18-640: Foundations of Computer Architecture

Project 2: Modern Tomasulo's Algorithm and Out-of-Order Execution

Out: October 6th 2015 12.00AM EDT

Due: October 25th 2015 11.59PM EDT

1. Introduction

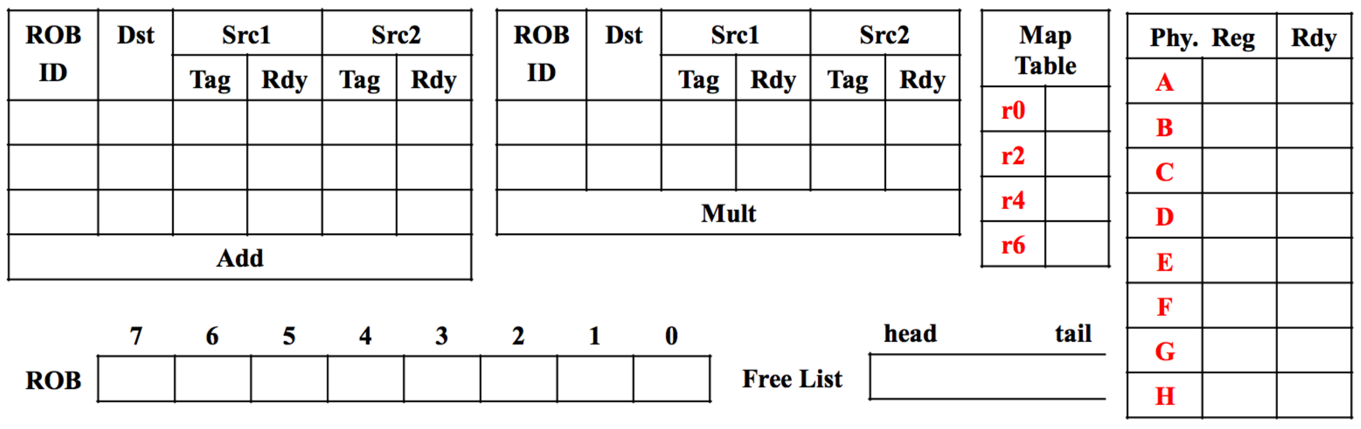
In the first part of the project, you will model and simulate Modernized Tomasulo's Algorithm (MTA). Tomasulo's algorithm dynamically resolves RAW hazards in dataflow order and relies on an early form of register renaming to eliminate WAW and WAR hazards. In the second part, you will be working with Out of Order model in gem5.

Part 1: Tomasulo's Algorithm

2. Tomasulo's Algorithm

Our version of the Tomasulo implementation features an in-order Instruction Queue that feeds into two reservation stations for integer (instead of floating-point) additions and multiplications, respectively. A scheduler at each reservation station identifies instructions with valid source operands and issues them to the functional unit. You will not be modeling the load/store portions of the Tomasulo's algorithm. One major departure from the original algorithm is that the adder and multiplier will have separate completion busses (i.e., instruction scheduling does not need to worry about structural hazards at the CDB).

Refer to SnL book "Modern Processor Design" that contains a thorough treatment of normal Tomasulo's algorithm for your reference (Section 5.2.4 - 5.2.6 - 2013 edition). We will build on this implementation .



On top of the base Tomasulo in SnL, an 8-entry ROB is added (see below). The floating-point register file from Tomasulo is replaced by a rename table and a physical register file. When an instruction is dispatched to a reservation station (RS), an entry in the ROB is allocated at the same time. If the instruction writes to a destination register, a rename register is also allocated at this time. MTA's RS no longer stores the values of the operands; the operand values are read from the physical register file for execution. The **Tag** fields hold the physical names of the source registers and the new **Ready** fields indicate whether the operands are available. Two new fields are added to each RS entry to record the instruction's ROB index and the physical name of the destination register.

3. Requirements

The simulator in Project 2 models a processor with 32 architectural registers, one fully-pipelined adder and one fully-pipelined multiplier. Each functional unit has its own reservation station. The number of reservation station slots are specified by the configuration file.

Configuration File Format

The default configuration file will be named "config.default", and will have five fields in the following format:

```
[number of add reservation station slots]
[number of multiply reservation station slots]
[maximum issue rate]
[adder latency]
[multiplier latency]
```

Here is one possible example configuration file:

```
3       // 3 reservation slots for the adder
5       // 5 reservation slots for the multiplier
2       // two instructions per cycle
2       // adder latency, fully pipelined
4       // multiplier latency, fully pipelined
```

This example configuration allows three reservation station slots for the adder and five reservation station slots for the multiplier. The last 3 parameters control our portions of the simulator and will be explained later. Your portion of the simulator should not depend on them. We will be testing your simulator with a number of different configurations.

Trace File Format

The simulator will be driven from an instruction trace file. Each line of the trace file will have four values separated by spaces in the following format:

[Instruction type] [destination register] [operand 1] [operand 2]

An example trace file (with all four required instruction types) is shown below:

```
mul r21 r3 r4          // r21 <- r3 * r4
add r0 r20 r10         // r0 <- r20 + r10
mul r5 r13 289         // r5 <- r13 * 289
add r11 r12 43         // r11 <- r12 + 43
```

Note that if an "r" is prefixed to an operand specifier, then it is a register operand. Operand specifiers without the prefix are immediates (can only be the second operand). For a register file with 32 entries, indexes range from r0 to r31. (***Note*** r0 is not special.)

Your part of the simulator

The purpose of this project is to help you understand how scheduling is achieved with Tomasulo's algorithm; therefore, to mitigate unnecessary details we have provided you with the simulator driver tomasulo_sim.o. The simulator driver is responsible for fetching instructions by reading the trace file from stdin, controlling the number of instructions issued per cycle, keeping track of the functional unit execution latencies, tracking the number of cycles in the simulation, and printing the final cycle count and register values.

Our simulator driver models the functional units and the instruction queue. Your task is to implement the 2 reservation stations, the physical register file, map table and reorder buffer. In addition, you need to implement the functions to

1. issue instructions to the reservation station
2. select readied instructions for execution
3. process broadcasted completed results.

To maintain a consistent interface between the simulator driver and your code, the following structs and enumerated types have been defined in tomasulo.h:

```
//used to choose a functional unit's reservation station
typedef enum { add = 0, mult = 1 } mathOp;
typedef enum { addImm = 0, addReg = 1, multImm = 2, multReg = 3 } instType;

typedef struct {
instType instructionType;
int dest;
int op1;
int op2;
} instruction_t;  //instruction that needs to be issued
```

```
typedef struct {
int tag;
int op1;
int op2;
} executeRequest_t;   //request sent to functional unit for processing

typedef struct {
int tag;
int value;
} writeResult_t;  //returned value from functional unit
```

To complete this assignment, you must implement the following five functions defined in tomasulo.h:

```
void initTomasulo()
int issue(instruction_t *theInstruction)
int execute(mathOp mathOpType, executeRequest_t *executeRequest)
void writeResult(writeResult_t  *theResult)
int checkDone(int registerImage[NUM_REGISTERS])  // NUM_REGISTERS = 32
```

Do not free any of the pointers passed into the function.

initTomasulo( ) is called before any processing begins for you to initialize your data structures

issue( ) is called each time the simulation driver attempts to issue an instruction. Return "1" if the instruction is accepted, and "0" otherwise.

execute( ) is called to query the reservation station for readied instructions. If there is an instruction in the "mathOp" reservation station that is ready for execution, set the *tag*, *op1*, and *op2* fields in the executeRequest struct accordingly and return "1"; otherwise return "0". If there are multiple instructions that are ready to execute, choose the one that has been in the reservation station the longest.

writeResult( ) is called when an instruction's result has been completed. The writeResult_t contains the same tag as the executeRequest_t that original spawned the execution.

checkDone( ) is called every cycle after all the instructions in the trace file have been issued. The return value should be "1" if the simulation has ended, and "0" otherwise. If the simulation has completed (i.e., no transient instructions in your part of the simulator), write the final values of the register file into the registerImage array, and we will print the values out.

*****Set the number of register file and reorder buffer slots to 64.*****

Our part of the simulator

The starter code can be found in the `/afs/ece/class/ece640/project/project2/release/` directory. For more details please read tomasulo.h.
This pseudo code gives you an idea of how the simulation driver will be calling your functions:

```
      initTomasulo();

      …do some initializing

      …read first instruction from stdin and place in instructionRead
while (true) {

           // Completion stage
           for each execution that completes this cycle {
                writeResult (&(completedExecution));
           }

           // Execution stage
           for each functional unit {
                validExecute =
                execute(functionalUnitType, &(executionRequest));

                if (validExecute)  {
                     … Start execution of the request
                }
           }

           // Issue stage
           while (trace file has instructions) &&
           (IssueRequestAttemptsThisCycle < maxIssueRequestPerCycle) {
                issueValid = issue(instructionRead);

                if (issueValid == 1) {
                     …read instruction from stdin and update
instructionRead
                } IssueRequestAttemptsThisCycle++;
           }


           // Continue driving the loop even if there are no more
    // Instructions to drain the pipeline in your implementation.

           if (trace file has no more instructions) {
                if (checkDone(registerImage) == 1) {
                print registerImage
                print number of cycles return;
                }
           }
```

```
        }
```

Each iteration of the outer-most loop corresponds to “1 cycle”. Thus, this driver code roughly corresponds to an implementation that issues “maxIssueRequestPerCycle” instructions per cycle, starts one addition and one multiplication per cycle, and completes up to 2 instructions per cycle.

Note that the driver loops calls “complete”, “execute”, and “issue” in reverse order. This is to simulate the effect of a pipeline. For example, on the first iteration of the loop (cycle 0), the completion and the execution stage has no work to do. However, the issue stage will issue an instruction into a reservation station. On the second iteration of the loop (cycle 1), the execution stage proceeds and begins executing in a functional unit. Finally, when the execution has completed some number of cycles later, the completion stage sends the results of the functional unit to your reservation stations and register file.

Note, as specified, we should be able to call any of your functions in any order at any time. If done correctly, none of your functions should be dependent on issue bandwidth, execution bandwidth/latency and completion bandwidth.
Test cases can be found in `trace` folder.

We expect to run your program with the following commands:

```
        make
        ./tomasulo_sim  <  <name of trace file.txt>
```

Report:
The report must address the following questions.

1. Explain how Tomasulo's algorithm avoids WAW, RAW, WAR hazards. If we made the Instruction issue out-of-order, do any of these hazards now exist?

2. How did you generate tags in your implementation? Is this how you would do it in hardware?

3. Try increasing the number of slots in your reservation stations and the maximum issue rate. Why does performance improve even though only 1 instruction can begin execution per cycle for a single reservation station? When is it better to increase the maximum instructions issued per cycle? When is it better to increase the number of reservations station slots?

4. Although your implementation only had a single reservation station per functional unit, it is possible to have multiple reservation stations with same functionality (e.g., 4 separate reservation stations, each with a multiplier functional unit) to increase overall throughput. How does one choose the right number of reservation stations?

Part 2: Out-of-Order Execution

4. Instrumentation in gem5

You will be using Full-System Simulation mode for Alpha architecture and PARSEC Benchmarks will be used for testing. Details about PARSEC Benchmark at http://parsec.cs.princeton.edu/

Setup:

- Go to your gem5 folder.
- Build gem5: `scons build/ALPHA/gem5.opt`
- Download parsec.tar.7z from `/afs/ece/class/ece640/project/project2/`
- Extract:
  `7z x parsec.tar.7z; mkdir parsec; tar -C parsec/ xvf parsec.tar`
- Modify two files:
  - In `configs/common/Syspaths.py`: [Line 53]
    `path = [ '/dist/m5/system', '/path/to/your/parsec_folder' ]`
  - In `configs/common/Benmarks.py`: (for Alpha) [Line 55]
    `return env.get('LINUX_IMAGE', disk('linux-parsec-2-1-m5.img'))`

- Download scripts to run benchmarks in FS mode:
  `runscript_1.rcS` and `runscript_2.rcS` from `/afs/ece/class/ece640/project/project2/`

Example Command to run:

```
  build/ALPHA/gem5.opt  configs/example/ruby_fs.py  --script=runscript_1.rcS
-- cpu-type=detailed  --caches
```

/* Simulation takes time, have patience */
You can telnet localhost 3456 to monitor status.

You should see 4 separate sets of statistics at the end of simulation corresponding to: (1) the start of simulation until the dumpin the runscript, (2) the beginning of the benchmark up to the beginning of the ROI in the benchmark, (3) the benchmark ROI, and (4) from the end of the ROI to when the simulation exits on '/sbin/m5_exit'.

Task:

In this section you will have to find buffer usage by adding counters in gem5. Specifically, you may add counters to any of the files in `src/cpu/o3/` to extract number of used Instruction Queue, Load Queue, Store Queue and ROB slots every cycle and report the results in this format:
For example, if in a particular cycle, number of used buffers in IQ is 10, and we know that total buffer size is 64, therefore, utilization of buffer is 10/64, which is less than 25%, so you should increment *InstQueueB1* counter.

Counters to add to find utilization of buffers

	Bucket 1 [0 - 25) %	Bucket 2 [25 - 50) %	Bucket 3 [50 - 75) %	Bucket 4 [75 - 100] %
Instruction Queue	InstQueueB1	InstQueueB2	InstQueueB3	InstQueueB4
Load Queue	LoadQueueB1	LoadQueueB2	LoadQueueB3	LoadQueueB4
Store Queue	StoreQueueB1	StoreQueueB2	StoreQueueB3	StoreQueueB4
ROB	ROBufferB1	ROBufferB2	ROBufferB3	ROBufferB4

For reporting in the table, find the InstQueueB1 value at the end of simulation and divide by total number of cycles to find percentage. Please include the results in your report in the tabular format. Load and Store Queue results can be combined.

Find above utilization for Blackscholes and x264 benchmark i.e. use runscript_1.rcS and runscript_2.rcS one by one in the command. Report with default buffer sizes: IQ: 64; LQ: 32; SQ: 32; ROB: 192

For example stats.txt file should have entries like:

```
*. InstQueueB1 100567          #Number of cycles in B1 of Inst Queue
*. InstQueueB2 123579          #Number of cycles in B2 of Inst Queue
```

In the end, for runscript_1.rcS stats.txt should be renamed to stat_1 and config.ini to config_1 and similarly, for runscript_2.rcS stats.txt should be renamed to stat_2 and config.ini to config_2. Please adhere to the format.

Report:

- Analyze the results you collected. Did you find anything interesting?
- If you were asked to report number of cycles an Instruction spends at various stages of pipeline, how would you go about it? What result do you expect?

5. Grading

Your project will be graded according to the following:

20%	Tomasulo model builds and runs without crashing
20%	Works correctly on a small test case
20%	Works correctly on full test cases
20%	gem5 Instrumentation
20%	Project Report

20% of your grade only requires your simulator to build and run without problems. 20% of your grade is determined if a small test case runs correctly and generates the correct final number of cycles and architectural state (register values). To earn the next 20% of your grade, we will be checking your simulator against expected number of cycles and architectural state using 5 selected instruction traces using varying tomasulo configurations and traces (e.g., changing the number of reservation slots). We have provided **sample** traces and configurations to assist your development. gem5 Instrumentation accounts for 20% as well. Finally, the remaining 20% of your grade will depend on the quality of your project report.

6. Bonus (10% bonus)

Optimized your Tomasulo's implementation to consume lesser cycles than the TA's implementation to score a 10% bonus. Correctness shouldn't be affected for achievement of lesser number of cycles. Test cases can be found in `trace` folder.

Report:

Explain us how did you achieve it? What was the improvement in the number of cycles compared to original implementation? Attach screenshots to better illustrate.

7. Hand-in Details:

You will need to hand-in `tomasulo.c, tomasulo.h, stat_1, config_1, stat_2, config_2` and Report (in PDF or ASCII) to `/afs/ece.cmu.edu/class/ece640/submission/group[groupid]/project2/`.

Also, hand-in the source-code:
`src/cpu/o3/*` to `/afs/ece.cmu.edu/class/ece640/submission/group[groupid]/project2/`.

`——————————————-***Start early and All the best!***——————————————`