Georgia Institute of Technology

CS 4290/6290, ECE 4100/6100:

Spring 2020 - Prof. Tom Conte

Project 2: Out of order execution in a superscalar pipelined processor with speculative execution

Due Date : Friday, March 27th at 11:59pm via Canvas

VERSION – 0.99

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 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 Register Alias Table and PRegs.
2. Implement a Yeh-Patt branch predictor to support speculative branch execution.
3. Implement a load/store queue which can complete one store instruction every cycle
4. Implement the “checkpoint-repair" scheme for state-update.
5. Use your simulator to determine the appropriate number of function units, fetch rate and branch predictor size for the given workloads.

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)

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 -y Y -i <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 reservation stations per function unit type
- Y– log(Number of entries in BTB) - Each Smith Counter in the predictor is 2-bits wide
- 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 j=3, k=2, l=1, R = 1, G=8, F=4, and P = 128 are used by the validation traces we will provide you

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>

<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	L
BRANCH	6	J

```
<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 upon resource conflicts
Schedule	Variable, depending upon data dependencies
Execute	Variable, depending on operation and store buffer hit
Status Update (Execute completion)	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, hence the only source of stalls is due to branches *(However, we don't explicitly stall the fetch stage, explained later)*. We assume that the frontend Fetch unit never misses in the Instruction Cache (which we don't model)).
2. When a branch occurs (OP_BR), the branch is predicted using the BTB and Yeh-Patt predictor.
3. The trace also has a target and the branch behavior which is checked against the prediction. If the prediction was incorrect, you must stop fetching instructions until the mispredicted branch completes in the k0 unit. Restart fetching (step 1) the cycle after the mispredicted branch completes.

Dispatch:

1. The dispatcher attempts to dispatch as many instructions as it can from the dispatch queue into empty slots in the appropriate reservation station queue, in program order, every cycle. When there are no more slots in the scheduling queue/reservation stations, it stalls.
2. If the dispatch unit also stalls if it is unable to assign a PReg to an instruction.
3. The dispatch unit also stalls when there is a branch misprediction. This has been described in a later section.
4. 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.

Schedule:

1. There are N*k*i* entries in the scheduling unit for function unit of type k*i.*
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. Loads and Stores are handled as per the rules described in a later section.

Function Unit Type	Number of Units	Default	Latency
0	Parameter: j	**3**	1
1	Parameter: k	**1**	3
2	Parameter: l	**2**	**Loads –** 1 – 2 1 – If load hits in the store buffer 2 – If load misses in the store buffer **Stores** – variable (The store buffer retires one instruction every cycle, i.e. The head of the queue completes every cycle)

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 and, if it is a mispredicted branch, the fetch unit. Those updates are visible in the following cycle. An instruction is considered "completed" after it spends the required cycles in execute.
2. The load/store queue is present here. All store instructions are inserted in the load/store queue. The queue completes one store instruction every cycle.
3. A load checks the store queue and completes in one cycle if it hits. Otherwise it must spend 2 cycle in the execute stage (for getting data from the cache, which we don't explicitly model)

Status Update (last stage of Execute):

1. Completed instructions broadcast their results in this state. The register file, the branch predictor and other entities are updated here.
2. Instructions that complete are also removed from the reservation stations.

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 this ordering of events)**:

Cycle half	**Action**
First half	The register file is written via a result bus
	Any independent instructions in the reservation stations are marked to fire
	The dispatch unit reserves slots in the scheduling queue
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 (reservation stations)

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 that can be assigned to the instruction's destination.

Bus Arbitration

Since we are modeling infinite result buses, it means that all instructions in the schedule queue (aka reservation stations) that complete may update the register file and schedule queue in the same cycle. Unless we agree on some ordering of this update, multiple different (and valid) machine states can result. This will complicate validation. Therefore, assume that the order of updates is the same as Preg order. Consider a scenario where Preg# 10, 12, 11 and 14 are waiting to complete: The order of completion is Preg# 10, 11, 12 and 14, all at the beginning of the next clock cycle!

Branch Prediction

1. Branches are predicted in the Instruction Fetch stage.
2. Branches are predicted using a 2^Y entry Yeh-Patt predictor using a bimodal branch predictor (Smith Counter) at each index of the Yeh-Patt predictor where the initial state is 01 for each counter. The per branch history is Y bits wide, and the instruction address is hashed to get an index into the Branch Target Buffer using this function:

```
(Address >> 2) % (1 << Y);
```

3. Branch behavior (whether the branch is taken and where the branch goes) is known after the instructions has executed. (After it has spent one cycle in execute in a 'j' (ADD) unit).
4. If the branch behavior and the prediction are correct (note: check the predictor value when branch is put into Dispatch Queue), continue executing the trace and there are no stalls. However, if the branch prediction and actual behavior don't match, you will stall the DISPATCH unit until the miss-predicted branch reaches the STATUS UPDATE stage of the pipeline. After the miss-predicted branch has performed state update, the DISPATCH unit can begin dispatching instructions again
5. **Note:** The FETCH unit is never stalled. This is because the trace only contains correct instructions and does not contain the incorrect instructions that a branch miss-prediction would have induced, and multipath fetch (as performed in real processors) would have fetched both the branch taken and not taken paths, squashing the incorrect instructions after branch resolution. This system logically performs the same function as an efficient instruction retirement system (such as checkpoint repair or ROB with future file) with branches would perform. The stall allows for the penalty of miss-prediction to be included in the IPC calculations, however, removes the complexity of implementing a smart retirement scheme in the simulator you are writing

6. The FUs will be freed once the instruction broadcasts on the common data bus. If the instruction is a branch, update the predictor when you free the FU.

7. If more than one branch exits in EX and are preparing to Status Updates, serve them in the lowest Preg order.

8. In Status Update, use the same function (point 2, above) that was used to index into the history table to update smith counters using the address of the branch instruction which is retiring. Then update the relevant history register.

Load and Store Reordering

Out-of-order Loads and Stores to the same address can cause errors in program execution. That means before a store instruction is scheduled (aka Fired) it needs to ensure that all loads and store before it (i.e. before it in program order) to the same address have finished executing. And before a load instruction is fired it needs to make sure that all the stores before it to the same address have finished executing. Here is how this can be achieved:

- A load, X, can be issued (i.e can occupy a Function Unit) if there are no stores to the same address before X in program order.
- Stores X can be issues if there are no stores or loads to the same address before X in program order.
- If a load is issued and the address exists in the load/store queue, it can be resolved in 1 cycle. Otherwise it has a latency of 2 cycles.
- Stores go to the load/store queue (infinite size). Where the load/store queue retires one store every cycle (i.e. the head of the load/store queue is retired every cycle)

2. Statistics

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

- Total number of instructions completed
- Total number of branch instructions
- Total number of correctly predicted branch instructions
- Prediction accuracy
- Total load instructions
- Total store instructions
- Average Dispatch queue size
- Maximum Dispatch queue size
- Average instructions retired per cycle
- Total run time (cycles)
- The cycle by cycle behavior of all cycles of execution

3. Given Framework

We have provided you with a tar ball which 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.tar.gz - A tarball containing the traces folder

There are three functions that you will be implementing along with any helpers that you might need. We strongly suggest you break down the processor implementation into fetch, dispatch, scheudle, execute and state update functions.

`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. Experiments

Validation

Your simulator must completely match the validation outputs that we will be providing on Canvas.

Experiments

After your simulator is validated, for each trace you will be finding the optimum pipeline configuration for each trace. More details will be provided shortly.

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