ECE4100/6100 CS4290/6290 - Fall 2024
Lab 2: Superscalar Pipeline with Branch Prediction

Dr. Moinuddin Qureshi

Version: 1.0

Part A Due: September 13th 2024 @ 11:59 PM ET
Part B Due: September 20th 2024 @ 11:59 PM ET

Changelog

- Version 1.0 (8/27/2024): Initial release.

1 Rules

- **This is an INDIVIDUAL assignment.** You may discuss this assignment with classmates, but you should code your assignment individually (i.e. no code sharing). **All honor code violations will be reported to the Dean of Students. NO EXCEPTIONS.**
- These lab assignments are very difficult and take a lot of time. It is in your best interest to start early.
- See the "Late Policy for Assignments" Canvas announcement for the course's late policy.
- Please utilize TA office hours, Piazza, and recitation if you have questions. **Do not go to the professor's office hours with lab-specific questions.**
- Read the entire document before starting. Not only is it critical to understanding the assignment, but most questions can be answered by reading the corresponding section.
- Sometimes, mistakes will be found in the PDF and the lab assignment. **It is solely your responsibility to ensure you are using the most up-to-date PDF and lab files.**
- Make sure that your code works with C++11 and on the provided reference machines as this what we will run the autograder on.

2 Introduction

The objective of this lab is to do a performance evaluation 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 deals with integrating a branch predictor with the superscalar pipeline.

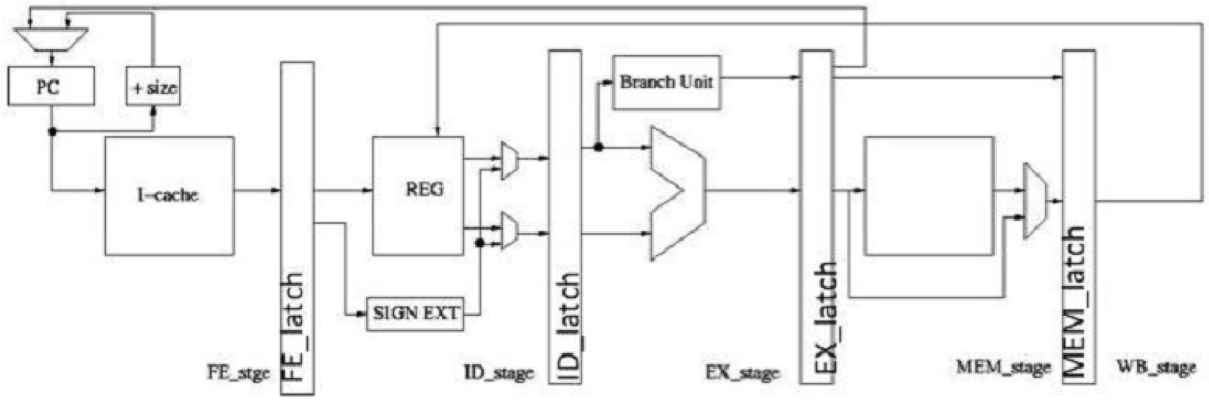

Figure 1: Five-stage pipeline

The five-stage pipeline that we discussed in class is shown in the figure above, and it consists of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Write-back (WB) stages. 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 do data forwarding from the WB stage to the ID stage. We will use a trace driven simulator that is strictly meant for doing timing simulation. To keep the framework simple, we will not be doing any functional simulation. This 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 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 machines such as with or without forwarding and varying superscalar width (N).You will be provided with a trace reader, as well as a sample pipeline machine that simulates an N-wide superscalar machine, without any dependence tracking.

3 Design

Part A: You will be provided with a trace reader, as well as a sample pipeline machine that simulates an N-wide superscalar machine, without any dependence tracking. Your job is to do the following:

- **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 a N wide super-scalar (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.
- **A.3 (2 points for ECE6100/CS6290, 3 points for ECE4100/CS4290)** Implement Data Forwarding (from both MEM and EX). Note that an 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 value available until the MEM stage, so you cannot forward the value of Load from 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 job is to consult 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 (i.e. reaches WB). Since part B is contingent upon having a working part A, if your part A does not work after the part A deadline we will provide you a solution for part A.

- **B.1 (2 points for ECE6100/CS6290, 3 points for ECE4100/CS4290)** Implement an "AlwaysTaken" predictor, and integrate it with your pipeline. We will evaluate your machine from A.3 (with N=2)
- **B.2 (2 points)** [Required for ECE6100/CS6290. Optional for ECE4100/CS4290] Implement a gshare predictor, shown in the following figure,with HistoryLength=12 (we will assume that you use the bottom 12 bits of the Instruction Address to XOR with the Global History Register, GHR) and a PHT consisting of 2-bitcounters, initialized to the weakly taken state (10)

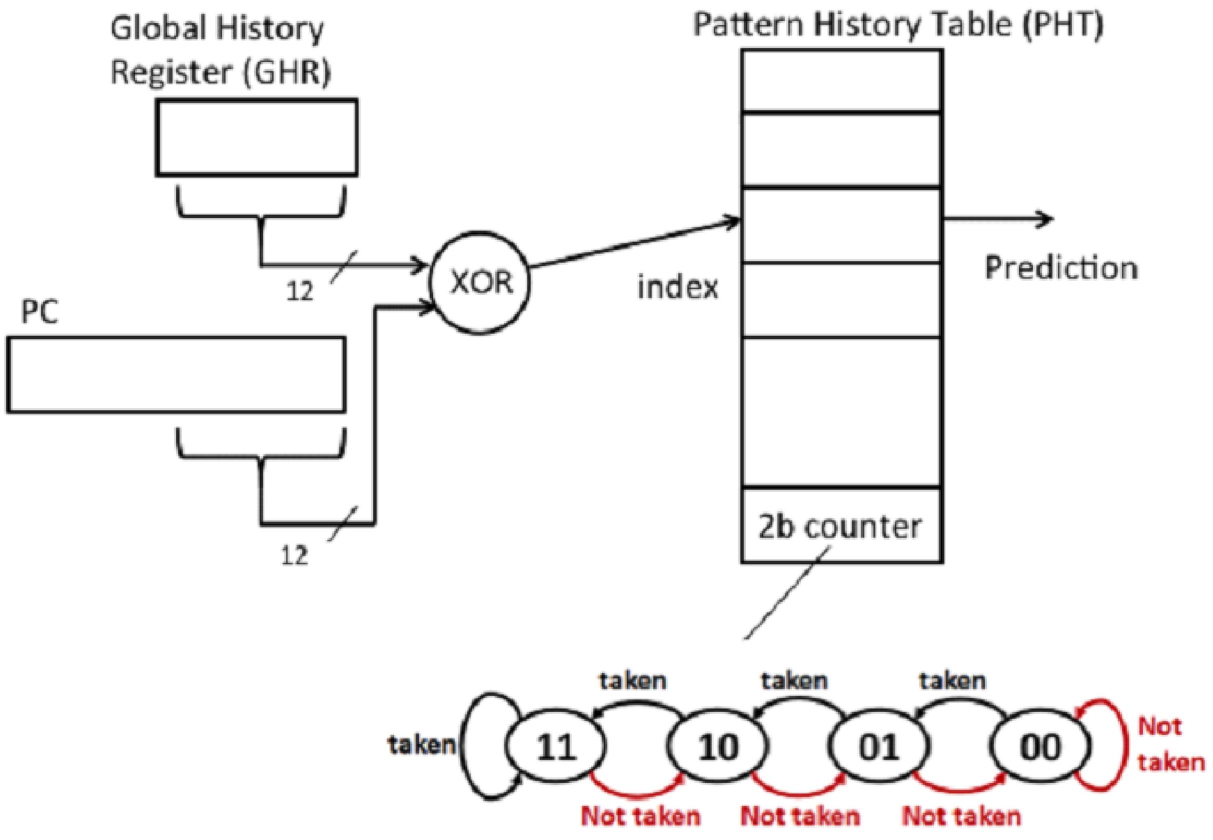

Figure 2: Figure 2: GShare branch predictor

4 Implementation Details

You have been provided the following files:

- `pipeline.cpp` - **(Part A & B)** The functions `pipe_cycle_IF()`, `pipe_cycle_ID()`, `pipe_cycle_EX()`, `pipe_cycle_MA()`, and `pipe_cycle_WB()` need to be implemented for providing pipeline functionality and handling of dependencies.

- `pipeline.h` - **Do not modify.** Header file containing structs and definitions necessary for the simulator.

- `sim.cpp` - **Do not modify.** Responsible for opening the trace, initialization, and instating and executing the pipeline.

- `trace.h` - **Do not modify.** Header file containing structs and definitions pertaining to instructions from the traces.

- `bpred.cpp` - **(Part B)** Where your branch prediction implementation will go; functions for getting the predicted value and for updating the predictor will need to be implemented.

- `bpred.h` - **(Part B)** Header file containg structs and definitions necessary for the branch predictor. `Makefile` - **Do not modify.** A Makefile for compiling your code. Can be used to invoked the following commands...
  - `make` and `make all`: Compiles your code and creates an output file so you can run your code.
  - `make clean`: Cleans out compiled files.
  - `make fast`: Compiles your code with the `-O2` flag. See this for details.
  - `make debug`: Compiles your code with the preprocessor definition DEBUG defined. This causes code blocks of `#ifdef DEBUG ... #endif` to compile. **Highly recommend using for debugging purposes!**
  - `make profile`: Compiles your code for use with `gprof`. Helps diagnose bottlenecks if your code is running slowly.
  - `make validate`: Compiles your code and runs `runtests.sh`.
  - `make runall`: Compiles your code and runs `runall.sh`
  - `make submit`: Creates a tarball for your submission. Run `make validate` beforehand to generate `report.txt`. DO NOT change the name of the "src" directory, or your submission will not run with the autograder!
- `traces/` - **Do not modify.** A directory containing the execution traces for this lab.
- `scripts/` - **Do not modify.** A directory containing the following scripts...
  - `runall.sh`: Runs your code on the provided traces. The results of each run is placed in the correspondingly-titled `.res` file in `results/`. A summary of all the results can be found in `report.txt`, which is placed in whichever directory `runall.sh` is ran from (e.g. if you call `make runall` the report will be put in `src/`).
  - `runtests.sh`: Runs your code on a subset of the provided traces and verifies that your output statistics match the reference statistics for each test.
- `results/` - **Do not modify.** A directory containing the output of your code from `runall.sh`.
- `ref/` - **Do not modify.** A directory containing the reference outputs for a subset of the provided traces.
  - `results/`: A directory containing the reference outputs for a subset of the provided traces for each individual part of the lab.
  - `*.pdf`: Reports that summarize the reference outputs.

4.1 Simulator Parameters

The following command line parameters can be passed to the simulator:

- `-pipewidth <width>`: The number of pipelines to simulate (1 by default)
- `-enablememfwd`: Enables forwarding from the Memory Access (MA) stage (off by default)
- `-enableexefwd`: Enables forwarding from the Execute (EX) stage (off by default)
- `-bpredpolicy <num>`: The branch predictor your simulator will use. There are 3 options:
  - `0`: Perfect branch prediction (default)
  - `1`: Always taken
  - `2`: Gshare
- `-h`: Print usage information

Here's an example of running your code on a single trace using these simulator parameters:

```
./sim -pipewidth 2 -enablememfwd -enableexefwd -bpredpolicy 2 ../traces/sml.ptr.gz
```

4.2 Simulator Statistics

The simulator keeps track of a variety of statistics, some of which you may have to update:

- `stat_retired_inst`: Counts the number of committed instructions.
- `stat_num_cycle`: Counts the number of simulated CPU cycles.
- `cpi`: Defined as `stat_num_cycle / stat_retired_inst`.
- `stat_num_branches`: The total number of branches the branch predictor has seen.
- `stat_num_mispred`: The number of branches the branch predictor mispredicted.

5 Debugging & Debug Outs

For debugging purposes, the Makefile has an option called `make debug`. This allows code enclosed in `#ifdef DEBUG ... #endif` blocks to compile alongisde your other code. For instance, say I have the following...

```
int x = 2;
#ifdef DEBUG
    x = 4;
#endif
printf("%d", x);
```

If you ran this with `make` by itself the output would be `2`. However, if you run it with `make debug` the output would be `4`. If we were to replace this example with `printf` statements, then we would only ever output when we call `make debug`!

Additionally, since debugging can be cumbersome we have provided debug outputs for you on Canvas. These debug outs are made by collecting the first X lines (indicated by the number on the debug outs folder) of the solution, then putting the output into a text file. In order to match these `debug_outs`, you **must** must add the `printf` statements provided in `debug_printfs.txt` to your code. The parameters for these statements have been purposefully left blank; you have to fill them in on your own depending on your implementation. When adding the `printf`'s, be sure to wrap them in `#ifdef DEBUG` blocks so they only print when you call `make debug`.

To check your implementation using the debug outs, edit your code according to the type you're using. Then, run your code with the trace and the simulator parameters you want. Then, set your output to only go to X lines and set the destination of your output to some file. For example (with 100,000 lines):

```
./sim <necessary flags> ../traces/gcc.ptr.gz | head -n100000 > gcc.my_debug_out.res
```

After running the above, you can compare your output to the provided debug out's output using `diff`...

```
diff gcc.my_debug_out.res <provided gcc trace> > diff.txt
```

This will create a file `diff.txt` and place the output of the `diff` command inside it. At this point, you can go to the line numbers of your `.res` files indicated in `diff.txt` to see what is going wrong and fix it. If you run the `diff` command above and `diff.txt` is empty, you should call `make validate` to see if your results match!

6 Deliverables

To prepare your submission, run `make submit` to generate a tarball (`tar.gz`) with your code. **Do NOT change the name of the "src" directory**, as the autograder will not be able to test your submission. Make sure to untar your submission to ensure all the required files are there before submitting. **You are solely responsible for what you submit.** If you submit improperly, you can face up to a 10% grade deduction. Please do not make us use this rule and submit properly.

To see if you can receive full credit, you should run `make validate` and ensure you are passing. Note that just because you pass `make validate`, this does **NOT** mean you get a 100%. We reserve the right to examine any submission if we suspect plagiarism or attempts to game the autograder in any way.

7 Grading

For both parts in total, both ECE4100/CS4290 and ECE6100/CS6290 students are graded out of 10 points. Note that this project is 10% of your final grade, meaning that each point equates to one point towards your final grade. Also note that part B2 is optional for ECE 4100 students, and is worth 2 points.

7.1 Part A (6 points [ECE6100/CS6290], 7 points [ECE4100/CS4290])

Your submission is evaluated based on how your simulator's CPI compares to the reference CPI for `bzip2`, `gcc`, `libq`, and `mcf`. If your CPI is within 1% of the reference CPI, you get full credit. If it is within 5%, you get half credit. The following is how many points you get **per trace**:

- A.1 (2 points)
  - **CPI**: 0.5 (within 1%), half-credit for within 5%, 0 otherwise
- A.2 (2 points)
  - **CPI**: 0.5 (within 1%), half-credit for within 5%, 0 otherwise
- A.3 (2 points for ECE6100/CS6290, 3 points for ECE4100/CS4290)
  - **CPI**: 0.5 or 0.75 (within 1%), half-credit for within 5%, 0 otherwise

7.2 Part B (4 points [ECE6100/CS6290], 3 + 2 points [ECE4100/CS4290])

Your submission is evaluated based on how your simulator's CPI and misprediction rate compare to the reference CPI and misprediction rate for `bzip2`, `gcc`, `libq`, and `mcf`. For both your CPI and misprediction rate, they are calculated separately. If they are within 1% of the reference CPI, you get full credit for that calculation. If it is within 5%, you get half credit **for only the part B1 calculation.** There is no half-credit awarded in part B.2. The following is how many points you get **per trace**:

- B.1 (2 points for ECE6100/CS6290, 3 points for ECE4100/CS4290)
  - **CPI**: 0.25 (within 1%), half-credit for within 5%, 0 otherwise
  - **Misprediction Rate**: 0.25 (within 1%), half-credit for within 5%, 0 otherwise
- B.2 (2 points for ECE6100/CS6290, 2 extra credit points for ECE4100/CS4290)
  - **CPI**: 0.25 (within 1%), 0 otherwise
  - **Misprediction Rate**: 0.25 (within 1%), 0 otherwise

While there is no strict efficiency requirement, please ensure your simulator finishes in less than a minute. This is to ensure expedient grading, since otherwise verification would be difficult and tedious. If your simulator is taking too long, we recommend you profile your code to see where most your code spends its time running. You can use `make profile` to allow your code to be used with `gprof`.

8 Reference Machines

The autograder will run on a reference machine to ensure that everyone conforms to one standard. There are two reference machines you can use for the course:

- **ECE**: ece-linlabsrv01.ece.gatech.edu (email)
- **CS**: shuttle3.cc.gatech.edu (help form)

How to access these machines depends on whether or not your are using Georgia Tech Wi-Fi (`eduroam`). If you are not, then you need to install the Georgia Tech VPN client (see Remote Access → VPN or Virtual Private Networking). Once you are connected either via VPN or Wi-Fi, you should be able to `ssh` into one of the above servers via the command line of your operating system like so:

```
ssh gburdell3@ece-linlabsrv01.ece.gatech.edu
```

Please note that `gburdell3` should be replaced with *your* GT username. If you receive `Permission denied` or `Access denied` when trying to connect, then use the above links to request access. Please include your GT username, the course name (ECE 4100/6100), and the instructor in the request.

We highly recommend that both CS and ECE students obtain access to each other's servers as these servers have been shown to go down at times. You can contact the respective help desks to request access. Since the servers can go down, it is crucial that you do not exclusively work on the servers while working on these projects. We recommend working on your code locally using services like WSL that will simulate a Linux environment for you to work and verify your implementation in.

9 FAQ

1. **How should I get started?**
   We strongly recommend that you read through the header files first to get a sense of what data structures are available to you and what you must implement. The header files should provide documentation foreverything you need to know to complete this lab.

2. **Why aren't instruction addresses 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. (op_id is supposed to be used for that)

3. **How do I implement Data Forwarding for operations with conditional codes or operations belonging to the OTHER op_type with a destination register?**

Handling the data forwarding for the above is similar to the handling for ALU operations. Load instructions having cc_write can only forward their conditional codes in the MEM stage

4. **What are the *_needed fields in the trace structure?**
   These are binary 1 -0 values, informing whether src1_reg, src2_reg and dest_reg fields are valid in an operation read from the trace file. If these are '1' the corresponding values in the src1_reg, src2_reg,and dest_reg fields represent the register being read from or written to.

5. **What are cc_read and cc_write?**
   Consider the following operation:
   if (condition operation)
   The condition operation 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 would perform a cc_read. The reading takes place in the Instruction Decode stage, similar to the source register values (Refer to: http://en.wikipedia.org/wiki/Status_register)

6. **What is pipe_print_state()?**
   This function allows you to print the state of the pipeline at any time for debugging purposes. If you use it when debugging, please remove any calls to it before you submit your work. Submitting your code with calls to pipe_print_state() will result in extraneous output, which may result in you not receiving full credit.

7. **How can I test my code?**
   Reference outputs for gcc.ptr.gz and sml.ptr.gz for all five parts of the lab are provided in the ref directory as refoutput_gcc.pdf and refoutput_sml.pdf, respectively. You can run the script runtests.sh located in the scripts directory to compare your implementation's output with these reference outputs.

8. **I get an error when I try to execute the runall.sh script or the runtests.sh script.**
   Check that you have execute permissions on both scripts. Add execute permissions to these files using the command: `chmod +x runall.sh runtests.sh`

9. **What do I do if I get a `no space left on device` error?**
   You can run `du -sh *` to see what is taking up space in your current directory and use `rm` and `rm -r` to remove files and directories respectively. If you are running `./runall.sh`, ensure that you have cleared out the `results/` directory as the files here can be very large if you ran `./runall.sh` with your print statements.

10. **Can I change header files?**
    You can add your own functions or class attributes, or class methods to the header file. However, please do not modify the existing signatures, and ensure that everything compiles successfully.

Appendix - Plagiarism

Preamble: The goal of all assignments in this course is for you to learn. Learning requires thought and hard work. Copying answers thus prevents learning. More importantly, it gives an unfair advantage over students who do the work and follow the rules.

1. As a Georgia Tech student, you have read and agreed to the Georgia Tech Honor Code. The Honor Code defines Academic Misconduct as "any act that does or could improperly distort Student grades or other Student academic records."

2. You must submit an assignment or project as your own work. Absolutely no collaboration on answers is permitted. Absolutely no code or answers may be copied from others — such copying is Academic Misconduct. NOTE: Debugging someone else's code is (inappropriate) collaboration.

3. Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "*submission of material that is wholly or substantially identical to that created or published by another person*").

4. Publishing your assignments on public repositories accessible to other students is unauthorized collaboration and thus Academic Misconduct.

5. Suspected Academic Misconduct will be reported to the Division of Student Life Office of Student Integrity. It will be prosecuted to the full extent of Institute policies.

6. Students suspected of Academic Misconduct are informed at the end of the semester. Suspects receive an Incomplete final grade until the issue is resolved.

7. We use accepted forensic techniques to determine whether there is copying of a coding assignment.

8. If you are not sure about any aspect of this policy, please ask Dr. Mahajan