Georgia Institute of Technology
Schools of Computer Science and Electrical & Computer Engineering
CS 4290/6290, ECE 4100/6100:
Spring 2018
Prof. Tom Conte

Project 2: Out of order execution in a superscalar pipelined processor with support for precise exceptions and interrupts

Due Date : Friday, March 30th at 11:55pm via Canvas

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 however 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. If you choose to use Java, it is your responsibility to port the given framework to Java. It is your responsibility to create a shell script that provides the same command line interface as the original framework. Any bugs introduced in either of these are your responsibility.
6. 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
7. .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.
8. Discussion on Piazza is highly encouraged but refrain from posting algorithm details

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 Physical Registers/Register Alias Tables approach.
2. Use a re-order buffer (ROB) to support the notion of consistent state or precise exceptions and interrupts. To support this use your idea of re-order buffer (ROB)

3. Use your simulator to determine the appropriate number of functional units, fetch rate ~~and result buses~~ for each benchmark for the default number of ROB entries and PREGs (explained later)
4. Use your simulator to determine the appropriate number of ROB entries and PREGs for the default number of functional units. (explained later)

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.hpp: Used for defining structures and method declarations : ***you may edit this file to declare or define structures and methods***
4. Procsim.cpp: ***All your methods are written here***
5. Traces: contains the traces to pass to the simulator (more details in the later section)

Note: procsim_c.tar.gz contains equivalent files

Assumptions:

For simplicity, you do not have to model issue width, retire width, number of result buses and PRF ports. Assume these do not stall your processor.

Understanding the 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 –f F –j J –k K –l L -r ROB -p PREG <trace_file>

The command line parameters are as follows:

- F – Dispatch rate (instructions per cycle)
- J – Number of k0 function units
- K –Number of k1 function units
- L – Number of k2 function units
- ROB - Number of ROB entries
- PREG - Number of PREGs
- trace_file – Path name to the trace file

Understanding the parameters:

All the parameters are to be used to conduct your experiments after your simulator is up and running.

Note: Default values for k0=3, k2=2, k3=1, ROB=12, F=4, PREGS=32

For doing your set of experiments after your simulator is validated, follow the following procedure

1. To determine the appropriate number of functional units and fetch rate for each benchmark, vary the number of ROB entries ~~and PREGs~~ based on the equations described in the scheduling stage

   For example, if you vary the numbers of k0, k1, k2 and F as the following,
   k0=4, k1=4, k2=3 F=8
   your parameters for ROB ~~and PREG~~ should be changed as
   ROB=2(4+4+3) = 22
   ~~PREG= 8*8 = 64~~
   Leave the number of PREGS as 32 (8*4)

   With these parameters, determine the minimum value of k0, k1, k2, and F
2. For the other part of the experiment, to determine the minimum number of ROB entries ~~and PREGs~~ for each benchmark

   Vary the number of ROB and ~~PREGs,~~ keeping k0, k1, k2, k3, F constant.
   For example, you can set 10 ~~100~~ entries in ROB ~~and 128 PREGs~~
   k0=3, k1=2, k2=1, F=4

Understanding the 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#>
<src2 reg #> are integers in the range [0….31] ~~[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 : Note: no destination register!

Note:

~~Instructions of type -1 are executed in the type 1 function units.~~

The type of functional unit required for each instruction is present in the instruction itself. For instructions in the trace that require units of type '-1', use units of k1 type.

Pipeline Structure:

For this project assume the pipeline has ~~5~~ four stages. Each of these stages is described below:

Stage Name	**Number of Cycles per instruction**
Dispatch	Variable, depending upon resource conflicts
Schedule	Variable, depending upon data dependencies
Execute	1
Status Update	Variable, depends on data dependencies

Understanding each stage:

Dispatch:

1. The dispatcher attempts to dispatch up to *F* instructions from the trace into empty slots in the scheduling queue/reservation station, in program (trace) order each cycle. When there are no more slots in the scheduling queue/reservation station, it stalls.
2. If there are empty slots, the sources and destination register numbers are checked and physical register file (PRF) is accessed along with the Register Alias Table (RAT) and Reorder buffer (ROB) (see lecture notes for the details).
3. Assume default size of PRF to be ~~8*F~~ 32 registers (Pregs).
4. The lowest numbered free Preg is the one chosen by Dispatch to assign to an instruction's destination register. This is recorded in the RAT.
5. Assume by default the ROB has the same number of entries as the scheduling queue (see below) entries. Each ROB entry must store the Areg number and the previous Preg for that Areg.
6. When there are no available physical registers in the PRF to remap an architectural register to, the dispatch unit stalls and cannot dispatch any new instructions.

7. The dispatch unit also needs to stall and cannot fetch any new instructions if the ROB is full or the scheduling queue is full.

Note:
There are 32 registers in the architectural register file (ARF)

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

The Scheduling Stage

1. The size of the scheduling queue (or reservation station) is 2*(number of k0 function units + number of k1 function units + number of k2 function units)
2. If there are multiple independent instructions ready to fire during the same cycle in the scheduling queue, service them in program order, and based on the availability of functional units.
3. A fired instruction remains in the reservation station until it completes

Function Unit Type	**Number of Units**	**Default**	**Latency**
0	Parameter: k0	**3**	1
1	Parameter: k1	**2**	1
2	Parameter: k2	**1**	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 of type -1 are executed in the type 1 function units.

Execute:
The function units are present in this stage and the outputs from the function units use the result buses to access the PRF, update the ROB.

The State Update Unit
This stage performs in-order retirement from the reorder buffer. It checks if the oldest entry in the ROB is ready, if its ready, retire the instruction and free up previous PReg. It writes the result to the Areg shown in the ROB. This unit can retire as many instructions as possible until the head of the ROB is an instruction that has not yet completed.

Clock Propagation and actual hardware:
Note that the actual hardware has the following structure:

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

Cycle Portion	**Action**
1	Retire the oldest completed instruction(s) from the ROB
2	Function Units write to the PRF and ROB for completing instructions
3	Any ready/independent instruction in the scheduling queue is marked to fire (depending upon availability of functional units)
4	The dispatch unit accesses the ARF, ROB, RAT, PRF and sends out entries to the reservation station. When the hardware structures get full, it stalls.
5	Instructions fetched from the 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.

Output

More on this later

~~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: IPC of retired instructions per cycle, average number of PRegs busy per cycle, average number of dispatch stall cycles per cycle.~~

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 Canvas as explained later.

More on this later

~~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.~~

Experiments

After your simulator is validated, for each trace:

1. Find the minimum value of F, k0, k1 and k2 to achieve a high value of IPC.

Suggested approach: Set new values of F, k0, k1 and k2 ~~to a very high number. Record the IPC. This is your target IPC.~~ Find the smallest values of F, k0, k1 and k2 that achieve at least 98% of ~~that~~ the IPC obtained using default parameters.

2. Find the minimum value of the number of ROB entries and and scheduler entries ~~registers in the PRF~~ to achieve a high value of IPC.

Suggested approach: Set new value of ROB ~~and PREG to a very high number~~. Record the IPC. ~~This is your target IPC.~~ Find the smallest values of ROB ~~and PREG~~ that achieve at least 98% of ~~that~~ the IPC obtained using default parameters.

Use all statistics to explain the solution you arrived at.

Grading

0% you hand in nothing or hand in something late

+50% you hand in code that shows a reasonable attempt and passes some of our validation tests

+30% your code passes all validation tests

+15% your experiments are completed
+5% your explanation of the results is exemplary and of research quality

Submission

Submit only the files procsim.cpp and procsim.hpp
No additional files or logfiles should be submitted.

Clarification/Hints:

Here are some hints, clarifications related to Project #2 that you might find very useful as you start coding.

1. For instructions in the trace that require units of type '-1', use units of k1 type
2. For simplicity, you can dispatch an instruction only when you have a free slot in the ROB, scheduler queue and Preg.

   In other words, you need all these three conditions to be met, before you read an instruction from the trace.

   If the conditions are not met, you stall
3. The dispatch rate is F, however if due to resource constraints you cannot dispatch all of them in a cycle, you have to dispatch as many as you can.

   For example, if F=4, and in some random cycle, you have only 1 slot in the ROB and 2 slots in the scheduler queue and multiple free Pregs, you will dispatch 1 instruction in this cycle
4. Whenever there is a tie between multiple instructions waiting to get scheduled to the same functional unit, the oldest instruction goes first
5. Instructions that have destination register as '-1' can be treated as ready to retire once they reach the status update stage
6. Once your simulator is set up, you have to perform two experiments to find the optimal number of functional units and ROB/PRF size for each trace. A good way to do this is collect additional statistics about the number of times your processor stalls due to non-availability of any of these. That will help you with ideas to set up additional experiments and tweak the number of different parameters.
7. For your reference several outputs are given and here is a description of each of the directories that you would notice in the tarball
   a. all_traces: contains detailed information about the instructions present in each stage every cycle

b. small_trace: to help you debug, I have provided even a more detailed output log for a very small trace (gcc.1k.trace that resides in this directory). This log contains the state of the scheduler queue, ROB, RAT in each cycle in addition to information regarding the state of the pipeline stages. gcc.1k.trace is essentially the first 1000 instructions of the gcc.100k.trace
c. submission: this directory contains the output log files with the minimal number of print statements. Only the statistics are printed. Make sure that the code you submit only generates such a log file. **Any additional print statements that you used for debug must be removed if you want our grading scripts to pass.**

8. For the purpose of statistics in this project, the avg. inst fired per cycle and avg. instruction retired per cycle is same as IPC.
9. We will run your submission against several other traces and hence hardcoding values of the statistics may not be a useful approach to complete the project.
10. Do not submit any files other than the ones provided to you, otherwise our grading scripts will fail.