# “Design of Assembler for any 10 instructions of MiniMIPS using Python”

**Special Assignment Report**

*Submitted in Partial Fulfillment of the*

*Requirements for completion of*

**Course on**

**2EC601 Computer Architecture**

By

## Chunara Monish Siraj

## Divyansh Rai

## (18BEC021 and 18BEC027)



**Department of Electronics and Communication Engineering,**

**Institute of Technology,**

**Nirma University,**

**Ahmedabad 382 481**

**April 2021**

**CERTIFICATE**

This is to certify that the Comprehensive Evaluation Report entitled “Design of Assembler for any 10 instructions of MiniMIPS using Python” submitted by Mr. Chunara Monish Siraj (18BEC021) and Mr. Divyansh Rai (18BEC027) towards the partial fulfillment of for completion of Course on 2EC601 Computer Architecture is the record of work carried out by them individually.

**Date:** April 16, 2021

**Faculty Coordinator:** Dr. Dhaval Shah

### Undertaking for Originality of the Work

We, Chunara Monish Siraj, 18BEC021 and Divyansh Rai, 18BEC027, give undertaking that the Comprehensive Evaluation Report entitled “Design of Assembler for any 10 instructions of MiniMIPS using Python” submitted by us, towards the partial fulfillment of for completion of Course on 2EC601 Computer Architecture, is the original work carried out by us and we give assurance that no attempt of plagiarism has been made. We understand that in the event of any similarity found subsequently with any other published work or any report elsewhere; it will result in severe disciplinary action.



Signature of the Student

Date: April 16, 2021

Place: Ahmedabad

### Abstract

Assembler is the tools which allows the programmer to leverage the overall capabilities of the processor while not worrying about the exact machine codes corresponding to the desired instruction. An assembler for MiniMIPS would come in handy to convert the assembly programs developed as application programs for understanding the underlying architecture, into binary instructions.

The assembler and the simulator designed would be helpful in understanding the ISA of MiniMIPS processor and the use of various instructions. These tools would also prove to be helpful for developing, verifying and tracing the execution of MiniMIPS programs written in assembly language. In addition to these, the execution and implementation of various instructions on MiniMIPS processor can also be studied through the designed execution procedure in the simulator.

We developed algorithm for conversion of source code to machine code by tackling all the challenges that occurred due to different operand specifying styles of instructions.

We designed assembler and simulator for various R type, I type and J type instructions for MiniMIPS. Our simulator has features like writing a source code and loading a .txt file, step execution, creating symbol reference table, displaying PC values and memory contents, provision for comments, displaying error when reference is not available.

Also we have tried to keep our simulator user friendly and easy to use unlike the PCSpim that has a complex and primitive kind of GUI.

INDEX

|  |  |  |  |
| --- | --- | --- | --- |
| **Chapter No.** | Title | | **Page No.** |
|  | Abstract | | **4** |
|  | Index | | **5-6** |
|  | List of Figures | | **7** |
|  | List of Tables | | **8** |
|  | Nomenclature | | **9** |
| **1** | Introduction | | 10-12 |
|  | 1.1 | Prologue | 10 |
|  | 1.2 | Importance of the Topic | 11 |
|  | 1.3 | Objectives the Report | 11 |
|  | 1.4 | Scope of the Report | 12 |
|  | 1.5 | Organization of the rest of the Report | 12 |
| **2** | Literature Review | | 13-14 |
|  | 2.1 | Work in the area of the topic | 13 |
|  | 2.2 | Details of Tools and Technologies used | 14 |
|  | 2.3 | Assembler and its functions | 14 |
| **3** | Details related to the topic | | 15-24 |
|  | 3.1 | About the designed Assembler | 15-16 |
|  | 3.2 | Methodology and Algorithms | 17-18 |
|  | 3.3 | Simulator Features and Description of the simulator window | 18-19 |
|  | 3.4 | Steps for simulation | 19 |
|  | 3.5 | Discussion | 20-21 |
|  | 3.6 | Assembler Outputs | 21-22 |
|  | 3.7 | Simulation Results | 22-24 |
| **4** | **Conclusion** | | 25 |
| **5** | **Future Scope** | | 26 |
| **6** | **References** | | 27 |
| **7** | **Appendix** | | 28 |

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **Figure**  **No.** | **Title** | **Page No.** |
| 1 | PCspim Simulator | 14 |
| 2 | MiniMIPS instruction formats | 16 |
| 3 | Simulator Window | 20 |
| 4 | The object file created on assembling the source code | 22 |
| 5 | Output of Assembler (Machine Code) | 23 |
| 6 | Simulation result for the demo code (After assembling and before executing) | 23 |
| 7 | Simulation result for the demo code (intermediate) | 24 |
| 8 | Simulation result - Execution completion message | 24 |
| 9 | Final simulation result | 25 |
| 10 | Reference to label not available error | 25 |

**LIST OF TABLES**

|  |  |  |
| --- | --- | --- |
| **Table No.** | **Title** | **Page No.** |
| 1 | Instructions considered for the Assembler design | 17 |
| 2 | Instruction Opcodes and Function fields | 17 |

## NOMENCLATURE

|  |  |
| --- | --- |
| **Nomenclature** |  |
| MiniMIPS | Hypothetical processor resembling real MIPS processor and supporting a subset of Instructions of MIPS processor |
| .o | Object file |
| $ra | Specification of register in MiniMIPS assembly instruction |
| Opcode | Operation code for machine |
| Source code | The file containing MiniMIPS assembly instruction |
| Object file | The machine code file generated by MiniMIPS assembler |
| SPIM | Conventional name for MIPS simulator |
|  |  |
| **Abbreviations** |  |
| ISA | Instruction Set Architecture |
| PC | Program Counter for the processor |
| GUI | Graphical User Interface |
| i.e. | That is |

**Chapter 1**

**Introduction**

**1.1 Prologue**

A processor or a digital device understands only the binary language. The instructions to be executed are also to be given in binary language to the processor. These binary instructions or the machine instructions have the information of the controls, the addresses of operands, the operand itself or memory address, everything encoded in zeros and ones. However, for a programmer, it is difficult to understand and remember machine instructions and to program in binary language (low-level language) [1].

Various middle-level and higher-level languages were developed for easing the task of programming. These languages allow the programmer to use various pre-defined symbols or mnemonics to develop programs. Assembly language is a middle-level programming language as it is closer to machine language in terms of convertibility and is still human understandable [2]. Higher-level languages like C, C++, Python etc. are further more easily understandable while they are comparatively more complex in terms of convertibility to machine language.

A programmer would develop the program in assembly or higher-level language but this program must be converted to machine language for actual implementation on the hardware. Assembler is the tool to convert assembly language to machine language while compiler or interpreter is the tool to convert the higher-level language into machine language.

Developing programs in assembly would have an advantage of customized optimization by the programmer as the conversion of assembly instruction into machine instructions is quite straight forward as compared to conversion of higher-language instruction into machine instruction. Assembly language provides a set of mnemonics and the defined methods for specifying operands. The main task of the assembler is to understand the instruction from its mnemonic and the mentioned operands in the assembly instruction and convert it into machine instruction by replacing the instruction mnemonic with its corresponding opcodes and mentioning the operands along with the opcode.

**1.2 Importance of the Topic**

Assembler is the tools which allows the programmer to leverage the overall capabilities of the processor while not worrying about the exact machine codes corresponding to the desired instruction. Assembly language for a target hardware maps the hardware functionalities to instruction mnemonics and the assembler converts them into the machine understandable format.

An assembler for MiniMIPS would come in handy convert the assembly programs developed as application programs for understanding the underlying architecture, into binary instructions.

Apart from MiniMIPS, any assembler in general converts the developed assembly program into binary or hexadecimal instruction format that can be downloaded on a target hardware to obtain the desired functionality.

**1.3 Objectives of the Report**

1. **Developing an assembler for MiniMIPS instructions**

The prime objective is to develop a python program that takes a source file having MiniMIPS assembly instructions and converts the assembly instructions into the corresponding machine instructions. These machine instructions will then be stored in the form of an object file which can be used to program the target processor.

1. **Designing a GUI simulator for MiniMIPS**

In addition to the design of assembler, a GUI based MiniMIPS simulator is also to be designed. This simulator would allow to write or load source code, convert it into machine code and also execute the code to update the registers and memory. This simulator can be used for functional verification of the MiniMIPS assembler code.

**1.4 Scope of the Report**

The assembler and the simulator designed would be helpful in understanding the ISA of MiniMIPS processor and the use of various instructions. These tools would also prove to be helpful for developing, verifying and tracing the execution of MiniMIPS programs written in assembly language. In addition to these, the execution and implementation of various instructions on MiniMIPS processor can also be studied through the designed execution procedure in the simulator.

**1.5 Organization of the Rest of the Report**

The rest of the report is organized as follows; Chapter 2 describes about the work in the area of the topic as well as the details about the tools and technology used. This is followed by Chapter 3 which briefs about the designed assembler, simulator and the implementation methodologies for the both. Chapter 4 describes the concluding remarks followed by the future scope in Chapter 5. Finally the report ends with References and appendix at the end.

**Chapter 2**

**Literature Review**

**2.1 Work in the area of the topic**

Various simulators have been developed in the past for assembling and verifying MiniMIPS assembly programs. SPIM (reverse of MIPS) is the conventional name given MIPS simulators. It is a simulator for the MIPS R2000/R3000 assembly language. The instruction set supported by this simulator is larger than that of the MiniMIPS and hence they are not targeted specifically for MiniMIPS [1].

SPIM simulator is available in three versions as PCSpim, xspim and spim for windows machines, Mac OS and Unix.Linux systems respectively. Figure 1 shows the user interface of PCSpim. The assembly program and the execution is displayed in four panels in the PCSpim window. The top panel shows the register contents, the second panel shows the program text segment and the program data segment in displayed in the third panel. The messages are shown in the fourth panel. The default I/O mode of PCSpim is via a system call instruction having the mnemonic *syscall* [1]*.*

Though PCSpim is robust in terms of simulation and functionality, it is not targeted specifically for MiniMIPS. And although it provides various panels for various fields, sometimes it becomes tedious for simultaneously understanding the updating of the panels while execution. These makes PCSpim less user-friendly to use.

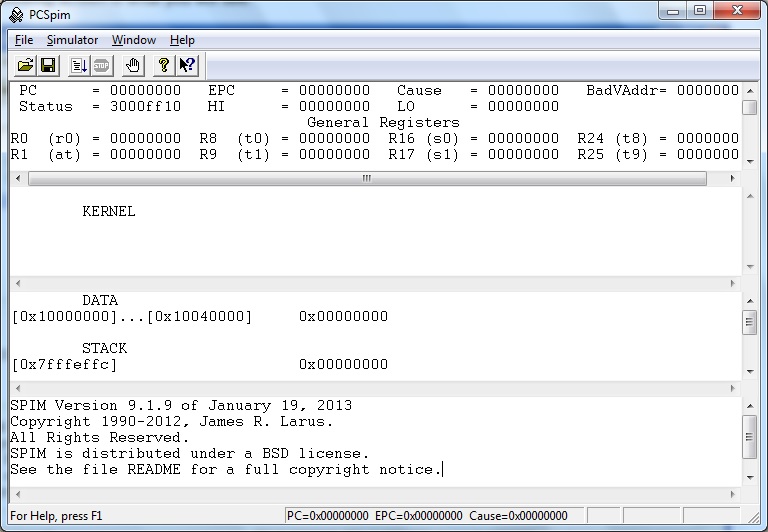


Figure 1: PCspim Simulator

**2.2 Details of Tools and Technologies used**

For developing the assembler and execution programs, Python 3.7.3 was used. The program was developed in Virtual Studio code IDE. The GUI for simulator was designed using Qt GUI design Framework for Python. For developing the assembler and the simulator, Object Oriented Programming approach was used.

The memory foot-print of the assembler and simulator together was observed to be 48 kB for the python program and the executable file generated out of the code was found to be 34 MB in size.

**2.3 Assembler and its functions**

As per [1], the assembler is defined as: “An assembler reads a source file containing assembly language program and accompanying information (assembler directives or certain bookkeeping details eg. debug statements) and in the process of producing the corresponding machine language program, it:

1. Maintains a symbol reference table consisting name-address correspondences
2. Builds the program's text and data segments
3. Forms an object file containing header (size of text/data segment), text, data, and relocation information
4. Checks for syntax and errors
5. Ignores comments
6. Understands assembler directives
7. Works on pseudo instructions and macros
8. Understands various addressing modes

Usually, an assembler works in two passes to resolve the issue of forward referencing. Forward referencing refers to the use of identifiers or labels before their declarations. Hence, the main objective of the first pass is to build the symbol reference table and in the second pass, the assembly to machine code conversion is carried out.

**Chapter 3**

**Details related to the topic**

**3.1 About the designed Assembler**

MiniMIPS assembly instructions are classified as Register type (R-type), Immediate type (I-type) and Jump type (J-type) instructions as per the instruction format. The assembler was designed for various R-type, I-type and J-type instructions. Table 1 gives the gist of the instructions used. Although as per the title, only 10 instructions were to be used, various instructions were found to be of similar type and hence they were also included. Figure 2 depicts the instruction format for R-type, I-type and J-type MiniMIPS assembly instructions. It can be observed that as MiniMIPS is based on RISC philosophy, all the instructions are of uniform length i.e., 32-bits.

Table 2 shows the opcode (and/or function fields) of the instructions in the instruction format for distinguishing the instructions. These op-codes were used in the assembler program as the op-code reference table. In addition to this, for getting the corresponding addresses of the operand registers in the instruction, register address reference table was also prepared as per the register file of MiniMIPS and it was provided in the assembler code. In addition to these features, the designed assembler is also capable of assembling procedure calls (as jal instruction is included), it provides the provision for comments as well as it also shows an error message if the reference to particular label used is not available in the code. For signed arithmetic, signed offset and PC relative addressing appropriate algorithms for twos complement and sign extension were used.

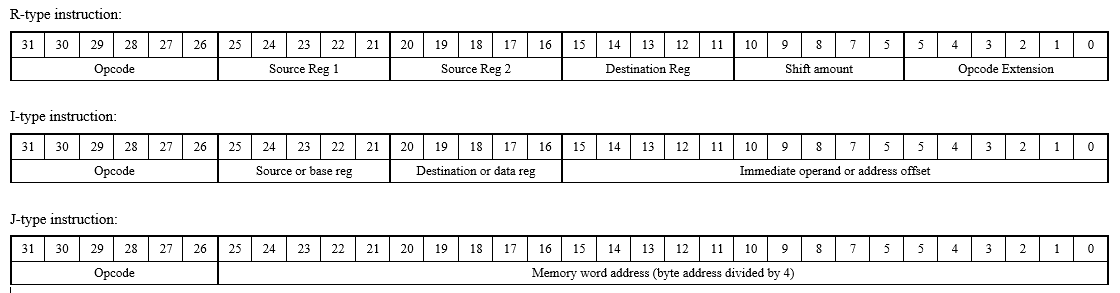


Figure 2: MiniMIPS instruction formats

Table 1: Instructions considered for the Assembler design

|  |  |  |
| --- | --- | --- |
| **R-type** | **I-type** | **J-type** |
| add | addi | j |
| sub | andi | jal |
| and | ori |  |
| or | xori |  |
| xor | lw |  |
| nor | sw |  |
| jr | beq |  |
| slt | bne |  |
|  | bltz |  |

Table 2: Instruciton Opcodes and Function fileds (Blank means not applicable)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Mnemonic** | **Opcode** | | | | | | **Opcode Extension (Function)** | | | | | |
| add | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| sub | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| and | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| or | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| xor | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| nor | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| jr | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| slt | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| addi | 0 | 0 | 1 | 0 | 0 | 0 |  |  |  |  |  |  |
| andi | 0 | 0 | 1 | ` | 0 | 0 |  |  |  |  |  |  |
| ori | 0 | 0 | 1 | 1 | 0 | 1 |  |  |  |  |  |  |
| xori | 0 | 0 | 1 | 1 | 1 | 0 |  |  |  |  |  |  |
| lw | 1 | 0 | 0 | 0 | 1 | 1 |  |  |  |  |  |  |
| sw | 1 | 0 | 1 | 0 | 1 | 1 |  |  |  |  |  |  |
| beq | 0 | 0 | 0 | 1 | 0 | 0 |  |  |  |  |  |  |
| bne | 0 | 0 | 0 | 1 | 0 | 1 |  |  |  |  |  |  |
| bltz | 0 | 0 | 0 | 0 | 0 | 1 |  |  |  |  |  |  |
| j | 0 | 0 | 0 | 0 | 1 | 0 |  |  |  |  |  |  |
| jal | 0 | 0 | 0 | 0 | 1 | 1 |  |  |  |  |  |  |

**3.2 Methodology and Algorithms**

The entire assembler and simulator design is divided into three algorithms, the first algorithm for the first pass to build the symbol reference table. The second algorithm for second pass of assembler to convert the assembly instructions to machine codes. The third algorithm is the execution part and the simulation flow. All the three algorithms are explained as follows:

1. **Algorithm for First Pass of Assembler**

**Step 1:** Read the source code from the file line by line

**Step 2:** Identify if the read line is a label, an instruction, a comment or a blank line

**Step 3:** If the line is a declaration of a label, the label and its corresponding address (PC value) are stored in the symbol reference table dictionary. If it is not a label declaration then continue

1. **Algorithm for Second Pass of Assembler**

**Step 1:** Read the source code from the file line by line

**Step 2:** Identify if the read line is a label, an instruction, a comment or a blank line

**Step 3:** If the read line is an instruction then the instruction mnemonic and the operands are separated and appropriate Instruction format method (R-type, I-type and J-type) is called

**Step 4:** Generate corresponding machine code by fetching corresponding opcode from opcode reference table, get the register addresses from the register address reference table, PC relative address wherever required, label address from the symbol reference table, perform 2s complement if necessary, etc. If label reference not available in symbol reference table, the display an error message.

**Step 5:** Store the machine code in the object file

1. **Assembler for Execution and Simulation Flow**

**Step 1:** Read a line from the object file, identify the instruction by comparing the opcodes

**Step 2:** Obtain the value of source and base registers from the register file as per the machine instruction

**Step 3:** As per the opcode, (and/or function field wherever required), perform operation and update memory as well as PC

**Step 4:** Update the destination registers and memory contents

**Step 5:** At the end of program, display the ‘Execution completion’ message otherwise read next line and repeat steps 2 to 5

**3.3 Simulator features and Description of the simulator window**

The GUI based simulator has the following features:

* Writing a source code (MiniMIPS assembly code)
* Loading a source code stored in .txt format
* Assembling and viewing machine instructions
* Observing the PC and its updating
* Observing the symbol reference table generated in first pass
* Step execution for debugging and tracing the execution flow
* Execution the whole program at once
* Observing the contents of all the 32 registers of MiniMIPS
* Observing the memory contents (and hence stack contents)
* Getting “Execution completion” message
* Getting “Reference to label not available” error

Figure 3 depicts the designed simulator window and various panels of the simulator. The tool bar has buttons to open file, assemble the source file, step execute and run the whole program. The memory panel displays 8 bytes of data starting from the base address. The base address is to be mentioned in integers and on pressing the *show* button, the updated memory contents will be visible. On assembling the source code, the symbol-reference table is displayed along with the zero register and stack pointer register getting their respective values. ($zero = 0 and $sp = 0x7FFC). Here, $sp is given value 0x7FFC instead of 0x7FFFFFFC because the memory is considered to be 64kB (=216) in size. Hence, 16 address bits only. The memory is considered of 64kB size instead of 4GB (=232) because on allocating such a large amount of memory (4 GB) to the program, the program throws a memory error saying that such a large amount of memory cannot be allocated at once. Hence, for this case 64kB memory is used.

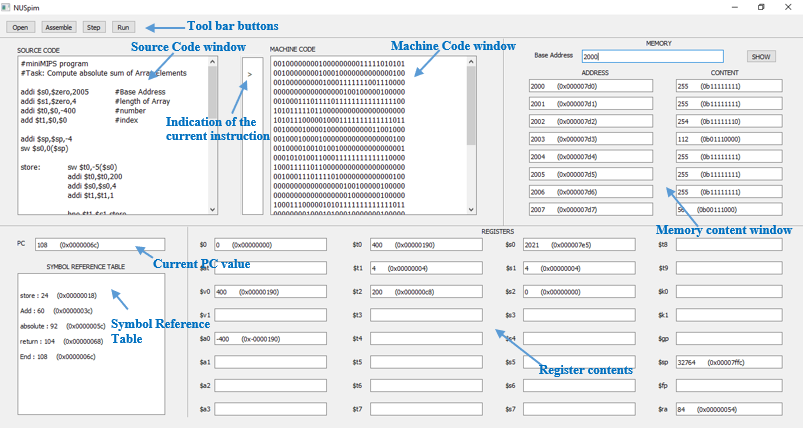


Figure 3: Simulator Window

**3.4 Steps for simulation**

The basic steps to be followed for loading and simulating a MiniMIPS program is as follows:

**Step 1:** Open a source file or type in the source code (MiniMIPS assembly code)

**Step 2:** Assemble the source file by pressing the *assemble* button to get the machine code in the machine code window. If “Reference to label not available” error is shown while assembly, check the code for a missing label declaration.

**Step 3:** Step execute or run the program. Check the ‘>’ pointer which points to the currently running instruction

**Step 4:** Check and verify the values of various registers, memory contents and PC value after the instruction execution

**Step 5:** On successful completion of the program, the “Execution completion” message will be displayed. Verify the final results

**3.5 Discussion**

For the demonstration purpose, a demo program is used. The program is to compute the absolute sum of the array elements. The MiniMIPS assembly program is as follows:

*#miniMIPS program*

*#Task: Compute absolute sum of Array Elements*

*addi $s0,$zero,2005 #Base Address*

*addi $s1,$zero,4 #length of Array*

*addi $t0,$0,-400 #number*

*add $t1,$0,$0 #index*

*addi $sp,$sp,-4*

*sw $s0,0($sp)*

*store: sw $t0,-5($s0)*

*addi $t0,$t0,200*

*addi $s0,$s0,4*

*addi $t1,$t1,1*

*bne $t1,$s1,store*

*Add: lw $t2,-5($s0)*

*add $a0,$a0,$t2*

*addi $s0,$s0,4*

*addi $t1,$t1,1*

*bne $t1,$s1,Add*

*jal absolute*

*slt $s2,$v0,$zero*

*j End*

*absolute: sub $v0,$zero,$a0*

*bltz $a0,return*

*add $v0,$a0,$zero*

*return: jr $ra*

*End:*

*lw $s0,0($sp)*

*addi $sp,$sp,4*

*add $t1,$zero,$zero*

*add $a0,$zero,$zero #initializing sum as zero*

*Add: lw $t2,-5($s0)*

*add $a0,$a0,$t2*

*addi $s0,$s0,4*

*addi $t1,$t1,1*

*bne $t1,$s1,Add*

*jal absolute*

*slt $s2,$v0,$zero*

*j End*

*absolute: sub $v0,$zero,$a0*

*bltz $a0,return*

*add $v0,$a0,$zero*

*return: jr $ra*

*End:*

In this program, first of all the array elements {-400, -200, 0 and 200} are stored in memory location starting from address 2000. For storing these numbers, a loop labelled as *store* is used, wherein constant value 200 is added to the number to be stored (*$t0*) in every iteration for 3 times. Along with this, the base address is incremented by 4 so that the next value can be stored at next memory location. For storing the numbers, Big-Endian Storage format is used. As the base address is modified in the loop, it is Pushed into the stack before the loop and Popped from the stack after the first loop.

Once, the loop terminates which means the array elements are stored in the memory, another loop starting from label *Add* is implemented where the stored array values are loaded from memory and added to *$a0* in each iteration to find the sum of array elements. Once the loop terminates, the sum is passed to Procedure to find the absolute value of the sum by using the *jal* instruction. In the procedure, the logic for finding absolute value is written where the result will be the value of *$a0* if the value passed in *$a0* is negative, otherwise the result will be same as *$a0.* The result is stored in *$vo* register and the control is returned to the next instruction after procedure call using *jr $ra* instruction.

For the verification purpose, the slt instruction is used which will indicate whether the obtained result in *$v0* is positive or negative. Finally, the program is terminated by jumping to the end using unconditional jump *j* instruction.

**3.6 Assembler Outputs**

Figure 4 shows the object file ‘program.o’ file created after assembling the source code. Figure 5 shows the machine language program corresponding to the demo assembly program. The generated machine instructions were compared to the opcode table and register address reference table and they were found to be correct.

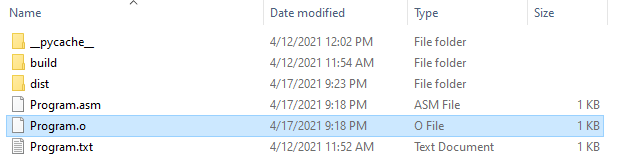
****

Figure 4: The object file created on assembling the source code

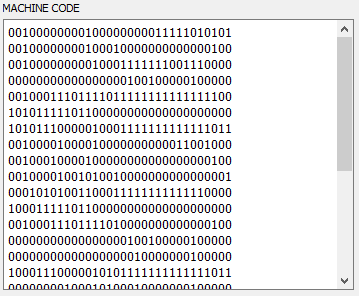
****

Figure 5: Output of Assembler (Machine Code)

**3.7 Simulation Results**

The demo code was step executed and the various intermediate steps are shown in Figure 6 and Figure 7. Figure 8 shows the “Execution completion” message shown at the end of the program. Figure 9 shows the “Reference to label not available” error when the *End* label in the code was removed from program.

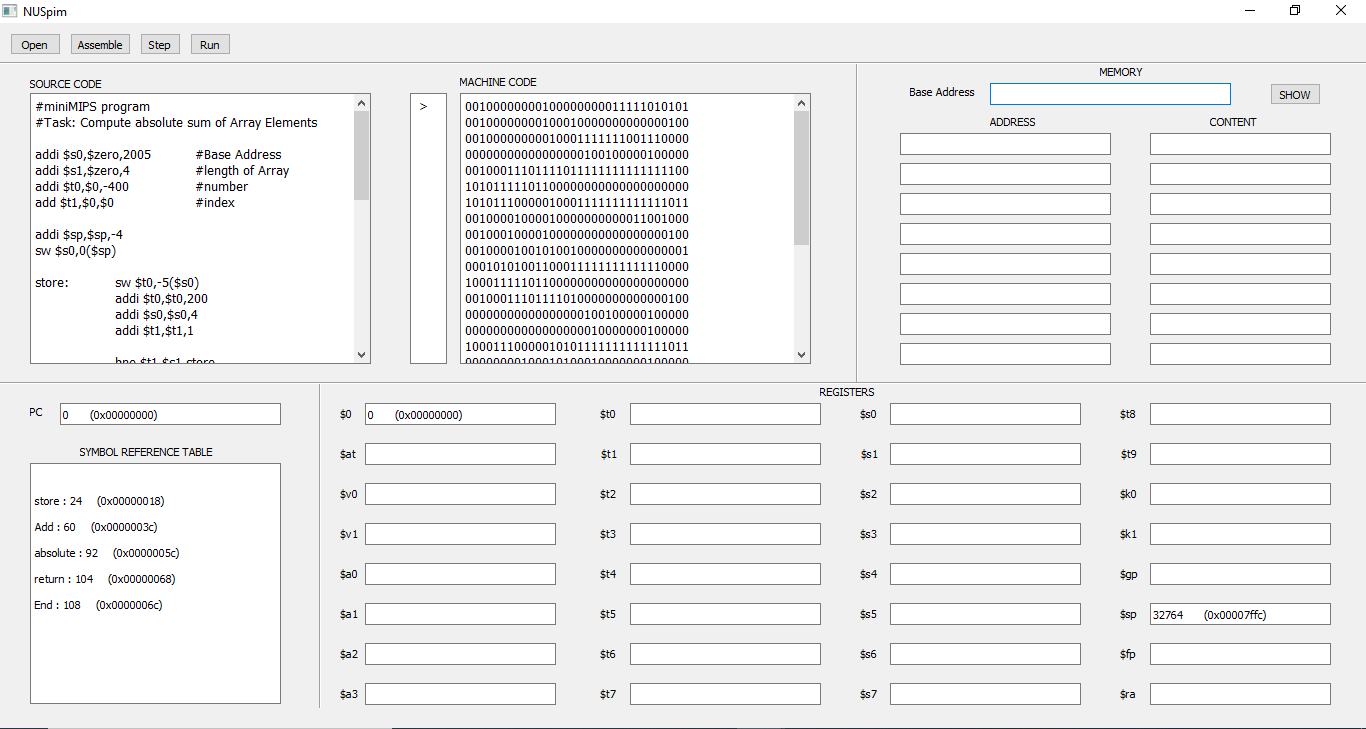


Figure 6: Simulation result for the demo code (After assembling and before executing)

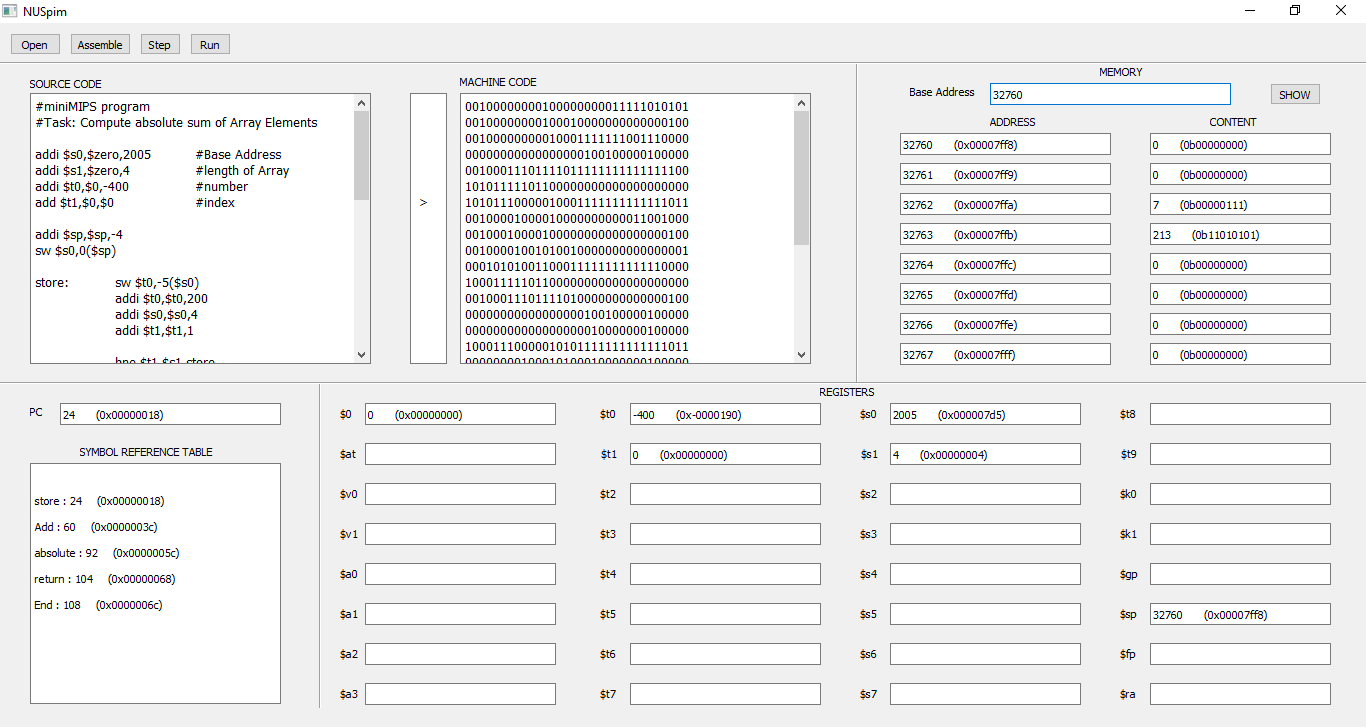


Figure 7: Simulation result of the demo code (intermediate)

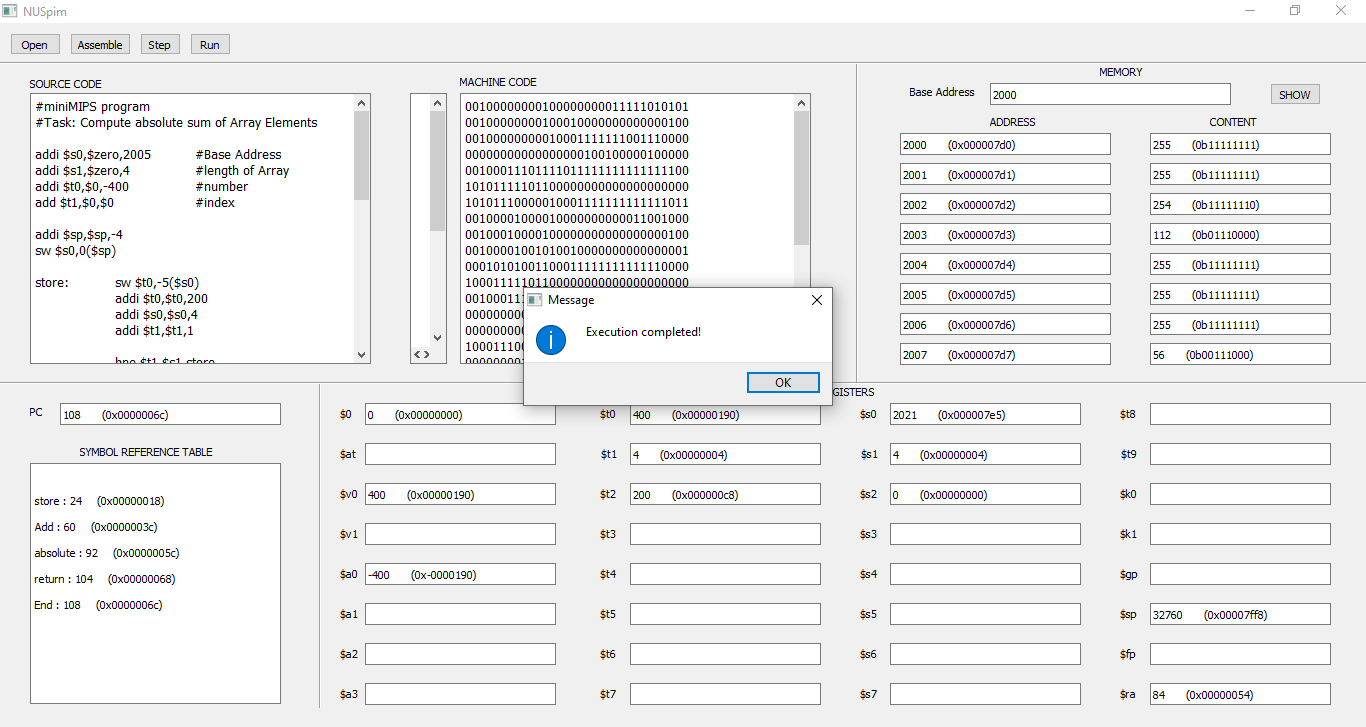


Figure 8: Simulation result - Execution completion message

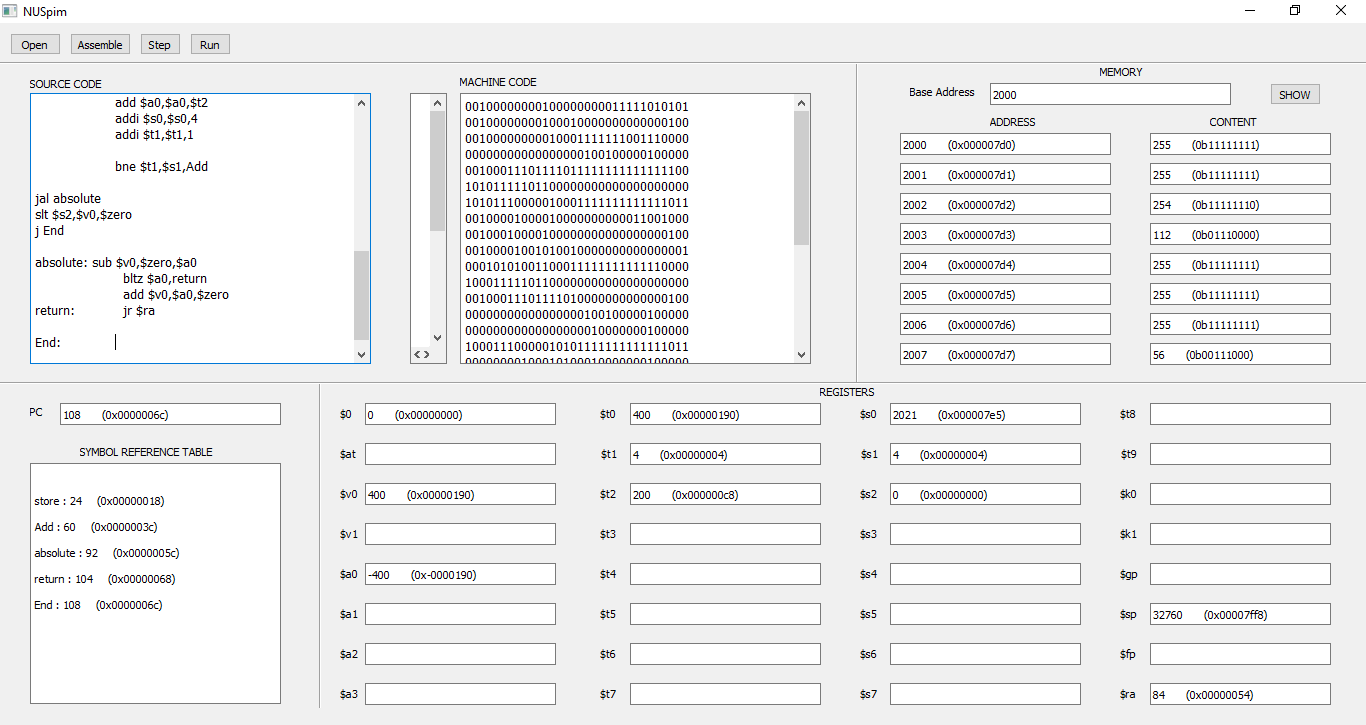


Figure 9:Final simulation result

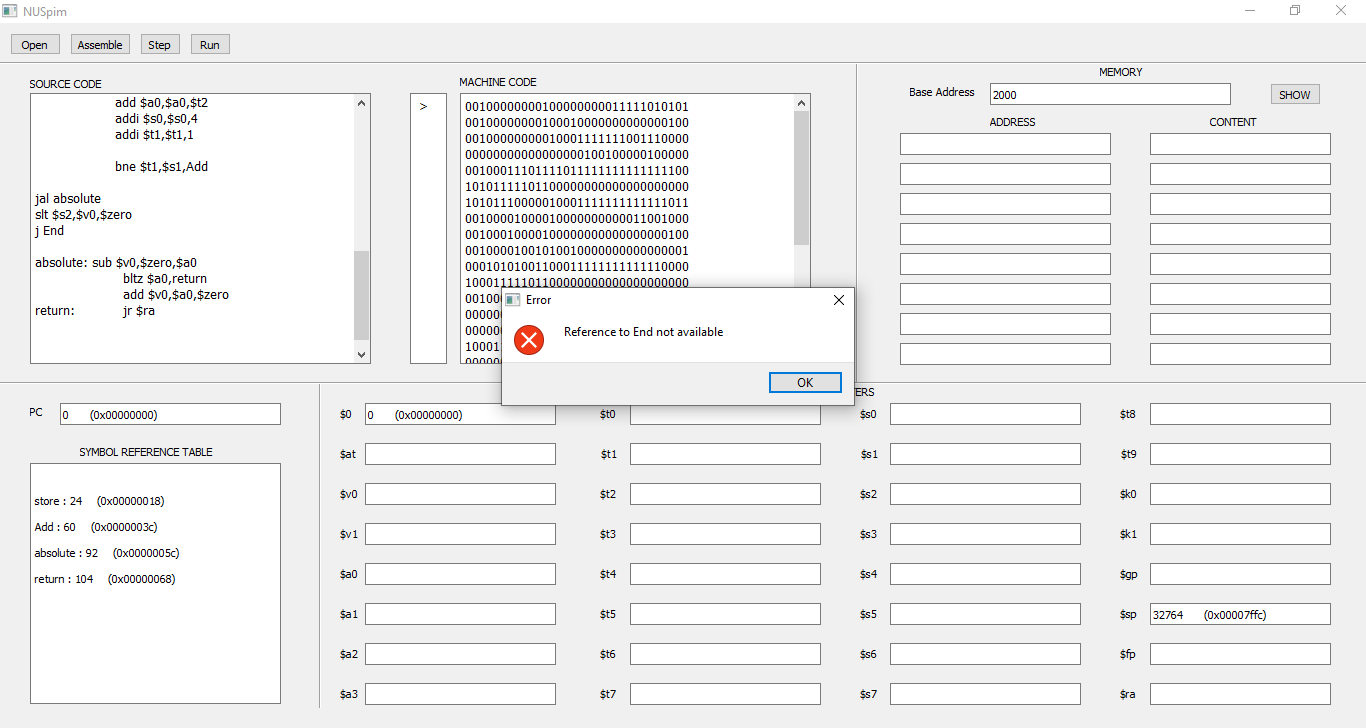


Figure 10: Reference to label not available error

**Chapter 4**

**Conclusion**

An assembler for total 19 instructions of R, I and J type for MiniMIPS processor was developed. In addition to assembler a simulator with capabilities of showing machine code, PC value, Symbol reference table, memory addresses and register values was also designed. The algorithms for conversion of source code to machine code were developed. The designed simulator is capable to execute the program in steps or the whole program at once. The simulator also shows the register and memory content updated after every instruction execution which can be used for debugging purpose.

It was observed that the assembler designers face challenges due to different operand specifying styles of instructions which we have learnt to tackle. In addition to this we also learnt how to design a simulator for given ISA of a particular processor.

PCSpim is one of the MIPS simulator available in market. It incorporates MIPS ISA of which MiniMIPS ISA is a subset. It was tried to keep the simulator user friendly and easy to use unlike the PCSpim which has a complex and primitive kind of GUI.

**Chapter 5**

**Future Scope**

For the design of the assembler and simulator, only 19 instruction were considered. However, the design can be extended to all the instruction of MiniMIPS for making the assembler and simulator robust. Also, the provision for assembler directives, pseudo-instruction and macros can be included in the assembler design so that the programmer can leverage those facilities.

The design of the assembler and simulator be generalized and the assembler and simulator for any given ISA can be designed. Hence, the designed assembler and simulator forms the basis for understanding and implementing the design of various RISC processors.

**References**

1. Behrooz Parhami, “Assembly language Programs” in *Computer Architecture from Microprocessor to Super Computer*, 1st ed. Oxford, England: Oxford University Press, 2006, pp. 123-135
2. Saiyam Jain, “Introduction of Assembler”, Geeks For Geeks, [online document], 2019. Available: Geeks for Geeks articels online, https://www.geeksforgeeks.org/ introduction-of-assembler/ [Accessed: March 28, 2021].
3. Elearningateria, “Assembler Design”, Elearninggateria, [online document], 2021. Available: https://elearningatria.files.wordpress.com/2013/10/assembler- design .pdf [Accessed: April 01, 2021].
4. Sentdex, “Python GUI Application Development with Python”, *YouTube,* May 25, 2015. [Video file]. Available: https://www.youtube.com/watch?v=JBM E1ZyHiP8&list=PLQVvvaa0QuDdVpDFNq4FwY9APZPGSUyR4, [Accessed April 10, 2021].

**Appendix**

The entire program for Assembler, Execution and GUI simulator consumed more than 1000 lines. Therefore, it is shared over google drive which can be accessed through the following link:

<https://drive.google.com/file/d/1tPmdxIvfbVlfNvz33zxaE8fX9djgneih/view?usp=sharing>