Assembler for MIPS Processor

Nishant Parmar
21bec085
Dept. of ECE,
Nirma University,
Ahmedabad, India

Meen Patel
21bec091
Dept. of ECE,
Nirma University,
Ahmedabad, India

Abstract— This paper presents an assembler developed in Python for programming in the MIPS assembly language. The assembler includes fundamental operators and instructions, memory management including subprogram calls, loading and storage, program flow control, and the important task of converting assembly language programs into machine-readable code -Generates an object file containing instructions Thus this effort helps to simplify the MIPS assembly process and increases the efficiency of software development for MIPS processors.

Keywords: Assembler, python, subprogram calls, machine readable codes, object file. MIPS assembly processors

I. Introduction of MIPS

The MIPS architecture, which stands for Microprocessor without Interlocked Pipelined Stages, is a significant player in the world of computer processors.

Core Design Philosophy:

- **Reduced Instruction Set Computing (RISC):** Unlike Complex Instruction Set Computing (CISC) architectures that boast a vast array of instructions, MIPS adopts a RISC approach. This means it focuses on a smaller set of simpler instructions.

- **Benefits of RISC:**
  This focus on simplicity translates into several advantages:

  1. **Smaller Instruction Decoder:** With fewer instructions, the hardware responsible for deciphering instructions (decoder) becomes smaller and less complex.
  2. **Reduced Hardware Complexity:** A simpler instruction set translates to a more streamlined processor design, requiring less hardware overall.
  3. **Potentially Faster Execution:** Simpler instructions can potentially be decoded and executed faster compared to complex CISC instructions.

Data Handling:

- **Load/Store Architecture:** A defining feature of MIPS is its load/store architecture. This means the processor cannot directly access data in memory. Instead, it follows these steps:
  1. **Load:** An instruction explicitly loads the desired data from memory into a register.
  2. **Process:** The processor performs operations on the data using the registers.
  3. **Store:** After processing, the results are stored back to memory from the register.
- **Comparison with CISC:** CISC architectures might allow for direct memory access in some cases, bypassing the load/store approach.

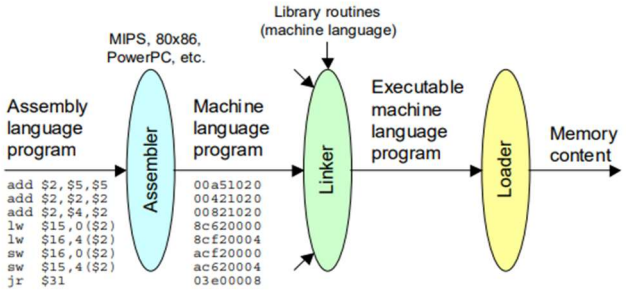
Instruction Set Types:

MIPS instructions set contain mainly 3 types:

- **Register (R-type) Instructions:** These instructions work primarily with data stored in the processor's registers. They involve operations like addition, subtraction, and logical operations performed on register contents.
- **Immediate (I-type) Instructions:** These instructions incorporate constant values directly within the instruction itself. This allows for simpler operations involving a register and a fixed value.
- **Jump (J-type) Instructions:** These instructions are responsible for controlling the flow of the program by directing it to jump to different memory locations. This enables functionalities like loops and conditional execution.

1. **Assembler**:
   - An assembler is a tool that converts **assembly code** (human-readable instructions) into **machine code** (binary instructions) that a computer's CPU can execute.

- In the case of MIPS assembly, you write your program using MIPS assembly instructions (such as add, lw, sw, etc.). The assembler translates these instructions into the corresponding binary format.
- Essentially, the assembler bridges the gap between human-readable code and the low-level machine instructions that the CPU understands.



[Fig.1 Block diagram of executing assembly language]

2. **Linker**:

- A linker is responsible for combining multiple compiled modules (object files) into a single executable program.
- When you write a program, it may consist of several source files. Each source file is compiled separately into an object file (containing machine code).
- The linker takes these object files and resolves references between them. It ensures that functions and variables defined in one module can be correctly accessed from other modules.
- Additionally, the linker can link external libraries (such as standard libraries) to your program.
- In summary, the linker ties everything together to create a complete executable program.

3. **Loader**:

- The loader is the final step before executing a program. It loads the compiled executable (usually in binary format) into memory (RAM).
- When you run a program, the loader reads the executable file from disk and allocates memory for it in RAM.
- It sets up the program's initial state (e.g., initializing global variables) and prepares the environment for execution.
- Once loaded, the CPU can start executing the program's instructions from memory.
- In the context of MIPS, the loader ensures that the program's binary code is correctly placed in memory and ready for execution.

Remember that these three components work together to transform your high-level source code into an executable program that the computer can run.

II. Introduction of assembler

1. **Purpose of MIPS Assembly Language**:
   - **MIPS** (Microprocessor without Interlocked Pipeline Stages) is a popular RISC (Reduced Instruction Set Computer) architecture.
   - Assembly language programming in MIPS involves writing code using mnemonic operation codes (e.g., ADD, SUB, LW, SW, etc.) that correspond to specific machine instructions.

2. **Basic Operators and Instructions**:
   - MIPS assembly language covers fundamental operators and instructions.
   - Examples include arithmetic operations (addition, subtraction), logical operations (AND, OR), memory access (load/store), and control flow (branches, jumps).

3. **Subprogram Calling**:
   - Subprograms (functions or procedures) allow modular code organization.
   - You can call and return from subprograms using instructions like JAL (Jump and Link) and JR (Jump Register).x`

4. **Memory Loading and Storing**:
   - MIPS instructions like LW (Load Word) and SW (Store Word) handle data transfer between registers and memory.
   - Understanding memory organization is crucial.

5. **Program Control Structures**:
   - Conditional branches (BEQ, BNE) and unconditional jumps (J, JR) control program flow.
   - Loops, conditionals, and function calls rely on these structures.

6. **Conversion to Machine Code**:
   - Assemblers convert human-readable assembly code into binary machine code.
   - The resulting object file contains executable instructions for the MIPS processor.

III. Important Elements:

1) **Lexer (Lexical Analyzer):**
   - **Function:** The lexer is the first step in the assembly process. It breaks down the assembly language source code into a stream of meaningful tokens. These tokens are the basic building blocks of the assembly language, such as keywords (e.g., add, sub), identifiers (e.g., variable names, labels), constants (e.g., numbers), and operators (e.g., +, -).

2. Parser (Syntax Analyzer):

- **Function:** The parser takes the stream of tokens generated by the lexer and verifies if they follow the grammatical rules of the assembly language. It checks if the tokens are arranged in a valid sequence to form instructions and data declarations.

3. Symbol Table:

- **Function:** The symbol table is a data structure that stores information about all the symbols encountered during the parsing stage. These symbols include labels, variable names, and any other user-defined identifiers.
- **Information stored:** The symbol table typically keeps track of the following information for each symbol:
  - **Symbol name:** The actual name of the identifier.
  - **Address:** The memory location assigned to the symbol. This could be a data address or an instruction address.
  - **Scope:** The visibility range of the symbol within the program.
- **Importance:** The symbol table plays a crucial role in the translation process. The code generator uses the symbol table to resolve symbolic references within instructions and data declarations.

4. Code Generator:

- **Function:** The code generator takes the parsed assembly program structure and the symbol table information and generates the equivalent machine code instructions. It translates the assembly language instructions into a format that the target processor can understand.

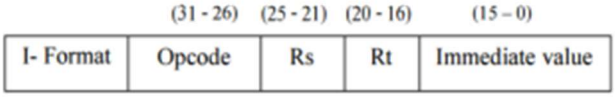
IV. Implementation Approach

1. Initialization:
   - The code begins with necessary imports, including 're' for regular expressions and pandas for data manipulation.
   - MIPS registers, instructions, functions, and initial data are defined.

2. Parsing Input:
   - The input MIPS assembly code is provided as a string named 'data'.
   - The code splits the input into lines and removes any empty lines.

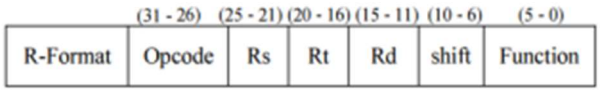
3. Instruction Type Functions: Functions are defined for different types of MIPS instructions:

- 'itype()' for immediate type instructions.



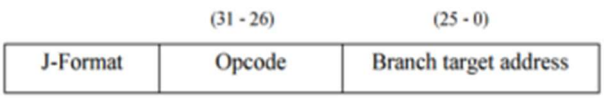
It shows the I-Type instruction format. Here the first 6 bits represent the opcode of the instruction. The 10 bits are of the registers rs,rd, rs is the source register and rd the destination register to store the result of this instruction and the last 16 bits represent the immediate value which would be the part of instruction .

- 'rtype()' for register type instructions.



It shows the R-type instruction format. Here the first 6 bits represent opcode. The next 15 bits represent the three registers rs, rt, rd where rs and rt are the source register and rd is the destination register. The next 5 bits are for the shift amount it determines no. bits to be shifted. The last 6 bits represent the function code which determines which function to be performed on the given registers given previously.

- 'jtype()' for jump type instructions.



It shows the J-Type instruction format. Here the first 6 bits represent the opcode of the instruction, The next 26 bits represent the offset address and to obtain the 32 bits address it is left shifted by 2 bits, and the 4 MSBs

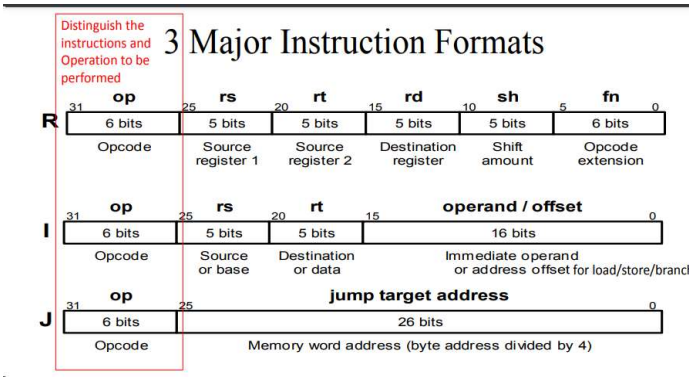
of the Program counter are replaced and then the 32-bit address is formed

4. Exception Handling**:**
   - The code uses raise Exception("error message") to raise exceptions in various functions like itype, rtype, jtype, and branchtype.

5. Specific Error Cases**:**
   - itype**:** The check for immediate values being less than 65536 might not be necessary for all instructions. Some instructions might allow larger immediate values. Consider checking for specific instruction types and their valid immediate operand ranges.
   - branchtype**:** The symbol lookup assumes the symbol exists. You could add a check to see if the symbol is found in symbolfinal before accessing its address.
   - rtype**:** The code currently checks for instructions like add, sub, etc. If the instruction is not found in the inst dictionary, it raises an exception. You could potentially expand the dictionary to include more R-type instructions.

6. Processing Symbol and Data Tables:
   • The code parses the input to identify symbols and data declarations.
   • Symbols are stored in the 'symbolfinal' list, and data is stored in the 'datauser' list.

7. Assembling Instructions:
   • R-type add, sub, and, or, xor, nor
   • I-Type-lui, addi, andi, ori, xori
   • Branch-type-bltz, bne, beq
   • Jump type-j, jr



8. Writing Output:
   • The assembled machine code is written to the output file (demofile2.txt) in hexadecimal format.

9. Symbol Table Editing:
   • The symbol table is updated based on the final addresses of symbols after assembling the instructions.

10. Output:
    • The final symbol table is printed both before and after the assembly process for reference.

V. Features

- **Translation from Assembly to Machine Code**:
  - An assembler converts instructions written in low-level assembly code into relocatable machine code.
  - It generates the information needed for the loader to load the program into memory.

- **Mnemonic Operation Codes**:
  - Assemblers use mnemonic operation codes (e.g., ADD, MOV, etc.) instead of numeric ones.
  - These mnemonics make the code more readable and human-friendly.

- **Symbolic Operand Support**:
  - Assemblers allow specifying symbolic operands (e.g., labels, variables) without requiring their machine addresses.
  - Symbols enhance code readability and maintainability.

- **Precise Control Over Hardware**:
  - Assembly language provides direct access to hardware components like registers.
  - This enables tailored solutions for hardware-specific issues.

VI. Software

Google Colab (short for **Colaboratory**). It's a fantastic cloud-based platform that allows you to write and execute Python code collaboratively in a Jupyter Notebook environment

- **Zero Configuration**: Google Colab requires no local setup; just open your browser and start coding.
- **Access to GPUs and TPUs**: Colab provides free access to powerful computing resources for machine learning tasks.
- **Collaboration and Sharing**: Create and share Colab notebooks with colleagues, allowing collaborative editing.

VII. Results

```
data='''meen: .word 91
chintan: .word 128
typr: nor $s0, $s1,$s1
addi $s0, $s1,0x154
bltz $s1,typr
bne $s1,$s1,typr
pol: j typr
syscall
lui $t1,0x1001
ppo: jal ppo
'''
```

[fig.2 Inputs provided]

It takes two main inputs:

1. **Assembly Language Code:** This is the core input, represented by the text content in the file /content/drive/MyDrive/Colab Notebooks/assembler-2.txt. This file contains lines of assembly language instructions, each potentially involving opcodes, registers, and data values.
2. **Register Definitions:** The script includes a built-in dictionary named register that maps register names (e.g., $s0, $t9) to their corresponding numeric codes (between 0 and 31). This pre-defined dictionary helps the script translate register names appearing in the assembly instructions to the appropriate binary format used in machine code.

```
symbol table
              0       1  2
0  [0x80000000]  [typr]  0
1  [0x80000010]   [pol]  0
2  [0x8000001c]   [ppo]  0
data table
              0       1          2
0  [0x10000000]  [0x5b]     [meen]
1  [0x10000004]  [0x80]  [chintan]
```

[fig3. Output]

demofile2.txt

```
1  80000000:02318027
2  80000004:22300154
3  80000008:0620FFFC
4  8000000C:1631FFFB
5  80000010:00000000
6  80000014:0000000C
7  80000018:3C091001
8  8000001C:0C000007
9
```

[fig.4 machine code generated]

Symbol Table and Data Table:

- The script first processes lines containing labels (symbols) and .word directives.
- It creates a symbol table (symbolfinal) that maps symbol names to their memory addresses (in hexadecimal format).
- It creates a data table (datauser) that stores the memory addresses and corresponding data values for .word directives.
- The script then prints these tables using pandas, which likely means the code includes import pandas as pd.

Machine Code Generation:

- The script iterates through the remaining lines (instructions) in the assembly code.
- For each instruction, it attempts to identify the instruction type (e.g., I-type, R-type, J-type) based on the first element (opcode).
- Based on the instruction type, it calls a specific function (e.g., itype, rtype, jtype, branchtype) to decode the instruction and generate the corresponding machine code (in hexadecimal format).
- The script handles different instruction types, including:
  - I-type instructions (e.g., addi, andi, ori, xori)
  - R-type instructions (e.g., add, sub, nor, and, or, xor)
  - J-type instructions (e.g., j, jal)

- Branch instructions (e.g., bne, beq, bltz) with special handling for branch offsets
- lui instruction with modification (adding $zero register)
- Special cases like jr $ra and syscall

Machine Code Output:

- The script generates machine code for each instruction and prints it along with the corresponding memory address (considering the data section).
- It also writes the machine code and memory address pair to a file named "demofile2.txt".

VIII. Conclusion

The script successfully translates assembly-like instructions into machine code. It handles different instruction formats and supports symbols and data directives. The symbol table is updated to reflect the actual addresses of symbols. Error handling is implemented to catch potential issues during parsing.

IX. Future scope of this project

1. **Resource Optimization**:
   - Assemblers offer efficient resource utilization due to low-level control.
   - Code optimization, resource awareness, and customization are possible.
2. **Customization and Optimization**:
   - Developers can fine-tune code for specific hardware architectures.
   - Assemblers allow precise control over memory, registers, and instructions.
3. Additional features, such as support for more instruction types or optimizations, could be implemented.

X. References

[1] Computer architecture: from microprocessors to supercomputersBehrouz Parhami
[2] Online MIPS assembler | Alan J. Hogan (alanhogan.com)