Project 2. Tomasulo Algorithm Pipelined Processor

Due 23:59, Thursday, May 2nd, 2024 (KST)

TA: cs510_ta@casys.kaist.ac.kr

1. Introduction

In this project, you will construct a simulator for an out-of-order superscalar processor with a reorder buffer (ROB) that uses the Tomasulo algorithm and fetches N instructions per cycle. This simulator should read traces of instructions from various applications, perform the simulation, and produce statistics regarding out-of-order processor behaviors.

Please read the descriptions below carefully and complete the missing components of the given framework. If you have any questions, feel free to ask TA via email, cs510_ta@casys.kaist.ac.kr. (Do not send emails to TA's individual email)

2. Explanation of Provided Framework

We are providing you with a framework to build the out-of-order superscalar processor simulator. You must fill in the following functions in the framework, defined in `procsim.cpp` :

```
void setup_proc(uint64_t r, uint64_t k0, uint64_t k1, uint64_t k2, uint64_t f, uint64_t
```

Subroutine for initializing the processor. You may add and initialize any global or heap variables as needed.

```
void run_proc(proc_stats_t* p_stats);
```

Subroutine that simulates the processor. The processor should fetch instructions as appropriate, until all instructions have executed.

```
void complete_proc(proc_stats_t *p_stats);
```

Subroutine for cleaning up any outstanding instructions and calculating overall statistics such as average IPC or branch prediction percentage.

You are only allowed to modify `procsim.cpp` and `procsim.hpp`.

3. Simulator Options

Build

```
$ make
```

Basic command

```
$ ./procsim [OPTIONS] < traces/[file.trace]
```

Options

- -f : Fetch rate (instructions per cycle)
- -m : Schedule queue multiplier
- -j : Number of k0 FUs
- -k : Number of k1 FUs
- -l : Number of k2 FUs
- -r : Number of ROB entries

Tips

We provide output and log files that would be helpful for your development. The output files show the simulator configurations and the cache statistics result that your simulator should generate. If your simulator produces identical results to the output files, it is implemented perfectly. The log files list how cache system behaves for each memory instruction. It would be of great help in debugging your simulator.

4. Specification of Simulator

1. Input Format

The input traces will be given in the form:

```
<address> <function unit type> <dest register #> <src1 register #> <src2 register #>
<address> <function unit type> <dest register #> <src1 register #> <src2 register #>
...
```

where

`<address>` is the address of the instruction (in hex),

`<function unit type>` is either "-1", "0", "1", or "2",

`<dest register #>`, `<src1 register #>`, `<src2 register #>` are integers in the range [0, 31].

- **Note**: If any register # is -1, then there is no register for that part of the instruction (e.g., a branch instruction has -1 for its <dest register #>). The instruction may still have immediate values or other useful work that is not pertinent for this project.

For example:

```
ab120024 2  1  2 3
ab120028 1  4  1 3
ab12002c -1 -1 4 7
bc213130 0  0  1 0
```

Means:

```
"operation type 2" R1, R2, R3
"operation type 1" R4, R1, R3
"operation type -1" -, R4, R7 (Branch, no destination register!)
"operation type 0" R0, R1, R0 (Destinations is also a source)
```

2. Function Unit Mix

Function Unit Type	Number of Units	Latency
0	parameter: `k0`	1
1	parameter: `k1`	2
2	parameter: `k2`	3

Notes:

- Instructions of type -1 are branches (see below) and are executed in the type 0 functional units.
- The number of function units is a parameter of the simulation and should be adjustable along the range of 1 to 3 units each.
- Function units of types 1 and 2 are pipelined into stages (2 stages for function unit type 1; 3 stages for function unit type 2).

3. Pipeline Timing

Assume the following pipeline structure:

Stage	Name	Number of Cycles Per Instruction
1	Reaches Instruction Fetch/Decode	Variable, depends on the dispatch queue occupancy
2	Reaches Dispatch	1
3	Reaches Scheduling	Variable, depends on resource conflicts (ROB and scheduler queue) and functional unit availability
4	Reaches Execute	Variable, depends on data dependencies
5	Reaches State Update	Variable, depends on function unit type (see table above)
6	Is Retired	Variable, depends on the state of the ROB

1. You should explicitly model the dispatch and scheduling queues
2. The dispatch queue has R entires.
3. **The scheduling queue is divided into three sub-queues.** There is one sub-queue each for operations of type 0, 1, or 2. The queue for type 0 operations has $m \times k0$ entires, the queue for type 1 operations has $m \times k1$ entires, and the queue for type 2 operations has $m \times k2$ entires (there are $k0$ type 0 function units, $k1$ type 1 function units and $k2$ type 2 function units). Any function unit can receive instructions from any scheduling queue entry for its type.
4. As a hint, you should store the original line number of the instruction in the trace file in the structure/object for an instruction. You do not have to use ROB index as tags and could use this, instead.
5. If there are multiple independent instructions ready to fire during the same cycle in the scheduling queue, service them in the program order (i.e., an earlier instruction in the trace file gets serviced first). (This will guarantee that your results can match ours). Since dispatch is inorder so the scheduling queues should already have instructions residing in order.
6. A fired instruction remains in the scheduling queue until it completes.

7. When the dispatch queue is full, stage 1 (Fetch/Decode) should be stalled until the cycle following when space is available in the dispatch queue.
8. When the scheduling queue is full, stage 2 (Dispatch) should be stalled until the cycle following when space is available in the scheduling queue.
9. The dispatch rate is limited by the capacity of the scheduling unit and the size of the ROB.
10. The fire rate (issue rate) is only limited by the number of available FUs.
11. **There are unlimited result buses (also called Common Data Buses, or CDBs).**
12. Assume that instructions retire in the same order that they were fetched.

A. Fetch/Decode Unit

The fetch/decode rate is up to N instructions per cycle. Each cycle, the Fetch/Decode unit can always supply N instructions to the Dispatch unit, provided there is room left in the dispatch queue.

For this project, we do not model an instruction cache. In other words, the instruction cache is ideal and always returns a hit.

B. Branch Prediction

The instruction trace gives the actual control path taken by the program. So the next instruction after a branch is the one that the branch would resolve.

We are not going to implement a branch predictor for this project. Branches are like any other instruction and do not cause any stalls. They do have data dependencies, though.

C. 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*
- (Execute) // Types 2 & 3
- *(Pipeline Register)* // Types 2 & 3
- (Execute) // Type 3
- *(Pipeline Register)* // Type 3
- 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 to fire
	The dispatch unit reserves slots in the scheduling queues and the ROB
	Instructions complete execution
Second half	The register file is read by Dispatch unit
	Scheduling queues are updated via a result bus
	The State Update unit deletes completed instructions from scheduling queue
	The State Update unit deletes completed instructions from the ROB

D. 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. An instruction can only get into the scheduling queue if it can reserve a slot in the ROB. Dispatch is inorder. Instructions starting from the top of the dispatch queue get into the scheduler queue.

E. Tag Generation

The index of the ROB entry is unique per instruction and thus can serve as the TAG. You could use either the ROB index or the "trace line #" as the tag.

F. Bus Arbitration

Since there are unlimited number of result buses (common data buses), all instructions that complete in the same cycle can update the schedule queues and register file in the following cycle. Unless we agree on some ordering of this updating, multiple different (and valid) machine states can result. This will complicate validation considerably. Therefore, assume the order or update is the same as the program order (or order of instructions in the trace file).

- Example:
  Line numbers of instructions waiting to complete: 100, 102, 110, 104
- Action:
  The instruction with the line number =100 updates the schedule queue/register file, then instruction 102, then 104, then 110. *All of these happen in parallel, at the same time, and at the beginning of the next cycle!*

G. State Update Unit

Implement an R-length ROB. Slots in the ROB are allocated in a circular FIFO queue by the dispatch unit. If there are no free slots in the ROB for an instruction, an instruction stalls in the dispatch queue.

In this project, you would be implementing a "ROB with Future File". However, you do not need to implement the architectural register file, as it is only needed in the case of interrupts and we are not implementing interrupts. For the purposes of implementing Tomasulo's algorithm, use a 32-entry register file.

Use a circular FIFO or double-ended queue to implement the ROB.

The ROB enforces in-order retirement of instructions entering the pipeline. So dispatch and retire is in-order but execution can be out-of-order.

The machine can retire up to N completed instructions per cycle. Although multiple instructions can retire at once, only a contiguous block of completed instructions at the head of the ROB can retire. The first uncompleted instruction halts retirement until that instruction completes.

Retire instructions starting from the head of the ROB. Since the ROB enforces a FIFO order, this should be deterministic.

5. Outputs / Statistics

The simulator outputs cycle information for each instruction. With the first instruction numbered 1 and the first cycle being 1, output the cycle when every instruction enters each pipeline unit.

Also, 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 retired per cycle.

6. 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++. Each "step" in the algorithm represents a 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++ directly.
- Keep a counter of each line in the trace file. Use this for the Tomasulo tags or use the ROB index.
- 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.).
- It should help 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).
- 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.
- **Start early: This is a difficult project.**

7. Grading Policy

Your code will be evaluated using four instruction traces provided to you, and will be graded based on its performance with both the default configurations defined in `procsim.hpp`, as well as a hidden configuration that has not been announced.

Your grade will be determined based on the following criteria:

- **100%**: Produces the same result as the correct answer for all configs.

- **95%**: Produces the same result as the correct answer for default config, but not for the hidden config.
- **85%**: Produces a result different from the correct answer, but with an average error of under **5%**.
- **50%**: Produces a result different from the correct answer, with an average error of over **5%**.
- **0%**: Code is not compilable or does not produces any result.

We will count the number of correct and wrong pipeline states for each cycle.

8. Submission

Zip your project directory and name it with **your student id**.

```
$ tar -cvf [student id].tar procsim.cpp procsim.hpp
($ tar -cvf 20241234.tar procsim.cpp procsim.hpp)
```

or

```
$ make submit
```

Submit the tar file to KLMS.

Please make sure that your code compiles and runs well. We will give 0 points to the code that does not compile without any exception.

gcc / g++ version = 7.5.0 / c++11

9. Late Policy & Plagiarism Penalty

You will lose **30%** of your score on the **first day** (May 3rd 0:00 ~ 23:59). We will **not accept** any works that are submitted after then.

Be aware of plagiarism! Although it is encouraged to discuss with others and refer to extra materials, **copying other students' or opened code is strictly banned.**

TA will compare your source code with some open-source codes and other students' code. If you are caught, you will receive a serious penalty for plagiarism as announced in the first lecture of this course.

If you have any requests or questions regarding administrative issues (such as late submission due to an unfortunate accident), please send an e-mail to TA.