Georgia Institute of Technology

Schools of Computer Science and Electrical &Computer Engineering

CS 4290/6290, ECE 4100/6100: Spring 2016 (Conte)

Project 2: Tomasulo Algorithm Pipelined Processor

Due Dates:

Checkpoint 1: Before 11:55 PM on March 31 2016

Checkpoint 2: Before 11:55 PM on April 8 2016

Checkpoint 3: Before 11:55 PM on April 15 2016

VERSION: 3 (See changelog at end)

Rules

The rules for project 2 are the same as project 1:

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 University policy.
3. It is acceptable for you to compare your results with other students to help debug your program. It is not acceptable to collaborate on the simulator design or the final experiments.
4. You should do all your work in the C or C++ programming language, and should be written according to the C99 or C++11 standards, using only the standard libraries.
5. The project may be updated if errors are discovered. It is your responsibility to check the website often and download new versions of this project description as they become available.
6. A Makefile with the frontend will be given to you; you will only need to fill in the empty functions and any additional subroutines you will be using. You will also need to fill in the statistics structure that will be used to output the results.

Project Description

In this project, you will complete the following:

- **Checkpoint 1:** Construct a simulator for an out-of-order superscalar processor that uses the Tomasulo algorithm and fetches *F* instructions per cycle.

- **Checkpoint 2:** Enhance the simulator to maintain consistent state in the presence of exceptions with two separate schemes: 1. ROB with bypass and 2. Checkpoint repair.
- **Checkpoint 3:** Use the simulator to find the appropriate number of function units, fetch rate and result buses for each benchmark.

1. Checkpoint 1

1.1. Directory Description

The procsim_cpp.tar.gz package contains:

1. **Makefile**: compiles your code
2. **procsim_driver.cpp**: contains the main() method to run the simulator. **Do not edit this file.**
3. **procsim.hpp**: Used for defining structures and method declarations. **You may edit this as you please.**
4. **procsim.cpp**: Where all your methods should go. **You may edit this as you please.**
5. **Traces folder**: contains the traces to pass to the simulator (described below).

Note: procsim_c.tar.gz contains equivalent files.

1.2. 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 < trace_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
- trace_file – Path name to the trace file

1.3. Input Trace Format

The input traces will be given in the form:

<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..127]

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!*

As mentioned in 1.7, instructions of type -1 are executed in the type 1 function units.

1.4. Pipeline Stages

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	1

5	State update	Variable, depends on data dependencies (see notes below)

1.5. The Dispatch Queue

- The fetch/decode rate is *F* instructions per cycle. Each cycle, the Fetch/Decode unit can always supply *F* instructions to the dispatch queue.
- The dispatch queue length is unlimited, so there is always room for instructions in the dispatch queue.
- 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.
- 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.

1.6. The Scheduling Queue

- The size of the scheduling queue is 2*(number of k0 function units + number of k1 function units + number of k2 function units)
- 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.)
- A fired instruction remains in the scheduling queue until it completes.
- The fire/issue rate is only limited by the number of available FUs.

1.7. The Function Units

Function Unit Type	**Number of Units**	**Latency**
0	*parameter: k0*	1
1	*parameter: k1*	1
2	*parameter: k2*	1

- The number of function units is a parameter of the simulation and should be adjustable along the range of 1 to 3 units each.
- **Instructions of type -1 are executed in the type 1 function units.**

1.8. Result Buses

- There are **R result buses** (also called common data buses, or CDBs). This means that up to R instructions that complete in the same cycle may update the schedule queues and register file in the same cycle.
- 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.
- The result buses do not have registers or latches. This means that broadcasts do not persist between clock cycles.

The order of result bus broadcasts is as follows:

- Within any given cycle, earlier tags get priority over later tags.
- Earlier cycles with pending broadcasts get priority over later cycles.

For example:

Pending instruction tag	…	7	6	8	5	12	…
Pending since cycle	…	5	5	5	6	6	…

Where R = 2

Action:

- Instructions with tags 6 and 7 will be broadcasted in parallel on the result buses in cycle N.
- Instructions with tags 5 and 8 will be broadcasted in parallel on the result buses in cycle N+1.
- Instruction with tag 12 will be broadcasted in parallel on the result buses in cycle N+2.

1.9. The State Update Unit

- Assume that instructions retire in the same order that they complete. Instruction retirement is unconstrained (imprecise interrupts are possible (for checkpoint 1)).

1.10. Clock Propagation

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.

Each stage of the pipeline can be divided into 2 cycle halves. Assume the following ordering of cycle halves **(you do not need to explicitly model this, but please make sure your simulator follows this ordering of events)**:

Cycle Portion	**Action**
1	Mark completed instructions as ~~completed/~~ retired
2	Broadcast results on the result bus and mark instruction as completed
3	The register file is written via a result bus
4	Any independent instruction in the scheduling queue is marked to fire and marked as issued
5	Scheduling queues are updated via a result bus
6	The dispatch unit reserves slots in the scheduling queues
7	The dispatch unit reads the register file

8	The state update unit deletes ~~completed~~ retired instructions from the scheduling queue
9	Fetch instructions
10	Program counter updated

Note: Not all events are dependent on each other, and thus it is possible to have a different order of events and still achieve correct output. However, following this order, you should be guaranteed correctness.

1.11. Output

For each trace, the output contains 2 files:

1. An **output** file, which contains:
    a. The processor settings
    b. A record of when each instruction was in each stage
    c. The processor statistics

Correctness of your output is required for validation. Your simulator should output results to the terminal (stdout) and it should match the validated output on T-Square. See 1.11.2 for details.

2. A **log** file, which contains the cycle-by-cycle behavior of the machine. This file is **not** required for validation. This is simply there to help debug your code. See 1.11.1 for details.

1.11.1. Log File Details

The log files are meant to help you debug and in particular to verify your algorithm if you're working out a few instructions on paper first. You do **not** need to match the log file with your simulator.

Log File Stage	**Represents the Cycle in Which**
FETCHED	Fetch stage pushes instruction to dispatch queue.

DISPATCHED	Dispatch stage pushes instruction to scheduler.
SCHEDULED	Schedule stage pushes the instruction to FU.
EXECUTED	The result of instruction has been computed and can be put onto the CDB if it is free.
STATE UPDATE	The instruction is deleted from the scheduling queue.

1.11.2. Output File Details

The way to map FETCH, DISP, SCHED, EXEC and STATE in the outputs to messages in the logs is:

Output File Stage	**Represents the Cycle in Which**
FETCH	The instruction is fetched and put into the dispatch queue (same as FETCHED).
DISP	(The fetch stage push an instruction into dispatch queue) + 1 (FETCHED + 1). You can also think of it as "The first cycle that the instruction starts out in the dispatch queue”.
SCHED	(Dispatch stage pushes instruction to scheduler) + 1 (DISPATCHED + 1). You can also think of it as "The first cycle that the instruction starts out in the schedule queue".
EXEC	(Schedule stage pushes instruction to FU) + 1 (SCHEDULED + 1). You can also think of it as "The first cycle that the instruction starts out in a FU".
STATE	Instruction is deleted from the scheduling queue (same as STATE UPDATE).

1.11.3. Statistics

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

1. Total number of instructions in the trace
2. Average dispatch queue size
3. Maximum dispatch queue size
4. Average number of instructions fired per cycle
5. Average number of instructions retired per cycle (IPC)
6. Total run time (cycles)

Your simulator should update the p_stats object to capture the statistics above.

1.12. Validation

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

1.13. Submission Instructions for Checkpoint 1

Submit the following files in a single compressed folder:

1. Makefile
2. procsim.cpp
3. procsim.hpp
4. procsim_driver.cpp

- Name the folder p2_c1_<your prism id>.tar.gz, for example, p2_c1_prabbat3.tar.gz.
- Please compress the folder using the tar.gz compression format only.
- When run, your simulator **must** write the output to the terminal (stdout) with identical formatting to the output files on T-Square.
- However, do **not** include the traces, logs, and outputs in your submission. We will run your code and generate the output ourselves.

1.14. Grading for Checkpoint 1

Checkpoint 1 is worth 50 points:

- 0% You do not hand in anything by the due date *or* you hand in someone else's simulator
- +50% Your simulator matches the checkpoint 1 validation output **exactly.**

1.15. Hints

- **Start early: this is a difficult project.**
- 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.
- Keep a counter for each line in the trace file. Use this for the Tomasulo tags.
- Make each pipeline stage into a function (method).
- Your simulator should continue running until all instructions have been retired. You may assume all instructions have retired when the number of instructions read equals the number of instructions retired.
- Add a "debug" mode to your simulator that will print out what occurs during each simulated execution cycle. The debug mode should match the style of our example verification output.
- Check T-Square and Piazza frequently for updates and notes. There will be a lot of clarifications required to get everyone matching our output.

1.16. Changelog

All changes are marked in red.

3/13/16 1:00 PM [Paul]:

- `procsim.c[pp]`:
  - Comment for `setup_proc()` said r = ROB size. Should be r = # result buses.
- Project description:
  - 1.11, 1.13: Output results to terminal (stdout), not to result.output.
  - 1.3: Clarified -1 type instructions.

3/15/16 7:00 PM [Paul]:

- Project description:
  - 1.8: Added note about result buses.
  - 1.10: Clarified ordering.

3/18/16 1:30 PM [Paul]:

- Project description:
  - 1.7: There are 1 to 3 function units of a given type, not 1 to 2.
  - 1.8: Updated result bus priority scheme.