Indian Institute of Technology Hyderabad

RISC V ASSEMBLER

Group Members

V. Vishal
cs23btech11064

M. Suhas
cs23btech11038

Introduction

This project is basically a RISC V Assembler for the 64-bit processors. It converts the RISC V assembly code into hexadecimal format. Below is the description of how you can use our project to generate the machine code for the assembly code.

Implementation

The project is divided into several files based on their functionality. This allows them to be included separately if required, reducing the need to compile the entire program when only parts of the code have been modified. The key files are `main.cpp`, `Rconvert.cpp`, `Rconvert.hh`, and `commonalgo.cpp`, among others. Below is a discussion of each file.

Main

In this section, we find the line numbers where the labels are present and store them in an array of structures. If the same label appears more than once, it is ignored, and the instruction is treated as a regular instruction without a label.

The program reads from an `input.s` file line by line using `getline()`, and it writes the output into an `output.hex` file. The input file is read twice: once to locate labels and again to compute the respective hexadecimal codes.

CommonAlgo

This file contains functions that are used extensively across the project:

1. `int search_label(std::string label, struct store_label label_line[])`
   This function searches for a label in an array of structures and returns its line number.
2. `unsigned int binary_to_decimal(std::string s)`
   Converts a binary string into a decimal number.
3. `std::string decimal_to_hex(unsigned int decimal)`
   Converts a decimal number into hexadecimal.
4. `std::string deci_to_bi(int x, int no_of_bits)`
   Converts a decimal number into a binary string with a specified number of bits.
5. `std::string register_to_bi(std::string s, int line_counter)`
   Converts a register name into binary format. For invalid input, it outputs an ”invalid register” message along with the line number. For example, `ra, sp, gp, t0, s11`.

Rconvert

This file contains the function `std::string Rconvert(std::string s)`, which processes R-type instructions. The function assigns the fields (`rd`, `rs1`, `rs2`, etc.), then constructs a 32-bit binary string and converts it into hexadecimal:

```
binary = funct7 + register_to_bi(rs2,line_counter) +
   register_to_bi(rs1,line_counter) + funct3 +
   register_to_bi(rd,line_counter) + opcode;
return decimal_to_hex(binary_to_decimal(binary));
```

It returns the hexadecimal equivalent of the R-type instruction.

Iconvert

This file contains the function `std::string Iconvert(std::string s)`, which performs the same tasks as `Rconvert` but for I-type instructions. Instructions are grouped by operations such as `addi`, `ld`, and `jalr`. The binary string is constructed and returned as hexadecimal in the same manner as in `Rconvert`.

Sconvert

This file contains the function `std::string Sconvert(std::string s)`, which processes S-type instructions. Immediate values are handled using `.substr()` to slice strings for the needed bits. The final binary string is constructed and returned as hexadecimal:

```
binary = imm_bin.substr(0,7) + register_to_bi(rs2,line_counter) +
   register_to_bi(rs1,line_counter) + funct3 + imm_bin.substr(7,5)
   + opcode;
return decimal_to_hex(binary_to_decimal(binary));
```

Bconvert

This file processes B-type instructions. The function `search_label` checks the label's line number. The immediate value is split into parts and the final binary string is returned as hexadecimal.

```
 binary = imm_1 + imm_2 + register_to_bi(rs2,line_counter) +
   register_to_bi(rs1,line_counter) + funct3 + imm_3 + imm_4 +
   opcode;
return decimal_to_hex(binary_to_decimal((binary)));
```

Jconvert

The `Jconvert` function processes J-type instructions, where jumps can occur without labels. Immediate values are split and concatenated, and the binary string is returned as hexadecimal.

Uconvert

The `Uconvert` function processes U-type instructions. Immediate values are simpler to handle since they are not divided into parts, and the binary string is returned as hexadecimal.

Sort_find

This function determines the type of instruction and calls the appropriate conversion function to return the corresponding hexadecimal code.

Struct.hh

We have used an array of structures to store labels with their respective line numbers.

Makefile

The Makefile compiles files based on whether their dependencies have changed. For example, dependencies for `main.cpp` and `sort_find.cpp` are their respective header files. This ensures that only modified files are recompiled, saving time. The following commands are supported:

- `make all` – Compiles all files and generates the final executable `riscv_asm`.
- `make k.o` – Compiles the respective file to create an object file.
- `make clear` – Removes all object files and the final executable.

Limitations

1. Some instruction types handle brackets as spaces, while others do not.
2. If labels are repeated, only the first occurrence is used for branching.
3. If more arguments are provided than required, only the necessary ones are used, and the extra ones are ignored.

Conclusion

The project supports all types of RISC V instructions (R, I, S, B, U, J) and generates the respective hexadecimal output, despite some limitations mentioned above. We successfully ran a few test cases. and we learnt that it is a really difficult process to handle all the possible errors that can be given as a code while trying to find themselves ourselves.