Tomasulo Algorithm Simulator

A project by

- Youssef Agiza: 900192237
- Jacqueline Azar: 900184062
- John El Gallab: 900193761

Overview

A simulator for Tomasulo's algorithm for a simplified out-of-order 16-bit RISC-V processor. The processor has 7 16-bit registers (R0 to R7), where R0 is hardwired to 0. The data memory is word addressable and uses a 16-bit address. its size is limited to 256 for easier testing and printing; however this can be customized by chaning `MEMORY_SIZE` in `mem.hpp` to any value up to 2^{16}.

output

The program displays the following output

- The register file
- The data memory content
- Metrics about the *executed* instructions. The cycles at which each instructions was issued, started execution, eneded execution, performed write back.
- Instruction per cycle (IPC).
- Total cycles done by the program.
- Branch Misprediction percentage.
  - **Note:** the simulator uses an always not taken predictor for the branch instructions.

Usage

- The program can be built and compiled using `make`

```
make tomasulo
```

- To run the program

```
./tomasulo <instruction file path>  <optional:output file path>
```

- **Input:** the instructions must follow the format mentioned in the supported operations section.
    - The register names can be any signle letter followed by the register number as long as the number ranges from 0 to 7.(i.e. x0, r0, s0 are all the same)
- **Output**: the results will be displayed on the console in all cases, but the user can provide output file to save the results if needed.

Customizations(bonus feature)

Once the program starts, the user is asked wether or not they want to customize some hardware parameters (e.g. the number of available load units). The user can also provide a txt file to initialize the data memory and choose the address from which the Program Counter(PC) starts.

Default Hardware parameters

By default the simulator has the following units available:

unit name	number of stations	number of cycles needed
Lw unit	2	2 (compute address) + 2 (read from memory)
SW unit	2	2 (compute address) + 2 (writing to memory)
BEQ unit	1	1 (compute target and compare operands)
JAL/JALR unit	1	1 (compute target and return address)
ADD/ADDI unit	3	2
NEG unit	1	3
ABS unit	1	2
DIV unit	1	10

Supported operations

1. **Load/store**
    - **Load**: Loads a word from memory into rd. Memory address is formed by adding imm with contents of rs1, where imm is a 6-bit signed immediate value (ranging from -32 to 31).

```
LOAD rd, rs1, imm
```

- **Store**: Stores a word from rs2 into memory. Memory address is computed as in the case of the load word instruction

```
STORE rs2, rs1, imm
```

2. **Conditional branch**
   - **Branch if equal**: branches to the address PC+1+imm if rs1=rs2

```
BEQ rs1, rs2, imm
```

3. **Control Transfer Instruction**
   - **Jump and link**: Stores the value of PC+1 in rd and branches (unconditionally) to the address PC+1+imm.

```
JAL rd, imm
```

   - **Jump and link register**: Stores the value of PC+1 in rd and branches (unconditionally) to the address in rs1. This instruction can also be used to return from a function.

```
JALR rd, rs1
```

4. **Arithmetic Operations**
   - **Add**: Adds the value of rs1 and rs2 storing the result in rd

```
ADD rd, rs1, rs2
```

   - **Add immediate**: Adds the value of rs1 to imm storing the result in rd

```
ADDI rd, rs1, imm
```

   - **Negate**: Computes the 2’s complement of rs1 storing the result in rd

```
NEG rd, rs1
```

   - **Absolute**: Computes the absolute value of rs1 storing the result in rd

```
ABS rd, rs1
```

   - **Divide**: Computes the quotient of dividing the value of rs1 by the value of rs2 storing the result in rd

```
DIV rd, rs1, rs2
```

Test Cases

By default the repo has 4 small test cases in the `tests` folder and their output is saved in `results`. Inside each test cases there are comments beside the instructions to explain the expected behavior.

1. **deps.txt**: a list of all the arithmetic operations and load/store instructions in which the destination register of each instruction is a source register for the following one. Thus, it tests how the simulator handles dependencies.
2. **beq.txt:** A program that tests multiple branches some of which are taken and some are not. If the program behaved incorrectly, register R4 and R5 will have -1 at the end.
3. **diffPc.txt:** a trivial program to try starting with different PC. The user should give PC=4 and the registerfile should have positive numbers only at the end. If something went wrong, one of the registers will have -1.
4. **jal_jalr.txt**: a program that tests different combinations of jal and jalr instructions.
5. **loop.txt**: a small program that has a loop.
6. **load_store.txt**: a program that tests sequence of loads/stores with and without conflicts.

To re-run the program against those test cases run:

```
make tests
```

Note: this will only run the program against each of the previous files, but the user still needs to answer the prompt messages for each case.

Breif Explanation of Logic and Main Data Structures

The two main data structure is `class rs`, standing for reservation station, and `struct rstable`, standing for reservation station table, declared in `rs.hpp`. `rs` as an generic that has all the fields needed by any instructions (e.g. Vj, Qk, etc.). However, has 4 function pointers which are used to customize the stations behavior according to the isntruction:

```
void (*issue_)(rs *);

bool (*can_exec_)(rs *);

void (*exec_)(rs *);

void (*wb_)(rs *);
```

- `issue_` : is reponsible for called to issue the instruction
- `can_exec_` : returns true if the instruction can start executing
- `exec_` : performs the instruction execution when the needed cycles are reached
- `wb_` : performs write back

`rstable` is a struct that has a vector of reservation stations for each unit supported. On initilization, each vector initializes the stations within it using the corresponding issue, can_exec, exec, wb functions(assigned to the pointers above).

All the instructions goes through the same life cycle where each instruction starts as idle, then issued, then executes, then writes back, then returns back to idle state.

The former states are represented using an enum

```
enum rsstate {IDLE = 0,BUSY,EXECUTING,WRITING,FINISHED};
```

- **IDLE:** didn't issue
- **BUSY**: issued but didn't start execution
- **EXECUTING**: started execution but didn't finish
- **WRITING**: finished execution and still writing
- **FINISHED**: Finished writing back and will be reset to IDLE

The logic is implemented as a finite state machine

idle $\rightarrow$ busy $\rightarrow$ executing $\rightarrow$ writing back $\rightarrow$ Finished $\rightarrow$ idle.

every iteration, we loop on all the stations in the processor and update them using the method `update_state`. `update_state` checks the current state and calls the appropriate function pointer according to the state(check `rs.cpp`).