ECE552, Fall 2014

Lab 3 Report

Timmy Rong Tian Tse (998182657)
Brian Aguirre (998528213)

Here are the following "total numbers of cycles with tomasulo' ' (`sim_num_tom_cycles`): we received 1814117 for `gcc.eio`, 1852942 for `go.eio`, and 1979818 for `compress.eio`. We will describe briefly our code for each stage of the algorithm.

1) **Fetech Stage**
   In this stage, we simply check if there are anymore instructions on the `trace` and if there are, we fill the `instr_queue` to the brim with instructions. In the real tomasulo algorithm one instruction gets put into the queue each cycle, we simulate this behaviour in the dispatch stage. We opted to fill the queue to the fullest every time because it made the implementation easier.

2) **Dispatch Stage**
   In the function `fetch_To_dispatch`, we use a `for` loop to scan through the `instr_queue` to find and assign the first instruction that is not yet been put into the dispatch stage to the dispatch stage. This essentially simulates in a hardware-implemented tomasulo, the assigning of an instruction to the dispatch stage once it enters the IFQ.

3) **Issue Stage**
   In the function `dispatch_To_issue`, we find the oldest instruction in the `intr_queue` (the first element) and attempt to find a free reservation station for this instruction. If that is successful, we check the `map_table` for each of the input registers of that instruction and if a value does exist in the `map_table`, then the dependencies in the reservation station (`Q[i]`) are assigned to the value held in the reservation stations. Finally, if the instruction writes to a register, we update the `map_table` accordingly with the output registers of this instruction (i.e., have the `map_table` to show that the output registers of this instruction is being produced by this instruction). We also made sure to skip any registers that had the `DNA` code. Note that unconditional and conditional branches are handled as special cases in this stage. Because they do not issue to the reservation stations, use any functional units, write to the common data bus and they do not cause control hazards, they can be immediately dequeued from the `intr_queue` to retire.

4) **Execution/Memory Stage**
   This stage is mainly realized through the `issue_To_execute` function. This function calls a helper function (`issue_To_execute_helper`) twice—the first time with integer reservation stations and functional units and the second time with floating point reservation stations and functional units. Essentially, the same operation is perform to both integer and floating point so we thought it would be a good idea to practice code reuse by implementing the operations into a function and calling this function twice. The first thing that this helper function does is create a copy of the current reservation station and removing all of instructions that are currently in the execute stage in the copy. We do this because we want to put into execute stage, only the instructions on the reservation station that are not already in the execute stage. Next, we keep assigning "ready" instructions to a matching functional units as long as there are still "ready" instructions remaining on the reservation stations and there is a matching functional unit that is free. An instruction is "ready" to enter the execute stage when

it is no longer waiting on any dependencies to finish (i.e., when all elements in `Q[i]` are `NULL`). We assign these instructions in the priority of the oldest instruction (instruction with the smallest `index`).

5) **Writing to the Common Data Bus**
This stage is realized in the `execute_To_CDB` function. In this function, we look through the reservation stations and find the oldest instruction in execute that is finished executing. After we have found this instruction, we remove this instruction from the reservations stations and functional units and finally, we broadcast this instruction onto the common data bus. Note that store instructions are handled as special cases in this stage. Because store instructions do no write to the common data bus, we can immediately remove all store instructions from the reservations stations and functional units and retire them once execution is complete.

In the function `CDB_To_retire`, we loop through the reservation stations and find any dependencies that are dependent on the value that is on the common data bus, and remove this dependency. Lastly, the instruction is retired by setting the common data bus to `NULL`.

Other helper functions that we coded in our implementation includes the follows: (1) `remove_insn`, a function that removes an instruction from a data structure that holds instructions (i.e., reservation stations and functional units). (2) `get_ready_fu`, this function retrieves the index of an empty integer or floating point functional unit or it returns negative one if there is no free functional unit. (3) `get_oldest_ready_rs`, this function finds the oldest instruction in the reservation station that is ready to go to the execute stage. (4) `dequeue`, this function dequeues an instruction out of the `instr_queue`. (5) `get_next_non_trap_instr`, this function finds the next instruction in the `trace` that is not a trap instruction. The simulation completes when the `instr_queue` is empty, the reservation stations are empty, and the functional units are empty.

To ensure that our code was correct, we made sure that the each subsequent stage is always assigned in an increasing order fashion (i.e., execute will always happen later than issue). We also checked that each subsequent instruction exhibited a general increasing cycle assigned to all stages. Lastly, we gauged to see that the total number of tomasulo instructions was not completely off. To test it more rigorously, we took the first few instructions and traced the stages manually to ensure that the program output was reasonable.

Indeed, we encountered many bugs along the coding process. Luckily, segmentation faults were relatively easier to solve thanks to `gdb`. Another tool that came real handy in detecting bugs was the `assert` statement. This statement is helpful because it allows programmers to perform sanity checks by inserting these statements into a line in code to test that a condition is what is expected. Last, the print statements are always an invaluable tool for debugging. Printing out the instructions, reservation stations, and functional units allows us to see a snapshot of the algorithm at a given cycle. One of the toughest bug that we encountered was that in the `issue_To_execute` function, every time we assigned an instruction that was ready to execute to a functional unit, we forgot to remove that instruction from the

copy of our reservation station. This lead the buggy program to generate tomasulo cycles that were extremely large: it was indicated in the output that the last instruction in the program finished in approximately three million cycles. Another hard bug that we encountered was that in a previous implementation of `issue_To_execute`, the function was not detecting non-executing instructions and assigning them to executing correctly. This lead the program to exhibit the behavior where instructions would be retired before going into the execute stage. In the output, it would show that all instructions have zero as the value for their execute stage. In the end, we solved this bug by rewriting the `issue_To_execute` function.

The work of this lab was distributed as follows: Tim implemented the functions `fetch`, `fetch_To_dispatch`, `dispatch_To_issue`, and wrote the part of this document that explained the functionality of the code. Brian worked on the functions `issue_To_execute`, `execute_To_CDB`, `CDB_To_retire` and the remainder of this document.