Tomasulo Processor Simulation
ECE552 Lab 3

Jay Mohile
Letian Zhang

I. BACKGROUND

Traditional in-order pipelined processors are susceptible to two critical issues that limit their throughput. First, they are incapable of executing independent instructions in parallel. Second, they are subject to head-of-queue blocking, where a complicated instruction with unresolved dependencies may prevent the execution of subsequent ready instructions. The Tomasulo architecture makes improvements to both these areas.

II. METHODS

For each major stage of the Tomasulo pipeline, a high-level method was developed. Within each stage, helper-methods were developed to simplify the logic and achieve code-reuse (e.g between the integer and floating point structures). They are detailed at an algorithmic level below.

Data Structures

- *Reservation Station:* Holds a pointer to the currently assigned instruction, or null.
- *Functional Unit:* Holds a pointer to the current assigned reservation station. Since these are deallocated together, it is easier to track stations than instructions.
- *Reservation Station / Functional Unit Arrays:* For each pipeline (int, fp), separate arrays of stations and units are stored.
- *Map Table*: A map from architectural registers to the in-flight producer instruction's pointer, if any.

Fetch To Dispatch

First, we try to fetch a new instruction from the trace. We skip this if the instruction fetch queue (IFQ) is full or if there are no more instructions to fetch (num_sim_insn reached). Once fetched, we buffer to the IFQ. Note that if a TRAP is found, we skip buffering and continue fetching until this is not the case.

Dispatch To Issue

If the IFQ is not empty and reservation stations are available, pop from the IFQ and, unless the instruction is a branch, allocate to a reservation station. Additionally, we apply register-renaming of outputs and dependencies using the *Apply Register Renaming* helper.

Helper: Apply Register Renaming

First, map inputs by copying in instruction pointers for each input register (r_in) based on the map-table. Ready or unused inputs are null-mapped.

Then, map outputs by copying a pointer to this instruction to each output register's (r_out) map-table entry.

Issue To Execute

We observe that at this stage, except for read-dependencies, the integer and floating point pipelines are independent. Thus, we offload all work to the *Move To Execute If Ready* helper, and invoke it separately for each pipeline.

Helper: Move To Execute If Ready

Given a bank of reservation stations, we filter down to those that have an instruction, whose instruction has been in the issue stage for at least one cycle, and whose instructions are free of RAW dependencies. Call this set X.

Then, for each unallocated functional unit within this pipeline, we assign the oldest station from X. When either units or stations are exhausted, we finish.

Execute To Broadcast
Once an instruction that generates an output finishes execution, it must broadcast to the CDB to notify any dependents. Although many instructions may finish execution at the same time, only one can use the CDB each cycle. When promoted to the CDB, the instruction's allocated resources (reservation station, functional unit) are available for other instructions to use. Finally, if an instruction finishes execution but does not require the CDB, its resources can be immediately reallocated.

1. Iterate over all functional units, filtering to those whose final execution cycle was the previous one.
2. If the FU contains a store, immediately deallocate. Otherwise, buffer and select the oldest one.
3. If an FU from (2) is buffered, it requires broadcast. Move to the CDB and deallocate.

Broadcast To Retire
As our system does not require precise state, we are not concerned with enforcing in-order retirement. As such, this stage simply:

1. Checks if there is an instruction currently being broadcasted on the CDB
2. Nulls the map-table entry currently claimed by that instruction, if present.
3. Notifies any waiting reservation stations that this value is ready.

Since we are simulating timing, not actual functionality, it is sufficient to simply clear dependencies with NULL, rather than copying values.

Helper: Notify Reservation Stations
Given a completed instruction, walks through a bank of reservation stations to cancel any dependencies (Q) for this instruction.

III. TESTING

Overall, this lab was difficult to verify. While the individual operations are straightforward, the emergent behaviors that exhibit over thousands of instructions are not. The following four strategies were used for verification.

First, after running a limited sample of 10-50 instructions, Tomasulo tables were dumped through the print_all_instrs method and GDB. This allowed us to compare the results to hand-calculations, and check for obvious issues. In particular, the first 15 instructions of gcc.eio served as a regular sanity check and surfaced many issues. Please see Appendix A for analysis.

In addition, given more nuanced behavior such as FU contention, GDB was used to individually step through and verify correctness.

Another source of concern was floating point operations. Using conditional breakpoints, we verified that neither the gcc nor the go benchmark utilized them - so a targeted portion of the compression bench was hand-checked.

Finally, the results seem statistically reasonable. GCC/Go are purely integer programs, so it makes sense that their results are within 1% of each other, while Compress is slightly higher.

IV. RESULTS

Full results are included in Appendix B.

Benchmark	Tomasulo Cycles (1M Instructions)
GCC	1681443
Go	1695064
Compress	1851550

V. DISCUSSION

The first interesting bug was caused by an incorrect register-renaming algorithm: updating the map-table before copying in dependency pointers. As discussed above, this leads to self-referential instructions, when of the form: *op **x1**, x2 -> **x1***. These will be stuck in the issue stage forever, since that last RAW dependence never clears.

In our case, this manifested as infinite loop behavior given an input trace long enough to eventually have instructions of this form. Triangulating the issue down to self-referencing Q tables was tedious, and required individually stepping through a number of cycles. Here, instrumenting the project to work with VS Code's GUI-based debugger was a good investment.

The second, and far more arduous, bug involved correctly ordering each stage of the pipeline within the runTomasulo method. This order is crucial, as it implicitly encodes factors such as: whether structural resources can be immediately re-used, whether data-forwarding exists, and how long an instruction has been in a stage. It was particularly complicated due to the presence of off-by-one errors, and varying conventions between discussing stage-enter and stage-leave restrictions.

Some examples of enforced orders:

- IssueToExecute must precede BroadcastToRetire, since completed RAW errors can only be executed on the next cycle.
- ExecuteToBroadcast must precede IssueToExecute, since freed FU resources can be immediately re-used.
- etc.

VI. STATEMENT OF WORK

Both partners worked on both aspects of this project, code and report.

VII. APPENDIX A: Tomasulo Analysis

TOMASULO TABLE

lw	r16,0(r29)	1	2	3	8	
lui	r28,0x1003	2	3	4	9	
addiu	r28,r28,20912	3	4	10	15	
addiu	r17,r29,4	4	5	6	11	
addiu	r3,r17,4	5	6	12	17	
sll	r2,r16,2	6	8	9	14	
addu	r3,r3,r2	7	9	18	23	
addu	r18,r0,r3	8	11	24	29	
sw	r18,-21500(r28)		9	14	30	0
addiu	r29,r29,-24	10	15	16	21	
addu	r4,r0,r16	11	17	18	24	
addu	r5,r0,r17	12	21	22	27	
addu	r6,r0,r18	13	23	30	35	
jal	0x5fb810	14	0	0	0	
addiu	r29,r29,-24	15	25	26	31	

Key Analyses:

1. Operations that use execute and broadcast (e.g I1, I2) spend 5 cycles in Execute, and the 6th in Broadcast.
2. No instructions share the CDB (right column)
3. Stores do not use the CDB (I9)
4. No more than 5 instructions use an integer RS in parallel (I6 stalls before entering Issue)
5. Instructions end dispatch when a reservation station will be available **next** cycle (I6 takes ownership in cycle 8, at the same time I1 moves to broadcast)
6. If a CDB broadcast resolves raw dependencies, that unblocked instruction will not execute until the **next** cycle (I2 unblocks I3 in cycle 9, but I3 does not execute until cycle 10)

IIX. APPENDIX B: Raw Results

Raw results from each simulation.

Gcc.eio

```
sim: ** starting functional simulation **

sim: ** simulation statistics **
sim_num_insn                1000000 # total number of instructions executed
sim_num_refs                 364517 # total number of loads and stores executed
sim_elapsed_time                  1 # total simulation time in seconds
sim_inst_rate          1000000.0000 # simulation speed (in insts/sec)
sim_num_tom_cycles          1681443 # total number of cycles with tomasulo
ld_text_base              0x00400000 # program text (code) segment base
ld_text_size                2166768 # program text (code) size in bytes
ld_data_base              0x10000000 # program initialized data segment base
ld_data_size                 264644 # program init'ed `.data' and uninit'ed `.bs
s' size in bytes
ld_stack_base             0x7fffc000 # program stack segment base (highest addres
s in stack)
ld_stack_size                 16384 # program initial stack size
ld_prog_entry             0x00400140 # program entry point (initial PC)
ld_environ_base           0x7fff8000 # program environment base address address
ld_target_big_endian              0 # target executable endian-ness, non-zero if
 big endian
mem.page_count                  657 # total number of pages allocated
mem.page_mem                   2628k # total size of memory pages allocated
mem.ptab_misses                 664 # total first level page table misses
mem.ptab_accesses           9689019 # total page table accesses
mem.ptab_miss_rate           0.0001 # first level page table miss rate
```

Go.eio

```
sim: ** starting functional simulation **

sim: ** simulation statistics **
sim_num_insn                1000000 # total number of instructions executed
sim_num_refs                 236033 # total number of loads and stores executed
sim_elapsed_time                  1 # total simulation time in seconds
sim_inst_rate          1000000.0000 # simulation speed (in insts/sec)
sim_num_tom_cycles          1695064 # total number of cycles with tomasulo
ld_text_base              0x00400000 # program text (code) segment base
ld_text_size                 621600 # program text (code) size in bytes
ld_data_base              0x10000000 # program initialized data segment base
ld_data_size                 578004 # program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base             0x7fffc000 # program stack segment base (highest address in stack)
ld_stack_size                 16384 # program initial stack size
ld_prog_entry             0x00400140 # program entry point (initial PC)
ld_environ_base           0x7fff8000 # program environment base address address
ld_target_big_endian              0 # target executable endian-ness, non-zero if big endian
mem.page_count                  283 # total number of pages allocated
mem.page_mem                   1132k # total size of memory pages allocated
mem.ptab_misses                 283 # total first level page table misses
mem.ptab_accesses           5914216 # total page table accesses
mem.ptab_miss_rate           0.0000 # first level page table miss rate
```

Compress.eio

```
sim: ** starting functional simulation **

sim: ** simulation statistics **
sim_num_insn               1000000 # total number of instructions executed
sim_num_refs                160349 # total number of loads and stores executed
sim_elapsed_time                 1 # total simulation time in seconds
sim_inst_rate          1000000.0000 # simulation speed (in insts/sec)
sim_num_tom_cycles          1851550 # total number of cycles with tomasulo
ld_text_base             0x00400000 # program text (code) segment base
ld_text_size                103840 # program text (code) size in bytes
ld_data_base             0x10000000 # program initialized data segment base
ld_data_size              44123012 # program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base            0x7fffc000 # program stack segment base (highest address in stack)
ld_stack_size                16384 # program initial stack size
ld_prog_entry            0x00400140 # program entry point (initial PC)
ld_environ_base          0x7fff8000 # program environment base address address
ld_target_big_endian             0 # target executable endian-ness, non-zero if big endian
mem.page_count                  87 # total number of pages allocated
mem.page_mem                   348k # total size of memory pages allocated
mem.ptab_misses                 87 # total first level page table misses
mem.ptab_accesses           4908202 # total page table accesses
mem.ptab_miss_rate           0.0000 # first level page table miss rate
```