CS 3330 Computer Architecture, Spring 2020
Lab 03: Simulating a Branch Predictor

Instructor: Prof. Samira Khan
TAs: Amel Fatima, Sihang Liu, Korakit Seemakhupt, Yasas Senerivathne, Yizhou Wei,
Min Jae Lee, Nikita Semichev, Yuying Zhang.

Assigned: Wed, 3/25, 2020
Due: **Fri, 4/03, 2020**

1. Introduction

In this lab, you will implement a *timing simulator* (written in C) to model a branch predictor. A *timing simulator* is not a direct or synthesizable implementation of the processor but a higher-level abstracted model that is designed to allow quick exploration of the system. Operating at a higher level allows the designer to quickly see how design choices (e.g., branch predictors) would impact performance.

We will give you the base simulator (it has been developed to match the processor that we have implemented in previous labs). We will also fully specify the behavior of the branch predictor. Your job is to extend the simulator so that it implements the branch predictor as specified.

2. Timing Simulator

Our goal in this lab is to compute the *number of cycles* a program execution would take on the simulated processor. Because of this, many simplifications are possible. We do not actually need to model control logic and datapath details in each block of the processor; we only need to write code for each stage that performs the relatively high-level function of that pipeline stage (read the register file, access memory, etc.). In general, the simulator's algorithms and structures *do not* need to exactly match the processor's algorithms and structures, as long as the *result* is the same.

3. Microarchitectural Specifications

3.1. Branch Predictor

Your goal is to *implement* the timing simulator so that it models a MIPS machine with a branch predictor. Below, we have fully specified the microarchitecture of the MIPS machine that you will simulate.

Organization. The *branch predictor* consists of *(i)* a *gshare* predictor and *(ii)* a *branch target buffer*.

Gshare. The gshare predictor uses an **8** bit global branch history register (GHR). The most recent branch is stored in the **least-significant-bit** of the GHR and a value of **'1'** denotes a taken branch. The predictor XORs the GHR with bits `[9:2]` of the PC and uses this 8 bit value to index into a **256-entry** pattern history table (PHT). Each entry of the PHT is a **2 bit** saturating counter that operates as discussed in class: a taken branch increments whereas a not-taken branch decrements; the four values of the counter correspond to strongly not-taken (00), weakly not-taken (01), weakly taken (10), strongly taken (11).

Branch Target Buffer. The branch target buffer (BTB) contains **1024 entries** indexed by bits `[11:2]` of the PC. Each entry of the BTB contains *(i)* an address tag, indicating the full PC; *(ii)* a valid bit; *(iii)* a bit indicating whether this branch is *unconditional*; and *(iv)* the target of the branch.

Prediction. At every fetch cycle, the predictor indexes into both the BTB and the PHT. If the predictor misses in the BTB (i.e., address tag != PC or valid bit = 0), then the next PC is predicted as **PC+4**. If the predictor hits in the BTB, then the next PC is predicted as the **target** supplied by the BTB entry when either of the following two conditions are met: *(i)* the BTB entry indicates that the branch is unconditional, or *(ii)* the gshare predictor indicates that the branch should be taken. Otherwise, the next PC is predicted as **PC+4**.

Update. The branch predictor structures are always updated in the execute stage, where all branches are resolved. The update consists of: *(i)* updating the PHT, which is indexed using the current value of the GHR *(ii)* updating the GHR, and *(iii)* updating the BTB. Unconditional branches *do not* update the PHT or the GHR, but only the BTB (setting the *unconditional* bit in the corresponding entry.

Initial State. All branch predictor structures are initialized to 0.

3.2. Flushing the Pipeline

When resolving a branch, the pipeline is flushed under any of the following conditions:

- The instruction is a branch, but the predicted direction does not match the actual direction.
- The instruction is a branch, and it is taken, but the predicted destination (target) does not match the actual destination.
- The instruction is a branch, but it was not recognized as a branch (i.e., BTB miss).

4. Lab Resources

4.1. Source Code

The source code is available at: "`http://www.cs.virginia.edu/~smk9u/CS3330S20/Lab3.zip`". Do NOT modify any files or folders unless explicitly specified in the list below.

- `Makefile`
- `refsim`: Reference simulator in machine-executable format
- `verify`: Script that compares your simulator against the reference simulator
- `src/`: Source code (**Modifiable; feel free to add more files**)
  - `pipe.c`: Your simulator (**Modifiable**)
  - `pipe.h`: Your simulator (**Modifiable**)
  - `bp.c`: The actual implementation (definition) of your branch predictor functions (**Modifiable**)
  - `bp.h`: The declaration of your branch predictor functions (**Modifiable**)
  - `mips.h`: MIPS related pound defines
  - `shell.c`: Interactive shell for your simulator (similar to Lab 1)
  - `shell.h`: Interactive shell for your simulator (similar to Lab 1)
- `exampleinputs/`: Example test inputs for your simulator
- `inputs/`: Your custom test inputs (**Modifiable; feel free to add more files**)

4.2. Makefile

We provide a `Makefile` that automates the compilation and verification of your simulator.

To compile your simulator:

```
$ make
```

To compile your simulator and verify it against the reference simulator using one or more test inputs:

```
$ make verify INPUT=exampleinputs/inst/addiu.x
$ make verify INPUT=exampleinputs/inst/*.x
$ make verify
```

5. Getting Started

5.1. The Goal

We provide you with a skeleton of the timing simulator that models a five-stage MIPS pipeline: `pipe.c` and `pipe.h`. As it is, the simulator is already architecturally correct: it can correctly execute any arbitrary MIPS program.[1] When the simulator detects data dependences, it correctly handles them by stalling and/or bypassing. When the simulator detects control dependences, it correctly handles them by flushing the pipeline as necessary.

By executing the following command, you can see that your simulator (`sim`) does indeed have identical architectural outputs (e.g., register values) as the reference simulator (`refsim`) for all the test inputs that we provide in `exampleinputs/`.

```
$ make verify
```

However, your simulator has different microarchitectural outputs (e.g., cycle count) than the reference simulator. Your job is to model the timing effects of the branch predictor and main memory so that your simulator becomes microarchitecturally equivalent to the reference simulator.

5.2. Studying the Timing Simulator

Please study `pipe.c` and `pipe.h` in detail. The simulator models each pipeline stage as a separate function – e.g., `pipe_stage_fetch()`. The simulator models the state of the pipeline as a collection of pointers to `Pipe_Op` structures (defined in `pipe.h`). Each `Pipe_Op` represents one instruction in the pipeline by storing all of the necessary information about the instruction that is needed by the simulator. A `Pipe_Op` structure is allocated when an instruction is fetched. It then flows through the pipeline and eventually arrives at the last stage (writeback), where it is deallocated once the instruction completes. To elaborate, each stage receives a `Pipe_Op` from the previous stage, processes the `Pipe_Op`, and passes it down to the next stage. The simulator models pipeline stalls by stopping the flow of `Pipe_Op` structures and pipeline flushes by deallocating the `Pipe_Op` structures at different stages.

5.3. Helping you understand

- In order to help you in getting started, we have already implemented the `bp.h` file which has the initialized data structures and the prototype declaration of the functions for your branch predictor. You can directly jump to coding your branch predictor in your `bp.c` file.
- You need to add more code in the `pipe.c` file to incorporate your branch predictor in the system. We have marked those regions with some "`TODO`" comments, indicating the regions where more code is needed and also specifying what that code should be doing.
- You may toggle the `DEBUG` flag at the beginning of `pipe.c` to enable/disable information printing on every cycle. By default it prints out the pipeline states and the predictor information to `stdout`. You can compare the output of your predictor against our reference predictor for debugging.
- To make things easier for you to understand, we have already added the functions in `bp.c` file to initialize and destroy your branch predictor. You may follow up, to write your own functions to further add the logic for your branch predictor.
- You may use

  ```
  $ make verify
  ```

  or

  ```
  $ ./verify
  ```

  to validate the correctness of your predictor using the entire set of test cases in `exampleinputs/`. Use

  ```
  $ make verify INPUT=exampleinputs/inst/addiu.x
  ```

[1] This is not entirely true since we pose the usual restrictions on system calls, exceptions, etc.

or

```
$ ./verify exampleinputs/inst/addiu.x
```

to validate the correctness of a single test case.

Here is a list of statistics the `verify` script uses for comparison. You can also print out the statistics by using `rdump` command in your shell:

- Final program counter (`PC`) and register file (`R0 - R31`, `HI`, `LO`) states
- Number of simulated cycles (`Cycles`)
- Number of fetched instructions (`FetchedInstr`)
- Number of retired instructions (`RetiredInstr`)
- Instructions per cycle (`IPC`)
- Number of pipeline flushes (`Flushes`)
- Branch prediction accuracy (`BPAccuracy`)

6. Extra Credit.

One important goal of the branch predictor is to perform predictions as accurate as possible. For extra credit on this lab, your goal is to improve the accuracy of your branch predictor by either optimizing your current Gshare predictor, or implement an entire new predictor algorithm. For example, your GShare predictor can be part of a tournament predictor (`http://classweb.ece.umd.edu/enee646.F2007/combining.pdf`). By adding another local predictor alongside your Gshare predictor, you can select the better one from the two predictions. Another idea could be implementing a new predictor like the perceptron-based predictor (`https://www.cs.utexas.edu/~lin/papers/hpca01.pdf`).

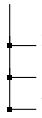
The extra credit is worth up to 20% additional credit for this lab, based on the amount of additional work you have done. Additionally, there will be prizes for students who have achieved best prediction accuracy on an undisclosed set of test inputs.

If you are submitting your extra credit implementation, please copy-paste the entire `Lab3/` folder to another folder named `Lab3_extra_credit/` and implement the extra credit part in that folder (example directory structure shown in Section 7). In other words, the implementation in `Lab3/` folder should be the original GShare predictor we required and the outputs should match those from the reference GShare predictor.

7. Handin.

Please make sure your source code compiles without any error before submission. You should submit your code to Collab by packing the folder of `Lab3/` and extra credit into a Zip file, and renaming it with your UVa computing ID, e.g., `yw2bc.zip`. More specifically, your directory should follow this structure (Ignore `Lab3_extra_credit/` folder if you don't have that):

```
yw2bc.zip
├── Lab3
│   ├── exampleinputs
│   ├── inputs
│   ├── Makefile
│   ├── refsim
│   ├── src
│   └── verify
└── Lab3_extra_credit
    ├── exampleinputs
    ├── inputs
    ├── Makefile
    └── README
```



In addition, please turn in at least 5 additional test cases and add them to the `inputs/` subdirectory when submitting Lab 3. We recommend adding comments to make your code more readable and describe any additional aspects of your design details in a separate README. We will test your simulator extensively with a suite of test cases. In case your simulator fails to pass some of the cases, the documentation and comments will help us give you partial credit.