ECE552: Computer Architecture

University of Toronto
Faculty of Applied Science and Engineering
Lab 3

Prepared By	Danja Papajani (1002136217), Kathryn Tremblay (1003286639)

Total number of cycles with Tomasulo

	gcc.eio	go.eio	compress.eio
sim_num_tom_cycles	1,681,443	1,695,064	1,851,550

Tomasulo Algorithm
addToInstrQ & removeFromInstrQ
For our Tomasulo implementation, we created two helper functions and two additional global variables: available_index held the next free index in the instr_queue while oldest_instr_index held the index with the oldest instruction in the instr_queue. addToInstrQ accepts a recently fetched and valid instruction and inserts it into the instr_queue, and then increments available_index and the instr_queue_size. removeFromInstrQ sets the oldest instruction in the instr_queue to NULL, increments the oldest_instr_index to the next oldest instruction, and decrements the instr_queue_size.

fetch_To_dispatch
In fetch_To_dispatch, we first call fetch(trace) to grab the next instruction. Here, we check to see if there is space in our instr_queue to fetch a new instruction. If there is, we increment the fetch_index. If our fetch_index is within a reasonable range (less than or equal to the sim_num_insn), we grab the next instruction and check to see if it IS_TRAP. While the instruction we grab is a trap, we increment the fetch_index until we have a valid instruction and then insert it into the instr_queue. Returning from fetch(trace), we then grab the most recently inserted instruction and set it's dispatch cycle to the current cycle, dispatching the instruction.

dispatch_To_issue
In dispatch_To_issue, we move instructions from the dispatch stage to the issue stage. We first assure that the instr_queue is not empty, and then begin to categorize the instruction. We grab the oldest instruction in the instr_queue using the oldest_instr_index. If the instruction is a conditional or unconditional branch, we simply remove it from the instr_queue as they are not allocated a Reservation Station entry. Otherwise, we check to see what type of Functional Unit the oldest instruction uses. For an instruction that uses an Integer Functional Unit, we iterate through all Integer Reservation Station entries until an empty one is found. From here, we update the issue cycle of the oldest instruction to the current cycle, and fill the free Reservation Station entry with this instruction. Afterwards, we iterate through the instruction's input registers to update the instruction's tags if an entry exists in the map_table, and we also iterate through the instruction's output registers to update their respective index in map_table with the current instruction. Once the

map_table and Reservation Station have been allocated and updated, we remove the oldest instruction from the instr_queue. This same process is repeated for an instruction that uses the Floating Point Function Unit with its respective Floating Point Reservation Station.

issue_To_execute

In issue_To_execute, we move instructions from the issue stage to the execute stage. Our implementation creates two separate queues to store instructions ready to execute for their respective Function Units. We first iterate through the Integer Reservation Station entries and store any instructions with no RAW dependencies that have yet to start execution in the first available index of our ready_to_execute_INT array, held by oldest_int_index. We repeat this process for the Floating Point Reservation Station, storing instructions in the first available index of the ready_to_execute_FP array, held by the oldest_fp_index. Once both queues have been filled, we use the bubble sort algorithm to sort instructions from oldest to youngest in each respective queue. Once each queue has been sorted, we iterate through their respective Functional Unit data structures until a free Functional Unit is found. If a Functional Unit is available, we set that entry to the first (which is our oldest) instruction from ready_to_execute_INT or from ready_to_execute_FP, increment the index of the oldest instruction it points to (held by oldest_index_int and oldest_index_fp), and set the execution cycle of the instruction to the current cycle.

execute_To_CDB

In execute_To_CDB, we move an instruction from the execution stage to the commonDataBus. Our implementation creates a queue which stores all instructions that have finished executing and are ready to broadcast, stored in finish_execute. We first iterate through both the Floating Point Functional Unit and the Integer Functional Units and insert any instruction who's execution has finished. We check this by ensuring the current_cycle is greater than or equal to the instruction's [execution time + FU_INT/FP_LATENCY], as an instruction can be stalled in its respective Functional Unit if it is not possible to access the CDB in the cycle that it finishes execution. Once entering finished instructions in finish_execute, we use the bubble sort algorithm to sort instructions from oldest to youngest. Once sorted, we iterate through each entry in finish_execute. If an instruction is a store, it does not use the CDB so we deallocate it's Integer Reservation Station and Integer Functional Unit accordingly. Otherwise, we only allow the first oldest instruction, that is not a store, to access the CDB, if the commonDataBus is currently NULL (not in use). Once the oldest instruction (that is not a store) has been found, we update it's CDB cycle to the current cycle and also update the commonDataBus with this instruction. Doing so will ensure the instruction is broadcasted on the next cycle. Once the commonDataBus has been set, we iterate through both Reservation Stations and Functional Units until we find the instruction that we are going to broadcast, and deallocate it's respective Functional Unit and Reservation Station entry.

CDB_To_retire

In CDB _To_retire, we retire the instruction that is writing to the commonDataBus. If an instruction is currently broadcasting, we iterate through both Reservation Stations and update the values of the instruction's tags to NULL if a match exists with the commonDataBus, indicating the instruction can execute in the next cycle. We repeat this process for the map_table, and then set the commonDataBus to NULL to allow instructions in execute_to_CDB to access it.

is_simulation_done
In is_simulation_done, we check to see if all Reservation Stations, Functional Units, and the instr_queue is empty, in addition to checking if the fetch_index has surpassed the available instructions in the trace. If any of these are not true, the simulation is not done.

runTomasulo
In runTomasulo, we call all functions stated above with the current cycle of the execution in reverse order due to priority.

Testing Correctness of Code
We primarily used print statements and gdb to test the correctness of our Tomasulo algorithm. We limited the maximum number of instructions executed by the trace to a smaller number so each cycle could be analyzed (~15 instructions). At the end of each cycle, we printed the relevant data structures and ensured it matched what we expected when we performed Tomasulo ourselves. For each cycle we printed the instruction fetch queue, the map table, and the reservation stations and the corresponding tags they were waiting on, and ensured it was consistent with our expectation. GDB also allowed us to step through each stage's function line-by-line and ensure that within a cycle, the stage was performing its expected functionality.

Bug #1
On our first run of the benchmarks, we ran into many segfaults. We used gdb to pinpoint exactly where these were occurring, and we found that these segfaults were due to trying to access data structures (the map table, the reservation stations, and the instruction fetch queue) at the beginning of program execution when they had all been initialized to NULL. We had designed the algorithm thinking about what to do in each stage and how to manipulate the data structures when they were populated, and had failed to consider the starting scenario when they are empty.

Bug #2
Another one of our bugs occurred in issue_To_execute, when we were checking the reservation stations for instructions that are no longer waiting on RAW hazards, but we were incorrectly storing these instructions that are ready to execute. When looping through each reservation station and checking if the instruction within it didn't have any tag values, we were storing that instruction in our 'ready_to_execute' array at the same index as the reservation station being analysed, which resulted in our 'ready_to_execute' array to have NULL values in between indices that contained instructions. These interspersed instructions and NULLs broke our bubble sorting of the 'ready_to_execute' array, as it expects contiguous instructions in the array.

Brief Statement of Work
KC Tremblay & Danja Papajani: worked on all deliverables (½ of workload each)