Georgia Institute of Technology

Schools of Computer Science and Electrical &Computer Engineering

CS 4290/6290, ECE 4100/6100: Spring 2015

Lab 3: Tomasulo Algorithm for a Superscalar Pipelined Processor

Due: Before 11:55 PM on 6 March 2015

VERSION: 1.0

Rules

1. **All students (CS 4290/6290, ECE 4100/6100) must work *alone*.**
2. **Sharing of code between students is viewed as cheating and will receive appropriate action in accordance with the University's policies. It is acceptable for you to compare your results with other students to help debug your program.**

Project Description:

In this project, you will construct a simulator for an out-of-order superscalar processor that uses the Tomasulo algorithm and fetches *F* instructions per cycle. Then you will use the simulator to find the appropriate number of function units, fetch rate and result buses for each benchmark.

Specification of simulator:

1. FUNCTION UNIT MIX

Function unit type	Number of units*	Latency
0	*parameter: k0*	1
1	*parameter: k1*	1
2	*parameter: k2*	1

Notes:

- The number of function units is a parameter of the simulation and should be adjustable along the range of 1 to 3 units each (see experiments below).

2. INPUT TRACE

The input traces will be given in the form of the **proc_inst_t** structure:

Although we are using the same traces as previous labs, the instruction format is slightly different. We have already implemented the transformation from the previous format to the proc_inst_t format in the **read_instruction** function.

<address> <function unit type> <dest reg #> <src1 reg #> <src2 reg#>
<address> <function unit type> <dest reg #> <src1 reg #> <src2 reg#>
...

Where

<address> is the address of the instruction (in hex)
<function unit type> is either "0", "1" or "2"
<dest reg #>, <src1 reg#> and <src2 reg #> are integers in the range [0..31]
note: if any reg # is **-1**, then there is no register for that part of the instruction (e.g., a branch instruction has **-1** for its <dest reg #>)

For example:

ab120024 0 1 2 3
ab120028 1 4 1 3
ab12002c 2 -1 4 7

Means:

"operation type 0" R1, R2, R3
"operation type 1" R4, R1, R3
"operation type 2" -, R4, R7   *no destination register!*

3. PIPELINE TIMING

Assume the following pipeline structure:

Stage	*Name*	*Number of cycles per instruction*
1	Instruction Fetch/Decode	1

2	Dispatch	Variable, depends on resource conflicts
3	Scheduling	Variable, depends on data dependencies
4	Execute	Depends on function unit type, see table above
5	State update	Variable, depends on data dependencies (see notes 7 and 8 below)

Details:

1. You should explicitly model the dispatch and scheduling queues
2. The size of the dispatch queue is unlimited.
3. The size of the scheduling queue is 2*(number of k0 function units + number of k1 function units + number of k2 function units)
4. If there are multiple independent instructions ready to fire during the same cycle in the scheduling queue, service them in tag order (i.e., lowest tag value to highest). (This will guarantee that your results can match ours.)
5. A fired instruction remains in the scheduling queue until it completes.
6. The fire rate (issue rate) is only limited by the number of available FUs.
7. **There are R result buses (also called Common Data Buses, or CDBs). If an instruction wants to retire but all result buses are used, it must wait an additional cycle. The function unit is not freed in this case. Hence a subsequent instruction might stall because the function unit is busy. The function unit is freed only when the result is put onto a result bus.**
8. Assume that instructions retire in the same order that they complete. Instruction retirement is unconstrained (imprecise interrupts are possible) and multiple instructions can retire in the same cycle.

3.1 FETCH/DECODE UNIT

The fetch/decode rate is *F* instructions per cycle. Each cycle, the Fetch/Decode unit can always supply *F* instructions to the Dispatch unit. Since the dispatch queue length is unlimited, there is room for these instructions in the dispatch queue. There is no need to implement branch prediction. The traces contain only the true path.

3.2 WHEN TO UPDATE THE CLOCK

Note that the actual hardware has the following structure:

- *Instruction Fetch/Decode*
  PIPELINE REGISTER

- *Dispatch*
  PIPELINE REGISTER
- *Scheduling*
  PIPELINE REGISTER
- *Execute*
  PIPELINE REGISTER
- *State update*

Instruction movement only happens when the latches are clocked, which occurs at the rising edge of each clock cycle. You must simulate the same behavior of the pipeline latches, even if you do not model the actual latches. For example, if an instruction is ready to move from scheduling to execute, the motion only takes effect at the beginning of the next clock cycle.

Note that the latching is not strict between the dispatch unit and the scheduling unit, since both units use the scheduling queues. For example, if the dispatch unit inserts an instruction into one of the scheduling queues during clock cycle *J*, that instruction must spend *at least* cycle *J*+1 in the scheduling unit.

In addition, assume each clock cycle is divided into two half cycles **(you do not need to explicitly model this, but please make sure your simulator follows this ordering of events)**:

Cycle half	*Action*
first half	The register file is written via a result bus
	Any independent instruction in scheduling queue is marked as ready to fire
	The dispatch unit reserves slots in the scheduling queues
second half	The register file is read by Dispatch
	Scheduling queues are updated via a result bus
	The state update unit deletes completed instructions from scheduling queue

3.3 OPERATION OF THE DISPATCH QUEUE

Note that the dispatch queue is scanned from head to tail (in program order). When an instruction is inserted into the scheduling queue, it is deleted from the dispatch queue.

3.4 TAG GENERATION

Tags are generated sequentially for every instruction, beginning with 0. The first instruction in a trace will use a tag value of 0, the second will use 1, etc. The traces are sufficiently short so that you should not need to reuse tags. In other words, the tag number is monotonically increasing.

3.5 BUS ARBITRATION

There are **R result buses** (common data buses), which means that up to R instructions that complete in the same cycle may update the schedule queues and register file in the same cycle. Unless we agree on some ordering of this updating, multiple different (and valid) machine states are possible. This will complicate validation considerably. Therefore, assume the order of update is the same as the order of the tag values.

Example:

Tag values of instructions waiting to complete: 100, 102, 110, 104 where R = 3

Action:

- The instruction with the tag value = 100 updates the schedule queue/register file, then instruction 102, then 104, *all of these happen in parallel, at the same time, and at* ***the beginning of the next cycle****!*
- On the next cycle the instruction with the tag value = 110 updates the schedule queue/register file

3.6 STATISTICS (OUTPUT)

The simulator outputs the following statistics after completion of the experimental run:

1. Total number of instructions in the trace
2. Total simulated run time for the input: the run time is the total number of cycles from when the first instruction entered the instruction fetch/decode unit until when the last instruction completed.
3. Average number of instructions completed per cycle (IPC)
4. Maximum and Average dispatch queue size

4. EXPERIMENTS

4.1 COMMAND-LINE PARAMETERS

Your project should include a Makefile which builds binary in your project's root directory named *procsim*. The program should run from this root directory as:

```
./procsim –r R –f F –j J –k K –l L –b B –e E –i trace_file > output_file
```

The command line parameters are as follows:

• R – Result buses

• F – Fetch rate (instructions per cycle)

• J – Number of k0 function units

• K – Number of k1 function units

• L – Number of k2 function units

• B – Begin dumping pipeline stats from instruction B (0 means no dumping)

• E – End dumping pipeline stats on instruction E (If B is 0, E should be 0)

• trace_file – Path name to the trace file

• output_file – Path name to the output file

Traces will be provided on T-Square.

4.2 VALIDATION

Your simulator must match the validation output that we will place on-line.

4.3 RUNS

For each trace on T-Square, you must run eight configurations and report the best performing configuration based in terms of IPC/(k0+k1+k2+R+F). This metric is a proxy for performance per unit area.

Experiment	k0	k1	k2	R	F
1	2	1	2	3	4
2	2	1	2	3	6
3	2	1	2	5	4
4	2	1	2	5	6
5	1	2	1	4	4
6	1	2	1	4	8
7	1	2	1	2	4
8	1	2	1	2	8

4.4 What to turn in

1. The output and statistics for the validation run to show your simulator matches ours.

2. The experimental results as described above. (As before, *there are multiple answers for each trace file, so I will know which students "collaborated" inappropriately!*)
3. The commented source code for the simulator program.

4.5 Hints

- Work out by hand what should occur in each unit at each cycle for an example set of instructions (e.g., the first 10 instructions in one of the traces). **Do this *before* you begin writing your program!**
- The algorithm in the notes is not intended to be implemented in C/C++/Java. Each "step" in the algorithm represents activity that occurs *in parallel* with other steps (due to the pipelining of the machine). Therefore, you will run into difficulty if you translate the algorithm from the notes to C/C++/Java directly.
- Start early: this is a difficult project.
- Keep a counter for each line in the trace file. Use this for the Tomasulo tags.
- Make each pipeline stage into a function (method)
- Execute the procedures in *reverse order* from the flow of instructions (e.g., execute the state update procedure, then the procedure for the second stage of execution, then the procedure for the first stage of execution, etc.)
- Add a "debug" mode to your simulator that will print out what occurs during each simulated execution cycle.
- Check the webpage frequently for updates and notes. There will be a lot of clarifications required to get everyone matching our output.

4.6 Grading

0%	You do not hand in anything by the due date
+50%	Your simulator doesn't run, does not work, but you hand in *significant* commented code
+25%	Your simulator matches the validation output posted on the website
+25%	You ran all experiments outlined above (your simulator ***must be*** validated first!)

Undergraduate students need to only run the first four experiments. If the students all the eight experiments, they will receive 2 extra points on top of the total 10 points of this lab assignment.

REFERENCE MACHINE:

CS STUDENTS: We will use **shuttle3.cc.gatech.edu** as the reference machine for this course. (https://support.cc.gatech.edu/facilities/general-access-servers)

ECE STUDENTS: We will use **ecelinsrv7.ece.gatech.edu** as the reference machine for this course. (http://www.ece-help.gatech.edu/labs/unix/names.html).

Before submitting your code ensure that your code compiles on this machine, and generates the desired output (without any extra printf statements). Please follow the submission instructions. If you do not follow the submission file names, you will not receive the full credit.

NOTE: It is impractical for us to support other platforms such as Mac, Windows, Ubuntu etc.