ECE552 Lab 3 Report

Marko Ciric (1006723967), Raymond Huynh (1007058238)

Tomasulo Cycles for 1M instructions

	gcc.eio	**go.eio**	**compress.eio**
sim_num_tom_cycles	2,044,707	2,087,686	2,213,139

Tomasulo Implementation

Implementing Tomasulo's algorithm into code requires the instantiation of multiple unique data structures, such as the Instruction Fetch Queue (IFQ), Reservation Stations (RS), Functional Units (FU), and the Common Data Bus (CDB). As instructions move through each pipeline stage, these structures are updated accordingly.

fetch fetches an instruction from the provided instruction trace if there is space in the IFQ. If there is no space in the IFQ, the function immediately returns. Otherwise, it will grab the next non-trap instruction from the trace. Finally, the function adds the fetched instruction to the next available free IFQ slot and increments the instr_queue_size variable.

fetch_To_dispatch fetches and dispatches an instruction from the provided instruction trace. It calls the fetch function, then sets the dispatch cycle of each instruction in the IFQ to the current cycle.

dispatch_To_issue moves instructions from the dispatch stage to the issue stage, provided there are enough resources for the instruction to dispatch. First, the function checks if the instruction at the top (index = 0) of the instruction queue is a control instruction. If so, it bypasses any assignment to an RS or FU and removes the instruction from the IFQ. Otherwise, the function checks whether the instruction uses the INT or the FP FU. For whichever FU it determines to be correct, it will check the entries in the FU to see if there is an empty (NULL) entry. If there is, it will occupy it with the instruction, then set the issue cycle of the instruction to the current cycle. Finally, if an instruction has been issued, it is removed from the IFQ. Additionally, the function checks if the instruction has any RAW dependencies, such that there is a map table entry for the register specified in the instruction's r[i] value. If this is the case, it sets the instruction's corresponding Q[i] value to the map table entry. Finally, the map table is updated if the instruction has output registers, by setting the corresponding output register map table entry (r_out[i]) to the current instruction. If there are no functional units available for the incoming instruction, then nothing happens in this function.

issue_To_execute moves a ready instruction, meaning it's resolved all its RAW dependencies, from the issue stage to the execute stage as long as there is a FU available. So a crucial characteristic of the execute stage is ensuring that the instruction moving to the execute stage has all its source operands marked as ready in their respective RS. Also, if several instructions contend for the same FU, then the oldest instruction is prioritized over the newer ones. Additionally, an instruction holds its RS entry and FU until it completes its execution. These features were accomplished in the code by iterating through both the INT and FP FUs checking if any are available for a ready instruction to occupy for execution, if there is a free FU for its respective instruction type, then that instruction gets inputted into the FU and gets assigned the execute cycle. The check for whether an instruction is ready or not, was done by iterating through the RS entries for both INT and FP and checking for entries that are occupied (not NULL), that are not already in the execute stage (instruction remain this stage for a specific latency), and that all tags for specific RS are NULL (meaning that all registers for those instructions occupying that RS are ready because there is no tag in the map table). If these conditions are true, then an instruction is considered "ready". Next, there is a check to determine priority for the same FU unit if multiple instructions are ready, to do this there is a check for whether the current ready instruction in the RS has an index smaller than the current oldest instruction, in which case that instruction gets updated to be the new oldest instruction; and that instruction is the one assigned to the FU first.

<u>execute_To_CDB</u> moves a completed instruction from the execute stage to the CDB, where it is broadcast to any instructions that require the result. One key characteristic of this CDB implementation is that it can only broadcast one instruction per cycle; therefore the oldest completed instruction is always prioritized for broadcast. To do this, the code checks both the INT and FP FUs for any completed instructions. A completed instruction is tracked by comparing when it first entered the FU plus the FU latency, and the current cycle. Once it finds an instruction, it holds that instruction as the oldest until it finds one older based on its index. This instruction then gets the current cycle as its CDB cycle, and is broadcast. If a completed instruction is found that does not use the CDB, its reservation station and functional unit entries are immediately deallocated and its CDB cycle is set to 0.

<u>CDB_To_retire</u> checks for any instructions in the reservation station dependent on the broadcasted instruction and clears the CDB and any relevant RS, FU, map table entries. The code searches through both the INT and FP reservation stations and clears any Q[i] values currently set to the instruction being broadcast. It also goes through the reservation station, functional units, and map table and clears any instructions matching the one currently being broadcast. After this is done, the CDB is set to NULL to indicate it is ready to broadcast the next instruction.

<u>is_simulation_done</u> checks if the simulation is done. It does so by first checking if the number of instructions fetched is greater than or equal to the number of instructions simulated. Afterwards, it checks that there are no instructions in the IFQ (instr_queue_size == 0). Finally, the code

searches through all entries of all reservation stations and functional units and ensures they are all NULL. If this is the case, the simulation is done.

Code Verification

We are able to verify the functionality of our code by running the benchmark traces at a low max instruction count (15-20) and hand tracing the expected cycle values, RAW hazards, and register values. In particular, we did most of our debugging through the gcc.eio benchmark. In order to validate the states of our structures, we used print statements to print all 4 parameters for each instruction when it retires, a print statement to keep track of the fetch index and instruction queue size within is_simulation_done, and printing out the entire Tomasulo table. By using these outputs, we were able to trace the cycles for each corresponding instruction to ensure it matched our hand traced calculations. In particular, we paid special attention to the latency between execute and CDB, as well as ensuring execute cycles occur 1 cycle after a FU has been freed (when the blocking instruction broadcasts to the CDB).

Challenges

Segmentation faults: Throughout our debugging process, segmentation faults were our most common bug encountered. For example, we would sometimes try to access an index of a NULL instruction, or go out of bounds in our reservation station or functional unit entries. Many of these issues were caused by not considering corner cases, specifically the intended functionality at the beginning (when there are few/no instructions) and at the end (when there are no more instructions). We were able to resolve the segmentation faults by using gdb and adding extra checks to ensure no invalid pointers were being accessed.

Miscalculation of instr_queue_size: One major bug we faced was the miscalculation of instr_queue_size, which was causing the program to never terminate due to the check in is_simulation_done always failing. Initially, the issue was it didn't decrement properly and always stayed at 10/9, even when no more instructions were being fetched. After reviewing our code, we then had it start from 0, but increment in seemingly random increments (e.g. 0, 2, 4, 8, 6, 2, 0) for each cycle, which we validated using print statements. After discovering an issue in the logic of how we decrement the instruction queue after a dispatch, we were able to resolve the bug completely.

Statement of Work

Members split up each of the subfunctions. Marko was in charge of fetch_To_dispatch
fetch, dispatch_To_issue, and issue_To_execute. Raymond was in charge of execute_To_CDB, CDB_To_retire, and is_simulation_done. Debugging and integration of the code was done collaboratively. The report was also written collaboratively.