Lab 6: Branch Predictor Design in PyRTL

Assigned:	*Wednesday, February 14th, 2024*
Due:	*Wednesday, February 21nd, 2024*
Points:	*100*

• MAY ONLY BE TURNED IN ON **GRADESCOPE as PYTHON files** (see below for details).

• There is NO MAKEUP for missed assignments.

• We are strict about enforcing the LATE POLICY for all assignments (see syllabus).

Goals for This Lab

By the time you have completed this lab, you should be able to utilize **PyRTL** to design and simulate branch predictor units, including a 2-bit branch predictor and a branch history table (BHT) for more efficient predictions.

Task

The skeleton code provided for this lab implements a 1-bit branch predictor, as introduced in the lecture. Your task is to implement a 2-bit branch predictor and a branch history table (BHT) which would optimise the flow in your instruction pipeline and mitigate the costs of branching. Note however that we will not be using a processor pipeline in this lab, but rather simulating the predictors standalone.

Provided Files

We have provided you with several starter files (see the Canvas “lab6” folder):

1. ucsbcs154lab6_1bitpred.py - This is a sample 1-bit branch predictor that you may find helpful to get started with your own predictor implementations.
2. ucsbcs154lab6_2bitpred.py - This is the skeleton file where you will need to implement your 2-bit branch predictor.
3. ucsbcs154lab6_predtable.py - This is the skeleton file where you will need to implement your branch history table predictor.
4. demo_prog.s - This is a sample assembly program from which demo_trace.txt and demo_trace_branch_only.txt are derived.
5. demo_trace.txt - This is the trace from the execution of demo_prog.s that outlines the expected branching behaviour for the program. The file is described in the following format: **col 1)** the program counter at each branch, **col 2)** whether the instruction at the given pc takes a branch (1 for taken, 0 for not), and **col 3)** whether the current instruction is a branch or not. You may use this file to test your branch predictor against.
6. demo_trace_branch_only.txt - Similar to demo_trace.txt, this text file shows the trace

of execution from demo_prog.s , but this time only tracing branch instructions; as such every number in column 3 is trivially a 1. You may find this trace to be helpful to test your predictor against as it incorporates the case of repeated branches, where you may potentially need to bypass the updated predictor state.

Instructions

The following figure shows the state machine for the 1-bit branch predictor present in the provided skeleton file ucsbcs154lab6_1bitpred.py. For this assignment, we want to implement two branch predictors: a 2-bit saturating counter and a branch history table containing 8 separate 2-bit saturating counters.

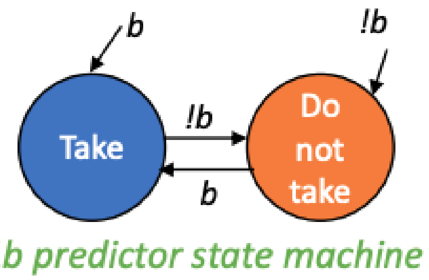

Figure 1: State machine for a 1-bit predictor

2-Bit Saturating Counter

Implement the 2-bit saturating counter logic in ucsbcs154lab6_2bitpred.py using the skeleton provided to you. If you have questions, you can observe Figure 4.63 from the textbook, which we have reproduced here as Figure 2. This state machine is slightly more complex than the 1-bit predictor (e.g. you need to use a larger register to store the state), but the implementation shouldn't be significantly more complex. You will submit your file to Gradescope.

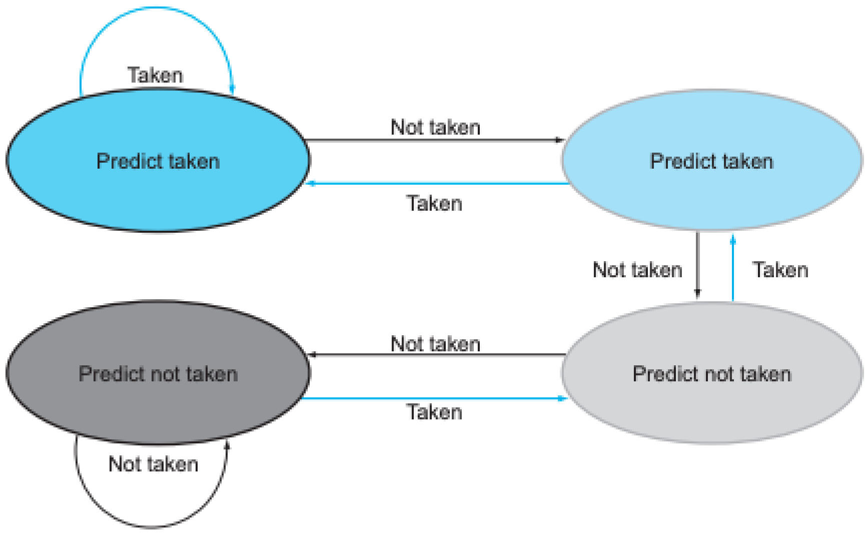

Figure 2: State machine for a 2-bit saturating counter predictor

8 Entry Branch History Table

A branch history table enables the processor to distinguish between branches based on their addresses. The table that we wish to implement will consist of eight entries, each of which is a 2-bit saturating counter as created in the previous section. Implement the branch history table logic in ucsbcs154lab6_predtable.py using the skeleton provided to you. You will use a PyRTL MemBlock instead of a Register to store the state of the predictor and use the lower, word-addressing bits of the PC to index that MemBlock (i.e. the LSBs above the bottom 2 bits, since instructions are aligned to 4-byte boundaries). You will submit your file to Gradescope.

Design Note

The predictors have two features. The first feature is to make a prediction based on the current PC in the fetch stage. This should be done every cycle, regardless of the type of the instruction. The other feature is to update the predictor state based on the PC and branch outcome from the decode/execute stage (depending on the pipeline design). If the predictor is being updated and making a prediction on the same cycle, you may (depending on the predictor design) need to bypass the updated prediction state to make the prediction on a given cycle, since the state won't be updated until the following cycle (because it's stored in a Register or synchronous MemBlock).

Test your Design!

We have provided for you, in each skeleton, an environment in which you may test your branch predictors.

Within the main function of the skeleton files, you will see code that simulates values for the 4 inputs of your branch predictor. This is done by reading in a trace.txt file (via the python **open** function) and parsing out information into the input wire vectors. Additionally, some bookkeeping is done to measure the accuracy of your branch predictor given a certain trace file, which is printed out at the end of the simulation.

How do you test your PyRTL branch predictor design?

1. Choose/write a trace file which you wish to test your branch predictor against
2. Read in the file via python by editing this line here in your skeleton

```
f = open("filename.txt", "r")
```

3. Run your code!

```
python3 ucsbcs154lab6_2bitpred.py
```
or
```
python3 ucsbcs154lab6_predtable.py
```

We have provided you with a sample test case in demo_prog.s. We have also provided this in the form of two traces, demo_trace.txt and demo_trace_branch_only.txt. You may use these trace files to help simulate your branch predictor for testing purposes, however you are encouraged to create additional, more comprehensive test cases on your own to cover cases that the traces provided do not.

Files to Submit

You need to submit two files for this lab:

- ucsbcs154lab6_2bitpred.py: Implement the 2-bit saturating counter branch predictor in this file using the skeleton code.
- ucsbcs154lab6_predtable.py: Implement the 8 entry branch history table of-2 bit saturating counters in this file using the skeleton code.

Please keep your eyes open for any Piazza announcements in case we make updates to the submission requirements.

Autograder

We will NOT be releasing an autograder to verify your predictors. Part of this assignment is to learn how to test your code! We WILL allow you to share **test code** (MIPS assembly) and **branch traces** with each other on Piazza. While this is optional, you may write your own comprehensive branch trace test suite and share it on Piazza under our stickied post, and you may also download your classmates' tests if they've uploaded them.

We (the instructors) will not be responsible for debugging any mistakes in these test suites, nor will we endorse any of them. If you think there is a bug or issue with a test suite, you should comment under the author's submission with your concerns. Lastly, passing these tests is not a guarantee of any score on the assignment or from the autograder.