<u>Lab Assignment 3: Dynamic Scheduling with Tomasulo</u>

Angel Serah: 1000491773
Harshita Huria: 1000980398

1. Report the "total numbers of cycles with tomasulo" for the first one million instructions of each EIO trace

EIO trace	**sim_num_tom_cycles**
gcc.eio	1814118
go.eio	1852943
compress.eio	1979819

2. Briefly describe your code for each Tomasulo stage (i.e., provided function) at an algorithmic level. Make sure to point out any cases that required special handling. Also include high-level descriptions of any significant helper functions you wrote.

- *fetch_To_dispatch:*
    - Calls fetch, which grabs an instruction from the instruction trace (if IFQ has space) and updates the dispatch cycle when the instruction enters the IFQ.
    - Keeps fetching instructions until valid instruction is found (i.e. not a TRAP instruction)
- *dispatch_To_issue:*
    - Moves instruction from the dispatch stage to the issue stage.
    - Checks if instruction at head of IFQ is branch. If so, do nothing. Else, determine the type of instruction (FP or INT) and check its corresponding reservation station availability and update issue cycle.
    - If issued, check if any of the input regs are waiting on another instruction (from the map table) and point to that instruction in its Q array (i.e tracking the RAW dependencies).
    - If the instruction writes to an output register, update the map table accordingly.
- *issue_To_execute:*
    - We have a Reservation Station status array (0: instruction is waiting for dependencies to be resolved. 1: Dependency has been resolved and it will wait for an available FU. 2: Instruction has been scheduled to its FU). We have a status associated with every instruction in the rsINT and rsFP.
    - We check if any instruction in the RS is ready to execute (no dependencies in Q array). If so, we set its status to 1.
    - As long as there are available Functional Units for that instruction type, we find the oldest instruction whose status is set to 1 and assign it to a FU. Once it has been assigned, we update its status to 2 and update the execute cycle. This way

we can schedule multiple instructions to start execution as long as there are FU available.
    - We also have a FU latency tracker table which stores the cycle when the instruction is ready to be broadcasted (i.e. the cycle after the computation finishes). If instruction is scheduled to FU, we update this latency tracker table (add 4 cycles for INT, 9 for FP to current cycle).
    - Then, we check if CDB is broadcasting and clear dependencies for dependent instructions.
- *execute_To_CDB:*
    - We have a Functional Unit status array (0: Instruction is still executing. 1: Instruction is done executing is ready to be broadcasted. We wait here if CDB is unavailable 2: Instruction has broadcasted on the CDB). We have a status associated with every instruction in the fuINT and fuFP.
    - Compare the current cycle with the latency tracker table entry for every instruction in the FU to check if it's done executing and is ready to be broadcasted. If so, update the status to 1. An exception is made to STORE instructions which are not broadcasted but directly moves to the clear instr stage with status set to 2.
    - Find the oldest instruction in fuINT and fuFP which have a status of 1. Broadcast the oldest instruction between the two and set its status to 2. Update the CDB cycle.
    - Clear the instruction from RS, map table and FU if its status is 2
- *CDB_To_retire:*
    - Set the CDB to NULL
- *Is_simulation_done*
    - Check if the pipeline is completely empty and we have fetched all the instructions in the trace. If so, we are done simulation.

3. Explain how you tested the correctness of your code

We wrote a print function (print_stats) which displayed cycle by cycle statistics of the IFQ, RS, FU and the instructions within them. We used this to debug and verify the correctness of our code. We started with 20 instructions in the beginning and looked through the cycle trace to determine correctness. We slowly increased the number of instructions until we encountered a bug. This bug was realized as either the program hanging, segmentation fault or logical discrepancies in the cycle trace, in which case, the print_stats function helped to narrow down the error.
This process was repeated for a million instructions for all the three traces.

4. Briefly describe the two toughest bugs you had while developing your Tomasulo code.

1) In the beginning, we called all the functions in pipeline order (fetch_To_dispatch -> dispatch_To_issue-> issue_To_execute -> execute_To_CDB -> CDB_To_retire). When we printed the instruction trace we saw that the instructions weren't going through the stages in separate cycles. For example, the first instruction fetched, decoded, issued and

started execution in the same cycle. We were able to fix this issue by calling the functions in reverse order. This ensured that the oldest instruction at a later stage executed first before a later instruction at an earlier stage.

2) Our program was hanging on the gcc trace because one instruction was never scheduled to the FU as it had a dependency. However, the instruction that it had a dependency on was already broadcasted. We realized that when the instruction was fetched and dependencies were calculated, the map table still had a value for the broadcasted instruction which the fetched instruction was dependent upon. This was because we were clearing the map table for broadcasted instructions in the CDB_To_retire stage (i.e one cycle later). This created a false dependency which caused that instruction to get stuck in the RS forever. We fixed this bug by clearing the map table in the same cycle that we broadcast the instruction.

5. Include a brief statement of work completed by each partner.

Angel Serah: 50% of the workload
Harshita Huria: 50% of the workload