ECE552, Lab Assignment 3: Dynamic Scheduling with Tomasulo

Sebastian Wardzinski - 1004070676

Simulation Values

EIO Trace	# of cycles to complete (without and with early dispatch/issue following the freeing of resources)
gcc	1 845 463 -(new)-> 1 681 443
go	1 879 383 -(new)-> 1 695 064
compress	1 942 847 -(new)-> 1 851 550

Code Walkthrough

Once again, the file that I submitted has a generous amount of in-line comments and will be the best way for you to get an idea of my code. Note that there are 3 blocks of added code: one in the variables section at the top, one for all of the functions I implemented in the middle, and one in the runTomasulo() function at the bottom (all labelled with the required comments).

I created the following three functions that are called on each cycle of the simulation:

1) execute_to_CDB(int current_cycle)
2) issue_to_execute(int current_cycle)
3) fetch_and_dispatch_to_issue(instrcution_t* trace, int current_cycle)

My implementation is such that these functions need to be executed in reverse order of pipelines stages to simulate synchronous operation. Note that since this simulation only cared about the control signals, the CDB was not used, only the map table was used to handle dependencies. Now I will explain the algorithms used in each for the three functions above:

1) Fetch_and_dispatch_to_issue: I implemented my IFQ as a circular buffer queue, which is the most intuitive data structure for this use-case. Here I fetch instructions until I reach a valid instruction (INT, FP, COND) to add to my IFQ, if I get a trap instruction I skip it but I increment the 'instructions complete' variable. I then try to dispatch the top instruction in the IFQ, by checking if any of its respective reservation stations have a free entry. If there is a free entry then I remove the instruction from the IFQ and give it the RS entry. I also update the Q array

with the tag values of the input registers in the map table, then I update the map table by setting the map table entries for the output registers to point to that instruction.

2) Issue_to_execute: This stage requires a lot of checks, and requires the resolution of any structural conflicts that may occur. First I try to find a free integer functional unit (FU), upon finding a free FU I then run an algorithm to find the oldest instruction in the reservation stations that has not yet started executing and that has all its required dependencies. If no valid instruction exists then I break out of the loop and explore the situation with the floating point (FP) FUs, otherwise I allocate the integer FU to the oldest instruction then continue checking the next integer FU (since multiple instructions can start executing in the same cycle). The algorithm for handling the integer and FP FUs is exactly the same.

* Update: I realized from Piazza that I can dispatch and issue in the same cycle as the last cycle of the executing block (that will have the CDB in the next cycle) is completing and freeing resources. For this, I added a bool flag that checks if the ready instruction got ready just this cycle (due to the CDB value), or if it was ready previously (was just waiting for FU). If the instruction was ready previously, I set its execution start time as if it was issued last cycle.

3) Execute_to_CDB: Here we also need to handle any possible structural hazards for the CDB, which requires us to find the oldest instruction that has completed, from both the integer and the floating point FUs. After finding the oldest such instruction, its reservation station entry and function units are freed, and the map table entries for the instruction's output registers are cleared (if it still points to this instruction). Also, the current cycle is copied to that instruction's tom_cdb_cycle property to mark the cycle at which this instruction was broadcasted on the cdb. Note that any store instructions do not compete for the CDB, and their RS entry and FUs are freed no matter what on the cycle after they completed the execution stage.

Correctness Testing

I tested the correctness of the code by running a limited number of instructions with each trace, calling "print_all_instr" to see the output table after the simulation has been completed. Then I would copy the instructions printed in the left column and try to manually fill out a (D, S, X, C) table myself to see if it matched the output created by the program. I did this for the first 30 instructions and realized that my dependencies code was faulty. When I fixed the dependency code, my manually filled table and the outputted table matched.

I then compared my values for 1 million instructions to the values of another group, we saw that they were slightly different (<1%), clearly some edge cases were handled wrong for one of us. It turned out that it was me. After fixing the bug (see tough bug #2), our values matched, at that point I convinced myself of the correctness of the program.

I had to go through multiple iterations of manually checking the table and comparing with friends as my interpretation of the specified algorithm also changed.

Toughest Bugs

1) I had trouble incrementing the “insn_complete” counter when needed and for all the times that it was needed. There were a lot of mini problems related to this:

- I knew that I had to skip trap instructions, but I didn’t know if I was supposed to increment the instructions complete count; later I figured out that I should.
- Some of the instructions were neither TRAP instructions nor any of the other valid instructions in the lab, this confused me for a bit. Later I noticed that those instructions had an OP_CODE of 0, which I believe are NO_OP instructions, so I just skipped them without incrementing the instructions complete count.
- There were some off-by-one errors for the last instruction because I was counting the CDB instruction as completed one cycle too late

The above problems were not too hard to fix once I discovered the “print_all_instr” helper function, which let me know if the final instructions of the trace were actually completed by the time that the simulation declared itself complete (when the instructions complete count reached the trace instruction count).

2) The last bug that I encountered was a tough one to catch, it had to do with the way that the CDB resolves structural hazards. This was such a minor mistake, that it only manifested itself starting from the 11000th instruction in the compress.eio file. Essentially, unlike in the correct version of the code below, the conditional statement checked the tom_ISSUE_cycle while the rest of the code was the same, which led to a mismatch of what was checked and what was set. This was a hard bug to catch because even though it's an explicit logical mistake, it only affects some CDB structural hazards, which is a small subset of all timing interactions.

```
if (executing_insn->tom_dispatch_cycle < oldestD) {
    oldestD = executing_insn->tom_dispatch_cycle;
    floatOldestDIdx = j;
}
```

* My two cents: As @286 on piazza said, for me the main issue was that instructions about handling structural hazards were unclear. What added to this problem is that this is different from the Tomasulo behavior assumed in class and for tests. My suggestion is to add some of the stuff from that piazza post to next year’s FAQ, or change the structural hazard behavior so that it's consistent with assumptions from the rest of the course.