CS 4290 / CS 6290 / ECE 4100 / ECE 6100
Advanced Computer Architecture

Lab 2: Dependency Tracking and Forwarding for 5-stage Superscalar Pipeline with Branch Prediction (10 Pts)

Part A Due: Friday, September 15th, 2023 (11:55 pm)
Part B Due: Friday, September 22nd, 2023 (11:55 pm)

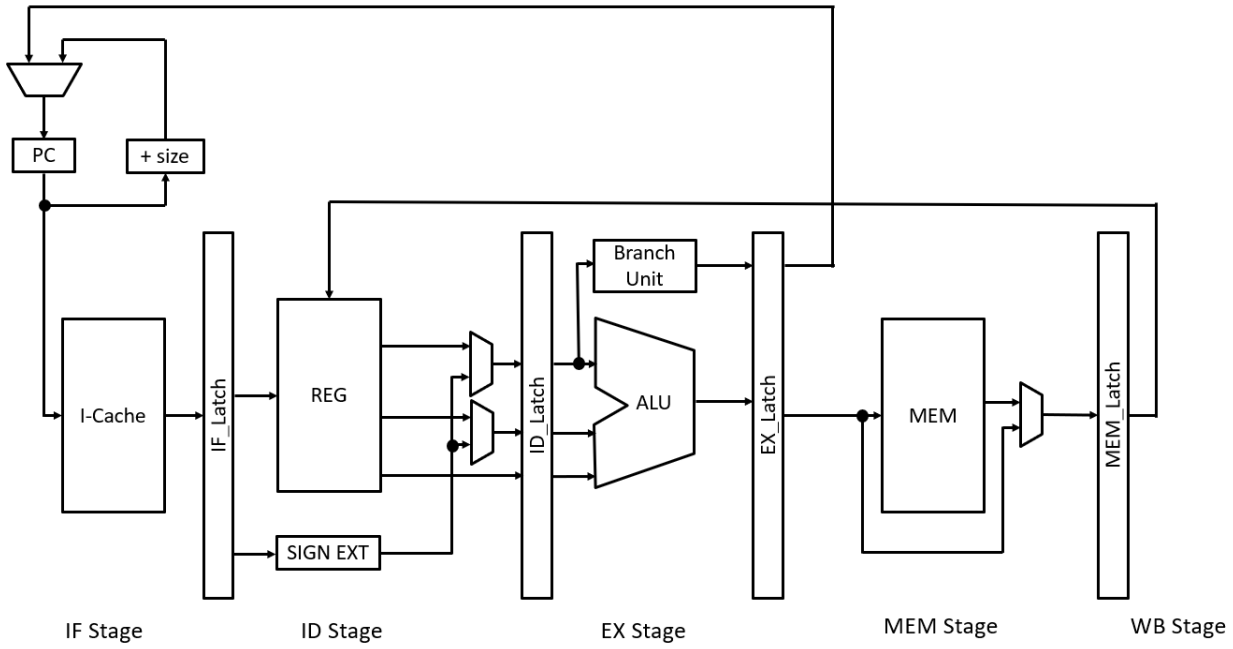

Figure 1: 5-stage pipeline example.

This is an individual assignment. You can discuss this assignment with other classmates, but you should code your assignment individually. You are NOT allowed to see the code of (or show your code to) other students. We will be using code similarity detection software. If you think you collaborated closely enough with another student that could lead to more code similarities than would normally be expected, please declare it upfront, at the time of submission. *Please note that such declaration is not granting you an excuse for copying code.*

OBJECTIVE

The objective of the second programming assignment is to evaluate the performance of a pipelined machine. In particular, you will equip the code to check data dependencies in a pipeline, implement a forwarding path, and extend your pipeline to be an "N-wide" superscalar machine. The second part of the assignment integrates a branch predictor with the superscalar pipeline.

PROBLEM DESCRIPTION

The in-order five-stage pipeline is a variant of the five-stage pipeline we discussed in class, as shown in the Figure above. It consists of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Writeback (WB) stages. In addition, this pipeline implements an ISA extension that supports a new type of ALU instructions with 3 source operands, such as ADD <dest_reg> <src1_reg> <src2_reg> <src3_reg>.

We will use a trace-driven simulator that is strictly meant for doing *timing* simulation to estimate the pipeline's achieved performance. To keep the framework simple, we will not be doing any *functional* simulation, which means the trace records that are fed to the pipelined machine do not contain any data values, and your pipeline will not track any data values (in Registers, Memory, or PC) either. Furthermore, the traces only contain the committed path of instructions. The purpose of our simulation is to figure out how many clock cycles it takes to execute the given instruction stream, for a variety of different microarchitectures such as **with or without forwarding** and **with a varying superscalar width (N).**

For this assignment, we will assume that the register file employs a write on falling edge, which means it is possible to write to the register file in the first half of the clock cycle and read from the register file in the second half of the clock cycle. Therefore, there is no need to stall an instruction in the decode stage if it has a RAW dependency with an instruction in the WB stage. Consequently, there is also no need to implement data forwarding from the WB stage to the ID stage.

You are provided with a trace reader, as well as a sample pipeline that simulates an N-wide superscalar machine, without any dependence tracking. Your objectives are the following:

Part A: Assume that the pipeline has perfect branch prediction and does not suffer any stalls due to control flow dependencies.

A.1 (2 points) Implement data dependency tracking and related stalls for a scalar machine (N=1).

A.2 (2 points) Generalize the above (A.1) to an N-wide superscalar pipeline. We will test for N=2, although your code should be general enough to work for any reasonable value of N. Note that for a superscalar machine you may have data dependencies not only from EX and MEM stages, but also from older instructions that are in the ID stage.
Tip: Remember that you are building an in-order processor; hence, if at any point in time a younger instruction is further ahead in the pipeline than an older instruction (which is perhaps stalled in the ID stage), your implementation has an error.

A.3 (2 points for CS6290/ECE6100, 3 points for CS4290/ECE4100) Implement Data Forwarding (from both MEM and EX). Note that the existence of a forwarding path does not necessarily mean that you can pass the value from a later instruction to an earlier instruction. For example, for a Load instruction, you would not have the produced value available until the MEM stage, so you cannot forward the value of Load from the EX stage to the ID stage for an instruction dependent on this Load instruction. We will test A.3 for N=2, although your program should work for any reasonable value of N.

Part B: Extend your pipeline to support Branch Prediction. For this part, we will assume that the machine has an idealized Branch Target Buffer (BTB), which identifies the conditional branches (CBR) as soon as the instruction is fetched and also provides the correct target address. Your objective is to consult the direction prediction on instruction fetch. If the prediction is correct, the fetch unit continues to fetch subsequent instructions, otherwise the fetch unit stalls until the branch resolves.
Tip: The reason you should stop fetching on misprediction is because the trace only contains instructions that are committed (i.e., instructions on the correct execution path), so you cannot fetch instructions on the wrong path and then flush them, as would happen in a real pipeline. Therefore, you will only simulate the performance effect of mispredicting the branch direction.

B.1 (1 point for CS6290/ECE6100, 2 points for CS4290/ECE4100) Implement an "AlwaysTaken" predictor and integrate it with your pipeline. We will evaluate your machine from A.3 with N=2.

B.2 (1 point) Implement a *gshare* predictor, shown in the following figure, with HistoryLength = 14 (use the bottom 14 bits of the PC to XOR with the bottom 14 bits of the Global History Register, GHR) and a PHT consisting of 2-bit counters, all initialized to the weakly taken state (10).

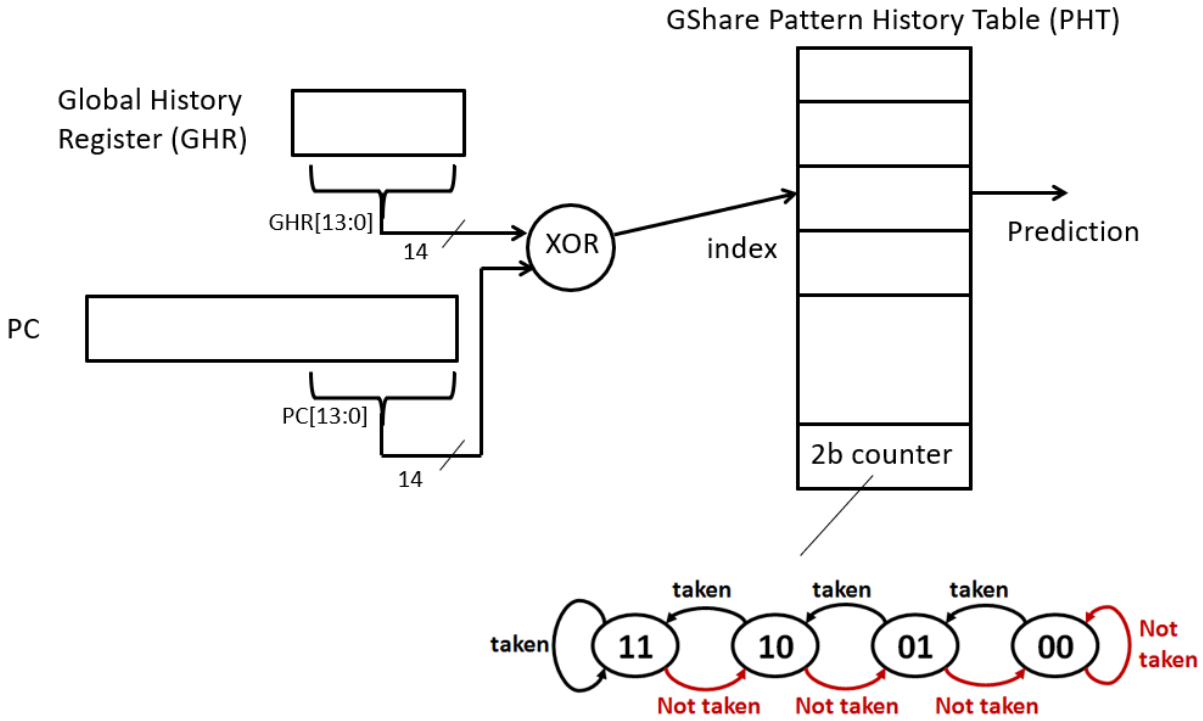

Figure 2: *gshare* branch predictor.

B.3 (1 point) [Required for CS6290/ECE6100. Optional for ECE4100/CS4290 (extra 1 point)] Implement a *gselect 7/7* predictor, shown in the following figure. The predictor has a configuration of HistoryLength = 14 (use the bottom 7 bits of the PC to concatenate with the bottom 7 bits of

the Global History Register, GHR) and a PHT consisting of 2-bit counters, initialized to the weakly taken state (10). Please note that the PC bits are followed by the GHR bits, i.e., index[13:7] = PC[6:0] and index[6:0] = GHR[6:0], as indicated in the figure.

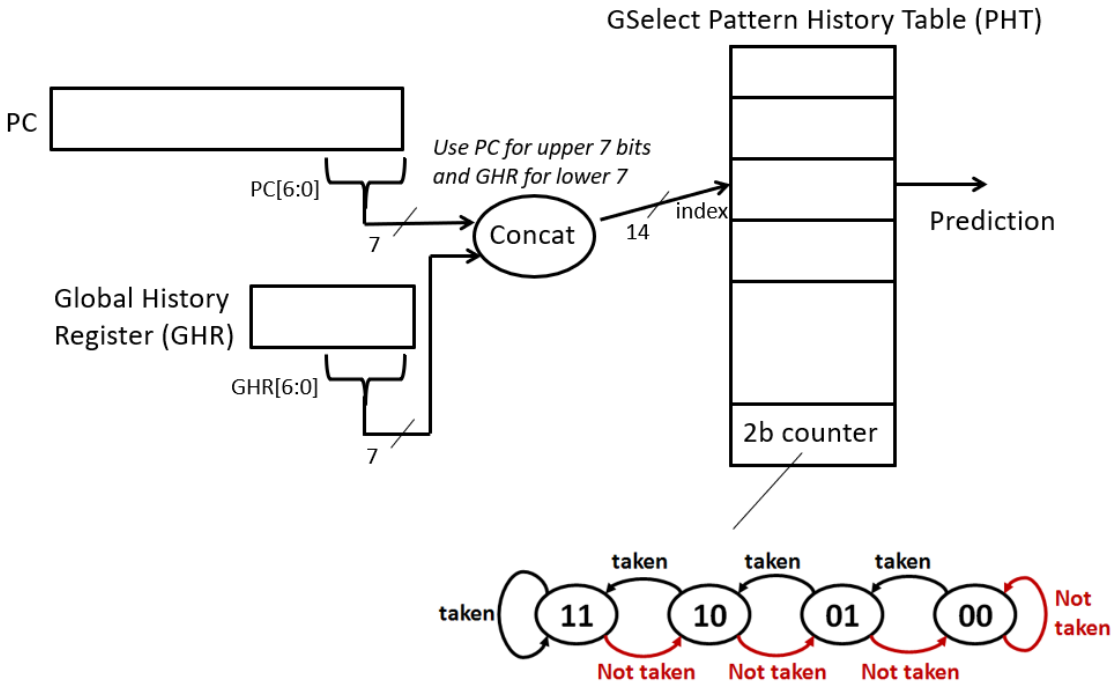

Figure 2: *gselect 7/7* branch predictor.

B.4 (1 point) [Required for CS6290/ECE6100. Optional for ECE4100/CS4290 (extra 1 point)] Implement a *tournament* predictor, shown in the following figure. The predictor has a HistoryLength = 10 (use the bottom 10 bits of the PC, PC[9:0]) and a PHT consisting of 2-bit counters, initialized to the weakly taken state (10). The tournament predictor will select the GShare Predictor's prediction if the counter is in the not taken state and will select the GSelect Predictor's prediction otherwise. When a branch is resolved, the GShare and GSelect predictors are updated normally. The Tournament PHT is updated only if the GShare prediction differs from the GSelect prediction, and the tournament counter shifts toward the direction of the predictor that matches the branch resolution.

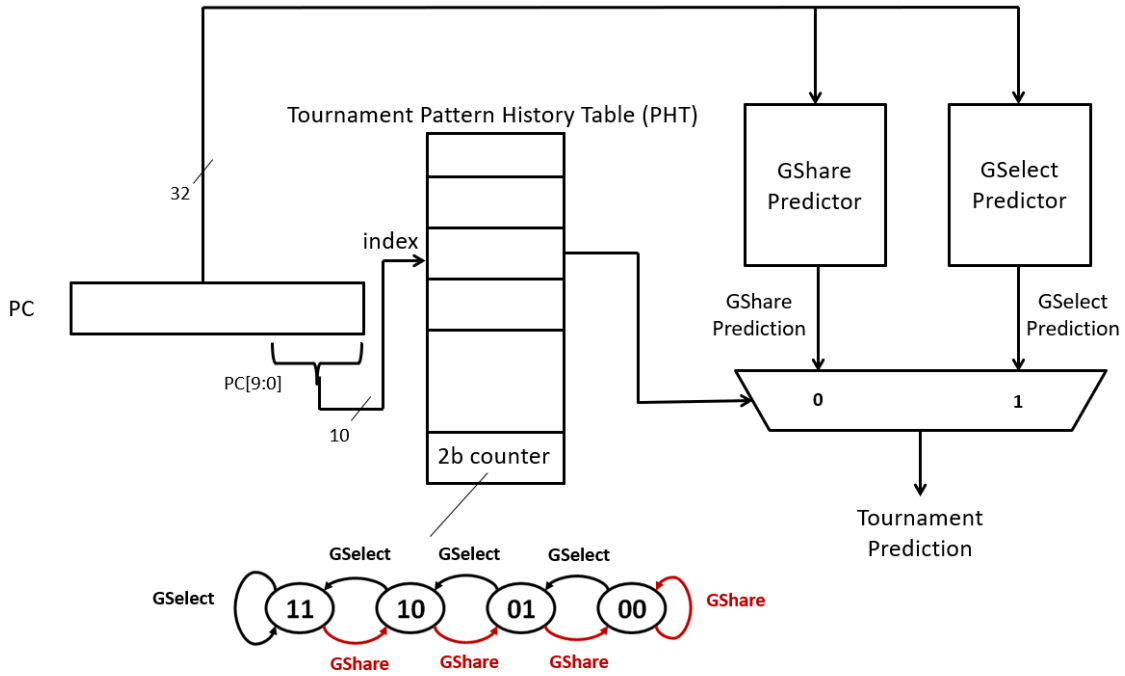

Figure 3: GShare-GSelect tournament branch predictor.

WHAT IS PROVIDED:

The simulator directory consists of sources and traces (note that these are different traces than Lab1, as we need to do dependency tracking). The src directory contains the source code that you will modify. The key files are the following:

1. sim.cpp and trace.h
   The sim.cpp file is responsible for opening the trace, initialization, instantiating and executing the pipeline until completion. The trace.h file serves a similar purpose as in Lab1, however it has a few additional fields needed for this assignment.

2. pipeline.cpp/.h
   These files contain the Pipeline class, internal structures and methods implementing the pipeline functionality. The simulator is a series of latches storing the operands on completion of the pipeline stages. You need to implement the functions pipe_cycle_IF() … pipe_cycle_WB() to accommodate pipeline functionality and dependency handling. You may add any additional structures you require. You also need to update the variables marked stat_* at the right conditions in the code.

3. bpred.cpp/.h
   These files contain the branch predictor interfaces. The interface contains only three functions: the predictor initialization function, a function for getting the predicted value, and a function for updating the predictor. You need to implement these functions according to the branch prediction policy used in each case.

It is strongly recommended that you go through all files carefully and make sure you understand the simulator's structure before you proceed to write any code.

How to run the simulator:

1) Download the zipfile and type "unzip Lab2.zip"
2) Type "cd Lab2/src"
3) Type "make"
4) Type "./sim -h" for command line options (pipewidth, bpredpolicy, etc.)
5) "./sim ../traces/mcf.ptr.gz" (to test the current pipeline for N=1)
6) "./sim -pipewidth 2 ../traces/mcf.ptr.gz" (to test the current pipeline for N=2)

For Part A, you need to modify the pipe_cycle_* functions in pipeline.cpp. For Part B, you need to add the data structures in bpred.h, functionality in bpred.cpp, and the pipe_check_bpred function in pipeline.cpp. You will also need to implement the stall of fetch on branch mispredictions and release the stall when the branch resolves (when the branch is in the MEM stage; however, you can fetch only in the next cycle).

We have provided a script called **runall.sh** that can run all of the experiments (A1, A2, A3, B1, B2, B3, B4) for all the four traces in the trace directory in a single invocation. This script is located in the Lab_2/scripts directory. You may need to do "chmod +x runall.sh" before running this script.

Reference output for testing:
We provide an additional small trace (*sml*) in the *traces* directory, along with a sample expected output for it in the *reference* directory, as a point of comparison for the outputs produced by your solution. The *reference* directory also includes a reference output for the gcc trace.

Local and Gradescope autograders:
We provide a Python script (*grade.py*) in the *scripts* directory to generate a score based on the reference outputs for the small and gcc traces. Compile the project and run *python3 grade.py* for the CS6290/ECE6100 score, and *python3 grade.py -u* for the CS4290/ECE4100 score. A simple autograder on Gradescope runs a similar script after submitting the required files. Please note that these autograders are not comprehensive and your code will be tested against additional traces.

WHAT to SUBMIT (on Gradescope):

For Part A

- pipeline.cpp
- bpred.cpp (implementation not required for bpred.cpp in Part A)

We will provide a sample solution for Part A after its deadline. Please use this (or your original one if it is correct) to implement Part B.

For Part B

- pipeline.cpp

- bpred.cpp

Note for CS4290/ECE4100 students: You are not required to do B.3 and B.4. However, you can still choose to do them for Extra Credit worth 1 point each.

REFERENCE MACHINE:
We will use the virtual machine **oortcloud.cc.gatech.edu** as a reference machine**.** You should be able to connect to it using ssh and transfer files between this machine and your local machine using scp, following the same steps as in Lab 1.

Before submitting your code ensure that your code compiles on this machine and generates the desired output (only the output produced by the **print_stats** function, without any extra printf statements to receive full credit). Please follow the submission instructions.

NOTE: It is impractical for us to support other platforms such as Mac, Windows, Ubuntu, etc.

FAQ:

1. **Reason for Instruction Address not being unique in the Trace**
   During Trace generation, the complex x86 instructions having multiple operations at a particular address were converted to simpler operations having the types provided in the trace header file. These simpler instructions would then have the same instruction address. The instruction address is thus not a unique identifier for an operation. Instead, op_id is supposed to be used for that.

2. **Data Forwarding for operations with conditional codes or belonging to the OTHER op-type with destination**
   Handling data forwarding for the above is similar to the handling of ALU operations. Load instructions having cc_write can only forward their conditional codes in the MEM stage.

3. **Meaning of *_needed fields in the trace structure**
   These are binary 1 / 0 values, informing whether src1, src2, src3, and dest fields are valid in an operation read from the trace file. If these are '1', the corresponding values in the src1, src2, src3, and dest fields represent the register being read from or written to.

4. **Meaning of cc_read and cc_write**
   Consider the following operation
   if (condition operation)
   The condition operation implicitly writes to a condition 'status' register. Such an operation would have cc_write set to 1. The following branch instruction based on the condition would have the cc_read set to 1.
   cc_read and cc_write are 1 / 0 values. Only branches perform a cc_read. The reading takes place in the Instruction Decode stage, similar to the source register values
   (Refer: http://en.wikipedia.org/wiki/Status_register).

5. What is the pipe_print_state function in pipeline.cpp?

This function is provided as a helpful tool for debugging purposes, and you can choose to use it while developing your solution. **Please make sure you do not have a call to this function in your submitted solution.** Leaving this on will result in additional printed statements, which, as mentioned earlier, will result in you not receiving full credit.

6. Why are the five pipeline stages invoked in reverse order?

You will realize that doing so simplifies flow control and handling backpressure. In a pipeline, if stage n needs to stall, all previous stages will also need to stall. Therefore, it is more practical to move things forward from the end towards the start of the pipeline.