Lab 4 - Project Report

AI24BTECH11031 - Shivram S
AI23BTECH11022 - Dhadheechi Ravva

Contents

1 Introduction

As part of the CS2323 Computer Architecure course, we had to develop a simulator for the RISC-V instruction set architecture. This report outlines the design of the simulator and the major design decisions taken during its development.

2 Usage

The project includes a `Makefile` which can be used to build the project and test it. To build, run:

```
$ make
```

This produces an executable called `riscv_asm` in the project directory.

The simulator can be run using the following command:

```
$ ./riscv_asm
```

3 Modifications to the Assembler

Some small changes had to be made to the assembler which we created in Lab-3.

Our assembler did not have the ability to parse the `.data` section. This functionality was added during the development of the simulator. The `text_section` field was added to the parser struct to indicate whether the parser was currently in the text section or the data section. An additional function was added to parse the data section and write the result to an array.

The emitter was modified to write machine code to a buffer instead of to a file. Additionally, the label finding and instruction encoding parts were split into separate functions.

4 Design of The Simulator

The simulator holds all state in a struct:

```
typedef struct Simulator {
    uint64_t pc, regs[32];
    uint8_t mem[MEM_SIZE];
    char *src;
    ParseNode *nodes;
    LabelVec *labels;
```

```
    BreakPointVec *breaks;
    StackVec *stack;
} Simulator;
```

The program counter, registers and memory are part of this struct. `src` is the source assembly program. `LabelVec`, `BreakPointVec` and `StackVec` are dynamic arrays for storing a list of labels, breakpoints, and stack entries respectively. They are defined as follows:

```
typedef struct LabelVec {
    int len, cap;
    LabelEntry *data;
} LabelVec;

typedef struct BreakPointVec {
    size_t len, cap;
    int *data;
} BreakPointVec;

typedef struct StackVec {
    size_t len, cap;
    StackEntry *data;
} StackVec;
```

`LabelEntry` and `StackEntry` are defined as:

```
typedef struct LabelEntry {
    char *lbl_name;
    int offset;
} LabelEntry;

typedef struct StackEntry {
    char *label;
    int line;
} StackEntry;
```

4.1 Loading Files

To use the simulator, we must first load a source file and assemble it. This is done using the `sim_init` and `sim_run` functions.

```
// Initializes the simuator; zeroes memory and registers
void sim_init(Simulator *s);
```

```
// Assemblers a file and loads it into the simulator's memory
int sim_load(Simulator *s, char *file);
```

4.2 Executing Instructioms

The bulk of the simulator is the `sim_run_one` function, which fetches one instruction and executes it. It is implemented in the form of a large `switch-case` statement, to conditionally execute code based on the opcode, funct3 and funct7 values.

Inside each case statement, the simulator fetches the read registers (depending on the instruction type). It might go into further switch-case statements based on the funct3 value, which correspond to different instructions belonging to the same instruction type. The simulator then performs the operation corresponding to the particular instruction, writes back to rd (the destination register), writes to memory (for the store instructions), or just updates the PC (for jal and B format instructions).

The `jal` and `jalr` instructions have extra code to push and pop values from the stack.

`sim_run_one` is not used directly, but is instead wrapped by the `sim_step` function. This function adds a zero-instruction check, calls `sim_run_one`, and also handles updates of the call stack.

`sim_step` is further wrapped by the `sim_run` function, which keeps executing instruction until we reach a breakpoint or the end of the program.

4.3 Breakpoints

The user can set breakpoints on certain lines in the program. When the simulator reaches that line, it stops execution and alerts the user that a breakpoint has been reached. Adding and removing breakpoints are done through the `sim_add_breakpoint` and `sim_remove_breakpoint` functions respectively.

```
// Adds a breakpoint
void sim_add_breakpoint(Simulator *s, int line);

// Removes a breakpoint
void sim_remove_breakpoint(Simulator *s, int line);
```

4.4 The Call Stack

The simulator keeps tracks of function calls using the call stack. When a function is called, it is pushed onto the stack along with the line number of the last executed instruction. On returning from the function, the entry is popped

from the stack. This is done using the `sim_stack_push` and `sim_stack_pop` functions.

```
// Pushes a label and line onto the stack
void sim_stack_push(Simulator *s, char *label, int line);

// Pops the topmost entry from the stack
void sim_stack_pop(Simulator *s);
```

Printing the stack is done through the `sim_show_stack` function.

```
// Prints the contents of the stack
void sim_show_stack(Simulator *s);
```

4.5 Command Line Interface

The user gives instructions to the simulator through a command-line interface. The code for this is in `main.c`. The available commands are:

- `load <filename>`: Loads the file containing RISC-V assembly code
- `run`: Executes a given RISC-V code till the end
- `regs`: Prints the values of all registers
- `mem <addr> <count>`: Prints count memory locations starting from address addr
- `step`: Runs one instruction
- `show-stack`: Prints the call stack
- `break <line>`: Sets a breakpoint at the specified line
- `del break <line>`: Deletes the breakpoint at the specified line.
- `exit`: Exits the simulator

5 Testing

Some example testcases have been provided in the `test/` directory. These can be run by loading the `input.s` file, executing the commands in `expected.txt`, and comparing the output.

6 Future Scope

The simulator could be further enhanced by:

- Adding support for pipelining, hazard detection, multiple issue, etc.
- Implementing a graphical user interface for more intuitive operation of the simulator.
- Allowing modification of memory or registers using commands.
- Adding support for stepping back/reversing execution.

7 Project File Structure

```
.
+-- Makefile
+-- report
|   \-- main.tex    // Source file for the report
+-- src
|   +-- asm         // Source code for the assembler
|   |   +-- emitter.c
|   |   +-- emitter.h
|   |   +-- lexer.c
|   |   +-- lexer.h
|   |   +-- parser.c
|   |   +-- parser.h
|   |   +-- tables.c
|   |   \-- tables.h
|   +-- main.c
|   +-- simulator.c
|   \-- simulator.h
\-- test             // Testcases
```