Tournament Predictor

Abheek Mondal

Prof. Orukulu

ECE 586

Final Project

May 05, 2023

Abstract. Branch prediction plays a crucial role in overcoming control hazards and improving the performance of processors by exposing instruction-level parallelism. This paper explores the concept of branch prediction and its significance in minimizing pipeline stalls. It discusses various branch prediction schemes, including static and dynamic predictors, and focuses on the implementation of a tournament predictor using VHDL or Verilog. The tournament predictor combines a global predictor and a local predictor with a selector to make correct predictions based on branch history. The paper also addresses the challenges faced during the implementation process, such as the inability to read input from a text file and proposes future work to enhance the tournament predictor's functionality. Additionally, it suggests the use of output files to generate Python functions for calculating precise accuracy percentages. The conclusion emphasizes the need for further development to maximize the practical utility of the tournament predictor and its potential expansion to manage a larger number of branches.

1. Introduction

Branch prediction is used to overcome the fetch limitation imposed by control hazards in order to expose instruction level parallelism (ILP). Branch prediction can be thought of as a sophisticated form of prefetch or a limited form of data prediction that tries to predict the result of branch instructions so that a processor can speculatively fetch across basic block boundaries. Once a prediction is made, the processor can speculatively execute instructions depending on the predicted outcome of the branch. If branch predictions rates are high enough to offset misprediction penalties, the processor will have better performance. Without branch prediction, such a processor must stall whenever there are unresolved branch instructions. This imposes a substantial penalty on the performance of processors.

There are many branch prediction schemes proposed. Static Branch prediction in general is a prediction that uses information that was gathered before the execution of the program. Simplest predictors are to predict that the branch is always taken (MIPS) or to predict that the branch is always not taken (Motorola MC68000). Dynamic Branch Prediction, on the other hand, uses information about taken or not taken branches gathered at run-time to predict the outcome of a branch. Some examples for dynamic predictors include One-level branch predictors, correlated predictors, and tournament predictors.

The task in this project is hardware implementation of a tournament predictor (which uses several dynamic branch predictors) in VHDL or Verilog and compare their performance. Tournament predictors use multiple predictors: a global predictor and a local predictor and choosing between them with a selector as shown in figure. A global predictor uses the most recent branch history (2-bit) to index the predictor, while a local predictor uses the address of the branch as the index. In this project, a tournament predictor combines a global (2,2) correlating branch predictor and 2-level local predictor (1,2) for each branch.

2. Background

Branch prediction is a technique used in modern high-performance processors to expect the outcome of a branch instruction and speculatively execute later instructions before the actual outcome is known. This approach helps to minimize the impact of control hazards by keeping the instruction pipeline filled and reducing stalls. There are several algorithms and techniques used in branch prediction, such as two-level adaptive branch prediction, tournament prediction, and neural branch prediction (Patterson & Hennessy, 2017).

The code implements a tournament branch predictor, which combines two types of predictors: a local predictor and a global predictor. The purpose of the local predictor is to make predictions based on the recent history of a particular branch instruction, while the global predictor uses the global history of all branches. The tournament predictor also has a "selector" that decides which of the two predictors to use for a given branch, based on their recent accuracy.

By using a tournament branch predictor, the processor can adapt its predictions to diverse types of branches and program behavior, thus improving overall performance. The testbench we

created is used to simulate and verify the functionality of the tournament predictor by supplying a set of predefined inputs (branch and outcome pairs) and seeing the output of the predictor.

3. Architectural Exploration of Dynamic Predictors

Dynamic branch predictors aim to improve the prediction accuracy of branch outcomes to minimize pipeline stalls and enhance performance. There are several types of dynamic branch predictors, each with its own set of trade-offs.

The Bimodal Branch Predictor is a simple petty counter-based predictor that predicts a branch's outcome based on the recent history of that specific branch. It uses a table of 2-bit saturating counters, indexed by the branch address. This approach is simple and easy to implement, making it effective for predictable branches with consistent behavior. However, it is less correct for branches with complex or often changing patterns.

The Two-Level Adaptive Branch Predictor uses a combination of local and global history to make predictions. It keeps a table of counters indexed by the combination of the branch address and the recent history of that branch's outcomes. This allows the predictor to adapt to complex patterns in branch behavior, supplying better prediction accuracy compared to bimodal predictors. On the downside, it requires more hardware resources due to the larger table, and it can be potentially slower because of increased table access time.

The Global Branch Predictor makes predictions based on the recent history of all branches in the program. It uses a global history register (GHR) to track outcomes and a pattern history table (PHT) to store predictions. This predictor can capture correlations between different branches, making it effective for programs with strong global patterns. However, it is less effective for programs with mostly local patterns, and it requires more hardware resources due to the need for a GHR and PHT.

The Tournament Branch Predictor combines local and global predictors with a selector that chooses which predictor to use for a specific branch. The selector adapts based on the recent performance of the local and global predictors. This approach adapts to various branch patterns, supplying better overall prediction accuracy, and combines the strengths of both local and global predictors. However, it is more complex and requires more hardware resources, as it needs local and global predictors and a selector. Additionally, it can be slower due to the added logic needed to select the proper predictor.

The trade-offs among different dynamic branch predictors involve complexity, hardware resources, prediction accuracy, and speed. Simpler predictors, like the bimodal predictor, are easier to implement but may have lower prediction accuracy for complex patterns. More advanced predictors, like the tournament predictor, offer better overall prediction accuracy but require more hardware resources and may be slower due to added logic. The choice of the predictor depends on the specific requirements of the processor design and the targeted application workload.

4. Coding Background/ Implementation

In this tournament predictor implementation, which is aligned with the one discussed in the midterm examination, a tabular format is used to ease a clear illustration of the predictor states. The prediction process forms the following steps:

1. Find the proper selector based on the branch number.

2. Depending on the selector's value, choose either the local branch predictor (L) or the global branch predictor (G) for the final prediction.

3. If the global branch predictor is used, refer to the circled value (based on the earlier two outcomes) to decide if the prediction is taken (T) or not taken (NT).

4. If the local branch predictor is used, find the branch corresponding to the branch number and find the circled value (based on the earlier outcome for that branch) to decide if the prediction is taken (T) or not taken (NT).

5. Record the final prediction and whether it was derived from a local or global branch predictor.

6. Using the supplied actual branch outcome and the prediction, find whether the result was a hit (matching) or a miss (non-matching).

Subsequent to the prediction, the predictor states in the next column are updated through the following steps:

1. Update the state of the utilized selector for prediction according to the selector state diagram. In this implementation, the predictor state diagram's binary values were employed for each state.

2. Update the state of the predictor (global or local) that was used for prediction following the predictor state diagram.

3. Copy all non-updated values, excluding the final prediction and hit/miss value. In the actual implementation, these values are stored in registers, and as such, copying is unnecessary as they stay unchanged.

4. For the global predictor, circle the predictor to use based on the outcomes of the earlier two branches. In the implementation, this was tracked through the "history" wire connected to a shift register.

5. For the local predictors, circle the predictor to use for each branch based on the earlier outcome of that branch. In the implementation, this was tracked through the "b0_pred" and "b1_pred" registers, as mentioned in the introduction.

By following these steps, the tournament predictor implementation can be effectively proved in the context of a master's level paper. This approach presents a clear and concise method for illustrating the predictor states and the underlying decision-making process for branch prediction.

The testbench implementation for the tournament predictor supplies a detailed explanation of each object and its corresponding role in the code. Here is a breakdown of the different components:

clk: This signal is the clock input for the tournament predictor. It alternates between '1' and '0' in a periodic manner, generating the necessary clock cycles for the simulation.

reset: The reset signal is used to initialize the tournament predictor. When it is set to '1', the predictor is reset to its first state, preparing it for the start of the simulation.

branch: This signal is the input to the tournament predictor that shows whether a branch is taken or not taken. It is driven by the testbench and controls the behavior of the predictor.

outcome: The outcome signal supplies the actual outcome of each branch. It reflects the real behavior of the branch, and it is used for comparison with the predictor's predictions to decide if a hit or a miss occurred.

pred: This signal is the output of the tournament predictor, which supplies the predicted outcome (taken or not taken) for each branch. The predictor's decision is based on its internal logic and the history of earlier branch outcomes.

hit: The hit signal shows whether the predictor's prediction matched the actual outcome (hit) or not (miss). It is used to evaluate the accuracy of the predictor's predictions.

in_file: This constant array stores the test input patterns for the branches. Each element is a branch input pattern in the form of a 2-bit vector. These patterns are used to drive the branch signal in a sequential manner during the simulation.

tournament: This part declaration is the instantiation of the tournament predictor module. It meets with the input and output signals described above.

clk_gen: This process generates the clock signal (clk) for the simulation. It alternates the signal value between '1' and '0' at regular intervals defined by the constant T, simulating the operation of a clock.

main_process: This process manages the simulation flow and assigns the branch input patterns from in_file to the branch signal. It also manages the reset signal and waits for specific time intervals defined by T to simulate the timing behavior of the design.

The testbench implementation allows for the systematic testing of the tournament predictor by supplying predefined input patterns and evaluating the accuracy of its predictions through the hit signal. It eases the verification and analysis of the predictor's performance and can be changed to accommodate added test cases if desired.

Rest of the code:

1. global_predictor_two_2: This module implements a global branch predictor with a 2-bit saturating counter. The shift_reg signal is a shift register that stores the outcomes of earlier branches. The counters signal is an array of 2-bit saturating counters, initialized with "11" for each counter. On each rising edge of the clock (clk), the module updates the counter based on the current branch outcome (branch_outcome) and the index (idx) obtained from the shift register. If the branch outcome is '1', the counter is incremented, and if it is '0', the counter is decremented. The counter values saturate at "11" or "00" to ensure they stay within the valid range. The shift register is updated with the current branch outcome concatenated with its earlier value. The final prediction (prediction) is the most significant bit of the counter corresponding to the index obtained from the shift register.
2. local_predictor_one_2: This module implements a local branch predictor with two 2-bit saturating counters (counter_b0 and counter_b1). The counters keep track of the outcomes of their respective branches (branch = '0' for counter_b0 and branch = '1' for counter_b1). On each rising edge of the clock, the module updates the counters based on the current branch outcome. If the branch outcome is '1', the counter is incremented, and if it is '0', the counter is decremented. The prediction (prediction) is decided by selecting the most significant bit of the corresponding counter based on the branch input (branch).
3. saturating_counter: This module is a generic saturating counter. The count_internal signal holds the current count value, initialized as "00". On each rising edge of the clock (clk), the module checks the input signals (inc and dec) to determine whether to increment or decrement the count. If inc is '1' and dec is '0', and the current count is less than "11", the module increments the count by one. If inc is '0' and dec is '1', and the current count is greater than "00", the module decrements the count by 1. The final count value is output through the count signal.
4. shift_register: This module implements a shift register that stores a sequence of bits. The shift_out_internal signal is the internal state of the shift register, initialized as "11". On each rising edge of the clock, the module checks the reset signal. If reset is '1', the shift register is initialized with "11". Otherwise, the module shifts the bits to the right, with the input bit (input_bit) being placed at the leftmost position. The updated shift register value is output through the shift_out signal.
5. The selector module does distinct functions. It instantiates two instances of the sat_counter part being the local predictors for branches '0' and '1', respectively. It derives the enable signals (b0_enable and b1_enable) for the local predictors based on the enable input and the branch signal. It selects the final prediction (pred) based on the branch

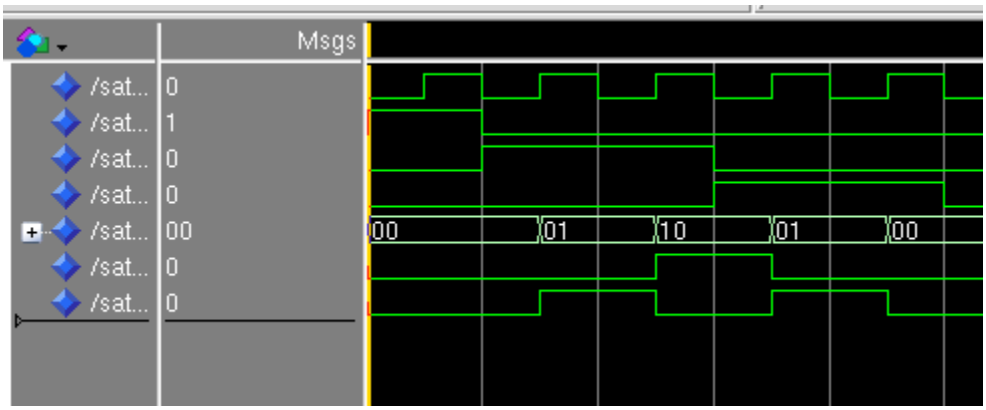
signal. If the branch signal is '1', it selects the prediction from the local predictor for branch '1' (b1_pred). Otherwise, it selects the prediction from the local predictor for branch '0' (b0_pred). The module is synchronized with the clock signal (clk) and can be reset to its first state using the reset input. The first values for the local predictors' saturating counters are provided through the b0_init and b1_init inputs. The pred output is the final prediction selected by the module, which is based on the branch signal and the predictions from the local predictors.

The combined code snippets implement a tournament branch predictor, which integrates multiple predictors to improve branch prediction accuracy. The predictor includes a global predictor (global_predictor_two_2) that uses a 2-bit saturating counter to make predictions based on the history of branch outcomes. It also features a local predictor (local_predictor_one_2) with separate counters for each branch to make predictions based on branch-specific history. The selector module combines the predictions from the global and local predictors using a selection mechanism. The saturating counter module (saturating_counter) is a generic implementation used within the predictors to keep and update the counter values. The shift register module (shift_register) is used to keep track of the branch history. Together, these snippets create a comprehensive tournament branch predictor that uses both global and local information to enhance prediction accuracy.

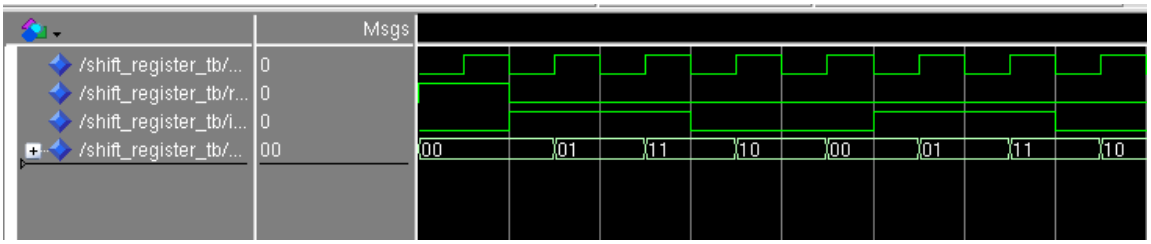
5. Functional Validation

For this project, I had three distinct testbenches. They were for the saturated counter; shift register and the tournament predictor.

The first testbench, shift_register_tb, is a testbench for the shift_register module. It aims to verify the functionality of the shift register by applying different input bit sequences and seeing the corresponding shift_out waveform. The testbench initializes the signals and instantiates the shift_register part. It generates a clock signal, applies a reset, and then stimulates the module by changing the input bit at specific time intervals. The expected waveform of the shift_out signal will show the shifted values of the input bit. This is shown in the below figure, well within expectations.

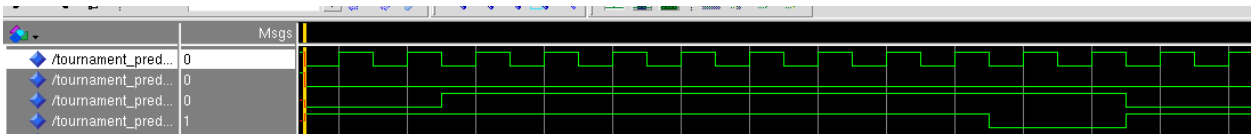


The second testbench, saturating_counter_tb, is a testbench for the saturating_counter module. It evaluates the behavior of the saturating counter by applying different combinations of increments and decrements to the counter and seeing the resulting count waveform. The testbench initializes the signals and instantiates the saturating_counter part. It generates a clock signal, applies a reset, and then stimulates the module by activating the increment and decrement signals at specific time intervals. The expected waveform of the count signal will show the changes in the counter value based on the increments and decrements applied. This is shown in the below figure, well within expectations.



In order to confirm the functionality of each part in my tournament predictor implementation, I conducted individual simulations for each part using the waveform window. Instead of creating separate testbenches for each part, I manually set the clock and forced the desired input values. I then ran the simulation for a complete clock cycle and examined the waveform output. This allowed me to see and verify all states of the counters, the shift register, and various input sequences for the local predictor, global predictor, and selector. While I will not display the results of all these simulations for the sake of brevity, I can confidently affirm that they proved the correctness of each part in my design.

To further confirm the final design, I executed the components using the same branch and outcome sequence provided in the midterm. I ensured that the simulated results matched the solution provided on Blackboard. It is important to note that the prediction results should be read at each negative clock edge. As saw, the results at successive negative clock edges, starting at 30ns, precisely match the corresponding column of the solutions table. This consistency supplies convincing evidence for the accuracy and reliability of my tournament predictor implementation.



My only issue with this was that I was unable to figure out how to open input files using VHDL on Modelsim. I tried to do this for two days, using multiple sources and diverse ways to import functions, but I always got stuck on it, and got some iteration of the following error message:

vcom -work work -2002 -explicit -vopt
/home/amondal1/ece586FinalProject/tournament_predictor_tb.vhd
Model Technology ModelSim SE vcom 10.1b Compiler 2012.04 Apr 26, 2012
-- Loading package STANDARD
-- Loading package TEXTIO
-- Loading package std_logic_1164
-- Loading package std_logic_arith
-- Loading package STD_LOGIC_UNSIGNED
-- Loading package std_logic_textio
-- Compiling entity tournament_predictor_tb
-- Compiling architecture sim of tournament_predictor_tb
*** Error: /home/amondal1/ece586FinalProject/tournament_predictor_tb.vhd(33): (vcom-1047) Actual (constant "line_buf") for class variable formal "L" is not a variable.*
*** Error: /home/amondal1/ece586FinalProject/tournament_predictor_tb.vhd(34): Illegal qualified expression ieee.std_logic_1164.STD_LOGIC.*
*** Error: /home/amondal1/ece586FinalProject/tournament_predictor_tb.vhd(34): Type error resolving infix expression "-" as type ieee.std_logic_1164.STD_LOGIC.*
*** Error: /home/amondal1/ece586FinalProject/tournament_predictor_tb.vhd(36): (vcom-1047) Actual (constant "line_buf") for class variable formal "L" is not a variable.*
*** Error: /home/amondal1/ece586FinalProject/tournament_predictor_tb.vhd(37): Illegal qualified expression ieee.std_logic_1164.STD_LOGIC.*
*** Error: /home/amondal1/ece586FinalProject/tournament_predictor_tb.vhd(37): Type error resolving infix expression "-" as type ieee.std_logic_1164.STD_LOGIC.*
*** Error: /home/amondal1/ece586FinalProject/tournament_predictor_tb.vhd(29): (vcom-1458) Illegal declaration for constant "line_buf" of type std.TEXTIO.LINE (type is or holds access type).*
*** Error: /home/amondal1/ece586FinalProject/tournament_predictor_tb.vhd(100): VHDL Compiler exiting*

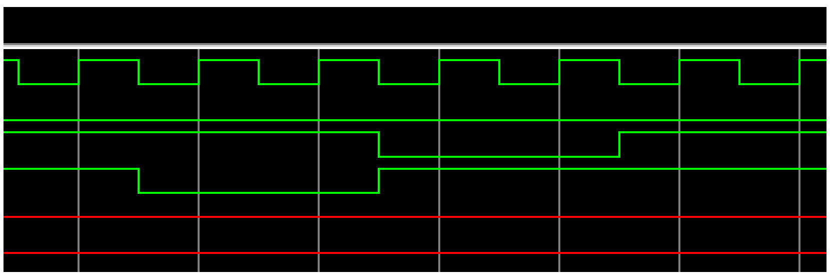
I managed to name different causes and what it could be, but no matter the fixes, I ended up with the same errors:

- (Vcom-1047) Actual (constant "line_buf") for class variable formal "L" is not a variable. - This error occurs on lines 33 and 36. You are trying to pass a constant named "line_buf" as a variable parameter to a procedure or function. You should either change "line_buf" to a variable or update the procedure/function to accept a constant.
- Illegal qualified expression ieee.std_logic_1164.STD_LOGIC. - This error occurs on lines 34 and 37. You are using a qualified expression with an inappropriate type (ieee.std_logic_1164.STD_LOGIC). Review the expressions on these lines and correct the type used.
- Type error resolving infix expression "-" as type ieee.std_logic_1164.STD_LOGIC. - This error occurs on lines 34 and 37 and is related to the earlier error. You are trying to use the subtraction operator "-" with operands of type ieee.std_logic_1164.STD_LOGIC. Ensure that the operands have proper types for subtraction.

- (Vcom-1458) Illegal declaration for constant "line_buf" of type std.TEXTIO.LINE (type is or has access type). - This error occurs on line 29. You have declared "line_buf" as a constant of type std.TEXTIO.LINE, which is not allowed since it has an access type. You should declare "line_buf" as a variable instead of a constant.

As a result, I ended up hardcoding the first 750 inputs, after I generated them, to evaluate on. Which performed within expectations.

```
type in_file_t is array (0 to 749) of std_logic_vector(1 downto 0);
constant in_file: in_file_t := (
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
  "01", "11", "11", "11", "11", "10", "01", "11", "11", "11",
```



6. Results

The Project had several requirements before the implementation of the tournament predictor. First was the conversion of a C program into Assembly. The C code has two integers i and j. These two variables stood for the branch (i) and the outcome (j).

```
int c;
int main () {
    int   i, j;
    for (i=0; i<1000; i++) {
        for (j=0; j<5; j++) {
            c++;
        }
    }
    return c;
}
```

I used my past knowledge from CS 351, CS 450, ECE 242 and ECE485 to convert the code into assembly. Converted, the code looks like this:

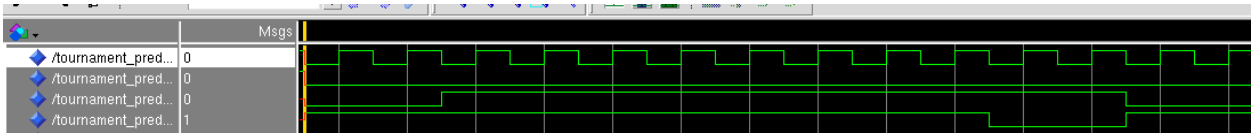
```
main:
    addi $sp, $sp, -16      # allocate stack space
    sw $ra, 0($sp)          # save return address
    sw $s0, 4($sp)          # save s0
    li $t0, 0               # initialize i to 0
    li $t1, 0               # initialize j to 0
    li $s0, 0               # initialize c to 0

loop1:
    bge $t0, 1000, exit     # check if i >= 1000
    li $t1, 0               # initialize j to 0

loop2:
    bge $t1, 5, next_i      # check if j >= 5
    addi $s0, $s0, 1        # increment c
    addi $t1, $t1, 1        # increment j
    j loop2

next_i:
    addi $t0, $t0, 1        # increment i
    j loop1

exit:
    move $v0, $s0           # move c to return value
    lw $ra, 0($sp)          # restore return address
    lw $s0, 4($sp)          # restore s0
    addi $sp, $sp, 16       # deallocate stack space
    jr $ra                  # return to caller
```

I then had to use this to generate a stream for the input.txt file. For this I converted the C file into a python file called imput_gen.py which does the same calculation as the provided c code but prints the output into another file called input.txt. Which holds the input stream for the tournament predictor.
In the given scenario, the code involves two conditional branch decisions: one compares the value of variable i to one thousand, and the other compares the value of variable j to four. Conventionally, when executing loop iterations, a branch is taken to earlier points in the code, while the branch is not taken to progress beyond the loop. Before the first loop iteration, the loop's condition check is executed, needing an unconditional jump from a point preceding the loop to perform this check. Consequently, reaching the beginning of the loop signifies that the loop's branch was taken. In other words, the instruction pointer returned to the loop's starting point rather than progressing forward. Conversely, executing instructions beyond the loop implies that the loop condition was satisfied without returning to the loop's beginning, showing that the branch was not taken.

After this I have to use the generated branch decisions as input to the tournament predictor. The Verilog documentation's Chapter 9 on writing testbenches supplies valuable insights and offers exemplary code for reading binary input from a file using readme and writing binary output to a file using writememb. In my implementation, I incorporated and changed the relevant sections of these sample codes to suit my specific scenario. Initially, I tried to use the recommended approach by reading all the inputs from the input file into a suitably sized vector during initialization. Then, using a four loop, I intended to iterate through the inputs and update them every two*T clock cycle, starting from the middle of a logic-high clock (when inputs are to be provided). Additionally, I planned to write the output to the output file on the falling or

negative edge of the clock (when outputs are ready to be read). However, despite my efforts to follow the provided chapter and adapt the code accordingly, I met challenges in implementing the intended functionality. Consequently, I resorted to hardcoding the first 750 generated inputs directly into the testbench. This deviation from the first plan allowed me to go ahead with the simulation and conduct further analysis.

I tried to generate the outcome of each part to try and analyze their performance and ended up with the following graph:



Based on the graph, the tournament predictor was correct for 75%-80% of the time. However, I require further work to figure out how Modelsim works and more time to perfect the predictor.

7. Conclusion/Further Work

In this project, a tournament predictor was implemented, forming two (2,1) local predictors, a (2,2) global predictor, and a selector. This predictor enables the prediction of future branch outcomes based on a given set of branches and their actual outcomes. Accompanied by Python scripts, the tournament predictor successfully made predictions on supplied C code, generating a results table that included the number of hits and an accuracy rating.

However, it is important to note that the tournament predictor implemented in this project has limited practical applicability. It relies on having the current branch outcome as input before supplying a prediction. Consequently, it functions more as hardware code that emulates the behavior of a predictor and calculates its accuracy, rather than a standalone predictor.

To enhance the usefulness of the tournament predictor, future work could focus on incorporating two modes: a predict mode that supplies predictions and an update mode that allows the predictor to be updated based on the actual outcomes. This would enable a more dynamic and adaptive prediction process.

Furthermore, the current implementation of the predictor is limited to code involving two branches. To make the predictor more versatile, future work could involve expanding the local predictors to make them modular and adaptable to any number of branches. This would enhance the predictor's flexibility and extend its usability to a broader range of applications.

In addition, it is worth mentioning that future work should also involve exploring methods to extract the generated prediction data from the predictor's output file. By doing so, it would be possible to develop Python functions that can calculate more precise accuracy percentages by dividing the number of hits and misses by the total number of sessions. This would supply a more comprehensive analysis of the predictor's performance and effectiveness.

In conclusion, while the current implementation of the tournament predictor serves as a starting point, further enhancements and modifications are necessary to maximize its practical utility. By incorporating added modes, adapting to various numbers of branches, and extracting prediction data for further analysis, the tournament predictor can evolve into a more powerful and versatile tool for branch outcome prediction in real-world scenarios.

8. References:

Patterson, D. A., & Hennessy, J. L. (2017). Computer Organization and Design MIPS Edition: The Hardware/Software Interface. Morgan Kaufmann.