Ray Chang
Tim Kim
ECE 154B

Project 3: Branch Prediction

Introduction

The goal of this project is to design and successfully implement two types of branch predictors which can save execution time for certain branch instructions in the MIPS pipelined processor. To achieve this, the local and global branch predictors were designed and tested on their own, then the two modules were integrated into the datapath of the processor. Any hazards or conflicts caused by the integration were addressed through significant debugging and testing until it was determined that our circuit met the design requirements of the project. This implementation is coded in Verilog and simulated in ModelSim.

It is important to note that we were unable to implement a fully working cache system along with our branch predictors and our processor; thus, we have concluded that it was best that we implement this project without the cache system from the previous project. Although we could not implement this system, we thought that we could list some possible strategies for a successful implementation of a processor with all parts included. The main issue with a cache system is when the instruction cache experiences a miss on a read, the processor would have to wait for a significant amount of time to fetch needed instructions. While waiting for the instruction to be loaded from the instruction memory to the cache, the branch predictor should be stalled so that it does not make any predictions until the correct instruction is outputted to the pipeline.

Block-Level Top Schematic

A modified diagram of the pipeline with the local or global branch predictor implemented is shown below.

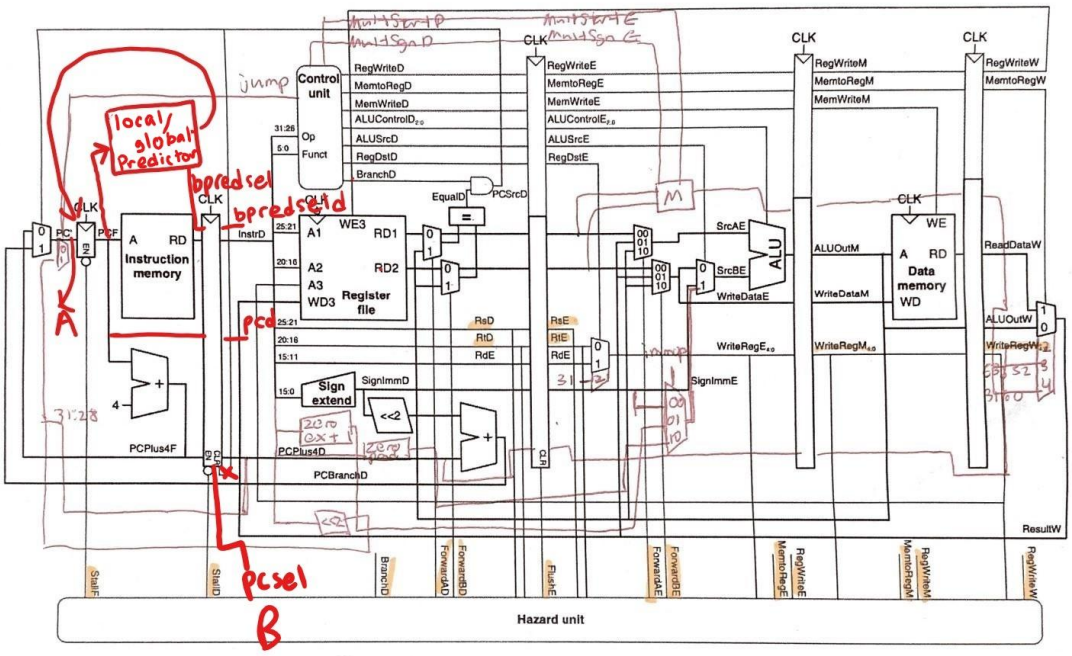


Figure 7.58 Pipelined processor with full hazard handling

Figure 1. Annotated Schematic of In-Order MIPS Pipeline

In the diagram below, a closer look at section A reveals how a taken prediction is implemented with the left-most mux, and how a prediction taken but outcome not taken branch is implemented with the middle mux. Section B shows the hardware required to prevent the decode register from being cleared if a branch's prediction and outcome are both taken. The right-most mux is used to proceed with the pipeline executions if a branch's prediction and outcome are both taken. The inputs and outputs of the local and global predictor modules are shown on the top right, and a diagram of the state machine used for the predictors is shown on the bottom right.

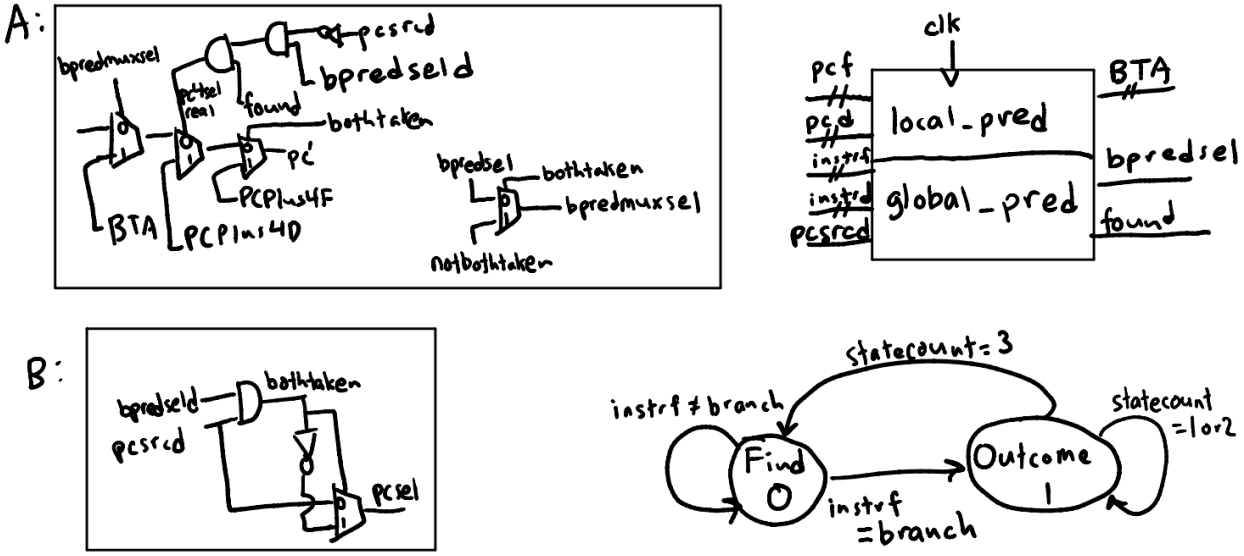


Figure 2. Diagrams of Pipeline Modifications, State Machine, and Block Diagram

Design Methodology

To design the local predictor, the first item that was considered was where the branch target buffer would be placed in the pipeline, and what inputs and outputs would be needed. After brainstorming and consulting with George, it was determined that the buffer should be placed in the fetch stage so that a branch prediction can be made within one clock cycle, so that if the prediction is correct, less execution time would be spent compared to the original pipeline where branches were resolved in the decode stage. The next step involved determining the structure of the local predictor so that it can store and output the correct data during the correct clock cycles. Our implementation involves a state machine with 2 states, the Find and Outcome states, and transitions occur on every rising and falling edge of the clock. While designing the state machine, inputs and outputs to the module were modified to improve the local predictor. After finishing a first draft of the module, modifications were made to the datapath in order to use the data outputted by the local predictor. Multiple programs were used to test the local predictor and reveal bugs which were resolved through significant debugging.

Designing the global branch predictor involved similar steps as the local branch predictor, but some differences include considering the usage of the branch history register and tag-checking just 2 bits of the PC. After every branch instruction, the branch history register was shifted to the left and updated according to whether the branch was ultimately taken or not. Then, the 2 least significant bits of the PC address were concatenated with the history bits in order to create an index for the branch history table. It is worth noting that the local and global branch predictors were designed for compatibility, meaning that no additional adjustments were needed to exchange a local for a global predictor, or vice versa.

Both local and global branch predictors utilized a four-state finite state machine (FSM) which established if a branch instruction was predicted to take the branch or not to take the

branch. There were four states to the FSM: strongly not taken, weakly not taken, weakly taken, and strongly taken. Such a system allowed for more concrete and accurate predictions to be made, as in order for a branch prediction to change (taken to not taken or vice versa), it would take two consecutive taken or not taken predictions to achieve this. Below is a diagram of the FSM.

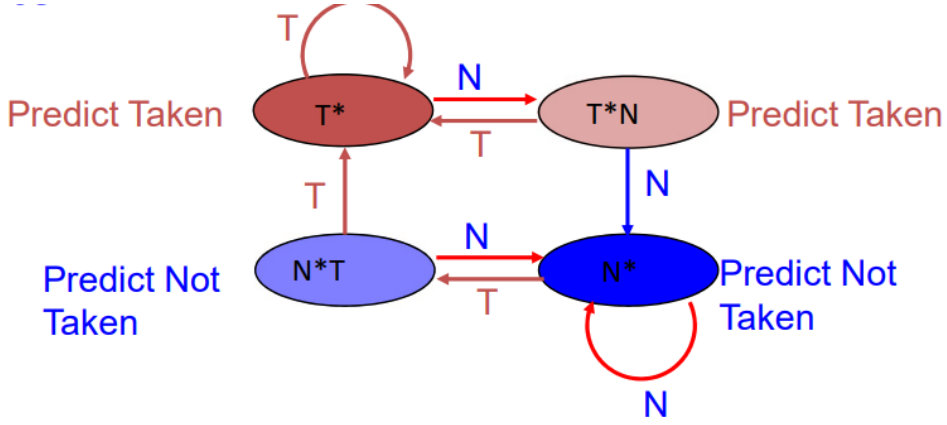


Figure 3. Finite State Machine of Branch Prediction

Modules Designed

1. Local Branch Predictor

   The local branch predictor module consists of a branch target table which is a 2^7 entry buffer in memory responsible for keeping track of the prediction state and the pc branch target address. Due to the limitation in space capacity, a module capable of dealing with conflicts is designed and implemented. The module has a tag-checking feature that accounts for an incorrect pc address indexing to a wrong entry in the table. In this case, the FSM corresponding to the targeted index is not used to make the prediction, instead the entry is replaced by the new tag and its FSM is initialized to strongly not taken. Otherwise, if tag-checking is successful, the entry's FSM is used to make a branch prediction. If the tag corresponds to a branch instruction but it is not found in the buffer, it is added to the buffer and its FSM is initialized.

2. Global Branch Predictor

   The global predictor module consists of a branch history table with 2^6 entries, as there are four history bits and two PC tag bits per index. As described before, a branch history register is responsible for keeping track of the outcomes of the last four branches. Such a system allows for correlation between the last few branches that have happened, which greatly improves the prediction accuracy of the predictor for programs with lots of correlating branches.

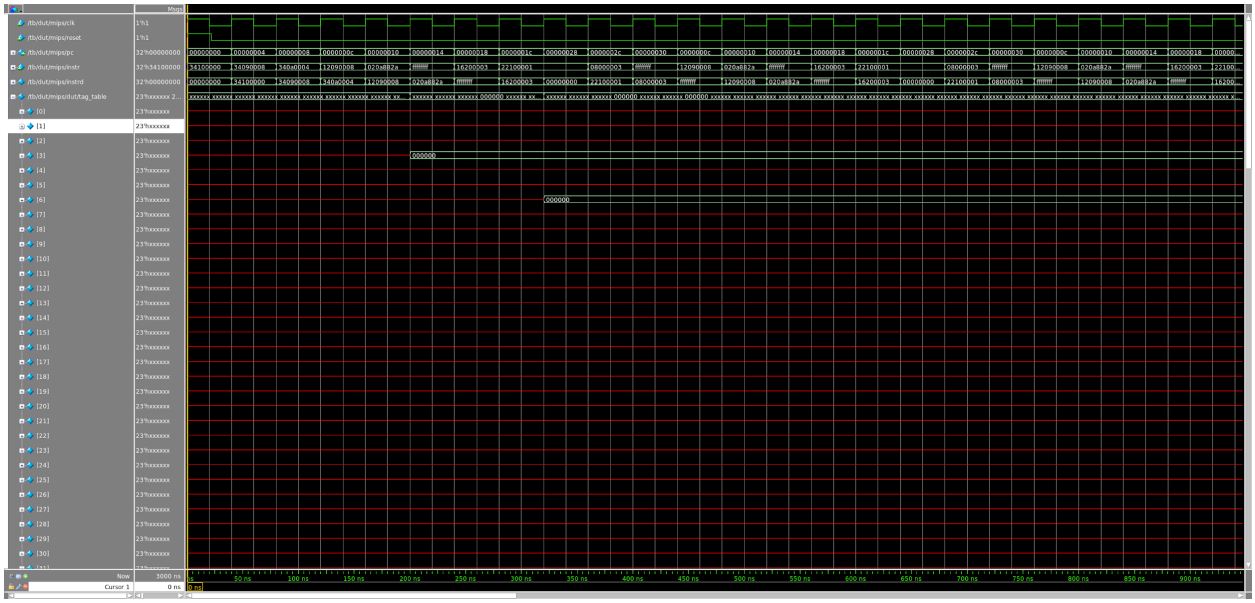
3. Datapath
   This module was improved from the first project in order to fully accommodate the local and global branch predictors. A series of multiplexers and logic gates were added so that the correct PC is inputted to the PC register depending on the branch prediction and outcome, and if the instruction is not a branch.

4. Hazard
   This module is slightly modified to prevent the decode register from being stalled if a branch's prediction and outcome are both taken so that the prediction can continue through the pipeline.

Test Programs

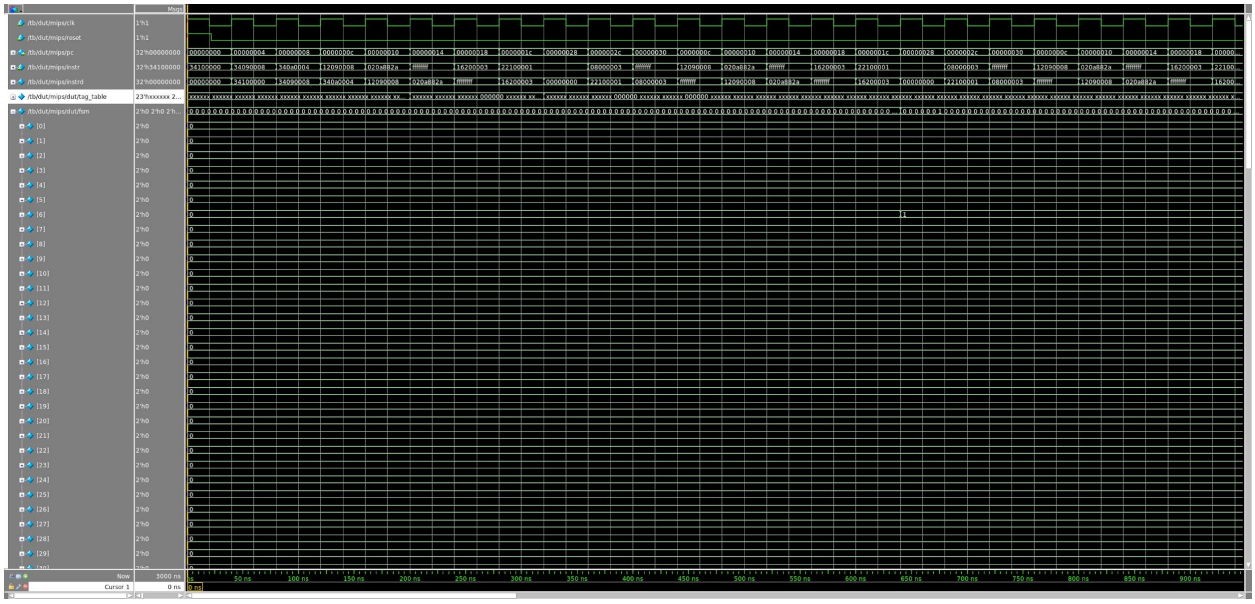
Interesting programs were designed in order to test the full functionality of both local and global branch predictors. In memfile8.dat, the assembly code would loop for 8 times, and on each loop iteration, the program would either branch or not branch on a specific program counter. For the first four iterations, the program does branch, and for the last four iterations, the program does not take the branch. In memfile9.dat, two branch instructions have the same tag, so this program is used to test the tag conflict resolution implemented in the predictors. Through these consecutive branches and non-branches, we were able to observe the different behaviors of the local and global predictors. Some key factors we looked out for were the correct changes in the finite state machine, and also if branches were accurate and occurred in a timely manner.

1. Local Branch Predictor



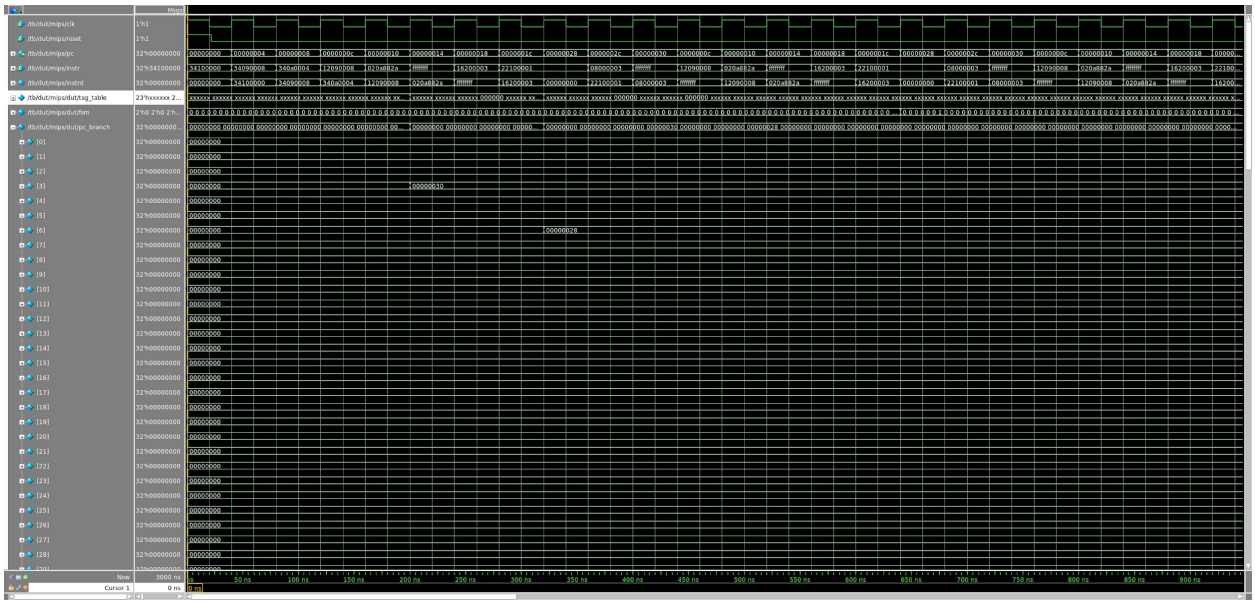
Waveform 1 of memfile8.dat

The tags of the branch instructions are saved in the tag_table register. Although they have the same tags, they are placed in different entries because their index bits are different.



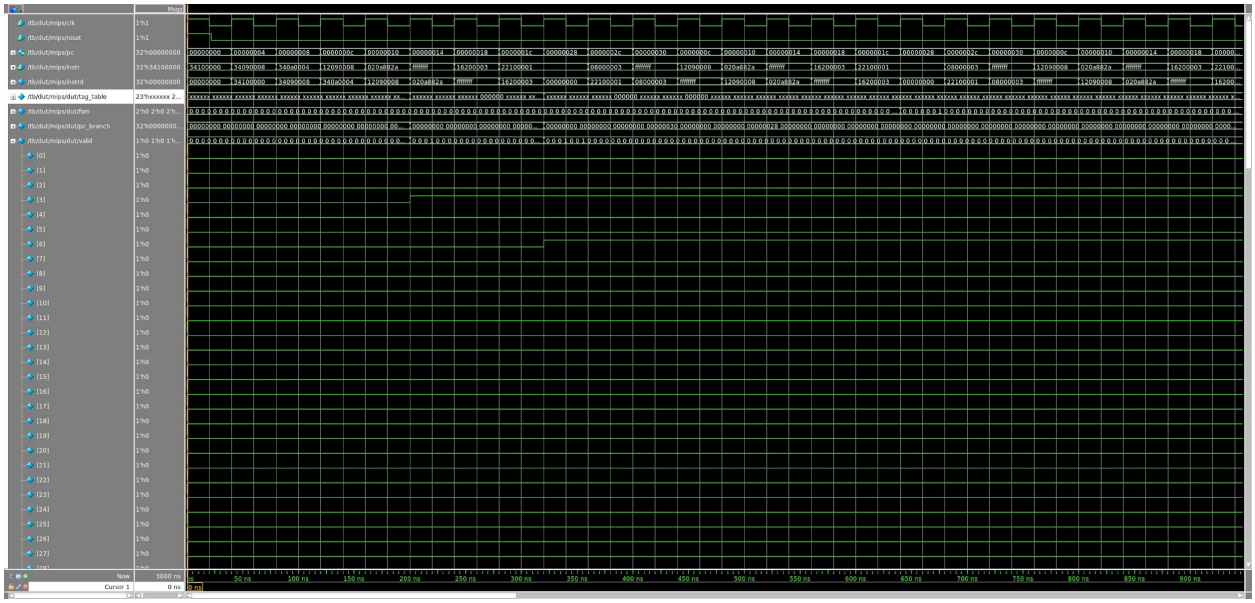
Waveform 2 of memfile8.dat

The FSM of the bne instruction is updated from strongly not taken to weakly not taken because the outcome of the branch was taken. Since the prediction was incorrect, another clock cycle is needed to clear the decode register and move to the correct branch target address.



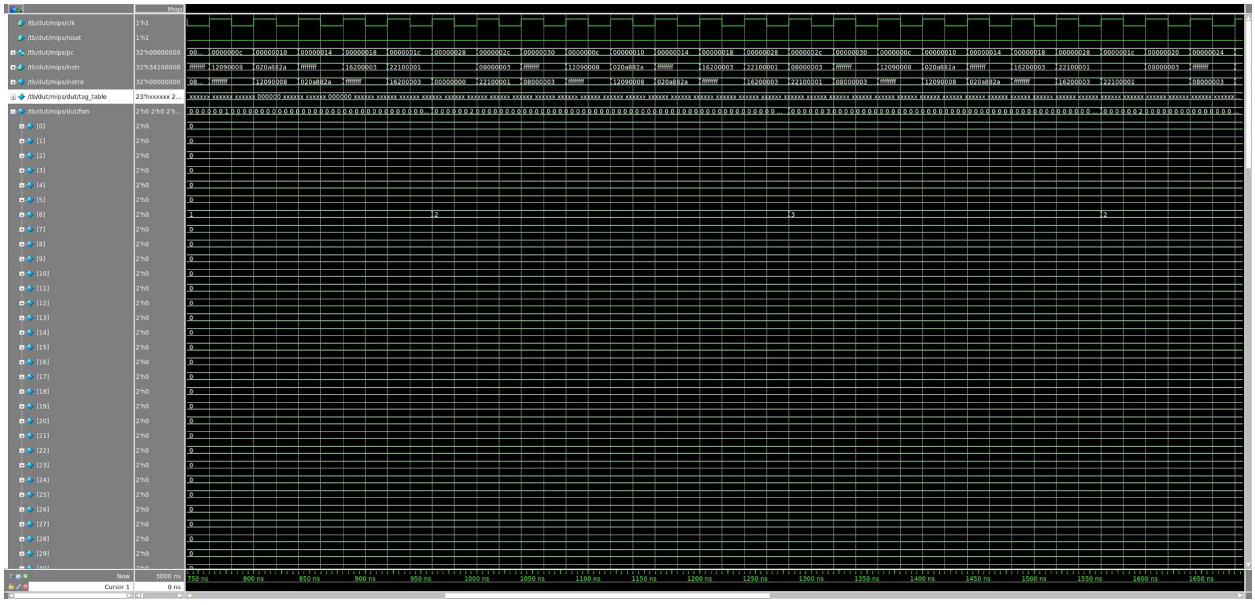
Waveform 3 of memfile8.dat

The branch target addresses are placed into the corresponding entries of the pc_branch register. These addresses are used when the branch prediction is taken.



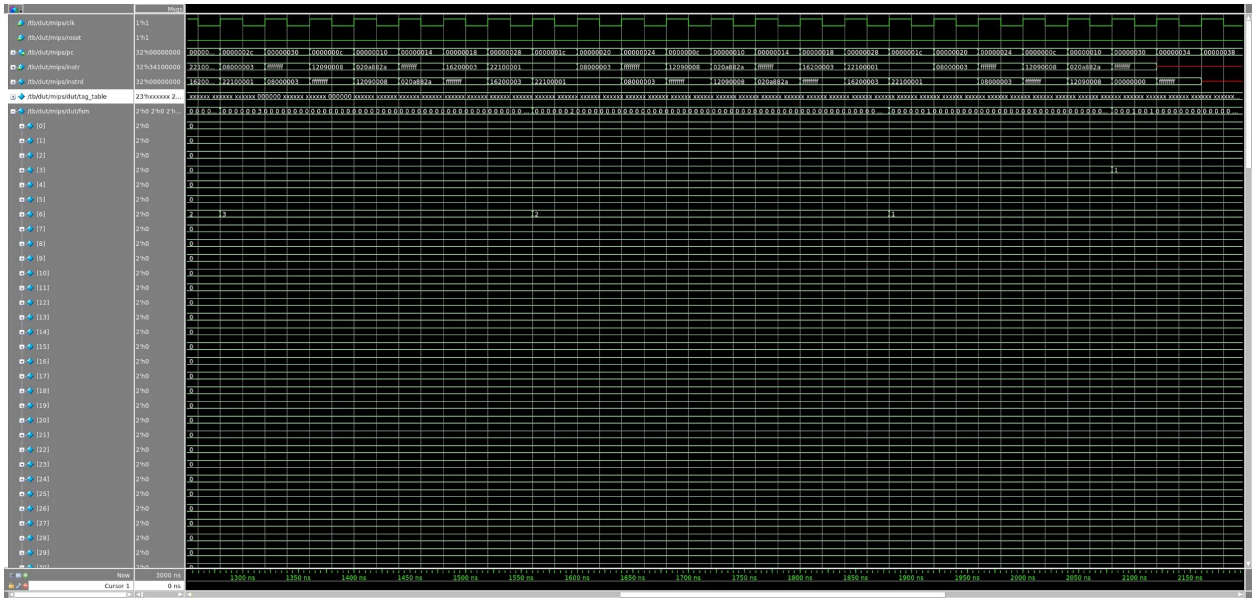
Waveform 4 of memfile8.dat

The valid bits of the corresponding entries are raised high when the branch instructions are placed into the buffer.



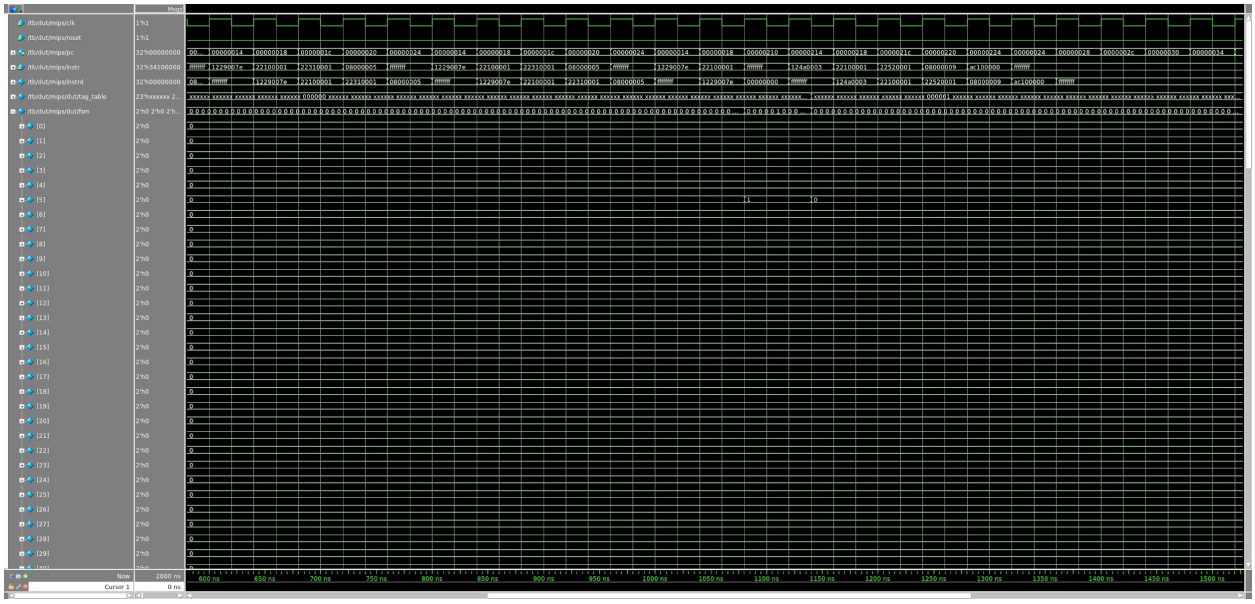
Waveform 5 of memfile8.dat

Around time=1250ns, the bne instruction is predicted to be taken and the outcome is also taken, so the instruction proceeds through the pipeline without any stalling or clearing. Since the prediction is correct, the branch only takes one cycle instead of the original 2 cycles when branches were resolved in the decode stage.



Waveform 6 of memfile8.dat

The FSM corresponding to the bne instruction is transitioned to weakly taken, then weakly not taken since the outcomes were not taken. The FSM corresponding to the beq instruction is updated from strongly not taken to weakly not taken since its outcome was taken.

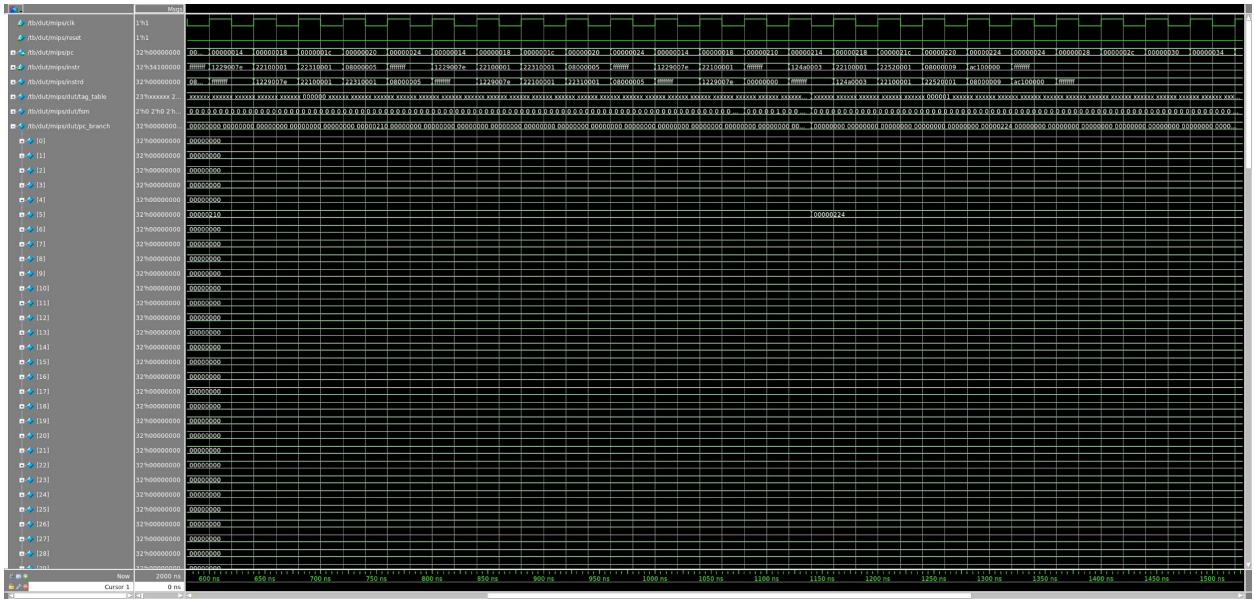


Waveform 1 of memfile9.dat

Since both branch instructions have the same index bits, they correspond to the same entry in the buffer, so when the beq instruction is executed, instead of using the existing FSM which corresponds to the bne instruction, it initializes the FSM to strongly not taken.

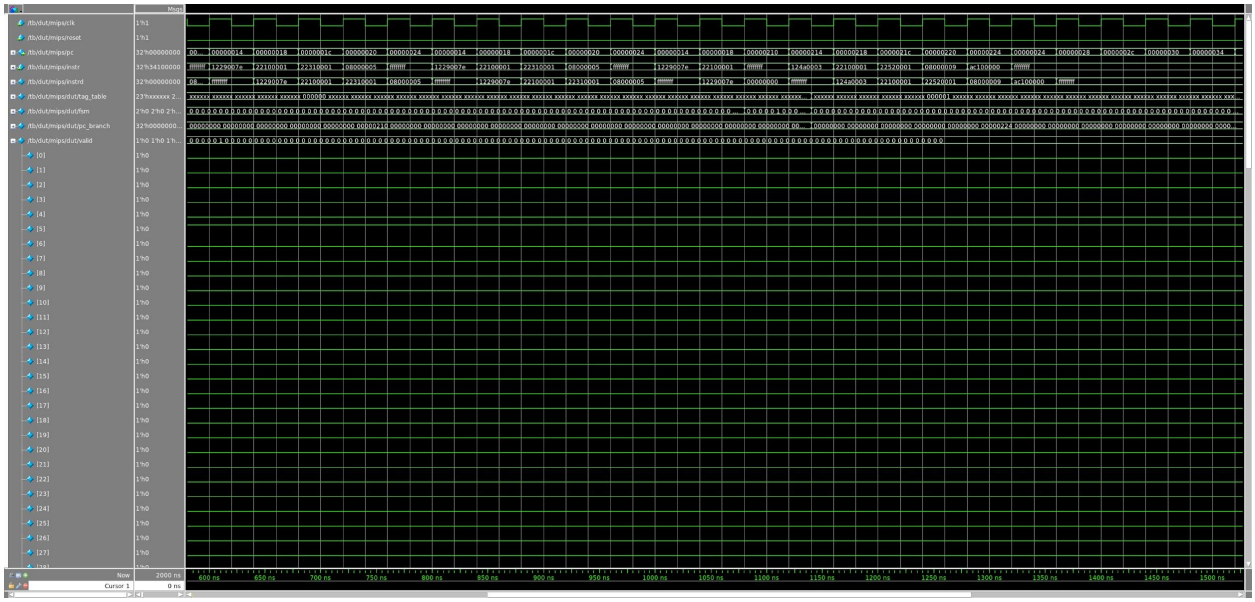
Waveform 2 of memfile9.dat

The tag bits of the two branch instructions are different, so the tag of the beq instruction replaces the existing tag of the bne instruction.



Waveform 3 of memfile9.dat

The branch target address of the beq instruction replaces the existing branch target address of the bne instruction.

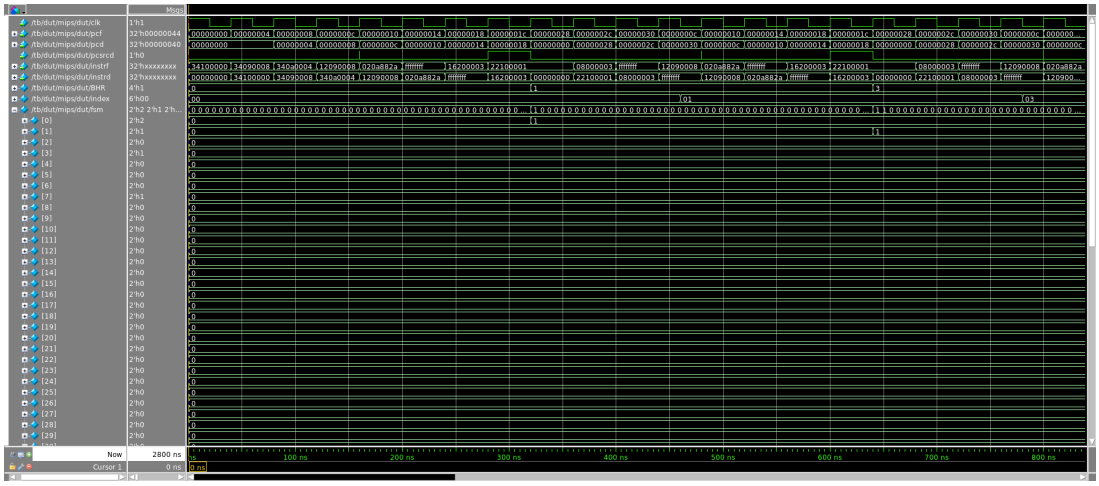


Waveform 4 of memfile9.dat

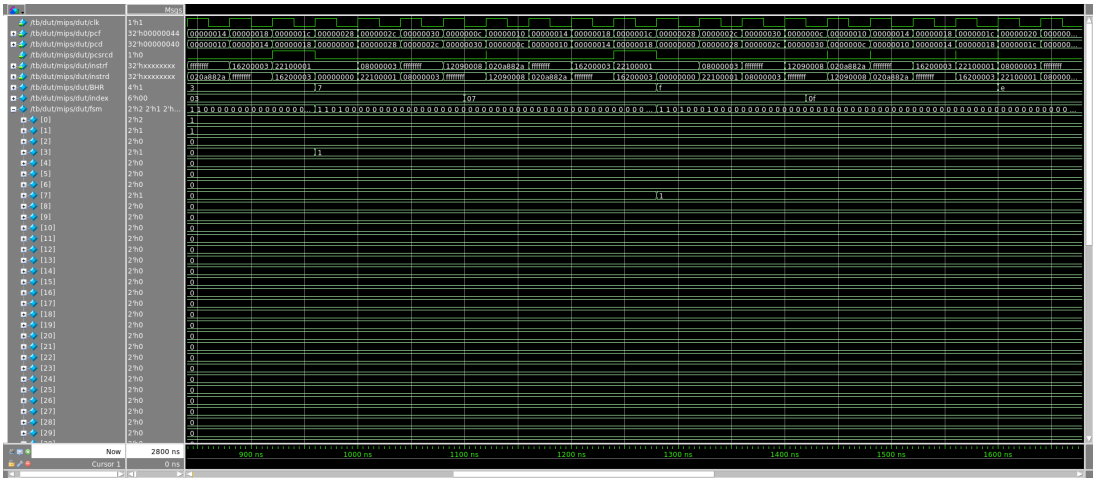
The valid bit stays high because the entry holds a branch target address for a valid branch instruction.

2. Global Branch Predictor

Below are the waveform data that were gathered from testing the global branch predictor.

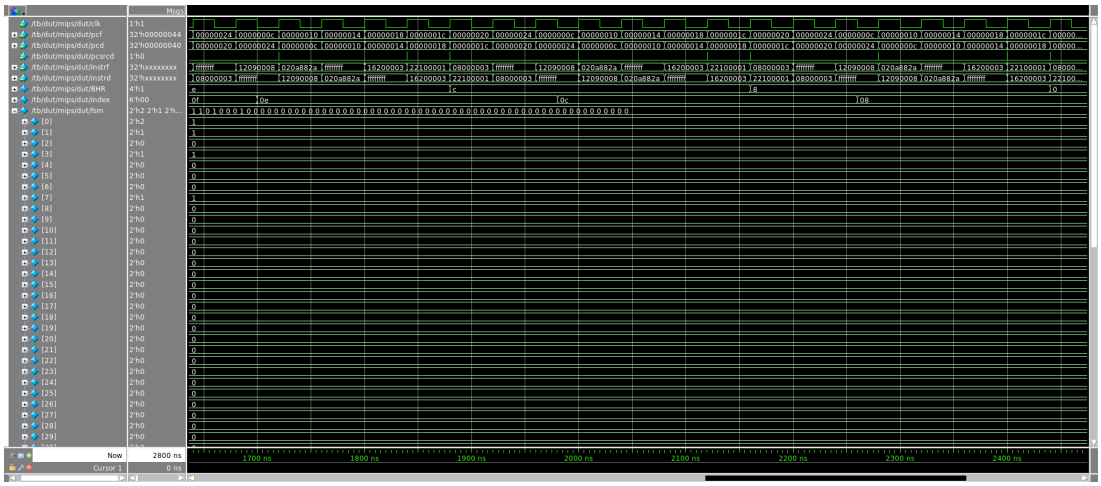


Waveform 1 of Global Branch Predictor

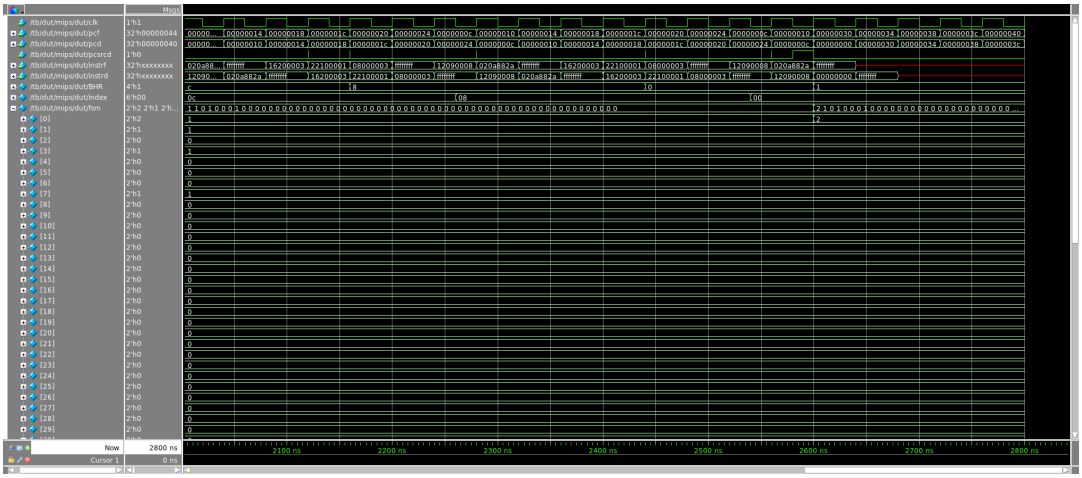


Waveform 2 of Global Branch Predictor

In the above waveforms, we can observe that our branch history register bits go from 0000 to 0001 to 0011 to 0111 to 1111 in the first four iterations of the loop, which shows correct shifting behavior of the register. In addition, the indices are correctly created using the two least significant bits of the PC address and the register bits, and are used to update the finite state machine of the different entries of the table.



Waveform 3 of Global Branch Predictor



Waveform 4 of Global Branch Predictor

In these two diagrams, we can observe that our branch history register bits go from 1111 to 1110 to 1100 to 1000 to 0000 in the last four iterations of the loop, which shows correct shifting behavior of the register.

Conclusion

In this project we designed local and global branch predictors and implemented them into separate pipelines and observed their effect on resolving branch instructions. The local branch predictor uses a 4-state FSM for each tag to make a prediction for a branch instruction, and the FSM is updated according to the outcome of the branch. The global branch predictor also uses the 4-state FSM but assigns tags to an entry in the prediction table based on the Branch History Register, which stores the outcomes of the previous 4 branches in the program. The datapath and hazard modules were modified in order to fully implement the predictors into the pipeline and take full advantage of correct predictions and resolve errors caused by incorrect predictions.