Design and Implementation of a Single Level Neural Branch Predictor for SDLX Processors

Rachit Verma (21110171), Mithil Pechimuthu (21110129), Sachin Jalan (21110183)

Abstract: **This paper looks at the basics of the branch instruction and the various control hazards that these instructions produce. It also covers the basics and motivation behind branch prediction and the various branch prediction schemes that are available in the current day. Most importantly, it contains a potential perceptron-based neural branch predictor design for two and three-stage SDLX pipelined processors. We have also reasoned our choice and provided the results of the plan.**

Introduction to Branch Instructions

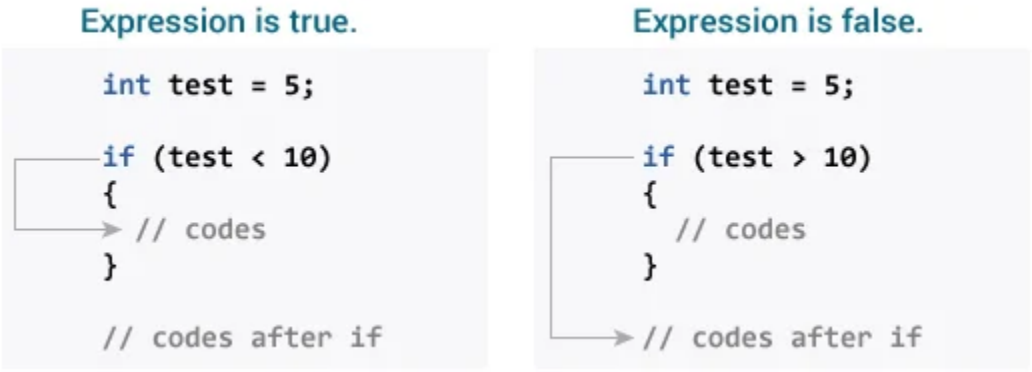

Figure 1: A simple if-else code snippet [1]

The if statement in the code snippet in Figure 1 shows us what branch instructions do. Branch instructions are used to conditionally divert the flow of instructions into another set of instructions. For example, here the condition to check is test > 10 or test < 10. Branch instructions are ubiquitous in programs. Around 15 - 30 per cent of all the instructions are branching instructions, especially for a uniprocessor like the two and three-stage SDLX processor [2, 3, 4, 5]. Generally a conditional branch instruction occurs every 5 - 8 instructions. [7, 8]. However these instructions disturb the processor's sequential execution of instructions. Hence, it is extremely important to manage these hazards that are produced by branch instructions. Branch prediction is a very important method to solve these control hazards.

Introduction to Branch Prediction:

A branch instruction is simply required to direct the flow of execution to a particular set of instructions given a specific condition. They are as expensive for the processor as quintessential they are for programming. Hence we need to be able to reduce the delays that they cause. Branch prediction is basically trying to predict if the branch will be taken or not, and if taken which instruction will occur next. Branch prediction is seen as a talisman to reduce the delays caused by branch instructions. Hence, there is a growing amount of research in the field of branch prediction. Since the 1990s many new methods and algorithms of branch prediction have been proposed and also implemented. This allowed pipelines to get deeper and more efficient. Broadly we categorize the prediction strategies as Static and Dynamic.

Obviously, if we do not use branch prediction, we will have to flush Nops into the pipeline and cause the pipeline to run useless instructions for 2 cycles in a three stage pipeline.

OLD	**NEW**
Less Accuracy. 65% - 70%	Very high Accuracy: 90% - 98%
Less hardware intensive	Require immense amount of hardware
Simple to implement	Relatively difficult to implement
Support pipelines of few stages	Built for high accuracy in deep pipelines

Table 1: Comparison between old and new methods of branch prediction.

On a side note, it will be important to note that, while branch prediction only predicts if a branch is taken or not, to predict the target of a branch will require separate branch target prediction strategies. Hence, branch prediction and branch target prediction are quite different and perform different tasks. However, we have implemented a simple approach for branch target prediction to improve the efficiency of the three-stage pipelined SDLX processor.

Static and Dynamic Branch Prediction

1) Static Branch Prediction: Static branch predictors, choose either taken or not taken and stay with this choice always. For example, we could always choose to take a branch whenever we encounter one. This strategy will provide a mis-prediction rate of 30 - 35 % [8]. However, today's problems demand much higher efficiency and lower mis-prediction rates. However, dynamic branch prediction strategies are adaptive and thus provide better performance.

2) Dynamic Branch Prediction: This is the most commonly used type of branch prediction. These have much higher accuracy than static branch prediction strategies. Hence static methods have become obsolete. Here we exploit the patterns and repetitive nature of the occurrence of branch instructions in a program.

Various Strategies of Branch Prediction:

Predictor	**Miss Rate**
Taken	62.552%
Not Taken	37.448%
2 bits	44.838%
3 bits	42.680%
Bimodal 2 bits	7.269%
Bimodal 3 bits	6.937%
Gshare-32K	4.594%

Gshare-64K	4.351%

Table 2: Accuracy comparison of different Predictors. Source: [11]

To know more about these branch predictors we encourage the reader to refer to [11].

Three Stage Pipelined SDLX Processor Basics :

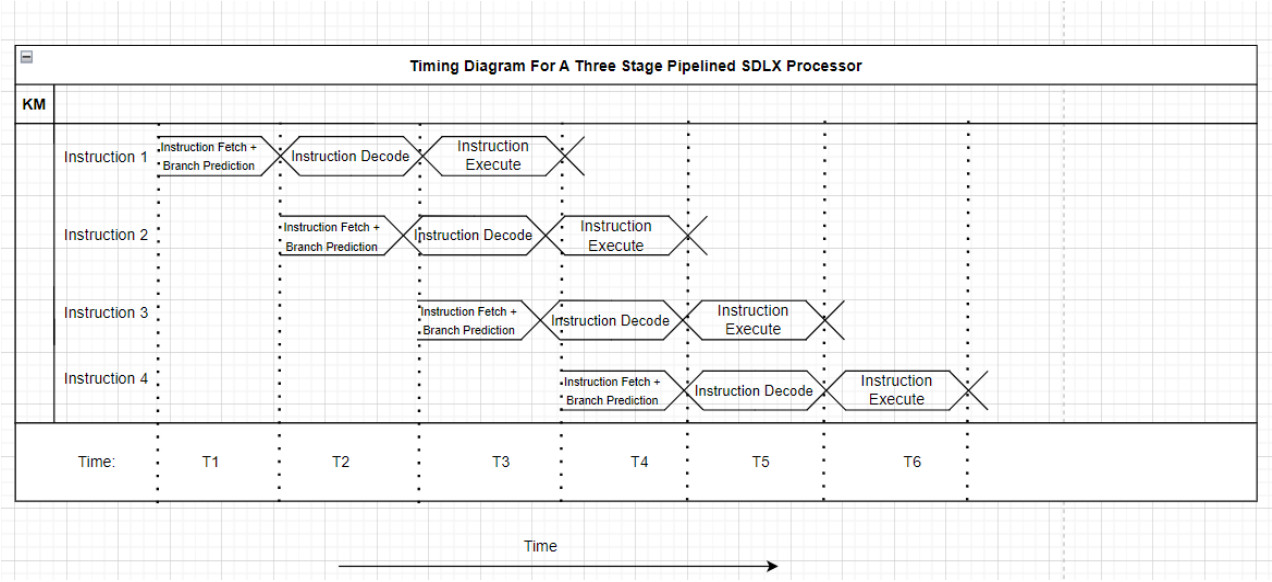

Figure 2: Timing Diagram

We have divided the SDLX processor data path into three stages: Instruction Fetch, Instruction Decode, and Instruction Execute. In the instruction fetch, we use the instruction address to predict if it is a branch instruction; if it is one, we predict the branch will be taken and its target. In the instruction decode stage, we produce all the control signals the processor requires and all the inputs the ALU would need. In the instruction execution stage, we deliver the final result to be put into the register file and compute the last target address of branch instructions.
Timing and control points:
How do we predict?
Initially, we check for the presence of the current PC address in the tag of the BTB. The BTB is directly mapped. The BTB is just like a Cache but differs from it in the data it stores. The BTB holds the corresponding target address for each PC. In case of a hit, we check if the "taken" signal from the perceptron predictor is high. If it is high, we pass the output of the BTB as the next PC value. However, if the taken signal is low, the PC is incremented by 1 in the next cycle. Now the prediction can be correct or wrong. Accordingly, two cases follow:-

Case 1:
If we happen to mispredict:

1. We will set the write enable signal of the register file from the ID_EX latch to 0. We do this to ensure that the next instruction after the mispredicted branch doesn't get executed and that no changes occur to the reg file's contents.
2. We will also reset IR to 32'b0. Hence a Nop instruction is put into the pipeline. Accordingly, this removes the next-to-next instruction following the misprediction from the pipeline, and a Nop is executed instead.
3. All of the above steps flush the pipeline of the mispredicted instructions.

4. We will give the BTB to make a tag out of the branch instruction's PC that is stored in another buffer meant to remember the PC address of Branch instructions, and the ALU output that represents the actual target address is stored at this tag. At the same time, the BTB is also being read to give a result if it is a hit or a miss.
5. We inform the predictor circuit that there has been a misprediction and that it has to correct its weights accordingly.

Case 2:
If we predict right:

1. We will let the instructions flow as they have to. Moreover, we will update the weights of the perceptron to indicate an optimistic prediction.
2. Here the execution of the process is trivial, like the two-stage-pipelined SDLX processor.

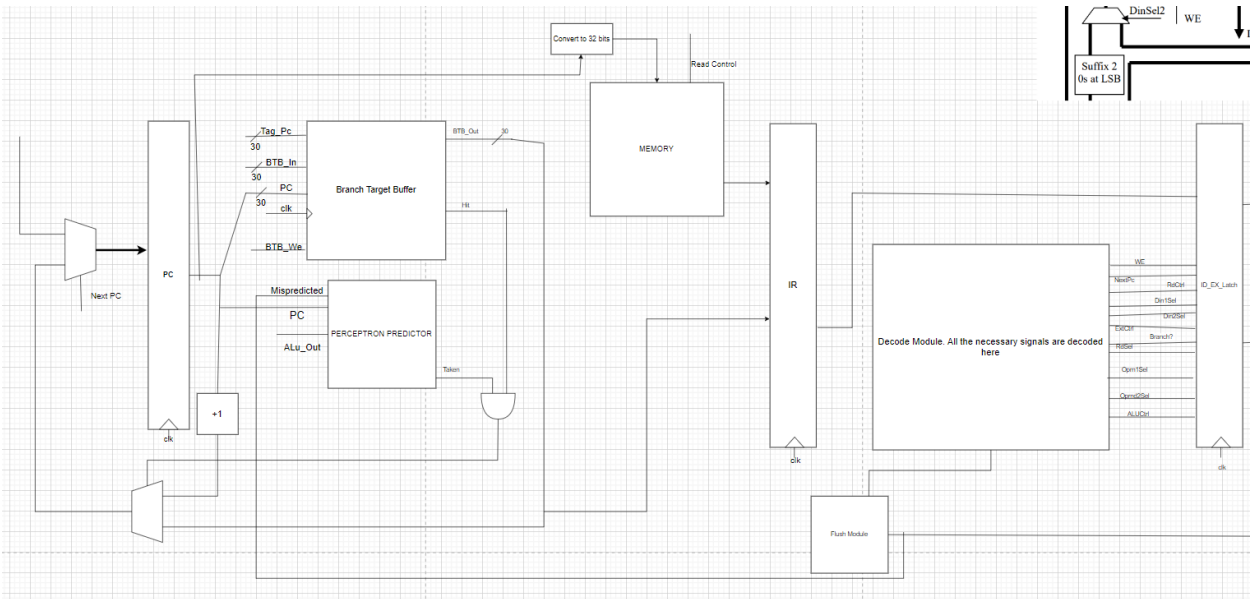

Figure 3: The datapath for the branch prediction module.

Perceptron Based Neural Branch Predictor

1. Why did we choose this for the SDLX processor?

    a) It is more accurate than other predictors.
    b) It involves the modern and evolving technology of machine learning. Hence, the many researchers interested in a similar field will scrutinize and improve the design.
    c) The space required to implement it in hardware grows linearly. This space requirement is much better than the exponential growth in a standard two-level branch predictor.
    d) It is also much easier to understand and implement for the SDLX processor when compared to the latest branch prediction methods, such as TAGE predictors.
    e) We could easily modify it to add personalized features for the SDLX processor.
    f) It results in reasonably good accuracy, given the relative simplicity of the model.

Some more facts about the Perceptron based branch prediction methods are given below.

The Perceptron predictor works better than most of the other branch prediction algorithms because the RISC architecture requires a more dynamic branch to perform the task and hence longer history is required to predict. Hence as the perceptron predictor is linear with space as the length of the history grows thus it is better than most of the predictors.[1]

Global perceptron has a misprediction rate of 4.6% which is an improvement of 26% over gshare (6.25%) and an improvement of 15% over 6KB budget bimode(5.4%). The perceptron has higher accuracy when a hybrid of global and local history is used which has an misprediction rate as 5.2%. [1]

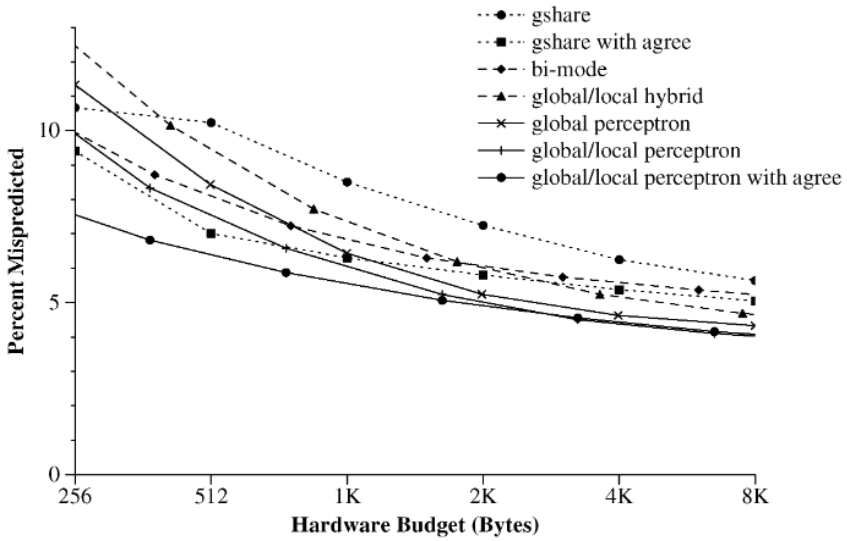

Figure 4: Graph representing accuracy of various predictors with specific sizes. Source: [1]

Evers multicomponent hybrid predictor was a hybrid predictor, which used a set of two gshare predictors one with shorter branch history and other with longer branch history. But the perceptron model performs much better than the evans multicomponent hybrid predictor when given the same hardware budget.[1]

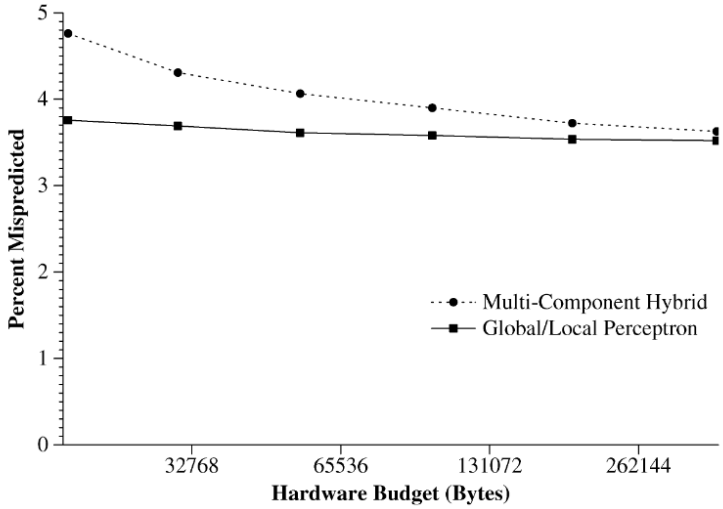

Figure 5: Graph representing accuracy of various predictors with specific sizes. Source: [1]

We hypothesize that the main advantage of the perceptron predictor is its ability to make use of longer history lengths. Schemes such as gshare that use the history register as an index into a table require space exponential in the history [1].

2. Implementation of the predictor.

Branch prediction is of particular importance in pipelines with many stages(such as those in modern processors), where the cost of a misprediction becomes high and, if not controlled, can act as major bottlenecks to a processor's performance. There are various techniques for branch prediction, and they can rely on various properties of the branches to make predictions. These properties include global and local similarities in the conditions of the branches, previous states of the branch(taken or not taken), and structures put in place to manipulate this data. Previous techniques like two-level branch prediction use 2 bits and a finite state machine to make predictions given the global and local branch history.

The perceptron branch predictor attempts to apply contemporary research in neural networks to make branch predictions. A perceptron attempts to model a human neuron. It takes in several inputs, processes them, and returns a single output. The function used to process the inputs is a linear function, with weights assigned to each input. The final output is the summation of the product of each weight-input pair with a bias term added. In our case, this output is compared with 0 to give a binary output[1].

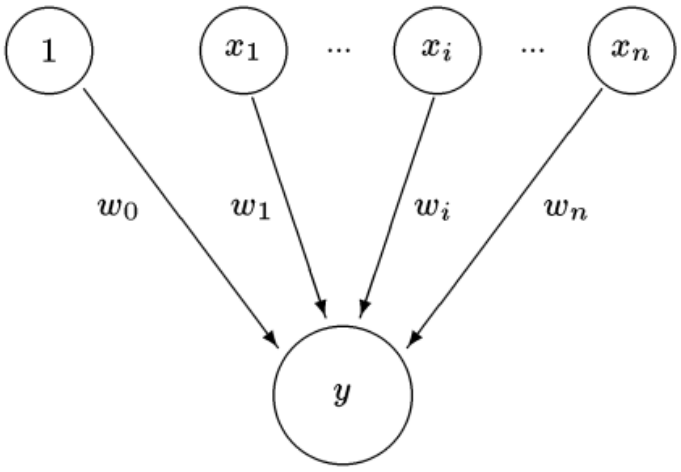

Figure 6: Perceptron. Source: [1]

$y = w_0 + \Sigma w_i x_i$

where y is the output, w_0 is the bias term, and w_i is the weight term corresponding to each input.

The perceptron is mainly meant to exploit linear relationships between the inputs and the outputs. While a lot of phenomena in nature are described by linear relationships among the parameters, a single perceptron does have its limitations in terms of the phenomena it can model. However, it has been empirically demonstrated that this model works well for our purpose [1].

We maintain a global history of branch executions of length k. We insert 1 in each slot when the branch was taken and insert a -1 when the branch was not taken. This global history is fed into the perceptron as the input vector, corresponding to which output is produced. This output is fed into another basic filter, which produces the final binary output. This output is used to fetch the appropriate instruction address for the next instruction. After the decode step of the branch instruction is completed, it is checked if the predicted output is the same as the real output. If they are equal, then the pipeline works without any interruption, else the entire pipeline is flushed with the proper instruction.

Reading the instruction

The address of the branch instruction is fed into the branch prediction in parallel to the IF stage in the SDLX processor. For a 0 cycle loss, the branch prediction needs to be available as soon as the IF stage in the pipeline ends.

We store N different perceptrons inside the branch predictor. In the first step, the address of the branch instruction is hashed to produce an index that is used to select one perceptron among the N perceptrons present[1].

Producing the output

Each perceptron has its distinct weights and biases. After we have selected the perceptron by hashing the address of the branching instruction, the weights and biases corresponding to that perceptron are available. The number of weights is equal to the length of the global history that we have chosen, with 1 weight corresponding to each result in the history. Additionally, we also introduce a bias term.

$$y = w_0 + \Sigma w_i x_i$$

y gives us the output from the corresponding perceptron for the present global history. w_0 is the bias term, w_i is the weight corresponding to the result of the branch encountered in branches ago. xi is the result of the corresponding branch (1 or -1). x_0 is always set to 1. Do note that this is the actual result of the branch, which was given by the ID step for that branch.

If y is greater than 0, then the branch is said to be taken, and if y is less than 0, the branch is said to be not taken, and the final output produced is 1 and -1, respectively. If the final output is 1, then the branch target address also needs to be supplied to the pipeline. This will be discussed further in the upcoming sections.

Updating The Weights

The weights are updated 1 cycle after the prediction is produced, and the actual result is obtained from the instruction decode step[1].

$for\ i = 0\ to\ n$:

$$w_i = w_i + x_i t \quad (7)$$

t is the actual result obtained from the ID step. This is the primary step in the branch predictor, the step where the perceptron actually 'learns' and improves itself to eventually make much more accurate predictions on an average.

Branch Target Buffer

The problem of branch prediction is naturally accompanied by the problem of branch target prediction. Consider the following scenario: We have been able to predict the direction of the branch through the branch address and need to provide the target address to the IF stage of the pipeline. The prediction and the target address need to be provided as soon as the ID stage of the branch instruction starts. However, doing it in a brute-force manner would require us to calculate the target address from the branch instruction. Doing this with the timing constraint imposed is very difficult.

To solve this problem, we maintain a branch target buffer, abbreviated as BTB[12]. BTBs can hold a variety of data depending on the needs of the user. They are very similar to caches in both form and function. Each cell consists of a tag, which is given by the address of the branch instruction, and the output corresponding to that tag, which gives the branch target[12]. This branch target is eventually supplied to the IF stage of the pipeline.

Owing to space constraints, the BTB cannot contain each and every branch instruction in the program being executed. Hence there is a need to discard and insert tags(or branch instructions) into the BTB methodically.

Firstly, we note that branches that mostly evaluate to not be taken have no penalty on the performance of the processor since it would keep functioning normally like a normal non-branching instruction[12]. Hence, in the design of the BTB, we do not insert those branches that evaluate to not be taken.

As a result, we start filling the BTB with the tags and their corresponding branch targets. Do note that if a branch evaluates to taken and if it is not available in the BTB, a 2-cycle delay is incurred to calculate the branch target. It is then inserted into the BTB so that it can be accessed for future use.

When the BTB is full, and we want to insert a new branch into the BTB, we discard the branch with the least potential for performance improvement. While there is no fixed way to determine the same, we employ the LRU algorithm as a heuristic.

According to S. Kumar et al., "The Least Recently Used replacement policy uses the access pattern of program memory to predict that the most recently accessed cache line is most likely re-accessed, and the cache line which has been Least Recently Used (a block in the set that is in cache for the longest and having no reference to it) will be replaced by the cache controller [10]."

After the LRU algorithm determines which branch is to be removed, we replace it with the present branch and its target address.

BTB Details:
We will have one extra signal called branch that is high if the instruction is a branch instruction and zero otherwise.
Inputs: clk, Tag_Pc, BTB_in, BTB_We, PC.
Outputs: BTB_out, hit.

So the BTB_We is the branch signal.
The BTB stores the BTB_in values in the Tag_Pc address when this is high. Else it doesn't write into the BTB. In every clock cycle, the BTB is checked for a hit with the PC address during the Instruction fetch cycle.

BTB takes input from the PC at every clock cycle and checks for a hit. If it is a hit, the hit signal becomes high; else, it is low. If it is a hit, also the Prediction unit tells us that the branch is taken, and then the following PC is given the BTB_out address to fetch the next instruction. Otherwise, if it is a miss, the NextPC is provided with the Pc+1 value. The BTB acts like a register file that can be read into and written into at the same clock edge.

Limitations of the Perceptron

Perceptrons cannot learn linear inseparable functions. Linear Inseparable functions can be defined as the true function which can predict the branch as taken or not taken by the history of k branches, it is this function that all the predictors try to learn, if this function is linearly inseparable then the perceptron would be unable to learn. Yet our neural predictor performs much better than the gshare algorithm on linearly inseparable functions.

References:

[1] D. A. Jiménez and C. Lin, “Neural methods for dynamic branch prediction,” ACM Transactions on Computer Systems, vol. 20, no. 4, pp. 369–397, 2002.

[2] Programiz Editors, “C if...else statement,” *Programiz*. [Online]. Available: https://www.programiz.com/c-programming/c-if-else-statement. [Accessed: 03-Apr-2023].

[3] Gibson, J.C. The Gibson Mix. TR00.2043, IBM, June 1970.

[4] Huck, Jerome C. Comparative Analysis of Computer Architectures. 83-243, Stanford University Computer Systems Laboratory, May 1983.

[5] Shustek, Leonard J. Analysis and Performance of Computer Instruction Sets. Ph.D. Th., Stanford University, January 1978.

[6] R. G. Wedig and M. A. Rose, “The reduction of branch instruction execution overhead using structured control flow,” *ACM SIGARCH Computer Architecture News*, vol. 12, no. 3, pp. 119–125, 1984.

[7] S. Reches and S. Weiss, “Implementation and analysis of PATH history in Dynamic Branch Prediction Schemes,” *Proceedings of the 11th international conference on Supercomputing - ICS '97*, 1997.

[8] Colin Egan, Gordon Steven, Patrick Quick, Rubén Anguera, Fleur Steven, Lucian Vintan, Two-level branch prediction using neural networks, Journal of Systems Architecture, Volume 49, Issues 12–15, 2003, Pages 557-570, ISSN 1383-7621, https://doi.org/10.1016/S1383-7621(03)00095-X. (https://www.sciencedirect.com/science/article/pii/S138376210300095X)

[9] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach (third ed.), Morgan Kaufmann Publishers (2002)

[10] S. Kumar and P. K. Singh, "An overview of modern cache memory and performance analysis of replacement policies," *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, Coimbatore, India, 2016, pp. 210-214, doi: 10.1109/ICETECH.2016.7569243.

[11] G. Pan and M. Lu, *Analysis of Branch Predictors* . [Online]. Available: http://www.ece.ualberta.ca/~elliott/ece510/seminars/2006f/project/Analysis_Of_Branch_Predictors_Ming&Guang/Report/Analysis%20of%20Branch%20Predictors.pdf . [Accessed: 04-Apr-2023].

[12] C. H. Perleberg and A. J. Smith, ‘Branch Target Buffer Design and Optimization’, EECS Department, University of California, Berkeley, Dec. 1989.