As we will discussed in class tomorrow, Speculative Execution Attacks (Spectre) are dependent on several key processor features in order to be exploited. One of these features is that the processor needs to have a dynamic branch predictor so that the adversary can mistrain the predictor and induce a long, speculative window.

In this lab, you are asked to play the role of a chip developer who wishes to target branch prediction as a means to defend against Spectre attacks. In particular, you work from the key insight – *if the branch predictor predicts a random target, then there is no way for an adversary to control the branch predictor's behavior*. Before fabricating the idea, which will take weeks or months of effort, you decide that you want to simulate the approach first to be able to determine whether or not such an approach will tank performance.

Your task in this lab will be to evaluate the performance of different branch prediction strategies by running various simulations and comparing the branch prediction outputs. Afterwards, you will be asked to implement a branch predictor that outputs a random value for each prediction and measure the performance overhead of such a scheme relative to the overhead of using a naïve approach compared to the state-of-the-art. This will allow you to say that, for these workloads, a random branch predictor is **x**% worse than the state-of-the-art and that such a result may or may not be tolerable give that a naïve dynamic branch predictor is **y**% worse than the state-of-the-art.

Purpose of This Lab. This lab will reinforce the following concepts:

1. Running gem5 simulations with various configurations
2. Analyzing various gem5 outputs to draw conclusions about program behavior
3. Modifying the simulator back-end to implement desired behaviors

Getting Started

In your `isa-assignment-stencil` directory, begin by running the following:

```
wget https://cs.pomona.edu/classes/cs181ca/labs/lab11.zip
unzip lab11.zip
chmod +x lab11/setup.sh
./lab11/setup.sh
scons build/RISCV/gem5.debug -j8
```

1 Profiling Current Branch Predictors

1.1 Understanding the Environment

Start by looking at our hardware model as declared and defined in `configs/example/gem5_library/lab11.py`. The configuration is relatively simple – there is a basic `Board` that uses a custom processor for the purposes of this lab that exposes the branch predictor. We can set the branch predictor without needing to recompile the simulator backend by using the `--bp` parameter as the input to our simulated run.

From the candidate options, we can see that the branch predictor options are "`LocalBP`", "`TournamentBP`", "`TAGE`", and "`Lab09`". The `LocalBP` describes a simple 2-bit dynamic branch predictor, the `TournamentBP` describes a branch predictor that tracks local and global updates to dynamically determine which is more effective at any moment in time, the `TAGE` branch predictor describes the state-of-the-art predictor that is used in many commodity processors today, and you will implement the `Lab09` branch predictor by the end of the lab (for now, it always predicts "not-taken").

We are also provided with several microbenchmarks (e.g., toy programs that implement well defined, toggle-able behaviors) in the `lab09-tests` directory. These are binary files fetched from gem5 resources that are precompiled for RISC-V, and exhibit various degrees of ease with which branches can be predicted. In particular, the external resources describe the workloads as the following:

1. `CCa.RISCV` describes a workload in which branches are biased
2. `CCe.RISCV` describes a workload in which branches are easy to predict
3. `CCh.RISCV` describes a workload in which branches are impossible to predict
4. `CCm.RISCV` describes a workload in which branches are heavily biased
5. `CRf.RISCV` describes a workload in which control instructions are recursive calls to implement Fibonacci computations

1.2 Your Task

Think about how you would expect the predictors to behave for each of these workloads. For example, how do you expect a simple predictor to perform relative to a state-of-the-art predictor for a simple prediction task? Would you expect the same result to hold for a more complex task? Would the relative differences be the same? Let these questions serve as an initial hypothesis for the expected behavior that we can then work towards confirming or denying. In asking and answering this question, we will inform our understanding of both the workloads and the effectiveness of the branch predictors!

After the runs complete, you may find it useful to digest the outputted `stats.txt` files. In particular, if you search for "branch" in this stats file, you will find all sorts of metrics and breakdowns of the behavior of the branch predictor for each of these workloads. Note, we are interested in understanding the **runtime performance** of the workload while tweaking this sole independent variable and the **prediction performance** of the branch predictor. This will help us correlate the impact of improvements/worsening performance in various contexts.

2 Implementing a New Branch Predictor

2.1 The Abstract Branch Predictor

Branch predictors in gem5 are declared and defined in the `src/cpu/pred` directory. From here, they extend a base abstract class declared and defined in `bpred_unit.{hh, cc}`. When you navigate to these files, you will notice that there are several key functions declared:

```
/**
 * Looks up a given conditional branch PC of in the BP to see if it
 * is taken or not taken.
 * @param pc The PC to look up.
 * @param bp_history Pointer that will be set to an object that
 * has the branch predictor state associated with the lookup.
 * @return Whether the branch is taken or not taken.
 */
virtual bool lookup(ThreadID tid, Addr pc, void * &bp_history) = 0;

/**
 * Ones done with the prediction this function updates the
 * path and global history. All branches call this function
 * including unconditional once.
 * @param tid The thread id.
 * @param PC The branch's PC that will be updated.
 * @param uncond Wheather or not this branch is an unconditional branch.
 * @param taken Whether or not the branch was taken
 * @param target The final target of branch. Some modern
 * predictors use the target in their history.
 * @param bp_history Pointer that will be set to an object that
 * has the branch predictor state associated with the lookup.
 */
virtual void updateHistories(ThreadID tid, Addr pc, bool uncond,
                         bool taken, Addr target, void * &bp_history) = 0;

/**
 * @param tid The thread id.
 * @param bp_history Pointer to the history object.  The predictor
 * will need to update any state and delete the object.
 */
virtual void squash(ThreadID tid, void * &bp_history) = 0;


/**
 * Updates the BP with taken/not taken information.
 * @param tid The thread id.
 * @param PC The branch's PC that will be updated.
 * @param taken Whether the branch was taken or not taken.
 * @param bp_history Pointer to the branch predictor state that is
 * associated with the branch lookup that is being updated.
 * @param squashed Set to true when this function is called during a
 * squash operation.
 * @param inst Static instruction information
 * @param target The resolved target of the branch (only needed
 * for squashed branches)
 * @todo Make this update flexible enough to handle a global predictor.
 */
virtual void update(ThreadID tid, Addr pc, bool taken,
                void * &bp_history, bool squashed,
                const StaticInstPtr &inst, Addr target) = 0;
```

What can we learn from these declarations? For a branch predictor to work, we need to

have some way to `lookup` whether some current PC value has some metadata (e.g., `bp_history`, which is defined as such to remain agnostic to the particular branch prediction algorithm) that can inform whether or not we should predict branch taken (`true`) or branch not taken (`false`).

The `updateHistories` function is an abstract function that will be called by the processor whenever it is appropriate to update any associated *global* metadata to help with the decision making process. Note, the parameters to this function are an over-approximation of all of the fields that might be relevant to the branch prediction algorithm. On the other hand, `update` is a function that performs similar operations on a local basis (as you can see from the TODO, these functions may some day be merged into a single function).

`squash` is called whenever a predicted branch has been determined *not* to have been evaluated correctly. This implies that the pipeline is cleared of the incorrect instructions until the hazard is ultimately resolved.

2.2 Your Task

Given this setup, your task is to implement a new branch predictor that implements each of these functions. The branch predictor is declared and defined in `src/cpu/pred/lab11.{hh, cc}` where the scaffolding predefines each of these functions with dummy placeholder values. You may find it useful to first look at the `src/cpu/pred/2bit_local.cc` to see how these functions are implemented when defining a two-bit dynamic branch predictor.

After you feel like you've implemented your random branch predictor, be sure to recompile gem5 and try testing your implementation to study the overhead of this approach!