Report

Member One
Tanishq Arvind Chirame
Roll Num.CS23BTECH11013

Member Two
Bolla Lokesh Reddy
Roll Num.CS23BTECH11011

Date:-August 2024

1 Introduction

This project aims to build the RISC V ISA assembler is used extensively in demand in both, the studies/academics and the industries, which can convert given instructions in set of R-type,I-type,S-type,U-type,J-type and B-type format into the low level machine understandable binary (or Hex) instruction. RISC-V's open-source nature has led to its widespread adoption in both academic institutions and industry. Project also includes the excellent error handling if user inadvertently makes an error in syntax of the instructions used in the RISC V ISA(Instruction Set Architecture), If syntax error creeps in, code returns the failure value and exits the input code gracefully.The particularity of this project is the use of the C++ programming language and standard libraries containing interfaces for file manipulation that are fundamental in parsing and encoding processes.

i.e. if input.s contains a line *beq x1, x2, 240*,
our code will change it to 0x0e208863
which is 00001110001000001000100001100011 in binary

2 Design Ideas

While writing the large code for this project, we have divided the whole code in different types of files, those are as follows:

2.1 .hh file

All the prototyping is done in the .hh file and it is included in all the other linked files using **include "Assembler.hh"**, and actual function implementations are wrote in following files...

2.2 R Type Instruction:

Code in this file has the framework with many utility functions to change the R-Type instruction in assembly level language to 32 bit binary instruction which is machine understandable... The check which tells us if the instruction is R-Type or some other type is performed in the **main.cpp** file...

2.3 I,S,B,U,J Type instructions:

For these type of instructions also, it is pretty same as the R-Type instruction code...

2.4 conversions.cpp

This file contains many utility functions such as,

- Is-semicolon
- convert
- to-binary-12
- to-hexadecimal
- to-binary-reg
- to-binary-20

What do they do: **Is-semicolon:** It gives information about the presence of the comments starting with the ';'

convert: This function converts pseudo registers to the actual ones, standard ones like x0,x1,...,x31.

to-binary-12: This function converts the given string to it to the equivalent binary base number consisting of only the 0s and 1s... and if the binary number has less than 12 digits, then it prepends the required number of zeroes to make the length of it as 12...

to-hexadecimal: It converts given input to number in hexadecimal base system...

to-binary-reg: This function returns the 5 bit string which is just the binary representation of the number which standard registers x0,x1,...,x31 have i.e. the number present after 'x'.

This is completely hard coded using the Binary-Storage map.

to-binary-20: it is just like the to-binary-12 function, and this function extends the return string value till the 20 bits...

2.5 Use of inbuilt concatenation:

A concatenation function which concatenates the two strings with the use of overloaded '+' operator to perform extra functionality... This type of function is used excessively to arrange the proper bits in their accurate positions in the 32 bit instructions...

2.6 Use of makefile:

If only one file, R.cpp was changed, then makefile will only compile that file and not touch the other files, in brief, makefile only compiles the files in which changes were made...

This helps to lower the time for compilation of the big codes... as redundant compiling is avoided...

3 Limitations of our Assembler

3.1 Lack of custom extensions

Our Assembler does not supports the custom instructions made by the user, which means that unfortunately, it has 0 customizability...

3.2 Lack of code optimization

Code Not Optimized: The assembler doesn't tweak the machine code it creates. This can lead to less productive code, without using tricks like shuffling instructions, getting rid of useless code, or making branches work better.

3.3 Not Scalable extensively

Scalability Issues: The assembler might not be optimized for large-scale assembly programs, potentially leading to slow performance when assembling large codebases or complex programs. Mainly due to absence of the lack of the code optimization...

3.4 Dependecy on CPP compilers:

C++ Compiler Dependency: The assembler's performance and compatibility are tied to the C++ compiler and libraries used in its construction. This connection can cause issues when you attempt to compile or execute the assembler on various platforms or with different compiler versions.

3.5 Endianess Assumptions:

If the assembler is hard-coded to assume a specific endianess (e.g., little-endian), it might not function correctly on systems or for applications requiring the opposite endianess (big-endian).

3.6 Comments Handling

Code assumes that there are no comments in the input.s file, if there exists one of more comments, code will not be able to handle it and may give unexpected results...

3.7 Purely Command-Line Interface:

Since the user does not have any graphical interface to interact with the assembler, it is going to be less friendly for users, especially the unadvanced ones or uncomfortable using a command-line tool.

3.8 Empty line Handling

Code assumes that there are no empty lines in the input.s file... and if there creeps in any, code will behave in unexpected way.

4 Verification Approach

4.1 Cross checking:

We gathered many RISC-V codes from the internet from the various websites such as some public github repositories and from the websites of different universities:

RISC V Codes I

RISC-V Codes II

Then we made Ripes simulator convert these codes into machine level understandable codes and then cross checked with output of our code... Although our assembler did not perform code optimizations on bigger chunks of the code, we verified for smaller chunks of code taking some lines from the code...

4.2 Use of input generator:

We also wrote as C++ program which works like a generator of instructions using nested for loops, and cross checked the results from the ripes for the same using another CPP program...

4.3 Individual Component testing:

Also, the individual component testing was done by creating the buffer CPP files to check just that single component's working or not...

4.4 Regression Testing:

We were compiling and running code whenever we modified it to check whether the code is working properly or not as it was working before the modifications...

4.5 Edge Cases Handling:

The following tests were performed with the assembler: extreme values of immediate fields, unusual sequences of instructions, and complicated arrangements of labels. All these helped to make sure that the assembler would not fail on some atypical but possible inputs.

4.6 Stress Testing:

Stress Testing: The assembler was tested with large programs or those with complicated structures to see whether it was strong enough and quick.

5 Key Learnings from the overall project...

5.1 Learned to use maps

Got to know about the cool in built data structure in C++, which is maps, we used them extensively to store the hard-coded data such as Binary-Storage values for 0-31...

5.2 Learnt about the File IO operations in deep

We delved deep into the world of File processing operation, subsequently reading about file handling through the C++ classes, although didn't use that, sticked to the simple IO streams like **ifstream**, **ofstream**, also learned about string iterators, *string::npos* function, as well as the .tellg() and .seekg() functions which come inbuilt with the fstream library of the CPP.

5.3 Learnt pragmatics of the coding software like RISC-V assembler:

One of the principles state that we must make common case faster... So we hard-coded many things such as binary-storage maps and some other things to remove the overhead which comes along with the function calls for the repetitve things and computationally small things...

5.4 Memory Management

I learned to manage memory efficiently by avoiding redundant allocations and keeping track of memory addresses during assembly code translation.

5.5 Debugging Skills and Error Handling

The assembler project required intensive debugging, which helped me hone my skills in this area. I learned how to systematically trace errors, including faulty memory addresses, incorrect instruction encoding.