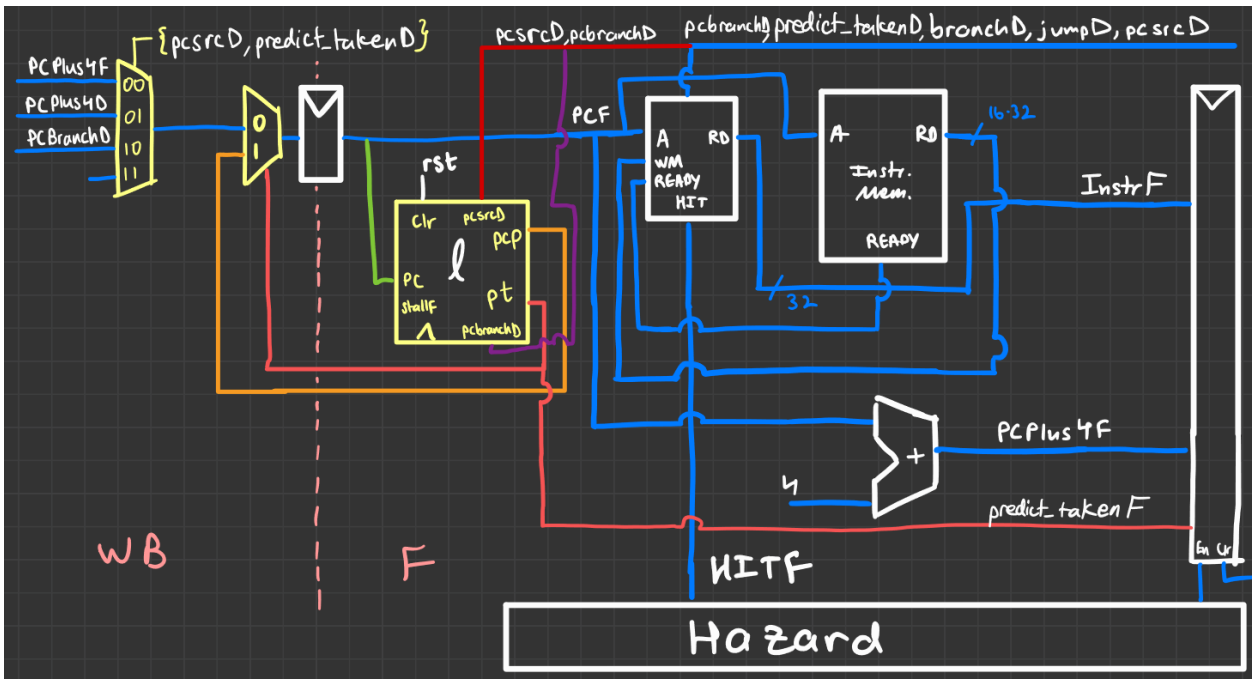
Project 03: Branch Prediction
Brian Lee, Arjun Vinod
ECE154B W22

Overview

In previous designs, whenever a program instruction caused a branch, the processor was forced to waste a cycle by flushing the instruction it fetched by assuming the program counter would increment by 4. A branch predictor would introduce a new heuristic-based approach to branching that may reduce the CPI for programs that repeatedly branch.

Design Specification

Local Prediction



Local Predictor - Writeback/Fetch Stage Modifications

We developed a local predictor module that takes input program counter, the signal from the decode stage that signals a branch taken (pcsrcD), the signal from the decode stage that has the branch target address (pcsrcD), clock, and asynchronous reset. The module outputs the signal that says if a predicted branch is taken (predict_takenF), and the predicted target program counter that's fed into a MUX in the writeback stage that determines the next program counter value.

Internally, there's a register with 2^7 34-bit entries (pht). The least two significant bits represent the state of the finite state machine encoding the branch predictor's state for the inputted program counter. The upper 32 bits is the target program counter corresponding to the inputted

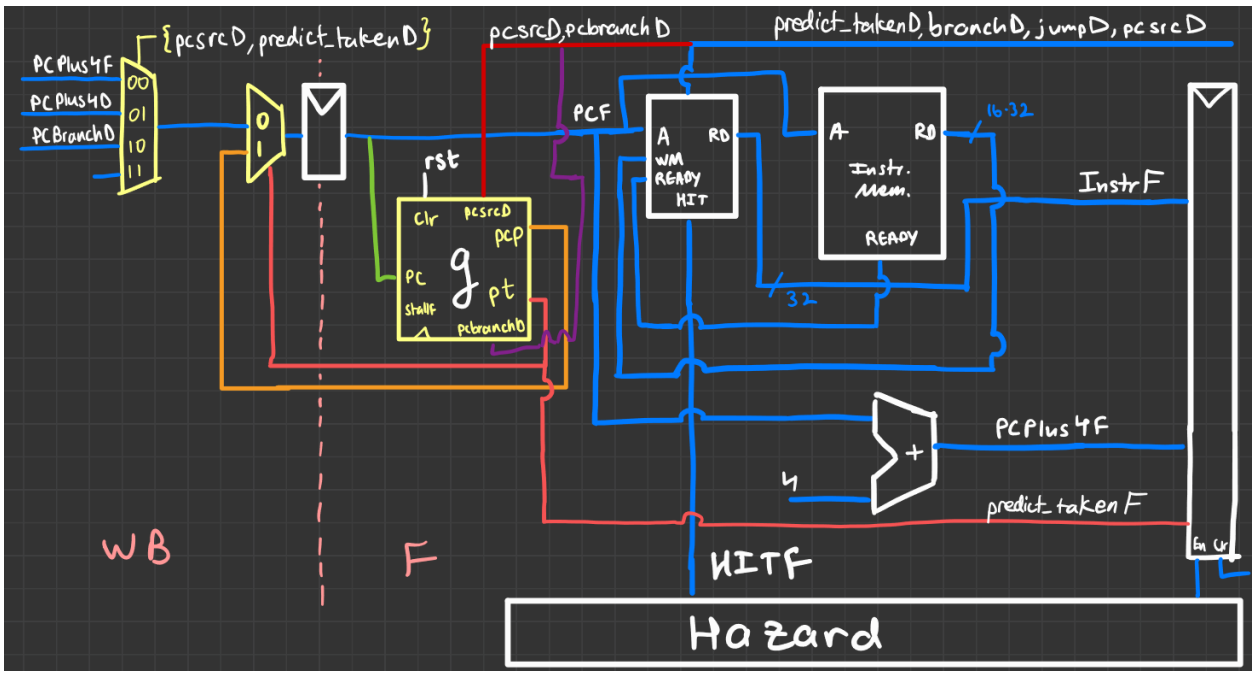
PC if it's a branch predict taken. A PC is mapped to an entry using its lower 7 bits. Once a PC is inputted into the module, the output is set asynchronously. The predicted branch target address and the predict taken bit is passed to the decode stage.

At every rising edge, the inputted PC is stored in an internal register storing the branch target buffer. The previously stored value in this register is used to update the FSM of the previous cycle's PC once it's determined in the decode stage if the branch is taken or not. If the decode stage indicates that the previously fetched instruction is a branch, it also updates the target branch address. This way, if there's a conflicting PC mapping that branches to a different address, this is updated in the branch target buffer.

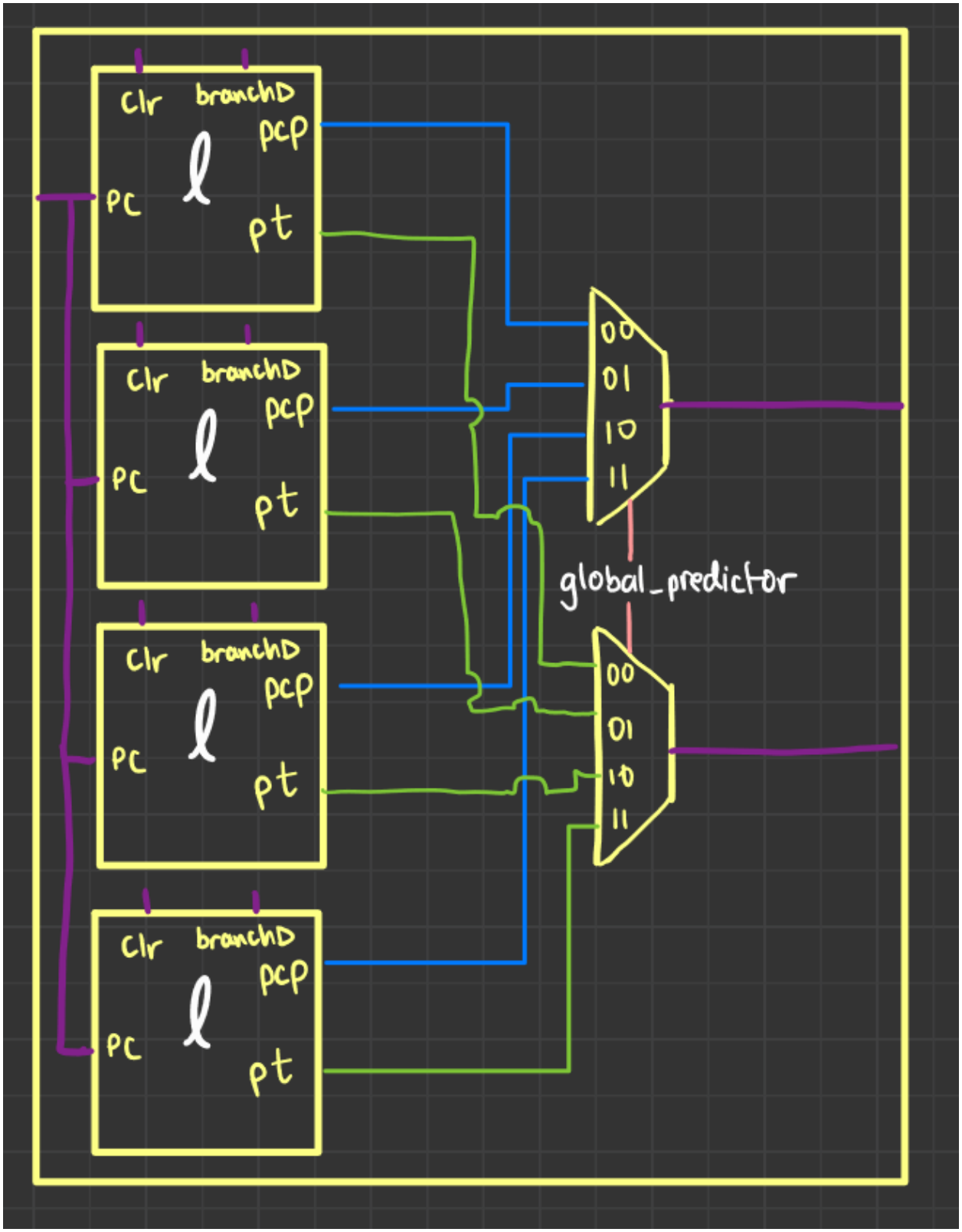
In the decode stage, we modified the condition in which we clear the pipeline register to insert a NOOP. This should now only happen under three conditions: (1) the branch is not taken but the predictor predicted branch taken, (2) the branch is taken but the predictor predicted branch not taken, or (3) the branch is taken but the predictor's target address does not match the branch target address that was decoded.

The last modification we had to make was to the writeback stage that determines the next value of PC passed into the fetch stage. As shown in the schematic, there are four cases: (1) the branch is predict taken (predict_takenF) and the PC should be the branch predict target address from the predictor module; (2) the branch predict from the previous cycle was taken but it was a branch not taken so the PC should be PCPlus4D; (3) the branch predict from the previous cycle was not taken but it was a branch taken so the PC should be PCBranchD; and (4) the branch predict was not taken so the PC should be PCPlus4F.

Correlating Prediction



Global Predictor - Writeback/Fetch Stage Modifications



Global Predictor Module Internal Schematic

For the correlating predictor, all our changes were contained in the fetch stage. We created a new module that represents the global branch predictor with all the same inputs and outputs as the local branch predictor. Then, we created four instances of the local branch predictor module within the global branch predictor. The outputs of the local branch predictor modules were multiplexed using the value of the global branch predictor state. This state is updated at every rising edge.

To ensure that the correct bits are updated upon encountering a branch, we used the state of the global predictor to determine which local branch predictor was used last in the global predictor. Even if pcsrcD was high, we used the global predictor bits to set the pcsrcD of the correct local branch predictor high.

Testing

Local Prediction

Sanity Check 1: Previous Project Programs

To ensure the processor did not lose any functionality, we ran the programs from the previous projects and compared the waveforms to the expected results. The results were consistent with what we had predicted.

Sanity Check 2: Nested For Loop

To test the behavior of the local branch predictor, we wrote a rudimentary program with a nested for loop that increments a counter variable. The C code and Assembly instructions are shown in the figures below.

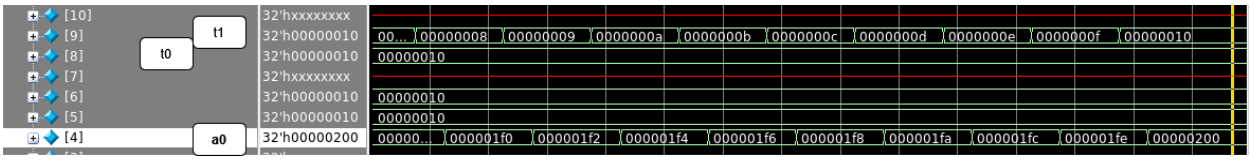
```c
int main() {
    int x = 0;
    for (int i = 0; i < 16; ++i) {
        for (int j = 0; j < 16; ++j) {
            x += 2;
        }
    }
}
```

Nested For Loop C Code

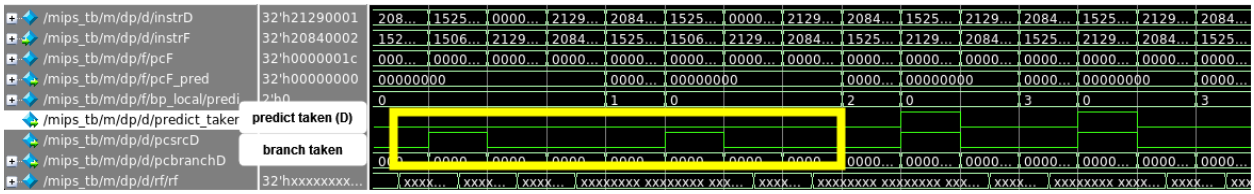
```
1   ADDI $a0 $zero 0x0000 # a0  = 0x00
2   ADDI $a1 $zero 0x0010 # a1  = 0x10
3   ADDI $a2 $zero 0x0010 # a2  = 0x10
4   ADDI $t0 $zero 0x0000 # t0  = 0x00 <- Outer for loop
5   ADDI $t0 $t0   0x0001 # t0 += 0x01 <- Inner for loop
6   ADDI $t1 $zero 0x0000 # t1  = 0x00
7   ADDI $t1 $t1   0x0001 # t1 += 0x01
8   ADDI $a0 $a0   0x0002 # a0 += 0x02
9   BNE  $t1 $a1   0xFFFD # if t1 != a1: PC -= 3*4 + 4 (Inner loop)
10  BNE  $t0 $a2   0xFFFA # if t0 != a2: PC -= 6*4 + 4 (Outer loop)
11  JAL                   0x000C
```

Nested For Loop Assembly Code

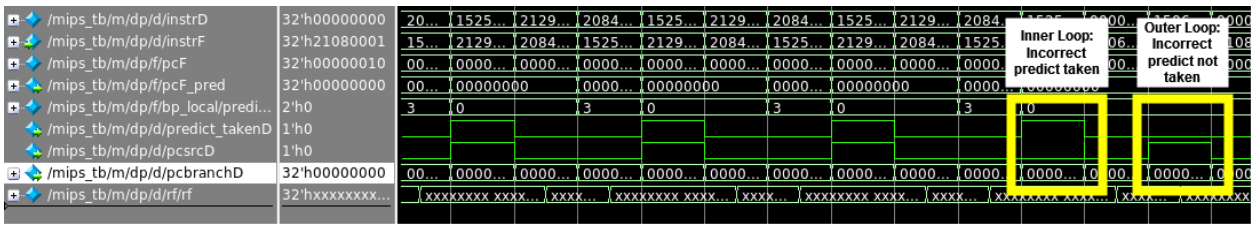
The expected value stored in the register once the program completes is 0x200. We can see that this matches the result stored in the register of the simulation in Modelsim.

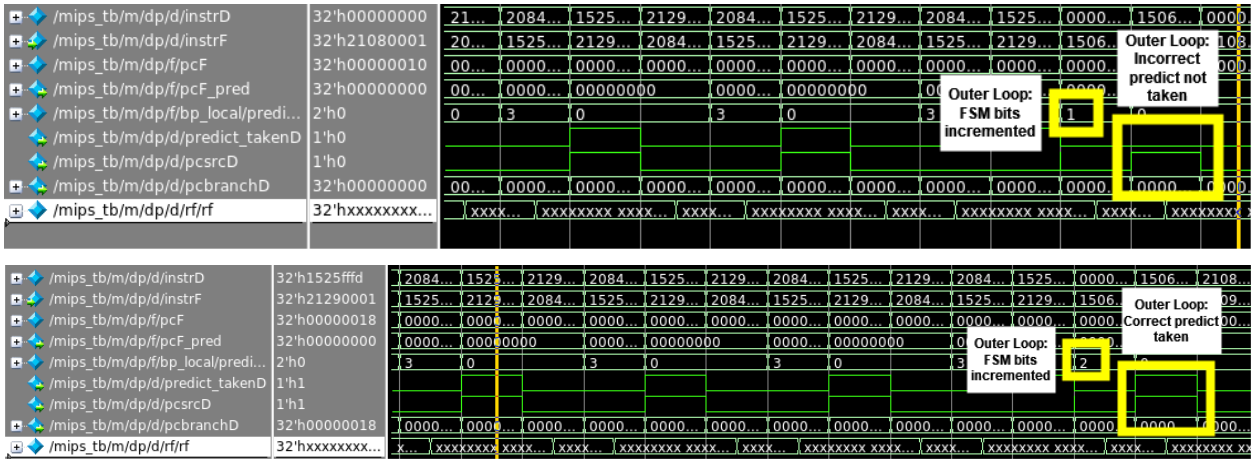


Now we'll take a closer look at the branching behavior at each iteration of the for loop. In the first two iterations, the branch predictor should be not taken.

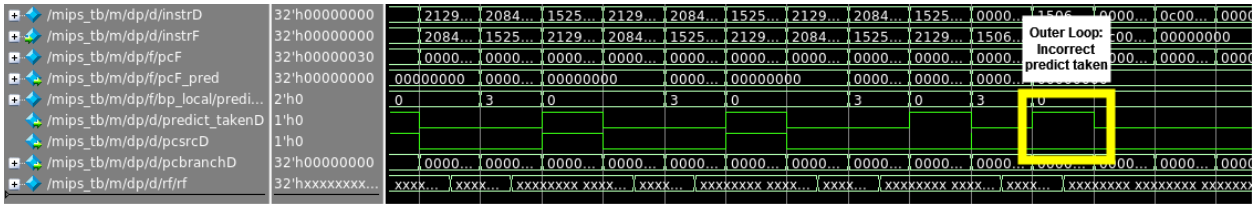


We can see that for the first two branches, the predictor predicts not taken. There is a NOOP inserted after the instruction to branch in the instrD signal for the first two branches. We can also see the value of the encoded bits of the finite state machine is incrementing at each branch. For the remainder of the inner loop's iterations, the predictor correctly predicts the branch and no NOOP is inserted.





For the last iteration of the inner loop, the predictor will incorrectly predict taken and a NOOP will be inserted. Then we encounter the second branch. This is the first time that the processor sees the outer loop's branch so it will, just like with the first branch, fail to correctly predict the first two iterations and the last iteration. In between, the processor will correctly predict taken.



In the last iteration of the outer loop, the predictor will incorrectly predict taken before proceeding to the end of the program.

Worst Case: Repeatedly Switching Branch Behavior

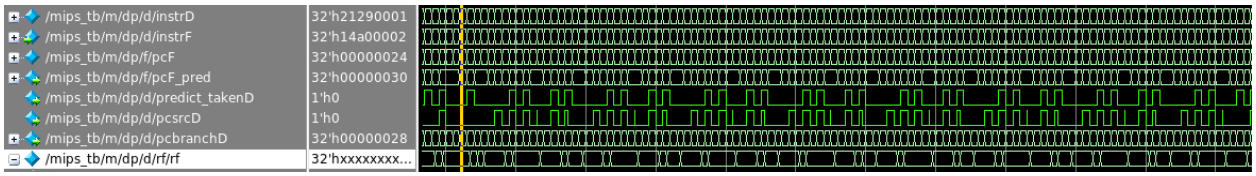
In the worst case, we have a situation where a branch's behavior changes every time the processor comes upon it. In this case, the local predictor will incorrectly predict at every chance and the state machine would flip back and forth between weakly taken and weakly not taken.

```c
int main() {
    int x = 0;
        int c = 0;
    for (int i = 0; i < 16; ++i) {
        for (int j = 0; j < 16; ++j) {
                        if (c = 0){
                                c += 1
                                x += 2
                        }
                        if (c = 1){
                                c -= 1
                                x += 1
                        }
            }
    }
}
```

Worst Case Behavior C Code

```
ADDI $a0 $zero 0x0000 # a0  = 0x00
ADDI $a1 $zero 0x0000 # a1  = 0x00
ADDI $a2 $zero 0x0010 # a2  = 0x10
ADDI $a3 $zero 0x0010 # a3  = 0x10
ADDI $t2 $zero 0x0001 # t2  = 0x01
ADDI $t0 $zero 0x0000 # t0  = 0x00
ADDI $t0 $t0   0x0001 # t0 += 0x01 <- outer loop
ADDI $t1 $zero 0x0000 # t1  = 0x00
ADDI $t1 $t1   0x0001 # t1 += 0x01 <- inner loop
BNE  $a1 $zero 0x0002 # if a1 != 0: PC += 2*4 + 4
ADDI $a0 $a0   0x0002 # a0 += 0x02
ADDI $a1 $a0   0x0001 # a1 += 0x01
BEQ  $a1 $zero 0x0002 # if a1 = 0: PC += 2*4 + 4
ADDI $a0 $a0   0x0002 # a0 += 0x02
SUB  $a1 $a1   $t2    # a1 -= 0x01
BNE  $t1 $a2   0xFFF8 # if t1 != a1: PC -= 8*4 + 4
BNE  $t0 $a3   0xFFF5 # if t0 != a3: PC -= 11*4 + 4
JAL                   0x0013
```

Worst Case Behavior Assembly Code



Worst Case Behavior Waveform Output

From a bird's-eye view, we can see that there are many more miss-predictions with our local predictor. This is expected because the branch for the if condition within the nested for loop never stabilizes to either branch taken or not taken. The correct predictions we do see are the result of the local branch predictor correctly predicting the branch behavior of the nested for loop. Note that there are parts of the waveform where the signal is high and the width seems thinner than other parts of the signal where it's high. This is because we write in the middle of the clock cycle to update the branch predictor FSM bits. Hence, there's only one real miss prediction and that corresponds to the incorrectly predicted behavior of the if statement.

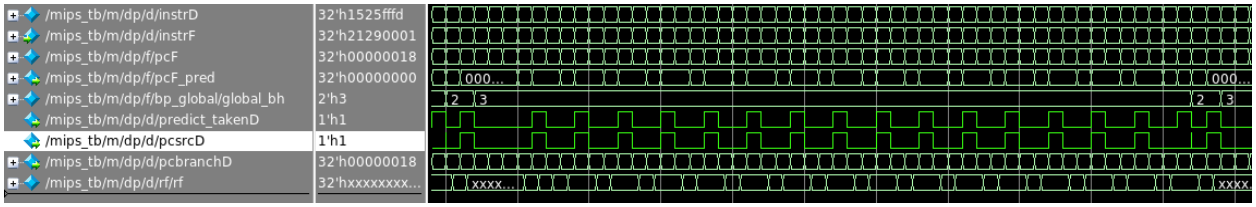
Correlating Prediction

Sanity Check 1: Previous Project Programs

To ensure the processor did not lose any functionality, we ran the programs from the previous projects and compared the waveforms to the expected results. The results were consistent with what we had predicted. Since most of the programs we wrote in the past did not involve complex branching behaviors, we noticed no difference in the waveforms between the local branch predictor and correlating predictor.

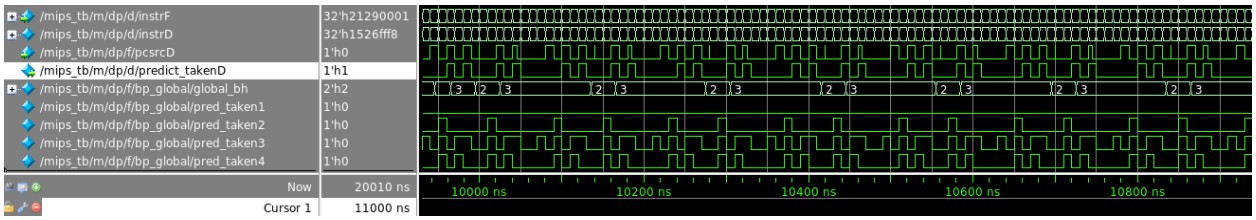
Sanity Check 2: Nested For Loop

We can get a strong intuition as to how well the correlating branch predictor learned the branching behavior by observing one of the last few iterations of the outer loop.



We can see that the correlating predictor correctly predicts all but one branch in an iteration of the outer loop. This happens to be the same behavior as the output of the local predictor. We hypothesize that this is because the program is too simple and does not take advantage of the additional complexity the correlating predictor provides.

Worst Case: Repeatedly Switching Branch Behavior



The correlating predictor does not fare any better than the local predictor. We hypothesize this is because the nested for loop clouds the temporal locality factor since the for loop's branch is the last branch before the if statement to execute. The global branch predictor register flips between 2 and 3 and since there are three branches, we believe that two the other two states of the

global branch predictor register are not being used to capture the if statement in the nested for loop. We tested this theory by running the program with the last branch deleted so that there's only one for loop. We found that this had no impact on the result and the branch predictor kept missing the if condition in the while loop.

Conclusion

By implementing a branch predictor, we can improve the CPI of our processor, especially for repetitive branching such as one found in a for/while loop. With testing, we can see that the branch predictor is able to pick up on repetitive branches well and no bubble is inserted in the best case. The predictor is still operating on a heuristic that can be summarized by the following statement: if we branched at a certain instruction before, we're likely to branch at the same instruction again.

We found that adding the local branch predictor had a strong positive impact on performance since we make a leap from assuming all branches are not taking to integrating some logic that's based on an intuitive heuristic. However, the global branch predictor did not show any improvement for any of our programs, including the worst case where there's a correlated if condition. We believe that a program that did not leverage a nested for loop would be able to better leverage the global branch predictor's temporal locality.