Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Giorgi Solomnishvili

Branch Prediction - a Low Power Solution to Performance Optimization

Master's thesis in Embedded Computing System
Supervisor: Thomas Tybell
Co-supervisor: Øystein Knauserud
June 2023

NTNU
Norwegian University of
Science and Technology

Giorgi Solomnishvili

Branch Prediction - a Low Power Solution to Performance Optimization

Master's thesis in Embedded Computing System
Supervisor: Thomas Tybell
Co-supervisor: Øystein Knauserud
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Giorgi Solomnishvili

Branch Prediction - a Low Power Solution to Performance Optimization

Master's thesis in Electronic Systems Design
Supervisor: Tybell, Thomas
Co-supervisor: Knauserud, Øystein
19 June, 2023

Norwegian University of Science and Technology

NTNU

ABSTRACT

The master thesis is carried out in collaboration with Silicon Labs. The main goal of the thesis is to increase the speed of the CV32E40X processor, developed by openHW group, without increasing the area usage and power consumption by more than 20%.

The metric used for speed evaluation is Instruction Per Clock Cycle (IPC); for the area usage - net area, cell area, and total area; for the power consumption - Watts (W).

The CV32E40X is an open-source RISC-V pipelined processor with four pipeline stages. The core can execute compressed (16-bit long) and uncompressed (32-bit long) instructions. The baseline IPC value is 0.46; the baseline net area - 5874.265, the baseline cell area - 16704.66, the total area - 22578.927; the baseline total power - 270 MicroW.

The thesis addresses how implementing a 2-bit branch prediction unit can improve the core speed and cost of the improvement in terms of power consumption and area usage. The results revealed that the 2-bit branch predictor with a cache that has 8 cachelines increases IPC by 8.7%, at the cost of 8.9% net area, 14.7% cell area, 13.1% of total area, and 10.4% total power increase.

PREFACE

First of All, I would like to thank NTNU for giving me the opportunity to study for a Master's Degree. Secondly, I am grateful for all the professors who tirelessly put their time and effort into giving me high-quality education. I want to sing out my professor and supervisor, Thomas Tybell. He was always willing to advise me on how to improve my writing and make my thesis more presentable and professional. I also want to express my gratitude towards Mr. Øystein Knauserud, my external supervisor from Silicon Labs. His help in understanding the RISC-V core was crucial for completing my work. I will always be grateful for Pr. Tybell and Mr. Knauserud. Finally, I would like to thank my family and friends, without whom I would have given up a long time ago.

CONTENTS

Abstract **i**

Preface **ii**

Contents **iv**

List of Figures **iv**

List of Tables **vi**

Abbreviations **viii**

1 Introduction **1**
1.1 Motivation 1
1.2 Main Goal and Objectives 1
1.3 Report Structure 2

2 State of Art **3**
2.1 RISC-V and Pipelined Design 5
2.2 Branch Instruction 6
2.2.1 Branch Effects on Pipelined Execution 7
2.3 Branch Prediction 8
2.3.1 Static Prediction 9
2.3.2 Dynamic Prediction 9
2.4 Comparison of Inteded 2-bit Predictor with Other Predictor Units . 14

3 Methodology **15**
3.1 CV32E40X Core 15
3.1.1 Controller 16
3.1.2 Instruction Fetch Stage 20
3.1.3 Instruction Decode and Execute Stages 22
3.2 Description of Branch Prediction Model 22
3.3 Description of Intended Design 24
3.4 Implementation of Intended Design 25
3.4.1 CV32E40X_BTB_BTH 26
3.4.2 CV32E40X_IF_STAGE 28
3.4.3 CV32E40X_ID_STAGE and CV32E40X_EX_STAGE . . . 29

3.4.4 CV32E40X_CONTROLLER_FSM 30
3.5 Metrics and Baseline Performance of CV32E40X Core 32
3.5.1 Baseline Performance of CV32E40X Core 33
3.6 Descriptions of Tests 33

4 Results **35**
4.1 Speed Improvement 35
4.2 Drawbacks 40

5 Conclusions **43**
5.1 Future Work 43

References **45**

Appendices: **47**

A - Github repository **48**
.1 Structure of Repository 48
.1.1 Master 49
.1.2 Workplace 50
.2 Running Model 51
.3 Setting up Simulation, Simulating the Core, Analyzing Simulation Data 51
.3.1 Setting up Simulation 51
.3.2 Simulating the Core and Analyze Simulation Data 51
.3.3 Plot the Analyzed Simulation Data 52
.4 Setting up Synthesis, Synthesizing Core, Analyzing Synthesis Data . 52
.4.1 Setting up Synthesis 52
.4.2 Synthesizing Core and Analyzing Synthesis Data 54
.4.3 Plot the Analyzed Synthesis Data 54

LIST OF FIGURES

2.0.1 42 years of microprocessor trend data[4] 4
2.0.2 The power dissipated per cm^2 with respect to transistor channel length scaling[5] 5
2.1.1 Pipeline Execution 6
2.2.1 Branch instruction and possible execution sequences[11] 7
2.2.2 Mispredicted Branch instruction propagation through the pipeline . 8
2.3.1 State diagram for BHT entries of the 1-bit branch predictor[1] . . . 10
2.3.2 Block Diagram for 1-bit branch predictor[1] 10
2.3.3 1-bit predictor making 2 mispredictions in a row[1] 12
2.3.4 State diagram for BHT entries of the 2-bit branch predictor[1] . . . 12
2.3.5 Block Diagram for 2-bit branch predictor[1] 13

3.1.1 CV32E40X core[14] 15
3.1.2 CV32E40X-Controller 16
3.1.3 Handshake between pipeline stages[14] 17
3.1.4 Controller FSM Data Type[15] 18
3.1.5 Controller FSM - Functional State Decision Diagram 19
3.1.6 The Logic for Branch Taken Detection 20
3.1.7 Code for Branch Taking[15] 20
3.1.8 IF_ID_Pipeline Register (Compressed)[15] 21
3.1.9 PC Multiplexer 22
3.2.1 Log file generated by simulating CV32E40X core[1] 23
3.2.2 flow diagram for the method branch prediction model[1] 23
3.3.1 Block Diagram of Predictor Modules in the Pipeline 24
3.4.1 Defined Datatypes for cache_cmd, branch_target_mux and cache_line 25
3.4.2 Indexing Cacheline with PC from IF Stage 26
3.4.3 Updating Cacheline Indexed with PC from EX Stage 27
3.4.4 Truth Table for 2-bit Prediction Counter Increment and Decrement 28
3.4.5 Block Diagram for Misprediction Recovery in EX Stage 30
3.4.6 Signals Used by FSM for Prediction/REcovery 30
3.4.7 Block Diagram for Part of the FSM handling Branching, Jumping, Prediction, and Misprediction Recovery 31
3.5.1 Baseline Performance 33
3.6.1 Flow Diagram of Python Script for Simulation and Synthesis 34

4.1.1 Results of Implementing 2-bit Branch Predictor with 8 cachelines for BTB_BHT. Speed (IPC), Area, and Power Comparison with the Baseline Performance 35
4.1.2 Number of clk with respect to cache size. Plot 1 - y-axis is numbers, Plot 2 - y-axis is percentages 37
4.1.3 IPC with respect to cache size. Plot 3-4 - y-axis are IPC; Plot 5 - Y-axis are percentages 38
4.1.4 Number of predictions with respect to cache size. Plot 6 - number of predictions vs. cache size; Plot 7 - same as plot 6, Y-axis is percentages; Plot 8 - number of correct/incorrect predictions vs. cache size; Plot 9 - same as plot 8, Y-axis is percentages 39
4.1.5 Cache usage with respect to cache size. Plot 10 - number of used cachelines vs cache size; Plot 11 - number of used/unused cachelines, Y-axis is percentages; 40
4.2.1 Plot 12 - total, cell, net area vs. cache size; Plot 13 - Cell area and Baseline value of Cell area; Plot 14 - Net area and Baseline value of Net area; Plot 15 - Total area and Baseline value of Total area . 41
4.2.2 Plot 16 - total power, register power, a fraction of total power used by register vs. cache size; Plot 17 - a fraction used by register vs. cache size. Y-axis is percentages; Plot 18 - an increase of Total, Register power, and fraction of total power used by registers compared to the baseline values. Y-axis is percentages; 42

.1.1 Code Tree of CV32E40X Core with 2-bit Branch Predictor 49
.4.1 cv32e40x_BTB_BHT module: Lines 104-153 store btb statistics into btb_statistics.txt file. 52
.4.2 TCL script reading core modules. These lines can be found in MasterThesis/workplace/tcl /synth.tcl file 53

LIST OF TABLES

ABBREVIATIONS

- **ALU** - Arithmetic Logic Unit
- **BHT** - Branch History Table
- **BR** - Branch
- **BTB** - Branch Target Buffer
- **CISC** - Complex Instruction Set Computer
- **CLK** - Clock
- **CPI** - Clock Cycle Per Instruction
- **CPU** - Central Processing Unit
- **DIV** - Divider
- **EX** - Execution Stage
- **EXC** - Exception
- **FIFO** - First-In-First-Out
- **FSM** - Finite State Machine
- **ID** - Instruction Decode Stage
- **IF** - Instruction Fetch Stage
- **IPC** - Instruction per Clock Cycle
- **ISA** - Instruction Set Architecture
- **JMP** - Jump
- **LSU** - Load-Store Unit
- **MI** - Maskable Interrupt
- **MUL** - Multiplier
- **NMI** - Non-Maskable Interrupt

- **PC** - Program Counter
- **RISC** - Reduced Instruction Set Computer
- **SPIN** - Special Instruction
- **WB** - Writeback Stage

CHAPTER

ONE

INTRODUCTION

Chapter 1 explains the motivation for developing a high-performance Central Processing Unit (CPU), states the main goal and objectives of the master's project, and describes the report structure.

1.1 Motivation

Nowadays, embedded systems are part of most modern devices. For example, one cannot design Internet of Things (IoT) networks without utilizing a significant number of embedded systems. The embedded System itself can be a small piece in much larger systems. Besides the sensors and actuators, an important part of the embedded System is its processing unit. These CPUs analyze the information captured by the sensors. Hence, developing a high-speed processing unit is essential for the fast performance of embedded systems.

The thesis is based on the results of the project carried out as part of the course TFE4580: Electronic Systems Design and Innovation, Specialization Project. The report [1], provided in the references, describes the project methodology and findings in detail.

1.2 Main Goal and Objectives

The main goal of the master thesis is to increase the speed of the CV32E40X RISC-V core by implementing a 2-bit branch predictor unit without increasing area and power by more than 20%. The metric for speed evaluation is the instruction per clock cycle (IPC). The area is measured in terms of cell area, cell count, and total area. Watt is used as a measure of power.

To attain the main goal, several objectives needed to be accomplished. The student needed to fulfill the following tasks.

- Familiarize yourself with the CV32E40X core.
- Explore different branch prediction strategies.

- Design a block diagram for the branch prediction unit.
- Implement the unit in the code base of the CV32E40X core.

The thesis addresses how implementing a 2-bit branch prediction unit can improve the core speed and cost of the improvement in terms of power consumption and area usage.

1.3 Report Structure

The report consists of five chapters:

1. **Introduction**
2. **State of Art**

   Chapter 2 explains the need for a faster CPU with low power cost. It also describes processing architecture - Reduced Instruction Set Architecture (RISCV), and innovations such as pipelined design that allow modern-day CPUs to have high throughput.

   The chapter also discusses branch instruction effects on pipeline execution and introduces branch prediction techniques.

3. **Methodology**

   Chapter 3 introduces CV32E40X RISC-V Core. Describes the block diagram of the predictor unit and explain where each part of the diagram fits in the core.

   Finally, the chapter states the core's baseline performance and elaborates on how the predictor is implemented, the tests performed on the core, and the measures used.

4. **Results**

   Chapter 4 analyzes information extracted from the tests, states speed increase, measured in instruction per clock cycle (IPC), and elaborates on the cost of performance improvement in terms of power consumption and area usage.

5. **Conclusion**

   Chapter 5 summarizes the findings attained for this master thesis, concludes with the effectiveness of the branch prediction unit on performance improvement, and suggests future work.

CHAPTER

TWO

STATE OF ART

Chapter 2 explains the need for faster CPUs, states power limitations, and discusses RISC-V architecture and pipelined design. The chapter also analyzes the effect branch instruction has on pipelined execution and presents branch prediction as a low-power solution.

One reason that motivates the development of fast CPUs for embedded systems is data deluge. The data deluge represents a situation where generated data vastly surpasses the computing system's capacity to analyze it[2]. In the case of embedded system computing, data deluge manifests itself in the microprocessor's inability to manage all the data obtained by the sensors[1]. Embedded systems are estimated to compute 1000 terabytes of information every minute, equivalent to reading 550 million books per minute [2]. Hence, there is a gap between generated and analyzed data, which can be coped with by designing faster CPUs for embedded systems.

Up to 2010, Moore's law was the main driver of the performance increase. According to Moore's law, the number of transistors on a chip doubles every 18 months[3]. The design technology improvements and scaling down transistors made it possible for the digital system designers to place a large number of transistors on the chip, which resulted in the increased computing capability[1].

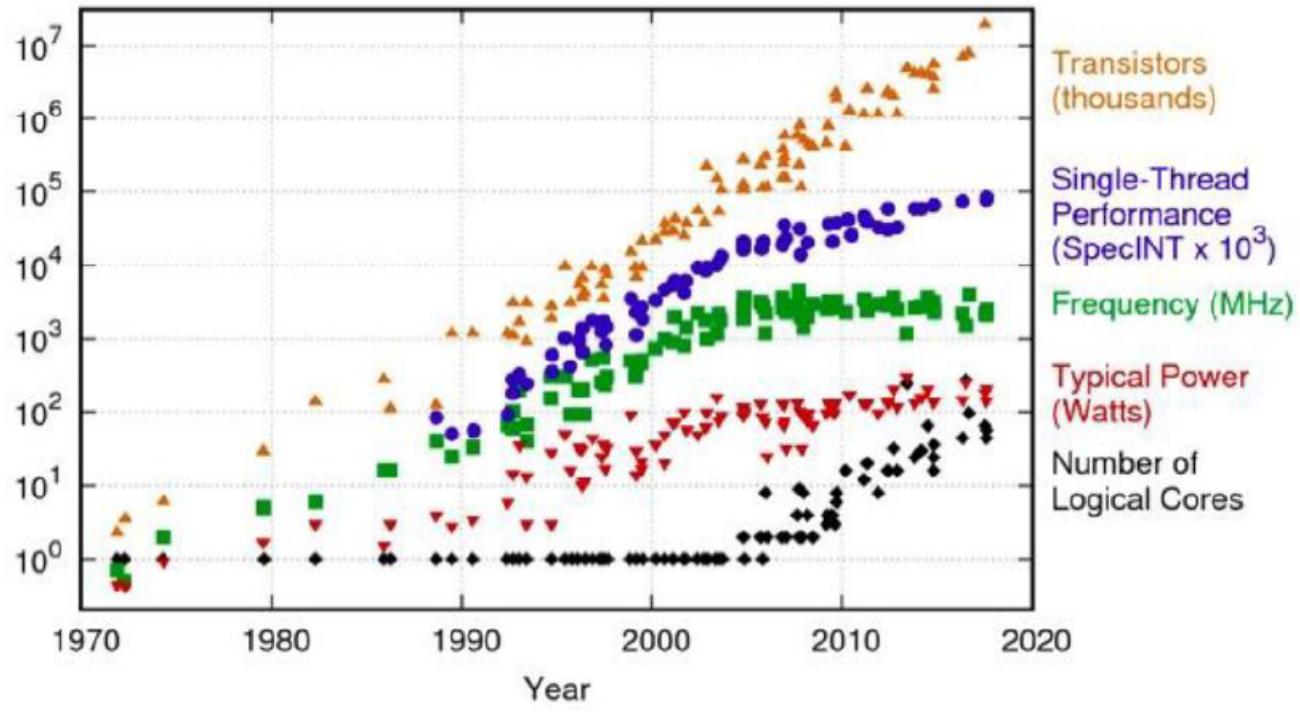


Figure 2.0.1: 42 years of microprocessor trend data[4]

However, figure 2.0.1 reveals that since 2010 the linear increase in the number of transistors on chip (brown triangles) no longer leads to a significant increase in operating frequency (green rectangles)[4]. One can observe that the frequency curve has saturated since 2010.

Unlike frequency, the increase in the number of transistors on the chip results in increased heat generated by power dissipation[5]. Figure 2.0.2 presents dissipated power per cm^2 for the Pentium processor family with respect to the transistor channel length. According to the curve presented in the figure, if Intel had not discontinued the production of the Pentium family, Pentium V would generate more power than a nuclear reactor, and further improvements would produce as much heat as a space rocket does during the launch[5].

The performance can be increased by modifying CPU architecture to allow engineers to run the design at a higher frequency[1]. However, the increased operating frequency increases the power consumption[1].

$$P = \alpha C V dd^2 f \tag{2.1}$$

According to 2.1, there is a direct proportional dependency between consumed power switching capacitance (C), supply voltage (Vdd), and operating frequency (f)[6]. Every attempt at increasing frequency results in an increase in consumed power[1].

Therefore, one of the ways to reduce the gap between the rate of data generation and the capability to analyze the data is increasing the performance of microprocessors used in embedded systems. However, the performance increase is limited by the chip's ability to dissipate generated heat and power, and the designers need to utilize clever design techniques such as creating fast processing architecture, like RISC-V, and improving the existing architecture (pipelining and branch prediction) to run the processing unit faster[7].

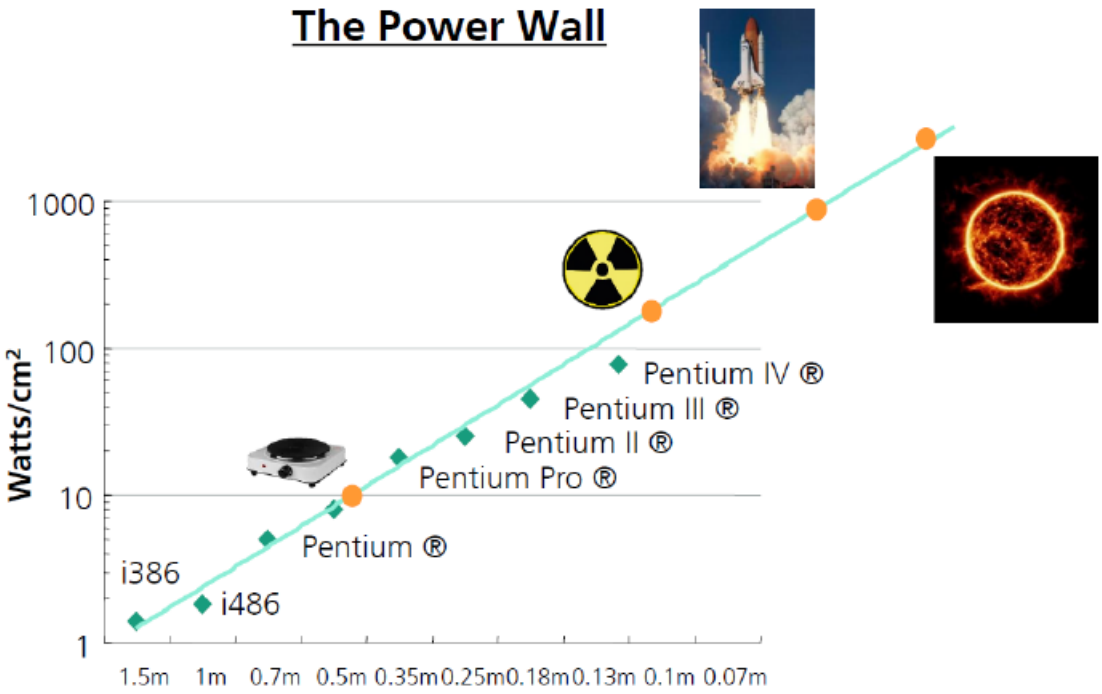


Figure 2.0.2: The power dissipated per cm^2 with respect to transistor channel length scaling[5]

2.1 RISC-V and Pipelined Design

Instruction Set Architecture (ISA) specifies all different instructions that are available for the programmer. RISC-V is an example of ISA developed by the University of Berkeley. RISC-V is an open-source architecture designed to be free and openly adoptable by the industry[8].

Computer architectures before the RISC family were complex instruction set architectures (CISC)[9]. CISC instructions can vary in length, and operands can be fetched from register files and memory. Such design techniques made programs written in CISC architectures compact and powerful[9]. On the other hand, RISC-V consists of less instruction than CISC, and all the instructions are the same length. Furthermore, RISC is load-store architecture, meaning operands are taken from registers[8]. Therefore, RISC architectures are easier to pipeline than CISC architectures[9].

Pipelining is an implementation technique such that executions of several instructions are overlapped[10]. The idea is to divide an instruction execution into several stages operating concurrently, providing that enough resources are available. The most common pipeline stages are instruction fetch (IF), instruction decode(ID), execution (EX), and write-back (WB) stages[10].

- **IF -** Instruction is fetched from memory, and the address of the next instruction is calculated[10].
- **ID -** Instruction is decoded, the branch target address is calculated, and operands are read from the register file[10].
- **EX -** This stage executes the operation or calculates the memory address for load-store instructions[10].
- **WB -** This stage accesses data memory or writes operation execution results in the register file[10]

Figure 2.1.1 demonstrates instruction execution in the pipeline. At the beginning (T0), all pipeline stages are empty. At time point T1, the first instruction (instr0) enters the IF stage. At T2, as instr0 is passed down the ID stage for decoding, instr1 enters into the IF stage. The pipelining does not reduce the time required for executing a single instruction. However, since multiple instructions are executed concurrently, pipelining increases the throughput, which increases processor performance. Different instructions can have different effects on pipeline execution. The upcoming section introduces branch instruction and elaborates on its effect on the pipeline.

Time/Stage	Instruction Fetch (IF)	Instruction Decode (ID)	Instruction Execute (EX)	Writeback (WB)
T0	empty	empty	empty	empty
T1	Instr0	empty	empty	empty
T2	Instr1	Instr0	empty	empty
T3	Instr2	Instr1	Instr0	empty
T4	Instr3	Instr2	Instr1	Instr0
T5	Instr4	Instr3	Instr2	Instr1
T6	Instr5	Instr4	Instr3	Instr2
T7	Instr6	Instr5	Instr4	Instr3

Figure 2.1.1: Pipeline Execution

2.2 Branch Instruction

One can understand how branch instruction works by analyzing 2.2.1. The left half of the figure displays instructions as they can be represented in instruction memory, and the right half portrays all possible execution sequence[11]. The program counter (PC) is a register that stores the address from which the currently executed instruction is fetched, and it is incremented after the instruction fetch[10].

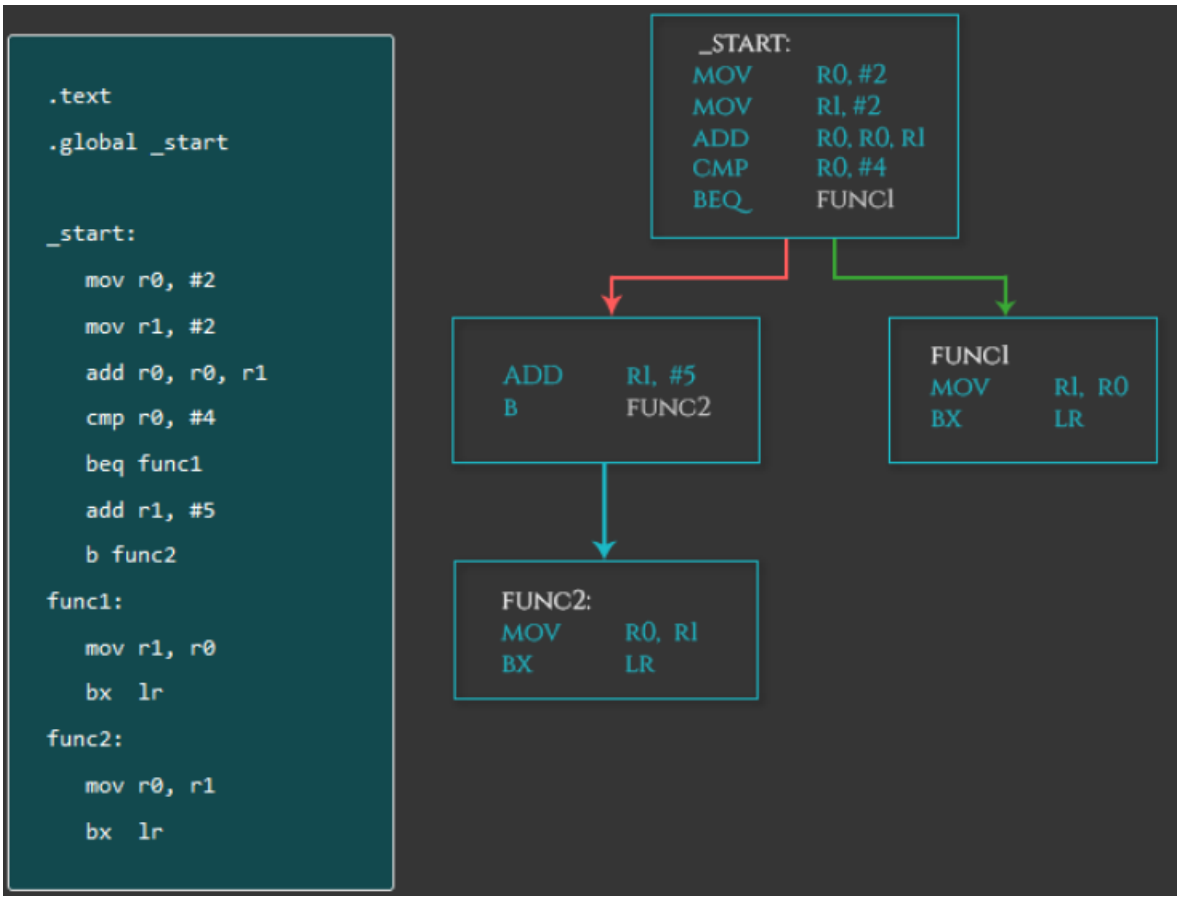


Figure 2.2.1: Branch instruction and possible execution sequences[11]

Branch instructions are used to deviate from consecutive instruction execution sequences. Branch instruction contains conditions and offsets to the branch target address[10]. If the condition is evaluated to be true, the instruction executed after the branch is fetched from the address is calculated by adding the offset to the PC[10]. For example, the first four instructions from figure 2.2.1 are executed consecutively. If the condition of instruction - BEQ FUNC1, is true, the execution jumps to FUNC1, otherwise, the instruction is written after the branch is executed.

2.2.1 Branch Effects on Pipelined Execution

Before the branch instruction reaches the EX stage, the pipelined CPU will not know whether the branch needs to be taken, and the processor will fetch instructions that are consecutive to the branch instruction. Hence, two consecutive instructions are in IF and ID stages when the branch reaches the EX stage. If the branch condition is evaluated to be true, the branch needs to be taken, and those two instructions should not be part of the program execution. Therefore, instructions in the ID and IF stages should be flushed away to avoid distorting the instruction execution order[8]. The fetching of two instructions after the branch that needs to be flushed away is called branch misprediction and is represented in figure 2.2.2.

At time-point T4, the branch instruction is fetched from the instruction memory.

Let's assume that if the branch is taken, it jumps over instr 4 and instr 5. However, since the CPU does not know the outcome of the branch condition at the IF stage, instr4 and instr5 are still fetched and propagated through the pipeline. In order to recover the execution order, distorted by the branch misprediction, the register file and data memory will not be updated when two unnecessary instructions reach the WB stage. Thus, branch misprediction delays the program execution by 2 clock cycles. Two cycles might not sound like an impressive delay. However, if the program has to iterate a loop 1000 times, the processor will mispredict 999 branch instructions, wasting 2000 cycles on the misprediction recovery[1]. In addition to imposing 2 cycle penalty, misprediction also wastes power on fetching and decoding instructions that will be flushed through the pipeline[1].

One of the methods to reduce the cycle penalty imposed by branch misprediction is to have a branch prediction unit in the design[1]. The following section introduces branch prediction and presents several algorithms for it.

Time/Stage	Instruction Fetch (IF)	Instruction Decode (ID)	Instruction Execute (EX)	Writeback (WB)
T0	empty	empty	empty	empty
T1	Instr0	empty	empty	empty
T2	Instr1	Instr0	empty	empty
T3	Instr2	Instr1	Instr0	empty
T4	Branch	Instr2	Instr1	Instr0
T5	Instr4	Branch	Instr2	Instr1
T6	Instr5	Instr4	Branch	Instr2
T7	Instr6	Instr5	Instr4	Branch
T8	Instr7	Instr6	Instr5	Instr4
T9	Instr8	Instr7	Instr6	Instr5
T10	Instr9	Instr8	Instr7	Instr6

Figure 2.2.2: Mispredicted Branch instruction propagation through the pipeline

2.3 Branch Prediction

Branch prediction is a design technique that allows the CPU to predict the outcome of the branch instruction condition at as early as the IF stage and fetch the next instruction based on the prediction[8]. Cleverly implementing a branch prediction unit can increase the speed of the CPU[1]. Since the performance improvement is not caused by doubling the number of transistors on the chip or increasing operating frequency, the branch prediction does not increase consumed power significantly[1].

There are many prediction algorithms, and they can be divided into two groups: Static and Dynamic prediction algorithms.

2.3.1 Static Prediction

Static prediction algorithms do not change their prediction algorithms at runtime[12]. An example of a static prediction algorithm is backward prediction. The branch is predicted to be taken if the branch target address is less than the address of the branch instructions. In other words, if the program jumps backward, the branch is predicted to be taken[12]. Another example of a static prediction algorithm is always not taken. As the name indicates, the predictor assumes that the branch is never taken[12].

The advantage of static prediction is that they are easy to implement in the design[1]. However, static prediction algorithms do not accurately distinguish themselves since they do not change the prediction strategy at runtime.

2.3.2 Dynamic Prediction

Unlike Static prediction, Dynamic Branch Prediction algorithms change the prediction strategies by analyzing the runtime information [1]. Such algorithms base their decision on the history of branch outcomes. Generally, two main parts of dynamic predictor units are a branch history table (BHT) and a branch target buffer (BTB).

As the name indicates, BHT stores the history of the previous evaluations of branch conditions. In most cases, BHT is designed as a memory indexed by the program counter corresponding to the branch instruction[10]. BTB, conversely, is a memory cache containing the target addresses of the branch instructions[10]. If implemented cleverly, dynamic branch predictors have the potential to decrease the number of mispredicted branches resulting in the decrease of branch misprediction penalty cycle and improving the CPU performance[1]. Although many dynamic branch prediction algorithms exist, the thesis focuses on n-bit predictors. If the readers want to explore other algorithms, they can refer to the specialization project report[1].

1-BIT PREDICTOR:

1-bit predictor contains BHT that utilizes a 1-bit counter to store the branch instruction history[1]. If the counter is 1, the branch outcome is predicted to be TAKEN. However, the prediction is NOT TAKEN if the counter's value is 0. Figure **??** portrays a state transition diagram that explains how the value stored in BHT changes based on the actual and predicted outcome of the branch instruction. If the prediction coincides with the actual outcome of the branch, the BHT value stays the same. On the contrary, if there is a mismatch between the predicted and actual outcome of the branch, the BHT value toggles.

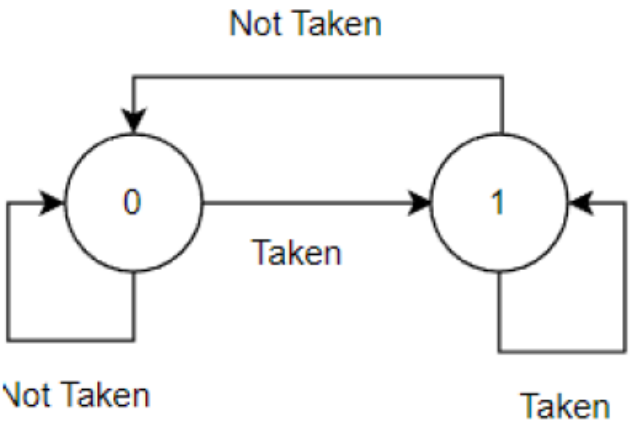


Figure 2.3.1: State diagram for BHT entries of the 1-bit branch predictor[1]

Figure 2.3.2 displays a block diagram implementing a 1-bit dynamic branch prediction algorithm. PC block represents a register containing the program counter. The current value of the PC indexes the block named BTB and implements a cache. If the PC points to a branch instruction that has already been executed by the CPU and stored in BTB, the BTB cache asserts the Hit signal. The cache also delivers the branch target address to the multiplexer. Similarly to BTB, BHT is indexed by the PC and outputs the value of the 1-bit counter associated with the branch instruction. If the prediction is to be TAKEN, the BHT outputs 1. Thus, in the case of a hit and TAKEN prediction, the address output by the BTB is assigned to the address of the next instruction (PCnxt). Otherwise, PCnxt attains the address of the consecutive instruction, which is calculated by adding 4 to the PC[1].

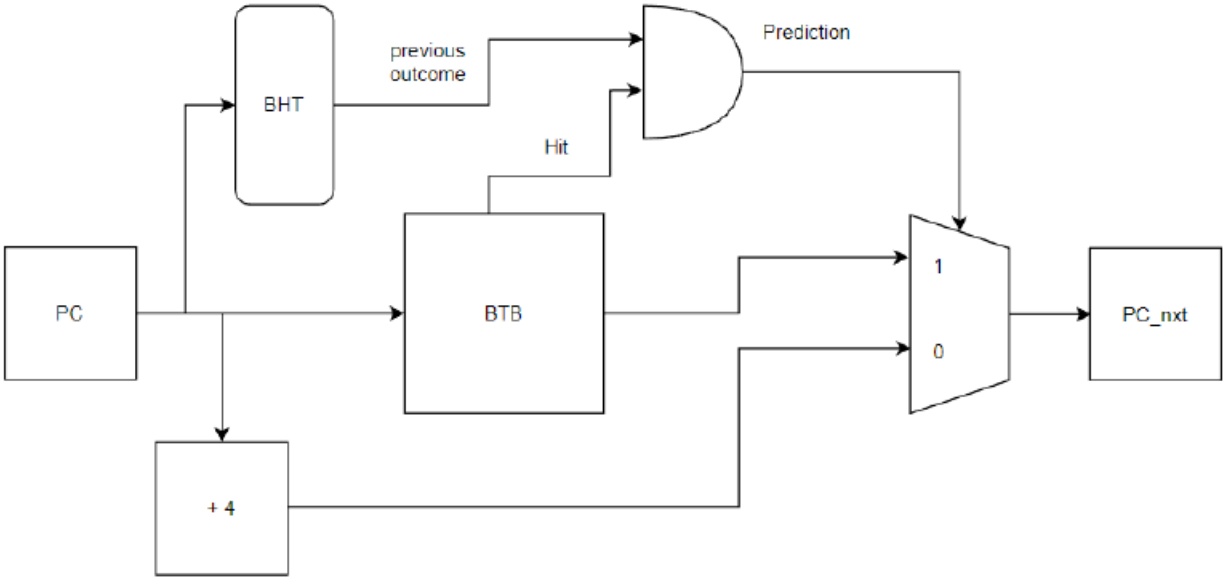


Figure 2.3.2: Block Diagram for 1-bit branch predictor[1]

1-bit predictor significantly improves performance if the block diagram is implemented in the IF pipeline stage[1]. At the beginning of the program execution,

both BHT and BTB are empty, and the predictor cannot make any predictions. After the CPU encounters branch instruction, it stores the branch target into BTB and sets the corresponding counter in BHT to 1. Thus, by default, it is assumed that the same branch instruction encountered the second time will be taken[1]. Hence, the 1-bit branch predictor cannot make predictions at the beginning of the program execution. It has a "warm-up time" and generates correct predictions after the core has encountered several branch instructions.

The 1-bit predictor can decrease the number of mispredicted branches. However, it is not a perfect predictor and does not eliminate mispredictions. As described above, the mispredicted branch instruction can only be discovered at the EX stage. At this point, the core must correct the execution order distorted by misprediction. Furthermore, the core also updates the BHT counter based on the correctness of the prediction. The misprediction causes the BHT counter bit to toggle[10].

2-BIT PREDICTOR:

2-bit predictor is an improved version of the 1-bit predictor. The difference between these 2 algorithms is that the former uses 2-bit counters to make the prediction while the latter has 1-bit counters as BHT entries [10]. As described above, a single misprediction is enough for a 1-bit predictor to change its prediction policy from TAKEN to NOT TAKEN. 2-bit predictor, on the other hand, needs two consecutive mispredictions to change the prediction policy[12].

Figure 2.3.3 demonstrates a drawback of changing prediction policy after just one misprediction. The figure demonstrates a case when the program execution encounters ten branch instructions. Letters T, TN, C, and I stand for TAKEN, NOT TAKEN, CORRECT, and INCORRECT. Branch number four is the first branch not taken in the sequence. Both 1-bit and 2-bit predictors mispredict the outcome of branch number 4. 1-bit predictor changes its prediction policy which causes the system to mispredict the outcome of branch number 5. However, since the 2-bit predictor requires two consecutive mispredictions to change the prediction policy, it correctly predicts the outcome of branch number 5. Although a 1-bit predictor can be used to the penalty imposed on the program execution by mispredictions, above discussed example proves that it is not as accurate as the 2-bit predictor. Based on the example, a predictor that changes prediction strategy based on one misprediction is 80% accurate. However, the predictor that requires two consecutive mispredictions to change its policy has 90% accuracy. This idea is utilized by a 2-bit branch predictor.

		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

Figure 2.3.3: 1-bit predictor making 2 mispredictions in a row[1]

The 2-bit predictor employs 2-bit counters as BHT entries. 2-bit counters can have one of the four possible values: 00, 01, 10, and 11. 00 and 01 correspond to NOT TAKEN prediction. The difference between these two values is that 00 represents a strong NOT TAKEN policy, while 01 corresponds to a weak NOT TAKEN policy. On the contrary, 10 and 11 correspond to weak TAKEN and strong TAKEN prediction policies. Figure 2.3.4 portrays the state transition diagram for BHT entries. The numbers inside the bubbles represent counter values, and TAKEN, NOT TAKEN writings at the arrows represent the branch outcomes. The counter value decreases for every branch that is not taken and increases for every branch[10]. Moreover, the counter update is performed with saturation arithmetic. There are minimum and maximum values in saturation arithmetic, and if the addition/subtraction causes overflow/underflow, the counter value is set to maximum/minimum value[13]. In the case of the predictor, the minimum value is 0, and the maximum value is 3. Furthermore, it can be observed that a single misprediction does not result in a change in prediction policy. It only reduces the "credibility" of the prediction. The only way to change the prediction policy is to encounter two consecutive branch mispredictions.[10].

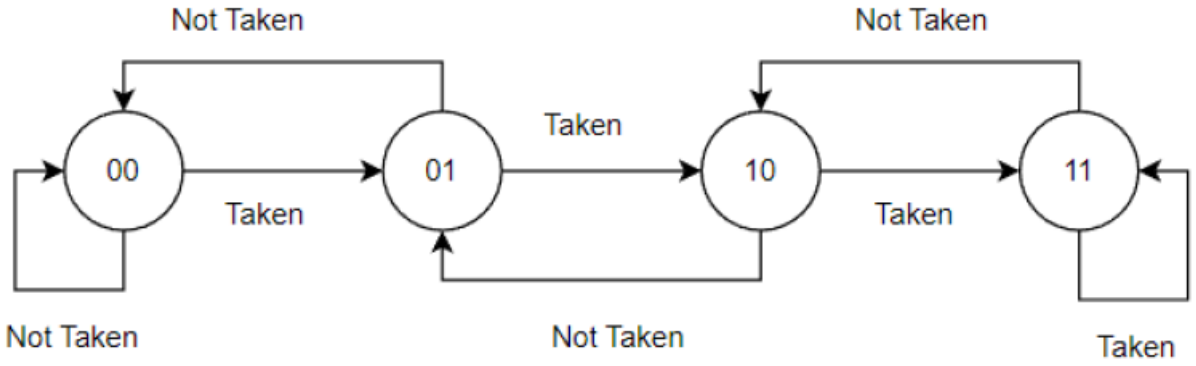


Figure 2.3.4: State diagram for BHT entries of the 2-bit branch predictor[1]

A 2-bit branch predictor can be implemented as a block diagram, displayed on 2.3.5[1]. PC box represents a register containing a program counter, BTB manifests a branch target buffer containing branch target addresses, and BHT contains branch predictions as 2-bit counters. Both BTB and BHT are indexed by PC. The predictor is implemented in the IF stage of the pipeline. At the beginning of the program execution, both BTB and BHT are empty. As the CPU encounters branch instructions, corresponding entries are registered in BHT and BTB for each branch [1].

Once the CPU fetches a branch instruction that has already been executed during the runtime, BTB asserts a signal named HIT and delivers the target address associated with the branch instruction currently present in the IF stage. In addition, BHT outputs the counter value associated with the encountered branch instruction. If the counter value is more than 1, the prediction is TAKEN. Otherwise, the prediction is NOT TAKEN. The prediction is determined by comparing the counter value with 1. If it is more than 1, PCnxt is assigned with an address forwarded by BTB (branch target address). Otherwise, PCnxt is assigned with PC + 4, representing the address of the instruction following the branch. The correctness of the prediction is checked when the branch reaches the EX stage. Based on the correctness of the prediction, BHT entries are modified as depicted in the state diagram (figure 2.3.4)[1].

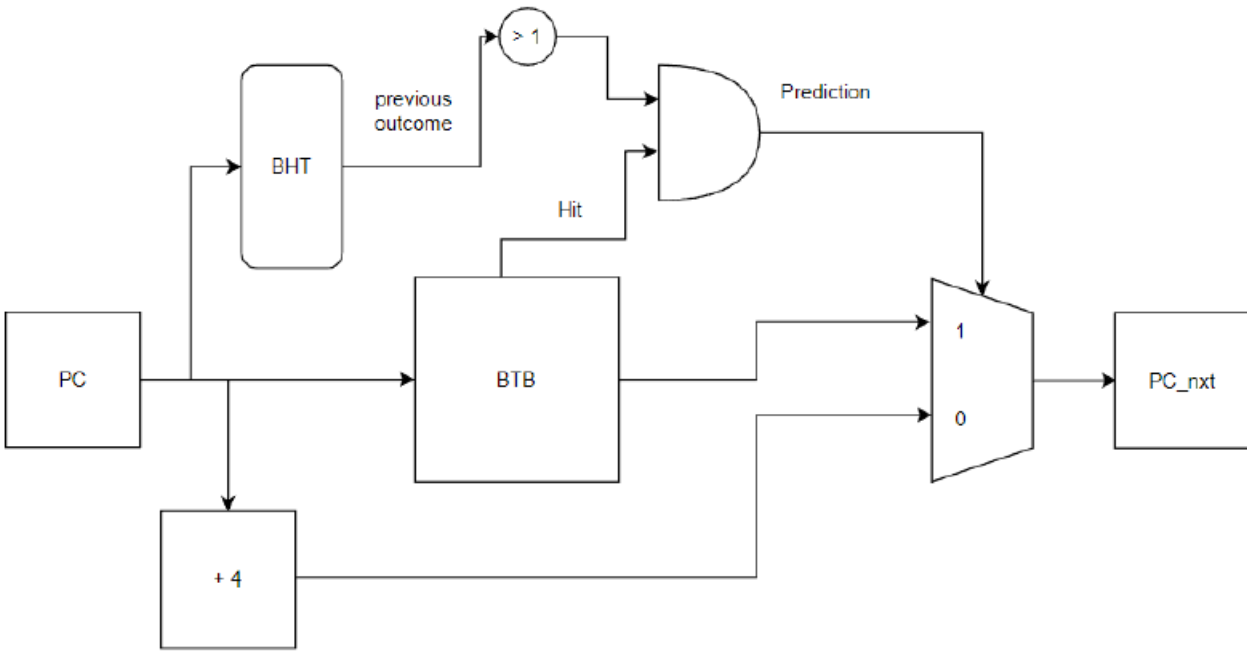


Figure 2.3.5: Block Diagram for 2-bit branch predictor[1]

2.4 Comparison of Inteded 2-bit Predictor with Other Predictor Units

The idea of increasing the processor's speed by implementing branch prediction has been explored by several researchers. An example of those researchers is Achyut Ray - "Branch prediction for a RISC-V processor core." Ray attempted to speed up the CV32E40X core by implementing backward prediction[7]. The type of predictor Ray uses is Static, and predictions are made in the ID stage[7]. According to Achyut, the backward predictor increases CPU performance by up to 4.25% if there are no stalls in the pipeline. Otherwise, the increase is only 0.97%[7]. Ray argues that the predictor did not generate a significant speed increase because the predictor was implemented at the ID stage, which forces the processor to flush the instruction in the IF stage even if the prediction is correct[7].

Another example of research (this work is based on) exploring how branch prediction increases CPU speed is "Low Power Solution to Performance Optimization" [1]. The article develops a model for a 2-bit branch predictor and claims that implementing it in the CV3240X core can increase the speed by 6.67%[1]. However, the modeled predictor only predicts uncompressed branch instructions, while the CV3240X core is capable of executing compressed instructions[1].

Throughout this thesis, the 2-bit branch predictor is a proposed solution to performance optimization. Unlike the backward predictor by Achyut, the 2-bit predictor will be implemented in the IF. Hence, the processor would not spend any clock cycle flushing instructions in case of the correct prediction. The 2-bit predictor implemented for this thesis is designed to identify and predict both compressed and uncompressed branch instructions.

The state of art chapter introduced the need for processor performance increase and identified power limitations that prevent the performance increase by doubling the number of transistors or increasing the operating frequency. The chapter also discussed pipelining and elaborated on branch instructions' effects on the pipeline. The following chapters will state the work done throughout the thesis and analyze the results.

CHAPTER

THREE

METHODOLOGY

Chapter 3 introduces the CV32E40X core, briefly presents the 2-bit branch predictor modeled for the specialization project[1], and describes the actual design for the 2-bit predictor and how it was implemented. The chapter also explains the metrics used for speed, area, and power measuring and states the baseline performance of the core. Furthermore, the chapter also describes the tests performed on the core with branch predictor.

3.1 CV32E40X Core

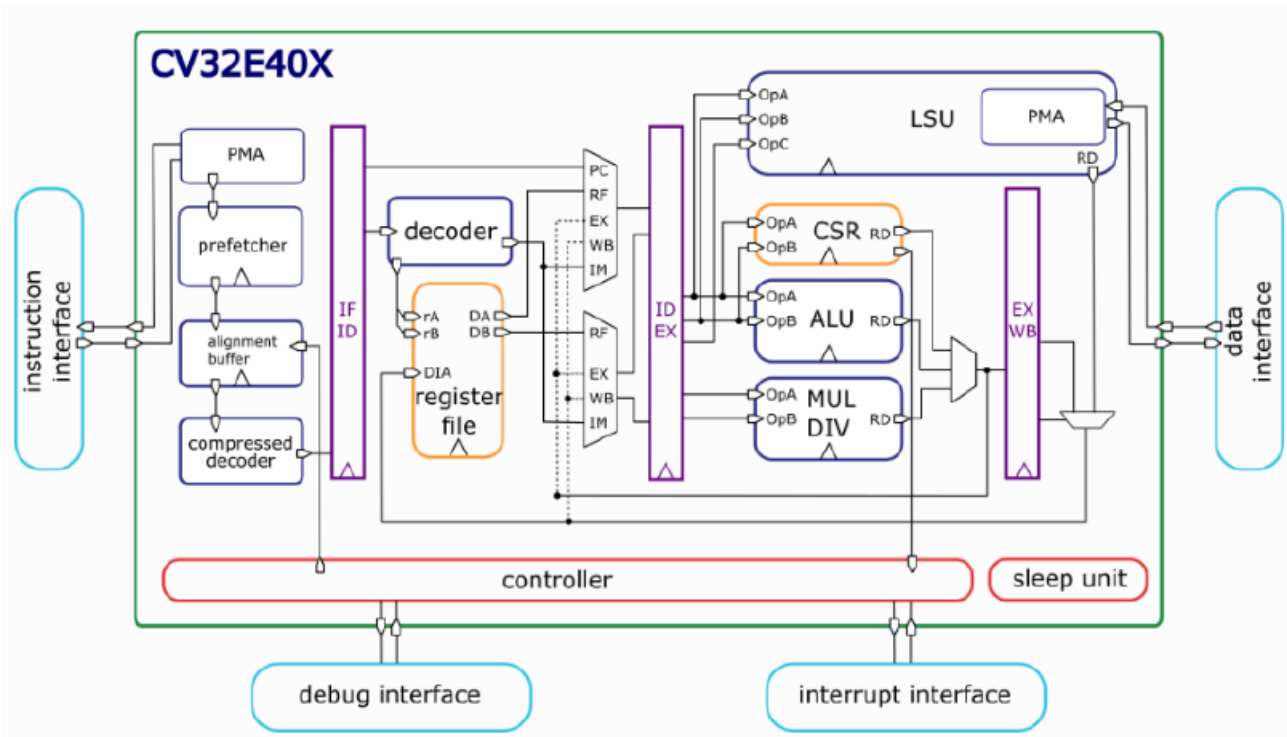


Figure 3.1.1: CV32E40X core[14]

CV3250X is pipelined RISC-V core developed by OpenHW group and is widely used by the industry[14]. As stated in the introduction chapter, the main goal of

the thesis is to increase the core's speed by implementing branch predictors. This section briefly discusses the core's architecture.

As can be seen from figure 3.1.1. The core is designed as an order processor with four stages[14]:

- Instruction Fetch stage – The core can work with regular, 32-bit long, and compressed, 16-bit long instructions. The instruction interface fetches 32-bit long words from the instruction memory[1][14].
- Instruction Decode stage – Based on the instruction opcode, the decoder unit identifies the nature of received instruction and retrieves source operands from the register file. This stage also calculates the branch target address. Jump instructions are also taken from ID stage[1][14].
- Execution stage – This stage contains load-store (LSU), arithmetic logic (ALU), multiply (MUL), and division (DIV) units. LSU evaluates data memory addresses for load and store instructions. Besides performing arithmetic and logic instructions, ALU also evaluates the branch condition. MUL and DIV units perform multiplication and division[1][14].
- Write-Back stage – At this stage, LSU communicates with data memory using a data interface. Regarding arithmetic/logic instructions, the destination register is updated with the result calculated at EX stage[1][14].

Apart from four pipeline stages, the core also contains a module called the controller. The controller performs functions including data forwarding, stalling, jumping/branching to target address, etc[14].

The core is implemented as SystemVerilog modules. In order to implement a 2-bit branch predictor, I had to modify modules implementign the controller and IF, ID, and EX stages. The following subsections analyze baseline implementations of them.

3.1.1 Controller

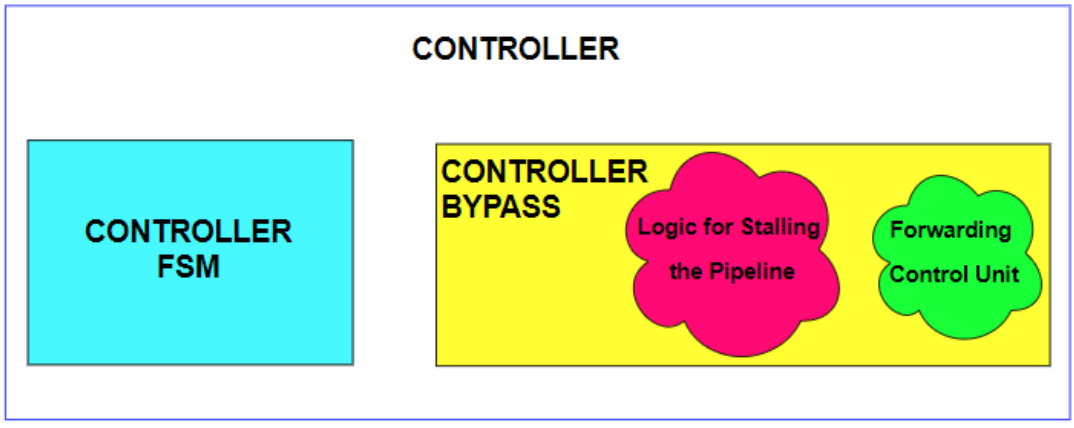


Figure 3.1.2: CV32E40X-Controller

The controller is implemented as a module named cv32e40x_controller[15]. As shown in figure 3.1.2, the controller instantiates controller_bypass and controller_fsm modules. The following two subsections elaborate on the functionality of two modules instantiated in the controller.

3.1.1.1 Controller_bypass Module

The bypass module contains logic responsible for data forwarding and stalling the pipeline[15].

Data forwarding is a pipeline optimization technique that makes source values available to the instruction if they have been calculated but not yet stored in source registers[16]. The controller_bypass module forwards the execution result from the EX stage to the ID stage, and the execution result or data loaded from memory from the WB stage to the ID stage so that the instruction at the ID stage can use the forwarded data as its source operand before the data is stored in the register file[15]. This way, the instruction at the ID stage does not have to wait for its source register values if they are already available in the later pipeline stages but have not yet been delivered to the source registers.

Before the instruction is passed down from one pipeline stage to another, the two stages perform a handshake[14]. The "receiving" stage asserts a ready signal indicating it is ready to accept the instruction, and the "transmitting" stage asserts a valid signal. Unless both ready and valid signals are asserted, the "transmitting" stage does not update the pipeline registers presenting between the "transmitting" and "receiving" stages[14]. Figure 3.1.3 depicts a block diagram implementing the handshake between pipeline stages.

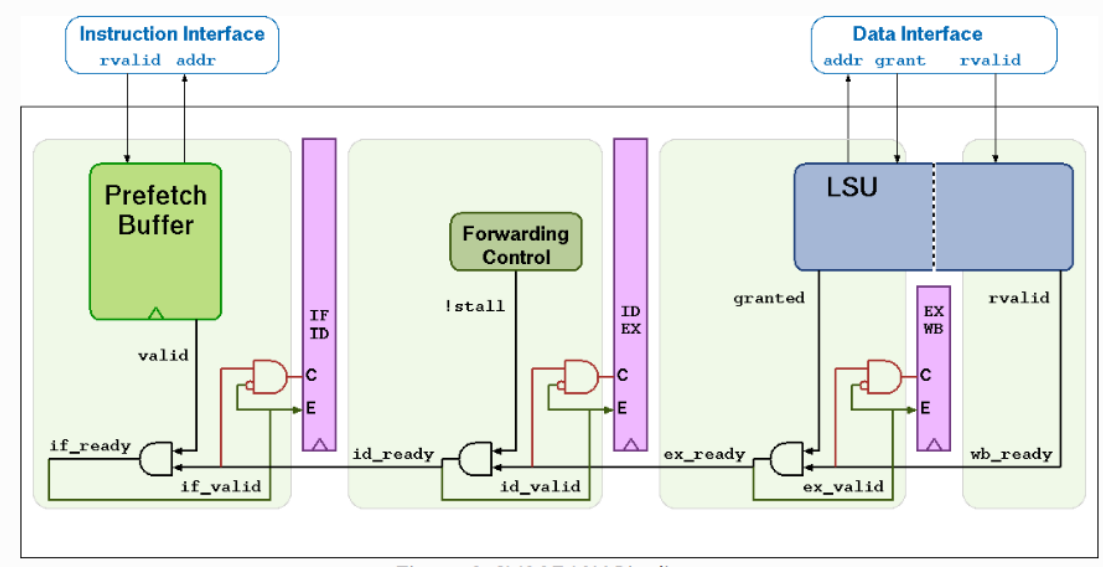


Figure 3.1.3: Handshake between pipeline stages[14]

If the operands for the instruction at the ID stage have not yet been calculated or retrieved from the memory, the controller_bypass module asserts stall signal[14]. It can be observed in figure 3.1.3 that stalling sets the id_ready signal to 0, which

causes the handshake between the fetch and decode stage to fail. The controller stalls the IF and ID stages of the pipeline until the operand value is forwarded to ID stage[14]. The work done for the thesis does not modify the Controller_bypass module.

3.1.1.2 Controller_fsm

The Controller_fsm is more important for the thesis than the Controller_bypass one because it contains the logic that handles branch instructions. It implements the main Finite State Machine (FSM). The FSM controls the pipeline by using a special datatype - ctrl_fsm_t. Figure 3.1.4 shows some of the signals in the defined datatype. The signals important to the thesis are the ones sent to the IF stage and kill signals:

```
// Controller FSM outputs
typedef struct packed {

  // to IF stage
  logic        pc_set;              // Jump to address set by pc_mux
  pc_mux_e     pc_mux;              // Selector in the Fetch stage to select the rigth PC (normal, jump ...)

  // Kill signals
  logic        kill_if; // Kill IF stage
  logic        kill_id; // Kill ID stage
  logic        kill_ex; // Kill EX stage
```

Figure 3.1.4: Controller FSM Data Type[15]

- **PC_SET** - the signal is asserted when the program execution needs to deviate from the consecutive order. This signal is asserted for branch and jump instructions, interrupts, exceptions, etc. By default, the signal is de-asserted[15].
- **PC_MUX** - this is the PC multiplexer control signal. It controls the multiplexer implemented in the IF stage (discussed later) and decides where the program execution should jump. By default, the PC_MUX is assigned with PC_BOOT[15].
- **KILL** - kill signals invalidate the instructions in the pipeline stages. For example, if the branch instruction is taken from the EX stage, instructions in the ID and IF stages are killed by asserting kill_id and kill_if signals. By default, all kill signals are de-asserted[15].

The FSM has six states: RESET, BOOT_SET, FUNCTIONAL, SLEEP, DEBUG_TAKEN, and POINTER_FETCH[15]. Since the FUNCTIONAL state handles all the encountered branch and jump instructions, only that state will be discussed in detail.

Functional state is implemented as IF-ELSE block and handles all the cases that deviate from consecutive instruction execution order. Each of those cases has different priorities and is handled accordingly. Figure 3.1.5 depicts a decision diagram for the FUNCTIONAL state. It can be seen from the figure that non-maskable interrupts (NMI) take the highest priority, and they are handled first. The maskable

interrupts (MI) have lower priority than NMI but higher priority than exceptions (EXC). Special introductions (SPIN) are next in line. Branch instructions take the second lowest priority. Finally, if none of the above-mentioned cases are present, the FUNCTIONAL state handles with jump instructions (JMP)[15]. Interestingly, BR has a higher priority than JMP. This can be explained by the fact that if the branch is taken, the baseline core needs to invalidate instructions in IF and ID stages.

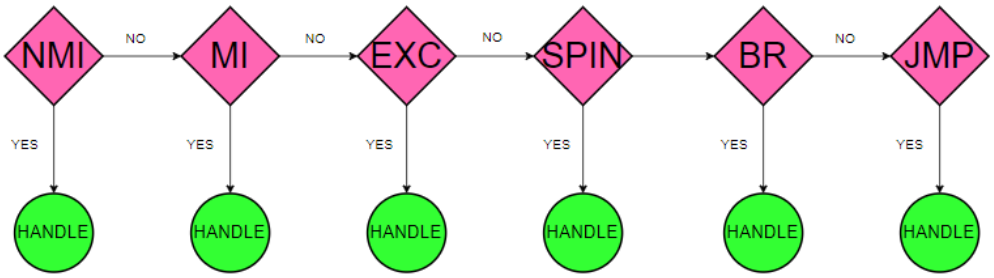


Figure 3.1.5: Controller FSM - Functional State Decision Diagram

The controller handles branch instructions if the branch is detected in the EX stage and the branch condition is considered true. If that is the case, branch_taken_ex is asserted. Figure 3.1.6 shows the logic used to calculate the branch_taken_ex signal. Signals, such as id_ex_pipe_i.alu_bch, id_ex_pipe_i.alu_en, id_ex_pipe_i.instr_vali and branch_decision_ex_i, come from the EX stage while branch_taken_q is implemented in the controller_fsm module.

- id_ex_pipe_i.alu_bch - asserted if instruction in EX stage is a branch.
- id_ex_pipe_i.alu_en - asserted if ALU is enabled.
- id_ex_pipe_i.instr_valid - asserted if instruction in EX stage is valid.
- branch_decision_ex_i - asserted if the branch condition is evaluated to be true.
- branch_taken_q - This signal is an output of a register. By default, the value of the register is logic 0. Once the branch instruction is taken, the signal becomes logic 1 and does not change until the EX stage receives new instruction. Thus, the signal prevents further branches to the same target.

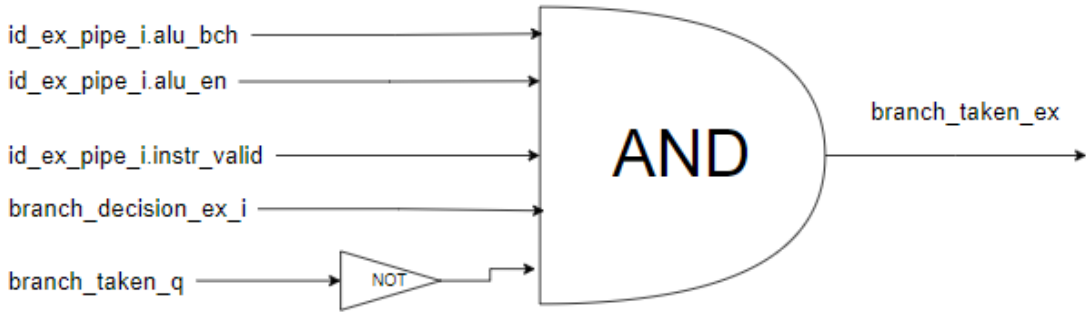


Figure 3.1.6: The Logic for Branch Taken Detection

Figure 3.1.7 depicts part of the FSM code that handles branch instructions. As discussed in the State of Art chapter, instructions in IF and ID stages must be invalidated if the branch is taken. This is achieved by asserting ctr_fsm_o.kill_if and ctr_fsm_o.kill_id signals. The code also sets pc_set and assigns pc_mux with PC_BRANCH value. Hence, the PC multiplexer, implemented in the IF stage, assigns a branch target address to the PC. Furthermore, setting the branch_taken_n flag guarantees that the same branch is not taken more than once. The flag remains assigned to the branch_taken_q register. According to figure 3.1.6 setting branch_taken_q to logic 1 makes branch_taken_ex signal logic 0. Thus, controller FSM kills IF and ID stages and branches to the target address, providing that there is a branch instruction in the EX stage and the condition is considered true.

```
end else if (branch_taken_ex) begin
  ctrl_fsm_o.kill_if = 1'b1;
  ctrl_fsm_o.kill_id = 1'b1;

  ctrl_fsm_o.pc_mux  = PC_BRANCH;
  ctrl_fsm_o.pc_set  = 1'b1;

  // Set flag to avoid further branches to the same target
  // if we are stalled
  branch_taken_n     = 1'b1;
```

Figure 3.1.7: Code for Branch Taking[15]

3.1.2 Instruction Fetch Stage

The cv32e40x_if_stage module implements the logic for the IF stage. The module implements the IF_ID pipeline register and PC multiplexer for branching, which decides where the program execution should jump[15]. The module consists of a prefetcher and alignment buffer. Those two modules fetch instructions from the memory, implement 3-entry long first-in-first-out (FIFO) memory for storing the

fetched instructions, and calculate the address of the next instruction[15].

Figure 3.1.8 displays systemverilog code implementing IF_ID_PIPE_REGISTERS. The code consists of an IF-ELSE statement. When the reset signal is asserted, all register signals are set to default values. The IF part (lines 351-394) updates the pipeline register. It can be seen on line 351 that the IF part of the code is guarded by two signals: if_valid_o and id_ready_i. The IF part updates the pipeline register if the IF stage has a valid instruction and the ID stage is ready to receive a new instruction. Either of those signals is logic low, the new instruction is not propagated from IF to the ID stage, and the content of the pipeline register is invalidated (LINE 395).

```
333   always_ff @(posedge clk, negedge rst_n)
334   begin : IF_ID_PIPE_REGISTERS
335     if (rst_n == 1'b0) begin···
348     end else begin
349       // Valid pipeline output if we are valid AND the
350       // alignment buffer has a valid instruction
351       if (if_valid_o && id_ready_i) begin···
394       end else if (id_ready_i) begin
395         if_id_pipe_o.instr_valid      <= 1'b0;
396       end
397     end
398   end
```

Figure 3.1.8: IF_ID_Pipeline Register (Compressed)[15]

Figure 3.1.9 portrays the part of the PC multiplexer. When program execution needs to deviate from consecutive order, the controller uses ctrl_fsm_i.pc_mux signal to choose the correct target address. From the figure, we can see three cases that require execution order deviation:

- PC_BOOT - in case of a boot request, branch_addr_n is assigned with the boot address.
- PC_JUMP - in case of jump instructions, branch_addr_n is assigned with the jump target address.
- PC_BRANCH - branch_addr_n is assigned with the boot address in case of branch instructions.

Signal branch_addr_n is delivered to the alignment buffer module, which assigns the address to the PC register so that the next instruction will be fetched from the target address[15]. Furthermore, the alignment buffer clears FIFO because the instructions inside the FIFO are no longer relevant to the execution order[15].

```
140   // Fetch address selection
141   always_comb
142   begin
143     // Default assign PC_BOOT (should be overwritten in below case)
144     branch_addr_n = {boot_addr_i[31:2], 2'b0};
145
146     unique case (ctrl_fsm_i.pc_mux)
147       PC_BOOT:      branch_addr_n = {boot_addr_i[31:2], 2'b0};
148       PC_JUMP:      branch_addr_n = jump_target_id_i;
149       PC_BRANCH:    branch_addr_n = branch_target_ex_i;
```

Figure 3.1.9: PC Multiplexer

3.1.3 Instruction Decode and Execute Stages

The logic for the ID stage is implemented in the module named cv32e40x_id_stage. The module contains logic for decoding the instruction, getting operands for arithmetic logic operations, and calculating brunch/jump target address[15]. Furthermore, similarly to the module describing the IF stage, cv32e40x_id_stage implements the ID_EX pipeline register, updated if and only if the ID stage has valid instructions and the EX stage is ready to accept new instructions[15].

The logic for the EX stage is implemented in the module named cv32e40x_ex_stage. The module implements the ALU, multiplier, and divider[15]. Furthermore, the module sends the branch target address to the PC multiplexer, implemented in the IF stage[15].

3.2 Description of Branch Prediction Model

The thesis is based on the specialization project[1]. A high-level model for a 2-bit branch predictor was developed and implemented in Python throughout the specialization project. This section briefly discusses the model.

The project utilized a verification environment provided by openHW group[17]. The environment simulates the CV32E40X core, runs test programs, and generates a log file[1]. Figure 3.2.1 displays a fraction of the log file. The file consists of a table with 11 columns[1]. Each row of the table gives us information about executed instruction[1]. The following 3 columns are important for the developed model:

- CYCLE - specifies the clock cycle in which the instruction execution finished[1].
- PC - specifies the address of the instruction corresponding to the raw[1].
- INSTR - specifies the instruction in hexadecimal[1].

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 3.2.1: Log file generated by simulating CV32E40X core[1]

Figure 3.2.2 displays the algorithm implemented in the model. As seen from the figure, the input to the model is the log from the verification environment. The idea is to generate a new log that the verification environment would have generated if the 2-bit branch predictor had already been implemented in the core[1]. The model implements BTB as a dictionary and introduces a variable called prediction_reward[1]. The variable is initialized with 0 and counts how many clock cycles were saved by the prediction. According to the algorithm in the figure 3.2.2, in the beginning, the model reads raw for the first instruction into the variable called current_line and reads the raw for the consecutive instruction into the variable next_line[1]. Next, the PC value is extracted from the current_line, and the CYCLE column of the current_line is increased by the value stored in the prediction_reward variable[1]. The modified current_line is written into the new log file. Next, if the PC is in the BTB, the model makes the prediction based on the prediction algorithm similar to the 2-bit prediction algorithm discussed in the State of Art chapter. Otherwise, the PC and corresponding target address are added to BTB[1].

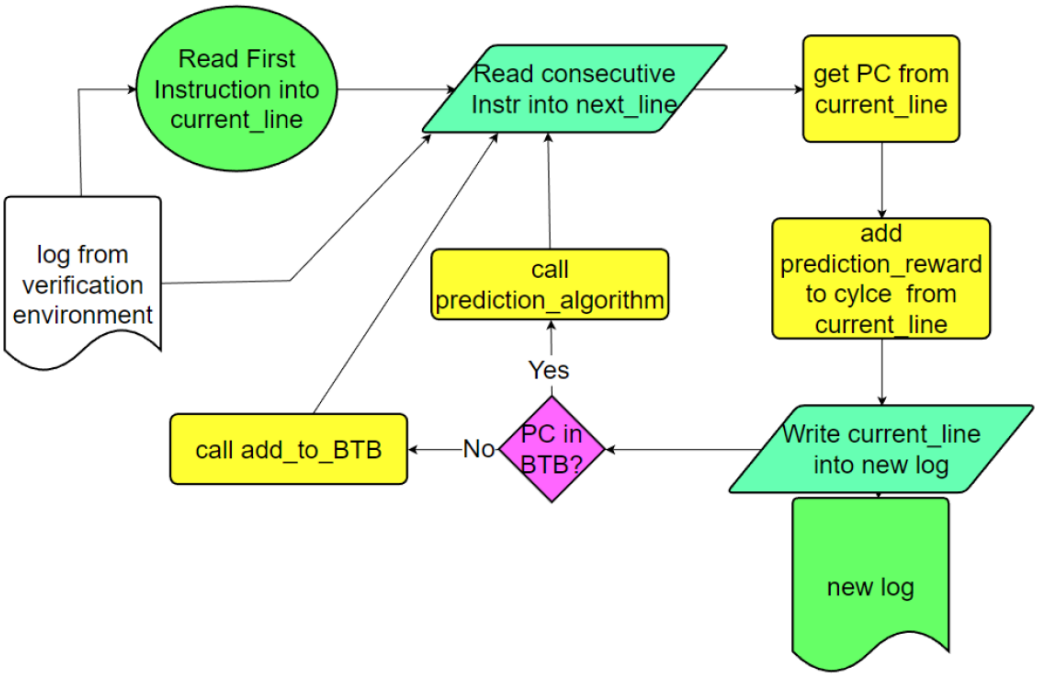


Figure 3.2.2: flow diagram for the method branch prediction model[1]

The detailed description of the model and test run can be found in the specialization project report[1]. The project concluded that implementing a 2-bit branch predictor can increase IPC by at most 6.67%[1]. The implementing predictor was one of the thesis objectives, and the following two sections discuss the intended design and implementation details.

3.3 Description of Intended Design

Figure 3.3.1 portrays a block diagram of the pipeline with modifications required for 2-bit branch predictor implementations. As we can see, the main modification is done to the IF stage. The module named BTB_BHT is a new module that implements BTB and BHT as a cache. The module implements the logic for the prediction and the cache update. It has three main inputs - pc_ex_i, target_pc_ex_i, and cache_operation. The first two inputs come from the EX stage and correspond to the branch's instruction and target address. Cache_operation comes from the controller, performing one of the three cache operations: adding new entries to the BTB_BHT cache, incrementing the BHT counter, or decrementing the BHT counter.

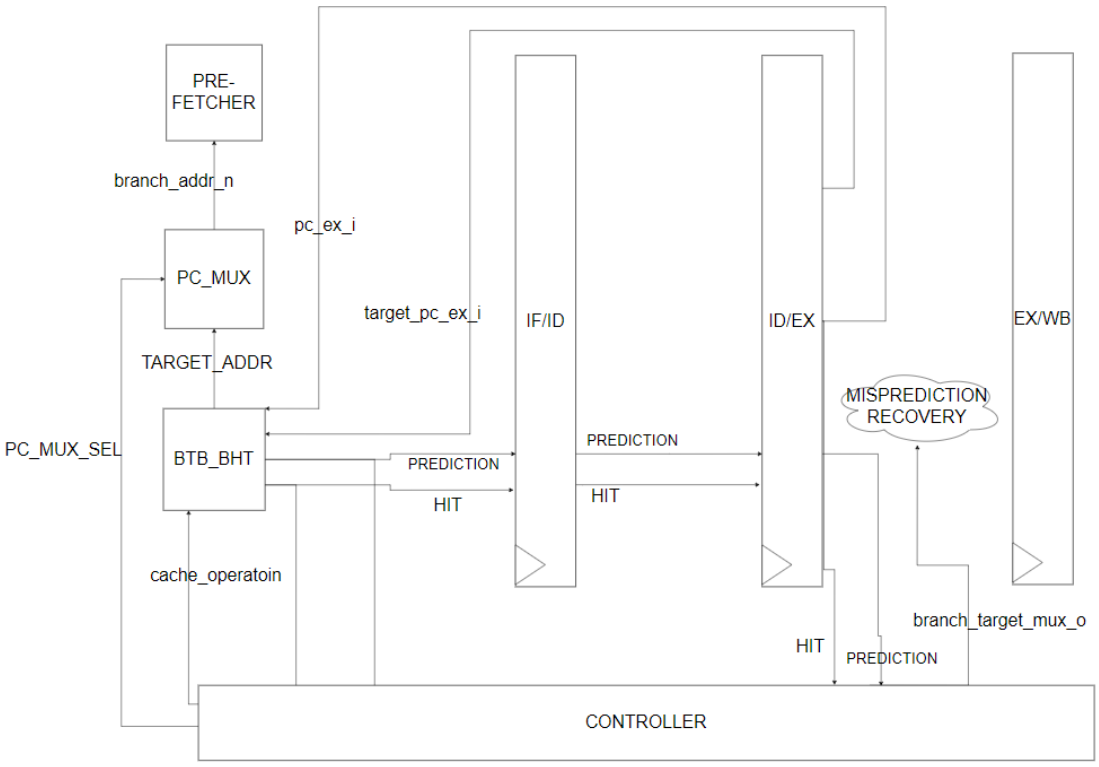


Figure 3.3.1: Block Diagram of Predictor Modules in the Pipeline

Furthermore, the module takes the current PC as an input and outputs the corresponding branch target address to PC_MUX, providing that the instruction in the IF stage is the branch. The BTB_BHT module also outputs two signals - HIT and PREDICTION, to the controller and IF_ID pipeline register. HIT is asserted if the branch instruction is fetched into the IF stage, and PREDICTION is asserted if the branch is predicted to be taken. Those two signals are

delivered to the Controller module to assign value to the PC_MUX_SEL signal that selects the branch target address (TARGET_ADDR) as the next PC. Furthermore, those two signals are propagated through the ID and EX pipeline stages.

It can be observed from figure 3.3.1 that the EX stage is also modified. It contains logic that recovers the execution order in case of a branch misprediction. EX stage delivers HIT and PREDICTION signals from the ID_EX pipeline register to the Controller module. The controller checks if the prediction coincides with the branch outcome, evaluated in the EX stage. In case of misprediction, branch_target_mux_o receives the value that recovers the execution order. The following section discusses modifications done to the baseline core code in detail.

3.4 Implementation of Intended Design

The work done for the thesis included implementing a 2-bit predictor from figure 2.3.5 into the CV32E40X RISC-V Core from figure 3.1.1. I implemented a new module, cv32e40x_BTB_BTH, and modified already implemented modules - cv32e40x_if_stage, cv32e40x_controller_fsm, and cv32e40x_ex_stage. Before discussing the modifications, one can observe datatypes defined for cache command, branch target multiplexer, and cacheline (Figure 3.4.1).

- **CACHE_CMD** - is defined as ENUM logic. Signals with cache_cmd type can have only four possible values: INCREMENT, DECREMENT, NEW_ENTRY, and NOP.
- **BRANCH_TARGET_MUX_T** - is defined as ENUM logic. Signals with this type can have two values: OPERAND_C and NEXT_PC.
- **CACHE_LINE_T** - is defined as a struct. The struct contains four variables: valid - indicating the validity of the line; tag - stored tag of the line; target_pc - stores branch target value; prediction_cnt - stores 2-bit counter.

```
typedef enum logic [1:0] {INCREMENT = 2'h0, DECREMENT = 2'h1, NEW_ENTRY = 2'h2,
                          NOP = 2'h3} cache_cmd;

typedef enum logic [1:0] {OPERAND_C = 2'h0, NEXT_PC = 2'h1} branch_target_mux_t;

parameter TAG_WIDTH = 31;

typedef struct
{
  logic valid;
  logic [TAG_WIDTH - 1:0] tag;
  logic [31:0] target_pc;
  logic [1:0] prediction_cnt;
}cache_line_t;
```

Figure 3.4.1: Defined Datatypes for cache_cmd, branch_target_mux and cache_line

3.4.1 CV32E40X_BTB_BTH

The work done for the thesis included implementing module CV32E40X BTB_BTH from scratch. This section describes the module.

CV32E40X_BTB_BTH module implements branch history table and branch target buffer as a cache and generates btb_statistics file containing the information about the maximum number of valid entries in the cache during the simulation, number of encountered branch instructions, number of predictions, number of correct and incorrect predictions. The module has one input from the IF stage (pc_if_i - current PC), two inputs from the EX stage (pc_ex_i - PC from EX stage, and target_pc_ex_i - branch target address), and one input from the controller (cache_operation). Moreover, the module has three outputs to the IF stage (hit_o - indicating if the branch instruction is detected at the IF stage, prediction_cnt_o - a value of the 2-bit counter associated with the detected branch, and target_pc_if_o - branch target address).

The main part of the module is BTB_BHT which is implemented as a cache. The cache is implemented as a SystemVerilog array of type CACHE_LINE_T (figure 3.4.1). The number of elements in the cache is determined by the module parameter called SIZE. Figure 3.4.2 portrays addressing a cache. The part of the input pc_if_i indexes the cache. Since the 0th bit of the PC is always 0, the indexing starts from the first bit of the PC and ends at the width bit (pc_if_i[width : 1]), where width is the parameter of the module specifying the bit width of the cache index. If the pc_if_i[width : 1] is less than the cache size, pc_if_i[width : 1] becomes the index to the cache. Otherwise, size is subtracted from pc_if_i[width : 1], and the difference becomes the index, preventing indexing to the cache line at the index that is more than the size. The indexing logic is implemented with a subtractor, comparator, and multiplexer.

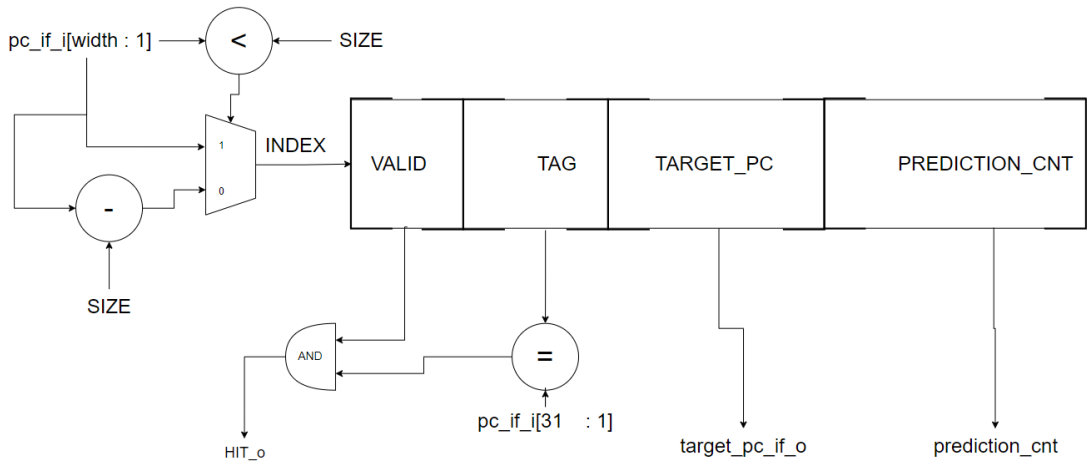


Figure 3.4.2: Indexing Cacheline with PC from IF Stage

Furthermore, it can also be observed from the figure that the cacheline contains TAG. In our case, the tag is the upper 31 bits of the PC which corresponds to the cacheline. If the cache line tag indexed by pc_if_i is the same as the upper 31 bit of the pc_if_i signal and cacheline is VALID, HIT_o signal is set to logic 1.

HIT_o, target_pc_if_o, and prediction_cnt are delivered to the IF stage. Those signals help the controller to make predictions about the branch outcome in the IF stage.

After the branch signal reaches the EX, the controller knows whether the prediction was correct or incorrect and updates the cacheline corresponding to the branch instruction. At this point, the branch instruction is in the EX, and the PC from the EX stage (pc_ex_i) is used to index the cacheline (figure 3.4.3). Since the module modifies cacheline that has already been addressed at previous pipeline stages, checking the line's validity is no longer necessary. The cache update logic is implemented inside the SystemVerilog ALWAYS_FF block as an IF_ELSE statement. IF part of the statement invalidates all cache lines during the active reset signal. ELSE part implements the logic for the update as a case statement. The case variable is the input cache_operation, taken from the controller. The cahce_operation can have one of the possible values:

- NEW_ENTRY - the branch instruction was not detected in the IF stage, and the new entry is added to the cacheline indexed by pc_ex_i. The cacheline becomes valid (VALID signal is set to logic 1), pc_ex_i[width : 1] is stored in the TAG of the cacheline, TARGET_PC is assigned with the branch target address, and PREDICTION_CNT becomes 3.
- INCREMENT - PREDICTION_CNT is incremented with saturation arithmetic.
- DECREMENT - PREDICTION_CNT is decremented with saturation arithmetic.
- NOP - the instruction at the EX stage is not branch. The cache is not modified.

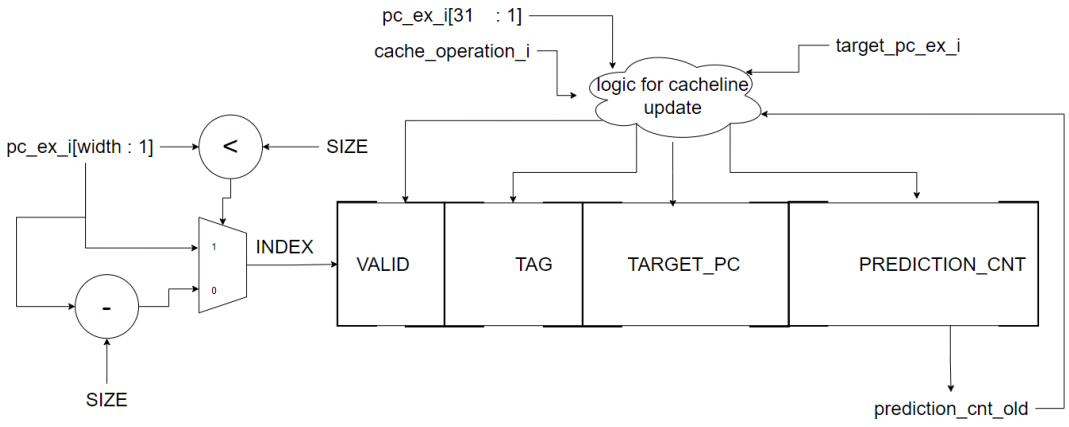


Figure 3.4.3: Updating Cacheline Indexed with PC from EX Stage

Figure 3.4.4 represents a truth table for incrementing and decrementing 2-bit counter with saturation arithmetic. The idea is that if the old value of the counter is 2'b11, the new value will still be 2'b11 after incrementing. Similarly, if the old value is 2'b00, the new value after decrementing would be 2'b00.

INCREMENT				DECREMENT			
OLD		NEW		OLD		NEW	
Prediction cnt[1]	Prediction cnt[0]	Prediction cnt[1]	Prediction cnt[0]	Prediction cnt[1]	Prediction cnt[0]	Prediction cnt[1]	Prediction cnt[1]
0	0	0	1	0	0	0	0
0	1	1	0	0	1	0	0
1	0	1	1	1	0	0	1
1	1	1	1	1	1	1	0

Figure 3.4.4: Truth Table for 2-bit Prediction Counter Increment and Decrement

Based on the truth table, the saturation incrementing is implemented by equations 3.1 and 3.2 and the saturation decrementing by equations 3.3 and 3.4.

$$prediction\_cnt[1]new = prediction\_cnt[1]ORprediction\_cnt[0] \tag{3.1}$$

$$prediction\_cnt[0]new = prediction\_cnt[1]OR(\ prediction\_cnt[0]) \tag{3.2}$$

$$prediction\_cnt[1]new = prediction_cnt[1]ANDprediction_cnt[0] \tag{3.3}$$

$$prediction_cnt[0]new = prediction_cnt[1]AND(\ prediction_cnt[0]) \tag{3.4}$$

3.4.2 CV32E40X_IF_STAGE

The work done for the thesis included modifying module CV32E40X_IF_STAGE. This section describes the modifications done to the module.

The module, implementing BTB_BHT cache, is instantiated in cv32e40x_IF_STAGE. As discussed in the previous section, hit_o, prediction_cnt_o, and target_pc_if_o are outputs to the IF stage. Hit_o is directly delivered to the controller so that the controller knows if there is a known branch instruction in the IF stage. Prediction_cnt_o is a 2-bit long signal with four possible values: 00, 01, 10, and 11. IF its value is more than 1, the prediction is TAKEN. Otherwise - NOT TAKEN. It can be observed that the 2-bit counter value is more than 1 if and only if the most significant bit is 1. Hence, the most significant bit of prediction_cnt_o is assigned to signal prediction_o, which is delivered to the controller so that it knows if the detected branch is predicted to be TAKEN or NOT. Moreover, the cv32e40x_IF_STAGE module ensures that hit_o and prediction_o are asserted if the IF stage has valid instructions and the ID stage is ready to receive a new instruction.

Furthermore, the PC multiplexer from figure 3.1.9 was slightly modified. One more case was appended to the case statement - PC_PREDICTED. If the controller sets ctrl_fsm_i.pc_mux signal to PC_PREDICTED, there is a branch instruction in the IF predicted to be TAKEN, and branch_addr_n is assigned with the branch target address taken from the BTB_BHT cache.

3.4.3 CV32E40X_ID_STAGE and CV32E40X_EX_STAGE

The work done for the thesis included modifying modules CV32E40X_ID_STAGE and CV32E40X_EX_STAGE. This section describes the modifications done to the modules.

As we have already seen, the IF stage makes the prediction. However, the outcome of the branch remains unknown until the branch instruction reaches the EX stage and the branch condition is evaluated. In order for the controller to check the prediction against the branch outcome, it needs to know if the prediction was made in the first place. Thus, hit_o and prediction_o signals are propagated from the IF to the EX stage through the ID stage. The only modification to cv32e40x_ID_STAGE is that hit_o and prediction_o signals are stored in the ID_EX pipeline register, providing that ID has the valid instruction and EX is ready to accept new instructions. Moreover, it is also important for future discussion of the implementation details to point out that cv32e40x_ID_STAGE instantiates the module that calculates the branch target address and assigns it to the pipeline register - id_ex_pipe.operand_c.

CV32E40X_EX_STAGE is modified to recover the execution order in case of misprediction. There are two kinds of misprediction:

- Prediction TAKEN, Outcome NOT TAKEN - The branch is predicted to be TAKEN, but the EX stage evaluated the branch condition as FALSE, meaning that the branch should NOT HAVE BEEN TAKEN. The next instruction's address should be the branch instruction PC incremented by the offset between two consecutive instructions.

- Prediction NOT TAKEN, Outcome TAKEN - The branch is predicted to be NOT TAKEN, but the EX stage evaluated the branch condition as TRUE, meaning the branch should have been TAKEN. The address of the next instruction should be the branch target address.

Block diagram for the misprediction recovery logic is portrayed by figure 3.4.5. One can see two multiplexers on the block diagram. The multiplexer to the right is controlled by the signal coming from the controller (branch_target_mux_i) and chooses between two values: OPERANC_C (Branch target address) and NEXT_PC(address of the instruction consecutive to the branch). NEXT_PC is calculated by adding an increment (offset to the next instruction) to the PC corresponding to the branch instruction. The increment can be 2 or 4. If the instruction in the EX stage is compressed, the increment is 2 because the offset between two consecutive instructions is 2. Otherwise, the increment is 4.

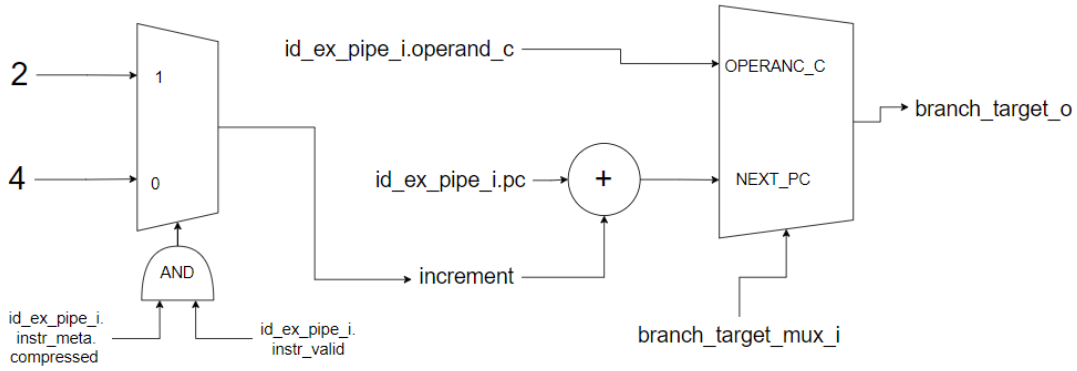


Figure 3.4.5: Block Diagram for Misprediction Recovery in EX Stage

3.4.4 CV32E40X_CONTROLLER_FSM

The work done for the thesis included modifying module CV32E40X_CONTROLLER_FSM. This section describes the modifications done to the module.

Before discussing modifications done to the controller FSM from figure 3.1.5, it will be helpful to introduce new signals implemented for the FSM to perform the branch prediction and misprediction recovery (figure 3.4.6).

- HIT_EX - The signal comes from the EX stage. If the signal is logic 1, the branch instruction outcome was predicted two stages ago.
- PREDICTION_EX - The signal comes from the EX stage, indicating the branch prediction.
- BRANCH_IN_EX - The signal comes from the EX stage, indicating a branch instruction in the EX stage that the controller has not yet handled. After the controller handles the branch, the bracnh_taken_q signal is set to logic 1, which deactivates BRANCH_IN_EX.
- BRANCH_TAKEN_EX - The signal comes from the EX stage and represents the branch decision. If the signal is logic 1, the branch should be taken.
- HIT_I - The signal comes from the IF stage and indicates if branch instruction has been detected.
- Prediction_I - The signal comes from the IF stage and indicates the branch prediction.

```
286   assign hit_ex          = id_ex_pipe_i.hit && id_ex_pipe_i.instr_valid;
287   assign prediction_ex   = id_ex_pipe_i.prediction && id_ex_pipe_i.instr_valid;
288   assign branch_in_ex    = id_ex_pipe_i.alu_bch && id_ex_pipe_i.alu_en &&
289                            id_ex_pipe_i.instr_valid && !branch_taken_q;
290   assign branch_taken_ex = branch_in_ex && branch_decision_ex_i;
```

Figure 3.4.6: Signals Used by FSM for Prediction/REcovery

Figure 3.4.7 portrays a flow diagram of the implemented logic for branch prediction and misprediction recovery. It can be seen from the figure that the first check performed is whether there is a branch in the EX stage or not (fig. 3.4.7 BRANCH_IN_EX). If the first outcome of the first check is FASLE, the FSM proceeds with checking and handling jump instructions in the ID stage (fig. 3.4.7 JUMP in ID). If there are no jumps, the controller checks if the branch instruction is detected in the IF and if it is predicted to be TAKEN (fig. 3.4.7 Hit_i and predict_i). If that is the case, FIFO from the alignment buffer is cleared, PC multiplexer in the IF stage selects the branch target address, output from the cv32e40x_BTB_BTH module, to be the next PC. PC_SET signal is set to logic 1, notifying the alignment buffer about the branching. This case does not modify the default value (OPERAND_C) of the misprediction recovery multiplexer in the EX stage (figure 3.4.5).

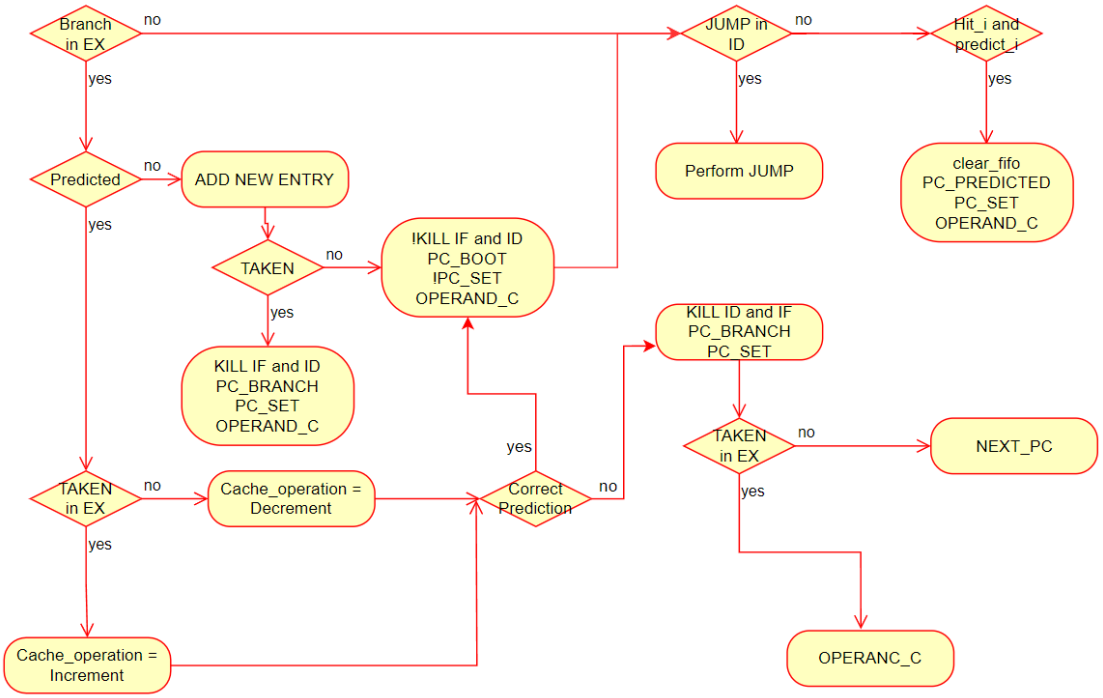


Figure 3.4.7: Block Diagram for Part of the FSM handling Branching, Jumping, Prediction, and Misprediction Recovery

If the outcome of the first check is true and there is a branch instruction in the EX stage, the controller proceeds to check if the branch has been predicted two stages ago (fig. 3.4.7 Predicted). If the check outcome is FALSE, the new entry is added to the BTB_BHT cache by setting cache_command to NEW_ENTRY. Next, the FSM checks whether the branch should be taken (fig. 3.4.7 TAKEN). If the answer is yes, IF and ID stages are killed, PC_SET is set to logic 1, PC multiplexer assigns the branch target address, delivered from EX stage branch recovery multiplexer delivers the branch target address (OPERAND_C), to the next PC. However, if the branch should not be taken, there is no need to kill any pipeline stages or recover the execution order. Hence, PC_SET is set to logic 0 to prevent branching. Next, the controller handles jump instructions in ID (fig. 3.4.7 JUMP in ID) and checks branch instructions (fig: **??** Hit_i and predict_i)

in IF.

If the branch instruction in the EX stage was predicted two stages ago, the controller checks if the branch should have been taken (fig: 3.4.7 from Predicted to Taken in EX). If it should be taken, the controller increments the 2-bit counter associated with the branch instruction in the BHT (cache_operation = INCREMENT). Otherwise, the counter is decremented (cache_operation = DECREMENT). Next, the controller checks if the prediction was correct (fig: 3.4.7 Correct Prediction). The check is performed by comparing branch_taken_ex and prediction_ex. If they equal each other, the branch prediction and outcome are the same. Hence the prediction was correct. The controller does not kill IF and ID stages, sets PC_SET to logic 0 to disable branching, and proceeds to handle jump instructions in ID (fig. 3.4.7 JUMP in ID) and checking branch instructions (fig: **??** Hit_i and predict_i) in IF.

On the other hand, if branch_taken_ex does not equal prediction_ex, the branch prediction and outcome differ, meaning that the branch has been mispredicted. If the branch should be taken, the control signal to the misprediction recovery multiplexer is set to OPERAND_C to deliver the branch target address to the IF stage (fig. 3.4.5). Otherwise, the control signal is set to NEXT_PC.

Overall, the student implemented a 2-bit branch predictor unit as a BTB_BHT cache, logic for detecting and predicting branch instructions in the IF stage, a multiplexer for recovering misprediction in the EX stage, and extended controller FSM to supervise branch predictions and recovery. The following two sections present the baseline performance of the core and describe tests performed on the core with the prediction.

3.5 Metrics and Baseline Performance of CV32E40X Core

As stated in the introduction chapter, the main goal of the thesis is to increase the speed of the CV32E40X core without increasing power consumption and area usage by more than 20%.

Metric for evaluating processor speed is instruction per clock cycle (ICP). According to equation 3.5, IPC is calculated by dividing the number of instructions in the program by the number of clock cycles the processor requires for the program execution. In other words, IPC tells us how many instructions can be executed in a clock cycle[18].

$$IPC = (number\ of\ instructions)\ /\ (number\ of\ clock\ cycles) \tag{3.5}$$

Metrics for area evaluation are Cell Area - referring to the area occupied by the logic cells of the design, Net Area - represents the area required for the wiring between all the logic cells, and Total Area - referring to the sum of Cell and Net area[19]. Finally, the metric used for power estimation is Watt.

3.5.1 Baseline Performance of CV32E40X Core

Figure 3.5.1 portrays baseline performance for CV32E40X core. It can be seen from the figure that the IPC of the core without a 2-bit branch predictor is 0.46, which means that the core executes 46% of a single instruction in one clock cycle. It can also be observed that most of the area is taken by the Cells. Finally, since the core is developed for embedded systems, it stands to reason that the power consumption is as low as 270 Micro Watts.

	IPC	Net Area	Cell Area	Total Area	Power
Baseline Core	0.46	5874.265	16704.66	22578.927	2.70943e-04

Figure 3.5.1: Baseline Performance

3.6 Descriptions of Tests

The control variable for the tests run on the core was the number of entries in the BTB_BHT cache. Throughout the testing, it changed between 2-256, and the size of the increment was 2.

The core was simulated using QuestaSim and a verification environment developed by the OpenHW group. During the simulation, the information was extracted about IPC, the maximum number of valid entries in the cache during the simulation, the number of encountered branch instructions, the number of predictions, and the number of correct and incorrect predictions. The scripts and guidelines for the core simulation are provided in the GitHub repository. Appendix A provides more information about how to simulate the core.

Furthermore, the core was synthesized using the Cadence Genus tool and STM 28nm technology. During the simulation, information about Cell Area, Net Area, Total Area, and Power dissipated into registers, and total power was extracted. The scripts and guidelines for the core synthesis are provided in the GitHub repository. Appendix A provides more information about how to synthesize the core.
Overall, the core was simulated and synthesized 128 times during the testing. The simulations required 16 hours, while the synthesis lasted almost 6 days. Python scripts developed by the student automated the test. Figure 3.6.1 portrays a flow diagram for the Python scripts running the simulations and synthesizes. It can be seen from the figure that at the beginning of the testing, the cache size is set to 256 entries of type cache_line_t. Next, the verification environment simulates the core, which generates a UVM log file (figure 3.2.1). The log file extracted after the simulation is compared with the log file extracted after simulating the core without prediction. Next, the script uses those two log files to compare executed instructions and the order in which they are executed. If they are not identical, the core has not executed the program correctly, and the testing stops. Otherwise, information about the maximum number of valid entries in the cache during the

simulation, the number of encountered branch instructions, the number of predictions, and a number of correct and incorrect predictions. The maximum number of valid entries in the cache during the simulation, number of encountered branch instructions, number of predictions, and number of correct and incorrect predictions are extracted from the btb_statistics file generated by the cv32e40x_BTB_BHT module. Next, the script calculates IPC from the UVM log file. After simulation, the core is synthesized, and reports about power consumption and area usage are extracted from the synthesis log. Next, the cache size is decreased by 2. If the size becomes less than 2, the testing stops because we have already simulated/synthesized the core 128 times. Otherwise, the scripts simulate and synthesize the core with a new cache size.

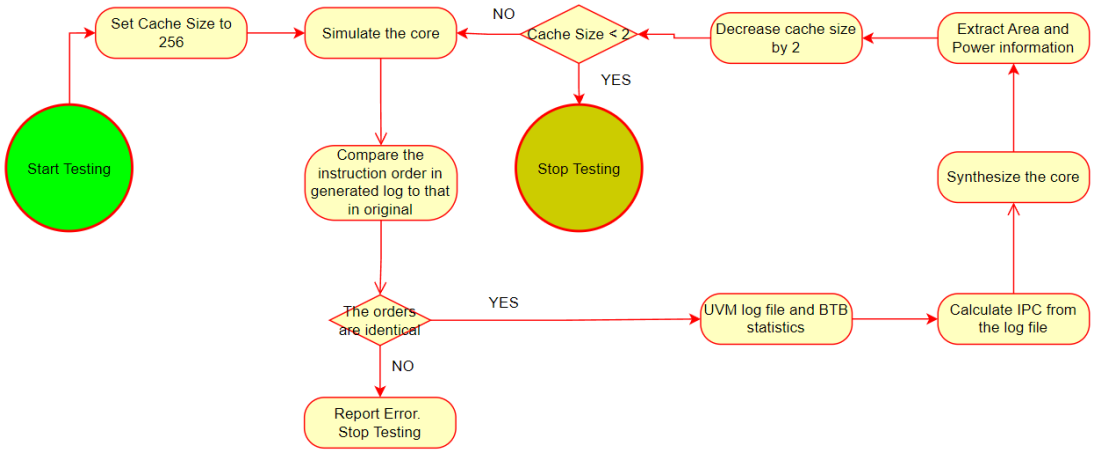


Figure 3.6.1: Flow Diagram of Python Script for Simulation and Synthesis

Overall, throughout the thesis 2-bit branch predictor was implemented in CV32E40X core by adding module cv32e40x_BTB_BHT and modifying modules implementing controller FSM, EX, ID, and IF stages. Furthermore, the core with 128 different cache sizes was simulated and synthesized to extract core speed, power consumption, and area usage information. Appendix A provides a link to the GitHub repository containing RTL files for the core with the predictor and Python scripts used for testing. The appendix also explains how to run simulations and synthesis in detail.

CHAPTER

FOUR

RESULTS

Chapter 4 analyzes information extracted from the tests, states the speed increase measured in ICP, and elaborates on the cost of performance improvement in terms of power consumption and area usage.

4.1 Speed Improvement

As stated in Chapter 1, the thesis aimed to increase the speed of the CV32E40X core by implementing a 2-bit branch predictor without increasing area usage and power consumption by more than 20%. Figure 4.1.1 present IPC, Net Area, Cell Area, Total Area, and Power values for baseline core and the core with 2-bit branch predictor using 8 66-bit long cachelines for implementing BTB_BHT cache.

	IPC	Net Area	Cell Area	Total Area	Power
Baseline Core	0.46	5874.265	16704.66	22578.927	2.70943e-04
Core with Branch Predictor	0.5	6396.825	19153.64	25550.467	2.99153e-04
Increase (%)	8.85%	8.9%	14.7%	13.1%	10.4%

Figure 4.1.1: Results of Implementing 2-bit Branch Predictor with 8 cachelines for BTB_BHT. Speed (IPC), Area, and Power Comparison with the Baseline Performance

It can be seen from the figure that implementing a 2-bit predictor has fulfilled the thesis goal:

- IPC - IPC increased from 0.46 to 0.5, meaning that with a 2-bit predictor, the core can execute 50% of instruction in a single clock cycle, while the baseline core can execute 46%. The speed of the core increased by 8.85%. The reason

for the increase is the idea that branch instructions do not slow down the pipeline significantly, providing that the branch outcome is predicted in the IF stage. It was explained in the State of Art chapter branch instruction slows down the pipeline because the branches are taken from the EX stage, and the controller has to kill instructions in the IF and ID stages. Figure 4.1.2 (analyzed below) shows that the core requires fewer clock cycles to finish the program after the branch predictor has been implemented in it. The speed increase can be explained by the branch predictor predicting the branch outcome at the IF stage, and the controller does not have to kill IF and ID stages as often as the baseline core. Moreover, it was also explained in the State of Art chapter data deluge is one of the important problems embedded systems are facing today, and the speed increase, resulting from implementing branch predictor is a step towards eliminating the gap between the sheer volume of received data from the sensors and processors' ability to analyze them. Hence, the effect of data deluge and branch instruction on the pipeline was counteracted by implementing a 2-bit branch predictor.

- Area - It can also be seen from the figure that the total area increased by 13.1%, which is less than the 20% constraint imposed by Silicon Labs. The main reason for the area increase is adding the BTB_BHT cache to the core. Each cache line is 66-bit long (1bit - VALID, 31bit - TAG, 32bit - Target Address, 2bit - Counter). The implemented branch predictor uses the cache with 8 cachelines. Thus, adding the cache increases the area by 528 bits. Furthermore, the logic for the FSM controller also becomes more complicated, increasing the Cell Area. Finally, the Net Area also increases since the Controller has new input/output signals from/to the IF stage, EX stage, and BTB_BHT cache.

- Power - Similarly to the area, the power consumption increase (10.4%) is less than the 20% constraint. However, based on the value of the power increase, it can be stated that the implementing 2-bit branch predictor unit is a low-power solution to the performance optimization (speed increase) problem. Since implementing a 2-bit branch predictor achieves speed increase without increasing operating frequency, the power consumption does not increase dramatically (Chapter 2, equation 2.1)

Although a 2-bit branch predictor with 8 cachelines is the suggested solution to the performance optimization, 128 cores were simulated and synthesized to observe the change of the performance with respect to the number of cachelines. The number of the cacheline changes from 2 to 256, and each time, the number increases by 2. The rest of the section presents findings obtained by analyzing test results

Each of the simulated cores was tasked with executing the test program with 9929 instructions from the OpenHW group. Figure 4.1.2 portrays two diagrams:

- 4.1.2 Plot 1 - The plot displays how the number of clocks (clk) cycles required for the test program execution changes with respect to the number of the cachelines. For cache with 8 cachelines, the program is executed in 19830 cycles (the point where the yellow line crosses the blue line). Furthermore, it

can also be observed that the increase in the number of cachelines increases the number of required clock cycles. The maximum point occurs when the core has 82 cachelines and requires 20284 cycles to finish the execution. The average number of required clk cycles is 20135.

- 4.1.2 Plot 2 - In addition to all the information displayed in plot 1, plot 2 also portrays the number of clk cycles required by the baseline core without predictor to finish the test program execution (Orange line). The baseline value of clk cycles is 21377. The clk cycles of the cores with predictor are lower than that of the core without predictor, meaning that the former is faster than the latter.

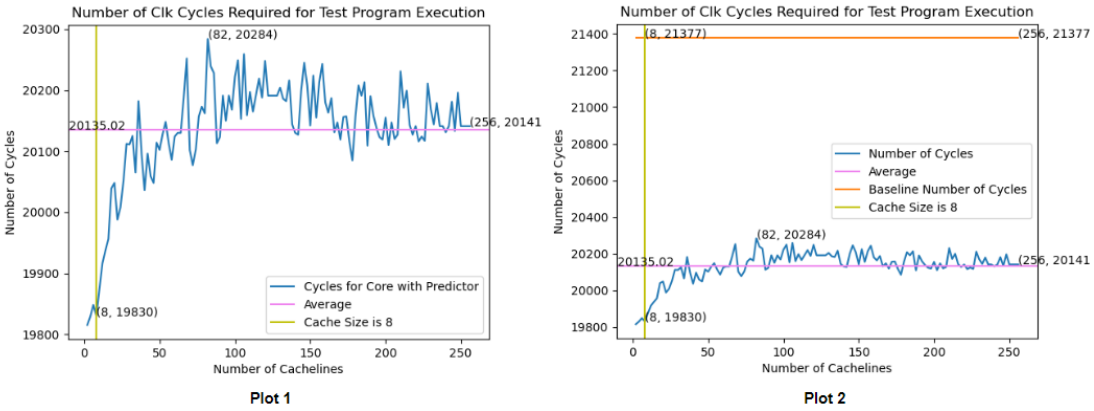


Figure 4.1.2: Number of clk with respect to cache size. Plot 1 - y-axis is numbers, Plot 2 - y-axis is percentages

Figure 4.1.3 portrays three diagrams:

- 4.1.3 Plot 3 - The plot displays how the IPC changes with respect to the number of the cachelines. For a cache with 8 cachelines, the IPC is 0.5 (the point where the yellow line crosses the blue line). Furthermore, it can also be observed that the increase in the number of cachelines decreases the IPC. The minimum point occurs when the core has 82 cachelines, and the IPC is 0.489. The average IPC is 0.49.

- 4.1.3 Plot 4 - In addition to all the information displayed in plot 3, plot 4 also portrays the IPC of the baseline core without predictor (Orange line). The baseline value of IPC is 0.46. The IPC values of the cores with predictor are higher than that of the core without predictor, meaning that the former is faster than the latter.

- 4.1.3 Plot 5 - The plot displays an increase of the IPC compared to the baseline value. For the cache with 8 cachelines, the IPC is increased by 8.85%. The average increase is 7.2%.

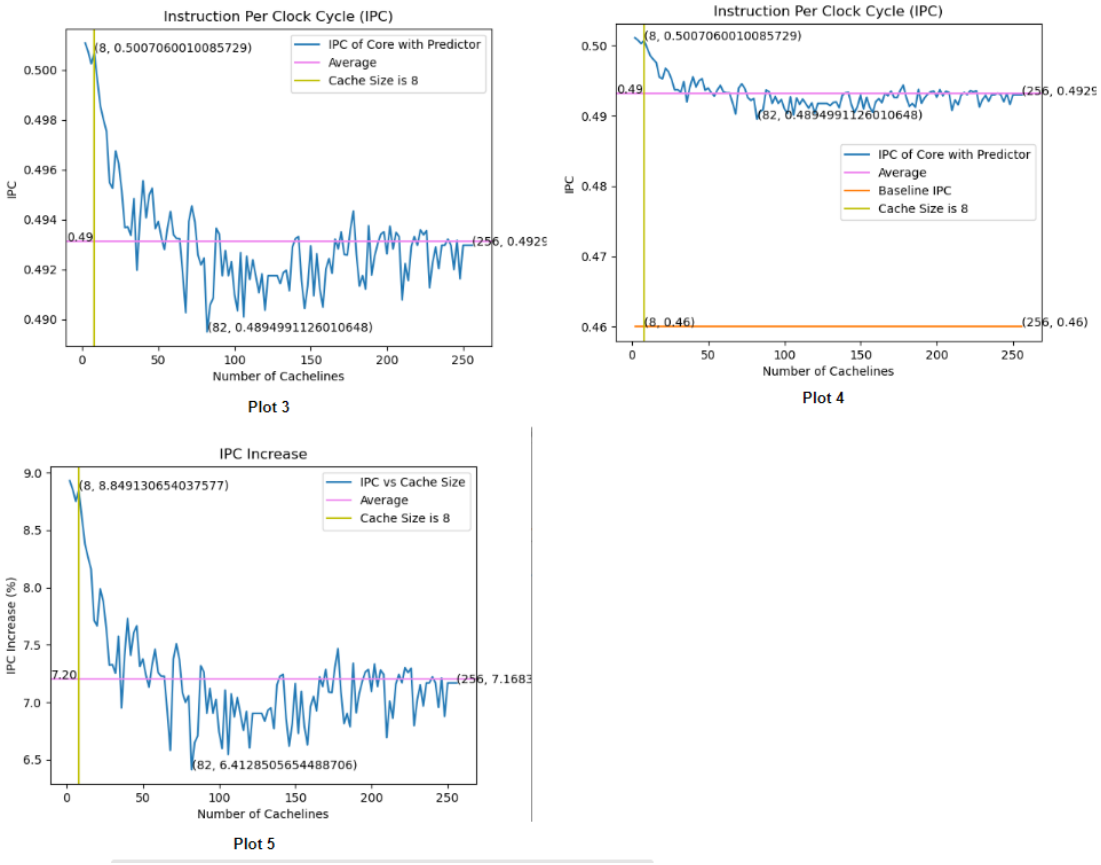


Figure 4.1.3: IPC with respect to cache size. Plot 3-4 - y-axis are IPC; Plot 5 - Y-axis are percentages

Based on diagrams displayed by figure 4.1.3, one can conclude that the increase in the number of cache lines used to implement BTB_BHT leads to the speed decrease. This can be explained by analyzing the data in figure 4.1.4. The larger the cache size, the more predictions the core makes. The more predictions lead to a higher number of mispredictions. Hence, increasing the cache size also increases the number of mispredictions, and the processor has to spend more time on execution order recovery. Therefore, the increase in the number of cachelines slows down the processor. The dependency between the number of predictions and cache size can be explored by analyzing figure 4.1.4:

- 4.1.4 Plot 6 - The plot displays how the number of predictions changes with respect to the number of the cachelines. It can be seen that the core with 8 cachelines makes 777 (the point where the yellow line crosses the blue line). Furthermore, it can also be observed that the increase in the number of cachelines increases the number of predictions. The maximum point (1370) occurs when the core has 256 cachelines. The average number of predictions is 1203.

- 4.1.4 Plot 7 - The plot displays the same information as Plot 6. The difference is that the Y-axis of plot 6 numbers (predictions) while the Y-axis of plot 7 is percentages representing how many percentages of branches were predicted. The core with 8 cachelines predicted 41.85% of branches, while the core with 256 cachelines - was 75.44 %.

- 4.1.4 Plot 8 - The plot portrays how the number of correct predictions (blue line) and the number of incorrect predictions (orange line) increase with respect to cache size. The average value of correct predictions is 979, and incorrect predictions - 223.

- 4.1.4 Plot 9 - Plot 9 portrays the same information as Plot 8. The difference is that the Y-axis of plot 8 numbers (predictions) while the Y-axis of plot 9 is percentages representing how many percent of predicted branches were correct and incorrect. As we can see from plot 9, the increase in the number of cachelines increases the percentage of incorrect predictions and decreases the percentage of correct predictions. The core with 8 cachelines predicts 89% of predictions correctly and 11% of the predictions incorrectly, while the core with 256 cachelines predicts 81% of the predictions correctly and 19% of the predictions incorrectly. Hence, the increase in the cache size generates a decrease in the accuracy of the predictor, which explains why the speed decreases as the number of cachelines increase (plots 1-5).

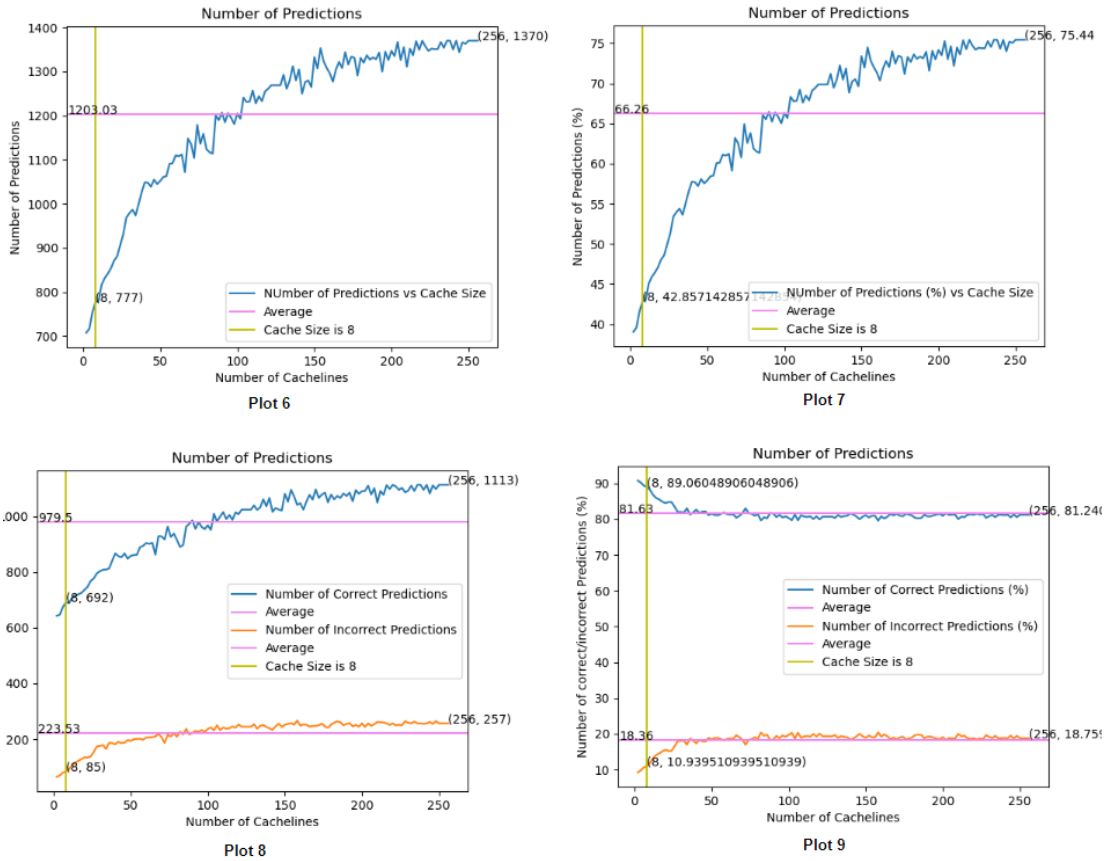


Figure 4.1.4: Number of predictions with respect to cache size. Plot 6 - number of predictions vs. cache size; Plot 7 - same as plot 6, Y-axis is percentages; Plot 8 - number of correct/incorrect predictions vs. cache size; Plot 9 - same as plot 8, Y-axis is percentages

Figure 4.1.5 presents the effect change in the number of cacheline has on the cache usage:

- 4.1.5 Plot 10 - The plot displays how the number of used cachelines changes with respect to the change in the cache size. For the core with 8 cachelines, the number of used cachelines is 8 (the point where the yellow line crosses the blue line). Furthermore, the number of used cachelines for the core with 256 cachelines is 144. The average value is 91.

- 4.1.5 Plot 11 - The plot displays how the percentage of used (blue line) and unused cache changes with increased cache size. IT shows that the core with 8 cachelines uses 100% of the cache while the core with 256 cachelines uses only 56.25%. Hence, increasing the cache size increases the fraction of the unused cachelines and decreases the used ones.

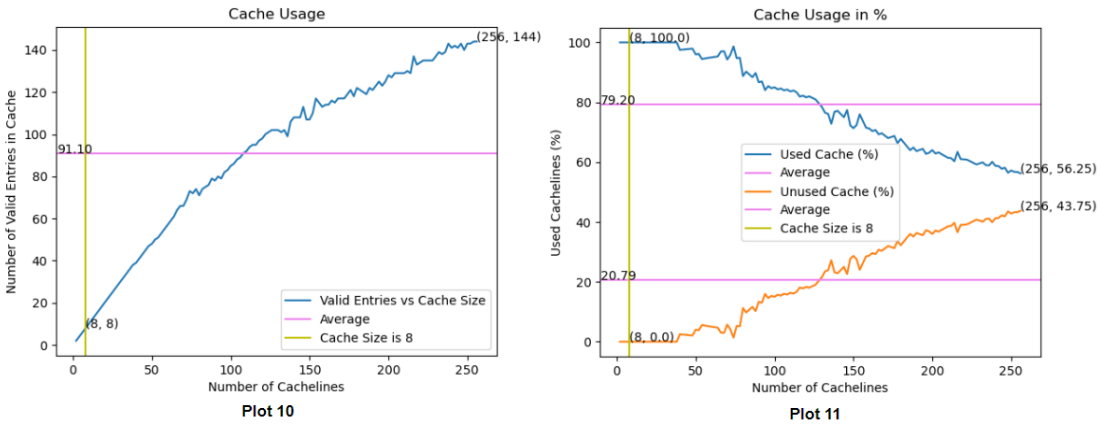


Figure 4.1.5: Cache usage with respect to cache size. Plot 10 - number of used cachelines vs cache size; Plot 11 - number of used/unused cachelines, Y-axis is percentages;

Therefore, it can be deduced that implementing a branch predictor increases the speed because it makes the prediction about branch out in the IF stage and prevents the controller from wasting two clk cycles on execution order recovery. However, increasing the cache size makes the predictor less accurate and increases the fraction of the cache that remains unused throughout the program execution, which means that we are just increasing the area without any speed improvement. Therefore, it is a good idea to implement a cache with only 8 cachelines. In this case, 100% of the cache is used, increasing speed by 8.85%.

4.2 Drawbacks

The speed increase by implementing branch predictor comes at the cost of increasing area usage and power consumption. Figure 4.2.1 displays area increase compared to baseline:

- Figure 4.2.1 Plot 12 - The plot displays how net, cell, and total area increase as the cache size increases.

- Figure 4.2.1 Plot 13 - The plot represents the increase in cell area with respect to cache size. The plot also portrays the baseline value of the cell area. It can be observed that the core with 8 cachelines uses 19153.642 cell

area, which is more than the baseline value (16704.66) by 14.7%. The cell area for all 128 values is 53733.

- Figure 4.2.1 Plot 14 - The plot represents the increase in net area with respect to cache size. The plot also portrays the baseline value of the net area. It can be observed that the core with 8 cachelines uses 6396.825 net area, which is more than the baseline value (5874.265) by 8.9%. The cell area for all 128 values is 12952.

- Figure 4.2.1 Plot 15- The plot represents the increase in total area with respect to cache size. The plot also portrays the baseline value of the total area. It can be observed that the core with 8 cachelines uses 25550.467 total area, which is more than the baseline value (22578.927) by 13.1%. The total area for all 128 values is 66685.

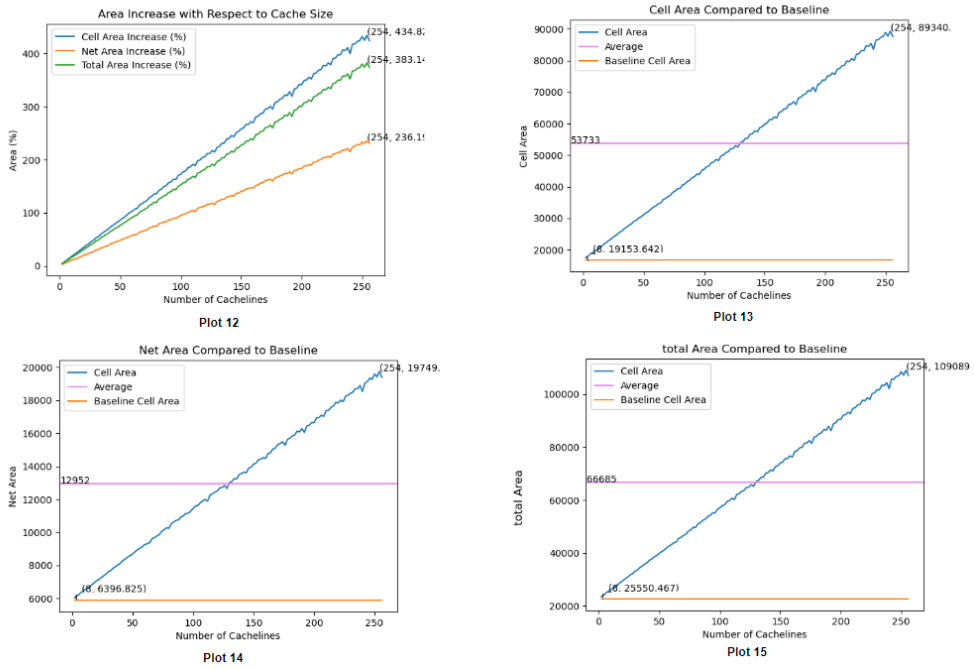


Figure 4.2.1: Plot 12 - total, cell, net area vs. cache size; Plot 13 - Cell area and Baseline value of Cell area; Plot 14 - Net area and Baseline value of Net area; Plot 15 - Total area and Baseline value of Total area

Figure 4.2.2 displays power increase compared to baseline:

- Figure 4.2.2 Plot 16 - The plot explains how the total power and the power used by registers increase with the cache size increase. The core with 8 cachelines is characterized by 299 MicroW total power and 137 MicroW register power.

- Figure 4.2.2 Plot 17 - The plot portrays how the fraction of total power used by registers changes as the cache size increases. It can be observed that the registers of the core with 8 cachelines use 45.6% of the total power. As the cache size increases, so does the power used by the registers. However, the rate of power increase decreases as the cache sizes become larger than 150 cachelines. This can be explained by the fact that alongside the cache increase, the fraction of cachelines used by the core decreases (figure 4.1.5 plot 11).
- Figure 4.2.2 Plot 18 - The plot elaborates on how the percent increase of total, register, and the fraction of power used by registers increases with respect to cache size. It can be observed that the core with 254 cachelines uses more power than the core without predictor.

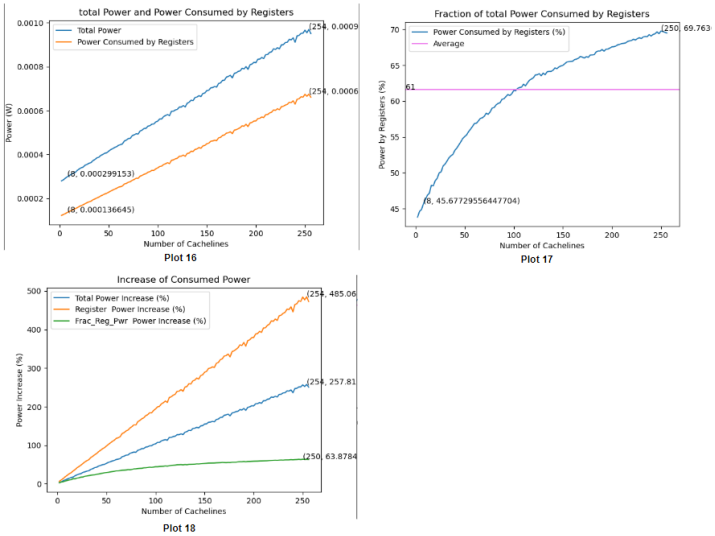


Figure 4.2.2: Plot 16 - total power, register power, a fraction of total power used by register vs. cache size; Plot 17 - a fraction used by register vs. cache size. Y-axis is percentages; Plot 18 - an increase of Total, Register power, and fraction of total power used by registers compared to the baseline values. Y-axis is percentages;

Overall, implementing the 2-bit branch predictor with 8 cachelines increased the speed of the CV32E40X processor. However, the speed improvement came at the cost of power and area increase. Furthermore, it was also discovered that the increase in the number of cachelines reduces the predictor efficiency and increases the number of unused caches.

CHAPTER

FIVE

CONCLUSIONS

The main goal of the thesis is to increase the speed of the CV32E40X processor, developed by the openHW group, without increasing the area usage and power consumption by more than 20%. The metric used for speed evaluation is Instruction Per Clock Cycle (IPC); for the area usage - net area, cell area, and total area; for the power consumption - Watts (W). The thesis addresses how implementing a 2-bit branch prediction unit can improve the core speed and cost of the improvement in terms of power consumption and area usage. The results revealed that the 2-bit branch predictor with a cache with 8 cachelines increases IPC by 8.7%, costing 8.9% net area, 14.7% cell area, 13.1% total area, and 10.4% total power increase.

The results revealed that, since the branch predictor predicts the branch outcome at the IF stage, the controller does not have to kill IF and ID stages as often as the baseline core, resulting in the speed increase. In conclusion, implementing a 2-bit branch predictor in the CV32E40X results in an 8.85% speed increase at 10.4% power and a 13.1% area increase.

5.1 Future Work

The thesis explored how a 2-bit predictor affects the speed, area usage, and power consumption of the CV32E40X core. Implementing more than one predictor with a selector unit that selects between several predictions can be an interesting research area. Such a design may generate a bigger speed increase.

Furthermore, it will be interesting to explore how the size change of each cacheline of a 2-bit predictor affects the prediction accuracy. The predictor designed for the thesis uses a cache with 66-bit long cachelines. Each cacheline has a TAG section storing PC[31:1] bits of the program counter associated with a branch instruction. The cache is indexed by the 3 least significant bits of the PC, and we do not have a cache hit unless PC[31:1] equals 31 bit-long TAG. Reducing the tag size can drastically reduce the area used by the cache. Investigating how the change in the TAG size affects prediction accuracy will also be an interesting research area.

REFERENCES

[1] G. Solomnishvili. "*Low Power Solution to Performance Optimization*". In: (Dec. 2022). URL: https://github.com/Giorgi-Solo/Specialisation-Project-TFE4580/blob/main/Report.pdf.

[2] M. Hilbert. "*Global information Explosion*". In: *Elsevier* (2015). URL: https://canvas.instructure.com/courses/949415/pages/9th-week-digital-divide?module_item_id=7573579.

[3] G. MOORE. "*Cramming more components onto integrated circuits.*" In: *Electronics* (Apr. 1965). URL: https://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf.

[4] N Wehn. "*Design of Microelectronic Systems Circuits.* " In: *Lecture, Technical University of Kaiserslautern* (2021).

[5] M. Jung. "*SystemC and Virtual Prototyping*". In: *Lecture, Technical University of Kaiserslautern* (2021).

[6] D. M. Harris. "*Integrated Circuit Design*". In: *Boston: Pearson* (2011).

[7] A Ray. "*Branch prediction for a RISC-V processor core*". In: *Department of Electronics and Telecommunications, NTNU - Norwegian University of Science and Technology* (2022). URL: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/3022383/no.ntnu%5C%3ainspera%5C%3a106811575%5C%3a21626164.pdf?sequence=1&isAllowed=y.

[8] J. L. Hennessy, D. A. Patterson, and K. Asanovic. "Computer Architecture: A quantitative approach". In: (Dec. 1998), p. 4.

[9] D Stoffel. "*Architecture of Digital Systems II*". In: *Lecture, Technical University of Kaiserslautern* (2021).

[10] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The hardware/software interface*. 1998.

[11] Azeris. "*Low Power Solution to Performance Optimization*". In: (Dec. 2022). URL: https://azeria-labs.com/arm-conditional-execution-and-branching-part-6/.

[12] J. C. Smith. "*A study of branch prediction strategies*". In: (). URL: https://courses.cs.washington.edu/courses/cse590g/04sp/Smith-1981-A-Study-of-Branch-Prediction-Strategies.pdf.

[13] U. Hoffman. "*Saturation Arithmetic*". In: (2014). URL: http://www.euroforth.org/ef14/papers/hoffmann.pdf.

[14] Traber A. et al. "*CV32E40X*". In: OpenHW Group. URL: https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/index.html.

[15] OpenHW Group. "*OpenHW Group CORE-V CV32E40X RISC-V IP*". In: (2014). URL: https://github.com/openhwgroup/cv32e40x.

[16] University of Maryland Baltimore County Computer Science and Electrical Engineering Departmen. "*CMSC 411 Lecture 19, Pipelining Data Forwarding* ". In: (2014). URL: https://www.cs.uaf.edu/2011/fall/cs441/lecture/09_20_pipelining.html.

[17] OpenHW Group. "*Core-V-Verif* ". In: (2014). URL: https://github.com/openhwgroup/core-v-verif.

[18] K. Hwang, N. Jotwani, and Dongarra. "*Advanced Computer Architecture: Parallelism, Scalability, Programmability* ". In: (2014). URL: https://www.cs.uaf.edu/2011/fall/cs441/lecture/09_20_pipelining.html.

[19] Cadence Design Systems. "*Genus User Guide for Legacy UI*". In: (June 2018). URL: https://picture.iczhiku.com/resource/eetop/wYIDduQHAklkSxcN.pdf.

[20] OpenHW Group. "*CORE-V-VERIF Quick Start Guide*". In: OpenHW Group. URL: https://docs.openhwgroup.org/projects/core-v-verif/en/latest/quick_start.html.

[21] EECS-NTNU Group. "*WiKi*". In: EECS-NTNU Group.

APPENDICES

A - GITHUB REPOSITORY

The SystemVerilog code for CV32E40X core with 2-bit branch predictor unit and Python scripts used to run the simulation, synthesis, and extract and analyze data can be seen in the following GitHub Repository:

Github repository link

- `https://github.com/Giorgi-Solo/MasterThesis/tree/master`

The repository's content has also been submitted to NTNU Insperra as a .zip file. When the users clone this directory, they **MUST** clone it **RECURSIVELY**.

The rest of Appendix A explains the repository structure and gives the curious readers guidelines about setting up the simulation/synthesis environment, running the branch predictor model developed for the specialization project, simulating/synthesizing core, and extracting and analyzing simulation/synthesis data.

In order to simulate, synthesize, automatic place and routing, and manual place and routing the user needs to have access to the following tools and technology library:

- Simulation - QuestaSim
- Synthesis - Cadence Genus
- Automatic Place-and-route++ - Cadence Innovus
- Manual Place-and-route - Cadence Virtuoso
- STM 28nm technology library

.1 Structure of Repository

MasterThesis repository consists of a README file, and two folders - Master and Workplace.

.1.1 Master

The folder contains another GitHub repository named CV32E40X (link to the repository - https://github.com/Giorgi-Solo/cv32e40x). The CV32E40X repository is forked from openHW group's repository (the baseline core - without branch predictor). I implemented the 2-bit branch predictor unit with 8 cachelines in the core from the forked repository. Figure .1.1 displays the fraction of the RTL module tree which implements a 2-bit branch predictor and misprediction recovery unit.

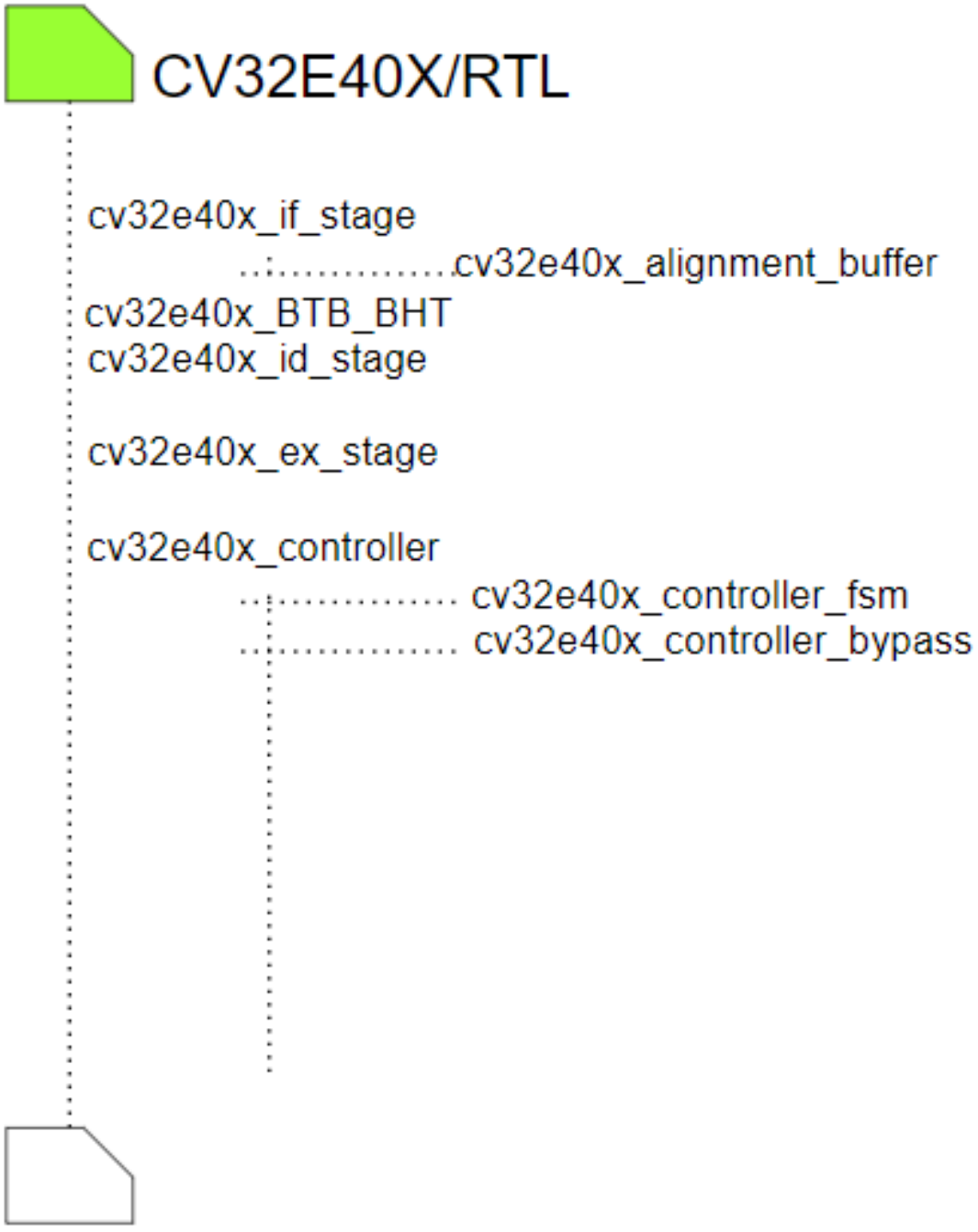


Figure .1.1: Code Tree of CV32E40X Core with 2-bit Branch Predictor

.1.2 Workplace

The folder contains the following Python scripts:

- logParser.py - Implements branch predictor model;
- sim.py - Simulates 128 core versions. The difference between each of the versions is the cache size. The script also extracts information about the maximum number of valid entries in the cache during the simulation, the number of encountered branch instructions, the number of predictions, and the number of correct and incorrect predictions (btb_statistics). Finally, the script runs the test program on each of the 128 versions of the core and extracts the UVM log file;
- sim_analyzer.py - Extracts information about the number of cycles required for each of the cores to finish the test program, clock cycle per instruction (CPI), number of valid entries in the cache, number of branch instructions, number of correct/incorrect predictions, and number of branch predictions. The script also calculates the CPI improvement, what fraction of the cache used by the predictor is, and what fraction of the predictions were correct/incorrect.
- sim_data_analyzer.py - The script analyzes simulation data extracted by sim_analyzer.py script and generates graphs used in chapter 4 (figures 4.1.2 - 4.1.5).
- synth.py - Synthesizes 128 core versions. The difference between each of the versions is the cache size. The script also extracts synthesis reports about area usage and power consumption.
- syn_analyzer.py - The script parses the synthesis reports and extracts information about cell area, net area, total area, total power, and power used by the registers. The script also calculates what fraction of total power is used by the registers
- syn_data_analyzer - The script analyzes synthesis data extracted by syn_analyzer.py and generates graphs used in chapter 4 (figures 4.2.1 - 4.2.2).
- show_log.py - The script displays information about errors and warnings encountered while running the previous six Python scripts

Workplace contains two folders:

- simulations - The folder stores all the simulation data extracted by sim.py and sim_analyzer.py. The folder stored all the synthesis report and synthesis data extracted by synth.py and syn_analyzer.py. The folder also contains a makefile used to simulate or synthesize the core.
- tcl - Contains .tcl codes that need to be added to .tcl scripts in the synthesis environment (explained later).

Finally, the workplace contains a makefile that is able to clone simulation and synthesis environments, run all the mentioned Python scripts, and erase generated logs.

.2 Running Model

In order to run the model, first, the user needs to clone the MasterThesis repository from the GitHub Repository Link, presented at the beginning of the appendix.

Next, the user needs to run a makefile target named "model" from the workplace directory. The target contains the following two commands:

- @echo "Running Model of the core th Predictor" - prints the message into the console.
- @python3 logParser.py - runs logParser.py script.

For more information about the output generated by logParser.py, refer to the specialization project report[1].

.3 Setting up Simulation, Simulating the Core, Analyzing Simulation Data

.3.1 Setting up Simulation

In order to simulate the core, the users need to clone the simulation/verification environment developed by openHW group. The simulation environment can be cloned from the following Github Repository - https://github.com/openhwgroup/core-v-verif. This repository contains a folder Core-V-Verif which represents the verification environment. More information about setting up the environment can be found in the article "CORE-V-VERIF Quick Start Guide" provided by openHW Group[20].

Core-V-Verif can be cloned by running a makefile target named clone_sim_env from the workplace directory. The target contains the following two commands:

- @echo "Simulation/Verification environment is being cloned from openhw-group github public repository" - prints the message into the console.
- @cd ../ && git clone https://github.com/openhwgroup/core-v-verif.git - clones Core-V-Verif and ensures that Workplace, Master, and Core-V-Verif are at the same level in the directory tree.

.3.2 Simulating the Core and Analyze Simulation Data

In order to simulate the core, the users need to run a makefile target named sim from the workplace directory. The target contains the following five commands:

- @echo "Makefile from workplace is running simulations" - prints message to the console.
- @python3 sim.py - runs sim.py script, which extracts btb statistics and UVM log files.

- @echo "Running simulations analysis" - prints message to the core.
- @python3 sim_analyzer.py - runs sim_analyzer.py script, which analyzes simulation data.
- @cd ../master/cv32e40x/rtl/ && git stash && git stash drop - Running sim.py modifies cache size by changing the control variable - size, in cv32e40x_if_stage module. This command restores the cache size to 8 cachelines.

.3.3 Plot the Analyzed Simulation Data

In order to plot the analyzed simulation data (figures 4.1.2 - 4.1.5), the users need to run a makefile target named sim_data_analyzer from the workplace directory. The target contains the following two commands:

- @echo "Running simulation data analyzer" - prints message to the console.
- @python3 sim_data_analyzer.py - runs sim_data_analyzer.py script.

.4 Setting up Synthesis, Synthesizing Core, Analyzing Synthesis Data

Before we discuss synthesizing the design, the users must comment lines from cv32e40x_BTB_BHT module that store btb statistics in btb_statistics.txt file. Figure .4.1 displays SystemVreilog ALWAYS_FF (Line 104-127) AND INITIAL (Lines 129-153) blocks. These blocks need to be removed from the module before the user tries to synthesize the design.

```
102     // THIS PART IS FOR THE SIMULATION ONLY. COMMENT IT OUT FOR SYNTHESIS
103
104 >   always_ff @(posedge clk or negedge rst_n) begin···
127     end
128
129 >   initial begin···
153     end
154
```

Figure .4.1: cv32e40x_BTB_BHT module: Lines 104-153 store btb statistics into btb_statistics.txt file.

.4.1 Setting up Synthesis

In order to synthesize the core, the users need to clone the synthesis environment developed by EECS-NTNU. The synthesis environment can be cloned from the following Github Repository - https://github.com/EECS-NTNU/asic-flow. Note that this is a private repository, and the users will need permission to access it. This repository contains a folder asic-flow which contains tcl scripts for setting up the synthesis environment. More information about setting up the environment

can be found in WiKi section of the asic-flow GitHub repository[21].

asic-flow can be cloned by running a makefile target named clone_synth_env from the workplace directory. The target contains the following two commands:

- @echo "Synthesis environment is being cloned from NTNU asic-flow github private Repository" - prints message to the console.
- @cd ../ && git clone https://github.com/EECS-NTNU/asic-flow.git - clones the repository and ensures that Workplace, Master, Core-V-Verif, and asic-flow are at the same level in the directory tree.

Before using the repository, we need to modify two files: asic-flow/stm28/counter/config_syn.tcl and asic-flow/stm28/counter/tcl /synth.tcl.

.4.1.1 Modificatoins to asic-flow/stm28/counter/tcl /synth.tcl

The user should delete line number 6 in asic-flow/stm28/counter/tcl/synth.tcl file and replace it with the content of MasterThesis/workplace/tcl /config_syn.tcl

.4.1.2 Modifications to asic-flow/stm28/counter/tcl /synth.tcl.

The user should replace lines 70-80 from asic-flow/stm28/counter/tcl /synth.tcl with lines from figure .4.2.

```
# Read source files

read_hdl -sv -lib "cv32e40x_pkg" $cv32e40x_pkg
foreach x $VERILOG_SOURCE_LIST {
    read_hdl -sv -library "cv32e40x_pkg" $x
}

foreach x $VHDL_SOURCE_LIST {
    read_hdl -language vhdl $x
}

# Elaborate
elaborate "cv32e40x_core"
```

Figure .4.2: TCL script reading core modules. These lines can be found in MasterThesis/workplace/tcl /synth.tcl file

Furthermore, the user needs to replace lines 95-97 from asic-flow/stm28/counter/tcl/synth.tcl with lines 21-84 from MasterThesis/workplace/tcl/synth.tcl

Finally, the user needs to add "quit" command at the end of asic-flow/stm28/counter/tcl/synth.tcl

.4.2 Synthesizing Core and Analyzing Synthesis Data

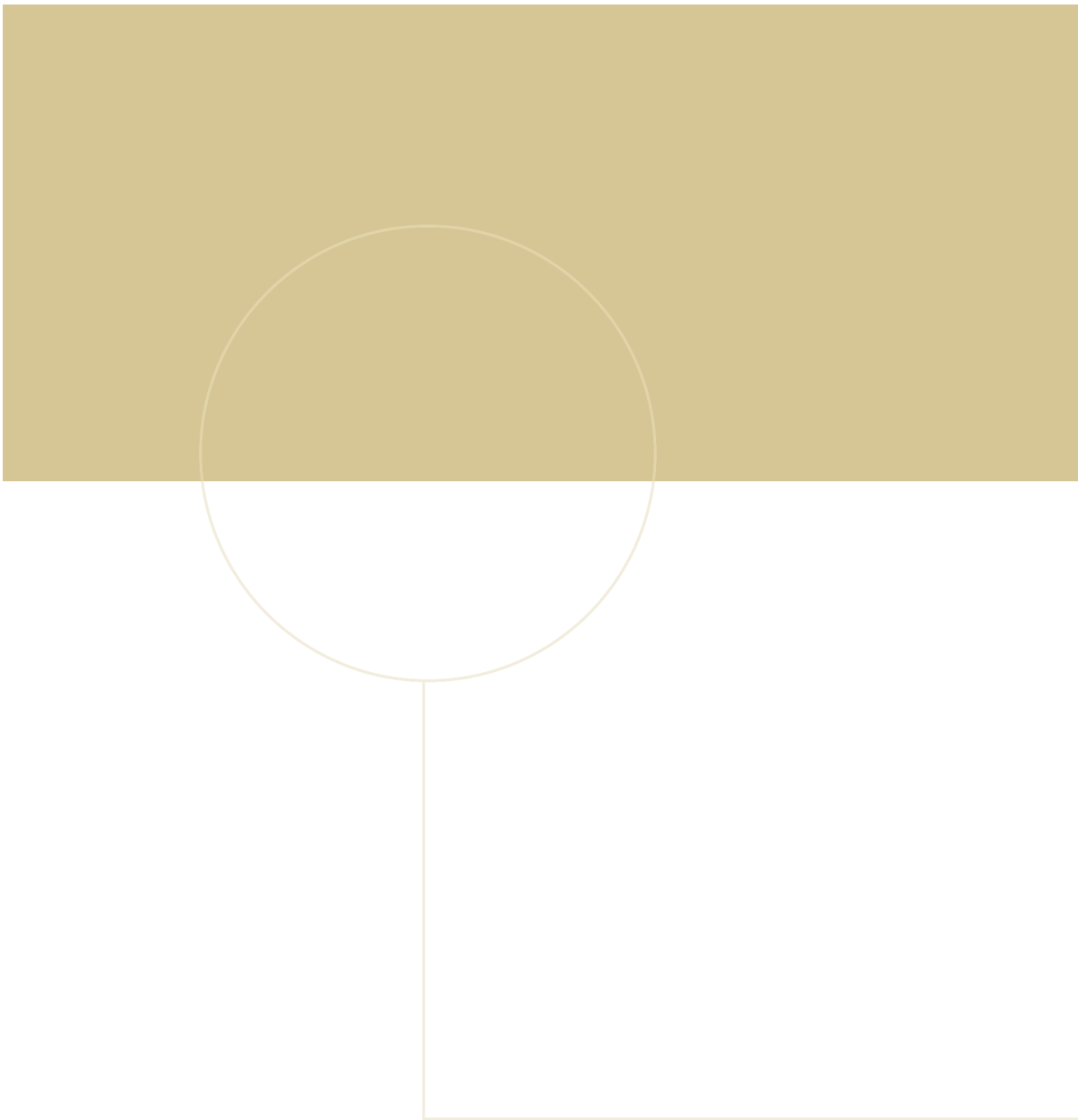
In order to synthesize the core, the users need to run a makefile target named syn from the workplace directory. The target contains the following five commands:

- @echo "Makefile from workplace is running synthesis" - prints message to the console.
- @python3 synth.py - runst synth.py script.
- @echo "Running synthesis analysis" - prints message to the console.
- @python3 syn_analyzer.py - runs syn_analyzer.py script.
- @cd ../master/cv32e40x/rtl/ && git stash && git stash drop - Running syn.py modifies cache size by changing the control variable - size, in cv32e40x_if_stage module. This command restores the cache size to 8 cachelines. Furthermore, cv32e40x_BTB_BHT module is also restored, and ALWAYS_FF and INITIAL blocs from figure .4.1 are uncommented.

.4.3 Plot the Analyzed Synthesis Data

In order to plot the analyzed synthesis data (figures 4.2.1 - 4.2.2), the users need to run a makefile target named syn_data_analyzer from the workplace directory. The target contains the following two commands:

- @echo "Running synthesis data analyzer" - prints the message to the console.
- @python3 syn_data_analyzer.py - runs syn_data_analyzer.py script



NTNU

Norwegian University of
Science and Technology