Georgia Institute of Technology

CS 4290

Spring 2021 – Patrick Lavin

Project 3: Out of order execution in a superscalar pipelined processor with precise interrupts via use of a ROB

Due Date : Tuesday, July 27th (No extensions)

VERSION – 1.2

Version 1.1

- Accidentally skipped

Version 1.2

- Removed references to experiments

Rules

The rules for project 3 are the same as previous projects:

1. All students 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 however **not acceptable** to collaborate on the simulator design.
4. You should do all your work in the C or C++ programming language, and your code 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 and when 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.
7. Discussion on Piazza is highly encouraged but refrain from posting algorithm details

1. Project Description

In this project, you will complete the following:

1. Construct a simulator for an out-of-order superscalar processor that dispatches *F instructions* per cycle and uses the Tomasulo algorithm with a RAT (Register Alias Table) and PRegs approach (Physical Registers).
2. Support the notion of precise exceptions and interrupts via the re-order buffer approach. Use your idea of a ROB to implement this feature.
3. ~~Use your simulator to determine the appropriate number of function units, and fetch rate for the default number of ROB entries and PREGs.~~

4. ~~Use your simulator to determine the appropriate number of ROB entries and PREGs for the default number of function units.~~

Directory Description

The procsim_cpp.tar.gz package contains:

1. Makefile: to compile your code
2. Procsim_driver.cpp: contains the main() method to run the simulator : ***Do not edit this file***
3. procsim.h/hpp: Used for defining structures and method declarations: ***you may edit this file to declare or define structures and methods***
4. procsim.c/cpp: ***All your methods are written here***
5. traces: contains the traces to pass to the simulator (more details in a later section)
6. debug: contains traces to help you debug your processor

Assumptions:

For simplicity, you do not have to model issue width, retire width, number of result buses and PRF (Physical Register File) ports. Assume these are unlimited and do not stall the processor pipeline.

Understanding the command line parameters

The program should run from this root directory as:

./procsim –f *F* -*p P* –j *J* –k *K* –l *L* -r *R* -i *I* -t <trace_file>

The command line parameters are as follows:

- F – Fetch rate (instructions per cycle)
- P – Number of PRegs
- J – Number of k0 function units # ALU Units (Used for BRANCH as well)
- K –Number of k1 function units # MUL Units
- L – Number of k2 function units # LOAD / STORE instruction
- R – Number of ROB entries
- I – Interrupt Interval (cycles)
- t – trace_file – Path name to the trace file

Note: Default values for J=3, K=1, L=2, R=12, F=4, and P = 64 are the default values.

Understanding the Input Trace Format

The input traces will be given in the form:

<Address> <Opcode> <Dest Reg #> <Src1 Reg #> <Src2 Reg #> <LD/ST Addr> <Br Target> <Br Taken>

<Address> <Opcode> <Dest Reg #> <Src1 Reg #> <Src2 Reg #> <LD/ST Addr> <Br Target> <Br Taken>

...

where

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

<Opcode> is one of the following:

OPERATION	Opcode	Functional Unit
NOP	1	NA
ADD	2	J (ADD unit)
MUL	3	K (MUL unit)
LOAD	4	L (LOAD/STORE unit)
STORE	5	Load-Store Queue (Not modelled so can just commit in order after spending 1 cycle in execute)
BRANCH	6	J (ADD unit)
INTERRUPT	7	J (ADD unit)

<Dest Reg #> [0..31]

<Src1 Reg #> [0..31]

<Src2 Reg #> [0..31]

<LD/ST Addr> is the effective load or store address (aka the memory address for the operation)

<Br Target> is the branch target for a branch instruction

<Br Taken> tells if the branch is actually taken

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 #>)
- If the instruction is not a branch you can ignore the <Br Target> and <Br Taken> fields
- If the instruction is not a load/store you can ignore the <LD/ST addr> field

Pipeline Structure:

For this project assume the pipeline has five stages. Each of these stages is described below:

Stage Name	Number of Cycles per instruction
Fetch	1
Dispatch	Variable, depending on resource conflicts
Schedule	Variable, depending on data dependencies
Execute	Variable, depending on operation
State Update	Variable, depends on data dependencies

Understanding each stage:

Fetch:

1. The fetch unit fetches up to F instructions from the trace into empty slots in the dispatch queue. Note that the dispatch queue is infinite in size. Assume that the frontend Fetch unit never misses in the Instruction Cache (which we don't model).
2. For simplicity, assume that the branch predictor has a 100 percent accuracy and hence do not perform any explicit branch prediction. Branch instructions are scheduled like any other instruction.

Dispatch:

1. The dispatcher attempts to dispatch as many instructions as it can from the dispatch queue into empty slots in the scheduling queue/reservation stations, in program (trace) order, every cycle. When there are no more slots in the scheduling queue/reservation stations, it stalls.
2. The source and destination register numbers are checked and the Physical Register File (PRF) is accessed along with the Register Alias Table (RAT) and Reorder Buffer (ROB). (See course notes for details)
3. The lowest numbered free Preg is assigned as the destination register for an instruction whose destination register is not equal to -1. This is recorded in the RAT. The dispatch unit stalls if it is unable to assign a PReg to an instruction requiring one.
4. A ROB entry is allocated for each instruction that is being dispatched. Each ROB entry must store the Areg number and the previously mapped Preg for that Areg.
5. NOP instructions are never scheduled. This means if you encounter a NOP instruction at the head of the dispatch queue, you can ignore it and move on to servicing the next instruction.

6. The First 32 physical registers of the PReg File are reserved as architectural registers (free bit is always false). All other physical registers can be free'ed and assigned as destination Pregs as required.

The most important job of this stage is to access the four different hardware structures namely scheduling queue, ROB, PRF, and RAT and update them as required

Schedule:

1. There are *2*(J + K + L)* entries in the scheduling unit (schedule queue).
2. If there are multiple independent instructions ready to fire during the same cycle in a reservation station, service them in program order, and based on the availability of function units.
3. A fired instruction remains in the reservation station until it completes. The latency of each unit type is listed below
4. The rules for firing follow the Tomasulo with Preg approach (Please look at class notes for details): ~~Loads and Stores are handled as per the rules described in a later section.~~
5. The MUL and Load Function units are pipelined
6. Only one store instruction (oldest, i.e., Earliest in program order) can be fired. Number of store unit is always 1.

Function Unit Type	**Number of Units**	**Default**	**Latency**
0	Parameter: j	**3**	1
1	Parameter: k	**1**	3
2	Parameter: l	**2**	**Loads – 2** **Stores** – 1

The number of function units is a parameter of the simulation and should be adjustable along the range of 1 to 3 units of each type.

Reminder: Instructions and the execute unit they use are listed in section 1.

Execute:

1. The function units are present in this stage. When an instruction completes, it updates the reservation stations, the PREG, and the ROB. Those updates are visible in the following cycle. An instruction is considered "completed" after it spends the required cycles in execute.
2. The ROB and PRF are accessed in this stage of the pipeline

State Update:

1. This stage performs in-order retirement from the reorder buffer. It checks if the oldest entry in the ROB is ready, and if so, the instruction is retired, freeing up the previous PReg.

2. The result is written to the Areg in the ROB. This unit can retire as many instructions as possible until the head of the ROB is an instruction that has not completed yet.

Precise Interrupts/Exceptions:

1. An interrupt called RAISE_EXCEPTION is set by the "run_proc" function every *'I'* cycles. When the flag is raised, an "Interrupt Instruction" should be added by the processor to the head of the dispatch queue. Dispatch should be stalled, and when the added interrupt instruction reaches the head of the ROB, the ROB along with the reservation stations must be flushed.(Note: Since dispatch is stalled till interrupt is retired, there is no speculative state in RS/ROB. No need to explicitly flush RS/ROB). Once the "Interrupt Instruction" has been retired from the processor, i.e popped from the head of the ROB, the interrupt flag should be lowered, and normal operations resume.
2. The "Interrupt Instruction" uses functional Unit of type K0, i.e. One of the ADD units. It has no destination nor sources. The opcode for the interrupt instruction is given is 7 (as shown in the table above).

When to Update the Clock

Note that the actual hardware has the following structure:

Fetch
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 the bordering of events)**:

Cycle Portion	Action
1	The oldest completed instruction retires from the ROB
2	Function Units write to the PRF and ROB for completing instructions
3	Independent/Ready instructions in the schedule queue are marked to fire
4	The dispatch unit accesses the ARF, ROB, RAT, PRF and sends out instructions to the schedule queue
5	Instructions are fetched from the instruction trace

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.

Operation of the Dispatch Queue

Note the dispatch queue is scanned from head to tail (in program order). When an instruction is inserted into the reservation stations (schedule queue), it is deleted from the dispatch queue. Dispatch stalls when it can't find a free reservation station or a free PReg or there are no available slots in the ROB.

Initial Conditions

On processor initialization, the following are valid:

- The ARegs are mapped to the first 32 registers in the PReg file. That is, the RAT contains mappings to the first 32 ([0..31]) PRegs, each of which are "ready". The other PRegs are initialy "free".
- All Function Units are ready.

2. Statistics

The simulator outputs some basic statistics such as instruction count, cycle count and IPC which you will be responsible for updating in the various stages of the pipeline. Apart from the statistics, the cycle-by-cycle behavior of each instruction is printed (section 4). Sample code for this is below:

3. Given Framework

The tarball for this project contains the following files:

- procsim_driver.c/cpp - Driver for the simulation. Implements the read_instruction function which you will use to "fetch" instructions from the traces
- procsim.h/hpp - Header file with instruction, configuration and statistics struct definitions along with functions that you will be filling in
- procsim.c/cpp - The file where you will be writing most of your code
- Makefile - To build the procsim executable
- traces – The traces used for testing
- debug – Cycle by cycle behavior of a correct simulator, provided for debugging purposes

The code comes with a framework with the processor timing and periodic interrupt generation already enabled. Apart from three primary simulator functions (below), each stage of the pipeline has an associated function which is provided to you in a run-loop. You will be responsible for declaring any global data structures you might need for your processor and filling in the functions corresponding to each pipeline stage.

```
void setup_proc(const proc_conf_t *config)
```

Use this to initialize globals, etc

```
void run_proc(proc_stats_t *p_stats, const proc_conf_t *config)
```

Run the processor until the input trace is consumed in this function. Call the read_instruction function to "fetch" an instruction from the input trace file. Make sure to update the appropriate stats as well.

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

Finalize statistics computation and print the cycle-by-cycle behavior as per the format given in the validation logs here.

4. Validation Requirements

Validation

To verify your simulator against the test cases, use the python verification script. You can run the script as:

```
$ python3 verify.out
```

Once you run the verify script a file called verify.out will be generated. You should diff that against the provided solution.log file by running:

```
$ diff solution.log verify.out
```

To help you debug, the cycle by cycle behavior of the 4 traces for the default configuration is included in the debug folder. The following print statement was used for this purpose. The instruction_number field refers to the "trace" line number of the instruction (i.e., the first instruction has its number field set to 1)

```
if (instruction.opcode == OP_INT) {
    printf("\nINT");
} else {
    printf("\n%lu ", instruction.instruction_number);
}
printf("%lu %lu %lu %lu %lu", instruction.fetch_cycle,
instruction.dispatch_cycle, instruction.schedule_cycle,
instruction.execute_cycle, instruction.state_update_cycle);
```

Hints

- Schedule queue slots freed in cycle X will be visible to dispatch in cycle X + 1.
- Count Dispatch Queue size at the end of every cycle for stats like average_dispatch_queue_size
- Include Interrupts but not NOPs in the retired_instructions count, and debug outputs.
- Although the idea of PRegs is to not have tags, since this is a simulator, matching completing instructions to ROB entries may be a lot easier with them.

What to Turn In?

You should turn in a tarball named project2-submit.tar.gz which includes the following files:

- procsim.c/cpp
- procsim.h/hpp
- procsim_driver.c/cpp
- Any other source files you might created
- Makefile

The Makefile has a "make submit" build option which can generate a tar for you. Please make sure the filenames in the Makefile exactly match the files you want to submit.

Grading

0% you hand in nothing or hand in something late (or something that does not compile or segfaults)

+60% you hand in code that shows a reasonable attempt (stats match within 10%) but does not pass validation

+30% your code passes all validation tests

+10% your code is well formatted and does not have memory leaks and other bugs