1. Purpose:

a) Be familiar with the common strategies of branch prediction.

b) Choose one branch prediction strategy to design a pipelined MIPS processor with branch prediction function.

c) Get hands-on experience with the pipelined MIPS processor and realize it in Modelsim.

d) Design a validation strategy to analyze the effectiveness of the branch prediction function.

2. Introduction:

Note: The project is based on the Pipelined MIPS Processor Version with branch execution taking place in EX cycle, not the optimized version which put branch execution in ID cycle.

Why to implement branch prediction for pipelined MIPS processor?

The requirement of implementing branch prediction comes from control hazard, which means the processor tries to make a decision on the next instruction to fetch before the branch condition is evaluated. The typical five-cycle pipelined MIPS processor can't get "taken or not" and target fetch address for next instruction until carrying out EX cycle, the 3rd, for branch instructions. So, there may be bubbles for the original pipelined version without branch prediction.

How to realize it?

Obtain branch direction (taken or not taken) and target address during the IF cycle of the current branch instruction is the main idea for all prediction strategies, which are divided into static method and dynamic method according whether it can dynamically adapt as program behavior changes.

2.1 Static Prediction:

The prediction is compiler-determined as permanent "taken" or "not taken" in the program compiling process, which means fixed for the program life. It's easy to implement but resulting with so-so accuracy (30-40% accuracy for not taken, 60-70% accuracy for taken).

2.2 Dynamic Prediction-- One-bit Branch Predictor

The dynamic prediction strategies are based on branch historical recording. To fetch the appropriate instruction for this branch, processor execute same as the previous branch executed.

The simplest implementation case, called one-bit branch predictor, sets one bit as the direction's recording table (0 as no taken, 1 as taken) for previous branch, and the bit is stored with the branch instruction address as the index used to look up if the actual instruction is branch and its direction.

For typical loop function, one-bit branch predictor generally causes misprediction twice.

- At the last loop iteration, since the prediction bit will say taken, while we need to exit from the loop.
- When we re-enter the loop, at the end of the first loop iteration we need to take the branch to stay in the loop, while the prediction bit says to exit from the loop, since the prediction bit was flipped on previous execution of the last iteration of the loop.

The worse situation is that the same index has been referenced by two different branches, and the previous history refers to the other branch, which cause continuous mispredictions.

2.3 Dynamic Prediction-- Two-bit Branch Predictor

In the two-bit branch predictor, 2 bits are used to encode the four states of a finite state machine for each index in the table which changes prediction only if get misprediction twice. The state is incremented on a taken branch and decremented on an untaken branch.

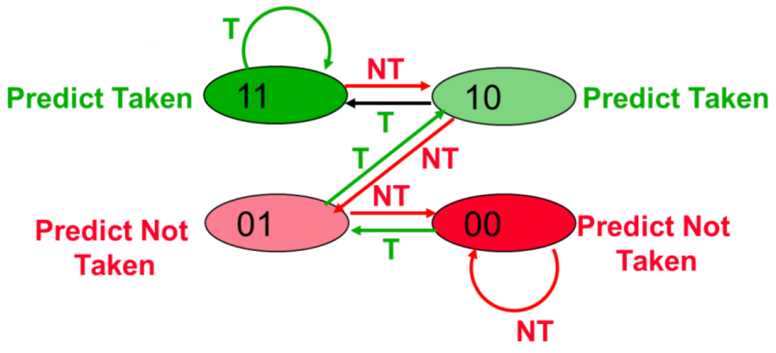

Fig1. two-bit branch predictor transition logic

For typical loop function, two-bit branch predictor causes misprediction only once and eliminate continuous mispredictions for the worse case described above.

Key parts to realize the branch prediction based on BTB (Branch Target Buffer):

a) Predicting branch

Predicting process includes looking up BTB (Miss or Hit) to obtain branch direction and target address, choosing appropriate PC to fetch the next instruction. The process is done during the IF cycle to support fetching the next instruction.

Miss: no corresponding branch instruction in the table with the current instruction, a default prediction should be broadcasted (**not taken is set as the default**, in fact the target address is unavailable if you choose taken as the default).

Hit: using data from BTB to do prediction.

b) Validating prediction and updating BTB

Validating if the prediction is failure and updating direction and target address in BTB table according the above validation. The process is done during the EX cycle after virtual direction and virtual target address being obtained from branch execution. So, when validation is done, two instructions have been fetched into pipeline after the branch instruction being fetched.

2 misprediction cases: predict not-taken, actual taken; predict taken, actual not-taken, or actual taken but wrong target. Essential actions to restore pipeline: flush IF/ID, ID/EX and EX/MEM to clear the fetched instructions, re-fetch instructions from correct path.

3 BTB updating cases: no corresponding branch instruction in BTB, which leads to Miss for looking up within BTB; Hit for looking up within BTB, but misprediction; Hit and correctly predicting, but the current state is “10” or “01”, should change state to “11” or “00”.

c) Recovering from misprediction

If the validation shows it’s a misprediction, the two ever fetched instructions should be flushed to ensure correct executing results for the whole program. In practice, the registers of IF/ID, ID/EX and EX/MEM are actively set with zero filling.

2.3.1 Branch Target Buffer

Branch Target Buffer (BTB) is constructed based on the theory of cache memory (registers are chosen as the physical storing medium in this project), which stores 2-bit state, target address, branch PC tag and validation for each branch instruction supported by the prediction system. Its structure is almost same as cache memory. The row number of BTB indicates how many branch instructions can be predicted. For an example showed in Fig2, we choose 6 bits as index, thus 2^6=64 branch instructions can be supported with predicting function and the BTB includes 64 elements with 59 bits for each one.

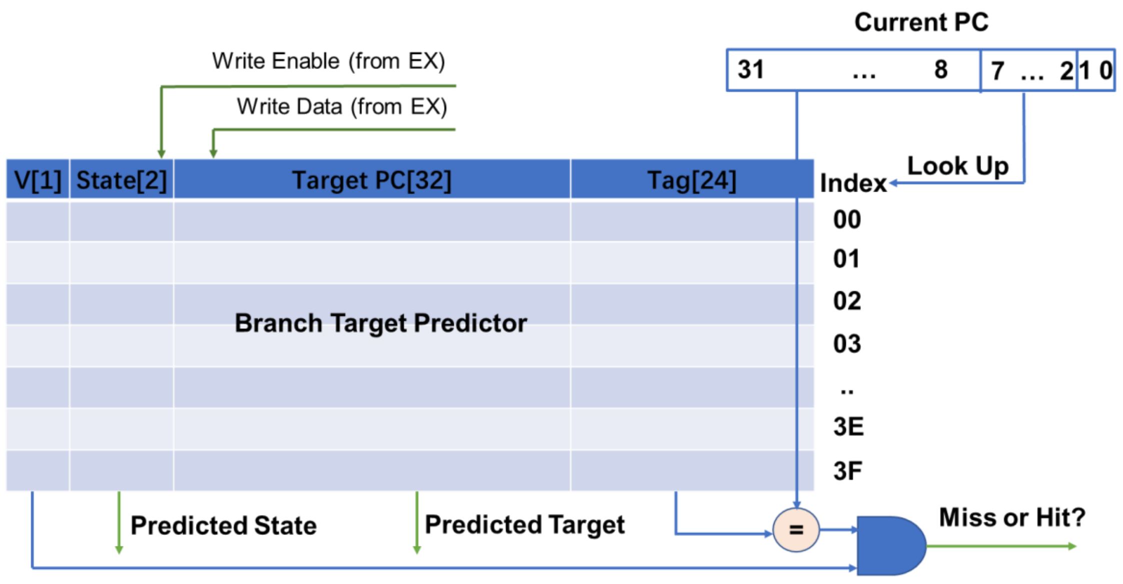

Fig2. Branch Target Buffer Structure with 6 bits index

As the common branch, V[1] (Valid bit) is used to show if the row has been written with meaningful prediction bits.

BTB updating: BTB will be written when 3 BTB updating cases happen, which are detected in the EX cycle. So, the trigger signal and written data come form EX cycle.

2.3.2 Branch Predicting Controller

Branch Predicting Controller correlatively works with BTB and the surrounding components (such as mux) to predict and fetch appropriate instruction. There are two levels mux-choosing for the next PC, the first of which chooses next PC according prediction. The second level adjust the third fetch instructions after branch to the executed target from EX cycle.

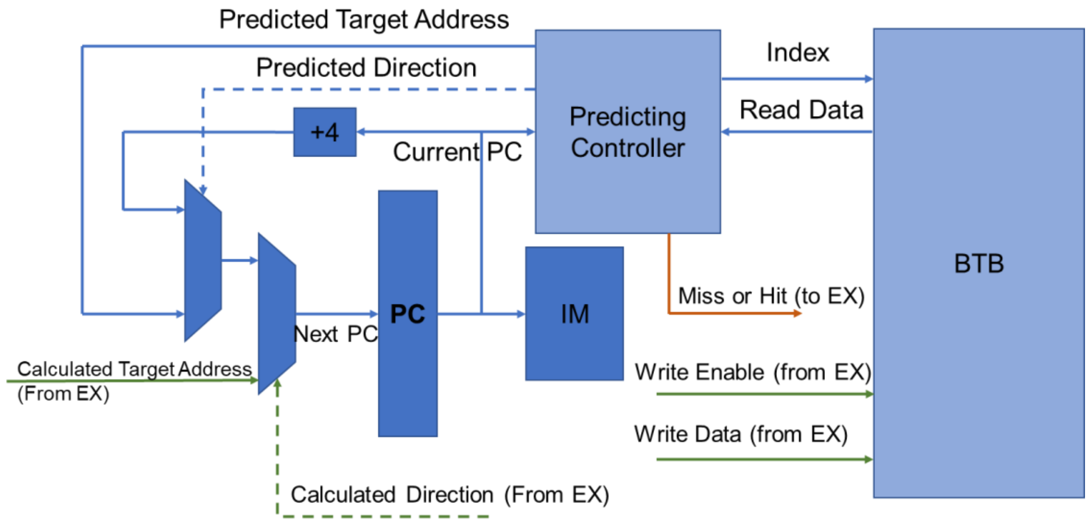

Fig3. Branch Predicting Controller

2.3.3 Branch Prediction Evaluating

This module use “**2 misprediction cases**” and “**3 BTB updating cases**” as the logics to drive pipeline registers resetting and BTB updating, as shown in Fig4.

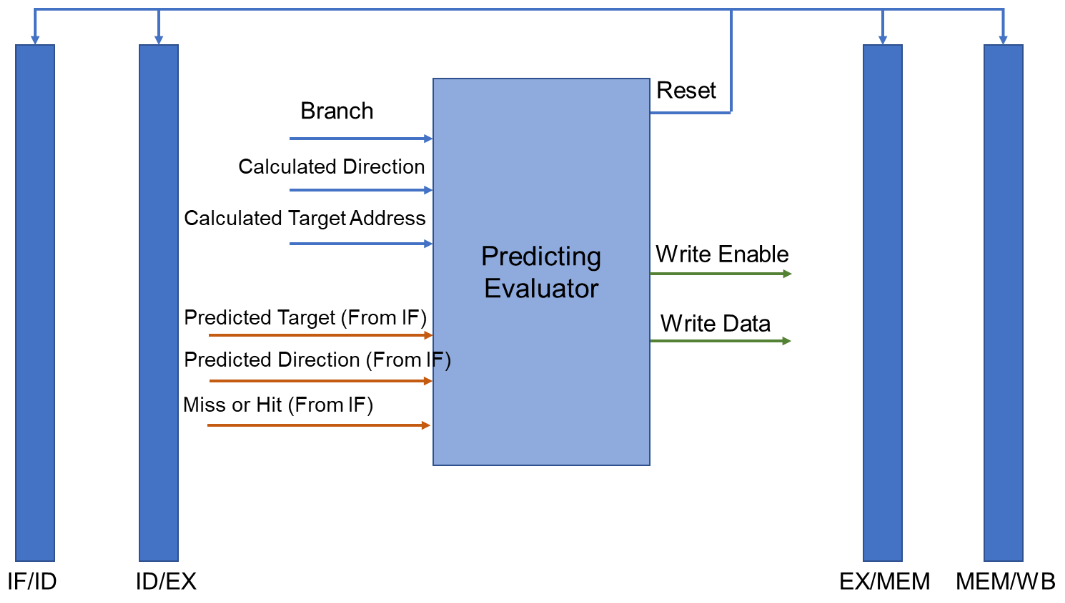

Fig4. Branch prediction evaluating and pipeline reset

Fig3. BTB Branch prediction in a pipelined structure

3. Build Modelsim project and validate with loop program

The branch prediction project is built based on the project of CE622 lab07, in which three new components are constructed: PredC.vhd (branch predicting controller), PredBTB.vhd (Branch Target Buffer) and PredEvalu.vhd (Branch Prediction Evaluating).

To validate efficiency of the 2-bit dynamic branch prediction, a do-file with two loop programs is designed, in which sum is repeatedly calculated with an increasing operand. The predictor’s processing capability with multi branch instructions can be validated with four branch instructions in the programs.

```
# ********** Loop program 1***********

# start: lw $a0,12($zero)

# lw $t0,00($zero)

# lw $t1,04($zero)

# lw $t2,08($zero)

# loop: slt $t3, $t1, $a0
# beq $t3, $zero, finish

# add $t0, $t0, $t1

# add $t1, $t1, $t2

# beq $zero $zero loop

# sw $t0,16($zero)

# ********** Loop program 2***********

# start: lw $a0,12($zero)

# lw $t4,00($zero)

# lw $t5,04($zero)

# lw $t6,08($zero)

# loop: slt $t7, $t5, $a0

# beq $t7, $zero, finish

# add $t4, $t4, $t5

# add $t5, $t5, $t6

# beq $zero $zero loop

# sw $t4,20($zero)

# ********** Loop programs end***********
```

After running the above programs on designed processor with prediction function, the sum result, 40 (hex) same as expected, is written to data memory twice with running duration of 106 cycles.

If we run the programs on the original processor, which uses 2 pipeline-stall for branch instructions, the running duration is 166 cycles with 40 being written twice.

So, the processor with 2-bit dynamic prediction not only works for multi-branch program but also gets a 1.57 times improvement comparing with the original pipeline stalling version.

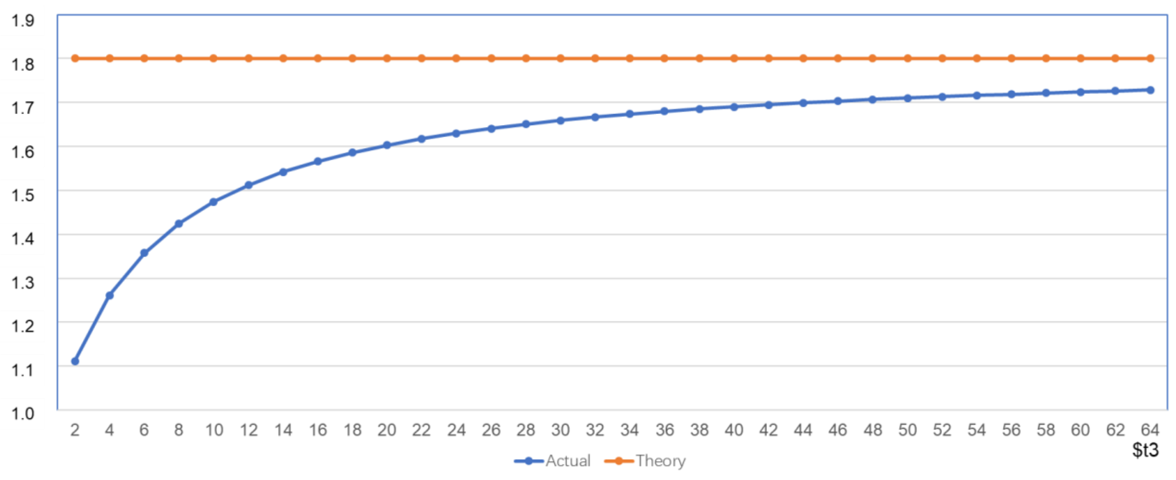

Fig5. The improvement follows times of loop

If we set $a0=12(hex), 14(hex) and 16(hex), the sum result will be 51(hex), 64(hex) and79(hex) and the running duration will be 116, 126 and 136 cycles for prediction processor. Correspondingly, the duration time on pipeline stalling processor is 184, 202 and 220 cycles. The improvement is about1.59, 1.60 and 1.62 times. So, the improvement increases as the loop number is set bigger and tends to approach the maximus theory improvement 1.8 times, as shown in Fig5.