Richard Wu | wuricha8 | 1008078296
Christian Vedtofte | vedtofte | 1012872125

ECE552 Lab 3 Report

1. Tomasulo Simulation Results for Given Benchmarks

Benchmark	**Total # Tomasulo Cycles**
gcc.eio	1,681,442
go.eio	1,695,063
compress.eio	1,851,549

2. Description of Tomasulo Code

2.1 Custom Data Types

We defined several data structures to make programming the Tomasulo functions easier. First, the instruction fetch queue (IFQ) was implemented as a circular buffer with an array of fixed size. Functional units (FU) took the `func_unit_t` type with fields tracking the instruction occupying the FU, number of cycles left to completion, and the reservation station (RS) entry. Both integer (INT) and floating point (FP) FUs were included in one array, separated by index tracking. Similarly, INT and FP RS entries were included in one array. RS entries were of type `res_stat_t`, which tracked if the entry was busy, if the instruction was currently executing and in which FU, and tags of operands. The CDB simply tracks which instruction is occupying it and the tag it needs to broadcast. Further, a linked list is used to track the age of instructions, this list is simply called the instruction list.

2.2 Helper Functions

Helper functions are all defined at the bottom of `tomasulo.c` and are prefixed with `h_`. The are functions supporting: basic push and pop operations for the IFQ; allocation and deallocation of RS entries and FUs; update of the CDB; and addition and removal of nodes to the instruction list.

2.3 Main Tomasulo Functions

2.3.1 static bool is_simulation_done(counter_t sim_insn)

In this function, we check if the IFQ, reservation stations, functional units, and CDB are empty. If so, return true to end the simulation.

2.3.2 void fetch (instruction_trace_t* trace)

In fetch we use get_instr() to get the next instruction, then we initialize all the tom_[stage]_cycle fields to 0, and if an instruction isn't a trap we push it to the instruction fetch queue (IFQ) if there is space in the queue.

2.3.3 void dispatch_To_issue(int current_cycle)
In this function we check for available reservation stations and allocate a reservation station and pop the instruction from the IFQ if there is an unused one. We also check for branches and simply pop them from the IFQ since they don't use reservation stations.

2.3.4 void fetch_To_dispatch(instruction_trace_t* trace, int current_cycle)
In this function we check if the IFQ is full and call fetch() if it isn't. At the end of the function it calls dispatch_to_issue(). This is the function that we call in runTomasulo() for the fetch/dispatch stage and mostly exists to merge the fetch and dispatch stages as described by the lab guide.

2.3.5 void issue_To_execute(int current_cycle)
This is the function where we go through the instructions from oldest to newest, and check if the operands are ready and if there is a free functional unit for the instruction to use. We allocate the corresponding FU to the instruction if these checks are passed, and then the instruction should enter the execute stage in the next cycle.

2.3.6 void execute_To_CDB(int current_cycle)
This function corresponds to the execute stage, and it is here that we decrement the cycles each instruction in execute has left before they can enter WB/CDB. This is done by decrementing the value of the field "cycles_to_completion". After this decrement we check if the field is 0, and if it is, and CDB is free to use we send it to CDB such that it enters CDB in the next cycle.

2.3.7 CDB_To_retire(int current_cycle)
This is the function where we broadcast to CDB, there are not any conditional checks in this function as any instructions reaching this stage should've already passed any necessary checks/conditions. We go through the list of reservation stations and if any of them need the current instruction being broadcast we set the value to -1 to indicate that its dependency has been resolved, we also do the same for the map table entries.

2.3.8 counter_t runTomasulo(instruction_trace_t* trace)
This is the function called by sim-safe. Here, we initialize all data structures and then enter the main loop, which tracks the cycle number each iteration and runs the above functions, until is_simulation_done returns true.

3. Testing for Correctness
To test our code, we added in print statements gated by a custom define added by us, `DEBUG_PRINTF`; this allowed us to easily turn on or off debugging. The print statements showed us the contents of various variables and the flow of code. We ran the gcc.eio benchmark with debugging on but only with a maximum of 20 instructions, we also uncommented the

`print_all_instr` line within `sim-safe.c` to see the cycles at which each instruction entered each stage. Then we would compare the contents of the RS entries, map table, and FUs to verify everything was correct. Further, we varied the number of FUs and RS entries to test how the program would behave under different amounts of resources; again we verified that the contents of all the data structures were correct.

4. Two Toughest Bugs

Our first major bug was that all instructions seemed to stay in the execute stage for 1 cycle too long. Through print debugging, we realized it was due to having made all functional units advance their execution by 1 cycle after checking for completed instructions. We swapped the order around (so advance first then check) to make it so that in the last cycle of execution, functional units will advance their last cycle of execution and then the oldest completed instruction will be moved to the CDB, if available.

Our second major bug was that many times, dependences would not be resolved correctly because tags were not being dealt with properly. At first, we simply used the index of the reservation station array as the tag for each RS entry, e.g. the instruction using the RS entry in the 0th index of the array would have a tag of 0. Again, through print debugging, we realized that, because our Tomasulo simulation frees RS entries and assigns them to new instructions before a CDB broadcast actually occurs, the newly allocated instruction will have incorrect tags in its RS entry fields. To fix this, we gave each RS entry a ‘tag’ field and each time an entry is allocated we assign it an entirely new tag. In this way, every time a tag is used it will be a unique one. This eliminates any ‘tag aliasing.’

5. Statement of Work

Both Richard and Christian worked collaboratively on all aspects and functions of the Tomasulo program. Richard spent more time on setting up all the custom data structures and helper functions for facilitating supporting operations (see section 2.2 above). Christian spent more time verifying the correctness of the program. We both worked together to solve major bugs.