ECE #563 **Tomasulo with Reorder Buffer Processor**

Zhiping Wang | Unity ID: zwang79 | Student ID: 200265085

04.13.2019

Objective

The goal of this project is to design and implement a C++ cycle-accurate simulator of a dynamically scheduled processor implementing the Tomasulo algorithm with Reorder Buffer.

Result

All testcases are matched to clock-cycle. There is a problem that if there is a misprediction when I need to flush all the buffers, the simulation will have a few-seconds lag. This does not affect the simulation result but does make the simulation of some testcases very time-consuming.

Data Structure

Data memory:

```
unsigned char *data_memory;
```

Data memory is where 32-bit data is stored with two parameters, mem_size and mem_latency, that was defined by the constructor. In this project, the mem_latency was defined by memory execution unit.

Instruction memory:

```
instruction_t instr_memory[PROGRAM_SIZE];
```

Instruction memory is defined as instruction_t type. The load_program function translates and decomposes the assembly code into instruction_t type variables like opcode, src1, etc. and store them into instruction memory for IF stage to fetch specific instruction.

General purpose registers:

```
unsigned gp_int_register[NUM_GP_REGISTERS];
unsigned gp_fp_register[NUM_GP_REGISTERS];
```

General purpose registers are the simulation of 32 register files of microprocessor, which stores the value of operands like R1 and F1.

Instruction register:

```
instruction_t Ins;
```

Used for temporarily store instruction fetch from instruction memory.

Reorder Buffer:

```
rob_t rob;
```

The main purpose of ROB is to fetch and commit instruction in program order. Instructions are written into ROB at ISSUE and its entries are cleared when instructions are committed or there is a misprediction. The values in each entry are constantly updated during program execution. There is one thing need to be

mentioned: the update of ROB after WR needs to be carried out at the end of EXE, because my program is running backwards each clock cycle.

```
// ------------update rob and res-----------------
if (update_load.rob_entry != UNDEFINED) {
    update_rob(update_load.rob_entry, update_load.output); // update rob and res at EXE because of i+1 cycle
    update_res(update_load.rob_entry, update_load.output);
    update_load.rob_entry = UNDEFINED;
    update_load.output    = UNDEFINED;
}
```

Also, the clear and flush of ROB entries needs to be carried out at the end of ISSUE, because the cleared entries can be used one clock cycle after the commit.

```
//-------------clear rob----------------
if (commit_tag != UNDEFINED) {
    instructions_executed++;
    if ((rob.entries[commit_tag].pc == eop_pc - 4) && (!mispredict)) end = true;
    clear_rob(commit_tag); // clear rob
    clear_window(commit_tag);
    commit_tag = UNDEFINED;
}
```

Reservation Stations:

```
res_stations_t reservation_stations;
```

The main purpose of res stations is to select which instruction can go into execution unit. The entries are written into at ISSUE and cleared after write result. Besides that, the values of operands are updated when rob.entries.value is updated, the address is updated when memory instruction goes into EXE. Similar to ROB, res entries are cleared at the end of ISSUE, because the newly cleared entry cannot be used in the same clock cycle.

Execution Units:

```
unit_t exec_units[MAX_UNITS];
unsigned num_units;
unsigned output[MAX_UNITS];
```

Almost the same as floating-point pipeline, except for the memory unit which takes one clock cycle to generate address. I will not go into detail.

Renaming Table:

```
renaming_t renaming_table[2*NUM_GP_REGISTERS];
```

Structure I created to record renaming mechanism introduced by Tomasulo. Updated and cleared with ROB.

Instruction Window:

```
instr_window_t pending_instructions;
```

For monitoring and recording reason, updated and cleared with ROB.

Hazard Handling

For Tomasulo with ROB, we only need to consider:

RAW introduced by data dependency;
Structural hazard for ROB, res stations, execution units, and memory access;
Memory address RAW introduced by LOAD following a STORE accessing the same address.

RAW introduced by data dependency

This kind of hazard is easy to handle with ROB and res stations. The operand dependency is recorded at ISSUE. What we need to do is to check if operands are ready before EXE. Still,there are some corner cases like XOR R0 R0 R0 that need to be excluded.

```
//-------check for RAW----------
  if (reservation_stations.entries[i].tag1 != UNDEFINED && reservation_stations.entries[i].pc != rob.entries[reservation_stations.entries[i].tag1].pc
      && !rob.entries[reservation_stations.entries[i].tag1].ready) continue; // check if operands are ready (RAW)
  if (reservation_stations.entries[i].tag2 != UNDEFINED && reservation_stations.entries[i].pc != rob.entries[reservation_stations.entries[i].tag2].pc
      && !rob.entries[reservation_stations.entries[i].tag2].ready) continue;
```

Structural hazard for ROB, res stations, execution units, and memory access

The basic solution for this hazard is to stall if the required unit is occupied. There is one thing quite annoying during my design, like I have mentioned above, the cleared unit or buffer cannot be used until the next clock cycle. This makes the design rather complicated. For example, if I clear the ROB at COMMIT stage when certain entry is committed. It will be detected and reused at ISSUE stage IN THE SAME CLOCK CYCLE! Because my program is running backwards. So, I have to put the ROB clearing after this clock cycle's ISSUE is finished.

Memory address RAW introduced by LOAD following a STORE accessing the same address

I have to say this is really complicated. I will list some key points that I have concluded during the design rather than go into details

1. Load and Store cannot be executed in parallel due to single memory unit. When there is no dependency, whoever happens first takes the unit and others must stall. In testcase8, a load and a store can access the memory in same clock cycle, the load takes the access even if it is after the store! I guess the reason is the load is in EXE and the store is in COMMIT, so the load would go first naturally.
2. Memory unit is divided into two parts: address generation and memory accessing. The address generation can be executed by multiple instructions clock cycle by clock cycle even if one instruction is accessing the memory.
3. The memory latency cannot be decremented by clock cycle, because there are two possible stages to access memory. What I did is the define a structure for memory busy time control which contains pc and opcode type indication.

4. Bypass is rather easy to implement. A load needs to check if there is a preceding store in the ROB that accessing the same address before accessing memory and retrieve its value.
5. Load instructions are not allowed to execute if preceded by potentially conflicting store instructions. Which means if a store is stalled for operand RAW, the following load has to be stalled.

Summary

I do not have a lot of time for this project. The code is not well structured but the results are fine.