Report on

Assembler for MIPS Processor

Scientific Programming (2CSOE78)
Special Assignment

Submitted to

Dr. Smita Agrawal

Submitted By:

21BEC091 (Meen Patel)

21BEC128 (Chintan Trivedi)

Table of Contents

1. **Objective**
2. **Reason**
3. **Introduction and Theory**
4. **Important Elements**
5. **Implementation Approach**
6. **Features**
7. **Input**
8. **Output**
9. **Conclusion**
10. **Further Improvements**
11. **Refrences**

Objective:

Assembler converts Assembly Language Instructions to Machine Language Program.

Reason:

One of our main reasons to select this topic is because this helps us to understand how the processes takes place after running the code.

Introduction and Theory:

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.

An essential instrument in the MIPS processor software development cycle is the assembler. Programmers write instructions in assembly language, which is translated into binary machine code that the processor can comprehend and carry out. To accomplish this translation, the assembler carries out a number of operations including lexical analysis, syntax parsing, and code generation.

Important Elements:

Lexer: The lexer reads the assembly code input and extracts tokens, like literals, identifiers, and keywords. Python's regular expression feature is used to provide effective tokenization.

Parser: The parser checks the assembly code's syntax in accordance with the MIPS instruction set architecture by analyzing the token stream produced by the lexer. It builds a parse tree that depicts the program's structure.

Symbol Table: A mapping between symbolic labels and the memory addresses that correspond to them is maintained by the symbol table. It resolves symbolic references and gives labels encountered during assembly memory addressing.

Code Generator: Using the MIPS instruction encoding format as a guide, the code generator iterates through the parse tree and creates the binary machine code for each instruction. It can handle R-type, I-type, and J-type instructions, among other instruction forms.

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.
- ‘rtype()’ for register type instructions.
- ‘jtype()’ for jump type instructions.
- ‘branchtype()’ for branch type instructions.

4. 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.

5. 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

6. Writing Output:

- The assembled machine code is written to the output file (demofile2.txt) in hexadecimal format.

7. Symbol Table Editing:

- The symbol table is updated based on the final addresses of symbols after assembling the instructions.

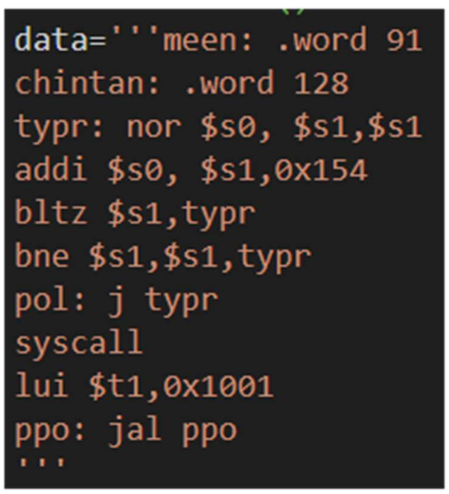
8. Output:

- The final symbol table is printed both before and after the assembly process for reference.

Features:

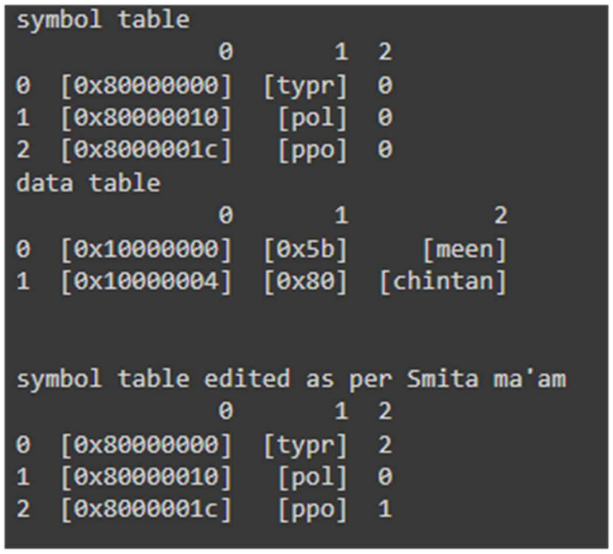
- Support for MIPS instruction set architecture including arithmetic, logical, branch, load-store, and control transfer instructions.
- Handling of symbolic labels and resolving branch targets during code generation.
- Error handling and reporting for syntax errors, semantic errors, and invalid instructions.
- Optimization techniques to improve code efficiency and reduce memory footprint.
- We have also that the number of times the variable updated or the number of times label has been runned.

Input



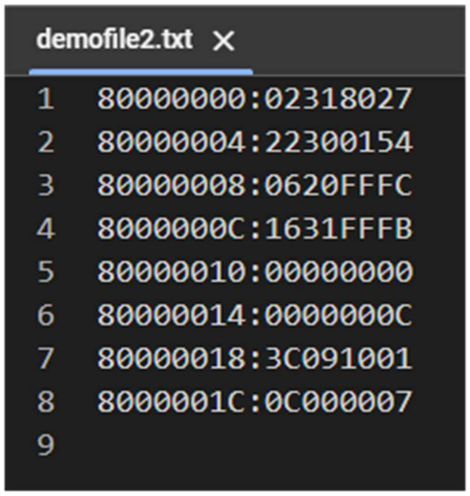
```
data='''meen: .word 91
chintan: .word 128
typr: nor $s0, $s1,$s1
addi $s0, $s1,0x154
bltz $s1,typr
bne $s1,$s1,typr
pol: j typr
syscall
lui $t1,0x1001
ppo: jal ppo
'''
```

Output



```
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]


symbol table edited as per Smita ma'am
              0        1  2
0  [0x80000000]  [typr]  2
1  [0x80000010]   [pol]  0
2  [0x8000001c]   [ppo]  1
```



demofile2.txt

```
80000000:02318027
80000004:22300154
80000008:0620FFFC
8000000C:1631FFFB
80000010:00000000
80000014:0000000C
80000018:3C091001
8000001C:0C000007
```

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.

Further Improvements:

- Error messages could be made more informative, providing details about the encountered errors.
- The code could be optimized for readability and efficiency.
- Additional features, such as support for more instruction types or optimizations, could be implemented.

References:

1. . Computer architecture: from microprocessors to supercomputers- Behrouz Parhami
2. Online MIPS assembler | Alan J. Hogan (alanhogan.com)