Project 3: Tomasulo Algorithm Pipelined

1 Overview

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. You will then use the simulator to find the appropriate number of function units and fetch rate for each benchmark.

2 Logistics

This is a partner project. You are permitted to share code only with your partner. You may use C or C++ for this assignment, and you should only rely on standard libraries.

You are extending the processor/ component into your own superscalar processor simulator

3 Simulator Specifications

You will write a pipelined processor simulator based on Tomasulo's Algorithm in processor.c and any other files you wish to add.

This simulator will take traces as input and output cycle information and statistics regarding the instructions. See below for more information.

3.1 Simulator Input

Your simulator must support the following usage: (all parameters default to 1)

-d <D>: Dispatch queue multiplier (range: 1, 2)
-f <F>: Fetch rate (instructions per cycle) (range: 1, 4)
-m <M>: Schedule queue multiplier (range 1, 2)
-j <J>: Number of "fast" ALUs (range 1, 3)
-k <K>: Number of "long" ALUs (range 1, 3)
-c <C>: Number of CDBs (range 1, 4)

3.2 Trace Format

If any register has the value −1, then there is no register for that part of the instruction. The instruction may still have immediate values or other useful work, however, that is not pertinent for this project. For example, a ret instruction has been converted to an ALU operation for this project; however, expect that the next project will split them out into different types.

Registers have a valid value of [0,33). FYI, 0-15 are integer registers, 16-31 are floating point, and 32 is eflags.

3.3 Simulator Details

Function Units

The types of function units (FUs) you must support are shown in the table below:

Function Unit Type	Number of Units	Latency
0	Command line input j	1
1	Command line input k	3

Recall the number of function units is a parameter to the simulator and should be adjustable within range [1, 3] units for each type. Additionally, FUs of type 1 are pipelined with three stages.

In addition, branch operations also use FU type 0; however, you are not required to support them until Project 4. Similarly, memory operations use FU type 0 with latencies determined by a cache simulator, but they are not required until Project 4.

Tag Generation

Tags are generated sequentially for every instruction, beginning with 1. The traces are short enough, so you will not need to reuse tags.

Bus Arbitration

Since there are a limited number of result buses (common data buses), all instructions that complete in the same cycle may not be able to update the schedule queues and register file in the following cycle. To ensure one resulting state from the traces, assume the order of update is the same as the order of the tag values.

> For example, consider tag values 100, 102, 110, and 104 waiting to complete. This results in the schedule queue/register file update order of: 100, then 102, 104, 110. Note that these all happen at the same time.

We will assume any completed instruction that did not receive access to the bus due to contention still clears its pipelined function unit. If we did not enforce this, then additional ordering discrepancies could arise from trying to determine which instructions are blocked in the function unit's pipeline and which advance.

3.4 Pipeline Stages

Pipeline Timing

Below is a table of the pipeline stages and the number of cycles per instruction for each stage:

Stage Number	Name	Cycles per Instruction
1	Instruction Fetch / Decode	1
2	Dispatch	Variable, depends on resource conflicts
3	Schedule	Variable, depends on data dependencies
4	Execute	Depends on function unit type
5	State Update	Variable, structural hazards

Some additional points to consider:

- You should explicitly model the dispatch and scheduling queues.
- The fire rate (issue rate) is only limited by the number of available FUs.
- There are a limited number of result buses (also called Common Data Buses or CDBs).
- Assume that instructions retire in the same order that they complete. Instruction retirement is unconstrained (imprecise interrupts are possible).

Fetch/Decode Unit

The fetch/decode rate is *F* (command line input) instructions per cycle. Each cycle, the fetch/decode unit can always supply *F* instructions to the dispatch unit, provided there is room for the instructions in the dispatch queue.

Dispatch Unit

The dispatch queue has $d \times (m \times j + m \times k + m \times l)$ entries. During operation, the dispatch queue is scanned from header to tail (in program order). When an instruction is inserted into the scheduling queue, it is deleted from the dispatch queue.

When the dispatch queue is full, the instruction fetch/decode unit halts until space is available.

Scheduling Unit

The scheduling queue is divided into two sub-queues, one for each operation type (0, 1).

1. The queue for type 0 operations has $m \times j$ entries.
2. The queue for type 1 operations has $m \times k$ entries.

Any function unit can receive instructions from any scheduling queue entry for its type. Additionally, a fired instruction must remain in the scheduling queue until it completes.

If there are multiple independent instructions ready to fire during the same cycle in the scheduling queue, service them in tag order (lowest tag value to highest). When the scheduling queue is full, the dispatch unit should stall until space is available. For this reason, the dispatch rate is limited by the capacity of the scheduling unit.

Clock Updates

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.

Assume each clock cycle is divided into two half cycles. Note that you do not have to explicitly model this, but we require your simulator follow the half cycle ordering of events seen below.

First Half:

- The register file is written via the result bus.
- Any independent instruction in the scheduling queue is marked 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 the scheduling queue.

3.5 Simulator Output

Statistics Output

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

- Average number of instructions fired per cycle.
- Total number of instructions in the trace.
- Total simulator run time for the input, where run time is the total number of cycles from when the first instruction entered the fetch/decode unit until the last instruction completed.

The reference processor has a verbose output mode, where the columns represent the first cycle in which that instruction is in that stage. The columns are: <Instruction ID> <Fetch> <Dispatch> <Schedule> <Execute> <State Update>

4 Evaluation

Evaluation of your submission is based on the correctness of the simulator. It can be difficult to compare the results for longer traces. The reference processor will print out for each stage the cycle for when the instruction first runs that stage. For longer traces, you may want to redirect this output to a file and use diff to compare the results. You can also provide “-n <lines>” to control how many lines of instruction output.

There is **no** report for this project.

5 Handin Instructions

Please submit a tar of your repos contents to Autolab.

6 Hints

Below are some hints to guide you through this project:

- We recommend you first work out by hand what should occur in each unit at each cycle for an example set of instructions before you start writing your program. (e.g., traces/ls-proc.trace)
- Keep a counter for each line of the tracefile to use for Tomasulo tags.
- Operand renaming is not necessary for this assignment.
- You may find it useful to make a separate function for each pipeline stage.
- Execute the procedures in reverse order from the flow of instructions.
- It may be helpful to have a data structure for all of the pipeline latches, e.g. a two dimensional array, one dimension for the number of execution units, another dimension for the number of pipeline stages of the execution units.

7 Conclusion

Please don’t hesitate to reach out regarding clarifications or questions. Since this class is new, we would love to act on any feedback you can provide about the assignments.

Good luck on your pipeline simulator!