Branch Prediction

1. Main Goal

Add a branch prediction unit (BPU) to implement the dynamic branch prediction scheme.

2. Motivation

Branch instructions are not detectable in the default structure, which means RISCV will load the next instruction at PC+4 instead of the branch target, and will cause a one-cycle-penalty if branch is taken. To reduce such penalty, a "Branch Prediction Unit" is proposed.

Branch Prediction is based on the concept that branch instructions are sometimes taken consecutively and can be further "predicted". Take the following assembly codes for example:

```
ADD $t0, $0, $0     #set $t0 to 0          for(int i=0; i != 100; i++)    {…}
ADDI$t1, $0, 100    #set $t1 to 100        for(int i=0; i != 100; i++)    {…}
SLL…                #loop body             for(int i=0; i != 100; i++)    {…}
ADD…                #assume this loop contains 3 instructions
SUB…
ADDI$t0, $t0, 1     #add 1 to $t0              for(int i=0; i != 100; i++)    {…}
BNE  $t0, $t1, -5   #branch if $t0 != $t1      for(int i=0; i != 100; i++)    {…}
```

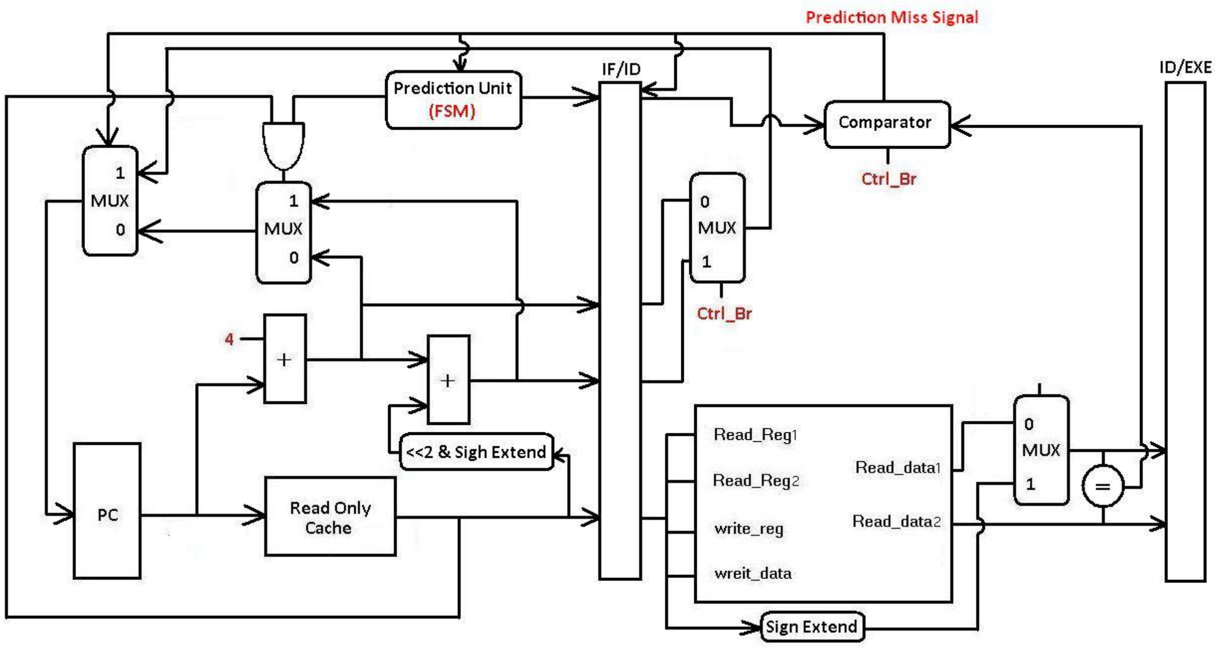
The BNE which forms the loop will be taken for 99 times, and leads to a waste of 99 cycles. But with the Branch Prediction Unit which will be discussed in the next section, this penalty will be reduced to 3 cycles only and thus improves the performance significantly.

With CPUs being deeply pipelined, the penalty is getting larger and more serious, and Branch Prediction is becoming more essential as a consequence.

3. Implementation

To implement Branch Prediction, the main idea is to identify and execute the branch instructions in IF stage. That is, the target address needs to be calculated and the prediction needs to be made in IF stage. The prediction has to be further verified in ID stage, in order to fix the miss-prediction and maintain the correctness. The penalty of miss-prediction is to take one cycle flushing the wrong instruction in IF stage and restore the right PC.

The block diagram of Branch Prediction Unit is as follows:



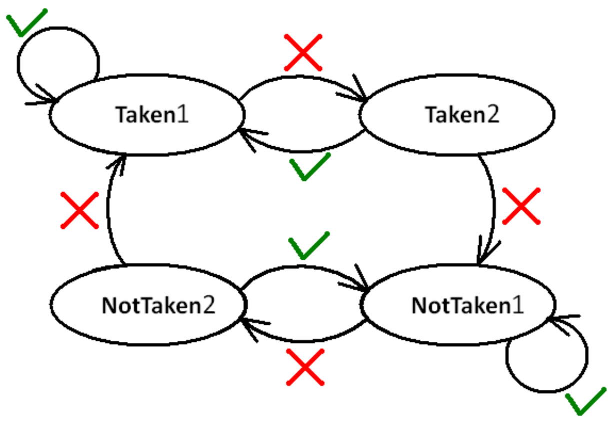
Description:

a. The "<<2 & Sign Extend" block and the adder above calculate the branch target.

b. The "Prediction Unit" makes the decision of whether the branch is taken or not.

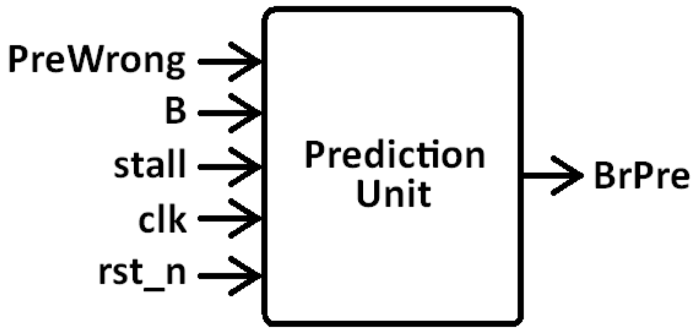
c. The comparator verifies the correctness of the prediction, and sends a signal to flush the wrong instruction if miss-prediction occurs.

The decision of "Prediction Unit" is based on a "Branch Prediction Model". And many models with different accuracy and complexity have been proposed up to now. For simplicity, a basic model is picked for example.



This model can be implemented with a Finite State Machine. The green check mark indicates a successful prediction while the red cross mark indicates a wrong one. At the beginning, the FSM is at the state "NotTaken1", which predicts that the branch will not be taken. As the miss-prediction occurs twice in a row, a loop is assumed to be under execution. And the FSM will change to state "Taken1", which makes a positive prediction on branching. Similarly, the state goes back to "NotTaken1" after another two consecutive miss-predictions, under the assumption that the loop condition has been broken.

The input/output ports of Prediction Unit are described as follows for reference:



```
module PredictionUnit (BrPre, clk, rst_n, stall, PreWrong, B);
output BrPre;          // Prediction Output
                       // 1/0 stands for a positive/negative prediction on branching
input  clk, rst_n,     //input clock and reset signal
       stall,          //stall signal from cache or hazard detection unit
       PreWrong,       //miss-prediction signal sent from the comparator
       B;              //branch signal
                       //which will be set to 1 if the inst. in IF stage is a branch inst.
```

With such mechanism, the penalty on branch instructions is thus reduced. Take the assembly codes in the first section for example. This model will leads to 2 miss-predictions at the beginning of the loop, with the following 97 prediction correct. And at the end of the loop, there will be one more miss-prediction due to the break of the loop condition. And the number of cycles wasted is reduced from 99 to 3.

This model is very simple, thus it is only capable of making accurate predictions on specific loops. To enhance the accuracy of Branch Prediction Unit, other models with deeper inspection of loop formations are required and can be accessed on the Internet.

4. Testbench

"**I_mem_BrPred**", and "**TestBed_BrPred.v**" collectively form a set of test module, which you can find under "a10b20c30/".

<u>Test Program Generation.</u>

This part will show you how to generate different test cases and the simulation process

A. Execute "BrPred_generate.py" in the directory "generate/":

-Python (version = 3.x)

-Modify nb_notBr, nb_interBr, nb_Br

-I_mem_BrPred_ref & TestBed_BrPred_ref should be placed in the same folder

-I_mem_BrPred & TestBed_BrPred will be generated (Provided file in "a10b20c30/")

B. Put all the files in the same folder:

-Final_tb.v, TestBed_BrPred.v, I_mem_BrPred

-CHIP.v, slow_memory.v

- Your RTL files (e.g., RISCV_Pipeline.v, cache.v)

-CHIP_syn.v, CHIP_syn.sdf (for gate level simulation)

C. Simulation commands:

RTL level: vcs Final_tb.v CHIP.v slow_memory.v [other RTL files] -full64 -R -debug_access+all +v2k +define+BrPred

Gate level: vcs Final_tb.v CHIP_syn.v slow_memory.v -v tsmc13.v -full64 -R -debug_access+all +v2k +define+BrPred +define+SDF

5. Requirement

Design. You should pass the given testbench "Final_tb.v" with the instruction memory and the testbed under the *a10b20c30/* directory.

Report. In the branch prediction part of your final report "**Report.pdf**", we suggest you have a detailed discussion on the followings:

(a) Design methodology for good score (before/after)

(b) The relationship between design BPU and parameter size for generating test program

(c) What you have learned?

(d) Other detailed discussion will be appreciated

Evaluation Metrics.

Base on the test program "I_mem_hasHazard" and "I_mem_BrPred"

Score 1 (BP_S1): Total execution cycles of I_mem_BrPred

BP_S1 = total cycle counts of the I_mem_BrPred

Score 2 (BP_S2): Total execution cycles of I_mem_hasHazard

BP_S2 = total cycle counts of the I_mem

Score 3 (BP_S3): Synthesis area of BPU (um^2)

Don't worry about the performance evaluation. It is just one of the criteria. **Focus more on what you design to solve problems you face.**