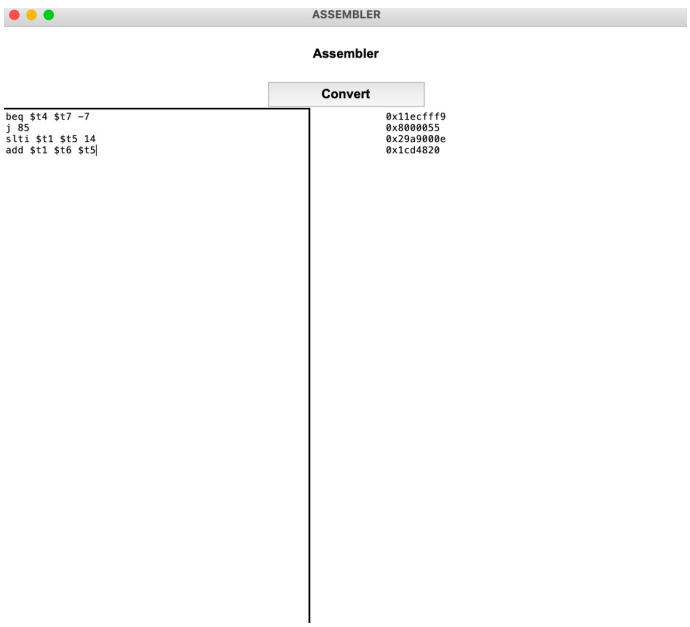
- We also proved the correctness of our Assembler and Disassembler Program by providing the machine code which is the output of the Assembler as the input of the Disassembler. The snippets are attached below.

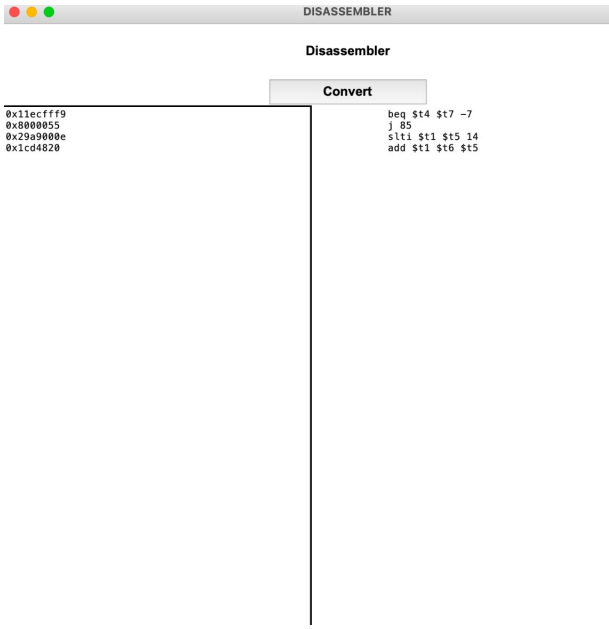
- Program 1





- Program 2





0.7 Limitations

- We need a perfect input which is exactly along the lines of syntax. Any wavering from this would not give an output.
- The data card created by us for our model is not completely exhaustive and contains only the instructions that are very often used.

- We have not given the option for inter-conversion of the machine code from binary to hexadecimal and visa versa.

0.8 Conclusion

- To summarize, we have successfully built an assembler as well as a disassembler that converts assembly language into machine code and visa versa. This involves a user friendly GUI through which we can type assembly code or machine code and get to see the corresponding output.

- Given more time we could have made a much more exhaustive data card and even tried to incorporate pseudo instructions conversion to machine code which would make the task of the user even more easy.

- We have also learnt in the course about Jump instructions and labels and if we were given more time we could have coded to involve usage of labels as well.

- We could even try to optimize the GUI even more and include the mechanisms of conversion with place even for involving the mechanisms in which we deal with hazards.

- Our model is mainly made keeping in mind the application of interactive learning and hence we have, made the interface very interactive and student friendly. In the future we can fine-tune this and develop it into an easy to use and accessible tool for learning.

0.9 References

- https://github.com/SamuelGong/MIPSAssemblerAndDisassembler
- http://alumni.cs.ucr.edu/ vladimir/cs161/mips.html
- https://github.com/robre/MIPS-Disassembler/blob/master/mipsdisassembler.p
- https://opencores.org/projects/plasma/opcodes