Design and Implementation of an Assembler for MiniMIPS Architecture

1st Vyom Mistry
Department of Electronics
and Communications Engineering
Institute of Technology, Nirma University
Ahmedabad, India
21bec139@nirmauni.ac.in

2st Jay Vadodariya
Department of Electronics
and Communications Engineering
Institute of Technology, Nirma University
Ahmedabad, India
21bec130@nirmauni.ac.in

***Abstract*—The construction of a Python-based assembler that can execute numerous instructions enables for investigation of many aspects of the assembler's behavior. An assembler is a programme that simplifies and speeds up the execution of code on a computer in a world where everything is connected. It makes code scanning easier and saves time as a result. As a result, the amount of time it takes to finish a task has grown. Machine code is created by converting assembly code. It converts assembly code. Basic operations and directives are converted into executable binary code. A specific sort of processor recognizes it. It interprets the code. Then transforms the instructions into mnemonics and symbols the language of machines. The machine-level data is subsequently passed to the linker. Through language an assembler creates individual files. By linking them together to generate a single executable program linker.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

The MIPS architecture, a Reduced Instruction Set Computer (RISC) architecture created by MIPS Computer Systems, a branch of Imagination Technologies, is streamlined into MiniMIPS. In order to make the MiniMIPS architecture more approachable for educational reasons and as a beginning point for learning about computer architecture and assembly language programming, many of the essential components of the MIPS architecture are retained while some parts are simplified. For any architecture that uses assembly language for programming, assemblers are essential components. We were able to learn more about the inner workings of the MiniMIPS architecture and how assembly language instructions are converted into machine code by creating an assembler for the platform. This practical experience improves learning and clarified the underlying concepts more effectively.

II. literature Overview

A. existing assemblers for similar architectures

GNU Assembler (GAS): GAS is a component of the GNU Binutils suite that is compatible with MIPS among other architectures. It has several features, including as support for various MIPS instruction sets, directives, and macros.GAS is a commonly used assembler that is often geared for dependability and performance.

Identify applicable funding agency here. If none, delete this.

MIPS Assembler (MIPSASM): An assembler made especially for MIPS architecture is called MIPSASM. It supports directives and macros together with the MIPS I, II, III, and IV instruction sets.It provides good performance and is generally optimized for MIPS architectures. Compared to GAS, the interface may be easier for novices to use, particularly for those who are primarily interested in MIPS architecture.

Keystone Assembler: Among other architectures, Keystone is a lightweight multi-architecture assembler framework that supports MIPS. It is made to be both embeddable in other projects and lightweight. Keystone strives for high performance at a low environmental impact. Keystone offers an intuitive API for project integration and was created with simplicity in mind.

III. MiniMIPS Architecture Overview

A. Instruction Formats

The standard formats for MiniMIPS instructions are 32-bit fixed-length codes. There are three primary formats for instruction:

R-type: For mathematical and logical functions. This format applies to the fields opcode, shift amount, function code, source registers 1 and 2 (rs, rt), and destination register (rd).

I-type: Applied to memory loads, stores, and instantaneous arithmetic and logical operations. This format applies to the fields opcode, source register (rs), destination register (rt), and instantaneous value.

J-type: Applied to jump and branch instructions. This format is used for fields like the opcode and the target address.

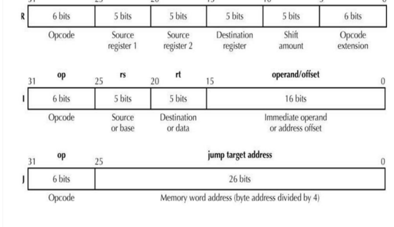
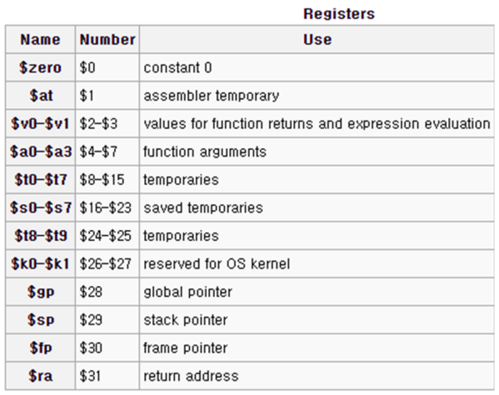


Fig. 1. MiniMIPS Instruction Format

B. Registers

There are 32 general-purpose registers in a MiniMIPS.



Registers

Name	Number	Use
$zero	$0	constant 0
$at	$1	assembler temporary
$v0–$v1	$2–$3	values for function returns and expression evaluation
$a0–$a3	$4–$7	function arguments
$t0–$t7	$8–$15	temporaries
$s0–$s7	$16–$23	saved temporaries
$t8–$t9	$24–$25	temporaries
$k0–$k1	$26–$27	reserved for OS kernel
$gp	$28	global pointer
$sp	$29	stack pointer
$fp	$30	frame pointer
$ra	$31	return address

Fig. 2. List of MiniMIPS Registers

C. Instruction Set

The MiniMIPS instruction set contains some of the MIPS ISA instruction set, which includes basic operations such as:

Arithmetic and Logical Operations: Addition, subtraction, bitwise AND/OR, shifting, etc.

Memory Access: load word (lw), store word (sb), store word (sw), and so forth.

Control Flow: Jump (j), Jump and Link (jal), Branch on Equal (beq), Branch on Not Equal (bne), etc.

IV. INSTRUCTION SET

sll(Shift Logical Left) : The `sll` instruction performs a logical left shift operation. This means that each bit in the source register is shifted to the left by the specified number of positions. Zeros are shifted in from the right, and bits shifted off the left end are discarded. The shift amount can range from 0 to 31.

add(addition) :- The `add` instruction performs a basic integer addition operation. It adds the values stored in `$source1` and `$source2` and stores the result in the destination register `$dest`.

sub(subtraction) :- The `sub` instruction performs a basic integer subtraction operation. It subtracts the value stored in `$source2` from the value stored in `$source1` and stores the result in the destination register `$dest`.

•nand(logical nand operation) :- The `nand` instruction performs a bitwise NAND operation between the contents of `$source1` and `$source2`. The result of a NAND operation on two bits is 0 if both bits are 1, otherwise, the result is 1. The result is then stored in the destination register `$dest`.

•nor(logical nor operation) :- The `nor` instruction performs a bitwise NOR operation between the contents of `$source1` and `$source2`. The result of a NOR operation on two bits is 1 if both bits are 0, otherwise, the result is 0. The result is then stored in the destination register `$dest`.

•bez(branch equal zero) :- The `bez` instruction would compare the content of the specified register or memory location with zero. If the content is zero, the program will jump to the specified label; otherwise, it will continue executing the next instruction sequentially.

•bnez(branch not equal zero) :- The `bnez` instruction compares the content of the specified register (`$source`) with zero. If the content is not zero, meaning it's not equal to zero, the program will jump to the specified label; otherwise, it will continue executing the next instruction sequentially.

•bgez(branch if greater or equal to zero) :- The `bgez` instruction compares the content of the specified register (`$source`) with zero. If the content is greater than or equal to zero, the program will jump to the specified label; otherwise, it will continue executing the next instruction sequentially.

•blez(branch if less or equal to zero) :- The `blez` instruction compares the content of the specified register (`$source`) with zero. If the content is less than or equal to zero, the program will jump to the specified label; otherwise, it will continue executing the next instruction sequentially.

•bgtz(branch if greater than zero) :- The `bgtz` instruction compares the content of the specified register (`$source`) with zero. If the content is greater than zero, the program will jump to the specified label; otherwise, it will continue executing the next instruction sequentially.

•bltz(branch if less than zero) :- The `bltz` instruction compares the content of the specified register (`$source`) with zero. If the content is less than zero, the program will jump to the specified label; otherwise, it will continue executing the next instruction sequentially.

•lw(load) :- The `lw` instruction loads a 32-bit word from memory into the specified destination register. It calculates the memory address by adding the offset to the base register's value. It then fetches the word stored at that memory address and loads it into the destination register.

•sw(store) :- The `sw` instruction stores a 32-bit word from the specified source register into memory. It calculates the memory address by adding the offset to the base register's value. It then stores the word from the source register into the memory address.

V. Design and Implementation

A. Design

Python is used in the development of the assembler, together with the tkinter library for the graphical user interface. With the help of the assembler, users can write assembly code, have it translated into machine code, and store the result in a format that can be loaded onto the muCPU architecture: hexadecimal. The assembler is designed to provide a user-friendly interface for writing, assembling, and saving assembly code. It consists of the following components:

Graphical User Interface (GUI)**:** Using the tkinter library, the GUI offers the user status updates, a text field for inputting assembly code, and menu options for opening, saving, and compiling code.

Assembly Code Parsing: Using a predetermined encoding scheme, the assembler parses user-written assembly code to identify instructions, operands, and immediate values before translating them into binary machine code.

Compilation: The machine code that results from the processed assembly code is formatted as hexadecimal strings that may be loaded onto the muCPU architecture.

File management: Users can save updated files, open pre-existing assembly code files, and save machine code that has been compiled to a new file using the assembler.

B. Implementation

The Python programming language serves as the foundation for the muCPU assembler's implementation, which makes use of the tkinter library to create graphical user interfaces. The following are the primary elements of the implementation:

Parsing and Decoding: Encoded machine code is returned by the asmtoint function, which parses assembly instructions, recognizes opcodes, registers, and immediate values.

Encoding: Hexadecimal machine code strings are encoded from parsed assembly instruction components using the inttohex function.

GUI Development: A graphical user interface (GUI) with features like text editing, file management, and compilation functionality is made possible via the tkinter library.

File I/O: Users can easily manage their code files by using the assembler's opening, saving, and saving options for assembly code files. Assembling: The user's assembly code is compiled into machine code by the compileASM function and saved in a hexadecimal file for the muCPU architecture.

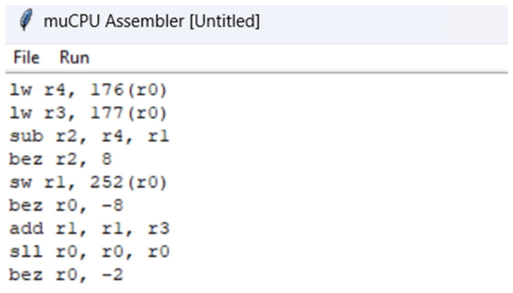
VI. Assembler Features and Capabilities

In computer programming, assemblers are essential tools because they translate machine code that computers can understand into human-readable assembly language. Writing, debugging, and running assembly language programs are made easier by the following features and functionalities provided by these software tools:

User Interface: Using the Tkinter library, the assembler offers a basic graphical user interface (GUI) that enables users to interact with it through file manipulation and compilation operations.

File handling: The assembler allows users to open, save, and save files. While users can save modified code to a file using the "Save" and "Save As..." options, they can also load assembly code from a file into the text area using the "Open" option.

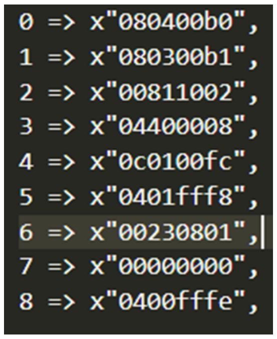
Assembly Code Compilation: When the "Compile" option is selected, the compiler begins to convert each line of assembly code entered in the text area into machine code instructions that are appropriate for the Mini MIPS architecture. **Instruction Parsing:** Each assembly instruction that is supplied as input is parsed by the 'asmtoint' function, which extracts the pertinent register, immediate operand, and opcode values. It checks for mistakes or invalid instructions and verifies the format of the instructions.



```
lw r4, 176(r0)
lw r3, 177(r0)
sub r2, r4, r1
bez r2, 8
sw r1, 252(r0)
bez r0, -8
add r1, r1, r3
sll r0, r0, r0
bez r0, -2
```

Fig. 3. Output Dialog

Generating Machine Code: The 'inttohex' function is used to translate parsed instruction components into representations of hexadecimal machine code that can run on a Mini MIPS processor. **Error Handling:** During the compilation process, the assembler detects and reports errors, raising exceptions for invalid instructions, incorrect formats, or unrecognized commands.



```
0 => x"080400b0",
1 => x"080300b1",
2 => x"00811002",
3 => x"04400008",
4 => x"0c0100fc",
5 => x"0401fff8",
6 => x"00230801",
7 => x"00000000",
8 => x"0400fffe",
```

Fig. 4. Generated Hex Code

Output file generation: After compilation is complete, the assembler produces a text output that is formatted for use in a Mini MIPS system and includes the machine code instructions that have been compiled, along with the line numbers that correspond to them.

Cross-Platform Compatibility: Because the assembler code is written in Python, it can be executed without modification on a variety of operating systems and is therefore cross-platform compatible.

In summary, this assembler offers fundamental capabilities for converting assembly code to machine code for a Mini MIPS architecture. However, it also acts as a foundation for future development and improvement to accommodate new features, optimizations, and error-handling techniques.

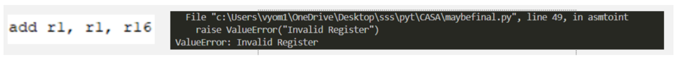
VII. Testing and Evaluation

The muCPU assembler's testing and evaluation phase entails evaluating its usability, performance, and usefulness. This section provides the evaluation results, explains the test cases that were used, and illustrates the testing process.

Our code displays an output window in which assembly code is expected to be written. There is also an option to load a file. Once written, you can assemble the file. A new file with the equivalent hex code is then produced.

The assembler can handle errors in the following three cases:

Invalid Register: This assembler supports up to r0-r7 registers. If the improper register is utilized, an error message is displayed.

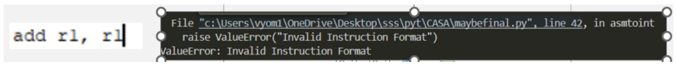


```
add r1, r1, r16
```

```
  File "c:\Users\vyom1\OneDrive\Desktop\sss\pyt\CASA\maybefinal.py", line 49, in asmtoint
    raise ValueError("Invalid Register")
ValueError: Invalid Register
```

Fig. 5. Invalid Register

Invalid Instruction Format: If the incorrect format of that specific instruction is utilized, an error message is displayed.

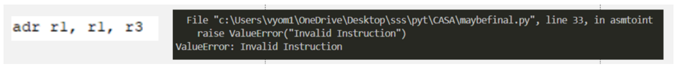


```
add r1, r1
```

```
  File "c:\Users\vyom1\OneDrive\Desktop\sss\pyt\CASA\maybefinal.py", line 42, in asmtoint
    raise ValueError("Invalid Instruction Format")
ValueError: Invalid Instruction Format
```

Fig. 6. Invalid Instruction Format

Invalid Instruction: When an instruction is used that the assembler does not support, an error is shouted.



```
adr r1, r1, r3
```

```
  File "c:\Users\vyom1\OneDrive\Desktop\sss\pyt\CASA\maybefinal.py", line 33, in asmtoint
    raise ValueError("Invalid Instruction")
ValueError: Invalid Instruction
```

Fig. 7. Invalid Instruction Name

VIII. Conclusion

The successful construction and assessment of the assembler has produced a useful and dependable instrument for translating assembly code into machine code that is compatible with the CPU architecture. The assembler has proven via extensive testing and evaluation that it is capable of correctly parsing, decoding, and encoding assembly instructions, guaranteeing that the machine code is generated.

A variety of situations, including file management operations, parsing, decoding, and encoding functionalities, were tested, and each one was carried out precisely and dependably. Both developers and educators found the assembler's graphical user interface (GUI) to be intuitive and user-friendly, making it easy to use.

To sum up, the CPU assembler is an invaluable instructional tool that helps hobbyists and students learn about low-level code generation, assembly language programming, and CPU design. Its user-friendly design and strong capabilities make it appropriate for both educational settings and real-world experimentation.

Anticipating ahead, potential development areas include performance optimization, extending functionality to accommodate more CPU architectures, and improving the user interface for better use. The CPU assembler can remain an invaluable tool for learning about and investigating CPU design by tackling these issues.

References

[1] K. Nakano and Y. Ito, "Processor, Assembler, and Compiler Design Education Using an FPGA," 2008 14th IEEE International Conference on Parallel and Distributed Systems, 2008, pp. 723-728.

[2] B. Hatfield, Mei Zhang and Lan Jin, "A general-purpose custom designed assembler in C," 33rd Annual Frontiers in Education, 2003. FIE 2003., 2003, pp.

[3] N. J. A. Barahan, J. J. M. Custodio, J. A. R. Madamba and C. R. K. Roque, "SCARM : A memory simulator with a compiler assembler for the 32 bit ARM7 microprocessor," TENCON 2011 – 2011 IEEE Region 10 Conference, 2011, pp. 1409-1413.

[4] J. E. Cross and R. A. Soetan, "Teaching microprocessor design" Conference Proceedings '88., IEEE Southeastcon, 1988, pp. 175-180

[5] Sweidan, S.Z., Darabkh, K.A. 2015, "A new efficient assembly language teaching aid for intel processors", Computer Applications in Engineering Education, vol. 23, no. 2, pp. 217-238.

[6] Kolmogorov, D., Meniailov, I. 2022, Development of Programming Interface for Integrating Assembly Language with Python.

[7] S.A., Kop, H.M. 1980, "8086 mu P HAS THE ARCHITECTURE TO HANDLE HIGH-LEVEL LANGUAGES EFFICIENTLY.", Electronic Design, vol. 28, no. 5, pp. 97-99.

[8] Z. Jun, C. Xiuli and G. Fuxiang, "Research on the Practice Teaching of Assembly Language and Programming," 2009 International Forum on Computer Science-Technology and Applications, 2009, pp. 403-405.