**ECE 586**

**Hardware Security and Advanced Computer Architecture**

**Branch Predictor Implementation**

*Acknowledgement – I acknowledge that I did not get any assistance from anyone or any other sources or copying from the Internet for completing this assignment.*

Date – 05/14/2021 Sign – Ishan Loganathan Reddy

ABSTRACT

This report encapsulates three different types of branch predictions which were implemented using VHDL language.

We created these three predictors by using different types of functional small blocks which were then in turn used in the implementation of an improved processor for choosing conditional branches. The goal of this project is to measure the effectiveness of various types of branch predictions (taken or non taken) on several implementations of conditional branch instructions.

We write a program in C or C++ that implement the above mentioned three different branch prediction schemes. We understand the working and real time implementation of the predictors.

INTRODUCTION

In the pipeline, the instruction i+1 will be launched before the instruction i is finished. With a branch present at each pipeline, the disadvantage here is to know what will be the next instruction to execute it. If the processor has to wait, a latency will occur and in turn be slowed by that.

The solution is to implement a branch predictor to find if the branch is taken or not taken and launch the next instruction following this prediction. If the prediction is right, the processor will continue without any latency. In the case of a misprediction, the next instruction is invalidated and the processor will start the right instruction on the next clock cycle, there is in this case a latency but a latency unavoidable because of the structure of a branch.

In this project we aim to implement the three branch predictors mentioned in the project statement. We implement the following predictors:

1-bit branch predictor

2-bit branch predictor

(3,2) correlating branch predictor

This project was given with three different problem statement. Hence, we can say that the project is split into the three parts as each design is an extension of the previously implemented design. And ultimately by implementing the final branch predictor a substantially complicated system was successfully designed and executed.

For creating the branch predictors we start with breaking down the 1-bit branch predictor. We build the 1-bit branch predictor in two different ways from scratch. Once with arrays instead of counters, and once with only counters, multiplexers, comparator and demultiplexer.

We targeted each unit implemented for the 1-bit and 2-bit branch predictor with small testbenches to test the functionality of the block. The first way suggested to build the 1-bit branch predictor design included using a table to store the predictions and only use one comparator and one counter for the misses. This system is very simple and acts more in the lines to test the functionality of the blocks.

In the second way suggested we build the 1-bit branch predictor using just counters, multiplexer, demultiplexers. After creating the 1-bit predictor using these two methods, we easily created the 2-bit branch predictor. And using both these predictor the final (3,2) correlating branch predictor was created and implemented. Here, the final aim for these branch predictors was to engage their own boundary of the number of misses each predictor gave to the system.

**LOCAL PREDICTOR AND GLOBAL PREDICTOR**

**LOCAL:**

The 1-bit and 2-bit predictors are local predictors. A local branch predictor has a separate history buffer for each conditional jump instruction. It may use a two-level adaptive predictor. The history buffer is separate for each conditional jump instruction, while the pattern history table may be separate as well or it may be shared between all conditional jumps.

We use a loop to calculate the chance that a branch will be taken if the branch before it is taken. The first predictor is a 1-bit predictor. That means that each branch will have a history of one bit, so only the last outcome for the branch. The basic understanding of the 1-bit branch prediction is that to go from taken prediction to a not taken prediction, the following two predictions must be not taken.

This allow the prediction to be more stable. The prediction will not change for one mistake.

**GLOBAL:**

The (3,2) correlating branch predictor is a global predictor implemented. A global branch predictor does not keep a separate history record for each conditional jump. Instead it keeps a shared history of all conditional jumps. The advantage of a shared history is that any correlation between different conditional jumps is part of making the predictions. The basic understanding of the global branch predictor is that the prediction is made using the outcomes of the past branches and not the history of the branch currently being used.

If the first branch is given a taken prediction, the next one will be taken too. With this theory, we implement the 3,2 correlating branch predictor. We store the branch predictions and their history stored in a register. This predictor keeps the outcomes of the previous three branches. These bits select one 3-bit counter over possible 8 bits. Hence, for each branch predicted, there are 8 possible counters over the previously taken 3 outcomes.

1-bit predictor is built using the following components:

* A clock for the counter
* A comparator by using XOR gates between the outcome and predict values
* An initializing signal to initialize the 1-bit and misses counters to 0
* A clk, initializing signal, adder and the outcome input ports are defined
* A predict and misses output ports are defined.

A 1-bit predictor stores the local history, selects a prediction from it, checks for a hit or a miss and then changes the value of the counter as applicable.

2-bit predictor

* The 2 bit counter uses saturating counters to implement the 2bit prediction. If the outcome input port is high or 1 then the counter is incremented, if the outcome is low or 0, the counter is decremented respectively.

3,2 Correlating Branch Predictor:

* Each branch has 8 2-bit saturating counters for each respective possibility from the last three outcomes. The components used (Demux and Mux) select one branch using the obtained memory location address from the last three outcomes.

**MIPS program of the code given:**

KEY FOR REFERENCE – s0 = c; t0 = I; t1 = j; t2 = 1000; t3 == 4;

//CODE //COMMENTS

move $s0, $zero # initializing c to 0

li $t0, 0 # initializing i to 0

li $t2, 1000 # storing value of j in a register for future predictions

B1: beq $t0, $t2, end # loop iteration 0 test

addi $s0, $s0, 1 # c++

addi $t0, $t0, 1 # i++

b B1

end:

ADD $s0, $zero, $zero # initializing c to 0

ADD $t0, $zero, $zero # initializing i to 0

ADD $t2, $zero, $zero # storing value of j in a register for comparison

ADDI $t2, $t2, 1000

ADD $t1, $zero, $zero # initializing j to 0

ADD $t3, $zero, $zero # storing value of j in a register for comparison

ADDI $t3, $t3, 4

B1: BEQ $t0, $t2, end # loop 1st iteration test

ADDI $t0, $t0, 1 # incrementing i by 1

ADD $t1, $zero, $zero # reinitialize j at 0

B2: BEQ $t1, $t3, B1 # loop 2nd iteration test

ADDI $s0, $s0, 1 # incrementing c by 1

ADDI $t1, $t1, 1 # incrementing j by 1

J B2

end:

**OUTPUTS FOR THE BRANCH PREDICTORS:**

**1-BIT PREDICTOR:**

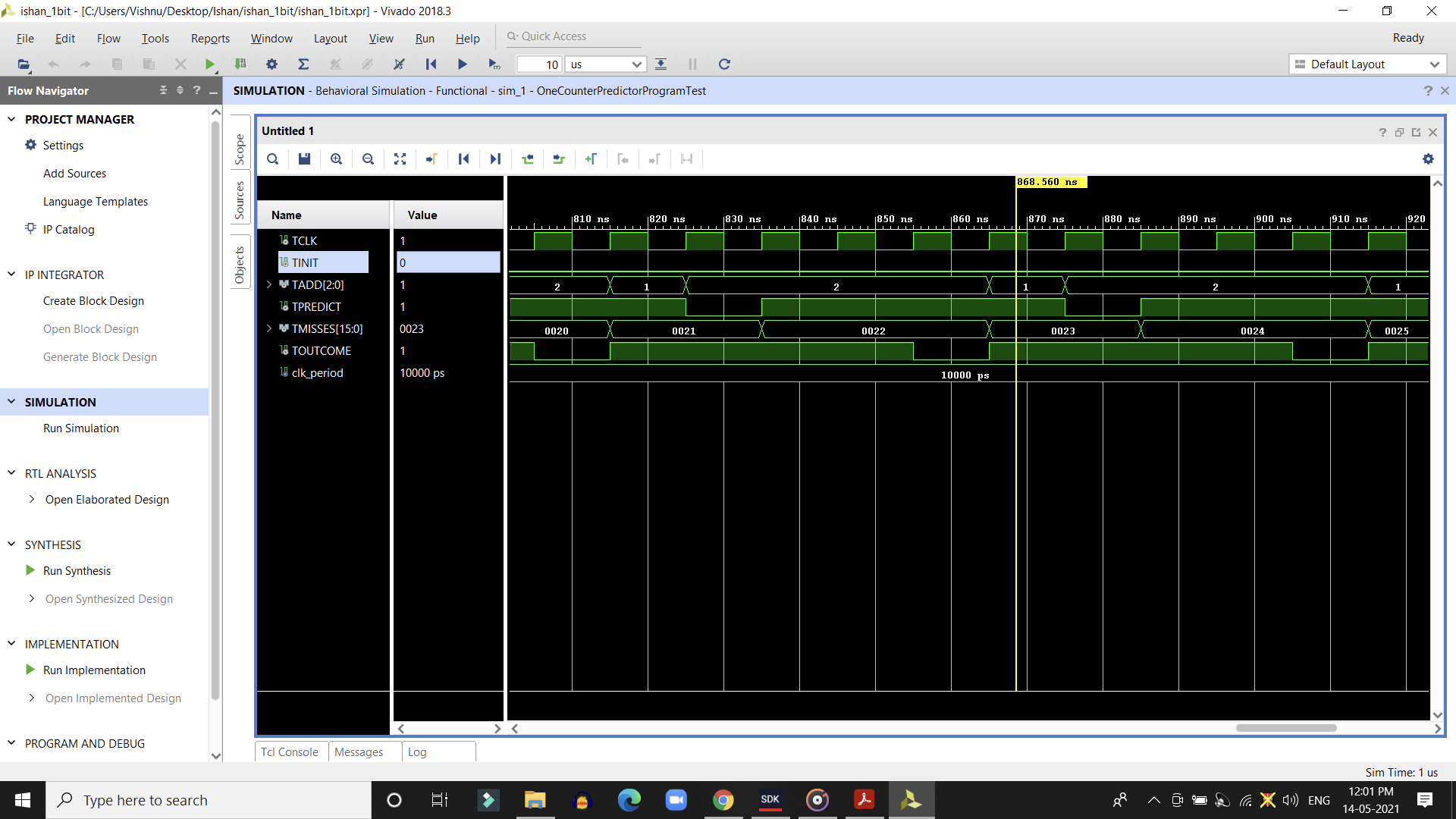


Figure - Simulation output for 1-bit predictor

As mentioned in the description of the 1-bit predictor, we implement 2 loops to predict the next branch. This is done using a counter from 0 to 3, involving a total of four iterations. We used a C file to create an input file for implementing in this branch predictor. This loop will follow the pattern of : T, T, T, NT.

As the prediction becomes the outcome of the last taken iteration the following output becomes as: NT, T, T, T.

**2-BIT PREDICTOR:**

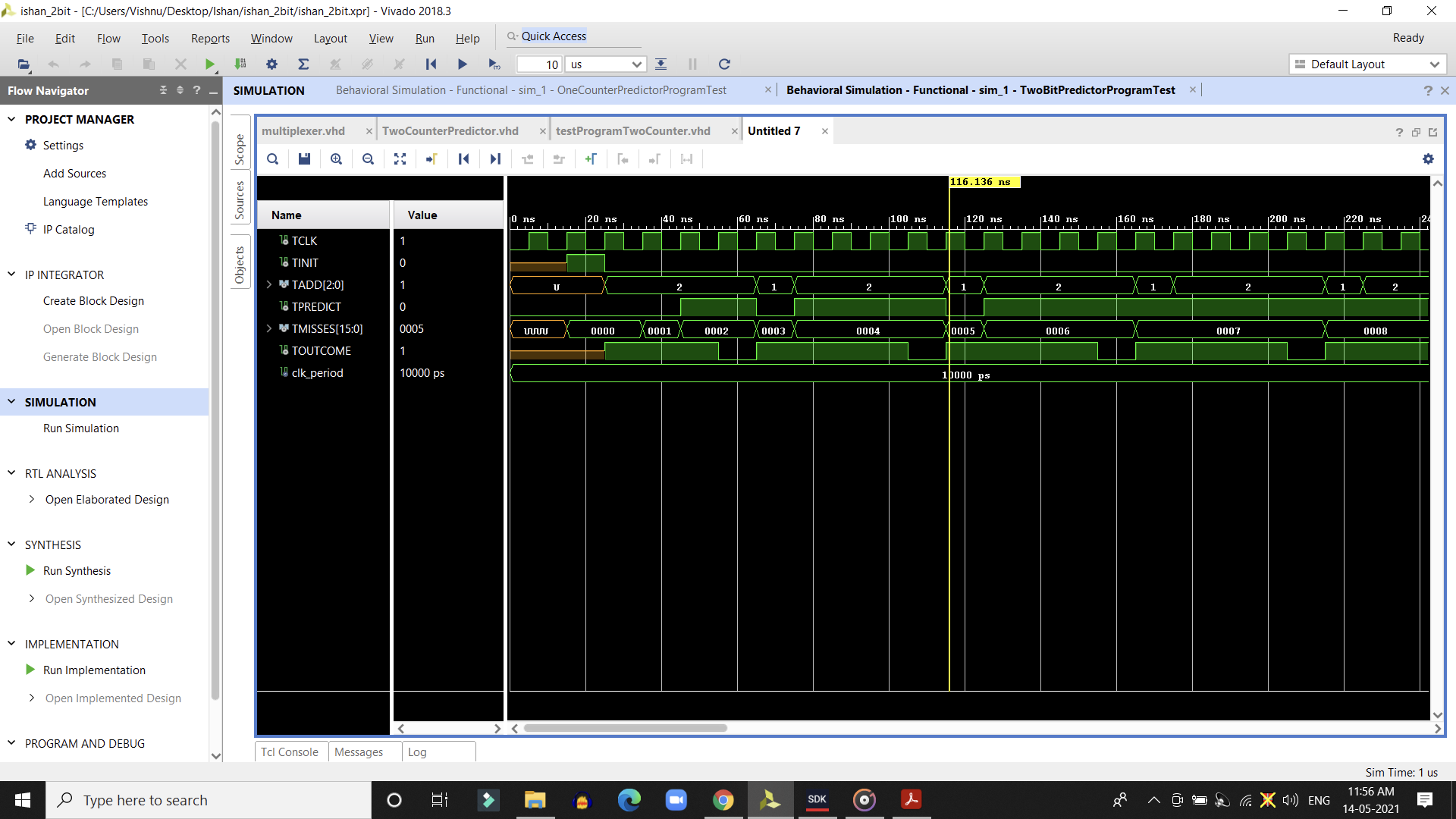


Figure – Simulation Output for 2-bit Branch Predictor

Here, the outcome for the 2-bit predictor resembles the output obtained in the 1-bit predictor. Even though that may be the case, their functionality Is very different. The same methodology of using two loops is implemented but here the first loop takes the pattern: NT, NT, T, T. This occurs as the 2-bit saturation counters initialize at 0 thus they take predictions in two iterations to go to the value 2. Hence, the output of T.

For the second loop all the iterations are considered as saturation counters are 3, hence, the loop takes the pattern of one NT and then another NT to return back to the not taken prediction. So now the prediction output is as follows for the second loop: T, T, T, T.

**3,2 CORRELATING PREDICTOR**

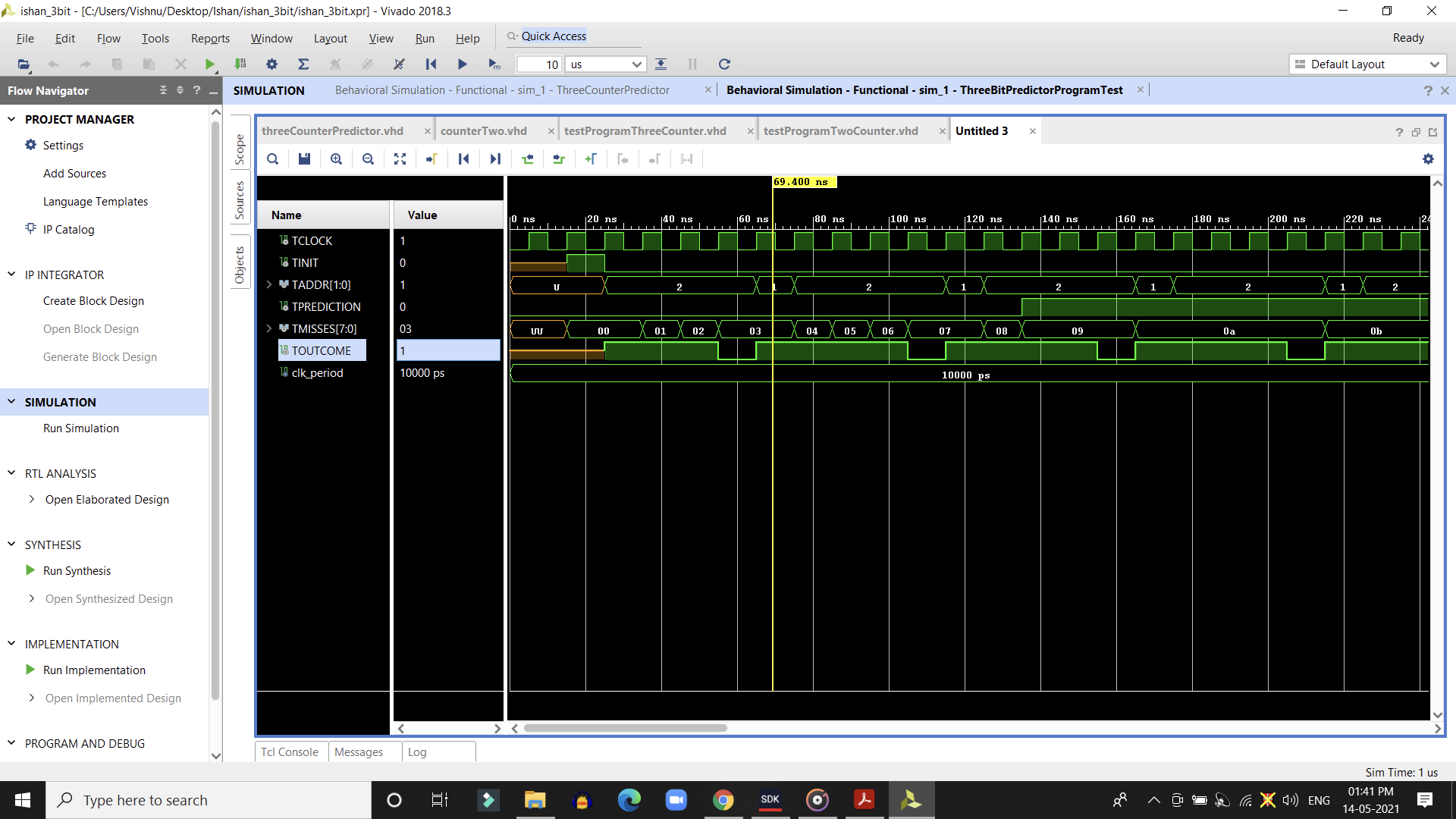


Figure – Simulation Result for 3,2 Correlating Branch Predictor

In this correlating branch predictor, the counter will be selected from one of the previous three outcomes. During the execution of the program.

**CONCLUSION**

In the implementation of the three branch predictors, we also implemented the use of testbenches for each separate components. The input file was created using a C script for generating an input to be given to the branch predictor input ports.

The 3,2 correlating branch predictor has the least miss rate compared to the other two branch predictors. The 3,2 correlating branch predictor compares results over the span of previous 3 outcomes hence, the miss rate is very minimal.

This project in conclusion has given us a great understanding of how branch prediction works. We also were given the opportunity to implement concepts relating to such processors.

**APPENDIX**

C script for generating input file:

// 1 – Branch Taken

// 2 – Branch Not Taken

#include <stdio.h>

**int** main() {

FILE \*fptr;

fptr = fopen("vhdl\_input.txt", "w"); //open the file to store the data

**for**(**int** i =0; i<1000; i++){

//fprintf(fptr, "1 T\n");

if(i != 0) //first time in the loop, no branch

fprintf(fptr, "1 1\n");

for(**int** j=0; j<4; j++){

//fprintf(fptr, "2 T\n");

if (j != 0) //first time in the loop, no branch

fprintf(fptr, "2 1\n");

}

//fprintf(fptr, "2 NT\n");

fprintf(fptr, "2 0\n");

}

//fprintf(fptr, "1 NT\n");

fprintf(fptr, "1 0\n");

return 0;

}

**REFERENCES:**

J. L. Hennesy & D. A. Patterson Computer Architecture, A Quantitative Approach, 6th edition.