Lab 3: Dynamic Scheduling with Tomasulo

Total Number of Cycles with Tomasulo

EIO Trace	Number of Cycles
gcc.eio	1732061
go.eio	1790990
compress.eio	1890807

Overview of Implementation

Core Functions

is_simulation_done()

This function returns true only when the whole machine is completely drained. First it checks if (fetch_index > sim_num_insn) meaning that all of the instructions have been processed. Secondly we check that the IFQ is empty. Thirdly we check that both INT/FP reservation station arrays are empty (i.e. NULL). Then that all INT/FP functional units are empty, and finally that the CDB is NULL. This ensures that we stop when the pipeline is empty, not simply when the last instruction is fetched.

CDB_To_retire()

This simply frees the one-cycle CDB at the start of the cycle. This is done by setting the commonDataBus to be NULL. So when an instruction finishes it can be written into an empty CDB. Current_cycle was not needed directly here because the one-cycle occupancy is enforced when tom_cdb_cycle is set in execute_to_CDB.

execute_To_CDB()

Function works by finding all instructions that finish executing this cycle by comparing latency and when they entered execution. It also handles which instructions can use the CDB. First it scans each INT FU and, if tom_execute_cycle > 0 and current_cycle ≥ tom_execute_cycle + FU_INT_LATENCY, then this is a contender instruction for using the CDB next cycle. Stores don't use the CDB, so these get immediately freed upon completion, and tom_cdb_cycle is set as 0. All INT writers go into a completing_int array, this is repeated for FP using the same logic and those are stored in a completing_fp array. From both lists it picks the oldest by index, which shows the order the instruction executes in. If the CDB is free, the oldest instruction is placed on the CDB, and tom_cdb_cycle = current_cycle, and its RS and FU is cleared. Younger instructions simply keep holding their RS and FU and wait in EX until they later win the CDB. This enforces one CDB user per cycle, oldest-wins, and the store special case.

issue_To_execute()

Function promotes ready RS entries into respective FUs. An instruction is eligible if it entered Issue in a prior cycle (tom_issue_cycle > 0 and < current_cycle), and has all of its r_in entries ready. For each entry/source, Q[i] == NULL means ready, otherwise the producer must have tom_cdb_cycle < current_cycle which enforces that there is no

forwarding/bypassing support directly to functional units. All ready INT and FP instructions are sorted in a list by index, and then are allocated to available FU slots. When they are allocated tom_execute_cycle = current_cycle is set and the RS are deliberately kept as is as only when it gets the CDB does it release its RS and FU (FAQ #15).

dispatch_To_issue()

This sets the Issue cycle for instructions that were placed into an RS in a prior cycle. For loop through all INT/FP RS entries, and if tom_dispatch_cycle < current_cycle and tom_issue_cycle == 0, then set tom_issue_cycle = current_cycle. The rationale is the lab timing: Dispatch is when an instruction enters the IFQ; in the same cycle, if it is at the head and a matching RS if free, move it into the RS. But the issue must be at least one cycle later.

fetch()

Brings at most one non-trap instruction into the IFQ tail, skipping traps. If the IFQ is full or the trace is completed then do nothing. If only traps remain, it sets fetch_index = sim_num_insn as an optimization so the simulator won't keep trying to fetch. This is done in one cycle as there is a while loop that skips traps until the first non trap instruction is found.

fetch_To_dispatch()

This ties fetch and in-order dispatch together. It saves the IFQ size, then calls fetch. If the size increased, then the newest entry's tom_dispatch_cycle = current cycle (Dispatch is defined as an IFQ entry). Then looking at only the head of the IFQ, if its a branch then it is removed from the iFQ, as they never occupy RS/FU/CDB, but they do get a dispatch cycle number. Otherwise if there is an empty RS, the head is placed there and the RAW tags are set for the three r_ins. If a source register is unused (DNA/negative) or register 0, Q[i] = NULL, otherwise Q[i] = map_table[src]. For destinations, the map table is set so that map_table[dest] = this instruction. Finally, the instruction is popped from IFQ. If there's no free RS, leave the head in place and stall.

Helper functions

get_free_rs_entry

Given a reservation table and its size, this function returns the index of the first free entry. If there are no free entries, then returns -1.

update_map_table

Given an instruction and the map table, this function checks each source register of the given instruction to see if an older instruction will produce that value. If so, it will mark those as RAW dependencies. Then, the map table is updated so that the given instruction becomes the latest producer of its output register.

remove_instr_from_ifq

This function takes the instruction queue, its size and an instruction as arguments. The function pops the instruction from the queue and updates the queue size.

instr_dispatched

This function returns true if an instruction has been dispatched in the previous cycle. It takes in the instruction and the value of the current cycle as arguments.

instr_ready_to_execute

This function returns true if an instruction is ready to be executed. Given the instruction and the current cycle, it checks if the instruction has been previously issued and if RAW hazards are resolved.

sort_instr

This function takes in a list of instructions and orders them oldest to youngest.

allocate_fu

This function takes in a list of INT of FP instructions ready to be executed and waiting for a functional unit. It also takes in INT of FP functional unit data structure, size of available INT or FP functional unit, and current cycle. It clears the dependency pointers of a ready instruction and assigns them to an entry in the respective functional unit.

free_entry

This function takes in the reservation station data structure, and an instruction. It frees the entry of the reservation station that the instruction occupies.

Correctness of Code

We uncommented the print_all_instr(instruction_trace, sim_num_insn); line in sim-safe.c and printed out the Tomasulo Table for the first 20 instructions for the compress.io benchmark. We then filled out the Tomasulo Table by hand for the first 10 instructions and compared it with the generated output.

Toughest Bugs Encountered

1. Segmentation Fault: We created a data structure that would point to all of the instructions that had finished executing. Since the oldest instruction had to be broadcast on the CDB, we needed to sort that data structure from oldest to youngest. However, we would begin ordering the list before any FU had finished execution, causing us to access invalid entries of that data structure.

2. Incorrect Dispatch Cycle Assignment: Another bug was related to when instructions were assigned their dispatch cycle timestamp. Initially, the dispatch cycle was set when instructions were allocated to reservation stations, rather than when they entered the IFQ. This caused a cascading timing error throughout the simulation. Instructions appeared to stall longer than expected in early pipeline stages. For example, an sll instruction at position 6 was showing a dispatch cycle of 8 instead of 6, creating artificial delays. After comparison with expected outputs, we realized that according to the spec, an instruction "enters dispatch” immediately upon entering the IFQ, not when it later moves to a reservation station.

Statement of Work

Both team members worked on all parts of the lab together.