Georgia Institute of Technology
Schools of Computer Science and Electrical & Computer
Engineering CS 4290/6290, ECE 4100/6100:
Spring 2019
Prof. Tom Conte
Project 2: Out of order execution in a superscalar pipelined processor with speculative execution
Due Date : Friday, March 29th 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 or copying from online sources is viewed as cheating and will be dealt with severely according to university policies
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, sharing any code or the on the final experiments
4. You should do all your work in the C or C++ programming language, and 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. Read the assignment thoroughly before you begin coding. Time spent reading the assignment will help make the debugging process easier!
8. 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 scheduling queue comprised of reservation stations.
2. Implement a Gshare branch predictor to support speculative execution.

3. Implement a basic direct mapped L1 Data Cache to speed up store and load instructions.
4. Use your simulator to determine the appropriate number of function units, fetch rate and branch predictor size.

Directory Description:

We have provided you with the following files:

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)

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. Some other assumptions that have been made have been described where relevant.

Understanding the command line parameters:

The generated binary (procsim) should be run from the root directory as:

```
./procsim –f F –k K –l L -m M -r R -g G -c C -i <trace_file>
```

The command line parameters are as follows:

- F – Dispatch rate (instructions per cycle)
- K – Number of k0 function units # ALU Units (Used for BRANCH as well)
- L –Number of k1 function units # MUL Units
- M – Number of k2 function units # LOAD / STORE instruction
- R – Number of reservation stations per function unit type. That means there are a total of R times (K + L + M) reservation stations
- G– `log2(Number of entries in BTB)` i.e. the BTB has 2^G entries
- C – `log2(Size of data cache in bytes)` i.e. the Cache is 2^C bytes in size
- <trace_file> – Path name to the trace file

The parameters configure your processor and will be used by you to perform experiments to find the optimal processor (see section 4, Experiments) for each trace.
The validation logs provided will use the default values:

K = 3, **L** = 1, **M** = 2, **F** = 4, **R** = 1, **G** = 8, **C** = 10

Understanding the Input Trace Format:

The input traces will be given in the following form. The driver file contains a function to parse this input and populates an *instruction structure*:

```
<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	k0
MUL	3	k1
LOAD	4	k2
STORE	5	k2
BRANCH	6	k0

`<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

<u>Pipeline Structure:</u>

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 cache misses
Status Update (Execute completion)	Variable, depends on data dependencies

<u>Understanding each stage:</u>

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 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.
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 each cycle. When there are no more slots in the scheduling queue/reservation station, it stalls.
2. The dispatch unit also stalls when there is a branch misprediction. This has been described in a later section.

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

Function Unit Type	**Number of Units**	**Default**	**Latency**
0	Parameter: k0	**3**	1
1	Parameter: k1	**1**	3
2	Parameter: k2	**2**	1 (Cache Hit) / 10 (Cache Miss)

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 above.

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 data cache is checked in this stage as well. If the address is a hit, the instruction completes in the next cycle, else it has to wait for the miss to be repaired, causing it to occupy the k2 function unit for 10 cycles.
3. **Note:** All the FU's are *pipelined*. This means that all FUs are available for ready instructions to be scheduled onto in each cycle of the processor simulation. Setup your scoreboard to take the pipeline behavior into account.

Status Update (last stage of Execute):

1. Completed instructions broadcast their results in this state. The register file, the branch predictor and the data cache 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. See notes slide ILP-25 ("Tomasulo algorithm, part 1")

Tag Generation:

Tags are generated sequentially for every instruction beginning with 0. The first instruction in a trace will use a value of 0, the second will use 1, etc. The traces are sufficiently short that you will not have to reuse tags.

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 Tag order. Consider a scenario where tags 100, 102, 110 and 104 are waiting to complete: The order of completion is tag 100, 102, 104 and 110, 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^G entry table of 2-bit bimodal Gshare predictor using the bimodal branch predictor where the initial state is 01 for each counter. The GHR (Global history register) is G bits wide. The instruction address is hashed to get an index into the Branch Target Buffer using this function: **`((Address >> 2) % (1 << G)) ^ <Value of GHR>;`** Note: '^' is the XOR function.

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 k0 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, it 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. Note that when you start the simulation, the smith counters are set to value ‘01’.

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

9. In Status Update, use the current GHR to update smith counters, and then update the GHR based on the branch real behavior.

Data cache:

To speed up load and store operations, you will be implementing a data cache that will be accessed in the EX stage. The cache should model the following behavior:

1. (C,B,S) define the cache where B is always 6, and S is always 0 (Direct mapped) C is a command line parameter.

2. The cache is accesses in the EX stage by Loads and Stores. The access address is given in the trace file as <LD/ST Addr>. Since we are simulating a cache, you need not model the data array of the cache.

3. The cache uses a Write Back / Write Allocate policy. This means that each block has a dirty bit. Note: Loads and Stores have different Opcodes in the given trace files.

4. All blocks are invalid when the simulation starts.

5. In order to realistically model cache operations, we limit the number of parallel cache accesses using the k2 function units as a proxy. An instruction that uses the cache can be fired only when k2 FUs are available, its sources are ready, and its destination is ready.

6. A cache hit completes in the next cycle (i.e. its latency is 1).

7. A cache miss on the other hand occupies the FU for 10 cycles. After which time, the cache is updated, and the instruction moves to the SU stage.

8. Note: Look at the Load / Store Reordering section for more details

Load and Store Reordering:

Recall that out-of-order loads and stores to the same address can cause errors in the program output. This problem is often called the memory disambiguation problem. In the Tomasulo pipeline the load/store queue is checked to ensure ordering. Since we are not explicitly modelling the load/store queue, here is what we will do instead:

- Loads can issue (i.e. proceed from reservation station to the Functional Unit) as long as there are no stores with same address before the load in program order (i.e. Check the FUs and make sure the reservation stations don't contain a preceding store to the same address)
- Stores issue if there are no loads or stores to the same address before the store
- Load and Store instructions wait in the Load / Store queue if the above conditions are not met.

To avoid load and store reordering's that might cause program bugs due to memory consistency violations, the traces have been modified to add 'fake dependencies' between loads and stores. That means that stores will have a 'fake' destination.

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
- Branch Prediction accuracy (Total number of branches / correctly predicted branches)
- Total load instructions
- Total store instructions
- Number of cache misses
- Data Cache miss rate (Total cache misses / Total cache accesses)
- Average Dispatch queue size
- Maximum Dispatch queue size
- Average instructions retired per cycle (IPC)
- Total run time (cycles)

Note: The cycle by cycle behavior of all cycles. That is print the PC value of each instruction that completes in the clock cycle. The format will look like:

```
<Cycle Count> <Instruction Address>
<Cycle Count> <Instruction Address>
. . .
```

Note: Match the exact format given in the verification logs using the `diff` command.

3. Validation Requirements

Your simulator must completely match the validation outputs that we will be providing you on Canvas before you move on to the next section. You must match the statistics and the output logs which print the cycle number in which each instruction completed.

4. Experiments

After your simulator is validated, for each trace you will be finding the optimum pipeline configuration for each trace. More details on this 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 but does not pass validation

+30% Your code passes all validation tests

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