NUST CHIP DESIGN CENTRE

RISC-V ASSEMBLER

Submitted by:

Muhammad Atif

Supervised by:

Mr. Umer Farooq

Ms. Hira Sohail

Mr. Muhammad Bilal

Date: 7-Sep-2024

NUST Chip Design Centre (NCDC), Islamabad, Pakistan

RISC-V ASSEMBLER

Abstract:

This project focuses on the development of a RISC-V assembler that converts assembly language instructions into machine code. The assembler processes standard RISC-V instructions and handles **pseudo-instructions** by translating them into valid machine instructions. The assembler is designed to output the corresponding machine code in both binary and hexadecimal formats, enabling its use in various **RISC-V-based applications**.

The assembler was implemented in **C language**, designed with a modular approach to ensure efficient parsing and translation of assembly instructions. This report explains the project's goals, the design and implementation of the assembler and the challenges faced during the development process.

Theoretical Background:

An assembler is a tool that converts human-readable assembly language into machine code, which can be directly executed by a processor. Assembly language is a low-level programming language that represents the instructions a CPU can execute using mnemonic codes. However, since computers cannot understand this textual representation, the assembler's role is to translate these instructions into binary machine code.

The process of converting assembly code into machine code typically involves the following key steps:

Lexical Analysis: The assembler first reads the assembly source code and breaks it down into tokens. Tokens are small units like opcodes (e.g., ADD, SUB), registers (e.g., x1, x2), immediate values (e.g., 10, 0xFF), and labels. During this stage, the assembler ignores whitespace and comments, focusing only on the syntactically significant elements.

Parsing and Instruction Encoding: After tokenizing the input, the assembler analyzes the instructions, which often involve parsing to understand the meaning of each instruction based on its format. Each instruction in assembly language corresponds to a binary instruction in machine code. The assembler matches the assembly instruction to its binary equivalent by looking it up in the instruction set architecture (ISA) definition. For example, a command like ADD x1, x2, x3 gets mapped to a specific binary sequence that instructs the CPU to perform the addition operation.

Handling Labels and Pseudo-Instructions: Labels in the code serve as markers for memory locations, and during the assembly process, the assembler replaces these labels with actual memory addresses. Additionally, many assembly languages support pseudo-instructions, which are convenient representations of more complex sequences of actual

machine instructions. The assembler must expand these pseudo-instructions into valid machine code.

Address Resolution: Some assembly instructions, such as jumps or branches, require the assembler to compute the memory addresses they reference. The assembler calculates the relative or absolute address of the target location and encodes this into the instruction.

Output Generation: Once all the instructions have been processed and translated into machine code, the assembler generates an output file, usually in binary, hexadecimal, or object file format. This file contains the machine code that the CPU can execute. The assembler may also produce auxiliary files, such as symbol tables, which map labels and variables to their corresponding memory addresses.

Challenges Faced and How They Were Resolved:

Developing a RISC-V assembler presented several challenges, ranging from handling complex instruction formats to implementing efficient error handling mechanisms. Here are the key challenges faced during the project and how they were addressed:

1. **Handling Labels and Branches:** Assembly code often contains labels that serve as markers for jump and branch instructions. Resolving the addresses of these labels, especially when they are forward-referenced, was a challenge, as the assembler needed to process the entire code to determine their final addresses.

   Solution: To handle this, a two-pass approach was used. In the first pass, the assembler identified and recorded the addresses of all labels without generating machine code. In the second pass, the assembler used this label information to encode the correct addresses into jump and branch instructions, ensuring that forward references were properly handled.

2. **Error Handling and Syntax Validation:** Ensuring robust error handling was critical, especially for detecting syntax errors, undefined labels, and invalid operands. Without proper error messages, debugging assembly code can become very difficult for users.

   Solution: The assembler incorporated detailed error checking at each stage of the process. During lexical analysis, syntax validation ensured that only valid tokens were processed. During the instruction encoding stage, checks were put in place to ensure that operands were valid for their respective instructions. When errors were detected, meaningful error messages were generated, indicating the line number and nature of the error, which greatly improved debugging.

Testing and Validation:

The testing and validation of the RISC-V assembler were critical steps in ensuring that it performed accurately and efficiently converted assembly code into machine code. The process involved several stages, each focusing on different aspects of the assembler's functionality to verify its correctness, robustness, and reliability.

Unit Testing of Individual Functions: Each function within the assembler, from parsing to encoding, was tested individually through unit tests. For example:

Instruction encoding functions were tested by providing known inputs (such as register values and immediate values) and checking if the generated machine code matched the expected output.

The label resolution mechanism was verified by testing forward and backward references to ensure proper address calculation.

Error handling was tested with various types of incorrect assembly input, ensuring that the assembler produced meaningful error messages without crashing.

Corner Case Testing: Edge cases, such as maximum and minimum values for immediate fields, boundary conditions in address offsets, and unusual label placements, were also tested to ensure that the assembler handled these cases properly. This helped ensure that the assembler didn't break or produce incorrect results under extreme conditions.

Comparison with Existing Assembler Outputs: For further validation, the output of the custom RISC-V assembler was compared with the output of well-established RISC-V assemblers such as the GNU assembler **riscv64-unknown-elf** and **Venus**. By assembling the same assembly program with both assemblers, the outputs were compared to ensure that the machine code produced by the custom assembler was accurate and matched the industry-standard tools.

Use of Python Scipting:

A Python script **(test_assembler_files.py)** is created which automates the testing of the RISC-V assembler by executing it on multiple assembly files and comparing the generated machine code with expected outputs. It first sets up the necessary directories and lists all relevant .s files. For each file, the script runs the assembler to produce machine code and then verifies this output against pre-defined dump files using a comparison script **(check.py)**. This approach ensures consistent and efficient testing, quickly identifying discrepancies and validating the assembler's functionality. Code of **test_assembler_files.py** is included in **Appendix Section**.

Terminal Output:

```
atif@DESKTOP-J2SHTS8:~/Assembler$ make all
gcc -Wall -std=c99 -g -c assembler.c -o assembler.o
gcc -Wall -std=c99 -g -c assembler_main.c -o assembler_main.o
gcc -Wall -std=c99 -g -o assembler assembler.o assembler_main.o
atif@DESKTOP-J2SHTS8:~/Assembler$ python3 test_assembler_files.py
========================================
         Assembly File Tester
========================================
Found 6 assembly files in 'TestingApplication':
test_stores_only.s, test_im_only.s, test_reg_only.s, test_jumps_only.s, test_ui_only.s, test_branches_only.s
========================================

================================================
 Starting Test for: 'test_stores_only.s'
================================================
Running assembler for: test_stores_only.s...
Successfully generated output file: output_machine_code/output_test_stores_only.txt

================================================
 Checking Output: 'output_machine_code/output_test_stores_only.txt'
   vs. 'TestingApplication/hex_stores_only.dump'
================================================
Comparison Result:
Files are identical


================================================
 Starting Test for: 'test_im_only.s'
================================================
Running assembler for: test_im_only.s...
Successfully generated output file: output_machine_code/output_test_im_only.txt

================================================
```

Using makefile for compilation

```
================================================
 Starting Test for: 'test_branches_only.s'
================================================
Running assembler for: test_branches_only.s...
Successfully generated output file: output_machine_code/output_test_branches_only.txt

================================================
 Checking Output: 'output_machine_code/output_test_branches_only.txt'
   vs. 'TestingApplication/hex_branches_only.dump'
================================================
Comparison Result:
Files are identical

========================================
             Testing Complete
========================================
Total Assembly Files Processed: 6
Files with Correct Output: 6
Files with Incorrect Output: 0

Files with Correct Output:
test_stores_only.s
test_im_only.s
test_reg_only.s
test_jumps_only.s
test_ui_only.s
test_branches_only.s

Files with Incorrect Output:
None
atif@DESKTOP-J2SHTS8:~/Assembler$
```

Final Result

Using Python Script to automate the Process of Generating Machine Code:

```
atif@DESKTOP-J2SHTS8:~/Assembler$ python3 assemble_files.py
==================================================
          Assembly File to Machine Code Generator
==================================================
Found 6 assembly files in 'input_files':
test_stores_only.s, test_im_only.s, test_reg_only.s, test_jumps_only.s, test_ui_only.s, test_branches_only.s
==================================================

==============================================
 Starting Generation for: 'test_stores_only.s'
==============================================
Running assembler for: test_stores_only.s...
Successfully generated output file: output_machine_code/output_test_stores_only.txt

==============================================
 Starting Generation for: 'test_im_only.s'
==============================================
Running assembler for: test_im_only.s...
Successfully generated output file: output_machine_code/output_test_im_only.txt

==============================================
 Starting Generation for: 'test_reg_only.s'
==============================================
Running assembler for: test_reg_only.s...
Successfully generated output file: output_machine_code/output_test_reg_only.txt
```

```
==================================================
 Starting Generation for: 'test_jumps_only.s'
==================================================
Running assembler for: test_jumps_only.s...
Successfully generated output file: output_machine_code/output_test_jumps_only.txt

==================================================
 Starting Generation for: 'test_ui_only.s'
==================================================
Running assembler for: test_ui_only.s...
Successfully generated output file: output_machine_code/output_test_ui_only.txt

==================================================
 Starting Generation for: 'test_branches_only.s'
==================================================
Running assembler for: test_branches_only.s...
Successfully generated output file: output_machine_code/output_test_branches_only.txt
==========================================
              Generation Complete
==========================================
Total Assembly Files Processed: 6
atif@DESKTOP-J2SHTS8:~/Assembler$
```

A script **“assemble_files.py”** is designed for user convenience. Users simply need to place their assembly files (*.s) in the “input_files” folder. Upon running the script, the corresponding machine code files will be automatically generated and saved in the “output_machine_code” folder. This streamlined process allows for efficient assembly code processing without any additional steps. **The code for this file is also included in Appendix Section**.

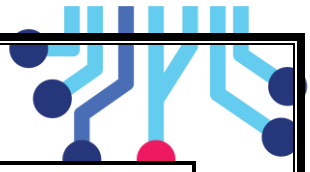
Future Deliverables:

- **Enhanced Instruction Set Support**: Expanding the assembler to handle a broader range of RISC-V instructions, improving its versatility and usability.
- **Integration with Simulators:** Developing integration with RISC-V simulators to allow for seamless testing and debugging of assembly code within a simulated environment.
- **User Interface Improvements:** Creating a graphical user interface (GUI) to make the assembler more accessible to users who prefer visual tools over command-line interfaces.

Appendix:

test_assembler_files.py

```
import os
import subprocess

# Define the paths
testing_application_path = 'TestingApplication'
output_directory = 'output_machine_code'

# Create the output directory if it doesn't exist
os.makedirs(output_directory, exist_ok=True)

# Get all assembly files in the TestingApplication directory
files = os.listdir(testing_application_path)

# Filter for assembly files
asm_files = [file for file in files if file.endswith('.s')]

# Print header
print("========================================")
print("          Assembly File Tester          ")
print("========================================")
print(f"Found {len(asm_files)} assembly files in '{testing_application_path}':")
print(", ".join(asm_files) or "None")
print("========================================")

# Lists to track results
correct_outputs = []
incorrect_outputs = []
```

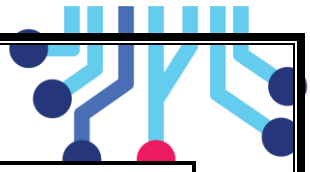


```
# Iterate through each assembly file
for asm_file in asm_files:
    # Construct the output file name
    output_file = os.path.join(output_directory, 'output_' +
asm_file.replace('.s', '.txt'))

    # Construct the corresponding dump file name
    dump_file = asm_file.replace('test_', 'hex_').replace('.s', '.dump')
    dump_file_path = os.path.join(testing_application_path, dump_file)

    # Print new test start message
    print("\n" + "=" * 50)
    print(f" Starting Test for: '{asm_file}'")
    print("=" * 50)

    # Construct the assembler command
    assembler_command = f"./assembler {os.path.join(testing_application_path,
asm_file)} {output_file} -h"

    # Execute the assembler command
    print(f"Running assembler for: {asm_file}...")
    try:
        subprocess.run(assembler_command, shell=True, check=True)
        print(f"Successfully generated output file: {output_file}")
    except subprocess.CalledProcessError as e:
        print(f"Error running assembler for {asm_file}: {e}")
        continue

    if os.path.exists(dump_file_path):
        # Construct the check.py command
        check_command = f"python3 check.py {output_file} {dump_file_path}"

        # Execute the check.py command
        print("\n" + "=" * 50)
        print(f" Checking Output: '{output_file}'")
        print(f"    vs. '{dump_file_path}'")
        print("=" * 50)

        result = subprocess.run(check_command, shell=True, capture_output=True,
text=True)

        # Print the result
        print("Comparison Result:")
        print(result.stdout)
```

```
        # Check the result for correctness
        if "Files are identical" in result.stdout:  # Modify this based on actual
success indicator
            correct_outputs.append(asm_file)
        else:
            incorrect_outputs.append(asm_file)
    else:
        print(f"\nWarning: Dump file '{dump_file}' not found for '{asm_file}'.")

# Print footer with results
print("=========================================")
print("              Testing Complete           ")
print("=========================================")
print(f"Total Assembly Files Processed: {len(asm_files)}")
print(f"Files with Correct Output: {len(correct_outputs)}")
print(f"Files with Incorrect Output: {len(incorrect_outputs)}")
print("\nFiles with Correct Output:")
print("\n".join(correct_outputs) if correct_outputs else "None")
print("\nFiles with Incorrect Output:")
print("\n".join(incorrect_outputs) if incorrect_outputs else "None")
```

assemble_files.py

```
import os
import subprocess

# Define the paths
input_directory = 'input_files'
output_directory = 'output_machine_code'

# Create the output directory if it doesn't exist
os.makedirs(output_directory, exist_ok=True)

# Get all assembly files in the Input_files directory
files = os.listdir(input_directory)

# Filter for assembly files
asm_files = [file for file in files if file.endswith('.s')]

# Print header
print("=========================================================")
print("          Assembly File to Machine Code Generator      ")
print("=========================================================")
print(f"Found {len(asm_files)} assembly files in '{input_directory}':")
```



```
print(", ".join(asm_files) or "None")
print("=====================================================")

# Iterate through each assembly file
for asm_file in asm_files:
    # Construct the output file name
    output_file = os.path.join(output_directory, 'output_' +
asm_file.replace('.s', '.txt'))

    # Print new test start message
    print("\n" + "=" * 50)
    print(f" Starting Generation for: '{asm_file}'")
    print("=" * 50)

    # Construct the assembler command
    assembler_command = f"./assembler {os.path.join(input_directory, asm_file)}
{output_file} -h"

    # Execute the assembler command
    print(f"Running assembler for: {asm_file}...")
    try:
        subprocess.run(assembler_command, shell=True, check=True)
        print(f"Successfully generated output file: {output_file}")
    except subprocess.CalledProcessError as e:
        print(f"Error running assembler for {asm_file}: {e}")
        continue

# Print footer
print("=========================================")
print("              Generation Complete          ")
print("=========================================")
print(f"Total Assembly Files Processed: {len(asm_files)}")
```

THE END