Lab Assignment 3: Dynamic Scheduling with Tomasulo

1 Objective

The objective of this assignment is to investigate the impact of Tomasulo's dynamic scheduling algorithm on performance. All work on this assignment is to be done in groups of two.

This assignment will require significantly more code writing compared to the previous assignments. You are encouraged to start early.

2 Problem Statement

In this assignment, you will use the SimpleScalar simulator to measure the total number of cycles it would take to run a benchmark with dynamic scheduling. Specifically, you are asked to simulate an **in-order dispatch & dynamic scheduling Tomasulo's algorithm with no speculation.** This scheme was discussed in class and is also described in Section 3.4.1 of your textbook [1].

Tomasulo's algorithm

You will model Tomasulo's algorithm with the following stages: Dispatch (D), Issue (I), Execution/Memory (EX/MEM), and Writing to the Common Data Bus (CDB). Detailed information for each stage is listed below. Also Table 1 summarizes the configuration parameters for our hardware implementation. The functional units are not pipelined.

Tomasulo Structures	Number	# Entries	Latency
Instruction Fetch Queue (IFQ)	1	10 instructions	-
Integer Reservation Stations (reservINT)	4	-	-
Floating-Point Reservation Stations (reservFP)	2	-	-
Integer Functional Units (fuINT)	2	-	4 cycles
Floating-Point Functional Units (fuFP)	1	-	9 cycles
Common Data Bus (CDB)	1	-	1 cycles

Table 1: Tomasulo Configuration Parameters

1. **Dispatch State (D)**:

   - We effectively merge fetch and dispatch into a single stage. Instructions are fetched and placed in program-order into the `Instruction Fetch Queue (IFQ)`. A new instruction can be fetched every cycle, as long as the instruction queue is not full.
   - A fetched instruction can be **dispatched** immediately, if there is an available reservation station. Structural hazards due to reservation stations are resolved here with stalls.
   - If there are no stalls, a reservation station entry is allocated based on each instruction's opcode. Any RAW dependences are marked in the reservation entry.

   - **Special Cases:**
     (a) Unconditional branches (e.g., jumps, calls) are not issued to the reservation stations. They do not use any functional units; they do not write to the common data bus; and they do not cause control hazards.

(b) Conditional branches are handled like unconditional branches to model the effect of perfect branch prediction. The dispatch cycle of the branch instruction is updated, but these instructions never occupy a reservation or occupy any of the subsequent stages.

(c) We do not have special load/store queues. Memory instructions use the integer functional units and their respective reservation stations.

(d) You need to skip any trap instruction from the instruction trace; i.e., do not insert it in the IFQ. You can identify trap instructions via the `IS_TRAP` macro.

2. **Issue Stage (I)**: An instruction enters the issue stage the cycle after it entered dispatch.

   - Instructions wait at this stage for all RAW hazards to be resolved (i.e., until all source operands are marked as ready in their reservation station).
   - Instructions also wait at this stage if no functional unit is available (structural hazard).

3. **Execution/Memory Stage (EX/MEM)**: An instruction enters execution once all its dependences have been resolved.

   - An instruction executes in the functional unit which matches its reservation station.
   - Memory instructions access memory during the execution stage. Note that we ignore dependences over memory accesses. In other words, loads and stores do not have to wait for each other if they access the same address.
   - Instructions remain in the reservation stations until they complete their execution.
   - Multiple instructions can enter the execute stage in the same cycle.

4. **Writing to the Common Data Bus (CDB)**: Once an instruction completes execution, it broadcasts its results via the Common Data Bus.

   - Stores, conditional/unconditional branches, jumps and calls do not write to the common data bus.
   - There is no forwarding/bypassing support. Register values broadcasted via the common data bus can be read the next cycle.
   - If two instructions compete for a resource (e.g., functional unit, CDB), then the older instruction, in program order, gets priority.

3 Methodology

Create a modified version of the sim-safe simulator to simulate the algorithm discussed above. Collect the "`total number of cycles with tomasulo`" statistic, and use the results to prepare a brief report as described in Section 5 of this handout. Section 3.1 briefly describes how to install and compile the simulator for this assignment, and how to run the benchmarks. Section 3.2 provides coding advice, focusing on the new aspects of the provided simulator. Finally, Section 3.3 provides some debugging tips.

3.1 How to compile and run the simulator

1. You can obtain the simulator source code and set it up in your ugsparc account using the following Unix commands:

   ```
   cd ~/ece552
   cp /cad2/ece552f/simplesim-3.0d-assig3.tgz .
   tar -zxf simplesim-3.0d-assig3.tgz
   cd simplesim-3.0d-assig3
   ```

   This sequence of commands extracts the simulator files into your working directory ∼/ece552/simplesim-3.0d-assig3. These instructions assume that the ece552 directory exists from previous assignments. Remember not to leave the Unix permissions to your code open.

2. You can build the `sim-safe` simulator using the provided `Makefile` by typing:

   ```
   make sim-safe
   ```

   You may safely ignore any warning messages you see during compilation.

3. Because the simulation will be slow in this assignment, you are asked to **collect results only for the first 1,000,000 instructions** of each EIO trace. You can run the benchmarks as follows:

   ```
   sim-safe -max:inst 1000000 /cad2/ece552f/benchmarks/gcc.eio

   sim-safe -max:inst 1000000 /cad2/ece552f/benchmarks/go.eio

   sim-safe -max:inst 1000000 /cad2/ece552f/benchmarks/compress.eio
   ```

4. Because you are not allowed to modify `sim-safe.c`, you can assume that the output of the benchmark will always be correct. You do not have to verify that.

5. CAUTION! When you are redirecting the output of the simulator, use the `-redir:sim` flag of `sim-safe`. Do NOT redirect the output using the pipe character '|' or the redirect character '>' as this may cause variation in the simulated instruction count. To redirect the output of the simulated program, use the `-redir:prog` flag.

3.2 Coding Advice

The goal of the assignment is to find how many cycles it **would** take to run a particular program under the Tomasulo dynamic scheduling algorithm. We will be doing a detailed cycle-by-cycle simulation of this algorithm. That is, we will be keeping track of all the hardware states (reservation stations, functional units, etc.) at every cycle.

You are only required to implement the functions given to you in `tomasulo.c`. We have already inserted a call to the `runTomasulo` function in `sim-safe.c`. **You should not modify** `sim-safe.c`.

1. Look at the sim-safe.c file and notice how we have modified the main while loop with respect to the previous assignments. The simulator now runs the program twice.

- First, it collects a trace of all executed instructions in the `instruction_trace_t` data structure as follows: `put_instr(instruction_trace, &m_instr)`.
- Then, it calls the `runTomasulo` function, which parses the collected instruction trace and returns the total number of cycles.

2. Delve deeper into the `instruction_trace_t` data structure, defined in `instr.h`. Each instruction in the trace is represented by the `instruction_t` data structure which contains a wealth of instruction-related information. For example, it contains the instruction opcode, the source and destination registers, and instruction PC. More importantly, it contains a detailed cycle-breakdown of when this instruction entered each tomasulo stage.

3. Look at the `instr.c` file for some helper functions:

   - `void put_instr(instruction_trace_t* trace, instruction_t* instr)`: Called from sim-safe.c, it adds the `instr` instruction to the `trace` instruction trace.
   - `instruction_t* get_instr(instruction_trace_t* trace, int index)`: Returns a pointer to the instruction at the `index` location from the instruction trace.
   - `static void print_tom_instr(instruction_t* instr)`: Prints detailed timing information for a single instruction.
   - `void print_all_instr(instruction_trace_t* trace, int sim_num_insn)`: Prints the detailed cycle-breakdown of all trace instructions.

4. Now focus on the `tomasulo.c` file. The `runTomasulo` function, called from sim-safe, must return the total number of cycles it would take to run all the instructions in the trace under the specified Tomasulo implementation. You are required to use the collected instruction trace to simulate the tomasulo algorithm.

   - You need to parse the collected instruction trace and assign **appropriate cycle values (i.e., the cycle at which each instruction enters dispatch, issue, execute, or writeback stage)** for each trace instruction. At the end of simulation, **sim_main** calls the aforementioned `print_all_instr` function which prints out these values.
   - If an instruction does not perform a particular stage, its cycle value should be assigned to 0.
   - Remember, **the trace starts by the dispatch of instruction 1 at cycle 1**, i.e., there is no instruction 0 or cycle 0.
   - In tomasulo.c there are specifications for each function you are required to implement. In the beginning of the file, you will also find macro definitions and variable declarations; you are highly encouraged to use them. You are also free to write your own helper functions to make your code more modular.

5. You can (un)comment the call to `print_all_instr` inside `sim-safe.c`, as you see fit, while working. You are encouraged to start working with a small number of instructions in the beginning.

3.3 Debugging Advice

Always try to use good coding practices; modular and well-commented code makes debugging (and marking) easier. Use the provided print functions and macros while coding, to debug your code step-by-step. Just remember to remove them before you submit your code! You can also use `gdb` and `assertions` to quickly identify code bugs.

Using gdb for segmentation faults

If your code crashes due to a segmentation fault, you can use `gdb` to quickly identify the offending code line. You can run it as follows:

```
gdb sim-safe
```

At the gdb command prompt type "run" and the remaining command-line arguments for sim-safe (the ones that crash your simulation). For example,

```
(gdb)run -max:inst 10000 /cad2/ece552f/benchmarks/gcc.eio
```

Your code will run within the debugger and will stop at the segmentation fault, listing the offending code instruction. In the following unrelated example, it is the statement in line 369 from file buggy_code.c. You can then type `bt` to run a backtrace, i.e., get a reversed call stack of the function calls that led to this segfault.

```
Program received signal SIGSEGV, Segmentation fault.
0x0806b04f in random_function1 (cycle=33) at buggy_code.c:369
369                              tmp = array[TYPE]->value[i];

(gdb) bt
#0  0x0806b04f in random_function1 (cycle=33) at buggy_code.c:369
#1  0x08053445 in tmp_main () at sim-unsafe.c:441
#2  0x08053d3b in main (argc=4, argv=0xbffff554, envp=0xbffff568) at main.c:417
```

Assertions

Sometimes while developing you might wonder if a specific scenario is ever possible. You can add an assertion as a sanity check, to verify that this scenario never occurs. For example, the assert statement below will fire, and end your program execution, if pointer `a` is NULL.

```
assert(a != NULL);
```

4 Prelab

The prelab is worth 1/6 of the overall lab mark. Please complete the following steps before coming to the lab:

- Familiarize yourself with all necessary background on dynamic scheduling (textbook, lecture slides, podcast example).
- Answer the following questions:
  1. How can the Tomasulo algorithm extract more `ILP` for a given program compared to a superscalar processor?
  2. What are `reservation stations` used for? List all the fields of a reservation station entry and explain their functionality.
  3. How does Tomasulo's algorithm deal with false-dependences? Provide a brief example.
- Write most of your code before coming to the lab.
- Finally, be prepared to answer any high-level questions about the simulator, your code, and the lab material.

5 Lab Deliverables

At the end of this assignment you should submit the following files using the submitece552f command:

1. **tomasulo.c**: the modified tomasulo algorithm file. Identify all modifications to `tomasulo.c` with the comments `/* ECE552 Assignment 3 - BEGIN CODE */` and `/* ECE552 Assignment 3 - END CODE */`. Any code outside these indicators will NOT be considered during marking. Make sure you have not modified any other files.

2. **report.pdf**: a brief three-pages pdf report (single-spaced with 12-point font size). Make sure your report can be viewed on the ug machines through `xpdf` or `acroread`.

   - Report the "total numbers of cycles with tomasulo" for the first one million instructions of each EIO trace.
   - 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.
   - Explain how you tested the correctness of your code.
   - Briefly describe the two toughest bugs you had while developing your Tomasulo code.
   - Include a brief statement of work completed by each partner.

The submit command should be similar to the following:

```
submitece552f 3 tomasulo.c report.pdf
```

You can view your submitted files via the command:

```
submitece552f -l 3
```

6 Due Date and Marking Scheme

This assignment is due on **Friday October 31, 2014 at 5:00pm**. It is worth **6**% of the total course mark. The pre-lab will constitute 1/6 of the overall lab mark.

An automarker will be used to compile and run your implementation and verify the generated statistics. Therefore your implementation is required to follow some strict rules. To begin with, use the simulator package specifically provided for this assignment as described in Section 3.1.

Things to watch for:

- enclose all your code modifications in the comments
- the only file to modify is tomasulo.c
- do not add additional files, they will not be taken into account. Your simulator should compile with the provided `Makefile` by typing `make sim-safe`
- do NOT leave any debugging statements (e.g., printfs) turned on in your submitted file
- an automatic comparison of your submissions will be done to check for copied code; changing variable names and formatting will not defeat the checker. Copying of code or otherwise cheating on the lab will not be tolerated and will be dealt with in accordance with university policy.

7 Questions

A list of frequently asked questions has been posted in the Lab Material section of CoursePeer. Please refer to this list first to see if your question has already been answered. If you post a question that the TA feels has already been answered in this document, they will refer you to the handout.

Please post clarifications questions on CoursePeer. Please give your questions on CoursePeer descriptive titles to help other students identify if their question has already been asked/answered. Titling your question: "Lab 3 question" is not helpful.

Do NOT post solution code or microbenchmarks code; this constitutes cheating and you will be penalized. If you have a code-specific question send a private message to a TA. Please start early!

References

[1] Dubois, Michel, Murali Annavaram, and Per Stenström. Parallel Computer Organization and Design. 1st ed. Cambridge: Cambridge University Press, 2012.