Indian Institute of Technology Hyderabad

Computer Architecture - CS2323

Lab-4 (RISC-V Simulator)

Group Members

V. Vishal
cs23btech11064

M. Suhas
cs23btech11038

Introduction

This project is a RISC-V Simulator for 64-bit processors. It simulates system operations, similar to Ripes. The user can perform the following actions with the simulator: view the values of registers, see the stored memory, track the lines being executed, and set breakpoints when needed. Additionally, it shows how functions are stacked and popped as the code is executed.

Implementation

This project is written in C++, and we have divided it into several parts. The first part involves the assembler code, which is spread across different files, as outlined in our previous assembler lab assignment. Most of the files used in that assignment are reused here, making this project a continuation of the previous one.

The main modifications were made to `commalgo.cpp`, and `main.cpp` (previously used for parsing) was replaced by `risc_sim.cpp`, which handles the entire simulator process—from parsing to generating the output. Below is a detailed overview of all the files used in this project.

risc_sim.cpp

This file is divided into various functions, each serving a specific purpose, as shown in the code snippets below. In addition to these functions, there is a main function.

The main function is responsible for initializing all registers, the structure that stores the labels, and a stack that tracks which function is invoked and on which line it is being executed. It also handles the initialization of breakpoints.

An infinite loop runs within the main function, which processes instructions that can be in one of the following formats:

```
load <filename>
run
regs
exit
mem <addr> <count>
step
show-stack
break <line>
del break <line>
```

For each instruction, the main function either handles the task directly or calls the appropriate function to perform the operation. Some of the key functions defined in this file are:

```
void check_inst(vector<pair<string, string>> instruction_line,
   struct regs *x, int *pc , int line_number , stack<stack_label>
   *show_func , struct store_label *label_line , int data_length);
```

```
void run(vector<pair<string, string>> instruction_lines, struct
   regs *x, int *pc, int *breakpoint , stack<stack_label>
   *show_func , struct store_label *label_line , int data_length);
```

```
void step(vector<pair<string, string>> instruction_lines, struct
   regs *x, int *pc, int *breakpoint , stack<stack_label>
   *show_func , struct store_label *label_line , int data_length);
```

```
void store_in_mem(string int_value, int *mem_adrr, int bytes);
```

```
void storing_label(string filename, struct store_label *label_line);
```

```
string remove_label(string assembly_instruction, int index, struct
   store_label *label_line);
```

```
string remove_leading_blanks(string assembly_instruction);
```

```
vector<pair<string, string>> parse(string filename, struct
   store_label *label_line, int *data_length);
```

```
void print_stack(stack<stack_label> show_func);
```

```
void print_mem(int address, int length);
```

```
void print_regs(struct regs *x);
```

```
void initialize_mem(void);
```

```
void initialize_stack(stack<stack_label> *show_func);
```

```
void initialize_regs(struct regs *x);
```

Memory, Registers, and Stack

We have defined structures for memory, registers, and function stacking. For memory, a globally defined array stores the data from addresses `0x10000` to `0x50000`. Registers and the stack are declared locally in the main function. To update them, we pass them by reference to the respective functions. However, memory can be updated by any function in the file.

The defined functions handle tasks such as initialization, printing, and storing data in their respective structures. The `parse` function reads the input file, converts each line to its corresponding binary code, and stores it in a vector of instruction-binary code pairs. The file is parsed for both memory (data section) and labels, which are stored accordingly.

Instruction Execution

The `check_inst` function is responsible for processing an instruction by taking the necessary parameters and executing the corresponding operation, while also updating the program counter.

The `run` and `step` functions call `check_inst` as appropriate, depending on the position of breakpoints.

Additionally, there are functions that handle removing labels from instructions for binary conversion and removing any leading spaces or tabs.

commalgo.cpp

This file contains most of the functions from the earlier assembler assignment, with a few additional functions added to facilitate easy conversion between different data types. Various data types are utilized for decimal, hexadecimal, and binary conversions as needed, without overloading the functions.

Some of the key functions defined in this file include:

```
int search_label(std::string label, struct store_label
    label_line[]);
```

```
std::string remove_starting_zeros(std::string s);
```

```
unsigned int binary_to_decimal(std::string s);
```

```
std::string decimal_to_hex(unsigned int decimal);
```

```
std::string decimal_to_hex_no(unsigned long long decimal, int
    no_digits);
```

```
std::string signed_decimal_to_hex_no(long long decimal, int
    no_digits);
```

```
std::string deci_to_bi(int x, int no_of_bits);
```

```
std::string register_to_bi(std::string s, int line_counter);
```

Limitations

1. In `.dword`, `.word`, and `.data` (in the data section), only numerical values are accepted. For example, `0x12345` is not accepted.
2. Blank lines are not handled properly.
3. If there is no line of code after a label, the code won't work.
4. Spaces at the end of a line are printed as they are.
5. There should not be a comma or spaces or tab characters at the end of `.dword` or `.word` commands.

Conclusion

In conclusion, the RISC-V Simulator developed in this project successfully demonstrates the fundamental principles of computer architecture and assembly language programming. By allowing users to visualize the flow of execution, memory management, and register manipulation, the simulator serves as an effective educational tool for understanding how RISC-V architecture operates.

While the simulator showcases various functionalities, it also has limitations that highlight areas for potential improvement. Future enhancements could involve more robust handling of edge cases, support for additional instruction types, and improved user interface features. Overall, this project not only reinforces the knowledge gained in computer architecture but also provides a hands-on approach to exploring the intricacies of assembly language and hardware interaction.