Lawrence Park (995936395) Sam Shao (1000084151)

Lab 3 Report: Dynamic Scheduling with Tomasulo

1. Benchmark results

Trace file	Total number of cycles with Tomasulo
gcc.eio	1839597
go.eio	1884003
compress.eio	2068600

Table 1. Total cycles obtained from executing 1 million instructions for each trace

2. Tomasulo stages

The simulation of Tomasulo OoO pipeline is broken into 5 distinct stages in the main loop, located in the **runTomasulo(instruction_trace_t *trace)** function. Each iteration of the main loop represent activities of every Tomasulo stage in a single cycle, and the stages run parallel to each other in a given cycle. The following describes how the 5 stages are implemented in its respective function.

1. Fetch-to-Dispatch Stage (D) : ***fetch_To_dispatch(instruction * trace, int current_cycle)***

   The stage consists of two main parts: fetch, and dispatch. The fetch process is implemented by a separate function named **fetch(instruction_t *trace)**, and it reads a next instruction in program-order from the trace pointer, and store it in the instruction fetch queue (IFQ). In reading the next instruction from a given trace, a TRAP instruction is considered "invalid," and the fetch function will increment its index until a valid instruction is found. The IFQ is represented by a global array **instr_queue[INSTR_QUEUE_SIZE]**, and operates as a circular buffer .

   On the other hand, the dispatch routine reads a single instruction in the same cycle, and dispatch the instruction to the reservation stations (RS), if one is available. If not, the dispatch stage is stalled due to a structural hazard on RS. Once a RS entry is selected, the dispatched instruction is checked for any RAW dependency by scanning the map table; if the dispatched register contains a source register that is tagged in the map table, the identified source operand is tagged on the RS entry.

   Finally, it is important to note that the reservation stations are organized into the two global arrays by opcode type: integer and floating-point. Therefore, a number of available RS entries is constrained by a dispatching instruction's opcode, and the dispatch stage may stall even if a RS entry, not matching the instruction's opcode, is available.

2. Dispatch-to-Issue stage (S) : ***dispatch_To_issue(int current_cycle)***
   A dispatched instruction on RS is issued on the following cycle. The stage scans through RS entries for an instruction that is not yet issued. Since all instructions are dispatched in program order and a single instruction is dispatched per cycle, it follows that the dispatch-to-issue stage also issues instructions in program order.

   Finally, it is important to note that RAW dependencies are enforced by delaying an instruction to proceed to a functional unit; this differs from the lecture slides since an instruction is issued only when RAW hazards affecting the instruction are resolved. In this assignment, the consecutive Tomasulo stage – the issue-to-execute stage – implements RAW hazard resolution.

3. Issue-to-Execute stage (X) : ***issue_To_execute(int current_cycle)***
   The stage assigns issued instructions from RS to idle functional units. The functional units are classified as integer or floating-point, just like RS entries, and they are modeled as instruction pointer arrays in the code : fuINT[FU_INT_SIZE] and fuFP[FU_FP_SIZE]. To identify an idle functional unit, the arrays are scanned to check if an element is a null pointer. If an idle unit is found (i.e. an array element is null), the stage searches a RS entry that is issued and free of a RAW hazard; this is done by checking its tags are cleared (i.e. null pointers). Once an eligible instruction is found, it is assigned to the available functional unit, and its tom_execute_cycle is assigned the current cycle. Unlike the previous two stages, the stage may execute multiple instructions in out-of-order; this enables the Tomasulo processor to exploit ILP in a program.

   Finally, should there be multiple instructions that are ready to execute, the instruction that is earlier in the program order is chosen. This comparison can be done by looking at its dispatch cycle which is assigned when it entered the RS table.

4. Execute-to-CDB stage (C) : ***execute_To_CDB(int current_cycle)***
   The main task of this stage is to identify a functional unit that completes its assigned instruction in the current cycle. For each active functional unit, the stage computes the cycle index at which the assigned instruction is due for completion; this is done by adding the latency of functional unit to the start cycle of execution (i.e. tom_execute_cycle). It is important to note that the functional unit and its RS entry should be “vacated” on the last cycle of execution, so that next instructions may occupy the completed instructions’ resources on the next cycle. The completed instruction is assigned to CDB, and it occupies CDB on a cycle after the completion. It is important to note that there can only be one instruction on CDB at a time; therefore, should there be multiple instructions due for completion, a later instruction(s) is stalled until CDB becomes available.

   A store instruction is an exception for this stage, because it does not require access to CDB, and does not cause a contention to CDB. Finally, the stage also clear stags on the map table and RS entries, if there is a valid instruction on CDB.

5. CDB-to-Retire stage (R) : ***CDB_To_retire(int current_cycle)***
   I believe the "retire" stage is a misnomer for Tomasulo algorithm without ROB. In the lecture, the retire stage ensures in-order commit of register values after the instruction completes its out-of-order execution using the ROB's FIFO data structure. On the other hand, the Tomasulo algorithm without ROB does not have a notion of retire stage, because it does not guarantee in-order commit of registers. For the assignment, the stage simply deallocates the reservation station of the completed instruction.

3. Testing strategy

We employed two main techniques to check functional correctness of the simulator: assertions and manual observations. Assertions are embedded in the Tomasulo code whenever we know certain conditions must hold true. For an example, we know that instructions in a trace must have their dispatch and issue cycles incrementing with the program order; therefore, an assertion can be made to check if all instructions are dispatched and issued in-order. Another example is the number of cycles an instruction may take between execution stage and CDB. We know that an integer instruction may take from 4 to 6 cycles because the instruction latency is 4, and it may stall up to 2 cycles due to the CDB contention. In our code, check_all(*trace, sim_num_insn) function run different assertions on each instruction in a given trace. This approach allows that the code to be tested incrementally, and this makes it easier to isolate a problem.

Manual observations of important program states, such as the reservation stations and functional unit arrays, was an important part of verification process. While this process is not easily reproducible, the manual inspection is a practical way to make sure that the simulator is correctly updating the data structures in each Tomasulo stage. Finally, we used the print_all_instr(* trace, sim_num_insn), and see if the printed Tomasulo cycles for each instruction matched the expected results.

One of the challenging issues we encountered is determining a correct stage & cycle at which RS/Functional unit are freed, and tags are updated for a completed instruction. A freed up resource should trigger scheduling other instructions to proceed in the same cycle, but the RAW tags should be updated on the next cycle; it wasn't obvious how to organize this in the code. Another tough issue was handling corner cases, such as a store instruction completing execution; since a store does not require CDB stage, we had to carefully design the control flow of the Execute-to-CDB stage.

4. Division of work

We pair-coded to complete most of the assignment because we believed that the tasks are tightly coupled and not easily divisible.