Bing Li, 1004191910
Tapasvi Patel, 1004176606

1.0 Results

Benchmark	Total number of cycles with Tomasulo - for first 1000000 instructions
gcc	1534452
go	1572438
compress	1650875

2.0 Function Implementation

Below are the functions involved and how they were implemented:

1. **`fetch`**
    - Check if instruction queue (IFQ) is full or if all the instructions have been read
    - If not, read the next instruction (skipping trap instructions) and push into the IFQ
2. **`fetch_To_dispatch`**
    - Attempt to fetch the next instruction
    - If a new instruction has been fetched, dispatch the instruction
3. **`dispatch_To_issue`**
    - Check if there are any instructions in the IFQ, if so, check if the first instruction can issue:
        - If instruction is a branch/control instruction, it skips the later stages and gets removed from the IFQ
        - If instruction is uses integer functional units (FUs), check if there is an open integer reservation station (RS) entry and push the instruction into it
        - If the instruction uses floating point FUs, check if there is an open floating point RS entry and push the instruction into it if available
    - If an RS was found, note the cycle instruction as issued and remove from the IFQ
    - Set the input register tags (Q) according to the current mapping table
    - Set the corresponding output registers in the mapping table to the instruction
4. **`issue_To_execute`**
    - Go through the integer RS and find the instructions that have not issued yet (the RS is also a queue so the instructions are always in chronological order) as long as the integer FUs are available
        - Check for RAW hazards by seeing if any of the input instructions have not written back yet, if the input instruction has written back, make sure it has been one cycle since the writeback (cannot enter exec in the same cycle the RAW hazard resolves)
        - If no RAW hazards, the instruction can enter execute and is pushed into the next available integer FU
    - Do the same things but for the floating point RS and FUs
5. **`execute_To_CDB`**

- Go through the integer FUs and find the instructions that have finished execution
    - If the instruction is a store, simply remove it from the integer FU and RS since it does not need to go to the CDB
    - Otherwise remember the first found instruction that is done executing
- Do the same things (minus checking for store instructions) for floating point FUs
- Between the execution completed instructions found from the integer and floating point FUs (if any), put the older one (smaller index) onto the CDB
- Remove the instruction sent to the CDB from its corresponding RS and FU

6. **`CDB_To_retire`**
    - Move the current instruction in the CDB out of the CDB
7. **`is_simulation_done`**
    - Check if all the instructions have been fetched, and that the IFQ and RSs are all empty, if they are, the simulation is done
8. **`main`**
    - Call the aforementioned functions in reverse order to prevent an instruction from going through multiple stages in a single cycle
9. Helper functions were created to help implement the queue structure used for the lab:
    - **`queue_front`** / **`queue_back`**: Get the first/last item in the queue
    - **`queue_is_empty`** / **`queue_is_full`**: check if the queue is empty/full
    - **`queue_find`**: Find the index of an instruction in the queue
    - **`queue_remove`** / **`queue_pop`**: Remove a certain/last instruction from the queue
    - **`queue_push`**: Add an item to the back of the queue

3.0 Correctness

The correctness of the tomasulo algorithm was done in three folds.

`Step 1:`

The first step was to make sure that at every cycle, the correct instructions were being added to the relevant queues. When developing the algorithm, we were continuously printing out what instruction was in which stage and whether it was being added to the relevant reservation station or functional unit queues. We kept a track of when instructions were entering certain queues so we could verify if they were leaving those queues at the correct cycles as well. Instructions entering a particular stage had to follow the rules of the algorithm. For example, an instruction enters the Issue stage once all its operands become available. An instruction must stall for an older instruction when competing for the CDB during the same cycle.

`Step 2:`

The second step was to check if the cycles that each instruction entered a stage was the correct cycle. We had to make sure we checked every kind of instruction type that would occur. For example, checking integer computation instructions, floating point computation instructions, load and store. Each of these instructions have their own latency cycles and different amount of

resources. We also had to make sure that instructions we didn't care about were not being stored into the queues, such as trap instructions. Branch instructions were issued but did not occupy any reservation stations or functional units, so we made sure of this as well. Furthermore, we also checked for instructions that had dependences, and whether they were being stalled correctly. Our algorithm was correct in all these different areas.

Step 3:

The last step we did was to actually compute the instructions by hand for the first 25 instructions. We traced these instructions to see if it matched the output from the simulation. Our algorithm successfully matched the computations done by hand.

4.0 Bugs

Bug 1:

When we completed our algorithm and were testing for correctness, we found out that instructions in the execute to CDB and CDB to retire were not being done in the correct order. For example, when two instructions finished executing at the same cycle, they both would be competing for the CDB. The older instruction should get the CDB, but in our case, the younger instruction was getting it. This was stalling the older instruction for many cycles. Therefore, we fixed this by looping through the fuINT and fuFP queues, finding all the instructions that had finished, and then choosing the one that was the oldest (had the lowest index). A similar bug was seen when trying to retire instructions. We were retiring out of order, however, we should've been stalling and only retiring when the instruction at the start of the IFQ was completed.

Bug 2:

When we first pulled the code, the parameters of the algorithm in the code were different from what was listed in the lab document. However, we assumed the code values were correct and developed our algorithm. Then, we found out we had to use the lab values. In our minds, it should not have made any difference, but when we substituted the values, the algorithm was stuck in an infinite loop. To debug, we printed out every single queue (IFQ, fuINT, fuFP, reservINT, reservFP, CDB) for the first 25 instructions to make sure instructions were being transitioned from one queue to the other correctly. The instructions were moving correctly, but the instructions were not being removed from each queue correctly. What happened was we had mixed up the order of decrementing the tail of the RS queue and setting the tail to NULL. Therefore, there were duplicates within the queue, which caused the infinite loops.

5.0 Statement of Work

Bing - Implemented queue, fetch, fetch_To_dispatch, dispatch_To_issue, and issue_To_execute functions, report

Tapasvi - Implemented main, execute_To_CDB, CDB_To_retire, and is_simulation_done functions, checked correctness, report