SECIALIZATOIN PROJECT

Low Power Solution to Performance Optimization

Master Student: Giorgi Solomnishvili

Supervisor: Prof. Thomas Tybell

Co-Supervisor: Øystein Knauserud

Norwegian University of Science and Technology

15/05/2021

Contents

List of Abbreviations:

BTB	– Branch target buffer
BHT	– Branch history table
CPI	– Clock cycle per instruction
GHT	– Global history table
PC	– Program counter
ISA	– Instruction set architecture
IF	– Instruction Fetch stage
ID	– Instruction Decode stage
EX	– Instruction Execution stage
WB	– Instruction Writeback stage

Abstract

The project is carried out on CV32E40X RISC-V core, designed for embedded system applications by OpenHW Group. The project is carried out in collaboration with Silicon Labs and NTNU. The main goal of this project is to increase the core's performance in terms of clock cycle per instruction (CPI) by utilizing low-power solutions to the performance optimization problem. Several design techniques can increase microprocessors' performance without drastically increasing power consumption. The project focuses on one of them, specifically branch prediction. The 2-bit branch predictor is modeled in python. The model emulates the CV32E40X core and runs custom programs on it as if the predictor has already been implemented in the hardware. According to the results obtained after running custom programs provided by Silicon Labs, a 2-bit branch predictor can increase the core's performance by up to 4.3%.

Introduction

Embedded System is an electronic unit consisting of processing cores. The unit can read information from its surroundings and make decisions based on the received data. Embedded System itself is a part of a bigger system. Nowadays, embedded systems are everywhere. For example, there are hundreds of microprocessors in modern cars, controlling fuel injection systems, braking systems, etc. Moreover, one cannot design IoT networks without utilizing a significant number of embedded systems.

The design of embedded systems has an extended history that goes back to Apollo missions and was utilized in the spacecraft navigation systems[1]. Traditionally, embedded systems were application/task-specific, consisting of closed architectures, and were unique to each device. However, embedded systems design has recently become more software-oriented, and closed architectures give way to open ones[2]. One of those open architectures is RISC-V. The main idea of its development was to introduce an open-sourced, practical instruction set architecture (ISA) that could be used in academic projects and deployed in any hardware or software design[3].

Most of the research done at the beginning of the 21st century focused on improving performance at any cost. However, it has been proved that performance increase is limited by the amount of power dissipated in hardware[5]. Since most embedded systems consist of microprocessor and software-oriented architecture, it stands to reason to utilize low-power techniques for processor performance optimization.

Modern embedded systems have highly efficient processors optimized for performance and power. An example of such a processor is CV32E40X RISC-V core, designed for embedded system applications by OpenHW Group.

The main goal of the project, done in collaboration with Silicon Labs, is to increase the performance of CV32E40X without increasing area and power by more than 20%. The metric for performance evaluation is the clock cycle per instruction (CPI). The current version of the core has 2.14 CPI, which means that the core needs approximately 21 400 clock cycles to execute a program with 10 000 instructions. The project will show that branch predictors can improve CPI by up to 4.31 % and reduce execution time by 900 clock cycles. Thus, branch predictors can increase the core performance.

This report is divided into five parts.

Part I – state of art, analyzes correlations between performance optimization and power constraints and discusses how the dissipated power restrains the performance increase.

Part II – CV3250X RISC-V core and Branch Prediction, explains the core architecture and several branch predictors that can be used as low-power solutions.

Part III – Methodology describes a high-level branch prediction unit model developed for CV32E40X RISC-V core.

Part IV – Results, discusses the extent of performance increase expected after implementing the high-level mode in the actual hardware.

Part V – Conclusion summarizes the author's work and discusses future work.

State of Art

The following section explains the motivation for performance optimization, states power limitations, and presents branch prediction as a low-power solution to the performance optimization problem.

The data deluge is the term used to define a situation where the sheer volume of generated data surpasses the system's capacity to manage it. When it comes to embedded system computing, data deluge reveals itself in the microprocessor's inability to analyze all the data received from sensors[4]. It is estimated that embedded systems around the globe have to compute 1000 Terabytes of information per minute, which is almost the same as reading 550 million books every minute. One way to reduce the gap between generated and analyzed data is to increase the performance of microprocessors used in embedded systems.

Throughout the past three decades, the performance increase was the direct result of Moore's law, which states that the number of transistors on a chip doubled every 18 months[11]. The design technology improvements and scaling down transistors enabled the digital system designers to increase the number of transistors on the chip, increasing computing capability.

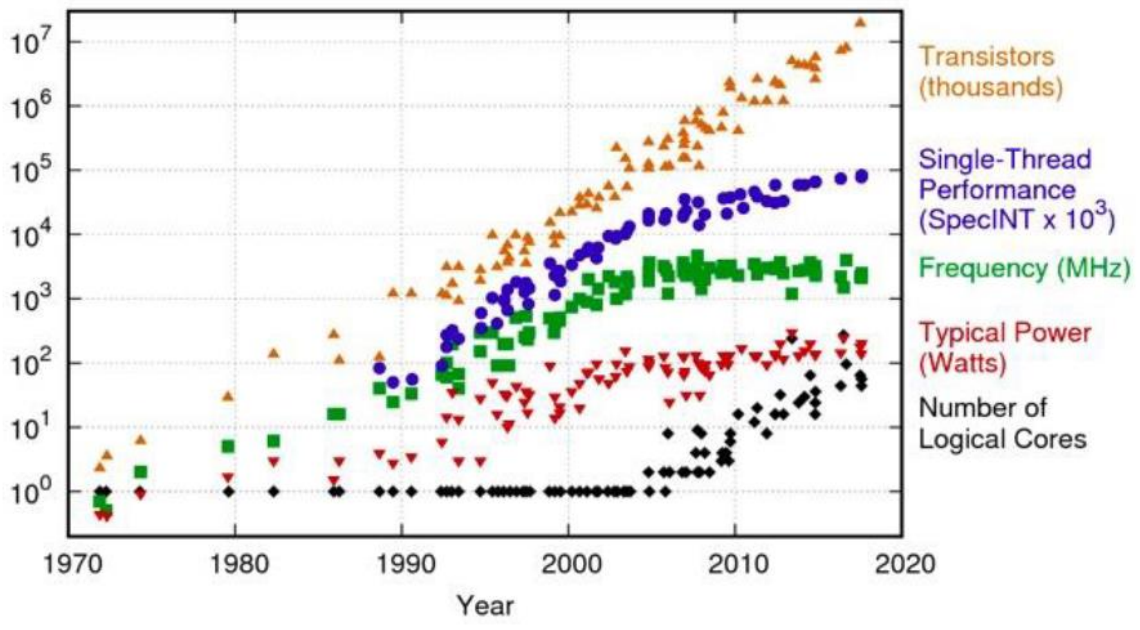


Figure 1: 42 years of microprocessor trend data[6]

However, it can be seen in figure 1 that since 2010 the increase in transistors no longer generates a significant increase in operating frequency[6]. As the number of transistors increases linearly, the frequency reaches saturation at the beginning of the previous decade.

Unlike frequency, as the number of transistors increases, so does the heat generated by power dissipation[5]. Figure 2 presents dissipated power per cm2 for the Pentium processor family with respect to the transistor channel length. Intel manufacturing discontinued Pentium family processors after Pentium IV. However, if they had continued this line, Pentium V would dissipate as much power as a nuclear reactor, and further improvements would result in dissipating as much power and heat as generated at space rocket launch[5].

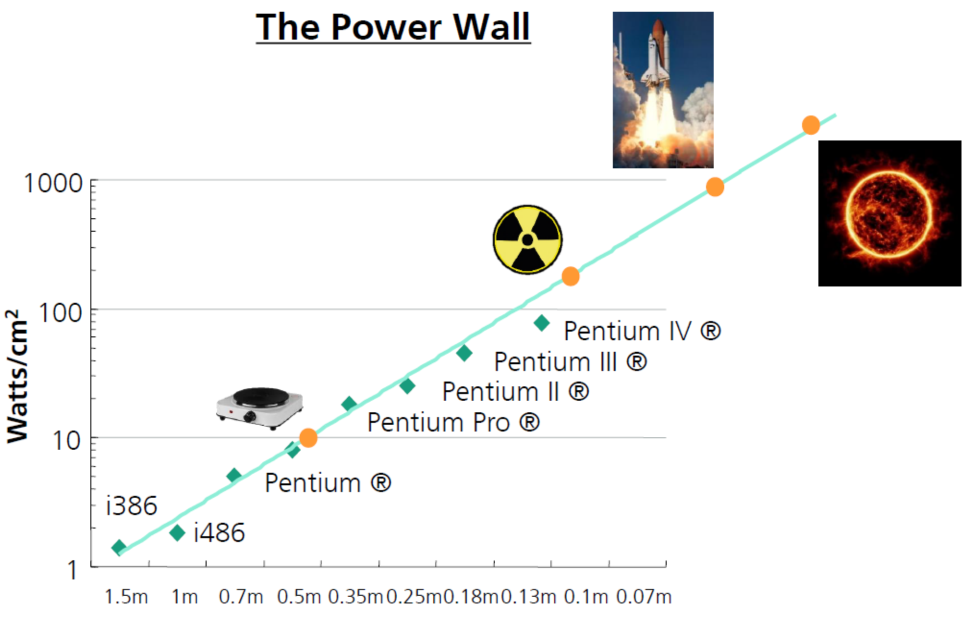


Figure 2: Power dissipated per cm^2 with respect to transistor channel length scaling[5]

Thus, increasing the performance of microprocessors used in embedded systems is essential if digital designers want to reduce the gap between generated and analyzed data. Since performance increase is limited by power dissipation, the designers need to employ clever design techniques that utilize low power and increase performance.

One way to increase performance is to change the microprocessor architecture in a way to be able to run it on a higher operating frequency. However, the power dissipated through the circuit is directly proportional to the operating frequency.

$$P = \alpha C V_{DD}^2 f \quad (1)$$

12

According to 1, dissipated power depends on switching probability (α), device-switching capacitance (C), squared supply voltage (V_{DD}), and operating frequency (f)[12]. Any design optimization that results in a frequency increase also significantly increases consumed power.

Hence, the performance optimization solution modeled in this project does not aim at frequency increase. The design approach, modeled throughout the project, is branch prediction which attempts to mitigate the cost of branching by predicting the correct branch outcome and fetching the correct instruction, resulting in the CPI decrease.

CV3250X RISC-V core and Branch Prediction

This section briefly describes CV3250X architecture, discusses several branch prediction techniques, and explains how a 2-bit branch predictor can be implemented in the core.

The project uses the CV3250X RISC-V core (Figure 3). It is simple in order core. The pipeline consists of four stages: Instruction fetch (IF), instruction decode (ID), execute (EX), and write back (WB) stage[7].

- IF stage – Instruction interface feeds 32-bit long value to the core. Depending on the core configuration, it can work with either 32 or 16-bit instructions[7].
- ID stage – The decoder decodes the instruction and fetches operands from the register file. If the instruction is a branch, the branch target address is calculated[7].
- EX stage – This stage evaluates operations based on the decoded instruction. The LSU unit calculates the memory address if the core executes load/store instruction. In the case of the branch instruction, ALU evaluates branch conditions and based on the result, the core decides whether the branch should be taken or not[7].
- WB stage – The register file is updated with the correct value, and memory interaction is finished[7].

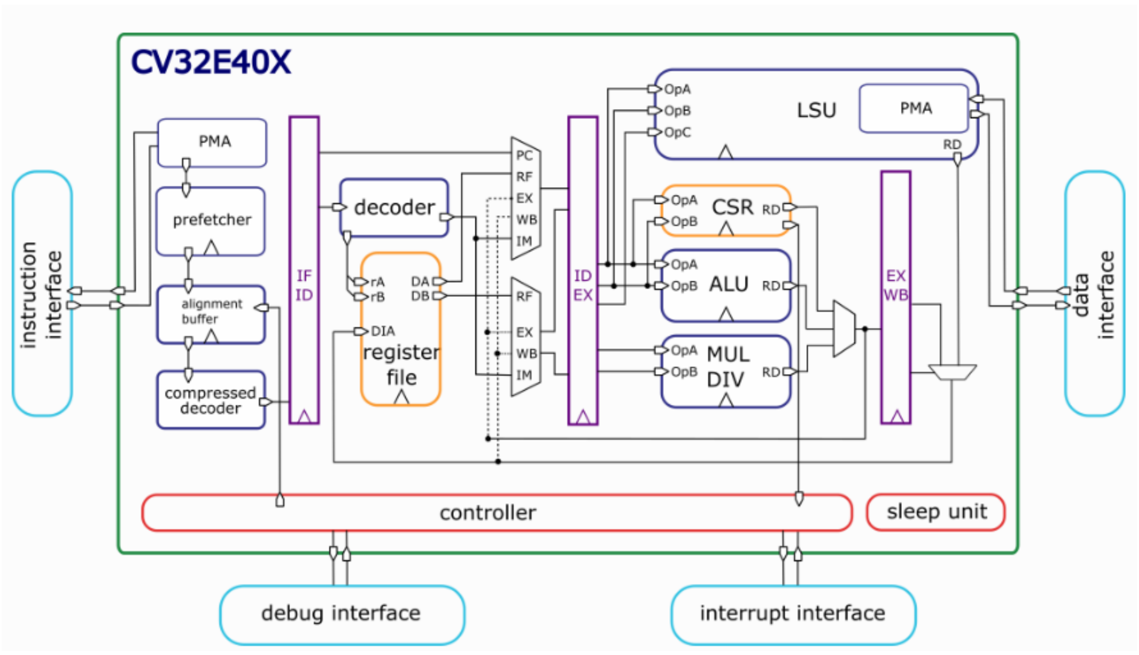


Figure 3: CV32E40X processor core[7]

Dynamic Branch Prediction predicts the branch outcome at runtime by analyzing the runtime information. Currently, the core does not use any dynamic branch prediction, and it assumes that none of the branches are taken. After branch instruction moves from the IF stage to the ID stage, the instruction is fetched from memory directly after the branch instruction. The core finds out the outcome of the branch condition in the EX stage. At this point, two instructions are fetched from memory and are in the IF and ID stage. If the branch condition is true, the program execution should not include the upper-mentioned two instructions. To avoid distorting the instruction execution order, whatever is present in the ID and IF stage needs to be flushed[8]. This event is called branch misprediction and is visualized in table 1.

Table 1: Mispredicted Branch instruction propagation through the pipeline

Time / Stage	IF	ID	EX	WB
T0	Branch	Instr2	Instr1	Instr0
T1	Instr3	Branch	Instr2	Instr1
T2	Instr4	Instr3	Branch	Instr2
T3	Instr5	Instr4	Instr3	Branch
T4	Instr6	Instr5	Instr4	Instr3
T5	Instr7	Instr6	Instr5	Instr4
T6	Instr8	Instr7	Instr6	Instr5

The green boxes represent branch instruction as it goes through the pipeline. The branch instruction jumps over instructions 3 and 4. However, since the core assumes that the branch is never taken, instr 3 and 4 are still fetched and propagated through the pipeline. To recover from the misprediction, the register file is not updated when they reach the WB stage. Hence, branch misprediction imposes 2 cycle penalty, delays the instruction execution, and wastes power on fetching and decoding unnecessary instructions. Two cycles might not seem a big deal. However, if the program consists of loops that iterate 1000 times, the core will mispredict 999 instructions and spend almost 2000 cycles on misprediction recovery.

Typically, a dynamic predictor consists of a branch history table (BHT) and branch target buffer (BTB). BHT is a small memory indexed by the branch instruction address and tracks the previous outcomes of the branch instruction. BTB is a small cache that stores the target address of the branch instruction[8]. The cleverly implemented dynamic branch prediction unit can reduce the number of mispredicted branches, resulting in performance increase and dissipated power reduction. Although this project implements a 2-bit predictor, several branch prediction schemes are worth discussing in detail.

<u>**1-bit Predictor**</u>:

Figure 4 portrays a block diagram for a 1-bit predictor. PC block is the program counter. BTB is a small cache indexed by the PC's current value. If the current pc points to a branch instruction recently executed, BTB asserts the Hit signal and outputs the branch target address. BHT uses 1-bit to store the history of branch instruction outcomes. 1 means that the branch prediction is TAKEN, 0 – NOT TAKEN. Figure 5 displays a state transition diagram for the value stored at BHT. If the value is 0 (prediction NOT TAKEN) and the branch was taken, the value changes to 1 (prediction Taken), and vice versa. However, if the prediction was correct, the value at BHT does not change.

If the branch has been recently taken, the bit associated with that particular branch is set to 1. Hence, whenever there is a hit and BHT outputs 1, the prediction is that branch will be taken, and the address of the next instruction (PC_nxt) will be assigned to the branch target. Otherwise, PC_nxt is assigned with PC + 4, the instruction address consecutive to the branch instruction.

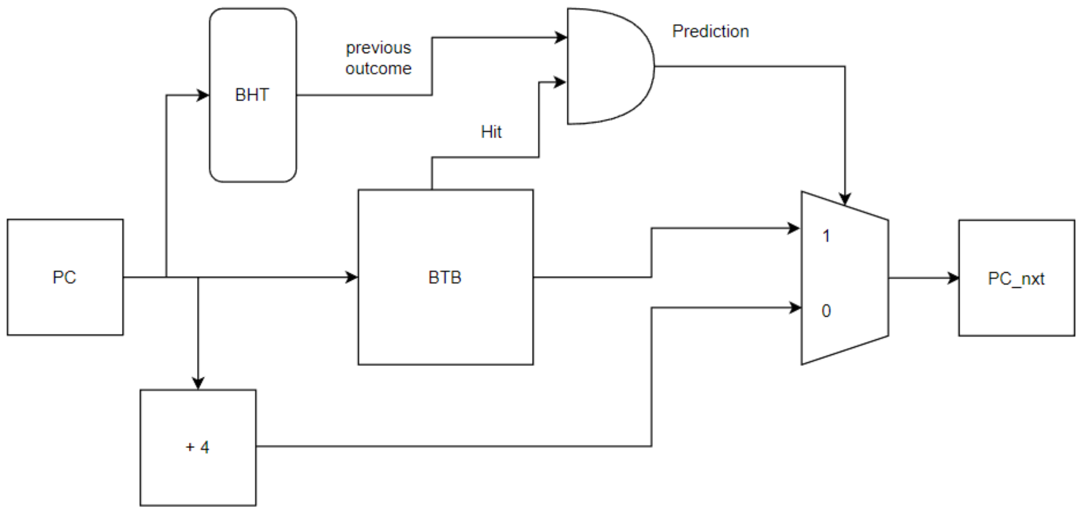


Figure 4: Block Diagram for 1-bit branch predictor

It should be noted that the block diagram from Figure 4 is implemented in the IF pipeline stage from figure 3. At the beginning of the program execution, BHT and BTB are empty. When the branch is fetched for the first time, its target address is identified in the ID stage and forwarded to BTB. The core also adds an entry for the fetched branch to BHT. By default, BHT assumes that the same branch will be taken next time. Hence, the BHT value for the fetched branch is 1. Thus, the 1-bit branch predictor has a "warm-up time." It starts generating correct predictions after the core has executed several branch instructions.

Although the 1-bit branch predictor reduces the number of mispredicted branches, it still mispredicts, and the core still needs to spend several clock cycles on instruction order recovery—moreover, the prediction changes based on the correctness of the previous prediction. As described above, the prediction is done in the IF stage. At the EX stage, the core has already evaluated the correctness of the prediction. If the prediction is wrong, the prediction bit toggles in BHT. Otherwise, BHT does not change[8].

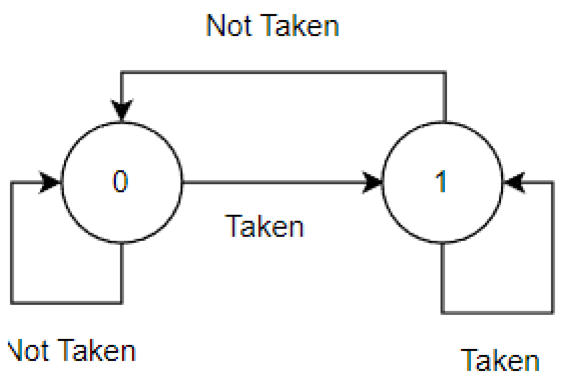


Figure 5: State diagram for BHT entries of the 1-bit branch predictor

2-bit predictor:

The main difference between 2-bit and 1-bit predictors is the number of bits used for each BHT entry[8]. As we have seen, a 1-bit predictor uses a single bit, and a single misprediction is enough to change the prediction from not taken to taken and vice versa.

While a 1-bit predictor has benefits, it sometimes does not work well. Sometimes a single misprediction is not enough to change the entire prediction. For example, table 2 displays a scenario where the prediction unit results in more than 1 misprediction simply because a single misprediction changes the entire prediction. T stands for Taken, TN – not taken, C – correct prediction, and I – incorrect prediction. After branch number 4 is executed, the 1-bit predictor mispredicts and changes the prediction to not taken, which results in mispredicting the outcome of branch number 5. As a result, the 1-bit predictor has 80% accuracy.

Table 2: 1-bit predictor making 2 mispredictions in a row.

		1	2	3	4	5	6	7	8	9	10
	Branch Outcome	T	T	T	NT	T	T	T	T	T	T
Single misprediction Changes prediction	Branch Prediction	T	T	T	T	NT	T	T	T	T	T
	Branch Prediction Correctness	C	C	C	I	I	C	C	C	C	C
Two mispredictions Changes prediction	Branch Prediction	T	T	T	T	T	T	T	T	T	T
	Branch Prediction Correctness	C	C	C	I	C	C	C	C	C	C

However, if the predictor needs two consecutive mispredictions to change the prediction, the accuracy increases to 90% because after branch 4 the predictor still predicts that the branch will be taken. 2-bit predictor utilizes this idea.

A 2-bit predictor uses two bits for each BHT entry. 00 means strong not taken prediction, 01 – weak not taken prediction, 10 – weak taken prediction, and 11 – strong taken prediction. Figure 6 displays the state diagram for BHT entries. A single misprediction does not result in changing the prediction; it just reduces the "credibility" of the prediction. The prediction change is possible in case of two consecutive mispredictions[8].

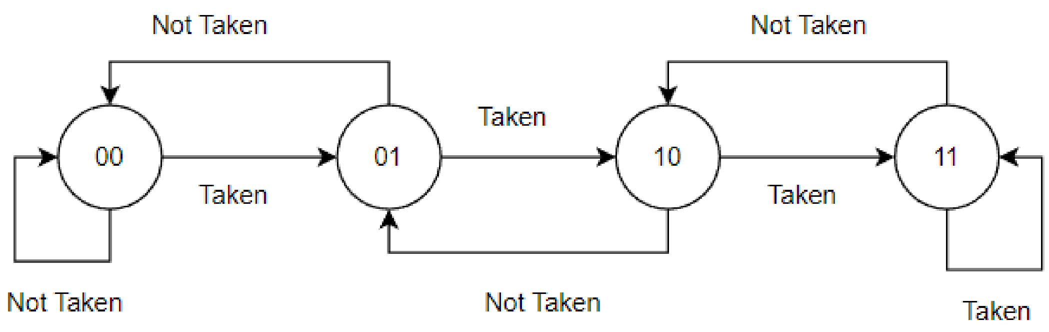


Figure 6: State diagram for BHT entries of the 2-bit branch predictor

Figure 7 displays a block diagram for a 2-bit branch predictor. Prediction is TAKEN if the output of BHT is 10_2 (weak taken prediction) or 11_2 (strong taken prediction). This is checked by comparing the value with 1_{10}. Like the 1-bit predictor, the 2-bit predictor is implemented in the IF pipeline stage from figure 3, the branch target is forwarded from the ID stage, and branch prediction correctness is checked in the EX stage. If the prediction is incorrect, BHT is updated according to the state diagram shown in figure 6.

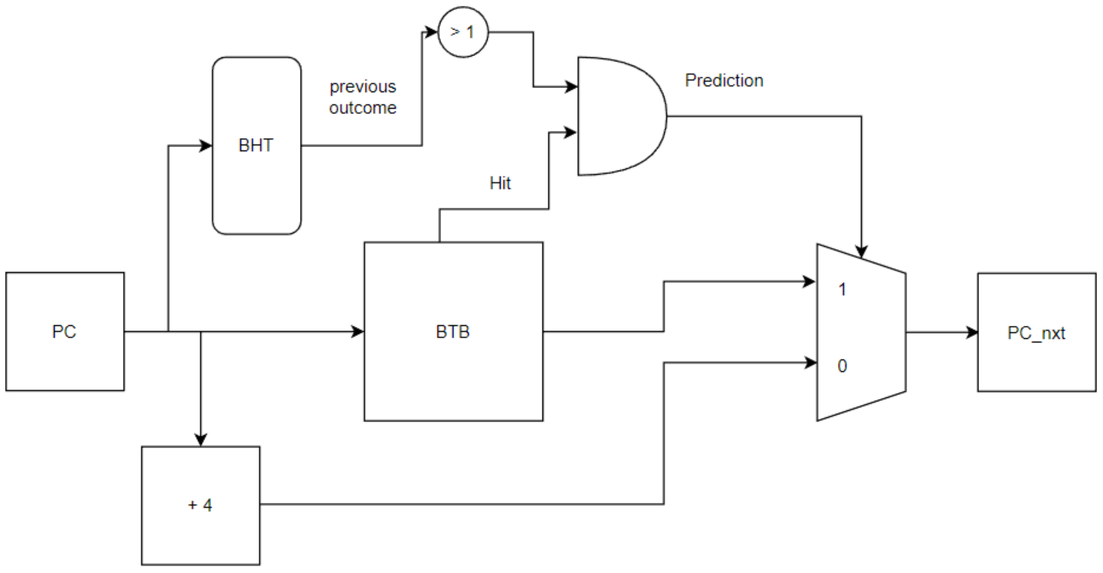


Figure 7: Block Diagram for 2-bit branch predictor

Correlating Branch Predictors:

A 2-bit predictor uses the history of a single branch to make a prediction. However, there are cases when the outcome of one branch affects the outcome of another. Hence, the accuracy of the 2-bit predictor can be improved if it considers the recent outcomes of several branches. For example, figure 8 displays a situation such that if the first two branches are taken, the last one will not be taken[9].

```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb)
```

Figure 8: outcome of the first two branches affects the outcome of the third one[9]

Branch predictors that consider other branches' behavior when making a prediction are called correlating branch predictors[9]. Such predictor has a global history table (GHT) that stores recent outcomes of m branches. The global history table can be implemented as an m-bit shift register[9]. As can be seen from figure 9, correlating predictor indexes BTB and BHT with branch address (pc) and value of GHT. (m, n) correlating predictor uses the behavior of m branches and n-bit predictor to make a prediction. Hence, (m,n) correlating predictor has 2^m n-bit predictors for each branch instruction[9]. If the example from figure 8 is executed on a microprocessor with (2,2) predictor, BTB will 4 have entries for the third branch instruction. One of those entries is for the scenario when the first two branches are taken. In this case, the unit will always predict that the third branch is not taken, resulting in higher accuracy than the 2-bit predictor[9].

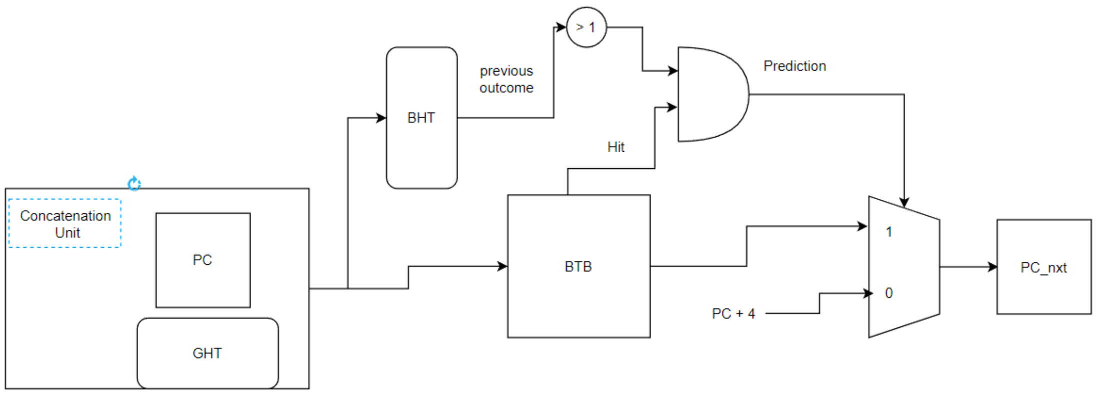


Figure 9: block diagram of correlating branch predictor

Tournament Predictors:

Tournament predictor uses a combination of different predictors to make the decision. Different predictors have different trust level, which changes at runtime, and a selector unit selects between different predictions based on their trust level[9].

As mentioned above, dynamic branch predictors have a "warm-up time," meaning the predictors start generating accurate predictions after program execution has gone through several branch instructions. Hence, it stands to reason to use a combination of predictors with different "warm-up times." An example of such tournament predictors is the combination of backward branch and 2-bit predictors. A backward branch predictor makes a prediction based on the direction of the branch. If the target address is lower than the current address, the branch is predicted to be taken. The backward branch predictor does not have a "warm-up" time, but it is less accurate than the 2-bit predictor. At the beginning of the program execution, the core will prioritize the outcome of the backward branch predictor. However, as the 2-bit predictor "warms up," the outcome of the backward branch predictor will be ignored. Hence, using a combination of those two units help the core start producing accurate predictions as early as possible.

2-bit predictor in the CV3250X Core:

This section briefly discussed several low-power solutions to the microprocessor optimization problem. All the solution was a variation of the branch prediction unit. Figure 10 portrays how a 2-bit branch predictor can be implemented in the CV3250X core. This block diagram is a combination of figure 7 and figure 3. The predictor is implemented in the IF stage of the pipeline. Branch target addresses are forwarded from the ID stage to BTB. The prediction is made in the IF stage. The branch condition is evaluated in the EX stage. If the condition is true, the branch should be taken, else – not taken. Based on the condition evaluation, update BHT checks whether the prediction was correct and updates BHT values according to the state transition diagram in figure 6.

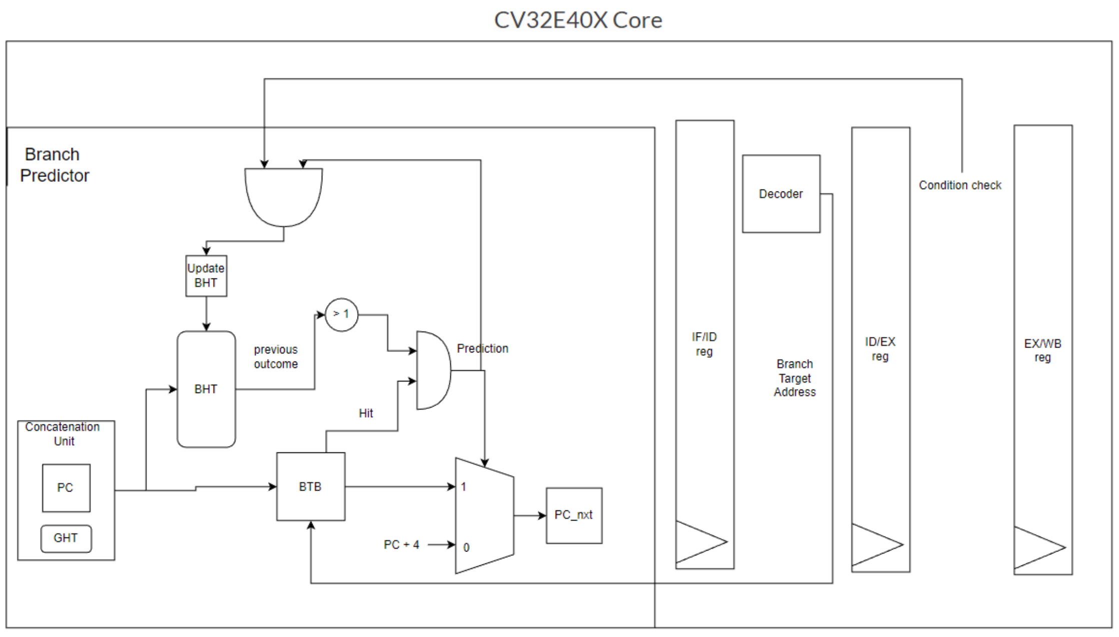


Figure 10: 2-bit predictor implemented in CV3250X core

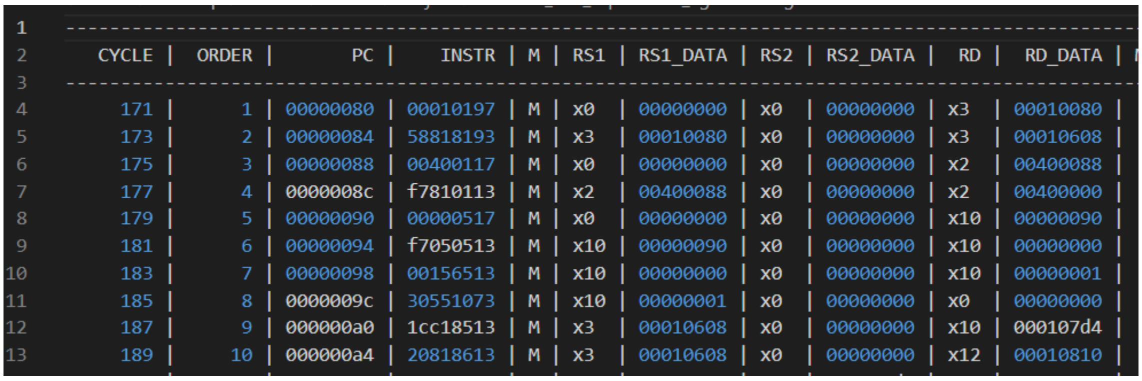
Methodology

The design approach, modeled throughout the project, is branch prediction which attempts to mitigate the cost of branching by predicting the correct branch outcome and fetching the correct instruction, resulting in the CPI decrease. The author of this report implemented a high-level model for a 2-bit branch predictor in python. The goal of modeling a 2-bit predictor is to evaluate performance improvement regarding CPI decrease. This section describes the high-level model.

The project utilized a verification environment provided by Silicon Labs[10]. The author used the environment to simulate the core and run custom programs written in high-level programming languages. The log file is generated at the end, which conveys information about executed instructions. Figure 11 portrays an example of such a log file. Each row represents one instruction. The table has the following columns:

- Cycle – specifies the clock cycle in which the core finished the instruction execution.
- Order – specifies the order in which the instructions were issued.
- PC – specifies the address of the fetched instruction.
- INSTR – specifies the instruction encoding as a hexadecimal number.
- RS1/2 – address of the first/second operand read from register file
- RS1/2_Data – value read from the register file.
- RD/RD_Data – specifies destination address/value to be stored in the destination register.

INSTR, PC, and CYCLE columns are essential for this project.



```
 -------------------------------------------------------------------------------------------------------------
   CYCLE |  ORDER |        PC |    INSTR | M | RS1  | RS1_DATA | RS2 | RS2_DATA |  RD |  RD_DATA |
 -------------------------------------------------------------------------------------------------------------
     171 |      1 | 00000080 | 00010197 | M | x0   | 00000000 | x0  | 00000000 | x3  | 00010080 |
     173 |      2 | 00000084 | 58818193 | M | x3   | 00010080 | x0  | 00000000 | x3  | 00010608 |
     175 |      3 | 00000088 | 00400117 | M | x0   | 00000000 | x0  | 00000000 | x2  | 00400088 |
     177 |      4 | 0000008c | f7810113 | M | x2   | 00400088 | x0  | 00000000 | x2  | 00400000 |
     179 |      5 | 00000090 | 00000517 | M | x0   | 00000000 | x0  | 00000000 | x10 | 00000090 |
     181 |      6 | 00000094 | f7050513 | M | x10  | 00000090 | x0  | 00000000 | x10 | 00000000 |
     183 |      7 | 00000098 | 00156513 | M | x10  | 00000000 | x0  | 00000000 | x10 | 00000001 |
     185 |      8 | 0000009c | 30551073 | M | x10  | 00000001 | x0  | 00000000 | x0  | 00000000 |
     187 |      9 | 000000a0 | 1cc18513 | M | x3   | 00010608 | x0  | 00000000 | x10 | 000107d4 |
     189 |     10 | 000000a4 | 20818613 | M | x3   | 00010608 | x0  | 00000000 | x12 | 00010810 |
```

Figure 11: Log file generated by simulating CV32E40X core

As seen in figure 12, the 2-bit predictor model parses through the log file and generates a new one. If a 2-bit predictor were implemented in the hardware, the log file generated by the verification environment and by the 2-bit predictor model would be identical.

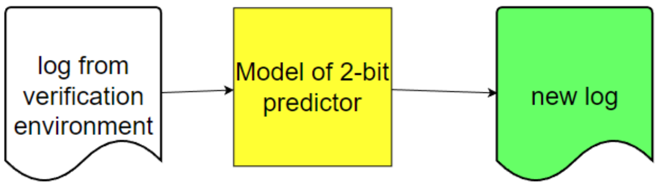
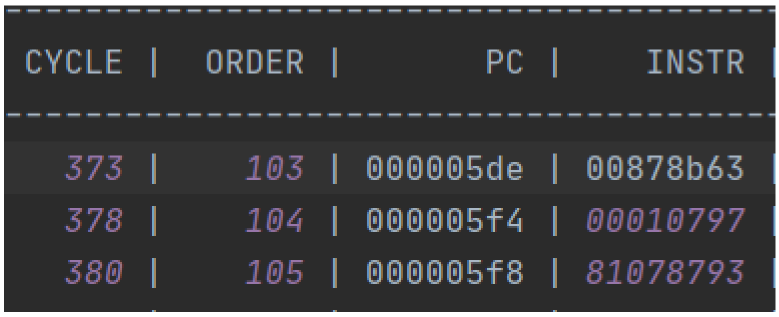


Figure 12: 2-bit predictor input/output

The model uses the last two digits of INSTR to identify branch instruction. The opcode for the branch is 1100011_2 and is encoded with the 7 least significant bits of the instruction word[3]. Thus, to identify branches, the model extracts the 7 least significant bits out of the last two digits of INSTR and compares them to 1100011_2.

Once we know which lines represent branches, we can determine which were TAKEN and NOT TAKEN by comparing the PC of the branch instruction and the PC of the following instruction. If the difference between those two values is 4, the branch was not taken; otherwise – taken. The example is shown in figure 13. Instruction number 103 is a branch instruction. The difference between PC of 104 and 103 is 22_{10} (PC values are in hexadecimal). Hence, the branch was taken.



```
-------------------------------------
CYCLE |  ORDER |        PC |    INSTR
-------------------------------------
  373 |    103 | 000005de | 00878b63
  378 |    104 | 000005f4 | 00010797
  380 |    105 | 000005f8 | 81078793
```

Figure 13: Branch taken

Finally, as mentioned above, the model takes the log file generated by the verification environment, applies a 2-bit branch predictor, and generates a new log file. The main difference between original and new log files is CYCLE. Suppose the predictor's prediction is correct, and the branch is taken. In that case, the core does not have to spend 2 cycles on execution order recovery, and all the instructions after the branch finish 2 cycles earlier. Hence, 2 is subtracted from the CYCLE values of all the instructions after the branch. For example, if we again consider the example from figure 13, the CYCLE values for instructions 104 and 105 will be 376 and 378 in the new log file.

On the other hand, if the prediction is incorrect and the branch is not taken, the core must spend an extra 2 cycles on execution order recovery. Hence, 2 is added to the CYCLE values of all the instructions after the branch.

It is worth mentioning that model's prediction – NOT TAKEN, does not change the log file because the current version of the core also assumes that branches are not taken.

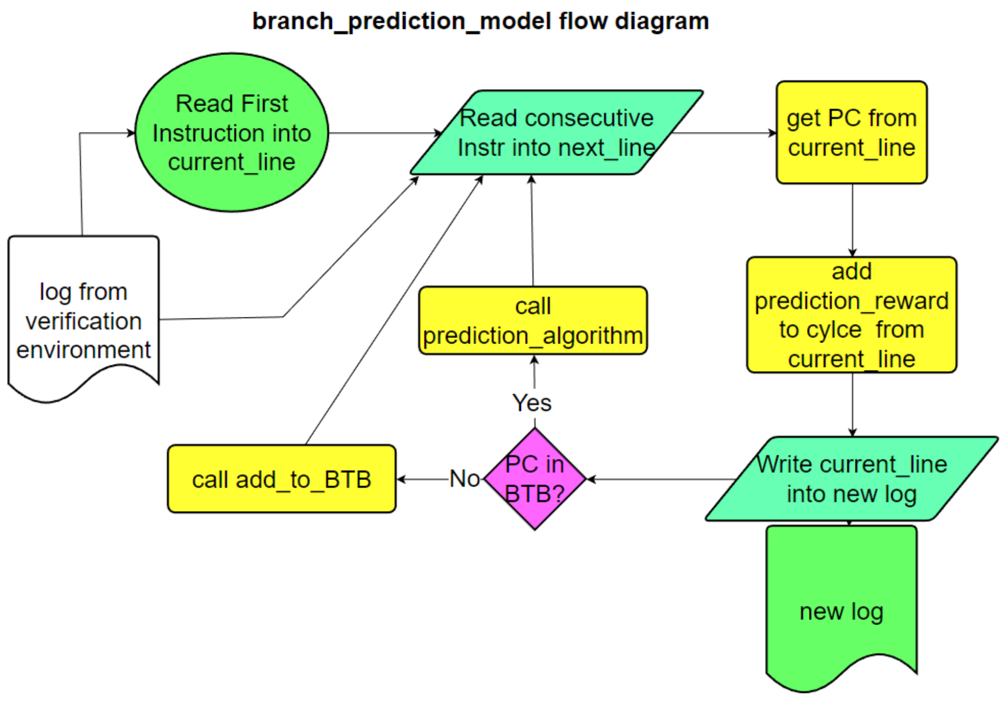
The model is represented as a python class. The class has four important attributes: BTB, size, access_hist_list, and prediction_reward.

- **BTB** is implemented as a python dictionary and emulates the branch target buffer and branch history buffer of the 2-bit predictor. The branch target buffer and branch history buffer are indexed with the branch instruction address, and the former stores branch target address and the letter stores prediction variable. Hence, PC values of branch instructions are used as BTB directory keys. BTB[PC] returns a list of two elements – branch target address and prediction variable from BHT. BHT is part of BTB.
- **Size** specifies the maximum number of entries in the BTB directory.

- **Access_hist_list** is a sorted list of BTB keys. The keys are sorted from the least recently accessed to the most recently accessed. Since BTB does not have an infinite size, when it is full and new branch instruction needs to be added, the least recently accessed entry of BTB will be evicted from the dictionary.
- **Prediction_reward** is initialized with 0 at the class object creation. When a new log is generated, the new entry's cycle column is changed by the value stored in prediction_reward. If the branch prediction is correct and the branch is taken, prediction_reward is increased by -2 (removing 2 cycles for branch prediction). If the prediction is wrong and the branch is not taken, prediction_reward is increased by 2 (adding 2 cycles for branch misprediction). However, if the prediction is NOT TAKEN, prediction_reward does not change.

The model reads the log file generated by the verification environment, line by line, to emulate instruction fetch from memory. The model applies a 2-bit predictor to "fetched" instructions and generates a new log file, which the verification environment would generate if the predictor were already implemented in the core. All these are accomplished by the following class methods: branch_prediction_model, add_to_BTB, and prediction_algorith.

- **branch_prediction_model** flow diagram can be seen in figure 14. The method gets the first line from the log file and stores it in the variable current_line. After that, the consecutive line is read into variable next_line.
  The cycle column of the current_line is updated with prediction_reward, and the current_line is written into the new log file.
  Next, if the pc of the current_line is in the BTB, the instruction is the branch, and the method prediction_algorithm is called. Otherwise, if the instruction is the branch, it is added to BTB by calling the method add_to_BTB.



branch_prediction_model flow diagram
Read First Instruction into current_line
Read consecutive Instr into next_line
get PC from current_line
log from verification environment
call prediction_algorithm
add prediction_reward to cylce from current_line
Yes
call add_to_BTB
No
PC in BTB?
Write current_line into new log
new log

Figure 14: flow diagram for the method branch prediction model

- **add_to_BTB** flow diagram can be seen in figure 15. The method gets two arguments – PC and instruction (underlined on the diagram).
  First, the method calculates the branch target address by extracting the branch offset from the instruction and adding it to the PC.
  Next, the method checks if the BTB is not full. If that is the case, the new entry is appended to BTB. The branch target and prediction variable – 3 are added to BTB. Otherwise, the new entry replaces the least recently accessed entry.

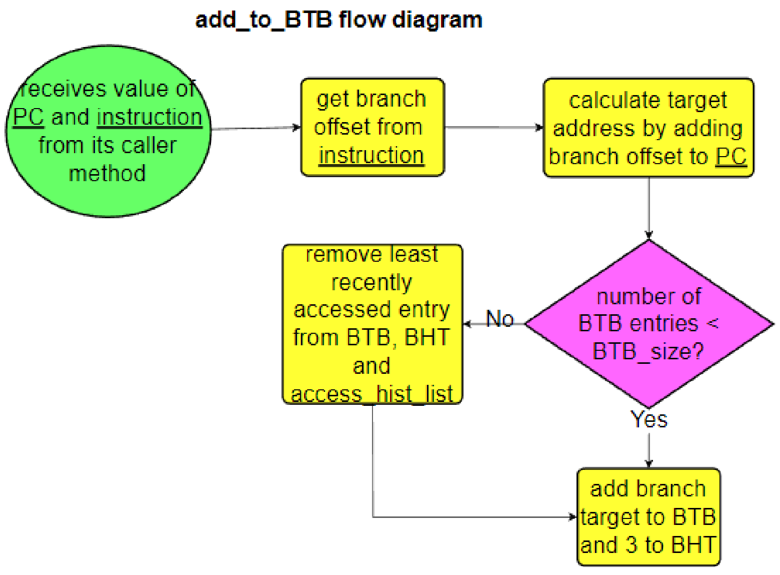

add_to_BTB flow diagram
receives value of PC and instruction from its caller method
get branch offset from instruction
calculate target address by adding branch offset to PC
number of BTB entries < BTB_size?
No
remove least recently accessed entry from BTB, BHT and access_hist_list
Yes
add branch target to BTB and 3 to BHT

Figure 15: flow diagram for method add_to_BTB

- **prediction_algorithm** (flow diagram in figure 16) receives pc – address of the executing instruction, current_line – executing branch instruction, and next_line – the instruction that should be executed after the branch (underlined on flow diagram).
  At first, the algorithm makes the prediction based on the value stored in the corresponding BTH entry. If the value is more than 1, the model predicts that the branch will be taken. Otherwise – not taken.
  If the prediction was TAKEN and the branch was taken, the prediction_reward variable decreases by 2, which means that the core does not have to spend 2 cycles on correcting misprediction. Otherwise, the prediction_reward variable increases by 2, meaning the core must spend another 2 cycles correcting the instruction execution order.
  If the prediction was NOT TAKEN, there is nothing to be modified because the core without predictor would have also assumed that the branch was not taken.
  Finally, the BTH section of BTB is updated. For correct predictions, if the branch was taken, the BTH entry is incremented by 1, else it is decremented by 1. For incorrect predictions, if the branch was taken, the BTH entry is decremented by 1, else it is incremented by 1.

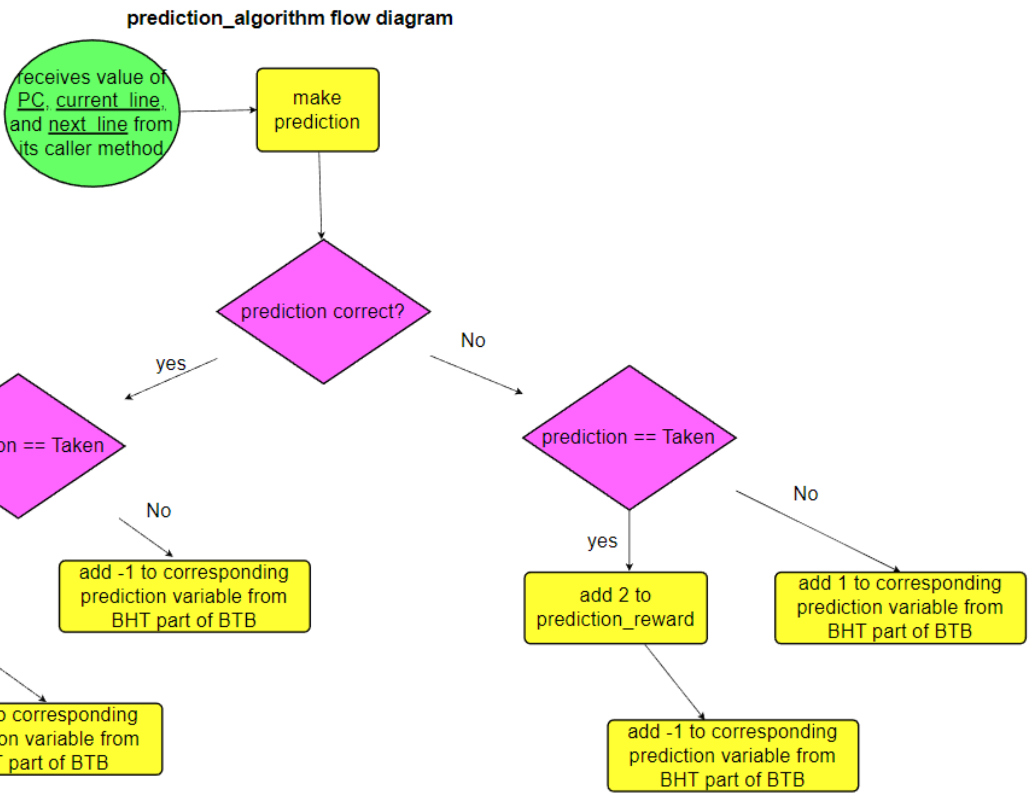

prediction_algorithm flow diagram
receives value of PC, current_line, and next_line from its caller method
make prediction
prediction correct?
yes
No
prediction == Taken
yes
No
add -2 to prediction_reward
add -1 to corresponding prediction variable from BHT part of BTB
add 1 to corresponding prediction variable from BHT part of BTB
prediction == Taken
yes
No
add 2 to prediction_reward
add 1 to corresponding prediction variable from BHT part of BTB
add -1 to corresponding prediction variable from BHT part of BTB

Figure 16: flow diagram for method prediction_algorithm

The source code for the 2-bit branch predictor model can be found in the appendix.

Results

The design approach, modeled throughout the project, was branch prediction which attempts to mitigate the cost of branching by predicting the correct branch outcome and fetching the correct instruction, resulting in the CPI decrease. The goal of modeling the 2-bit predictor was to evaluate performance improvement in terms of CPI decrease. This section describes the test performed on the model, summarizes and elaborates on results, and discusses the drawback of implementing a 2-bit branch predictor.

The test analyzes the log file generated by the verification environment to establish the baseline and produces statistics for the core without dynamic branch prediction (table 3). All the results obtained from the dynamic predictions are compared with the baseline values. The program has 9929 instructions. The current version of the core took 21207 clock cycles to finish the execution. The program contained 1217 branch instructions. The current version of the core assumes that branches are never taken, and this assumption was correct 510 times and incorrect 707 times. The CPI of the core without dynamic prediction is 2.14. The current version of the core mispredicts more than 50% of branch instructions and spends more than a thousand clock cycles on execution order recovery.

Table 3: statistics for core without dynamic branch prediction – baseline case

Number of Instructions	9 929
Number of Clock Cycles	21 207
Number of Branch Instructions	1 217
Number of Correct Predictions	510
Number of Incorrect Predictions	707
CPI	2.14

Next, to emulate the ideal case, the test generates statistics for the core with perfect branch prediction (table 4). The perfect predictor does not mispredict any of the branches. Since it does not waste clock cycles on the order recovery, the core requires 19739 cycles to finish execution. The baseline value for incorrect predictions is 707, and correcting 707 mispredictions requires 707*2=1414 clock cycles. Hence, the ideal case requires 1414 fewer clock cycles than the baseline case. The minimum CPI obtained as a result of branch prediction is 1.99. This is the maximum obtainable performance increase.

Table 4: statistics for core with perfect branch prediction – ideal case

Number of Instructions	9 929
Number of Clock Cycles	19 739
Number of Branch Instructions	1 217
Number of Correct Predictions	1 217
Number of Incorrect Predictions	0
CPI	1.99

Finally, to emulate a realistic case, the test runs prediction_model on the log generated by the verification environment. It generates a new log as if a 2-bit predictor has already been implemented in the real hardware. The test analyzes the new log file and produces statistics for the core with 2-bit branch prediction (table 5). This case mispredicts 457 fewer branches than the baseline. Hence, the core with a realistic predictor spends 457 * 2 = 914 clock cycles less on the order recovery than the baseline. Therefore, the core with 2-bit branch prediction needs fewer clock cycles (20293 cycles) to execute the program than the baseline (21207 cycles). This results in a CPI decrease from 2.14 to 2.04. Hence, the core with 2-bit branch prediction is faster than the core without dynamic branch prediction.

Table 5: statistics for core with 2-bit branch prediction – realistic case

Number of Instructions	9 929
Number of Clock Cycles	20 293
Number of Branch Instructions	1 217
Number of Correct Predictions	967
Number of Incorrect Predictions	250
CPI	2.04

Table 6 summarizes performance estimates for the core without branch predictor, the core with perfect predictor, and the core with 2-bit predictor, and identifies performance improvement of cores with ideal and realistic predictions compared to the baseline core. The performance improvements are calculated according to (2).

$$improvement = \frac{|new\ value - baseline\ value|}{baseline\ value} \times 100\%\ (2)$$

The system was tested with 9929 instructions. If there is no dynamic branch prediction (baseline), the core needs 21 207 clock cycles to finish the program execution. Furthermore, 58% of branch instructions are mispredicted, and 6.67% of clock cycles are spent on recovery. CPI for such microprocessor is 2.14.

If the unit had the perfect branch prediction (ideal case), the core would need 19 793 clock cycles to finish 9929 instructions. The CPI would decrease to 1.99; hence, the maximum performance increase based on CPI is 6.67%.

Unfortunately, there is no such thing as a perfect branch instruction. The simulation revealed that the core with the realistic 2-bit predictor requires 20293 clock cycles, and the CPI decreases from 2.14 to 2.04, which results in performance increases of 4.67%. The performance increase caused by a realistic predictor is close enough to that of the ideal predictor. Thus, the 2-bit predictor results in a performance increase. Furthermore, the realistic predictor's prediction accuracy increased by 89.6%, meaning that the core spends much less time fetching and decoding instructions that will be discarded, resulting in reduced power consumed while recovering the execution order. The 2-bit predictor results in a performance increase, which was the goal of this project.

Table 6: Summary of performance for the core without predictor, with perfect predictor, with the 2-bit predictor. BTB size 400 Bytes

	Statistics without branch prediction (baseline)	Statistics with perfect branch prediction (ideal)	Improvement Compared to baseline (%) (ideal)	Statistics with 2-bit branch prediction (realistic)	Improvement Compared to baseline (%) (realistic)
Number of Instructions	9929	9929	-	9929	-
Number of Clock Cycles	21207	19793	6.67%	20293	4.31%
Correct Predictions	510	1217	138%	967	89.6%
Incorrect Predictions	707	0	100%	250	64.6%
CPI	2.14	1.99	6.67%	2.04	4.67%
Clock Cycles Spent on Recovery	6.67%	0%	100%	2.5%	62.5%

Branch predictor drawback: Since dynamic predictors focus on branches, the program's performance increase is limited by the number of branch instructions. This is the main drawback of implementing a 2-bit branch predictor. Although the predictor consumes power, the performance increase for programs with negligible number of branches will be insignificant.

Furthermore, the predictions and branch target addresses are stored in BTB. Hence, it stands to reason to argue that BTB size significantly affects performance increase. However, as seen in figure 17, the performance does not improve drastically for larger BTBs. This can be explained by the limitation mentioned above. If the program has 2000 branches, BTB with 2000 entries will increase the same performance as BTB with 2 000 000.

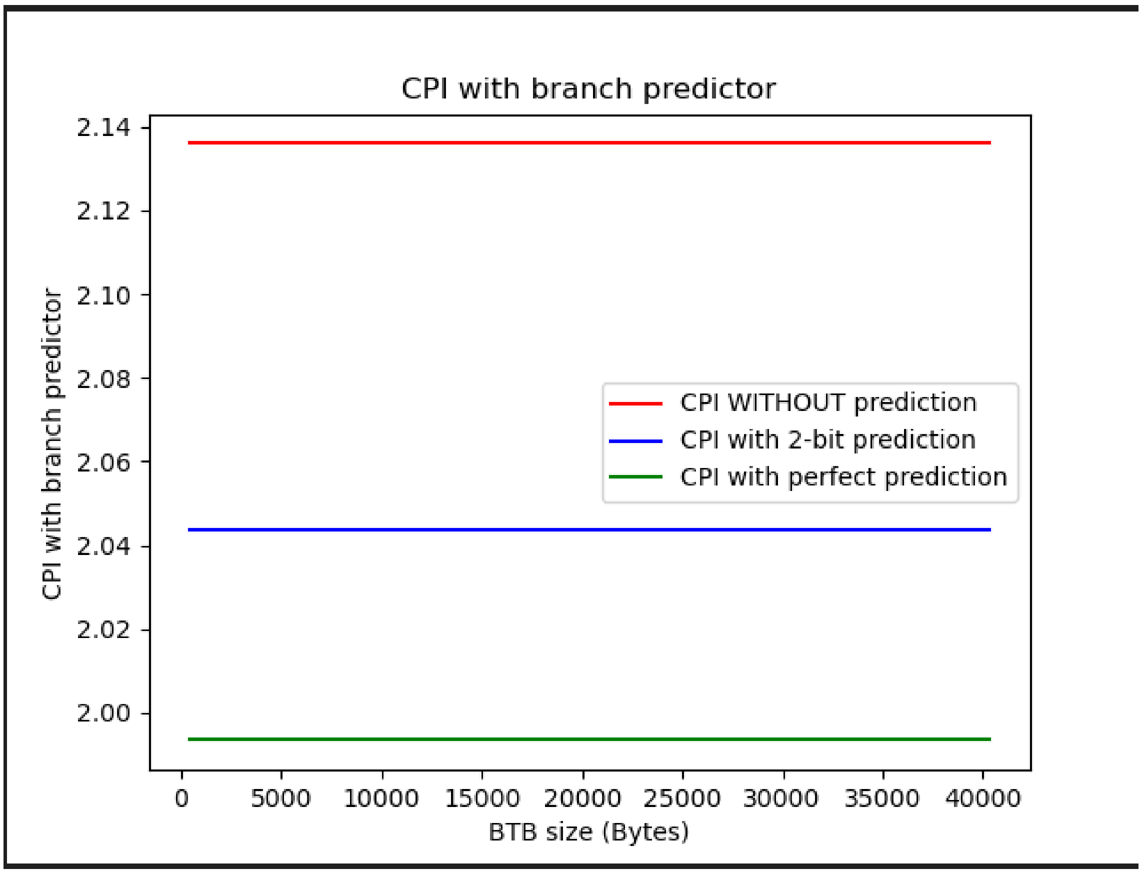


Figure 17: CPI with respect to BTB size (large size variation). Red -baseline, Blue – realistic, Green - ideal

Overall, the results revealed that the 2-bit branch predictor mitigates the cost of branching by predicting the correct branch outcome and fetching correct instruction, resulting in higher performance in terms of CPI decrease.

Conclusion

In conclusion, Nowadays, embedded systems are everywhere. Digital design has recently become more software-oriented, and closed architectures give way to open ones. To keep up with computing the overwhelming information, microprocessors for embedded systems need to be performance optimized. Power is one of the major limiting factors when it comes to performance optimization.

This project addressed the problem of developing a low-power solution for the performance optimization problem. The project was carried out on CV32E40X RISC-V core. Several techniques for low-power performance optimization were analyzed, and one of them, 2-bit branch prediction, was modeled in python. The results revealed that even a simple 2-bit branch prediction with just 100 entries increases performance. The next step is implementing a 2-bit predictor in actual hardware and testing the new core with real-life applications.

References

1) The apollo guidance computer. (2021, January 26). Retrieved December 19, 2022, from https://future-markets-magazine.com/en/innovators-en/the-apollo-guidance-computer/

2) Embedded systems: The Evolution of Embedded System design. (n.d.). Retrieved December 19, 2022, from https://www.electronicspecifier.com/products/design-automation/embedded-systems-the-evolution-of-embedded-system-design

3) Waterman, Andrew; Asanović, Krste (13 December 2019). "The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA version 20191213" (PDF). RISC-V International. Retrieved 5 November 2021.

4) Data deluge. (n.d.). Retrieved December 19, 2022, from https://itlaw.fandom.com/wiki/Data_deluge

5) Jung, M. (2021, October). *SystemC and Virtual Prototyping*. Lecture, Kaiserslautern.
6) Wehn, N., Dr. (2021, October). *Design of Microelectronic Systems & Circuits*. Lecture, Kaiserslautern.

7) CV32E40X (n.d.). Retrieved December 19, 2022, from https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/intro.html

8) Patterson, D. A., & Hennessy, J. L. (2022). *Computer Organization and Design: The hardware/software interface, sixth edition, RISC-V Edition*. Beijing Shi: Ji xie gong ye chu ban she.

9) Hennessy, J. L., Patterson, D. A., & Asanovic, K. (2019). *Computer Architecture: A quantitative approach*. Cambridge (Estados Unidos): Morgan Kaufmann.

10) Openhwgroup. (n.d.). Openhwgroup/core-V-verif: Functional verification project for the core-V family of RISC-V cores. Retrieved December 19, 2022, from https://github.com/openhwgroup/core-v-verif

11) Moore's law - kth. (n.d.). Retrieved December 19, 2022, from https://www.kth.se/social/upload/507d1d3af276540519000002/Moore%27s%20law.pdf

12) E., W. N., & Harris, D. M. (2011). *Integrated Circuit Design*. Boston: Pearson.

Appendices

python model is in the attached zip file. Source Code:

```
from collections import defaultdict
import matplotlib.pyplot as plt
# Define path to the log files you want to parse
logPath =
"uvm_test_top.env.rvfi_agent.trn.log"#"/home/giorgis/project/core-v-
verif/cv32e40x/sim/uvmt/vsim_results/default/hello-
world/0/uvm_test_top.env.rvfi_agent.trn.log"
branchInstrsPath = "branchInstrs.rvfi_agent.trn.log"

#"/home/giorgis/project/branchPredictionModel/files/branchInstrs.rvfi_
agent.trn.log"

takenBranchPath    = "takenBranchPath.rvfi_agent.trn.log"
notTakenBranchPAth = "notTakenBranchPAth.rvfi_agent.trn.log"

size = 100
newLogPath = "new_uvm_test_top.env.rvfi_agent.trn.log"

class model:
    # Initialize the paths
    def __init__(self, logPath, branchInstrsPath, newLogPath, size):
        self.logPath = logPath
        self.branchInstrsPath = branchInstrsPath
        self.newLogPath = newLogPath

        self.number_of_instructions = 0
        self.number_of_clkCyc_pre_prediction = 0

        self.number_of_brnch_instr = 0
        self.number_taken     = 0
        self.number_not_taken = 0

        self.number_of_mispredict_pre_prediction = 0
        self.mispred_clk_cyc_penalty_pre_prediction = 0

        self.fraction_mispredict_branch_pre_prediction = 0
        self.fraction_mispredict_penalty_cyc = 0

        self.cpi_pre_prediction = 0

        self.number_of_clkCyc_perfect_prediction = 0
        self.cpi_perfect_prediction = 0

        self.number_of_clkCyc_brnach_prediction = 0
        self.cpi_brnach_prediction = 0

        self.BTB = {}
        self.access_hist_list = []
        self.BTB_size = size
        self.prediction = 3

        self.prediction_reward         = 0
        self.num_correct_prediction    = 0
        self.num_incorrect_prediction = 0
```

```
######################################################################
######################################################3

        self.instr_strtIndex = 0 # this is accurate
        self.instr_endIndex  = 0 # this is accurate

        self.cycle_strtIndex = 0 # = self.cycle_endIndex - size(cycle)
        self.cycle_endIndex  = 0 # this is accurate

        self.order_strtIndex = 0
        self.order_endIndex  = 0

        self.notTakenBranchDelay_dict = {}
        self.takenBranchDelay_dict    = {}

        self.notTakenBranchOrder_dict = {}
        self.takenBranchOrder_dict    = {}

        self.notTakenBranch_instr_name_dict = defaultdict(set)
        self.takenBranch_instr_name_dict    = defaultdict(set)

        print("\nClass object has been created\n")

    # Selects branch instructions from the log File and saves them
into separate file
    # @param:  none
    # #return: number of branch instructions
    def read_branch_instrs(self):

        print("Reading log file from "+self.logPath)
        print("Storing Branch Instructions in "+self.branchInstrsPath
+ "\n")

        with open(self.logPath, "r") as logFile_ptr,
open(self.branchInstrsPath, "w") as branchInstrsFile_ptr:

            for i in range(3):
                first_3_lines = logFile_ptr.readline()
                branchInstrsFile_ptr.writelines(first_3_lines)
            print("Finished copying "+ str(i+1) + " header lines\n")
            i = 0

            for line in logFile_ptr.readlines():
                self.number_of_instructions += 1
                if ("e3 | M |" in line) or ("63 | M |" in line):
                    branchInstrsFile_ptr.writelines(line)
                    i = i+1
            self.number_of_brnch_instr = i
            print(str(self.number_of_instructions) + " instructions
detected")
            print(str(self.number_of_brnch_instr) + " Branch
instructions were detected\n")
            return self.number_of_brnch_instr

    def distinguish_Taken_NotTaken_branches(self):
         with open(self.logPath, "r") as logFile_ptr,
open(takenBranchPath, "w") as logTakenBranch_ptr,
open(notTakenBranchPAth, "w") as logNotTakenBranch_ptr :
            second_line = logFile_ptr.readline()
            second_line = logFile_ptr.readline()
            instr_strtIndex = second_line.index("      PC |")
            instr_endIndex  = instr_strtIndex + 8

            self.instr_strtIndex = instr_strtIndex
            self.instr_endIndex  = instr_endIndex

            cycle_strtIndex = second_line.index("    CYCLE |")
            cycle_endIndex  = second_line.index(" |  ORDER |")

            self.cycle_strtIndex = cycle_strtIndex
            self.cycle_endIndex  = cycle_endIndex

            order_strtIndex = second_line.index("  ORDER |")
            order_endIndex  = second_line.index(" |       PC |")

            self.order_strtIndex = order_strtIndex
            self.order_endIndex  = order_endIndex

            logNotTakenBranch_ptr.writelines(second_line)
            logTakenBranch_ptr.writelines(second_line)

            branch_line = ""
            i = 0
            last_read_line = second_line
            for line in logFile_ptr.readlines():
                last_read_line = line
                if i < 2:
                    if i == 1:
                        first_instr_finish_clkCyc =
int(line[cycle_strtIndex:cycle_endIndex])
                    i += 1

                if branch_line != "":
                    instr_name_strtIndex = line.index(".c")

                    tmp_str = line[instr_name_strtIndex:len(line)]

                    instr_name_strtIndex = tmp_str.index("-")+2
                    instr_name_endIndex  = tmp_str.index("x")

                    instr_name =
tmp_str[instr_name_strtIndex:instr_name_endIndex]

                    diff_currPC_prevPC =
int(line[instr_strtIndex:instr_endIndex], 16) -
int(branch_line[instr_strtIndex:instr_endIndex], 16)

                    cycle_diff =
int(line[cycle_strtIndex:cycle_endIndex]) -
int(branch_line[cycle_strtIndex:cycle_endIndex])
                    order = int(line[order_strtIndex:order_endIndex])

                    if diff_currPC_prevPC == 4: # not taken

                        if(cycle_diff in
self.notTakenBranchDelay_dict):
                            self.notTakenBranchDelay_dict[cycle_diff]
+= 1

self.notTakenBranchOrder_dict[cycle_diff].append(order)
self.notTakenBranch_instr_name_dict[cycle_diff].add(instr_name)
                        else:
                            self.notTakenBranchDelay_dict[cycle_diff]
= 1
                            self.notTakenBranchOrder_dict[cycle_diff]
= []

self.notTakenBranchOrder_dict[cycle_diff].append(order)


self.notTakenBranch_instr_name_dict[cycle_diff].add(instr_name)

                        self.number_not_taken = self.number_not_taken
+ 1
                        logNotTakenBranch_ptr.writelines(branch_line)
                        logNotTakenBranch_ptr.writelines(line)
                        logNotTakenBranch_ptr.writelines(second_line)
                    else: # Taken

                        if(cycle_diff in self.takenBranchDelay_dict):
                            self.takenBranchDelay_dict[cycle_diff] +=
1

self.takenBranchOrder_dict[cycle_diff].append(order)


self.takenBranch_instr_name_dict[cycle_diff].add(instr_name)
                        else:
                            self.takenBranchDelay_dict[cycle_diff] = 1
                            self.takenBranchOrder_dict[cycle_diff] =
[]

self.takenBranchOrder_dict[cycle_diff].append(order)


self.takenBranch_instr_name_dict[cycle_diff].add(instr_name)

                        self.number_taken = self.number_taken + 1
                        logTakenBranch_ptr.writelines(branch_line)
                        logTakenBranch_ptr.writelines(line)
                        logTakenBranch_ptr.writelines(second_line)

                if ("e3 | M |" in line) or ("63 | M |" in line):
                    branch_line = line
                else:
                     branch_line = ""

            self.number_of_mispredict_pre_prediction =
self.number_taken
            self.mispred_clk_cyc_penalty_pre_prediction =
self.number_of_mispredict_pre_prediction * 2

            last_instr_finish_clkCyc =
int(last_read_line[cycle_strtIndex:cycle_endIndex])
            self.number_of_clkCyc_pre_prediction =
last_instr_finish_clkCyc - first_instr_finish_clkCyc + 1

    def getPC(self, line): #TEST it
        return int(line[self.instr_strtIndex:self.instr_endIndex], 16)
    def is_branch(self, line): #TEST it
        if ("e3 | M |" in line) or ("63 | M |" in line):
            return True
        else:
            return False

    def add_to_BTB(self, pc, line):
        ind = line.index("3 | M |")
        instr = int(line[ind-7:ind+1], 16)

        nxt_pc = (instr >> 8) & 15 # nxt_pc = imm[4:1]

        tmp = (instr >> 25) & 63 # tmp = imm[10:5]

        nxt_pc = nxt_pc + (tmp << 4) # nxt_pc = imm[10:1]

        tmp = (instr >> 7) & 1  # tmp = imm[11]

        nxt_pc = nxt_pc + (tmp << 10)  # nxt_pc = imm[11:1]

        tmp = (instr >> 31) & 1  # tmp = imm[12]

        nxt_pc = nxt_pc + (tmp << 11)  # nxt_pc = imm[12:1]

        if tmp == 1: # offset is negative
            nxt_pc = 0 - nxt_pc

        nxt_pc += pc

        num_BTB_entries = len(self.BTB.keys())
        # print(self.access_hist_list)
        # print(1)
        if(num_BTB_entries < self.BTB_size): # BTB is not full
            self.BTB[pc] = [nxt_pc, 3]
            self.access_hist_list.append(pc)
            # print(self.access_hist_list)
            # print(2)
        else:
            self.BTB.pop(self.access_hist_list[0])
            self.BTB[pc] = [nxt_pc, 3]
            self.access_hist_list.remove(self.access_hist_list[0])
            self.access_hist_list.append(pc)

    def update_BTB(self, pc, line):
        self.add_to_BTB(self, pc, line)

    def prediction_algorithm(self, pc, current_line, next_line):

        prediction = self.BTB[pc][1] > 1 # True -> branch is predicted
to be taken

        diff_currPC_prevPC =
int(next_line[self.instr_strtIndex:self.instr_endIndex], 16) \
                                            -
int(current_line[self.instr_strtIndex:self.instr_endIndex], 16)

        was_taken = diff_currPC_prevPC != 4

        if prediction:      # predicted to be take
            if was_taken:   # correctly predicted
```

```
                self.prediction_reward -= 2
            else:           # incorrectly predicted
                self.prediction_reward += 2

        if prediction == was_taken:  # correct prediction
            self.num_correct_prediction += 1
            if prediction:
                self.BTB[pc][1] = min(3, self.BTB[pc][1] + 1)
            else:
                self.BTB[pc][1] = max(0, self.BTB[pc][1] - 1)
        else:                        # incorrect prediction
            self.num_incorrect_prediction += 1
            if prediction:
                self.BTB[pc][1] = max(0, self.BTB[pc][1] - 1)
            else:
                self.BTB[pc][1] = min(3, self.BTB[pc][1] + 1)

        self.access_hist_list.remove(pc)
        self.access_hist_list.append(pc)

    def branch_prediction_model(self):
        with open(self.logPath, "r") as logFile_ptr,
open(self.newLogPath, "w") as newLogFile_ptr:

            for i in range(3):
                first_3_lines = logFile_ptr.readline()
                newLogFile_ptr.writelines(first_3_lines)

            current_line = logFile_ptr.readline()

            first_instr_finish_clkCyc =
int(current_line[self.cycle_strtIndex:self.cycle_endIndex])

            for next_line in logFile_ptr.readlines():
                pc = self.getPC(current_line)

                cycle = int(current_line[self.cycle_strtIndex :
self.cycle_endIndex])

                cycle = cycle + self.prediction_reward

                cycle = str(cycle)

                current_line =
current_line.replace(current_line[self.cycle_endIndex - len(cycle):
self.cycle_endIndex], str(cycle))

                newLogFile_ptr.writelines(current_line)

                if pc in self.BTB.keys(): # PC is     in BTB
                    self.prediction_algorithm(pc, current_line,
next_line)
                else:                     # PC is not in BTB
                    if self.is_branch(current_line): # if this instr
is new branch
                        self.add_to_BTB(pc, current_line)

                        diff_currPC_prevPC =
int(next_line[self.instr_strtIndex:self.instr_endIndex], 16) \
                                              -
int(current_line[self.instr_strtIndex:self.instr_endIndex], 16)
```

```
                        was_taken = diff_currPC_prevPC != 4

                        if was_taken:
                            self.num_incorrect_prediction += 1
                        else:
                            self.num_correct_prediction += 1


                current_line = next_line

            cycle = int(current_line[self.cycle_strtIndex:
self.cycle_endIndex])

            cycle = cycle + self.prediction_reward

            self.number_of_clkCyc_brnach_prediction = cycle -
first_instr_finish_clkCyc + 1
            # print(first_instr_finish_clkCyc)
            cycle = str(cycle)

            current_line =
current_line.replace(current_line[self.cycle_endIndex - len(cycle):
self.cycle_endIndex],
                                                        str(cycle))

            # current_line[self.cycle_endIndex - len(cycle):
self.cycle_endIndex] = cycle

            newLogFile_ptr.writelines(current_line)


    def generate_statistics(self):
        self.read_branch_instrs()
        self.distinguish_Taken_NotTaken_branches()

        x = [self.BTB_size]
        cpi_improvement_inProcent = []
        cpi_improvement = []
        clock_cycle_improvement = []
        clock_cycle_improvement1_inPrcnt = []

        prediction_reward = []
        num_correct_prediction = []
        num_incorrect_prediction = []

        for i in range(10):
            self.BTB = {}
            self.access_hist_list = []
            self.prediction_reward = 0
            self.num_correct_prediction = 0
            self.num_incorrect_prediction = 0

            self.branch_prediction_model()

            print("\n-------------------------------------------------------
--------------")
            print("STATISTICS BEFORE BRANCH PREDICTION\n")

            print("Number of Instructions: " +
str(self.number_of_instructions))
```

```
            print("Number of clock cycles: " +
str(self.number_of_clkCyc_pre_prediction))
            self.cpi_pre_prediction =
self.number_of_clkCyc_pre_prediction/self.number_of_instructions
            print("\nCPI without Branch Prediction: " +
str(self.cpi_pre_prediction))

            print("\nNumber of Branch Instructions: " +
str(self.number_of_brnch_instr))
            print("Number of CORRECTLY   predicted Branch
Instructions: " + str(self.number_not_taken))
            print("Number of INCORRECTLY predicted Branch
Instructions: " + str(self.number_of_mispredict_pre_prediction))
            print("\nNumber of branch misprediction penalty  (in
cycles): " + str(self.mispred_clk_cyc_penalty_pre_prediction))

            self.fraction_mispredict_branch_pre_prediction = 100 *
self.number_of_mispredict_pre_prediction/self.number_of_brnch_instr
            print("\nMispredicted branch instructions  (%): " +
str(self.fraction_mispredict_branch_pre_prediction))

            self.fraction_mispredict_penalty_cyc = 100 *
self.mispred_clk_cyc_penalty_pre_prediction/self.number_of_clkCyc_pre_
prediction
            print("Branch misprediction penalty      (%): " +
str(self.fraction_mispredict_penalty_cyc))
            print("---------------------------------------------------
------------")
            print("\n-------------------------------------------------
--------------")

            print("STATISTICS WITH PERFECT BRANCH PREDICTION")

            self.number_of_clkCyc_perfect_prediction =
self.number_of_clkCyc_pre_prediction -
self.mispred_clk_cyc_penalty_pre_prediction
            print("\nNumber of clock cycles with perfect prediction: "
+ str(self.number_of_clkCyc_perfect_prediction))

            improvement = 100 * (self.number_of_clkCyc_pre_prediction
-
self.number_of_clkCyc_perfect_prediction)/self.number_of_clkCyc_pre_pr
ediction
            print("Improvement compared to without prediction (%): " +
str(improvement))

            self.cpi_perfect_prediction =
self.number_of_clkCyc_perfect_prediction/self.number_of_instructions
            print("\nCPI with perfect prediction: " +
str(self.cpi_perfect_prediction))

            improvement = 100 * (self.cpi_pre_prediction -
self.cpi_perfect_prediction)/self.cpi_pre_prediction
            print("Improvement compared to without prediction (%): " +
str(improvement))

            print("---------------------------------------------------
------------")
            print("\n-------------------------------------------------
--------------")
```

```
            print("STATISTICS WITH 2-BIT BRANCH PREDICTION")

            print("Number of CORRECTLY   predicted Branch
Instructions: " + str(self.num_correct_prediction))
            print("Number of INCORRECTLY predicted Branch
Instructions: " + str(self.num_incorrect_prediction))

            print("\nNumber of clock cycles with 2-bit branch
prediction: " + str(self.number_of_clkCyc_brnach_prediction))


            improvement = 100 * (self.number_of_clkCyc_pre_prediction
-
self.number_of_clkCyc_brnach_prediction)/self.number_of_clkCyc_pre_pre
diction
            print("Improvement compared to without prediction (%): " +
str(improvement))

            clock_cycle_improvement1_inPrcnt.append(improvement)

            self.cpi_brnach_prediction =
self.number_of_clkCyc_brnach_prediction/self.number_of_instructions
            print("\nCPI with 2-BIT BRANCH PREDICTION: " +
str(self.cpi_brnach_prediction))

            improvement = 100 * (self.cpi_pre_prediction -
self.cpi_brnach_prediction)/self.cpi_pre_prediction
            print("Improvement compared to without prediction (%): " +
str(improvement))

            print("-------------------------------------------------
------------")
            print("\n-----------------------------------------------
--------------")

            # print(self.BTB_size)
            self.BTB_size += 10
            x.append(self.BTB_size)
            cpi_improvement_inProcent.append(improvement)
            cpi_improvement.append(self.cpi_brnach_prediction)

clock_cycle_improvement.append(self.number_of_clkCyc_brnach_prediction
)

            prediction_reward.append(self.prediction_reward)
            num_correct_prediction.append(self.num_correct_prediction)

num_incorrect_prediction.append(self.num_incorrect_prediction)
            print(i)

        x.remove(x[len(x)-1])

        newX = [i * 4 for i in x]
        x = []
        x = newX

        maxImprovement = [6.667609751497147 for i in x]

        cpi_improvement_inProcent.reverse()
        plt.plot(x, maxImprovement, color = 'g', label = 'CPI
improvement with perfect prediction (%)')
```

```
        plt.plot(x, cpi_improvement_inProcent, color='r', label='CPI 
improvement with 2-bit prediction (%)')
        plt.xlabel('BTB size (Bytes)')
        plt.ylabel('CPI improvement (%)')
        plt.title('CPI improvement')
        plt.legend()
        plt.show()

        maxCPI = [self.cpi_perfect_prediction for i in x]
        without = [self.cpi_pre_prediction for i in x]
        cpi_improvement.reverse()
        plt.plot(x, without, color='r', label='CPI WITHOUT 
prediction')
        plt.plot(x, cpi_improvement, color='b', label='CPI with 2-bit 
prediction')
        plt.plot(x, maxCPI, color = 'g', label = 'CPI with perfect 
prediction')
        plt.xlabel('BTB size (Bytes)')
        plt.ylabel('CPI with branch predictor')
        plt.title('CPI with branch predictor')
        plt.legend()
        plt.show()

        tmp = []
        for i in range(len(x)):
            tmp.append(self.number_of_clkCyc_pre_prediction)

        perfectPredictor = [self.number_of_clkCyc_perfect_prediction 
for i in x]
        clock_cycle_improvement.reverse()
        plt.plot(x,tmp, color = 'g', label = 'Total number of clk 
cycle WITHOUT predictor')
        plt.plot(x, clock_cycle_improvement, color='r', label='Total 
number of clk cycle with predictor')
        plt.plot(x,perfectPredictor, color = 'b', label = 'Total 
number of clk cycle with perfect predictor')
        plt.xlabel('BTB size (Bytes)')
        plt.ylabel('Number of Clk Cycles')
        plt.title('Total Number of Clk Cycles')
        plt.legend()
        plt.show()

        # plt.plot(x, clock_cycle_improvement1_inPrcnt, color='r')
        # plt.xlabel('x - BTB size (Bytes)')
        # plt.ylabel('y - Number of Clk Cycles improvement (%)')
        # plt.title('Total Number of Clk Cycles improvement (%)')
        # plt.show()
        #

        # prediction_reward.reverse()
        # plt.plot(x, prediction_reward, color='r')
        # plt.xlabel('x - BTB size (Bytes)')
        # plt.ylabel('y - prediction reward (clk cycles)')
        # plt.title('Branch prediction reward')
        # plt.show()

        # plt.plot(x, num_correct_prediction, color='r', label = 
'number of correct predicitons')
        # plt.plot(x, num_incorrect_prediction, color='g', label = 
'Number of incorrect predicitons')
        # plt.xlabel('x - BTB size (Bytes)')
        # plt.ylabel('y - number of predictions')
        # plt.title('Branch predictor accuracy')
        # plt.legend()
        # plt.show()




obj = model(logPath,branchInstrsPath, newLogPath, size)

obj.generate_statistics()

# num = "   5 "

# print(int(num))
```