ECE552 Lab 4 Dynamic Scheduling with Tomasulo

Freddy Chen 997363124
Anthony Alayo 997487401

Report the "total numbers of cycles with tomasulo" for the first one million instructions of each EIO trace.

gcc.eio
sim_num_tom_cycles 1814117 # total number of cycles with tomasulo
go.eio
sim_num_tom_cycles 1852942 # total number of cycles with tomasulo
compress.eio
sim_num_tom_cycles 1979818 # total number of cycles with tomasulo

Briefly describe your code for each Tomasulo stage (i.e., provided function) at an algorithmic level. Make sure to point out any cases that required special handling. Also include high-level descriptions of any significant helper functions you wrote.

Functions to work with arrays of instructions

We noticed that we were going to manipulating arrays of instructions a lot, so we have some helper functions to make things easier. The functions below are described by the comments directly above them.

```
// Print contents of insns array
void insn_array_print(instruction_t** insns, int size, int current_cycle)

// Check if an array of insns is empty
int insn_array_is_empty(instruction_t** insns, int size)

// Check if an array of insns is full
int insn_array_is_full(instruction_t** insns, int size)

// Get the oldest insn in an array of insns
instruction_t* insn_array_get_oldest(instruction_t** insns, int size)

// Get the oldest insn in an array of insns
// Modified to get oldest instruction ready for execute (all inputs ready)
instruction_t* insn_array_get_oldest_ready_for_execute(instruction_t** insns, int size)

// Get the oldest insn in an array of insns
// Modified to get oldest instruction ready for CDB (finish execution latency)
instruction_t* insn_array_get_oldest_ready_for_CDB(instruction_t** insns, int size, int
```

```
current_cycle)

// Remove store instructions from the functional units that are finished
instruction_t* insn_array_remove_completed_stores(instruction_t** insns, int size, int
current_cycle)

// Insert specified insn into array
// Inserts into first empty slot, else don't insert anything at all
void insn_array_insert_insn(instruction_t* insn, instruction_t** insns, int size)

// Remove specified insn from array
// Won't remove anything if the instruction isn't there
void insn_array_remove_insn(instruction_t* insn, instruction_t** insns, int size)
```

Circular Instruction Queue

Since the instruction queue was actually an array, we made it a circular array to avoid having to shift around too many elements.

```
// Use a circular queue for instr_queue since instructions have to be
// dispatched in program order
static int instr_queue_head = 0;

// Print the instruction queue
int instr_queue_is_empty()

/* Helper functions for instruction queue */
// Check if there's a next instruction
// Return 1 for true, 0 for false
int instr_queue_is_empty()

// Check if the instr queue is full
// Return 1 for true, 0 for false
int instr_queue_is_full()

// Peek at the next instruction
instruction_t* instr_queue_peek()

// Get the next instruction
instruction_t* instr_queue_dequeue()
```

```
// Insert instruction into queue
void instr_queue_enqueue(instruction_t* insn)
```

```
static bool is_simulation_done(counter_t sim_insn)
```

To make sure that the simulation is done, we check that all of the following conditions are true:

- The fetch index is greater than the sim_insn variable, meaning that there's nothing left to fetch
- If the reservation stations for integer and floating point operations are empty
- If the functional units for integer and floating point operations are empty

```
void CDB_To_retire(int current_cycle)
```

The main goal of this function is to remove the instruction on the common data bus. Before we can do that, we have to do the following things:

- Look for instructions in the reservation station that have dependencies on the result being broadcasted in the common data bus and get rid of the dependencies.
- Go to the map table and remove references to the instruction in the common data bus

```
void execute_To_CDB(int current_cycle)
```

We want to make sure that the CDB is free and choose the oldest instruction that's finished executing and put it there. We do this by getting the oldest instructions from the integer and floating point functional units, then getting the oldest instruction overall.
There is a special case when removing store instructions from the functional units. We want to remove all ready store instructions because they do not get written to the CDB.
When we remove an instruction from its functional unit, we also remove it from the reservation station.

```
void issue_To_execute(int current_cycle)
void issue_To_execute_helper(int current_cycle, instruction_t** reserv, int reserv_size,
instruction_t** fu, int fu_size)
```

The main goal here is to look for instructions in the reservation station with ready source operands. We want to move those instructions into free functional units if there are any. So, while the reservation stations are not empty and while functional units are not full, we do the following:

- Get the oldest instruction with ready source operands
- Put that instruction into a free slot in the functional unit

We use a helper function so that we don't have to repeat the same code for integers and floating points.

```
void dispatch_To_issue(int current_cycle)
```

The main goal here is to move an instruction at the head of the instruction fetch queue to an available reservation station. We use the following algorithm:

- Peek at the instruction at the head of the fetch instruction queue
- **Special case**: If the instruction is an unconditional or conditional control instruction, we simply return, because those instructions do not go to any other stage
- Figure out what kind of functional unit the instruction uses
- Check if a reservation station is available. If so, put the instruction in and dequeue it from the fetch instruction queue.
- Check if that instruction has any dependencies by checking if any of its sources matches any instructions in the map table
- Finally, update the map table with the current instruction's output so other instructions can figure out if there are any dependencies

```
void fetch(instruction_trace_t* trace)
void fetch_To_dispatch(instruction_trace_t* trace, int current_cycle)
```

The main goal here is to figure out if there is a free space in the fetch instruction queue so we can fill it with more instructions. We also want to dispatch an instruction if possible. Here are the steps we take to fetch an instruction:

- Check if the fetch instruction queue is full and if there are any more instructions to fetch
- **Special case**: If the instruction is a trap instruction, we want to ignore it
- Keep going until we get a valid instruction to put into the IFQ
- Insert the instruction into the IFQ
- Dispatch an instruction if the instruction fetch queue isn't empty

Explain how you tested the correctness of your code.
We tested the corrected of our code by picking one of our benchmarks, gcc.eio, and performing the tomasulo algorithm on paper as would be happening if it was implemented properly. Due to the tediousness of walking through tomasulo on paper, we only did up to 20 instructions on paper and verified that the output of our program matched our paper work.

Briefly describe the two toughest bugs you had while developing your Tomasulo code.
1. Making sure that instructions weren't null
While dereferencing pointers and looking at their contents, we would often get segmentation faults if we didn't properly check whether or not the pointers were valid. Segmentation faults are often hard to pinpoint, which made it difficult to debug. The solution was to check whether or not an instruction is valid everywhere before we needed to access its contents.
2. Special case of the store instruction not writing to the common data bus
When determining which instructions to put on the common data bus after execution, we weren't considering the special case of the store instruction properly. When we caught the logical error, we had to change other parts of the code so that the CDB tom cycles would be correct for store instructions (0, unchanged) and that we would remove all store instructions that are finished with their functional units.

Include a brief statement of work completed by each partner.
Freddy Chen - Wrote code, wrote parts of report that pertain to explaining how the code works.

Anthony Alayo - Wrote code, verified correctness of the program by doing tomasulo on paper.