CMU 18-741: Advanced Computer Architecture

Handout 7: Tomasulo's Algorithm

**** Due 10am, 11/3/2005 ****

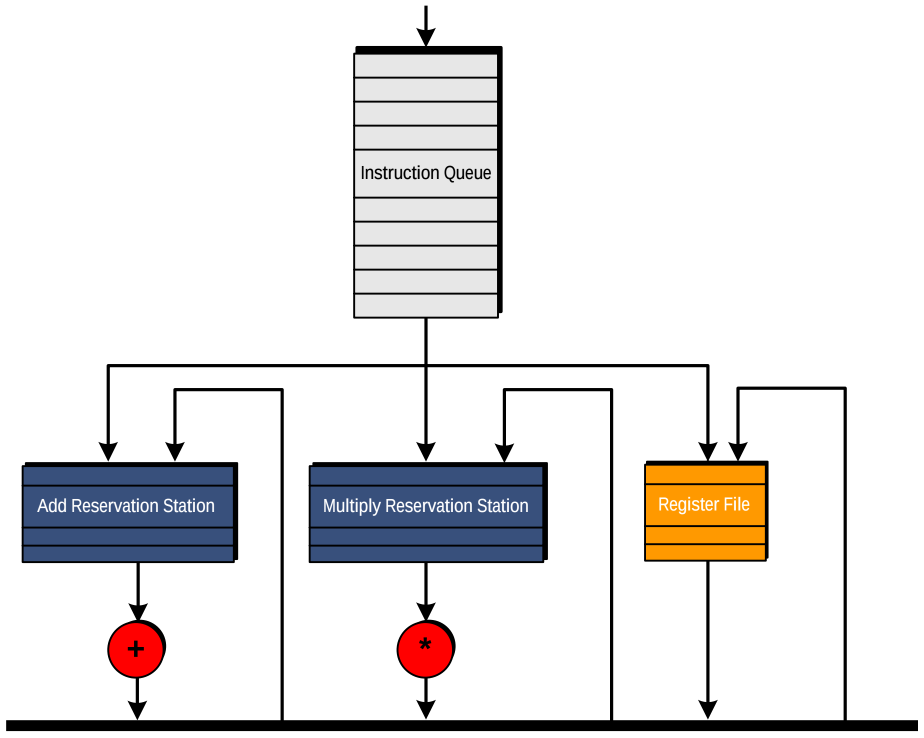
1. Introduction

Modern processors rely on out-of-order execution of instructions to extract instruction-level parallelism from sequential programs. In this project, you will model and simulate Tomasulo's algorithm, which was originally implemented in the IBM 360/91 floating-point unit. 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.

This project must be completed in groups of 2 or 3 students. After forming a group, one member should email the TAs with the names and email addresses of all members by October 20. Use the subject heading "Project 3 Group".

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. The figure below illustrates the high-level organization of Tomasulo's algorithm that you will be implementing. 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). Page 184 in Hennessy and Patterson contains a description of Tomasulo's algorithm. Page 246 in "Modern Processor Design" by John Shen and Mikko Lipasti also contains a thorough treatment of Tomasulo's algorithm for your reference.



3. Requirements

The simulator in Project 3 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 r3 r13 r23         //r3  <- r13 * r23
add r20 r25 r29        //r20 <- r25 + r29
mul r13 r20 793        //r13 <- r20 * 793
add r2 r31 200         //r2  <- r31 + 200
```

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 (see figure on the previous page). Your task is to implement the 2 reservation stations and the register file. In addition, you need to implement the functions to 1. issue instructions to the reservation station, 2. select readied instructions for execution, and 3. process broadcasted completed results. You may use any language as long as you can provide a wrapper that will interface to our object code developed for C, but you must use the provided tomasulo_sim.o.

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.

Our part of the simulator

The starter code can be found in the `/afs/ece/class/ece741/project3/startercodeC/` directory. For more details please read tomasulo.h.

This psuedocode 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 roughly 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.

We expect to run your program with the following commands:

```
make tomasulo
zcat tracefile.gz | ./tomasulo_sim
```

Please include a README file with your code, which should contain the names of the group members, and e-mail addresses at which you can be contacted in case there are any problems. If there are any other comments about the project (e.g. project does not work), please include it in the README file. A sample README file can be found at `/afs/ece/class/ece741/project3/`.

4. Grading

Your project will be graded according to the following:

40%	Tomasulo model builds and runs without crashing
20%	Works correctly on a small test case
20%	Works correctly on full test cases
20%	Design review answers
15%	Extra Credit

40% 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 will provide **sample** traces and configurations to assist your development. Finally, the remaining 20% of your grade will depend on your design review answers.

5. Extra Credit

In Project 3, you have the opportunity to earn an extra 15% by, in addition to Tomasulo's algorithm, implementing register renaming as described in the MIPS R10000 paper [Yeager, 1996]. To do this, you must implement a minimal reorder buffer, a register map table, a physical register file, a ready list and a free list. This register-renamed implementation should operate against the same set of interface functions as in the Tomasulo portion of this project. The extra credit work cannot be done in lieu of the Tomasulo portion of the project. **If you do choose to implement this, please include a short description of your implementation in the design review. This section will be graded by inspection of your implementation. You may keep variables such as the number of ROB entries or number of physical registers parameterized in a header file.**

6. Design Review

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?

5. What changes would need to be made in order to support branches, loads and stores?

7. Submission

The final submission will be done electronically. Copy all relevant files to the directory in `/afs/ece/class/ece741/project3/submission/your_email` which matches the email of the student who e-mailed the group names. The directory is unlocked until the deadline so you can make changes until then if you submit early.