Introduction

Branch instructions are used to perform if statements and loops to control the flow of the program. With this processor design, the processor will not know whether a branch should be taken or not until the second (decode) stage with the early branch resolution. When the processor puts the next instruction address into the program counter, if it is wrong, then the processor needs to stall for one cycle to correct itself. The goal is to predict when to take the branch with the highest accuracy as possible. The prediction is made in the first (fetch) stage and the processor will not need to stall if it predicts the branch correctly.

Static Branch Prediction

Static branch prediction refers to the default prediction at the start of the program. This processor's prediction is initially set to zero or "always not taken". This means that the processor will initially load PC+4 to the program counter instead of the branch target address. If the branch is taken, then the processor will always take an extra cycle to stall.

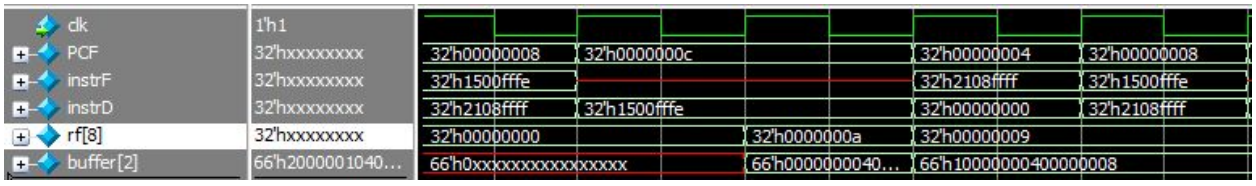
Branch Target Buffer

This buffer stores the branch instruction address, the corresponding branch target address, and a 2 bit saturating counter for the prediction of that instruction. If the instruction is a branch, then the necessary information is stored into the buffer, else PC+4 is stored and the program is executed as usual. If that branch is taken, then the prediction for that instruction increments by one, else it decrements by one. The processor still does not know if the branch was predicted correctly until the decode stage. If the branch was predicted incorrectly, then the processor has to stall for one extra cycle. Once the counter for a branch instruction reaches at least 2, then the processor will predict the branch to be taken instead of PC+4. If the branch is no longer taken enough times, then the processor's prediction will revert back to PC+4 for that branch instruction.

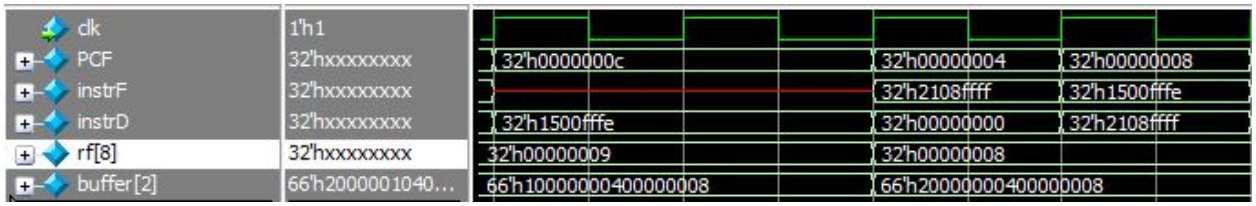
memfile5.dat

This program is a simple do-while loop from 10 down to 0 to test if the processor will start to make the prediction to take the branch after a certain number of cycles. rf[8] stores the iterator and buffer[2] stores the prediction for the branch instruction.

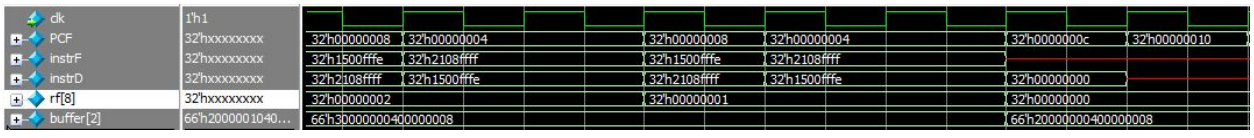
Initially the prediction is “branch not taken”, so the processor will predict PC+4 incorrectly and will be forced to waste a cycle stalling.



Once the prediction is at least 2, then the processor will start to make the prediction to take the branch. 0xC is no longer stored in PCF. 0x4 is stored instead, which means the branch is predicted correctly and instrD does not need to be flushed.



The processor will continue to correctly predict the branch until the end of the for loop. At the end when rf[8] is supposed to be 0, the branch is predicted incorrectly and the loop ends.



Two-level Correlating Global Branch Predictor

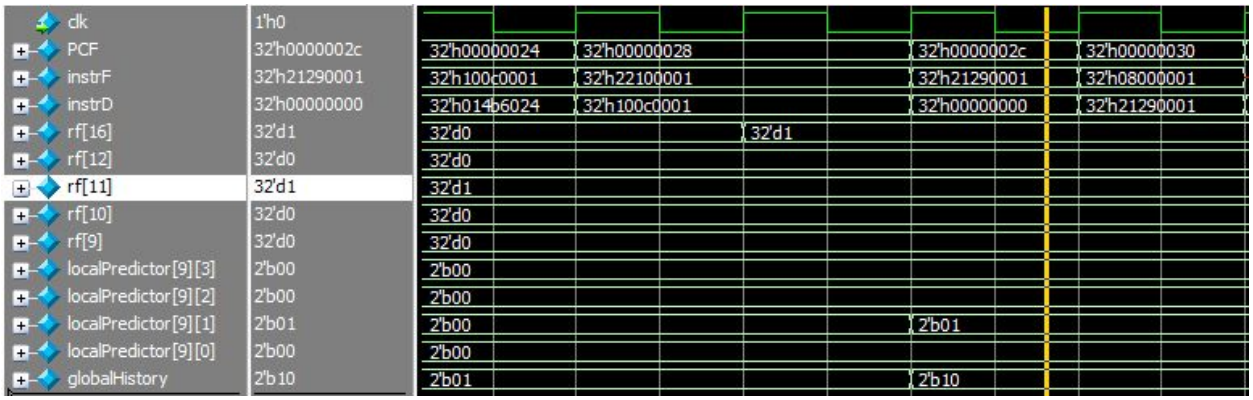
In programs, it is common for if statements and for/while loop conditions to be somewhat dependent on previous conditions, so a global branch predictor also keeps track of the previous branches taken to make a prediction. A two-level global branch history is a 2 bit shift register

that shifts the history of the previous two branches. Four different 2 bit local predictors are initialized and indexed with the global branch history. The branch target buffer is also initialized in the global branch predictor, but it is only used to store the branch target addresses. These local predictions are not used.

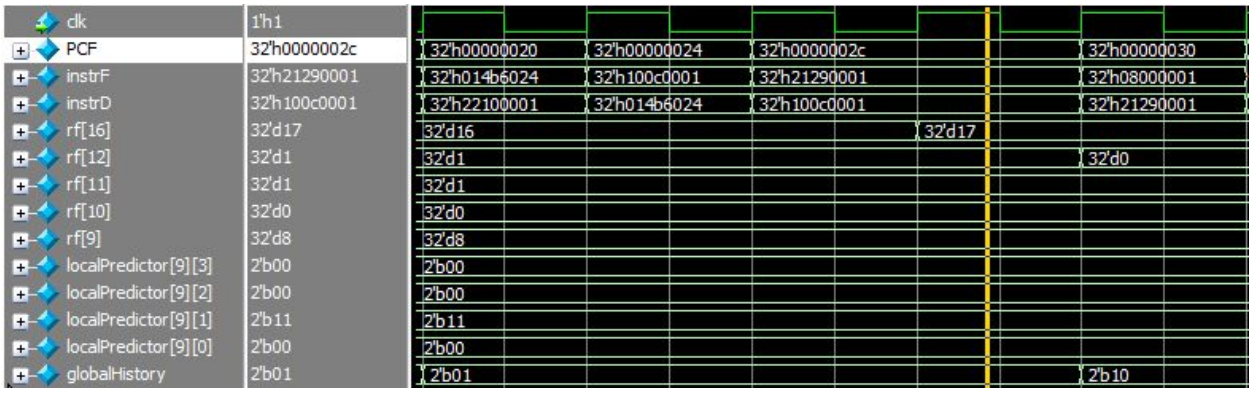
memfile6.dat

The program is structured as a for loop that counts from 0 to 19 with 3 if statements inside. The first if statement checks if the iterator is an odd number. The second if statement checks if the iterator is less than 15. The third if statements is the and of the first two if statements. This program tests if the last if statement can be predicted correctly once the branch history has been updated enough times using the first two if statements as the global history. The if statement condition being satisfied is equivalent to the branch not being taken. At the end of the program, the processor will predict to not take the last branch only if the first two branches were also not taken (globalHistory == 00). rf[9] is the iterator value. rf[10:12] stores the boolean values in the three if statements in ascending order. rf[16] keeps track of how many if statements are being satisfied. The only global predictor registers that are being examined are corresponding to the third if statement, which is in localPredictor[9].

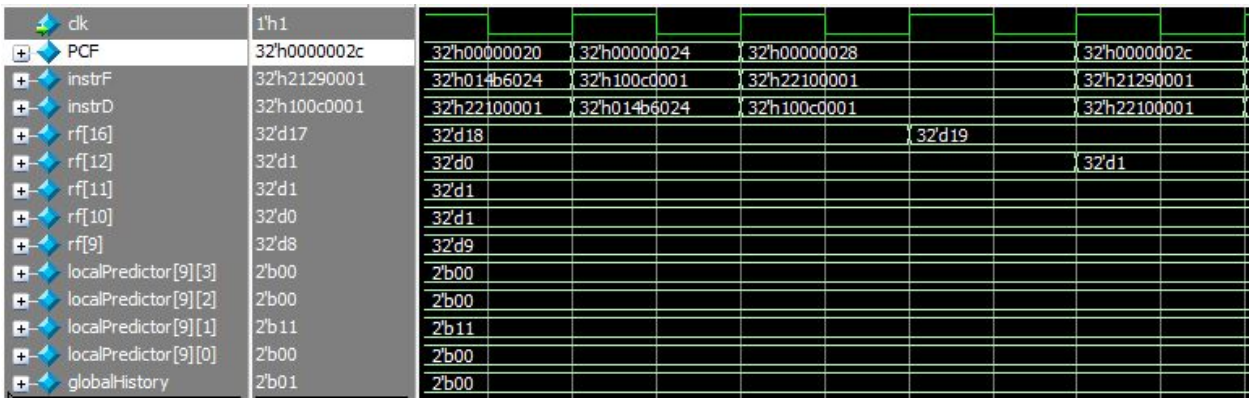
In the first iteration, when iterator == 0, the processor will predict the last branch was not taken, which is incorrect, so an extra cycle is used to input the correct address and flush the incorrect instruction (instrD == 32'h0).



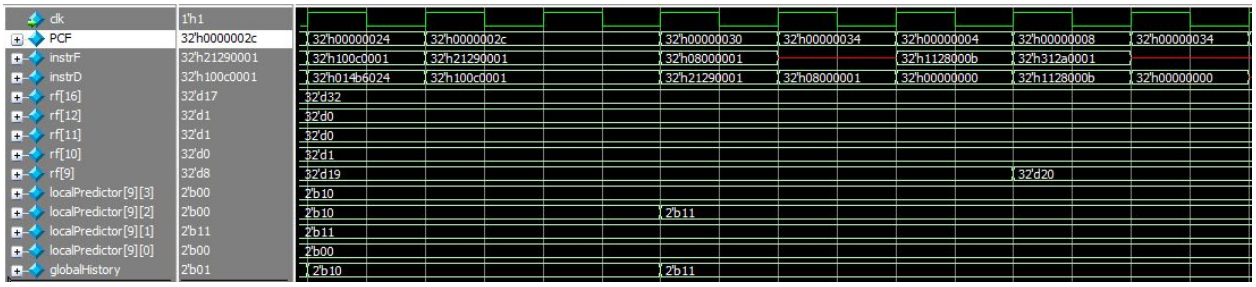
Once the program has executed a few cycles, the processor will make the prediction to branch if the iterator was an even number and less than 15 (globalHistory == b01). Instead of inserting 0x28 into PCF, 0x2C is stored instead. This takes away the need to flush the instruction and decreases the amount of cycles required to stall by 1.



On odd numbers that are less than 15, the processor will still predict correctly to not take the branch because localPredictor[9][0] still correctly makes the prediction that it should not take the branch if the previous two branches were also not taken.



Towards the end of the program, the numbers will start becoming greater than or equal to 15, so initially the prediction will be incorrect, but over a few iterations the predictions will get filled in. At the end, the processor will perfectly predict if you should take the third if statement given the history of the first and second if statements.



Conclusion

If the processor does not know when to take the branch until the decode stage, then it is possible to stall for an extra cycle every time a branch instruction is called. To fix this, the processor tries to predict the branch in the first cycle by storing the instruction addresses and using previous branch histories as a way to predict the current branch instruction. It is not always correct, but if there is a loop or statements that are dependent on other branches, then this is helpfully to place the correct instruction address into the program counter most of the time.