ECE552 Lab 3: Tomasulo

Jianwei Sun 1000009821 Yi Fan Zhang 1000029284

Tomasulo Simulation Results

Benchmark	Total tomasulo Cycles	CPI
Go	1 679 858	1.68
GCC	1 630 780	1.63
Compress	1 593 609	1.59

The Tomasulo Algorithm and Implementation

The Tomasulo algorithm contains several data structures, such as the IFQ, Functional Units, Reservation Stations, Map Table, and CDB. Each of these structures are individually defined with their respective internal values to keep track of information as it passes through the pipelined stages.

In the fetching_to_dispatch stage, the IFQ is represented by a circular array that “wraps around” itself. This array is indexed by a push_index which indexes the location to insert new instruction, and a pop_index, which keeps tracks of the head of the IFQ. By checking the location of these two indexes with respect to each other, the program can determine if the IFQ is full or empty. As long as the IFQ is not empty, there are instructions to fetch, and the next instruction isn't a TRAP, the fetch_to_dispatch stage will take the latest instruction from the trace and place it into the IFQ. Finally, a counter will be incremented to keep track of how many instructions were fetched. This counter will be used to keep track of when the program terminates.

In dispatch_to_issue, we first check if the IFQ contains any instructions, and if these instructions are ones that use the reservation stations (branches do not and are simply removed when they are dispatched). Next, the reservation stations are checked to see if there are any stations available by looking at their busy bits. If so, the instructions will be popped from the IFQ, and issued to a corresponding reservation station. At this time, the instruction operand is recorded and the output registers, if relevant, (as Stores do not use output registers) are mapped to the Mapping Table where its tag will be renamed to correspond to the current reservation station number. The input registers are also compared with the Mapping Table for RAW dependencies; if so, then the RS# they're waiting for will be recorded.

At issue_to_execute, we first check if any instructions are available to be executed; this corresponds to checking their T0, T1, and T2 values to see if the RAW dependencies they're waiting on, if any, are available. Afterwards, corresponding functional units are checked to see if they are available. If the program is ready to execute, and a functional unit is available, then we will move that instruction to the executing stage by flipping its “executing” flag on the Reservation Station. It is also important to note that this section is achieved by looping through the reservation stations and observing each respective conditions. Thus, if during the loop, we indexed an instruction that is currently executing, then we will simply skip that instruction and check the next one too see if it's available to execute.

Next, in execute_to_cdb, we first check if any instructions are done executing; this is done by looking

at a variable within the functional unit, cycle_to_completion, which tracks how many cycles it needs before it has finished executing. Next, the instruction are checked if they do write to the CDB, as instructions such as stores do not. Finally,, we check if the CDB is available to be written to. If all these criteria are fulfilled, then the executing executing instruction will be written to the CDB, by recording its RS# on the CDB. Once it is written, the functional units and reservation stations are then cleared and reset so that they're available for the next instructions. Lastly, this stage also decrements the "cycles_to_completion" variable of all functional units that contain instructions; this is how we keep track of the latency of each functional unit.

Finally, in cdb_to_retire, the new value will be broadcast to all the reservation stations. This is done by searching through all Reservation stations to check if any entries are dependent on the specific RS# in the cdb by looking at their T0, T1 and T2 values. If so, we clear this market to indicated that the value it's waiting for is available to be used. The mapping table entries that corresponds to the current "registers" on the CDB will also be cleared to indicated that writeback to memory has happened, and future instruction requesting this specific register can just grab it's value from memory. Finally, the CDB is "cleared" to allow another instruction to write to it in the next cycle.

After each cycle, the Tomasulo function will check if the program has reached the end. We do this by first checking if the instruction fetched is equal to the total number of simulated instructions. Then, we check the IFQ, Reservation Stations, Mapping Table and CDB to see if they're empty. If so, then no instruction are left inside the pipeline and the program has truly ended.

Lastly, to keep track of program order, a global queue of linked list is used. Implementing this linked list are respective enqueue and dequeue helper functions to add instructions onto this queue. As instructions enter the Reservation Stations, they are pushed onto the queue, and popped after they are written to CDB. The queue is traversed during issue_to_execute and execute_to_cdb functions to give precedence to the oldest instruction.

Verification Methodology:

Two major stages of verification were performed for Tomasulo. Firstly, using the gdb debugger we stepped through the Tomasulo, line by line, over 10 cycles of iteration to check if each instruction from the trace correctly goes through the pipelined stages. At the same, time, we would be checking what's stored in RS, and the Mapping Tables to keep track of all the values inside our data structures.
We would also change the number functional units and reservation stations to 1 and 100, to test how the program would behave when there's a lot of competition for resources, or if there is no competition for resources.

Finally, we would run Tomasulo over a few instructions, and compute them by hand using the Tomasulo tables from class. Inside the main loop, we printed each instruction and their respective cycles in each stage of the Tomasulo. This would be compared with hand calculations to verify correctness. We would also run Tomasulo over a larger amount of instructions and check if there's any unusual cycle reports, such as instructions that spend an extremely large amount of cycles in a particular stage (near 100 or more) or mismatched execution latency. This may indicate that the instruction was stuck in a particular cycle, and the particular case is explored and debugged.

Toughest Bugs:

Below are two of the toughest bugs we've encountered. These bug were caused by a number of sources involving writing over memory references, errors in mapping values across data structures, checking for loop break conditions in the wrong order, missing corner cases. Our solution to these faults were to setup hardware breakpoints in gdb, allowing us to observe when the data at the referenced memories changed. In addition, we single stepped through the code in gdb to see when expected conditions were not met.

1. **Segmentation faults :** One of the most difficult bugs is the infamous segmentation fault. The Tomasulo code uses a wide range of different data structures, as well as multiple checks going across the pipeline. As instructions come into the reservation station at different rates, and leave out of order, it's very easy to index values that are NULL, and access out-of-bound memories. For example, one of the segmentation faults occurred in execute_to_CDB, where we checked if the functional unit has completed it's execution; because we originally checked it's cycle_to_completion before determining whether or not there's actually an instruction running inside, we would often witness segmentation faults. The order of checking values is always important, and we must always first check for null values. In addition to checks, due to the mapping of instructions across a variety of data structures, indexes and tags must be carefully coded to not go out of bounds or access the wrong data. As we use "-1" or "NULL" to represenety empty spaces, accidentally using one of these values to index a data structure will raise many errors. In general, mismapped data causes create many error in the instruction flow across Tomasulo, with some of the worse cases being segmentation faults.

2. **Infinite Loops:** During the first few runs of Tomasulo, we encountered infinite loops within the program. After stepping through every stage we found that there were many causes for this. Firstly, fetching must be stopped after the last instruction goes into the IFQ; bad conditions such as off-by-one errors can lead to fetching extra instructions, and if it happens, "garbage" instructions, without matching functional units will be stuck inside the IFQ or Reservations stations, causing Tomasulo to never terminate. Secondly, as instruction moves on to each stage, certain flags must be raised and values cleared. As logic pile up, sometimes these "resets" and markers are misplaced, or forgotten, which causes instructions to be forever "stuck" inside structures. Particularily for corner cases, such as stores, that has a slightly different data path for Tomasulo, special checks needs to be considered; one of our errors involved "SW" instructions being stuck in the pipeline because we originally only cleared RS values at cdb_to_retire stage, which does not hold store instructions. Finally, cycle-by-cycle breakdown of each task and movement need to be carefully planned out. For example, in our code, we had a case where cdb broadcasting, clearing values, and moving instructions out of executing was all in the same cycle. What this mean is that register values will be retired, and used at the same time as the transfer of data from execute and cdb. Of course, these conflicts can often result in instructions on the RS not getting the values they were waiting for, making them wait forever for a value that was already cleared.

Conclusion

Tomasulo was implemented in sim-safe, and it's performance, algorithm, verification methodology are detailed above. Through a series of discussions, both members of the group contributed equally to the implementation of the Tomasulo algorithm, as well as the debugging of the code. During verification, Yi Fan focused on the hand computation vs simulation results of the structure, while Jianwei focused on stepping through each part of the code. However, both members contributed to all parts of the verification stage.