Master's thesis

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

Ray, Achyut

Branch prediction for a RISC-V processor core

Master's thesis in Electronic Systems Design
Supervisor: Kjeldsberg, Per Gunnar
Co-supervisor: Knauserud, Øystein
June 2022

NTNU
Norwegian University of Science and Technology

Ray, Achyut

Branch prediction for a RISC-V processor core

Master's thesis in Electronic Systems Design
Supervisor: Kjeldsberg, Per Gunnar
Co-supervisor: Knauserud, Øystein
June 2022

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

NTNU
Norwegian University of
Science and Technology

Achyut Ray

Branch prediction for a RISC-V processor core

Master thesis
for the degree MSc in Embedded Systems, Electronics

Trondheim, June 2022

Norwegian University of Science and Technology

Department of Electronics and Telecommunications

NTNU
Innovation and Creativity

NTNU

Norwegian University of Science and Technology

Master thesis
for the degree of MSc in Embedded Systems, Electronics

Department of Electronics and Telecommunications

Abstract

A processor's performance is measured using metrics of speed and accuracy. These are, however, not independent of one another. The more accurately a processor does what it is supposed to, the less time is spent correcting mistakes, in turn increasing its speed. An inevitable source of inaccuracy involves branch instructions. Not knowing whether a branch is going to be executed or not before it needs to be executed means that a considerable amount of clock cycles are wasted doing incorrect things. This can be avoided using a technique called Branch Prediction. This thesis focuses on understanding the ®CV32E40X processor core, and exploring how the branch prediction will affect the performance, area, and power consumption for the processor core. The Backwards Branch Predictor was implemented based on the models made for the course TFE4580: Electronic Systems Design and Innovation, Specialization Project. The thesis builds upon the work done in the previous course. This thesis provides corrected results for the modeling done for the Specialization Project, as these results were central for the scope of this thesis. The thesis also presents Further Work that can be done to further investigate and potentially improve the performance of the ®CV32E40X processor core. It was found that implementing a backwards predictor allowed for decreasing the number of cycles consumed per benchmark test by about 0.9% in the worst case, and 4.5% for the best case.

Preface

I would like to thank my professor and superviser Per Gunnar Kjeldsberg for being a very supportive mentor throughout this project. Thank you Mr. Kjeldsberg for providing great feedback and constructive criticism to improve my work. I would also like to thank Øystein Knauserud, my external supervisor, for providing me with extensive debugging tips, and educating me on the RISC-V architecture. Mr. Knauserud played a vital role in helping me with getting the simulations for the experimentation for this thesis to work. This thesis builds upon the work done in the Specialization course carried out during the Autumn of 2021. The results from that report needed to be corrected for the scope of this thesis. For helping me sanity check my results, Mr. Knauserud has been a true life saver. For this, I am extremely grateful. Lastly, I would like to thank all my friends and family who have made an effort to motivate me throughout the course of this thesis.

Contents

List of Tables

List of Figures

Source code

List of abbreviations

ADC Analogue to Digital Converter

CPU Central Processing Unit

FPGA Field Programmable Gate Array

RISC Reduced Instruction Set Computer

CISC Complex Instruction Set Computer

ISA Instruction Set Architecture

RTL Register Transfer Language

Chapter 1

Introduction

Increase in demand for high-performance Central Processing Units (CPUs) for modern day applications demands two things: developing faster processing architecture, and making the existing architecture faster [Mor]. Most existing CPUs employ a technique called instruction-pipelining to increase the speed of execution and throughput [JLH11]. This thesis builds upon the results and findings of the project work completed as part of the course TFE4580: Electronic Systems Design and Innovation, Specialization Project. The project work from that course was documented in Specialization Project Report. Since this thesis builds directly upon the Specialization Project Report, the term "Project Report" will be used, whenever referring to the Specialization Project Report. The Project Report [Ray21] provides a deeper understanding surrounding the workings of a Branch Predictor. In addition to this, effects of implementing different branch-prediction strategies on a RISC-V CPU core, namely the CV32E40X, were also modeled.

Chapter 1 will serve as an introductory chapter for this thesis providing the following information:

- Overview of the report structure
- Introduction to concepts vital for understanding the contents of this report
- Quick run-down of the work done during the course of this project.

1.1 Report structure

This report is structured to conform with the following hierarchy:

- **Introduction: Chapter 1**
- **Background and Theory: Chapter 2**
  Chapter 2 is intended to present a formal discussion about important concepts about things like Branch Prediction, it's effects on an instruction pipeline, and various branch prediction algorithms.
- **Correction of Branch Prediction models: Chapter 3**
  The results in the Project Report from last semester, had some issues in the presented results. This chapter explains why it was necessary to fix them, and how the data will be used in making design decisions for the scope of this thesis.

- **CV32E40X processor core and intended branch predictor design: Chapter 4**
  This chapter shall focus on introducing the CV32E40X processor core, giving an insight to its pipeline architecture, and providing a suggested design for the Backwards Branch Predictor.
- **Implementation of the Backwards Predictor: Chapter 6**
  This chapter documents the process of trying to implement the Backwards Predictor on the CV32E40X core. This shall also include details about why the implementation turned out to be the way it did.
- **Results: Chapter 7**
  Lastly, this chapter aims to entail results of running the benchmarking tests on the CV32E40X processor core with and without the branch predictor. This section also includes a comparison where the results of the modeling from the Project Report and the experimental results from the implementation are compared.
-

1.2 Introduction to vital concepts

Conforming to the aforementioned structure, this provides a brief introduction to concepts that are vital for understanding the upcoming content in this thesis. Some of the the concepts are identical to the ones mentioned in the Project Report, but are mentioned here regardless for the sake of completion.

Instruction-pipelining

A modern processor deals with instructions by dividing them up into different stages [JLH11]. Each of these stages have a specific task. In order to perform an operation, an instruction is fetched into the CPU from the program memory using the Fetch circuit. To understand what this instruction means, it is then decoded, or deciphered, using the decode circuit. The execute circuit then carries out the requested action. Upon completion, the results are written-back to a register or another memory location [Ray21]. This is merely an example of implementation of an instruction pipeline. Contemporary processors have pipelines that are normally between 3 and 5 stages [JLH11]. Without the use of an instruction pipeline, the CPU can only use one of these circuits at any given point in time. This creates the problem of unused circuitry which negatively affects the performance of the CPU[JLH11]. Instruction-pipelining is a technique which tries to keep each of these circuits active concurrently. It is a way for the processor to achieve instruction level parallelism [JLH11]. The term pipeline is used to refer to the meta-circuit that is achieved when the aforementioned stages are put together. In this thesis, the processor core to be introduced shortly, has a 4 stage pipeline, the workings of which is identical to the explanation mentioned above.

Each stage is implemented using a dedicated circuit. As seen above, the following are the circuits used to implement specific stages of the pipeline: Fetch circuit, Decode circuit, Execute circuit, and the Write-back circuit. Figure 1.1 explains the same.

As can be seen in Figure 1.1, for every clock cycle after the first one, multiple circuits inside the CPU are active. For instance, during clock cycle 2, the processor is using the decode circuit to decode instruction 1, while simultaneously using the fetch circuit to fetch instruction 2. This allows for greater throughput and higher speeds at which the processor can carry out tasks [Ray21].

	##1	##2	##3	##4	##5	##6
Fetch	Ins1	Ins2	Ins3	Ins4	Ins5	Ins6
Decode		Ins1	Ins2	Ins3	Ins4	Ins5
Execute			Ins1	Ins2	Ins3	Ins4
Writeback				Ins1	Ins2	Ins3

Figure 1.1: This figure demonstrates how a simple pipelined processor works

Branch Instructions

A branch is an instruction in a computer program that can cause a computer to begin executing a different instruction sequence, and thus deviate from its default behavior of executing instructions in order, at least conceptually. [Brac]

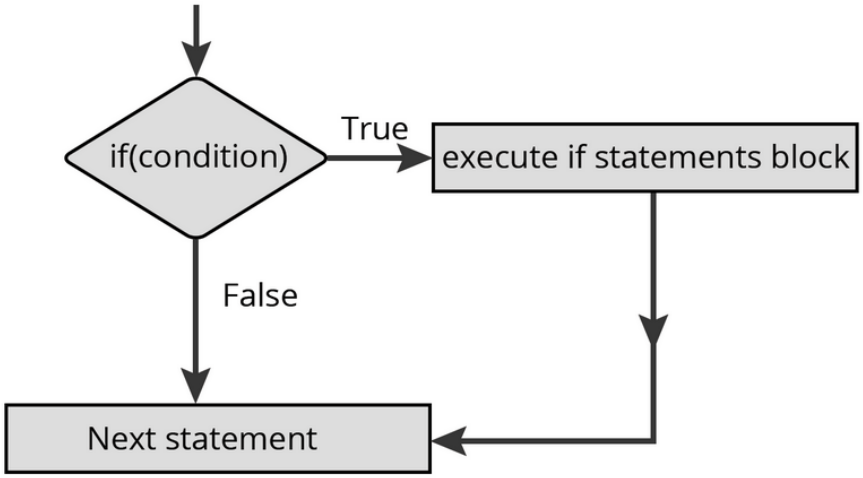


Figure 1.2: This figure demonstrates a simple branch instruction [Brac]

In the Figure 1.2, the effect of a simple branch instruction on the flow of instruction is shown. From this figure, assuming that in most cases the outcome of the if(condition) is false, Next Statement block is the block is next instruction block to be executed. When, however, the if(condition) is true, the execution sequence changes, in the sense that the Next Statement is executed after the instruction sequence in the code block of the if() statement [Ray21].

Branch Predictors

A branch predictor is a digital circuit whose main aim is to predict the outcome of a branch instruction before this is known definitively. The decision concerning whether a branch is to be taken, or not, is available after the condition for the branch has been evaluated. This evaluation happens in the execute stage [Brac]. By making a prediction about the outcome of the branch, the CPU can start fetching the predicted next instructions into the pipeline. If the branch is taken indeed, then clock cycles have been saved by not fetching incorrect instructions into the pipeline. This shall be explained in more detail in Chapter: 4

RISC-V

An Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the CPU is controlled by the software. The ISA acts as an interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done. [ISA][Ray21]

RISC-V is an open standard instruction set architecture that uses RISC principles of basic, fewer instructions than compared to CISC.[RIS] It was first introduced in 2010 and currently supports 32, 64, and 128 bit CPUs [Ray21].

The CPU architecture uses a load-store topology, meaning that the processor is unable to directly operate on data in memory, and instead requires the user to move data from memory to its internal registers before it can operate on them. While this makes RISC-V slower than a Complex Instruction Set Computer (CISC), it allows RISC-V to be simpler in hardware design, and therefore uses less silicon space [Ray21].

This concludes the introduction of vital concepts. These concepts will be further explored in the upcoming chapters.

1.3 Objectives

In order to carry out this thesis, varied tasks needed to be completed. The following lists presents the work done during this thesis:

- Familiarize myself with the existing documentation, and Register Transfer Language code, for the CV32E40X processor core.
- Correct the results from the modeling of the branch predictors in the Project Report [Ray21].
- Choose a branch predictor algorithm to implement based on the aforementioned results.
- Setup the simulation environment for simulating the circuits built using SystemVerilog.
- Develop a block diagram for the intended implementation of the branch predictor.
- Modify the existing code to implement the chosen branch predictor.
- Modify the modeling of the chosen branch predictor to provide a comparison between the theoretical and experimental results.

The tasks were carried out in the same chronological order as they are presented in the aforementioned list.

Having discussed all the things mentioned at the start of this chapter, I conclude Chapter 1 here. The following chapters will dive deeper into the work done and explain my choices along the way.

Chapter 2

Background and theory

This thesis, as mentioned in Chapter 1, builds upon the results obtained from the Specialization Project completed during the Autumn semester of 2021. The findings from that project were documented in the Project Report. It is not expected that the reader of this thesis has also read the Project Report, hence the purpose of this chapter shall be to introduce and explain concepts that are vital to understanding the work done as part of this thesis.

2.1 Branch instructions and it's effect on the instruction pipeline

Branching refers to the task of deviating from the original execution of an instruction sequence to one that contains a detour as a result of a branch instruction. Normally, the Program Counter is incremented after every completed instruction. A conditional Branch Instruction, when true, instead of incrementing the Program Counter by one to fetch the next instruction, sets the next program counter value to the "Branch Target". Branching thus allows the programmer to branch out from the normal sequence of execution, and execute the code contained at the branch location. Figure 2.1 aims to clarify this by means of a flowchart [Ray21].

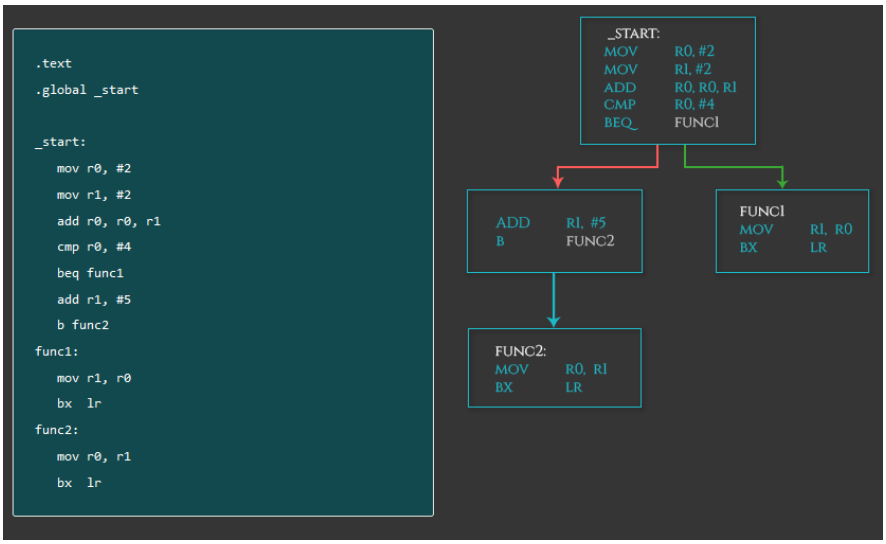


Figure 2.1: Effect of a branch instruction on execution sequence [Braa]

Branch instructions can alter the contents of the CPU's Program Counter (PC). The PC holds the memory address of the next machine instruction to be fetched into the instruction pipeline. In a program without branch instructions, the PC holds the value of the next sequential instruction. When a branch is evaluated to be taken, instead of fetching and executing the next sequential instruction, the PC value is changed such that it points to a different sequence of instructions than the one being executed.

It is not known for certain whether a conditional jump will be taken or not taken until the condition has been evaluated. This evaluation happens when the conditional jump has passed the execution stage in the instruction pipeline. Knowing whether or not a branch is "taken" is called the outcome of a branch.

Lack of knowledge regarding how the Program Counter is going to be affected as a result of a branch instruction will impact the processor gravely. Most modern processors make use of pipelining to achieve instruction-level parallelism. Considering the example of a 4-stage pipeline, the problem with branch instructions can be demonstrated.

	##1	##2	##3	##4	##5	##6
Fetch	bne	PC+1	PC+2	CorrPC	Ins5	Ins6
Decode		bne	PC+1	PC+2	CorrPC	Ins5
Execute			bne	PC+1	PC+2	CorrPC
Writeback				bne	PC+1	PC+2

Figure 2.2: This figure demonstrates a bubble created in the pipeline because of mis-predicting the outcome of the branch

The green rectangles symbolize instructions that are valid. The red rectangles symbolize instructions that were killed. Killing an instruction means that the status of that instruction is set to be invalid. This means that the instruction itself is propagated through the pipeline, but no computational activity is done on it. In the Figure 2.2 presented above, bne (branch if not equal) is a branch instruction. In the case of the branch instruction, shown in Figure 2.2, the mis-predicted instruction was killed when the EX stage evaluated the branch outcome to be "not taken". It is visible how the branch instructions can lead to bubbles in a pipeline. The instruction pipeline can be thought of as a super-circuit, in the sense that it works as a circuit, which is comprised of smaller circuits. Figure 2.2 shows a four stage pipeline, which means that there are 4 circuits that work together to make up the instruction pipeline. For the instruction pipeline to be efficient, it is quintessential that each "stage" is as efficient as possible. Having dead instructions (like the ones caused by branch instructions) means that one or more stages in the pipeline is not being utilized optimally. As a result, not only is the processor losing important clock cycles that could have been used to produce useful results, it also means that the processor is wasting energy while doing the useless work. The CV32E40X processor core has no dedicated branch predicting circuit. It is assumed that no branches are taken, which in itself is a branch prediction strategy, and is called the "Always not taken" prediction strategy [JLH11] [pen]. Since no prediction is made, the processor core assumes that the next instruction to be fetched into the pipeline is always the instruction that comes sequentially after the branch instruction. This means that for every branch whose outcome turns out to be not-taken, the CPU has predicted the correct target address. The branches which are taken, however, would necessitate killing of the mis-predicted target instruction(s), and fetching the correct instruction(s). This imposes a 2 cycle mis-prediction penalty for every branch whose outcome is mis-predicted, in the case of the four stage pipeline shown in Figure 2.2. The CV32E40X processor core, the one central to this thesis, also has a four

stage pipeline. The explanation above, therefore, also applies to the CV32E40X processor core. This will be discussed in greater detail in Chapter 4.

2.2 Branch Prediction

Without branch prediction, the processor would have to wait until the conditional jump instruction has passed the execute stage before the correct PC can be fetched into the pipeline. The branch predictor attempts to avoid this latency by trying to predict the result of the conditional jump. A branch is speculated to be taken (or not taken) then fetched and speculatively executed. If it is later evaluated that the prediction was incorrect, the incorrect instructions are killed and the pipeline starts over with the correct instructions which are fetched at the target address pointed to by the correct PC. It is desirable to predict the outcome of a branch as early as possible in order to make the prediction as rewarding as possible. The earlier a correct prediction is made, the higher the number of cycles saved. Consider an example where the output of the branch predictor is ready before the fetch stage. In such a case, given that the prediction is correct, the pipeline is always full with the correct instruction sequence and is thus maximizing resource usage. In the case that a branch is mis-predicted, however, there will be a penalty as shown in Figures 2.2. A metric of how fast a program can run on a CPU can be measured by looking at the amount of time used per execution of a benchmark program.

$$\frac{time}{program} = \frac{time}{cycle} \times \frac{cycles}{instruction} \times \frac{instructions}{program} \tag{2.1}$$

From equation 2.1, it is arguable that in order to speed up execution, a designer can choose to reduce the amount of time per cycle, which can be done by increasing the clock frequency. This a designer's choice that has rather serious consequences. Too high a frequency could lead to excessive dynamic switching losses[dyn]. A discussion on dynamic switching losses is one that is out of scope for this project, but is nevertheless crucial to understand Equation 2.1. Another variable that depends on the software designer is the number of instructions per program. Cutting down on the number of instruction in a given program would speed up it's execution. This, however, depends on the designer's style of coding. The metrics time per cycle, and instructions per program are examples of subjective things that affect the speed of execution. The objective metric is the number of cycles per instruction, as this is fixed by the design of the CPU. No matter what clock frequency a designer decides to implement, implementing a branch predictor, as explained in section 2.1 can help reduce the number of clock cycles used per instruction. This not only means that the CPU is using its resources better, but also that it is executing instructions faster [ADAH16]. This can be an advantage in low-power applications where the CPU can finish doing what it needs to do faster and enter an energy saving mode [ADAH16]. This technique is called the race-to-idle and helps the processor become more energy efficient [ADAH16]. This reflects well with the goals of RISC-V ISA where the aim is to make fast and energy efficient processors.[RIS]

2.2.1 Branch Prediction Strategies

Having discussed the pitfalls introduced by branch instructions and branch prediction as a potential solution to the problem, different branch prediction strategies can now be introduced. Branch prediction strategies can be classified:

– Static branch prediction

– Dynamic branch prediction

Static Prediction

Static prediction, makes static or constant predictions about encountered branches [Smi]. By constant/static prediction, the prediction is the same throughout the run-time of the program and is not affected by the history of branch outcomes. For example, a static predictor can predict that all incoming branch instructions are taken(or not-taken). The name static arrives from the same exact prediction made by the predictor regardless of the branch instruction encountered. This is different from a predictor that uses data like the history of previous outcomes for a branch, or the knowledge of the type of branch instructions to affect the outcome of the prediction [Smi]. This difference will become more apparent with the introduction of Dynamic Prediction later in this section.

A more nuanced form of static prediction presumes that backward branches will be taken and that forward branches will not, or vice-versa [Smi]. A backward branch is one that has a target address that is lower than its own address. In other words, the target PC value is lower than the PC value at which the branch instruction was encountered. This technique can help with branch prediction for loops, which are usually backward-pointing branches, and are taken more often than not taken [Smi]. Static prediction is used as a fall-back technique in some processors where dynamic branch prediction is the primary prediction algorithm. Both the Motorola MPC7450 (G4e) and the Intel Pentium 4 use this technique as a fall-back [Smi].

In static prediction, all decisions are made at compile time, before the execution of the program. As mentioned above, this just predicts whether a branch will be taken or not, and no execution time events affect this prediction.

Dynamic branch prediction

In contrast to the static prediction techniques, dynamic prediction techniques aim at making predictions, and improving these predictions during program execution. For the scope of this thesis, only One-bit saturating predictors will be explored. Other techniques like Two-level saturation counters or Local history tables are also possible alternatives. These, however, will not be explained as they were not implemented in the scope of this thesis.

Global History Predictor

A Global History predictor working with a 1-bit saturation counter keeps a shared history of immediately previous outcomes for n conditional jumps[Smi]. A table consisting such a history can be checked to see if the branch instruction that entered the pipeline can be found. If so, the prediction is set to be equal to the previous outcome of said branch. A 1-bit saturating counter (essentially a flip-flop) records the last outcome of the branch. This is the simplest version of a dynamic branch predictor. There are multiple advantages with using this technique as listed below.

– Relatively straight-forward algorithm that doesn't warrant an excessive computational overhead.

– Given that a flip-flop can be used to store the outcome of the branch, it is rather space and resource efficient.

– Works well when the program is relatively straight forward, where the outcome of the branch does not change too often.

There are also considerable problems with this algorithm. They are mentioned in the following list.

- It is a very sensitive prediction algorithm. Since the prediction made is based solely on the outcome of the previous time the branch was encountered, the prediction is unstable. It is unstable compared to an algorithm which would favor the outcome that occurs in majority. If a branch instruction encountered is taken more often than not, the averaging algorithm would better be able to "predict the correct outcome" than a 1-bit saturation counter.
- To make the prediction algorithm as accurate as possible, it needs to be able to store the PC at which the branch is encountered, the target address of the branch, in case it is taken, and the history. Considering that this algorithm is used with a processor core that uses 32-bit instruction words, both the PC and the target address would be 32 bits. On this scale, using 1 bit for saving the branch history is only making use of $1/65^{th}$ of the assigned 65 bits. Using 2 more bits (3 bits in total) allows for storing the weighted average of 8 previous instances of said branch, which is a far more stable data value to base the prediction on.
- Implementing this in hardware will require the use of some form of a history table solution that stores the information mentioned in the earlier point. Assigning storage to the history table may become overkill for straight-forward programs. It may perhaps be easier to evaluate static predictors for such applications.

2.2.2 Replacement Policies for replacing entries in the History Table

In an ideal environment where no limitations were implied on the size of memory required to store the branch history, all branches along with their history could be saved. Computer architecture, however, places real limitations on the size of the memory that can be used by a branch history table. This is so that not all memory available is used only to store branch history. The memory still needs to be available for storing more important things like variables, and data elements, among other things. This means that the entries in the limited sized history tables would have to be replaced so as to always have sufficient data that is relevant in the temporal domain. To understand this, consider the case of a branch history table that can hold eight entries. If a program has nine branch instructions, and assume that the first eight of these are seen very frequently. It would then make sense for the table to hold the history of these eight branches longer. It is then the task of the replacement policy used to make sure that it allows for the first eight entries to be available in the history table longer. The Most Recently Used (MRU) replacement policy does not make sense as the choice of replacement policy for table entry updating. The reason for this being that each time an entry is updated in the table, it becomes the MRU entry in the table. Using the MRU would lead to only one entry being updated repeatedly causing a case where branch history is stored inefficiently. Consider an empty table which stores 8 entries. Consider an algorithm that is designed to make use of the MRU replacement policy that starts by filling entries which are empty in the table. If no empty places were found, MRU entries in the table would be replaced. Following this, after the table has been initially filled with entries, only the 8^{th} entry would be replaced upon all further replacements. In the case where the history table size is of 8 entries, and 10 branch instructions are encountered, this is a waste of resources as only $1/8^{th}$ of the table size is being used to store the history as compared to using all the entries in the table. Figure 2.3 visualizes how the entries in the history table would be updated under the MRU replacement policy. 2.3

Entry 1	Entry 2	Entry 3	Entry 4	Entry 5	Entry 6	Entry 7	Entry 8
B1							
B1	B2						
B1	B2	B3					
B1	B2	B3	B4				
B1	B2	B3	B4	B5			
B1	B2	B3	B4	B5	B6		
B1	B2	B3	B4	B5	B6	B7	
B1	B2	B3	B4	B5	B6	B7	B8
B1	B2	B3	B4	B5	B6	B7	B9
B1	B2	B3	B4	B5	B6	B7	B10
B1	B2	B3	B4	B5	B6	B7	B11

Figure 2.3: Most Recently Used Replacement Policy

Least Recently Used (LRU) replacement policy however, is a much better option than the MRU replacement policy for storing history of branch outcomes [JLH11]. In contrast to the MRU, LRU policy would allow for the oldest entry in the table to be updated. This would mean that all new entries in the table would be available for prediction much longer than when compared to the MRU replacement policy. Having more diverse branch outcome history, that is available for equal amounts of time, when talking about how long each entry is available in the table before it gets replaced, allows for more stable predictions [JLH11]. This is also a much better use of resources as all assigned memory elements actually hold data that is relevant and may be used in the near future. Figure 2.4 visualizes how the entries in the history table would be updated under the LRU replacement policy.

2.4

Entry 1	Entry 2	Entry 3	Entry 4	Entry 5	Entry 6	Entry 7	Entry 8
B1							
B1	B2						
B1	B2	B3					
B1	B2	B3	B4				
B1	B2	B3	B4	B5			
B1	B2	B3	B4	B5	B6		
B1	B2	B3	B4	B5	B6	B7	
B1	B2	B3	B4	B5	B6	B7	B8
B9	B2	B3	B4	B5	B6	B7	B8
B9	B10	B3	B4	B5	B6	B7	B8
B9	B10	B11	B4	B5	B6	B7	B8

Figure 2.4: Least Recently Used Replacement Policy

The LRU method requires an algorithm to keep track of which entry in the table is the oldest [JLH11]. This can be done using a 3 bit number for the case of a table which has a total size of 8 entries. An algorithm can be designed to check if the table has empty entries first. If no empty entries are found, it looks for the largest 3 bit number denoting the age of the entry, and replaces that with the new data. The new data gets an age of 0 to start with. Upon each update to the table, age of all entries except the new one could be increased by 1. If an entry with age 7 is found, replace that with new data, and give new data an age of 0 (this being the case because 7 is the highest 3-bit number possible, implying the oldest available data). Else, the oldest entry could be replaced. The only drawback with this

method is that it necessitates the presence of 3 extra "age bits" per entry, a 3-bit register to hold the highest age in the table, and a comparator to find out the largest age in the table. This may not seem like excessive hardware to have for small table sizes, but as sizes increase, so will the number of "age bits" and the size of the register to hold the oldest age [JLH11]. This is where using the Random Replacement Policy comes in handy. As opposed to the LRU and MRU policies, the random policy can be made to randomly updates entries in the table, given that no empty entries are found. The advantage with this method is that it doesn't need quite as much hardware as the LRU method. The drawback however, is that one risks losing important entries that could potentially be useful [JLH11].

2.3 Modeled Branch Prediction strategies

The Project Report modeled three different branch-prediction strategies:

- Backwards predictor
- Global History predictor
- Hybrid predictor

For both the history predictor and the hybrid predictor, variable table sizes were modeled. This meant that the effect of storing different numbers of branch instructions in a global table were modeled. Backwards predictor has been discussed in adequate detail in section 2.2.1, and so has Global History predictor. The Hybrid Predictor is a predictor where the branch instruction is first looked up in the history table. If it's history is found, the prediction is made using the available data. If, however, the data is not found in the table, the Backwards predictor is used as a backup algorithm to predict the outcome of the branch. The reason for modeling these were based primarily on metrics relating to power consumption, area, and speed.

A Backwards predictor is arguably the one that consumes the least amount of power, and is smallest in area. This is because a digital implementation of such a branch predictor only needs a 32-bit comparator which compares the value of the current program counter to the branch target address. The prediction is then made as described in 2.2.1. A comparator is a digital circuit that compares two input signals. One of these signals is a reference signal while the other is the input. The comparator outputs a digital "high" signal if the input signal is greater than the reference signal. Since we want the comparator to provide a "high" output when the input is lower than the reference signal, we can simply invert the output signal using a NAND gate.

The Global History Predictors and the Hybrid Predictor use more area, and power than the Backwards Predictor. This is because both the Hybrid and Global History Predictor models consisted of a history table with a size between 4 and 128 entries. This means that every entry in the table consists of a branch target address, which is 32 bits long, and the history of the outcome of that branch. In addition to this, the outcome of the branch was also stored using an additional bit. For every branch instruction in the table, 65 bits were used to store the required data.

Chapter 3

Correction of Branch Prediction models

In order to understand why the results from the modeling of different branch prediction algorithms needed to be corrected, consider the following: Branch Prediction is a technique to predict the outcome of a branch instruction and fetch the predicted next instruction into the pipeline. If the branch outcome is correctly predicted, then the pipeline remains fully packed with correct instructions, in turn leading to more efficient program execution. Without branch prediction, there is no way of knowing what the outcome of a branch instruction may be prior to its evaluation in the EXECUTE stage. One way to tackle this problem could be to continue fetching instructions assuming that the branch will not be taken. In other words, assume no branching from original program execution flow. In this case the pipeline can remain full with instructions allowing for optimal utilization of resources. If however, it is found out in the EXECUTE stage, that the branch indeed needs to be taken, then pipeline needs to flush out the incorrect instructions, and fetch in the correct ones. As a consequence, important clock cycles are lost trying to rectify for fetching in incorrect instructions. Making use of branch prediction algorithms should thus make it possible to save clock cycles. In other words, usage of branch prediction should show a decrease in total clock cycles needed to run a program.

Figures 3.1, and 3.3 show some interesting results. From Figure 3.1, it is visible that the clock cycles saved by using this strategy and a table size of less than 32 entries gives a **negative** impact on the clock cycles saved. This is exactly the opposite of what the branch predictor was supposed to do. Given also, that the hybrid predictor is just a history predictor working together with a backwards predictor, the numbers reported by the hybrid predictor did not seem correct either.

This prompted a round of debugging the modeling code. It was found that the Global History Predictor and Hybrid Predictor models had an error in the modeling algorithm. Furthermore, the method using which number of cycles saved were calculated also had a bug which resulted in too many cycles being reported as saved for the Backwards Predictor. Both History Predictor and Hybrid Predictor worked by storing the outcome of the previous instance of the branch. It was found that updating the entry, in which the outcome of the branch was to be updated, was being done incorrectly. This led to values being updated in random places in the history table, and not where they should have been updated. This resulted in history data corruption, in turn misleading the branch prediction results. In addition to the flaw mentioned above, the method of calculation of total cycles saved was also incorrect. It was thus essential to fix these errors before beginning the implementation of the branch predictor circuit. This would allow for a better, stronger, and more correct

foundation for making the choice of implementing chosen branch predictor(s).

```
1  def history_predictor( pc , bt , status, table_size):
2      global program_counter , branch_target, success, failure, i
3      i = ((i)%(table_size))
4      absence_counter = 0
5
6      for index in range(table_size):
7          if history_table[index][1] == program_counter :
8              prediction = history_table[index][0]
9              if prediction == status:
10                 success += 1
11                 history_table[i][0] = status
12                 break
13             else:
14                 failure += 1
15                 history_table[index]0] = status
16                 break
17         else:
18             absence_counter += 1
19             if  absence_counter == table_size:
20                 failure += 1
21                 break
22
23      history_table[i][0] = status
24      history_table[i][1] = pc
25      history_table[i][2] = bt
26      i += 1
```

Listing 3.1: Update code for History Predictor

In the listing above, the value of the table is updated in lines 23 through 25. The iteration through the table, however, is done in lines 6 through 20. It is visible right away that the variable that is used to iterate through the table is called “index” (line 6, in the "for" loop definition), is not the same as the variable “i” which is what is used to update the table. The same error was also found in the hybrid predictor model. Due to the similarity, the code for the hybrid model is not shown here. The error was fixed by rewriting the model in the following way:

```
1  def hybrid_predictor(pc, bt, status, table_size):
2      global  program_counter, branch_target, success, failure, cycles_saved
3
4      for index in range(table_size):
5          if history_table[index][1] == program_counter:
6              prediction = history_table[index][0]
7              if prediction == status:
8                  if ((prediction == 0) and (status == 0)):
9                      cycles_saved += 0
10                 else:
11                     cycles_saved += 2
12                 history_table[index][0] = status
13                 return
14             else:
15                 if ((prediction == 0) and (status == 1)):
16                     cycles_saved -= 2
17                 else:
18                     cycles_saved += 0
19                 history_table[index][0] = status
```

```
                return
    for index in range(table_size):
        if history_table[index] == [0,0,0]:
            history_table[index] = [status, pc, bt]
            backwards_predictor(pc, bt)
            return
    backwards_predictor(pc,bt)
    random_replacement = random.randint(0, table_size - 1)
    history_table[random_replacement][0] = status
    history_table[random_replacement][1] = pc
    history_table[random_replacement][2] = bt
    failure += 1
    return
```

Listing 3.2: Update code for Hybrid Predictor

In the new models, entries were added to empty elements in the table. If no empty places were found, random entries in the table were replaced. The random replacement of entries in the table nuanced the algorithms as ones using the random replacement policy. Other policies like the Least Recently Used (LRU), and Most Recently Used (MRU) could just as well be implemented.

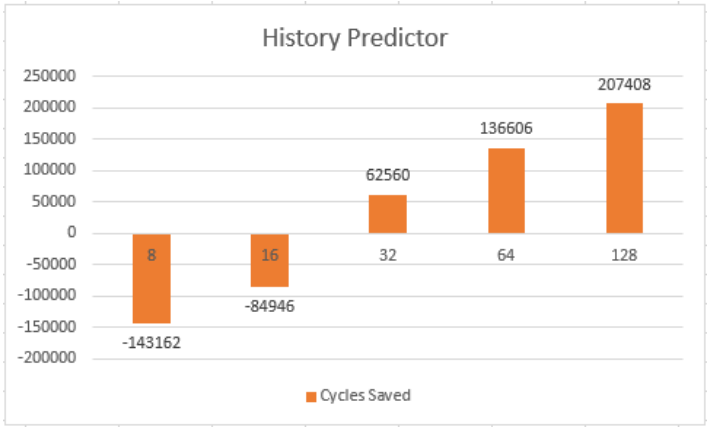


Figure 3.1: Incorrect history predictor.

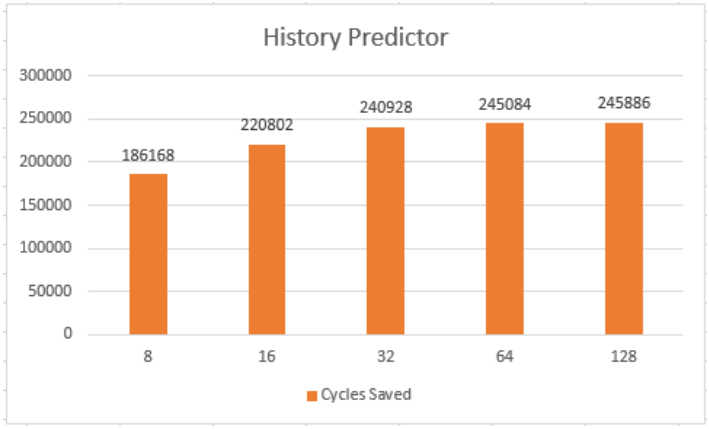


Figure 3.2: Corrected history predictor.

After rewriting the models, the following results were acquired by running the corrected models. For reference purposes, the total number of cycles used to complete the execution of the benchmarking test were 3,275,819 cycles, in which a total of 2,608,664 instructions

were completed. These results are showcased in Figure 3.2 and Figure 3.4. As is clearly evident from the Figures 3.1 and 3.3, the difference between the results was quite substantial. Not only was it important to correct the results of the modeling to help choosing the branch prediction algorithm to implement. It was also necessary to correct the results so that I could make sure that I had understood the replacement policies correctly.

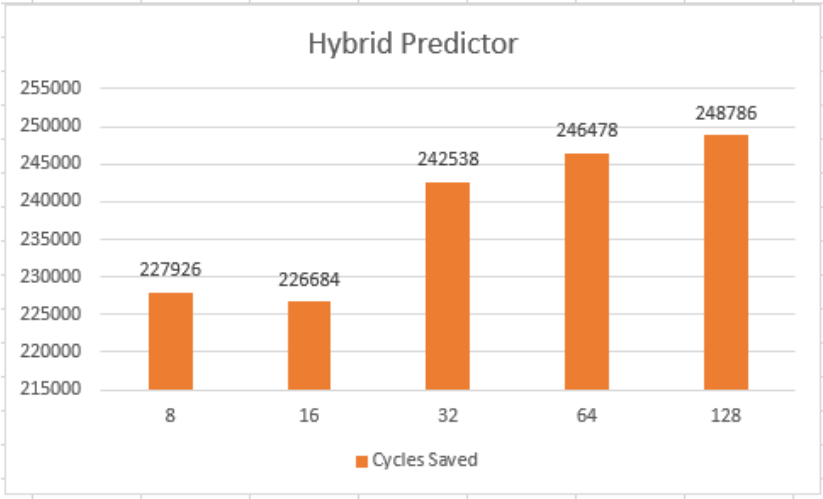


Figure 3.3: Incorrect hybrid predictor.

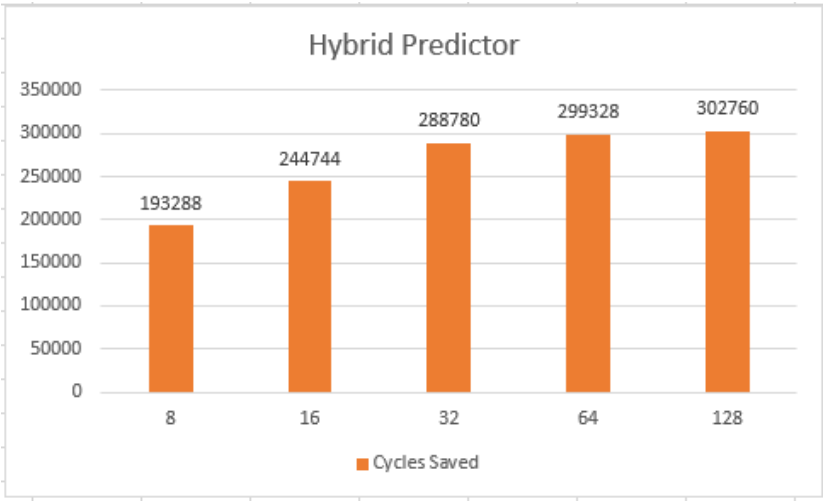


Figure 3.4: Corrected hybrid predictor.

In the Project report, it was reported that the Backwards Predictor saved roughly 121,000 cycles while running the Coremark ©, when in reality, only 59,454 cycles were saved.

From the corrected results for both history and hybrid predictors, I had a much stronger foundation to choose between what algorithm to implement. From the results, it is obvious that Hybrid predictor should be the implementation of choice, however, things are more complicated than they may seem. It is important to notice that the results from the modeling were achieved by feeding and run-time instruction log to a high-level model, that only models the behavior of the algorithm. This allows for a better understanding of how each prediction algorithm works. The purpose of modeling is to better understand **how** the models need to be implemented in hardware. The implementation itself however, is a much more complicated task. The plan of attack for this thesis was to build the backwards predictor first, analyze

the results and if time permits try implementing the other predictors. This shall be discussed further in Chapter: 4.

Chapter 4

CV32E40X processor core

For the scope of this thesis, I will focus on the implementation of the RISC-V architecture for the construction of the CV32E40X processor core. CV32E40X is a 4-stage in-order 32-bit RISC-V processor core. [CV3] Figure 4.1 showcases the block diagram for the same. This chapter presents the existing architecture for the CV32E40X processor core. The details entailed herein are important for two main reasons: it provides important insight into how the processor core functions, and it makes my contribution to the code base and changes easier to justify.

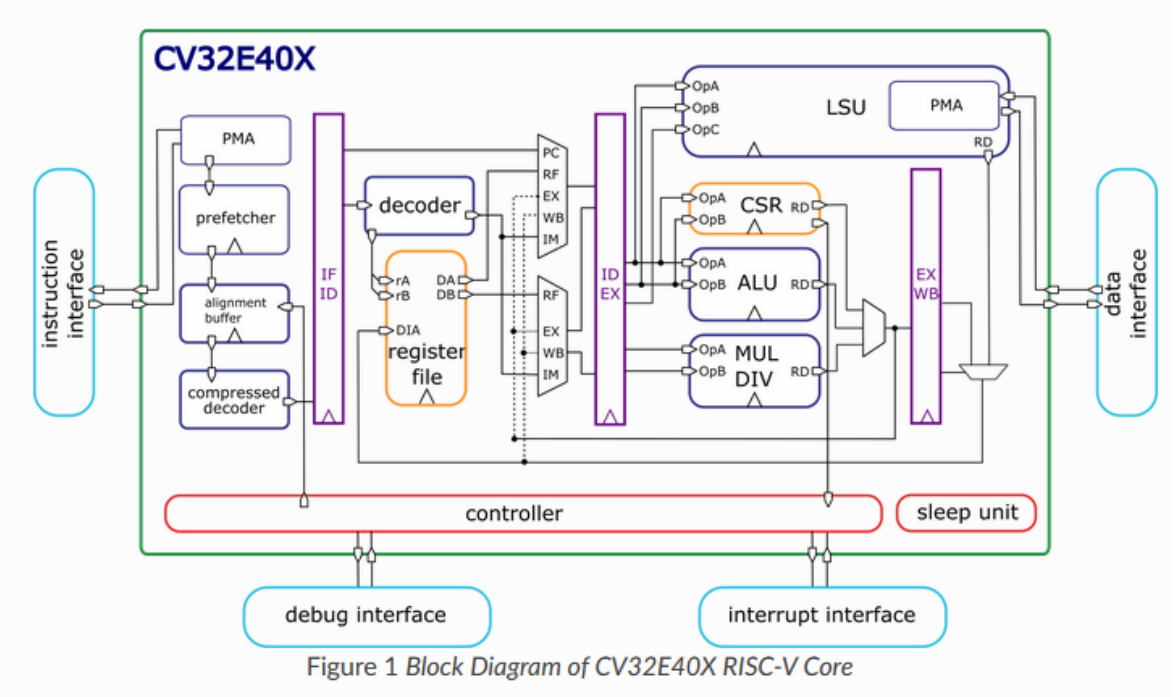


Figure 4.1: Pipeline diagram for the CV32E40X [CV3]

In this chapter, the branch handling for the CV32E40X processor core will be explained. In order to do so, there are some details that must be understood regarding the pipeline and the flow of instructions.

4.1 Pipeline Details

As mentioned above, the core has a 4-stage in-order completion pipeline. This means that the pipeline has 4 stages, and the core executes the instructions in the same order as they appear in the pipeline. The four different stages in the pipeline are: Instruction Fetch, Instruction Decode, Execute, and Writeback [CV3].

- **Instruction Fetch (IF)**:
  Fetches instructions from memory via an aligning prefetch buffer, capable of fetching 1 instruction per cycle if the instruction side memory system allows [CV3].
- **Instruction Decode (ID)**:
  Decodes fetched instruction and performs required register file reads. Jumps are taken from the ID stage [CV3].
- **Execute (EX)**:
  Executes the instructions. The EX stage contains the ALU, Multiplier and Divider. Branches (with their condition met) are taken from the EX stage. Multi-cycle instructions will stall this stage until they are complete. The address generation part of the load-store-unit (LSU) is contained in EX as well [CV3].
- **Writeback (WB)**:
  Writes the result of ALU, Multiplier, Divider, or Load instructions instructions back to the register file [CV3].

Each pipeline stage is connected to a Controller block, as shown in figure 4.1, which is responsible for handling interrupts/exceptions that arise. It has access to all the pipeline stages, and does tasks like setting the right Program Counter value to be fetched in via the IF stage in the case that a branch is shown to be taken in the execute stage. The Controller also incorporates data handling for Performance Counters. Performance Counters are bits of code that monitor, count, or measure events in software, which allow us to see patterns from a high-level view.

Before presenting the intended design of the backwards predictor, it is important to understand how information flows through the processor core pipeline. And before I can explain how all the information flows, it is vital that I present the structure of the code base and introduce how Register Transfer Language (RTL) code that describes the hardware description of the processor core is written. The language of choice to write the RTL code is SystemVerilog.

4.1.1 SystemVerilog

Hardware Description Languages (HDL) like Verilog and VHDL are used to describe hardware behavior so that it can be converted to digital blocks made up of combinational gates and sequential elements. In order to verify that the hardware description in HDL is correct, there is a need for a language with more features in Object Oriented Programming that will support complicated testing procedures and is often called a Hardware Verification Language. SystemVerilog is an extension of Verilog with many such verification features that allow engineers to verify the design using complex testbench structures and random stimuli in simulation. SystemVerilog is the industry's first unified hardware description and verification language (HDVL) standard. It was developed originally by Accellera to dramatically improve productivity in the design of large gate-count, IP-based, bus-intensive chips. SystemVerilog

is targeted primarily at the chip implementation and verification flow, with powerful links to the system-level design flow. SystemVerilog has been adopted by 100's of semiconductor design companies and supported by more than 75 EDA, IP and training solutions worldwide [Sys]. This introduction to the language was necessary since the entire RTL code for this thesis was written in System Verilog.

4.2 Introduction to the Code base

The include folder contains the cv32e40x_pkg.sv file, which contains the type definitions for all the data structures used in the code. cv32e40x_core.sv is the top level file that connects the different circuits together. It is here that the Controller block is connected to the IF, ID, EX, and WB stages. The cv32e40x_if_stage.sv, cv32e40x_id_stage.sv, cv32e40x_ex_stage.sv describe the IF, ID, and EX circuits respectively, while the cv32e40x_controller_fsm.sv file describes the behavior of the controller_fsm circuit. The controller_fsm circuit is the same as the Controller block seen in Figure 4.1. Communication between the Controller block and all the other circuits happens using data structures defined in the cv32e40x_pkg.sv file. The data structures that are of interest for this thesis are: ctrl_fsm_i (input to the IF stage), ctrl_fsm_o (output from the Controller), id_ex_pipe_i (input to the Controller, and EX stage), and id_ex_pipe_o (output from the ID stage). The HDL code for the CV32E40X processor core is structured in the following way:

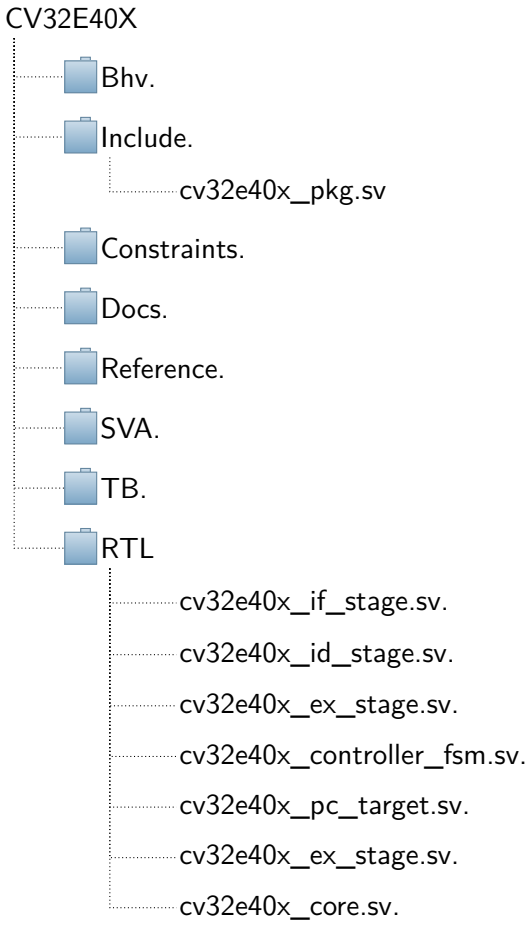


Figure 4.2: CV32E40X Code Structure

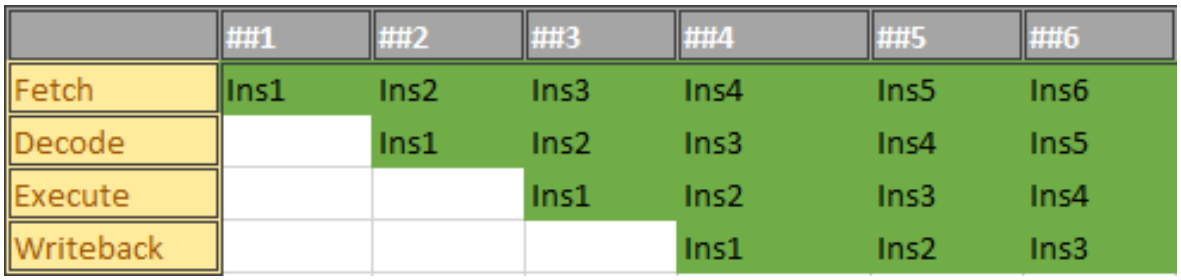


Figure 4.3: Block diagram for the CV32E40X
[CV3]

Figure 4.3 shows how the CV32E40X pipeline would work in an ideal scenario. This means that no stalls occur in the pipeline, no instructions need to be killed, and the pipeline is full with useful and correct instructions. The top row in the figure depicts the clock cycles and the yellow column contains information about what stage the instructions are in. It is visible from the figure that during the first clock cycle, Instruction 1 (Ins1) is brought into the fetch stage. The IF stage of the CV32E40X can supply one instruction to the Instruction ID stage per cycle if the external bus interface is able to serve one instruction per cycle. The processor core makes use of a prefetcher for optimal performance and timing closure reasons [CV3]. The prefetcher fetches instructions via the external bus interface from for example an externally connected instruction memory or instruction cache. The prefetch unit performs word-aligned 32-bit prefetches and stores the fetched words in an alignment buffer with three entries [CV3]. As a result of this (speculative) prefetch, CV32E40X can fetch up to three words outside of the code region [CV3]. Fetch address selection is done in the IF stage, but the Controller module contains a Finite State Machine that controls the selection. This means that the Controller FSM is responsible for choosing the next address to be fetched from the available multiplexer in the IF stage. This is done by setting the desired pc_mux value in the ctrl_fsm_o.pc_mux which is the output struct from the Controller FSM. This struct is called the ctrl_fsm_i in the IF stage to signify that it is an input from the Controller FSM. In the following code listing, each multiplexer input sets the value for the branch_addr_n to a specific PC. For instance, the upon receiving a PC_BRANCH input to the multiplexer, the branch_addr_n variable is set to equal branch_target_ex_i value. Here, branch_target_ex_i is an input to the IF stage from the EX stage. When EX stage receives a branch instruction, it sets the target address, branch_target_ex_i variable. If the branch is evaluated to be taken, the Controller FSM provides PC_BRANCH as an input to the multiplexer in the IF stage thus leading to the choice of branch_target_ex_i as the next PC value to be fetched in.

```
1  always_comb
2   begin
3     // Default assign PC_BOOT (should be overwritten in below case)
4     branch_addr_n = {boot_addr_i[31:2], 2'b0};
5
6     unique case (ctrl_fsm_i.pc_mux)
7       PC_BOOT:     branch_addr_n = {boot_addr_i[31:2], 2'b0};
8       PC_JUMP:     branch_addr_n = jump_target_id_i;
9       PC_BRANCH:   branch_addr_n = branch_target_ex_i;
10      PC_MRET:     branch_addr_n = mepc_i;
11      // PC is restored when returning from IRQ/exception
12      PC_DRET:     branch_addr_n = dpc_i;
```

```
13        PC_FENCEI:   branch_addr_n = ctrl_fsm_i.pipe_pc;
14        // Jump to next instruction forces prefetch buffer reload
15        PC_TRAP_EXC: branch_addr_n = {mtvec_addr_i, 7'h0};
16        // All the exceptions go to base address
17        PC_TRAP_IRQ: branch_addr_n = {mtvec_addr_i,
18        ctrl_fsm_i.mtvec_pc_mux, 2'b00};
19        // interrupts are vectored
20        PC_TRAP_DBD: branch_addr_n = {dm_halt_addr_i[31:2], 2'b0};
21        PC_TRAP_DBE: branch_addr_n = {dm_exception_addr_i[31:2], 2'b0};
22        PC_TRAP_NMI: branch_addr_n = {mtvec_addr_i, NMI_MTVEC_INDEX, 2'b00};
23        PC_TRAP_CLICV: branch_addr_n =
24        {mtvt_addr_i, ctrl_fsm_i.mtvt_pc_mux[SMCLIC_ID_WIDTH-1:0], 2'b00};
25
26        // CLIC spec requires to clear bit 0.
27        //This clearing is done in the alignment buffer.
28        PC_TRAP_CLICV_TGT: branch_addr_n = if_id_pipe_o.instr.bus_resp.rdata;
29        default:;
30      endcase
31    end
```

Listing 4.1: PC_mux in IF stage

[GIT]

It is important to note that in the IF stage, the processor core does not know what type of instruction is being fetched. Even though there is a program counter multiplexer which has names for different choices, the processor core itself does not know what instruction is being fetched. The names are merely for readability purposes. When the instruction is fetched, given that the ID stage is ready to receive data from the IF stage, and that the data present in the IF stage has not been invalidated by the Controller FSM, all the relevant data from the IF stage is loaded into the if_id_pipe struct. This is a data structure that is used to propagate relevant data from the IF stage to the ID stage. It can be seen in the listing below that information pertaining to the validity of the instruction being propagated, the PC value, if the instruction is compressed or not, and other things are contained in the if_id_pipe_o data structure. For the purpose of branch prediction, the PC value, the information about the validity of the instruction, and whether or not the instruction is compressed is all that is needed. It is visible how upon reset, all the fields in the if_id_pipe_o data structure are set to be zero, or initial values. This makes sense because upon a received reset signal, the pipeline must be flushed and the processor core needs to be brought to the initial state. In the initial state, it does not make sense to have a pipeline which is already filled with instructions.

```
1   always_ff @(posedge clk, negedge rst_n)
2    begin : IF_ID_PIPE_REGISTERS
3      if (rst_n == 1'b0) begin
4        if_id_pipe_o.instr_valid      < = 1'b0;
5        if_id_pipe_o.instr            < = INST_RESP_RESET_VAL;
6        if_id_pipe_o.use_merged_dec   < = 1'b0;
7        if_id_pipe_o.instr_meta       < = '0;
8        if_id_pipe_o.pc               < = '0;
9        if_id_pipe_o.illegal_c_insn   < = 1'b0;
10       if_id_pipe_o.compressed_instr < = '0;
11       if_id_pipe_o.trigger_match    < = 1'b0;
12       if_id_pipe_o.xif_id           < = '0;
13     end else begin
14       // Valid pipeline output if we are valid AND the
15       // alignment buffer has a valid instruction
```

```
16        if (if_valid_o && id_ready_i) begin
17          if_id_pipe_o.instr_valid      <= 1'b1;
18          // instr_decompressed may be a pointer in case of CLIC or Zc, handled
        within the compressed decoder.
19          if_id_pipe_o.instr            <= instr_decompressed;
20          if_id_pipe_o.use_merged_dec   <= use_merged_dec;
21          if_id_pipe_o.instr_meta       <= instr_meta_n;
22          if_id_pipe_o.illegal_c_insn   <= illegal_c_insn;
23          if_id_pipe_o.pc               <= pc_if_o;
24          if_id_pipe_o.compressed_instr <= prefetch_instr.bus_resp.rdata[15:0];
25          if_id_pipe_o.trigger_match    <= trigger_match_i;
26          if_id_pipe_o.xif_id           <= xif_id;
27        end else if (id_ready_i) begin
28          if_id_pipe_o.instr_valid      <= 1'b0;
29        end
30      end
31    end
```

Listing 4.2: if_id_pipe_o

[GIT]

The if_id_pipe_o struct from the IF stage is used as an input in both the Controller FSM and the ID stage. In the ID stage, the instructions that were fetched in IF are decoded. The ID stage instantiates the Decoder circuit which only takes two inputs, one being for exception handling, and the other being the if_id_pipe data structure. Based on this the decoder sets various enable signals, and decides on values that will be useful in the ID stage. For instance, the Decoder outputs a variable called bch_jmp_mux_sel_o, which is an enumerate (ENUM) data type that decides whether the encountered control transfer instruction is a Jump, Jump relative or Branch instruction. The bch_jmp_mux_sel_o variable is used by the cv32e40x_pc_target circuit. This circuit calculates the target PC value for any control transfer instruction encountered, in other words for any branch/jump instruction.

```
1   assign bch_target_o = pc_target;
2   assign jmp_target_o = pc_target;
3
4   always_comb begin : pc_target_mux
5     unique case (bch_jmp_mux_sel_i)
6       CT_JAL:  pc_target = pc_id_i   + imm_uj_type_i;
7       CT_BCH:  begin
8                    pc_target = pc_id_i   + imm_sb_type_i;
9                end
10      CT_JALR: pc_target = jalr_fw_i + imm_i_type_i;    // Forward from WB,
      but only of ALU result
11      default: begin
12          bch_prediction_id_o = 0;
13          pc_target = jalr_fw_i + imm_i_type_i;
14        end
15    endcase
16  end
```

Listing 4.3: cv32e40x_pc_target

[GIT]

The Listing shows a snippet from the code describing the behavior of the cv32e40x_pc_target circuit. Here the pc_target that is calculated is then stored into a variable called Operand C in ID stage. This will be used by the ALU in the EX stage. If it turns out the condition(s) for the control transfer instruction is satisfied, then the Operand C is output from the EX

stage to be used by the multiplexer in the IF stage as the next PC to be fetched. Operand A, Operand B, and the ALU operator describe what check needs to be performed to satisfy the condition placed by the control transfer instruction. These are all decoded by the Decoder. The alu_operator, along with Operands A, B, and C are provided to the EX stage by the ID stage inside the id_ex_pipe struct.

```
1  // ALU OP C Mux
2   always_comb begin : operand_c_mux
3     case (op_c_mux_sel)
4       OP_C_REGB_OR_FWD:  operand_c = operand_b_fw;
5       OP_C_BCH:          operand_c = bch_target;
6       default:           operand_c = operand_b_fw;
7     endcase // case (op_c_mux_sel)
8   end
9
```

Listing 4.4: Operand C

[GIT]

In the listing 4.2, op_c_mux_sel is an ENUM variable that is output by the Decoder. The EX stage instantiates the Arithmetic Logic Unit. It is this circuit that provides the result regarding the fulfillment of condition for a control transfer instruction. The ALU features a comparator that performs the operation specified by the alu_operator on operand_a and operand_b. The result of the ALU operation is stored in the branch_outcome_o variable, which is a boolean variable. If the branch_outcome_o variable has a value equal to a decimal 1, aka TRUE, the branch condition has been satisfied and the branch needs to be taken. If, however, the branch_outcome_o variable has a value equal to a decimal 0, meaning FALSE, the branch condition was not satisfied and the branch is not to be taken. The branch_outcome_o variable is provided as an output from the EX stage is used by the Controller FSM to make sure that the branch instruction is handled correctly. The EX stage also output the branch target address so that the multiplexer in IF stage can make use of this to fetch the branch target. This means that if the branch_outcome_o variable is TRUE, meaning the branch is to be taken, the Controller FSM needs to kill the instructions in IF and ID stage, and set the pc_mux to PC_BRANCH. As is shown in the Listing 4.2, choosing PC_BRANCH as the pc_mux multiplexer input value sets the branch target to the value defined by the EX stage.

```
1  else if (branch_taken_ex) begin
2              ctrl_fsm_o.kill_if = 1'b1;
3              ctrl_fsm_o.kill_id = 1'b1;
4
5              ctrl_fsm_o.pc_mux  = PC_BRANCH;
6              ctrl_fsm_o.pc_set  = 1'b1;
7
8              // Set flag to avoid further branches to the same target
9              // if we are stalled
10             branch_taken_n     = 1'b1;
11
```

Listing 4.5: Controller FSM Branch handling

[GIT]

It is worthwhile understanding how the Controller FSM circuit works given its importance in the branch handling of the CPU. The Controller FSM is as the name suggests, a Finite

State Machine for the Controller block shown in Figure 4.1. The task of the Controller is to control the instruction pipeline and act as the main controller for the processor core. The FSM is structured in such a way that it allows checks for things to do in an order of importance. The RTL code for the FSM is placed inside an always_comb block and is used in combinational design. A combinational design is a design that is time independent, meaning that the output does not depend on previous inputs. Upon each iteration, it is checked whether the Controller FSM has encountered a RESET state, BOOT_SET state, a FUNCTIONAL state, a SLEEP state, DEBUG_TAKEN state or a POINTER_FETCH stage. For the scope of this thesis, only the RESET and FUNCTIONAL states are important. In the RESET stage, the next state for the FSM is state to be the BOOT_SET state. In the BOOT_SET state, the initial pc_mux input is chosen to allow for selecting which PC address to start fetching from upon boot, in other words start-up. This means that the RESET state essentially restarts the entire CPU. Most of the actions take place in the FUNCTIONAL state in the FSM. This means that given the CPU is not booting up for the first time, reset, set to sleep or any of the other cases mentioned above, it will find itself in the FUNCTIONAL state. In other words normal execution. Naturally then, this is the biggest block of code describing how the FUNCTIONAL state must behave. In the Functional state, a lot of conditions are checked in a declining order of importance. This is to say that in the FUNCTIONAL state the highest priority event condition is checked first. If it is found to be true, then the code block associated with that event is executed. If not, then the next event, which has a lower priority, is checked. This way, during FUNCTIONAL state, the pipeline controller can decide what order the events need to be handled in. Naturally then, it makes sense that the highest priority event in the FUNCTIONAL state is then the arrival of a Non-Maskable Interrupt (NMI). It is a hardware interrupt that standard interrupt-masking techniques in the system cannot ignore. Note here that there is a difference between NMI and maskable interrupts. A Non-maskable interrupt is one that can not be ignored or disabled by the system. An NMI is used typically in cases where response time is critical. Examples of such cases include reporting of non-recoverable hardware issues, or handling the occurrence of a RESET event. If an NMI has not been handled yet, and the flag to decide whether NMI's are allowed are both set, the code block associated with NMI handling is executed. If not the next condition checked is if a DEBUG state is requested, in which case the stages in the pipeline are halted, and the next state for the FSM is set to be the DEBUG state. If neither NMI, nor DEBUG conditions are met, the FSM checks to see if there are any interrupts that need to be handled. This gives us an idea of how the FSM is built to favour events with higher importance. This is important to understand as this feature will be used to our advantage when designing the branch predictor. Without the branch prediction circuitry, the FSM checks if a branch instruction has a "taken" outcome after its evaluation in the EX stage. This is checked in the FSM after all the higher-priority checks mentioned above. The code for this is shown in the Listing 4.2. It is clear now why the pipeline experiences a bubble on the event that the branch instruction evaluates to a taken branch.

Having now looked at how instructions, especially branch instructions propagate through the pipeline, it is easier to understand the execution pattern in the Pipeline.

Figure 4.4 shows the pipeline when a branch instruction is encountered. For the sake of the argument let us assume that the branch is taken. Green rectangles in the figure denote active and valid instructions. Red rectangles denote instructions that have been killed. An instruction that has been killed still travels through the pipeline, however, no computation is done on it. As explained in this section, the outcome of the branch is calculated in the EX stage. It is clear by looking at this figure that the processor core, prior to knowing that the branch would be taken, fetched in instructions PC+1 and PC+2, where PC is the program

	##1	##2	##3	##4	##5	##6
Fetch	bne	PC+1	PC+2	CorrPC	Ins5	Ins6
Decode		bne	PC+1	PC+2	CorrPC	Ins5
Execute			bne	PC+1	PC+2	CorrPC
Writeback				bne	PC+1	PC+2

Figure 4.4: Branch instruction's effect on the Pipeline

counter for the branch instruction itself. This means that instead of branching out to a different PC, the pipeline now has the next instructions assuming that the branch was not taken. When in EX stage it is discovered that the branch was to be taken indeed, as visible in Listing 4.2 the IF, and ID stages are killed. They still propagate through the pipeline but no computation is done on them, effectively rendering them useless in the pipeline. This leads to a 2-stage bubble in the pipeline, where the pipeline has been polluted with useless instructions.

Chapter 5

Intended Design

As the processor core is today, it does not have any branch prediction strategies implemented. This leads to the 2-stage bubble as seen in Figure 4.4. The Project Report presented 3 branch prediction algorithms to help combat this, namely the Backwards Predictor, the History Predictor, and the Hybrid Predictor. For the purposes of modeling, it was assumed that all the required information was available in the IF stage. As can be seen from Chapter4, this is unfortunately, not true. The information about what type of instruction has come into the pipeline, and all other relevant details become available earliest in the ID stage, after the information has been decoded. This means that for algorithms that require the knowledge of the instruction type, the earliest a branch prediction can be made is in the ID stage. This is a significant change and affects how the branch predictor can be implemented.

The corrected results in the previous chapter showcase how many cycles were really saved with each branch prediction technique. For comparison, the Backwards Predictor saved 59,354 cycles, the History Predictor saved 185,958 cycles and the Hybrid Predictor with a history table size of 8 entries saved 193,288 cycles. It is clearly visible that the Backwards predictor saved the least amount of cycles among the three. It is, however, a very interesting result. A table size of 8 entries for the History Predictor would mean that the total number of bits used for storing the History Table would be $8 \times (32 + 32 + 1) = 540$ bits in the worst case. The Hybrid Predictor uses the History Predictor and the Backwards Predictor which means that the total number of bits needed to be stored for the predictor to work would be $8 \times (32 + 32 + 1) = 540$ plus one more bit to store the outcome of the branch if the branch instruction is not found in the table, meaning 541 bits in the worst case. It is evident that both the History and Hybrid predictors use about 540 times more bits than the Backwards Predictor while only saving roughly 3 to 4 times as many cycles. Although, the History and Hybrid Predictors save quite a lot more, they also use a very large area on the processor core. Keeping this in mind, and knowing that the Backwards Predictor has the lowest computational overhead of the three branch prediction techniques, it seems like a wise choice to start out by implementing a Backwards predictor. As seen above, it provides a decent speed-up, while not changing the power, and are demands by too much. It is worthwhile noting here that not all 32 bits are needed of the PC are needed to identify a branch in the History table. It is possible to use less than 32 bits to do so. Doing this would compromise the accuracy of the prediction but would allow for a more stable prediction. This was not modeled in the Project Report, but would make for an interesting area to look into in the future.

The predictor, in and of itself, is simple to understand. It checks whether the branch target is a value that is smaller than the current program counter. If it is indeed smaller, then the branch is predicted to be taken. The rationale behind this being that most branches

are backwards pointing in nature, in the sense that they point to a PC value in that is lower than the current branch. A while loop is a good example of this. Ease of understanding the concept, however, does not mean that the Backwards Predictor is easy to implement. Integrating this branch predictor with the existing design is a challenge. As I shall present in the next chapter, there are quite a few pitfalls when trying to get the prediction circuit to work in sync with the remainder of the pipeline. In the remainder of this chapter, I shall focus on presenting my initially intended design for the branch predictor. Figure 5.1 shows the block diagram for the first suggestion for the implementation of the backwards predictor.

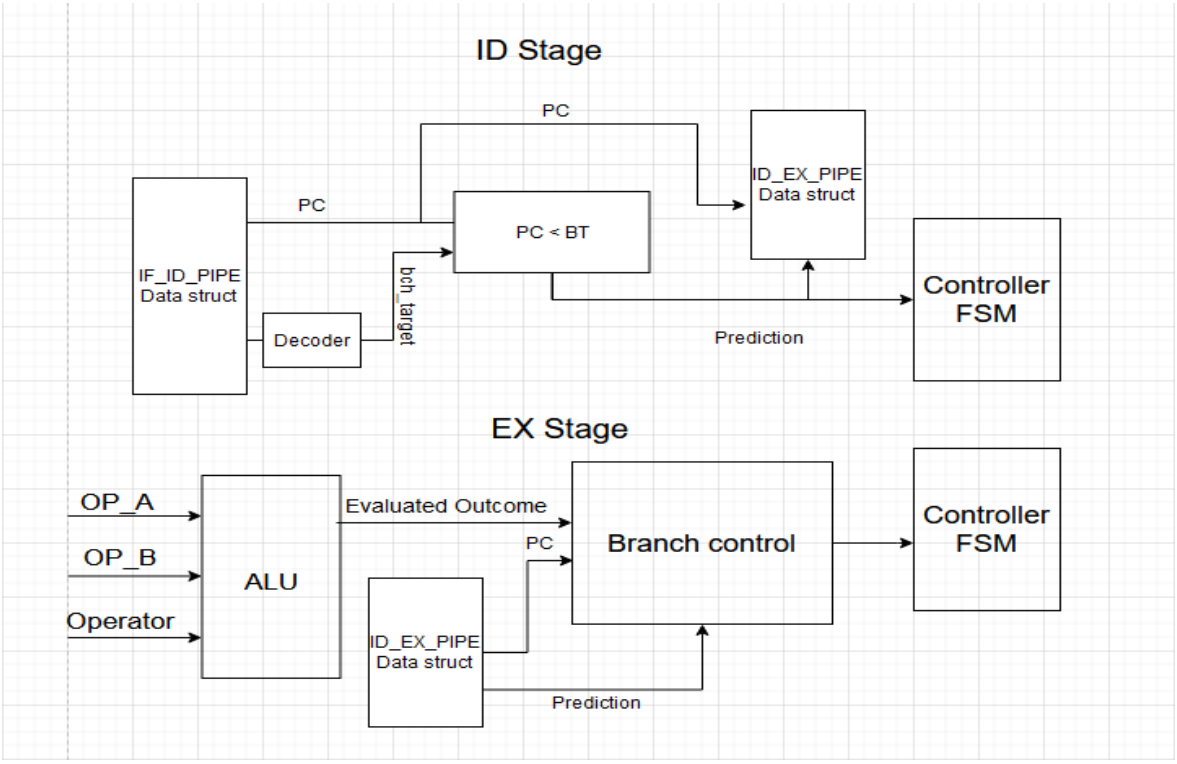


Figure 5.1

To get the Backwards Predictor to work, two important pieces of data are needed. The PC value at which the branch instruction occurred, and the target address of the branch instruction. The PC value is available through the if_id_pipe from the Fetch stage. As mentioned earlier in this chapter, the Decoder only needs if_id_pipe and exception handling inputs to be able to decode the fetched instructions. As can be seen from the Figure 5.1, the Predictor block "PC < BT" would get the PC value from the if_id_pipe and the bch_target in other words, branch_target, as inputs. The Predictor block would then output a binary number based on whether the PC was indeed larger the branch target. This would be the prediction. It requires therefore, that the prediction is checked in the EX stage to see whether the branch is indeed taken. The Prediction is also sent to the Controller FSM. If the Prediction is true, meaning the branch is predicted to be taken, the Controller FSM would kill the IF stage (since this stage would have fetched in the incorrect PC), and set the PC equal to the branch target. The information that is sent to the EX stage includes, PC, along with other information in the id_ex_pipe. The other information includes signals like the ALU_en and setting of the Operands A, B, and C where Operand C is the branch target, if a branch instruction was decoded. In the EX_stage in Figure 5.1, it is visible how the ALU provides an output for whether the branch indeed should be taken or not after performing

the calculation required to be satisfied by the branch instruction. The Prediction from ID stage and the outcome of the branch calculated in the EX stage are then sent to a Branch Control Block. The purpose of this block is to check if the Prediction was the same as the outcome of the branch. Four possible events can occur as a result of checking to see if the Prediction is the same as the outcome of the branch. If the Prediction from the predictor is true, signifying a taken branch, the Controller FSM killed the IF stage without having to wait and see if the branch is really taken in the EX stage. If it is discovered in the EX stage that the branch was mispredicted, the Controller FSM is prompted by the Branch Control block to fix this error by killing both the IF and ID stages and setting the PC to the correct value. The following list explains the different actions taken by the EX stage and the Controller FSM based on the Prediction from the predictor.

- Predicted Taken and Outcome Taken:
  This would mean that the branch was predicted correctly. The Controller FSM killed the IF stage, based on the prediction in the ID stage. This prediction is found to be true in the EX stage and no further correction is required.

- Predicted Taken and Outcome Not Taken:
  The prediction is found to be false in the EX stage correction is required. This would mean that the branch was predicted incorrectly. The Controller FSM needs to also kill the ID stage, after already having killed the IF stage. After this, the Controller FSM needs to set the PC to be equal to the next instruction after the PC that is in the EX stage currently. In other words, PC needs to be set to $id\_ex\_pipe.pc + 2$ for a compressed instruction and $id\_ex\_pipe.pc + 4$ for a normal instruction.

- Predicted Not Taken and Outcome Taken:
  This prediction is found to be false in the EX stage and correction is required. This would mean that the branch was predicted incorrectly. The Controller FSM must kill the IF stage, and ID stage, and set the PC for the next stage to be equal to the branch target. This is done by choosing PC_BRANCH in the pc_mux in the IF stage. This is shown in Listing 4.2.

- Predicted Not Taken and Outcome Not Taken:
  This would mean that the branch was predicted correctly. The Controller FSM need not make any corrections.

The development of the design for the first implementation idea of the Backwards predictor happened roughly in parallel with trying to become more familiar with the RTL code. This was the reason why the assumption was made that the Decoder was responsible for calculating the branch target. As seen earlier in this chapter, this is not true. The cv32e40_pc_target circuit is the one responsible for this. This was one obvious flaw with the design. Upon discussion with my external supervisor, it was also discovered that having the Branch control in the EX stage beats the purpose of having a dedicated Controller FSM block. The sole purpose of the Controller FSM was to manage the instructions in the pipeline. Asking the stage circuits to take upon managing tasks degrades the value of the Controller FSM. These were thus some things that needed to be fixed with the first proposed implementation of the Backwards Predictor.

Concluding this chapter here seems correct, as the introduction to my proposed implementation for the Branch Predictor, including its preliminary shortcoming has been completed. The next Chapter contains the information regarding the work done in order to get the Backwards Predictor running on the CV32E40X core.

Chapter 6

Implementation of the Backwards Predictor

This chapter shall focus exclusively on documenting the work done in order to get the Backwards Predictor to work with the CV32E40X processor core. Chapter 5 presented the preliminary block diagram for the Backwards predictor in Figure 5.1. It also introduced the problems with the presented idea. Since this chapter shall focus on implementing the Backwards Predictor, it is best to start with trying to improve the block diagram in Figure 5.1.

Starting with the most obvious mistake, the bch_target is to be calculated not from the Decoder, but rather from the cv32e40x_pc_target circuit. Since this circuit is already doing calculation of the branch target, it makes sense to incorporate the branch prediction also in the same circuit. This would allow for the overall design to appear much cleaner. In order to accomplish this, the cv32e40x_pc_target circuit was changed to look like the one shown in the Figure A

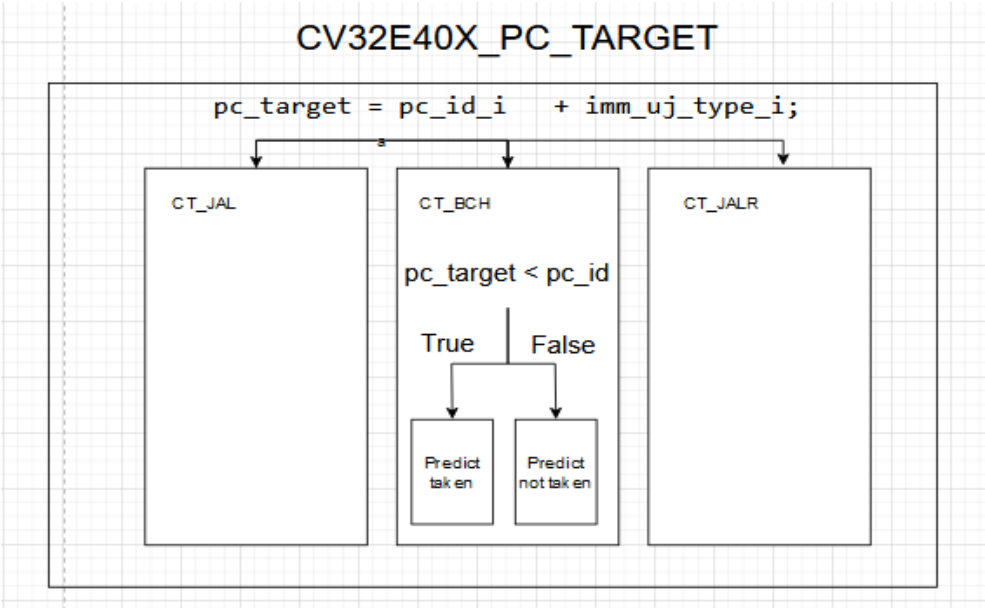


Figure 6.1

As shown in the Figure 6.1, the cv32e40x_pc_target has been changed to include the comparator which checks whether the pc_target variable is smaller than the pc_id_i, which

is the PC value for the instruction in the ID stage. The variable imm_uj_type_i is a signed 32-bit number, meaning that the 32^{st} bit (imm_uj_type_i[31]) determines whether or not this variable is a negative number. Given how pc_target is calculated by adding pc_id_i and imm_uj_type_i, it is visible how the pc_target would evaluate to a lower value than pc_id_i when the imm_uj_type_i is a negative number. In this case, i.e. when pc_target is lower than the pc_id_i, the branch must be predicted to be taken. The prediction is output from the cv32e40x_pc_target circuit. The ID stage stores the output from the cv32e40x_pc_target circuit in a logic type variable, called bch_outcome. The bch_outcome is then output from the ID stage circuit. The Controller FSM uses this to handle a branch instruction event. The revised block diagram for the Backwards Predictor looks like the one shown in Figure 6.2.

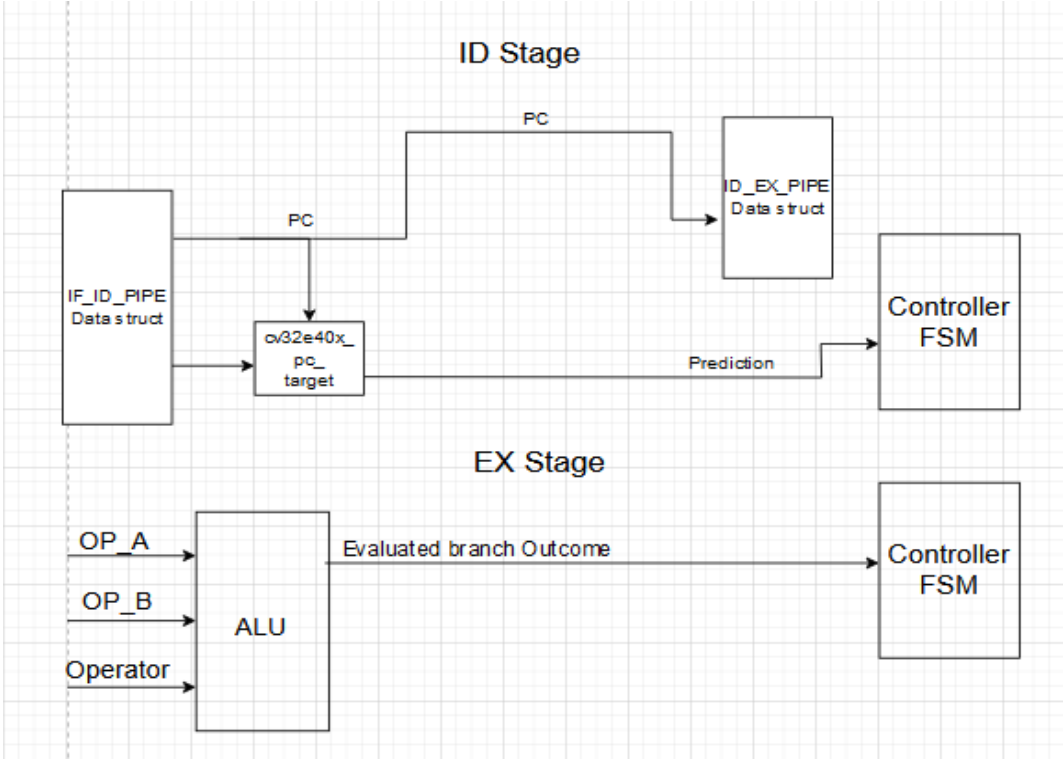


Figure 6.2

The Controller FSM circuit is responsible for assuring that the branches that need to be taken, are indeed taken. The details regarding how the FSM works were mentioned in Chapter 4. From its original functionality, the Controller FSM needed to be changed to allow for the branch prediction to have an impact on the execution. In other words, the Controller FSM should not only receive the prediction for any and all branch instructions, but also act upon it. This means changing the next PC value to be fetched, in accordance with the prediction. As can be seen in the code Listing 4.2, the method for changing the next PC value to be fetched in, upon a taken evaluation of a branch instruction is already done in the FSM already. This part of the Listing needed to be expanded and adapted to allow for incorporating branch prediction. This will become clear shortly.

It is this that makes the implementation more tedious than the modeling. In the modeling of the Backwards Predictor, it was assumed that the prediction would be handled, meaning that an external circuit would strive to always fetch in the predicted target instructions, and take care of cases when the prediction didn't match the outcome. This "external circuit" was

treated as a black box in the modeling since it's behavior was assumed to be correct and such that it would satisfy the extra demands placed by the branch prediction. In the hardware implementation of the branch predictor however, not only was the predictor added to the CV32E40X processor core, but the Controller FSM was also tweaked to work together with the predictor.

It is worthwhile noting that each stage in the pipeline, given a full pipeline, has a different PC value. This is because each stage is designed to build handling different instructions. Given this, in order to verify if the prediction made by the Backwards Predictor is correct after the evaluation of the branch condition in the EX stage, a method for storing the Prediction from the ID stage had to be found. This was accomplished by adding a data field in the id_ex_pipe data structure called "bch_prediction_from_id". As the branch instruction moved from the ID stage to the EX stage, the prediction was also passed on to the EX stage. This is done using pipeline stage data structs. Each stage transition has one such struct. The if_id_pipe is a struct for transporting data between the IF and the ID stage. The id_ex_pipe struct transport data between the ID and EX stage, and finally the ex_wb_pipe transports data between the EX and the WB stage. In the Controller FSM module, when it is discovered that there is a branch instruction in the EX stage, the Prediction from the ID stage for that instruction is retrieved from the data struct. Figure 6.3 shows the block diagram for how the branch instruction handling happens in the Controller FSM.

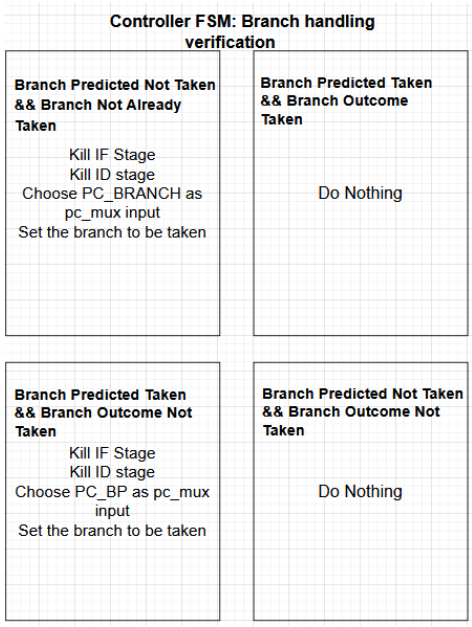


Figure 6.3

For each branch prediction, there will be 4 possible combinations of the prediction versus the real branch outcome. This was explained in the Chapter 5. This brings up an important milestone in the design process. As mentioned in the previous paragraph, each pipeline stage may contain information about different instructions in the pipeline. This means that for the branch verification to work properly, it is important that the correct information is used. Not only this, it was also important to decide the priority of events for their placement in the FSM.

Listing 6 shows the necessary variables for branch prediction handling in the FSM. Listing 6 shows the code for the first attempt at trying to getting the branch predictor to work

with the FSM. Essentially, in Listing 6, the branch verification idea shown in Figure 6.3 is implemented The simulation failed with this setup. The reason for this is explained below.

```
1      logic branch_taken_id;
2    // Events in EX
3      logic branch_in_ex;
4      logic branch_taken_ex;
5    // Blocking on branch_taken_q, as a branch has already been taken
6    assign branch_in_ex = id_ex_pipe_i.alu_bch && id_ex_pipe_i.alu_en &&
       id_ex_pipe_i.instr_valid && branch_decision_ex_i;
7    assign branch_taken_ex = branch_in_ex && !branch_taken_q;
8    //Branch handling when decode stage predicts a branch should be taken
9    assign branch_taken_id = id_ex_pipe_i.alu_bch && id_ex_pipe_i.alu_en &&
       id_ex_pipe_i.instr_valid && id_ex_pipe_i.bch_prediction_from_id && !
       branch_taken_q;
10
```

Listing 6.1: Variables needed for Branch Handling

[GIT]

```
1      end else if (branch_taken_id) begin
2          ctrl_fsm_o.kill_if = 1'b1;
3          ctrl_fsm_o.pc_mux  = PC_JUMP;           //Can Use PC_Branch here too,
       looks like the code does the same thing?
4          ctrl_fsm_o.pc_set  = 1'b1;
5          // Set flag to avoid further branches to the same target
6          // if we are stalled
7          branch_taken_n     = 1'b1;
8      end else if (branch_taken_ex) begin
9          if (!(id_ex_pipe.bch_prediction_from_id)) begin
10         ctrl_fsm_o.kill_if = 1'b1;
11         ctrl_fsm_o.kill_id = 1'b1;
12         ctrl_fsm_o.pc_mux  = PC_BRANCH;
13         ctrl_fsm_o.pc_set  = 1'b1;
14         // Set flag to avoid further branches to the same target
15         // if we are stalled
16         branch_taken_n     = 1'b1;
17           end else begin
18          // Do nothing when the branch is predicted taken in Decode stage and
       is actually taken in the execute stage
19         end
20       end  else if (!branch_taken_ex) begin
21         if ((id_ex_pipe.bch_prediction_from_id)) begin
22         ctrl_fsm_o.kill_if = 1'b1;
23         ctrl_fsm_o.pc_mux  = PC_ID;
24         ctrl_fsm_o.pc_set  = 1'b1;
25         branch_taken_n     = 1'b1;
26         end else begin
27         //Do nothing if the branch is
28         end
29
```

Listing 6.2: Branch handling with prediction

[GIT]

As mentioned in Chapter 4, the FSM checks and executes things based on their priority. Listing 4.2 shows how branch handling is done based on the outcome, evaluated in the

EX stage, of the branch instruction. During debugging, a question regarding whether the prediction from ID stage was to have a higher Priority than the real evaluation of the branch instruction was brought up. The answer, rather straight-forward, was no. The prediction takes place in the ID stage, while the real outcome of the branch is evaluated in the EX stage. This means that the data regarding the branch outcome is more reliable when coming from EX stage, as this is the real outcome of the branch instruction. The data from ID stage is merely a prediction. There is no guarantee that a predicted instruction actually turns out to be taken. Whether branch is taken or not for real can thus only be figured out in the EX stage. It then makes sense to have the EX stage event for branch handling, listed in Listing 4.2 as a higher priority than the prediction. Knowing this, it is slightly easier knowing where the branch predictor was to be added to in the FSM. This was then one thing that needed to be changed right away. It was also observed that even the branch_taken_id, which held the predicted branch outcome from ID stage was calculated incorrectly. In Listing 6, it is visible that the branch_taken_id used variables from the id_ex_pipe struct. As explained earlier, this struct carried data from the ID stage to the EX stage, in other words, this was data available during the EX stage, and not the ID stage. Thus, even though the prediction itself was done in the ID stage, it was not used until the EX stage, effectively annulling any advantage branch prediction would have showed. Furthermore, there was an error in correcting for a mis-predicted branch outcome in Listing 6 lines 20-26. To understand this better consider the following figures. They have been labeled step 1, step 2, and step 3 respectively.

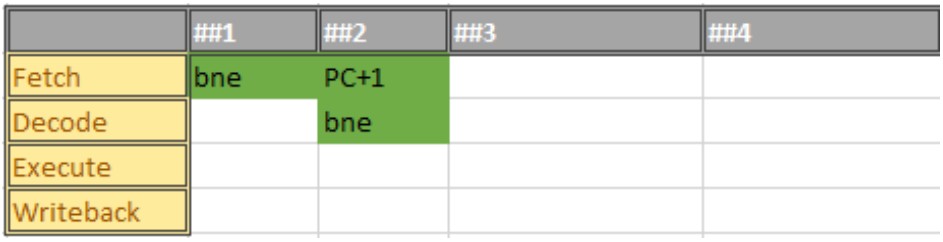


Figure 6.4: Step 1

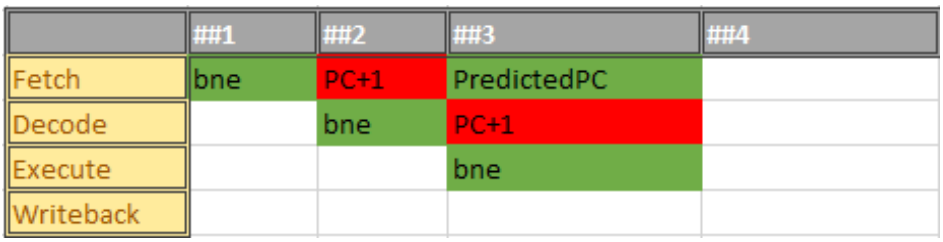


Figure 6.5: Step 2.

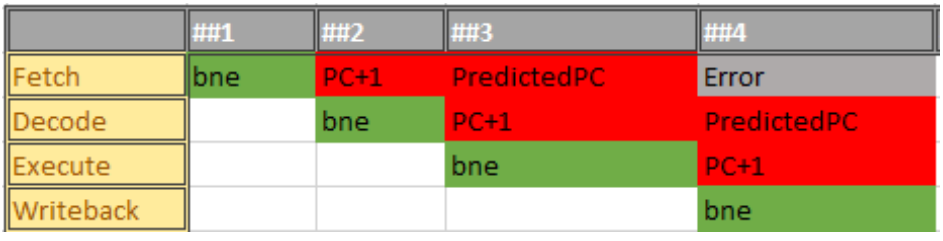


Figure 6.6: Step 3.

The following assumptions and over-exaggerations will help understanding the bug better:

– For the sake of keeping the illustrations, and explanation as clean as possible, the

pipeline is not shown to be full.

- Assume that the backwards predictor is working on the processor.
- The type of instruction fetched after the bne could be any possible instruction, for instance add or ld (load), even another branch instruction. To keep the explanation simple, it is assumed here that no branch instruction entered the pipeline after the first bne instruction.
- Even though the pipeline circuitry works combinationally, meaning that there are no delays from input to output, it is easier to highlight the bug considering that a slight delays exists between the instructions entering the different stages to the stabilization of all values in that stage.
- Assume that the branch was predicted taken in the ID stage, but is later evaluated to be not taken in the EX stage.

During clock cycle 1, the bne (branch if not equal) instruction is fetched. The processor does not know what this instruction means, only that "an instruction" has entered the pipeline. When the bne instruction moves from the IF stage to the ID stage, the pipeline fetches in instruction with a PC value PC+1. The type of instruction fetched does not matter, thus the next instruction is merely denoted using a PC+1 denotation, simply implying that the next instruction was fetched. When PC+1 enters the IF stage before the prediction is ready in the ID stage, the instruction in the IF stage is valid. When the ID stage prediction is ready, and it is predicted that the branch will be taken, then the IF stage instruction is killed, and the pc_mux input is selected to be PC_JUMP as shown in Listing 6. When the bne instruction is then moved into the EX stage, the killed IF stage is brought into the ID stage and the predicted instruction at the branch target, i.e. the predicted PC is fetched into the IF stage. When the EX stage has evaluated the branch and finds that the branch was **not** to be taken, it kills the IF stage and sets the pc_mux to be PC_ID, in other words equal to the PC value available in the ID stage. This induces two bugs in the code. As mentioned in Chapter 4, a killed instruction may pass through the pipeline, but no computation is done upon it. This means that when the pipeline goes from Figure 6.5 to 6.6, and sets the pc_mux to be equal to PC_ID, this operation fails. This happens because the instruction PC+1 was killed in the IF stage. This means that upon entering the ID stage, since the instruction is not "alive", the ID circuitry does not update the PC_ID value to PC+1. Furthermore, from Listing 4.2, it is visible that the pc_mux has no input called PC_ID. So in clock cycle 4, it shows that an error has occurred. The idea about fetching PC+1 is correct, however, the way it was implemented was not. This is why the first attempt at a hardware implementation of the predictor failed. Having now understood all the bugs in the first attempt, they were attempted to be fixed in the second iteration of the backwards predictor design.

```
1  assign branch_in_ex = id_ex_pipe_i.alu_bch && id_ex_pipe_i.alu_en &&
     id_ex_pipe_i.instr_valid;
2  assign branch_taken_ex = branch_in_ex  && branch_decision_ex_i;// && !
     branch_taken_q;
3  assign branch_outcome_id = id_ex_pipe_i.bch_prediction_from_id;
4  //Branch handling when decode stage predicts a branch should be taken
5  assign branch_taken_decode = alu_bch_bp_i && alu_en_raw_id_i && if_id_pipe_i
     .instr_valid && bch_prediction_from_id_i && !jump_taken_q;
6
```

Listing 6.3: Branch variables Attempt 2

[GIT]

Listing 6 shows the variables needed to correct the branch prediction algorithm. Compared to Listing 6, it is visible that the calculation for branch_taken_decode has been corrected to include data from the ID stage, and not EX stage. The biggest changes however, are made to the branch verification code shown in 6. The priority of events here is fixed such that the FSM first checks if there are any branch instructions in the EX stage that need to be handled. If not, then it checks the ID stage to see if there are any branch instruction in the ID stage, and if they were predicted taken. It is visible from the Listing 6, that control signals like branch_taken_q, branch_taken_n, jump_taken_q, and jump_taken_q are used. These signals make sure that in case of a pipeline stall (an event where instructions are stuck in the same pipeline stage over multiple cycles), the same branch is not taken multiple times. Also in Listing 6, it is visible that the FSM sets the pc_mux input to be equal to PC_BP. This is one change that was made to the pc_mux in the IF stage to account for the error visualized in Figures 6.4, 6.5 and 6.6. Choosing this input sets the PC value for the next fetch to be equal to the PC value of the instruction in the EX stage plus the decimal number 2 for a compressed instruction, or 4 for non-compressed instructions. This is the equivalent of fetching the next instruction in case the branch instruction is not taken. This is also shown in the following code listing:

```
1     PC_BP: branch_addr_n = id_ex_pipe_i.pc + (id_ex_pipe_i.instr_meta.
      compressed ? 32'd2 : 32'd4);
2
```

Listing 6.4: Branch state Attempt 2

[GIT]

```
1     end else if (branch_in_ex && branch_taken_ex) begin
2       if(!branch_outcome_id && !branch_taken_q) begin
3         ctrl_fsm_o.kill_if = 1'b1;
4         ctrl_fsm_o.kill_id = 1'b1;
5         ctrl_fsm_o.pc_mux  = PC_BRANCH;
6         ctrl_fsm_o.pc_set  = 1'b1;
7         // Set flag to avoid further branches to the same target if stalled
8         branch_taken_n     = 1'b1;
9       end
10    end else if (branch_in_ex && branch_outcome_id) begin
11      if(!branch_taken_ex && !branch_taken_q) begin
12        ctrl_fsm_o.kill_if = 1'b1;
13        ctrl_fsm_o.kill_id = 1'b1;
14        ctrl_fsm_o.pc_mux  = PC_BP;
15        ctrl_fsm_o.pc_set  = 1'b1;
16        // Set flag to avoid further branches to the same target if stalled
17        branch_taken_n     = 1'b1;
18      end
19      //If the branch is shown to be !Taken in EXECUTE and also predicted !
      Taken in DECODE do nothing
20  end else if (branch_taken_decode) begin
21      ctrl_fsm_o.kill_if = 1'b1;
22      //ctrl_fsm_o.kill_id = 1'b1;     Need this in EXECUTE stage to check if
      the Branch was predicted correctly
23      ctrl_fsm_o.pc_mux  = PC_JUMP;
24      ctrl_fsm_o.pc_set  = 1'b1;
25      // Set flag to avoid further branches to the same target if stalled
26      jump_taken_n     = 1'b1;
```

```
27
```

Listing 6.5: Branch variables Attempt 2

[GIT]

This algorithm worked much better than the previous implementation, and didn't fail, however, it didn't pass either. An interesting bug was discovered with this implementation. To understand this, we need to look at how the branch_taken_n, and jump_taken_n values are set and what they do. The branch_taken and jump_taken flags are stored using flip-flops. After each iteration of the FSM, where the relevant code block is executed, the input of the flip-flop branch_taken_n and jump_taken_n flip-flops are propagated to the outputs. This means that the next time the value stored in the flip-flop, in other words, the flip-flop's output branch_taken_q and jump_taken_q are read, and they indicate whether branch/jump has been taken or not. Naturally then, these flags need also to be reset so that branches/jumps are taken appropriately. It is here that the RTL code was sent into a deadlock. Consider the following code listing.

```
1      if (branch_taken_q && id_valid_i && ex_ready_i) begin
2        branch_taken_n = 1'b0;
3       end
4       if (jump_taken_q && if_valid_i && id_ready_i) begin
5        jump_taken_n = 1'b0;
6      end
7
```

Listing 6.6: Branch state Attempt 2

[GIT]

The branch_taken_q flag is used in the calculation of the variable branch_taken_decode. This means that while the branch_taken_q flag is set, Listing 6 will not work as expected. It is visible that line 2 and line 11 both check to see if the branch_taken_q flag is 0 before executing any code block. While this doesn't happen, the necessary kills can not be made and thus pipeline becomes polluted with incorrect instructions. This variable instead needs to be cleared when the fetch into IF is valid and the ID stage is ready to receive instructions. This happens because if the predictor predicts that a branch must be taken, and then it is evaluated to be not taken in the EX stage, the ID stage is killed, meaning that this stage is no longer valid. This prohibits the flag from being cleared.

```
1      if (branch_taken_q && if_valid_i && id_ready_i) begin
2        branch_taken_n = 1'b0;
3       end
4
```

Listing 6.7: Branch state Attempt 2

[GIT]

Listing 6 shows the corrected version of this flag clearing. With this change in place, the backwards predictor worked flawlessly with the Controller FSM in turn leading to a successful simulation.

The implementation as it is presented worked, and the results will be presented in the Chapter 7. Before this section is brought to a finish, it is important to consider probable optimizations in the RTL code that can produce the same results while using less area on the chip. Most of the Control FSM code is as optimized as can be, and no excessive hardware is

being used. In the cv32e40x_pc_target however, the branch prediction is done by comparing the numbers pc_id_i and pc_target. Instead of doing this, checking the 32^{nd} bit of the imm_uj_type_i (imm_uj_type_i[31]) will be more than sufficient. As explained earlier in this section, this is a signed number and thus having the MSB equal to would mean that the pc_target = pc_id_i + imm_uj_type_i will lead to a pc_target value lower than that of the pc_id_i. The reason why this is an important optimization is because it avoid having to synthesize a 32-bit magnitude comparator.

A magnitude comparator is a digital combinational circuit that compares two binary numbers to check whether they are equal in size, greater than or smaller than each other.

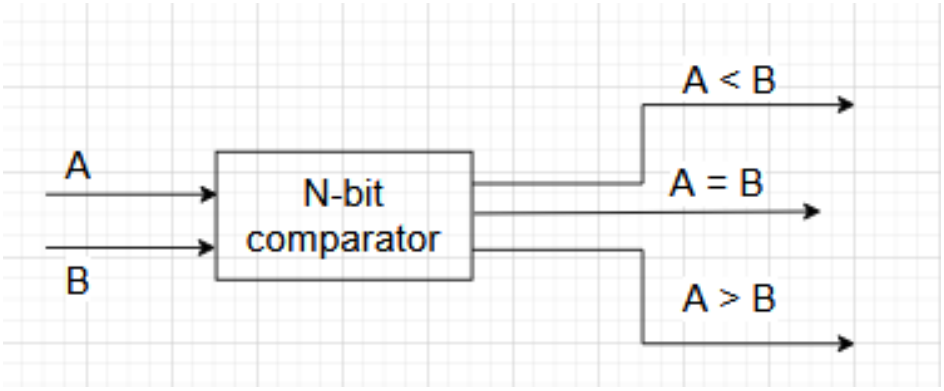


Figure 6.7: N-bit comparator

Figure 6.7 shows the block diagram for an N-bit comparator. The ALU which is connected to the EX stage has access to such a comparator. For making a prediction in the ID stage however, a comparator was synthesized by the simulation software QuestaSim ©. Although such a comparator was not constructed from scratch for the scope of this thesis, to understand the impact of including a magnitude comparator in the prediction design, it has to be introduced here briefly. For comparing two 32-bit numbers, 8 cascading 4-bit comparators can be used.

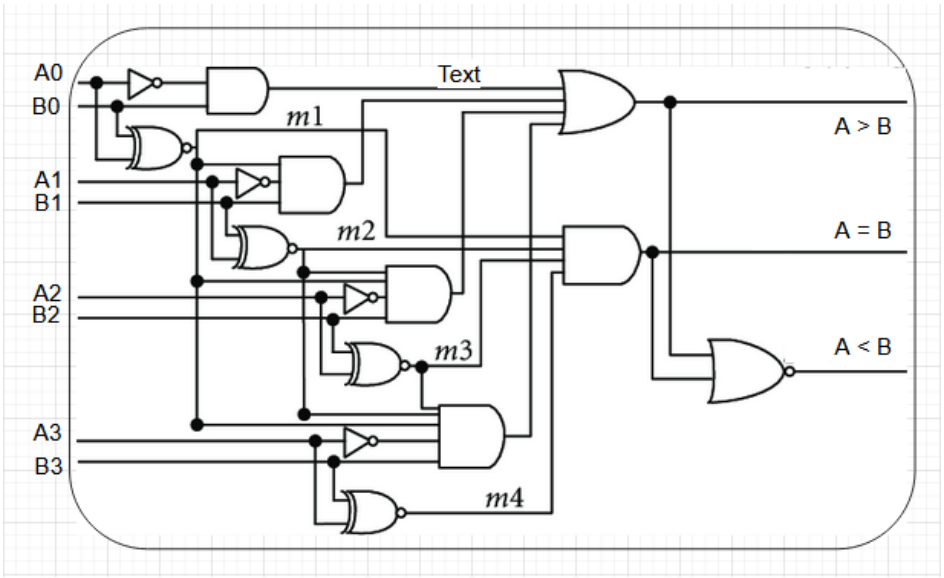


Figure 6.8: 4-bit comparator gate level diagram

Notice how Figure 6.8 uses 15 digital gates for comparing two different 4-bit numbers. For a 32-bit scenario, we would need 8 of these and an additional AND gate to compare two 32 bit numbers. That is an additional 121 digital gates, where just checking one bit (imm_uj_type_i[31]) using a NAND gate would produce the same result. Not only would the 32-bit comparator use significantly more area, it would also increase the power consumption. Since the circuit would be combinational in nature, theoretically no delays would occur. In practice, however, gate level delays are real, and with 121 gates to compare two numbers, would induce a delay to the circuit.

This concludes the discussion about the implementation of the backwards branch predictor for the CV32E40X processor core. The next chapter talks about the results obtained from the experimentation process.

Chapter 7

Results

Having looked at how the implementation of the backwards predictor was done in the previous chapter, this chapter will discuss the results from the simulation. In order to find out whether a simulation passed or not, the Universal Verification Methodology (UVM) Report Summary could be used.

```
# --- UVM Report Summary ---
#
# Quit count :     0 of     5
# ** Report counts by severity
# UVM_INFO :   84
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [BASE TEST]     5
# [BUSERRSB]     7
# [CFG]     1
# [CLKNRST]     4
# [CORESB]     3
# [CV32E40XCORECTRLAGT]     1
# [DEBUGCOVG]     1
# [END_OF_TEST]     1
# [FENCEI_MON]     2
# [FETCHTOGGLE]     1
# [INTERRUPTCOVG]     1
# [MEMREADMEMH]     1
# [OBIVPSEQ]    37
# [OBI_MEMORY_MON]     4
# [OVPWRAP]     1
# [PMASB]     2
# [RNTST]     1
# [RST_VSEQ]     3
# [RVVIOVPAGT]     1
# [TEST]     4
# [TEST_CFG]     1
# [TEST_DONE]     1
# [UVM/RELNOTES]     1
#
# ** Note: $finish    : /eda/tools/mentor/questasim.2021.4_2/verilog_src/uvm-1.2/src/base/uvm_root.svh(517)
#    Time: 68586300 ps  Iteration: 69  Instance: /uvmt_cv32e40x_tb
#
# uvmt_cv32e40x_tb.end_of_test: *** Test Summary ***
#
#     PPPPPPP    AAAAAA    SSSSSS    SSSSSS   EEEEEEEE  DDDDDDD
#     PP    PP  AA    AA  SS    SS  SS    SS  EE        DD    DD
#     PP    PP  AA    AA  SS        SS        EE        DD    DD
#     PPPPPPP   AAAAAAAA   SSSSSS    SSSSSS   EEEEE     DD    DD
#     PP        AA    AA        SS        SS  EE        DD    DD
#     PP        AA    AA  SS    SS  SS    SS  EE        DD    DD
#     PP        AA    AA   SSSSSS    SSSSSS   EEEEEEEE  DDDDDDD
#     ------------------------------------------------------------
#                         SIMULATION PASSED
#     ------------------------------------------------------------
# End time: 18:59:56 on Jun 14,2022, Elapsed time: 0:00:28
# Errors: 0, Warnings: 15
```

Figure 7.1: Example of a passed UVM report

Figure 7.1, or variants where the message PASSED was displayed conveyed that the simulation has passed. This message by itself was used as a way to figure out whether or not a simulation passed. It conveys no real information about the number of instructions run, or the amount of cycles wasted/saved/used in carrying out a test. It is worthwhile mentioning that this UVM report, and all others that will be mentioned in this chapter are all generated as a result of running the hello-world test on the CV32E40X on the processor core. This was a custom test run on the processor core to check if it worked as it was expected to. Upon successful simulation the following was also printed out into the console:

```
# HELLO WORLD!!!
# This is the OpenHW Group CV32E40X CORE-V processor core.
# CV32E40X is a RISC-V ISA compliant core with the following attributes:
#       mvendorid = 0x602
#       marchid   = 0x14
#       mimpid    = 0x0
#       misa      = 0x40001104
#       XLEN is 32-bits
#       Supported Instructions Extensions: MIC
```

Figure 7.2: Example of hello world message

The results from the simulation were contained in the runtime log which was made available in the results folder after each simulation. In order to understand the the information that was available from each runtime log, consider the following figure:

```
        TIME | RVFI |    CYCLE |  ORDER |      PC |    INSTR | M | RS1 | RS1_DATA | RS2 | RS2_DATA |  RD |  RD_DATA | MEM | MEM_ADDR | MEM_DATA | INSTRUCTION
    528.000 ns | RVFI |      170 |      1 | 00000080 | 00003197 | M | x0  | 00000000 | x0  | 00000000 | x3  | 00003080 |  --  | -------- | -------- | _start crt0.S 26 - auipc x3,0x3
```

Figure 7.3: Example runtime log

Figure 7.3 shows the format of the runtime log with one example line in the log. The column Cycle includes the clock cycle count for running the hello-world program on the processor core. Order column includes data about the number of instructions run by the processor core. The order counter is incremented by 1 for each completed instruction. PC shows the current PC value, and the Instruction column shows what assembly instruction was run. In the example log line, the instruction that was run was an auipc which stands for Add upper immediate to PC. This instruction adds a 20-bit immediate value to the upper 20 bits of the program counter. In order to understand the results from running the backwards-predictor, we need to consider the following two fields: "Cycle" and "Order". Before we can analyze the results, we need a benchmark performance which runs with no backwards prediction. This can be run by commenting out the backwards predictor from the cv32e40x_pc_target RTL code. In order to get the most reasonable and meaningful data, stalls in the pipeline also need to be simulated in the simulation. The results from running these simulations are presented in the following table:

Type of data	Benchmark (no stalls)	Benchmark (with stalls)	Backwards (no stall)	Backwards (with stalls)
Cycle	22758	33410	21790	33085
Order	14223	14223	14223	14223

Table 7.1: Results from the simulation

From Table 7.1, we can see how the branch predictor behaves under different conditions. One way of sanity checking the results is to confirm that all the instances ran the exact same number of instructions. This means that adding a branch predictor did not make the processor core skip instructions. Seeing a lower total instruction count on results with branch predictor would mean that the predictor caused the processor core simulation to fail, due to not running the correct number of instructions. Comparing the benchmark (no stalls) to the backwards predictor(no stalls) shows that the cycles saved was equal to $22758 - 21790 = 968$ cycles whereas in the case of experienced pipeline stalls the cycles saved was equal to $33410 - 33085 = 325$ cycles. This means that in the case of no stalls, the cycles saved for the backwards predictor was 4.25%, whereas in the case of the backwards predictor with stall, 0.97% of the total cycles were saved. This goes on to show that the predictor works best in the case of a pipeline that has no stalls. In the case of stalls in the pipeline, the results are less impressive. For the same test program, the model for the backwards predictor developed for the Project Report, which assumed that the prediction could be made in the IF stage showed that around 1014 cycles could be saved in the best case. This equates to a total cycles saved percent of that equal to 4.6 %.

There are a few important takeaways from these results. The first and foremost being that the backwards predictor does indeed work. It does save cycles while using minimal possible extra resources on the processor core. It is worthwhile noticing that for each correct prediction, 1 cycle is saved, since the prediction is made in the ID stage which is 1 cycle away from the EX stage. For every mispredict however, 2 cycles are lost. This goes on to show that backwards predictor works better with software that has mostly backwards branches. This is an important takeaway. During modeling of the backwards predictor, it was assumed that the prediction could be made in the IF stage meaning that for every correct prediction, 2 cycles could be saved. The number of cycles wasted on mispredicted branches, however, did not change. This meant that the predictor did not appear to be biased towards backwards pointing branches when it came to producing results. With the numbers obtained above, it is arguable that the backwards predictor may even lead to worse results in the case that the software has more forward facing branches. This means different things for a software designer compared to a digital designer. From a software perspective, it may help to design software that contains and takes mostly backwards facing branches. Doing the opposite would negatively impact the performance of the processor core. This would mean that it is worse than having no branch prediction algorithm at all. From a digital designer perspective, it may be beneficial to look into when to use the backwards predictor. One possible solution is if maybe, during the compile time, when the compiler looks at the code, it could decide whether to turn on the backwards predictor to exploit it's benefits. Machine learning could also be incorporated here to further improve the results. It is possible that a separate circuit looks at the accuracy of the branch predictor and turns it on or off during regions of code to make sure that the predictor is being used when there is a higher chance of success, and turned off otherwise. This would require more physical circuitry on the processor core, but could potentially lead to better overall performance. Switching off the branch predictor during regions where the "branch predictor watcher" circuit thinks the branch predictor may do more harm than good, could also lead to saving power, as the processor core would not need to correct for quite as many mispredicts.

Chapter 8

Conclusion

The goal of this thesis was to understand how a branch prediction algorithm running on the CV32E40X RISC-V processor core would it's performance. After having modeled various prediction algorithms, implementing the backwards predictor and analyzing the results, the main goals of this thesis have been accomplished. Throughout the scope of this thesis, a better understanding of the CV32E40X processor core was developed and a Branch prediction algorithm was implemented in hardware. It is clear that the modeling of the algorithms gave accurate indications of what would happen for the hardware implementation. Comparing the modeling results to the hardware implementation, it is clear that the modeling results showed better results, and hid the bias for backwards branches discovered in the hardware implementation. As mentioned in the last chapter, the results from the hardware implementation are not in vain. They do allow for making interesting remarks about the backwards predictor. The last chapter also presented ideas regarding how the results could be improved while trying to use the backwards predictor. As mentioned earlier in the report, the Backwards Predictor is one of the simplest branch predictors available. The CV32E40X core had no dedicated branch prediction algorithm in place, which is the same as always predicting the branch outcome to be not taken. This is another type of static branch prediction algorithm. The results presented in the last chapter, essentially compared the "Always not taken" predictor to the Backwards predictor.

8.1 Future Work

If this thesis was to be developed further, it would be very appealing to look into the hardware implementation of either a Global History Predictor, or a Hybrid Predictor. Both of these were, at least during modeling for the Project Report, able to make predictions in the IF stage. This could potentially be carried over into the hardware implementation. The question then would be how to implement this. The first thing required for both the aforementioned algorithms would be to create a history table, which would be capable of holding the branch outcome history for a given number of branches.

Figure 8.1 shows a branch outcome history table which can be designed using an array. Each element in the array could be made to hold a 65 bit integer (non-optimized design), where the first 32 bits are used to store the identifier for the branch, this can for example be the PC value at which this branch was encountered. The next 32 bits could be used to save the target address of the branch in case it is taken, and the last bit could then be used to store the last outcome of that branch. Number of bits needed to store the branch address can not be reduced, as all the bits are necessary. However, at the cost of accuracy of prediction, the identifier bits could be reduced. For example, using 31 bits to store the identifier would

Branch identifier	Branch target	Outcome
0x12ac94b1	0x05081ecc	Taken
0x12a3e4b2	0xa91e94b6	Not Taken
0x77ac94b3	0x0aab3122	Not Taken
0x12ff94b4	0x74729acc	Taken
0x12ac4654	0x00bade1	Not Taken
0xa91e94b6	0x050cd0c	Taken
0x12ffcc123	0x134134ca	Taken
0x0685aebd	0x98bcfaed	Not Taken

Figure 8.1: Example history table

mean that there is a $\frac{2^{31}}{2^{32}} \times 100 = 50\%$ chance of finding the identifier of a branch in the array table. Reducing the number of identifiers bits further would decrease the probability of finding the correct identifier in the table, while also significantly reducing the resource usage. Spatial locality can be used at an advantage here, by assuming that the identifier available in the history table is close to the identifier being looked up in the table. Realistically, storing 5 to 8 bits of the identifier in the table should suffice while providing a significant chance of success.

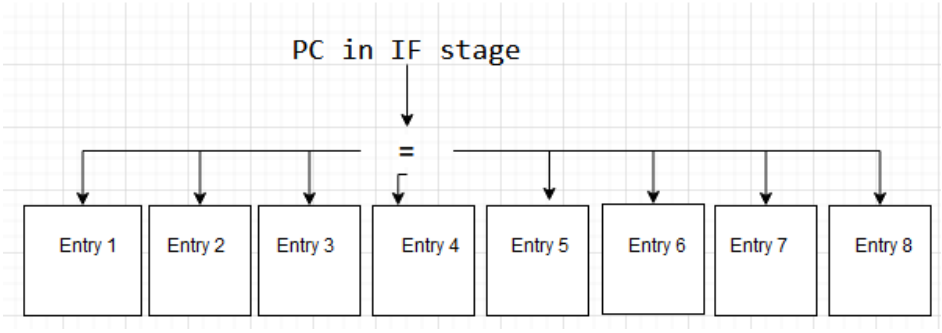


Figure 8.2: Checking to history table

Figure 8.2 shows how the PC that was fetched into the IF stage can be searched for in the branch history table. This can be done by performing a bitwise AND operation on the identifier bits in the history table and an equal number of bits from the PC in IF stage. If the bit-wise AND operation returns a 1, it can be assumed that the PC was found in the history table. This may not be the case given how, only some bits of the identifier can be used to identify a branch. If the entry in the table seems to contain the PC that entered the IF stage, the following figure shows the handling action that can take place:

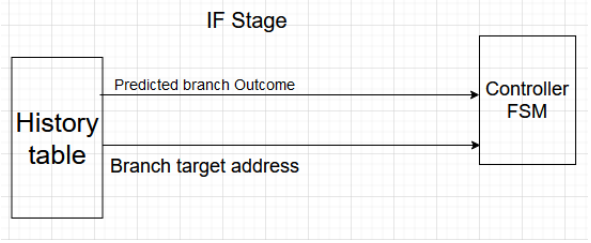


Figure 8.3: Transaction between the IF stage and the FSM

Figure 8.3 shows what values can be sent to the Controller FSM. The branch handling and verification then could be done identically as for the Backwards predictor. This means that the pc_mux input is chosen, and branch verification is also implemented. The branch verification can also be identical to how it is done for the Backwards predictor. Entries could be added/replaced from the table by using data from the WB stage. The WB stage data about a branch instruction would contain data like the branch target, the identifier and the outcome of the branch. This could be supplied to the FSM and the FSM could then make the updates to the history table. Doing this would necessitate a lot more resources than those used by the Backwards Predictor, but the high prediction accuracy for a History predictor, presented in the Project Report, would make for a good tradeoff.

The Hybrid predictor could be implemented using the same setup as above, except that if a branch is not found, the Backwards Predictor could be used to make a branch prediction. The predictors could not be implemented for the scope of this thesis, due to a lack of time, however, their implementation would make for very interesting further work.

Appendix A

Appendix 1 chapter title

In this appendix chapter, the detailed source code for my modeling is provided. The code for the hardware implementation of the backwards predictor spans over multiple files in a very large code base, thus has not been included here. It is however, openly available to the public at this address: https://github.com/achyutray/cv32e40x/tree/master/rtl

A.1 Corrected Python code for modeling of the Prediction algorithms

C-code A.1: main.py

```
# Imports
import os
import time
import random
start_time = time.time()

i = 0
count = 0
state = 0
lines = []
failure = 0
success = 0
cycles_saved = 0
prediction = 0
instruction = ""
branch_target = 0
taken_counter = 0
program_counter = 0
predictor_penalty = 0
not_taken_counter = 0
next_program_counter = 0

history_record = [[0, 0, 0]]
history_table = history_record * 128


branch_instructions = ["beq", "bne", "bge", "blt", "ble", "bgt"]

# , "bgeu", "bltu", "bgtu", "bleu"

location = r'C:\users\achyu\Desktop'


def read_log():

```

```
36      for filename in os.listdir(location):
37          global lines
38          if filename == 'uvm_no_predict_test_top.env.rvfi_agent.trn.log':
39              f = open(os.path.join(location,
        'uvm_no_predict_test_top.env.rvfi_agent.trn.log'), "r")
40              lines = f.readlines()
41              f.close()
42
43
44  def branch_state_checker(a_list):
45      global count, taken_counter, not_taken_counter, branch_target,
        next_program_counter, program_counter, instruction, state, success, failure,
        cycles_saved
46      for line in a_list:
47          count += 1
48          if count ≥ 4:
49              columns = line.split('|')
50              instruction = columns[-1]
51              branch_instruction_string = instruction.split('-')
52              program_counter = int(columns[4], 16)
53              branch_instruction_container = branch_instruction_string[1].split()
54              branch_instruction = branch_instruction_container[0]
55              for ins in branch_instructions:
56                  if branch_instruction.find(ins) != -1:
57                      branch_target_container =
        branch_instruction_container[1].split(',')
58                      # backwards_predictor(program_counter, branch_target)
59                      if len(branch_target_container) < 3:
60                          branch_target = branch_target_container[1]
61                          branch_target = int(branch_target, 16)
62                      else:
63                          branch_target = branch_target_container[2]
64                          branch_target = int(branch_target, 16)
65                      next_program_counter_container = a_list[count].split('|')
66                      next_program_counter = next_program_counter_container[4]
67                      # print(branch_target_container, instruction,
        next_program_counter)
68                      if int(next_program_counter, 16) == branch_target:
69                          taken_counter += 1
70                          state = 1
71                      else:
72                          not_taken_counter += 1
73                          state = 0
74                      #hybrid_predictor(program_counter, branch_target, state, 8)
75                      # none_predictor()
76                      #history_predictor(program_counter, branch_target, state, 8)
77                      backwards_predictor(program_counter, branch_target)
78
79      ###
80      # print("success:" + str(success))
81      # print("failure:" + str(failure))
82      # print("penalty for missed cycles: " + str(failure * 2))
83      ###
84      print("Really taken: \t\t\t\t\t" + str(taken_counter))
85      print("Really not taken: \t\t\t\t" + str(not_taken_counter))
86      print("Cycles wasted by predicting all not taken: \t" + str(taken_counter * 2))
87      print("Cycles Saved: \t\t\t\t\t" + str(cycles_saved))
88      print( "Total branches: \t\t\t\t" +str(taken_counter + not_taken_counter))
89      # print("Success(%): " + str((success/(success+failure))*100))
90      # print("Percentage of cycles saved:" + str((failure * 2) -(taken_counter *
        2)))
91
92
93  def backwards_predictor(pc, bt):
94      global success, failure, predictor_penalty, prediction, cycles_saved
95      if bt < pc:
96          prediction = 1
97      else:
98          prediction = 0
99      if ((prediction == 0) and (state == 0)):
```

```
100         cycles_saved += 0
101     elif ((prediction == 0) and (state == 1)):
102         cycles_saved -= 2
103     elif ((prediction == 1) and (state == 0)):
104         cycles_saved += -2
105     elif ((prediction == 1) and (state == 1)):
106         cycles_saved += 2
107 
108 
109 def history_predictor(pc, bt, status, table_size):
110     global  program_counter, branch_target, success, failure, cycles_saved
111 
112     for index in range(table_size):
113         if history_table[index][1] == program_counter:
114             prediction = history_table[index][0]
115             if prediction == status:
116                 if ((prediction == 0) and (status == 0)):
117                     cycles_saved += 0
118                 else:
119                     cycles_saved += 2
120                 history_table[index][0] = status
121                 return
122             else:
123                 if ((prediction == 0) and (status == 1)):
124                     cycles_saved -= 2
125                 else:
126                     cycles_saved += 0
127                 history_table[index][0] = status
128                 return
129     for index in range(table_size):
130         if history_table[index] == [0,0,0]:
131             history_table[index] = [status, pc, bt]
132             return
133     random_replacement = random.randint(0, table_size - 1)
134     history_table[random_replacement][0] = status
135     history_table[random_replacement][1] = pc
136     history_table[random_replacement][2] = bt
137     failure += 1
138     return
139 
140 
141 def hybrid_predictor(pc, bt, status, table_size):
142     global  program_counter, branch_target, success, failure, cycles_saved
143 
144     for index in range(table_size):
145         if history_table[index][1] == program_counter:
146             prediction = history_table[index][0]
147             if prediction == status:
148                 if ((prediction == 0) and (status == 0)):
149                     cycles_saved += 0
150                 else:
151                     cycles_saved += 2
152                 history_table[index][0] = status
153                 return
154             else:
155                 if ((prediction == 0) and (status == 1)):
156                     cycles_saved -= 2
157                 else:
158                     cycles_saved += 0
159                 history_table[index][0] = status
160                 return
161     for index in range(table_size):
162         if history_table[index] == [0,0,0]:
163             history_table[index] = [status, pc, bt]
164             backwards_predictor(pc, bt)
165             return
166     backwards_predictor(pc,bt)
167     random_replacement = random.randint(0, table_size - 1)
168     history_table[random_replacement][0] = status
169     history_table[random_replacement][1] = pc
```

```
170     history_table[random_replacement][2] = bt
171     failure += 1
172     return
173 
174 
175 def none_predictor():
176     global success, failure, predictor_penalty, prediction
177     prediction = 0
178     if prediction == state:
179         success += 1
180     else:
181         failure += 1
182 
183 
184 read_log()
185 branch_state_checker(lines)
186 
187 
188 ###
189 # Not Taken, and predicted not taken no saving
190 # Not taken and predicted taken +2 cycles (wasted)
191 # Taken and predicted taken -2 cycles (saved)
192 # Taken and predicted not taken, 0
193 # Plot the effects on cycles saved###
```

C-code A.1: main.py

Bibliography

[ADAH16] G. V. Merrett A. Das and B. M. Al-Hashimi. 2016 design, automation test in europe conference exhibition. In *The slowdown or race-to-idle question: Workload-aware energy optimization of SMT multicore platforms under process variation*, pages 535, 536, 537, 538, Washington, DC, USA, 2016. IEEE Computer Society.

[Braa] Branch example. https://azeria-labs.com/arm-conditional-execution-and-branching-part-6/ on 14 November 2021.

[brab] Branch types. https://course.ece.cmu.edu/ ece447/s13/lib/exe/fetch.php?media=onur-447-spring13-lecture11-branch-prediction-afterlecture.pdf on 14 November 2021.

[Brac] If statements in javascript. https://tutorialcodeplay.com/javascript/javascript-if-statement.html on 14 November 2021.

[CHC06] A. Chao-Hung Cheng. *Application-specific architecture framework for high-performance low-power embedded computing*. PhD thesis, Ann Arbor, MI, USA, 2006. Adviser-Tyson, G. S. and Adviser-Mudge, T. N.

[CV3] Cv32e40x. https://docs.openhwgroup.org/projects/cv32e40x-user-manual/intro.html on 14 November 2021.

[DCDM97] K. Danckaert, K. Catthoor, and H. De Man. System level memory optimization for hardware-software co-design. In *CODES '97: Proceedings of the 5th International Workshop on Hardware/Software Co-Design*, page 55, Washington, DC, USA, 1997. IEEE Computer Society.

[dyn] Dynamic loss due to load capacitance switching. https://www.ti.com/lit/an/slvaed3/slvaed3.pdf?ts=1639624400588ref_url=https%253A%252F%252Fwww.google.com%252F on 14 November 2021.

[ESR] Understanding esr in capacitors. https://www.avnet.com/wps/portal/abacus/resources/article/understanding-esr-in-electrolytic-capacitors/ on 17 December 2021.

[fin] Python string find(). https://www.w3schools.com/python/ref_string_find.asp on 16 December 2021.

[GIT] Git openhwgroup. https://github.com/openhwgroup/cv32e40x.

[ISA] Arm glossary. https://www.arm.com/glossary/isa on 14 November 2021.

[JLH11] David A Patterson John L Hennesy. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, 2011.

[KKA97] L. Keselbrener, M. Keselbrener, and S. Akselrod. Nonlinear high pass filter for r-wave detection in ecg signal. *Medical engineering & physics*, 19(5):481–484, 1 June 1997.

[lis] Lists in python. https://towardsdatascience.com/15-things-you-should-know-about-lists-in-python-c566792eca98 on 16 December 2021.

[Mor] John Morris. Moore's law may be dying, but there's still plenty of demand for faster chips. https://www.zdnet.com/article/moores-law-may-be-dying-but-theres-still-plenty-of-demand-for-faster-chips on 14 November 2021.

[pen] Mis-prediction penalty. https://docs.openhwgroup.org/projects/cv32e40x-user-manual/pipeline.html?highlight=branch predictionmulti-and-single-cycle-instructions on 14 November 2021.

[pro] Timing in digital circuits. https://nptel.ac.in/content/storage2/courses/117106114/Week 5 Slides/5.2Delays.pdf on 16 December 2021.

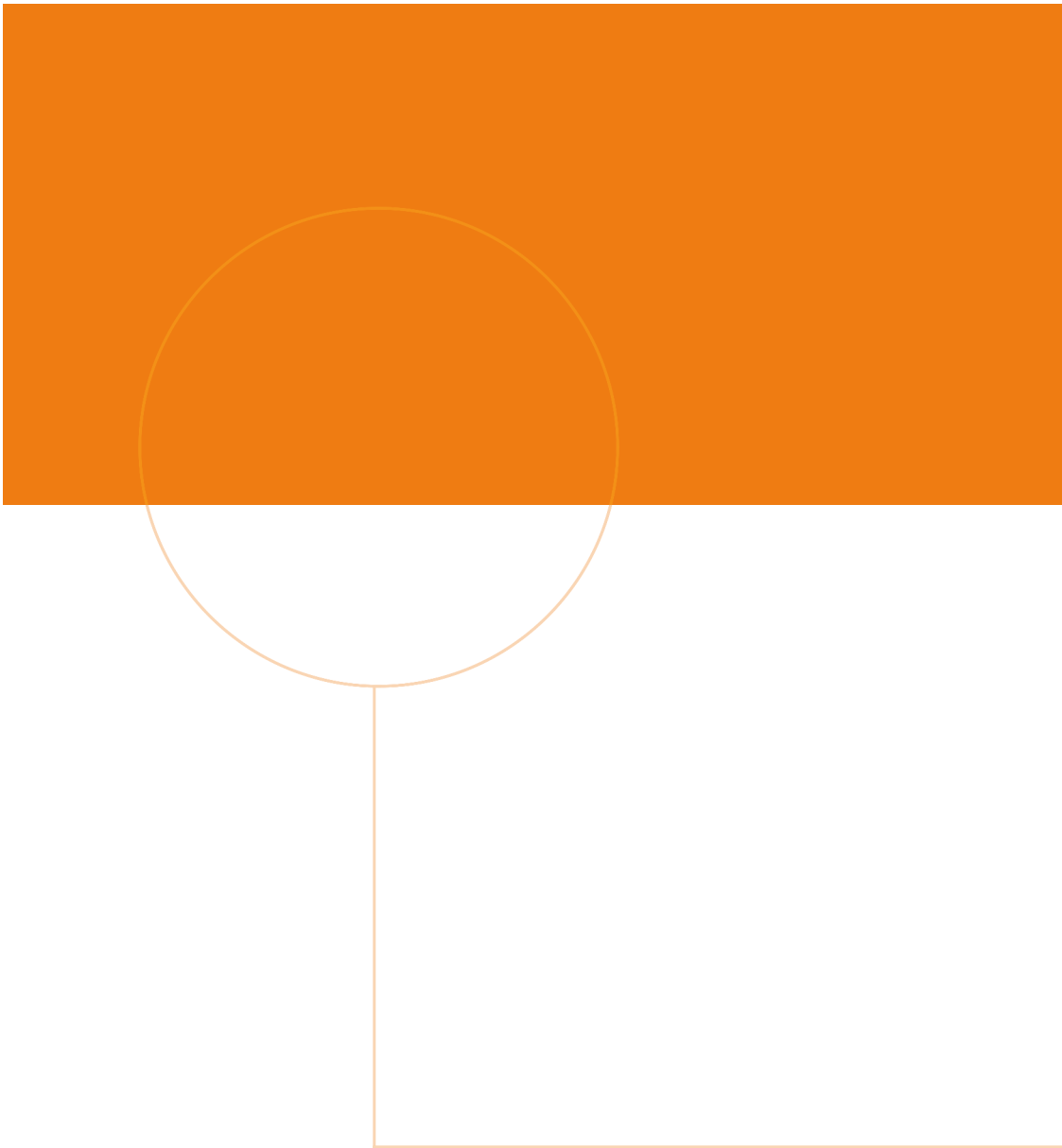
[Ray21] Achyut Ray. Branch prediction for a risc-v processor core. Project report in TTM4502, Department of Electronics and Telecommunications, NTNU – Norwegian University of Science and Technology, Dec. 2021.

[RIS] Risc-v vs arm. https://www.electropages.com/blog/2021/03/arm-vs-risc-v on 14 November 2021.

[Smi] James C Smith. A study of branch prediction strategies. Control Data Corporation, Arden Hills Minnesota.

[Sys] System verilog. INF4431 design with SV Zwolinski - UiO. (n.d.). Retrieved June 9, 2022, from https://www.uio.no/studier/emner/matnat/ifi/INF3430/h11/undervisningsmateriale/roarsk/INF4431_Design_with_SV_Zwolinski.pdf.

[Tex09] Texas Instruments. *MSP430 Ultra-Low-Power Microcontrollers*, 2009.



NTNU

Norwegian University of
Science and Technology