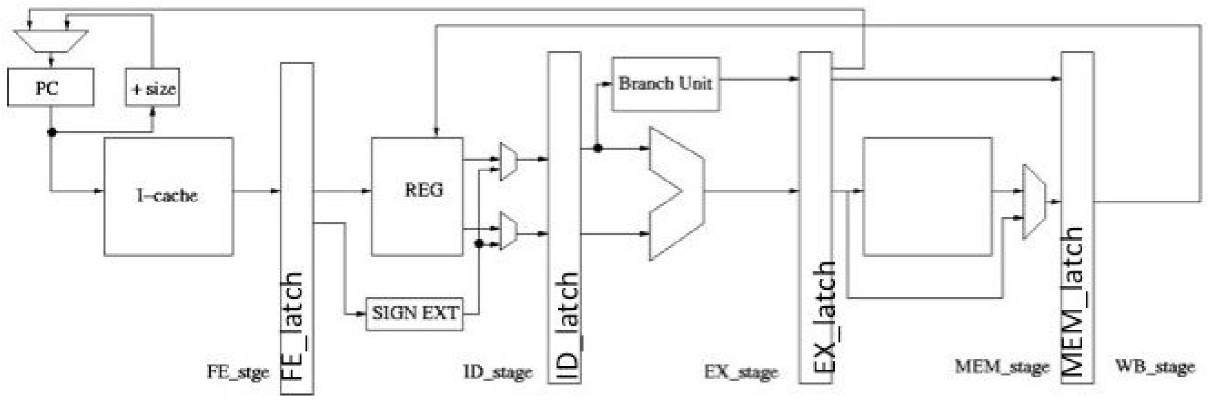
# Lab 2: Dependency Tracking and Forwarding for 5-stage Superscalar Pipeline with Branch Prediction (10 Pts)



**Due:** Part A due on Sep 17 (11:59pm) Part B due on Sept 24 (11:59pm)

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.

**OBJECTIVE**

The objective of the second programming assignment 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 forwarding path, and extend your pipeline to be an “N-wide” superscalar machine. The second part of the assignments deals with integrating a branch predictor with the superscalar pipeline.

**PROBLEM DESCRIPTION**

The five stage pipeline that we discussed in class is shown in the Figure above, and it consists of FETCH (FE), Instruction Decode (ID), Execute (EX), Memory (MEM) and Writeback (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 -- which means the trace records that is fed to the pipelined machine does not contain any data values, and your pipeline will not track any data values (in REG, 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. Your job is to do the following:

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

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

**A.2** 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** 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.

**B.1** Implement “AlwaysTaken” predictor, and integrate it with your pipeline. We will evaluate your machine from A.3 (with N=2)

**B.2** Implement a Gshare Predictor, 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-bit counters, initialized to the weakly taken state (10).

**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 as follows:

1. sim.cpp and trace.h
   The sim.cpp file is responsible for opening the trace, initialization, instantiating and executing the pipeline till completion. The trace.h file serves 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. The functions pipe_cycle_FE() … pipe_cycle_WB() need to be implemented by the students for providing pipeline functionality and handling of dependencies. Any additional structures required by students can be added. Students 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 two functions, one for getting the predicted value and another for updating the predictor. Students need to implement these functions as per the branch prediction policy.

**How to Run the Simulator:**
1) Download the tarball and type “tar -xvzf Lab_2.tar.gz”
2) Type “cd Lab_2/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 implementing Part A, your job is to modify the pipe_cycle_?? functions in pipeline.cpp.
For implementing Part B, you will 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,B3) for all the four traces in the trace directory in one shot. This script is located at Lab_2/scripts directory. If you are unable to execute this script you may need to do “chmod 755 runall.sh” before running this script.

**WHAT to SUBMIT (on Tsquare):**

A. For Part A, pipeline.cpp (you are not allowed to change functions in any other file)
B. For Part B, you can tarball the src directory and upload as src.tar.gz (no traces please). We will also provide a script to generate the report to be submitted.

For Part A and B, you can tarball the src directory and upload as src.tar.gz (no traces please) and the report.txt.

**Note for ECE4100/CS4290 students:** You are not required to do B.2. However, you can still choose to do it for Extra Credit worth 2 points.

**REFERENCE MACHINE:**
**ECE STUDENTS:** We will use **ecelinsrv7.ece.gatech.edu** as the reference machine for this course. (http://www.ece-help.gatech.edu/labs/unix/names.html).

**CS STUDENTS:** We will use **shuttle3.cc.gatech.edu** as the reference machine for this course. (https://support.cc.gatech.edu/facilities/general-access-servers)

Before submitting your code ensure that your code compiles on this machine, and generates the desired output (without any extra printf statements). Please follow the submission instructions. If you do not follow the submission file names, you will not receive the full credit.

**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. (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 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.
3. **Meaning of *_needed fields in the trace structure**
These are binary 1 - 0 values, informing whether src1, src2 and dest fields are valid in an operation read from the trace file. If these are '1' the corresponding values in the src1, src2 and

dest field 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 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 : http://en.wikipedia.org/wiki/Status_register )